

## File 1 - conftest.py:

```

1: (0) """
2: (0)     Pytest configuration and fixtures for the Numpy test suite.
3: (0) """
4: (0)     import os
5: (0)     import tempfile
6: (0)     import hypothesis
7: (0)     import pytest
8: (0)     import numpy
9: (0)     from numpy.core._multiarray_tests import get_fpu_mode
10: (0)    _old_fpu_mode = None
11: (0)    _collect_results = {}
12: (0)    hypothesis.configuration.set_hypothesis_home_dir(
13: (4)        os.path.join(tempfile.gettempdir(), ".hypothesis")
14: (0)    )
15: (0)    hypothesis.settings.register_profile(
16: (4)        name="numpy-profile", deadline=None, print_blob=True,
17: (0)    )
18: (0)    hypothesis.settings.register_profile(
19: (4)        name="np.test() profile",
20: (4)        deadline=None, print_blob=True, database=None, derandomize=True,
21: (4)        suppress_health_check=list(hypothesis.HealthCheck),
22: (0)    )
23: (0)    _pytest_ini = os.path.join(os.path.dirname(__file__), "..", "pytest.ini")
24: (0)    hypothesis.settings.load_profile(
25: (4)        "numpy-profile" if os.path.isfile(_pytest_ini) else "np.test() profile"
26: (0)    )
27: (0)    os.environ["NUMPY_EXPERIMENTAL_DTYPE_API"] = "1"
28: (0)    def pytest_configure(config):
29: (4)        config.addinivalue_line("markers",
30: (8)            "valgrind_error: Tests that are known to error under valgrind.")
31: (4)        config.addinivalue_line("markers",
32: (8)            "leaks_references: Tests that are known to leak references.")
33: (4)        config.addinivalue_line("markers",
34: (8)            "slow: Tests that are very slow.")
35: (4)        config.addinivalue_line("markers",
36: (8)            "slow_pypy: Tests that are very slow on pypy.")
37: (0)    def pytest_addoption(parser):
38: (4)        parser.addoption("--available-memory", action="store", default=None,
39: (21)            help=("Set amount of memory available for running the "
40: (27)                "test suite. This can result to tests requiring "
41: (27)                "especially large amounts of memory to be skipped.
42: (0)
43: (27)                "Equivalent to setting environment variable "
44: (27)                "NPY_AVAILABLE_MEM. Default: determined"
45: (27)                "automatically."))
45: (0)    def pytest_sessionstart(session):
46: (4)        available_mem = session.config.getvalue('available_memory')
47: (4)        if available_mem is not None:
48: (8)            os.environ['NPY_AVAILABLE_MEM'] = available_mem
49: (0)    @pytest.hookimpl()
50: (0)    def pytest_itemcollected(item):
51: (4) """
52: (4)     Check FPU precision mode was not changed during test collection.
53: (4)     The clumsy way we do it here is mainly necessary because numpy
54: (4)     still uses yield tests, which can execute code at test collection
55: (4)     time.
56: (4) """
57: (4)     global _old_fpu_mode
58: (4)     mode = get_fpu_mode()
59: (4)     if _old_fpu_mode is None:
60: (8)         _old_fpu_mode = mode
61: (4)     elif mode != _old_fpu_mode:
62: (8)         _collect_results[item] = (_old_fpu_mode, mode)
63: (8)         _old_fpu_mode = mode
64: (0)    @pytest.fixture(scope="function", autouse=True)
65: (0)    def check_fpu_mode(request):
66: (4) """

```

```

67: (4)             Check FPU precision mode was not changed during the test.
68: (4)             """
69: (4)             old_mode = get_fpu_mode()
70: (4)             yield
71: (4)             new_mode = get_fpu_mode()
72: (4)             if old_mode != new_mode:
73: (8)                 raise AssertionError("FPU precision mode changed from {0:#x} to
{1:#x}")
74: (29)                     " during the test".format(old_mode, new_mode))
75: (4)             collect_result = _collect_results.get(request.node)
76: (4)             if collect_result is not None:
77: (8)                 old_mode, new_mode = collect_result
78: (8)                 raise AssertionError("FPU precision mode changed from {0:#x} to
{1:#x}")
79: (29)                     " when collecting the test".format(old_mode,
80: (64)                                         new_mode))
81: (0) @pytest.fixture(autouse=True)
82: (0) def add_np(doctest_namespace):
83: (4)     doctest_namespace['np'] = numpy
84: (0) @pytest.fixture(autouse=True)
85: (0) def env_setup(monkeypatch):
86: (4)     monkeypatch.setenv('PYTHONHASHSEED', '0')
87: (0) @pytest.fixture(params=[True, False])
88: (0) def weak_promotion(request):
89: (4)     """
90: (4)         Fixture to ensure "legacy" promotion state or change it to use the new
91: (4)         weak promotion (plus warning). `old_promotion` should be used as a
92: (4)         parameter in the function.
93: (4)     """
94: (4)     state = numpy._get_promotion_state()
95: (4)     if request.param:
96: (8)         numpy._set_promotion_state("weak_and_warn")
97: (4)     else:
98: (8)         numpy._set_promotion_state("legacy")
99: (4)     yield request.param
100: (4)    numpy._set_promotion_state(state)

```

-----

## File 2 - ctypeslib.py:

```

1: (0)             """
2: (0)             =====
3: (0)             ``ctypes`` Utility Functions
4: (0)             =====
5: (0)             See Also
6: (0)             -----
7: (0)             load_library : Load a C library.
8: (0)             ndpointer : Array restype/argtype with verification.
9: (0)             as_ctypes : Create a ctypes array from an ndarray.
10: (0)            as_array : Create an ndarray from a ctypes array.
11: (0)            References
12: (0)            -----
13: (0)            .. [1] "SciPy Cookbook: ctypes", https://scipy-
cookbook.readthedocs.io/items/Ctypes.html
14: (0)            Examples
15: (0)            -----
16: (0)            Load the C library:
17: (0)            >>> _lib = np.ctypeslib.load_library('libmystuff', '.')      #doctest: +SKIP
18: (0)            Our result type, an ndarray that must be of type double, be 1-dimensional
19: (0)            and is C-contiguous in memory:
20: (0)            >>> array_1d_double = np.ctypeslib.ndpointer(
21: (0)                ...
22: (0)                dtype=np.double,
23: (0)                ...
24: (0)                ndim=1, flags='CONTIGUOUS')      #doctest: +SKIP
25: (0)            Our C-function typically takes an array and updates its values
26: (0)            in-place. For example::
27: (4)                void foo_func(double* x, int length)
28: (4)                {
29: (8)                    int i;

```

```

28: (8)           for (i = 0; i < length; i++) {
29: (12)             x[i] = i*i;
30: (8)         }
31: (4)     }
32: (0)   We wrap it using:
33: (0)   >>> _lib.foo_func.restype = None          #doctest: +SKIP
34: (0)   >>> _lib.foo_func.argtypes = [array_1d_double, c_int] #doctest: +SKIP
35: (0)   Then, we're ready to call ``foo_func``:
36: (0)   >>> out = np.empty(15, dtype=np.double)
37: (0)   >>> _lib.foo_func(out, len(out))          #doctest: +SKIP
38: (0)
39: (0)   """
40: (11)      __all__ = ['load_library', 'ndpointer', 'c_intp', 'as_ctypes', 'as_array',
41: (0)              'as_ctypes_type']
42: (0)   import os
43: (4)   from numpy import (
44: (0)       integer, ndarray, dtype as _dtype, asarray, frombuffer
45: (0)   )
46: (0)   from numpy.core.multiarray import _flagdict, flagsobj
47: (0)   try:
48: (4)     import ctypes
49: (0)   except ImportError:
50: (4)     ctypes = None
51: (0)   if ctypes is None:
52: (4)     def _dummy(*args, **kwds):
53: (8)       """
54: (8)           Dummy object that raises an ImportError if ctypes is not available.
55: (8)           Raises
56: (8)           -----
57: (12)           ImportError
58: (8)               If ctypes is not available.
59: (8)       """
60: (4)       raise ImportError("ctypes is not available.")
61: (4)   load_library = _dummy
62: (4)   as_ctypes = _dummy
63: (4)   as_array = _dummy
64: (4)   from numpy import intp as c_intp
65: (0)   _ndptr_base = object
66: (0) else:
67: (4)   import numpy.core._internal as nic
68: (4)   c_intp = nic._getintp_ctype()
69: (4)   del nic
70: (4)   _ndptr_base = ctypes.c_void_p
71: (8)   def load_library(libname, loader_path):
72: (8)     """
73: (8)         It is possible to load a library using
74: (8)         >>> lib = ctypes.cdll[<full_path_name>] # doctest: +SKIP
75: (8)         But there are cross-platform considerations, such as library file
76: (8)         plus the fact Windows will just load the first library it finds with
77: (8)         NumPy supplies the load_library function as a convenience.
78: (12)         .. versionchanged:: 1.20.0
79: (12)             Allow libname and loader_path to take any
80: (8)                 :term:`python:path-like object` .
81: (8)   Parameters
82: (8)   -----
83: (12)   libname : path-like
84: (12)     Name of the library, which can have 'lib' as a prefix,
85: (8)       but without an extension.
86: (12)   loader_path : path-like
87: (8)     Where the library can be found.
88: (8)   Returns
89: (8)   -----
90: (11)   ctypes.cdll[libpath] : library object
91: (8)     A ctypes library object
92: (8)   Raises
93: (8)   -----
94: (12)   OSError
95: (8)     If there is no library with the expected extension, or the

```

```

95: (12)                     library is defective and cannot be loaded.
96: (8)
97: (8)
98: (8)
99: (8)
100: (8)
101: (12)
102: (12)
103: (12)
104: (12)
105: (16)
106: (12)
107: (16)
108: (12)
109: (12)
110: (12)
111: (16)
112: (8)
113: (12)
114: (8)
115: (8)
116: (12)
117: (8)
118: (12)
119: (8)
120: (12)
121: (12)
122: (16)
123: (20)
124: (16)
125: (20)
126: (8)                     raise OSError("no file with expected extension")
127: (0)                     def _num_fromflags(flaglist):
128: (4)                         num = 0
129: (4)                         for val in flaglist:
130: (8)                             num += _flagdict[val]
131: (4)                         return num
132: (0)                     _flagnames = ['C_CONTIGUOUS', 'F_CONTIGUOUS', 'ALIGNED', 'WRITEABLE',
133: (14)                         'OWNDATA', 'WRITEBACKIFCOPY']
134: (0)                     def _flags_fromnum(num):
135: (4)                         res = []
136: (4)                         for key in _flagnames:
137: (8)                             value = _flagdict[key]
138: (8)                             if (num & value):
139: (12)                                 res.append(key)
140: (4)                         return res
141: (0)                     class _ndptr(_ndptr_base):
142: (4)                         @classmethod
143: (4)                         def from_param(cls, obj):
144: (8)                             if not isinstance(obj, ndarray):
145: (12)                                 raise TypeError("argument must be an ndarray")
146: (8)                             if cls._dtype_ is not None \
147: (15)                                 and obj.dtype != cls._dtype_:
148: (12)                                 raise TypeError("array must have data type %s" % cls._dtype_)
149: (8)                             if cls._ndim_ is not None \
150: (15)                                 and obj.ndim != cls._ndim_:
151: (12)                                 raise TypeError("array must have %d dimension(s)" % cls._ndim_)
152: (8)                             if cls._shape_ is not None \
153: (15)                                 and obj.shape != cls._shape_:
154: (12)                                 raise TypeError("array must have shape %s" % str(cls._shape_))
155: (8)                             if cls._flags_ is not None \
156: (15)                                 and (obj.flags.num & cls._flags_) != cls._flags_:
157: (12)                                 raise TypeError("array must have flags %s" %
158: (20)                                     _flags_fromnum(cls._flags_))
159: (8)                         return obj.ctypes
160: (0)                     class _concrete_ndptr(_ndptr):
161: (4)                         """
162: (4)                             Like _ndptr, but with `shape` and `dtype` specified.
163: (4)                             Notably, this means the pointer has enough information to reconstruct

```

```

164: (4)          the array, which is not generally true.
165: (4)
166: (4)          def __check_retval__(self):
167: (8)            """
168: (8)              This method is called when this class is used as the .restype
169: (8)              attribute for a shared-library function, to automatically wrap the
170: (8)              pointer into an array.
171: (8)            """
172: (8)          return self.contents
173: (4)          @property
174: (4)          def contents(self):
175: (8)            """
176: (8)              Get an ndarray viewing the data pointed to by this pointer.
177: (8)              This mirrors the `contents` attribute of a normal ctypes pointer
178: (8)            """
179: (8)            full_dtype = _dtype((self._dtype_, self._shape_))
180: (8)            full_ctype = ctypes.c_char * full_dtype.itemsize
181: (8)            buffer = ctypes.cast(self, ctypes.POINTER(full_ctype)).contents
182: (8)            return frombuffer(buffer, dtype=full_dtype).squeeze(axis=0)
183: (0)          _pointer_type_cache = {}
184: (0)          def npointer(dtype=None, ndim=None, shape=None, flags=None):
185: (4)            """
186: (4)              Array-checking restype/argtypes.
187: (4)              An npointer instance is used to describe an ndarray in restypes
188: (4)              and argtypes specifications. This approach is more flexible than
189: (4)              using, for example, ``POINTER(c_double)``, since several restrictions
190: (4)              can be specified, which are verified upon calling the ctypes function.
191: (4)              These include data type, number of dimensions, shape and flags. If a
192: (4)              given array does not satisfy the specified restrictions,
193: (4)              a ``TypeError`` is raised.
194: (4)          Parameters
195: (4)          -----
196: (4)          dtype : data-type, optional
197: (8)            Array data-type.
198: (4)          ndim : int, optional
199: (8)            Number of array dimensions.
200: (4)          shape : tuple of ints, optional
201: (8)            Array shape.
202: (4)          flags : str or tuple of str
203: (8)            Array flags; may be one or more of:
204: (10)           - C_CONTIGUOUS / C / CONTIGUOUS
205: (10)           - F_CONTIGUOUS / F / FORTRAN
206: (10)           - OWNDATA / O
207: (10)           - WRITEABLE / W
208: (10)           - ALIGNED / A
209: (10)           - WRITEBACKIFCOPY / X
210: (4)          Returns
211: (4)          -----
212: (4)          klass : npointer type object
213: (8)            A type object, which is an ``_ndptr`` instance containing
214: (8)            dtype, ndim, shape and flags information.
215: (4)          Raises
216: (4)          -----
217: (4)          TypeError
218: (8)            If a given array does not satisfy the specified restrictions.
219: (4)          Examples
220: (4)          -----
221: (4)          >>> clib.somefunc.argtypes = [np.ctypeslib.npointer(dtype=np.float64,
222: (4)                           ndim=1,
223: (4)                           ...
224: (4)                           ...
225: (4)                           ...
226: (4)                           ...
227: (4)                           ...
228: (4)                           ...
229: (8)                           if dtype is not None:
230: (4)                             dtype = _dtype(dtype)
231: (4)                           num = None
231: (4)                           if flags is not None:

```

```

232: (8)
233: (12)
234: (8)
235: (12)
236: (12)
237: (8)
238: (12)
239: (12)
240: (8)
241: (12)
242: (16)
243: (12)
244: (16)
245: (12)
246: (4)
247: (8)
248: (12)
249: (8)
250: (12)
251: (4)
252: (4)
253: (8)
254: (4)
255: (8)
256: (4)
257: (8)
258: (4)
259: (8)
260: (4)
261: (8)
262: (4)
263: (8)
264: (4)
265: (8)
266: (4)
267: (8)
268: (4)
269: (8)
270: (4)
271: (8)
272: (4)
273: (17)
274: (18)
275: (18)
276: (18)
277: (4)
278: (4)
279: (0)
280: (4)
281: (8)
282: (8)
283: (12)
284: (12)
285: (8)
286: (4)
287: (8)
288: (8)
289: (8)
290: (8)
291: (8)
292: (12)
293: (12)
294: (12)
295: (12)
296: (8)
297: (8)
298: (4)
299: (4)
300: (8)

        if isinstance(flags, str):
            flags = flags.split(',')
        elif isinstance(flags, (int, integer)):
            num = flags
            flags = _flags_fromnum(num)
        elif isinstance(flags, flagsobj):
            num = flags.num
            flags = _flags_fromnum(num)
        if num is None:
            try:
                flags = [x.strip().upper() for x in flags]
            except Exception as e:
                raise TypeError("invalid flags specification") from e
            num = _num_fromflags(flags)
        if shape is not None:
            try:
                shape = tuple(shape)
            except TypeError:
                shape = (shape,)
        cache_key = (dtype, ndim, shape, num)
        try:
            return _pointer_type_cache[cache_key]
        except KeyError:
            pass
        if dtype is None:
            name = 'any'
        elif dtype.names is not None:
            name = str(id(dtype))
        else:
            name = dtype.str
        if ndim is not None:
            name += "_%dd" % ndim
        if shape is not None:
            name += "_"+"x".join(str(x) for x in shape)
        if flags is not None:
            name += "_"+"_".join(flags)
        if dtype is not None and shape is not None:
            base = _concrete_ndptr
        else:
            base = _ndptr
        klass = type("ndpointer_%s"%name, (base,), {
                    "_dtype_": dtype,
                    "_shape_": shape,
                    "_ndim_": ndim,
                    "_flags_": num})
        _pointer_type_cache[cache_key] = klass
        return klass
    if ctypes is not None:
        def _ctype_ndarray(element_type, shape):
            """ Create an ndarray of the given element type and shape """
            for dim in shape[::-1]:
                element_type = dim * element_type
                element_type.__module__ = None
            return element_type
        def _get_scalar_type_map():
            """
            Return a dictionary mapping native endian scalar dtype to ctypes types
            """
            ct = ctypes
            simple_types = [
                ct.c_byte, ct.c_short, ct.c_int, ct.c_long, ct.c_longlong,
                ct.c_ubyte, ct.c_ushort, ct.c_uint, ct.c_ulong, ct.c_ulonglong,
                ct.c_float, ct.c_double,
                ct.c_bool,
            ]
            return {_dtype(ctype): ctype for ctype in simple_types}
        _scalar_type_map = _get_scalar_type_map()
        def _ctype_from_dtype_scalar(dtype):
            dtype_with_endian = dtype.newbyteorder('S').newbyteorder('S')

```

```

301: (8)          dtype_native = dtype.newbyteorder('=')
302: (8)          try:
303: (12)            ctype = _scalar_type_map[dtype_native]
304: (8)          except KeyError as e:
305: (12)            raise NotImplementedError(
306: (16)              "Converting {!r} to a ctypes type".format(dtype))
307: (12)            ) from None
308: (8)          if dtype_with_endian.byteorder == '>':
309: (12)            ctype = ctype.__ctype_be__
310: (8)          elif dtype_with_endian.byteorder == '<':
311: (12)            ctype = ctype.__ctype_le__
312: (8)          return ctype
313: (4)          def _ctype_from_dtype_subarray(dtype):
314: (8)            element_dtype, shape = dtype.subdtype
315: (8)            ctype = _ctype_from_dtype(element_dtype)
316: (8)            return _ctype_ndarray(ctype, shape)
317: (4)          def _ctype_from_dtype_structured(dtype):
318: (8)            field_data = []
319: (8)            for name in dtype.names:
320: (12)              field_dtype, offset = dtype.fields[name][:2]
321: (12)              field_data.append((offset, name, _ctype_from_dtype(field_dtype)))
322: (8)            field_data = sorted(field_data, key=lambda f: f[0])
323: (8)            if len(field_data) > 1 and all(offset == 0 for offset, name, ctype in
field_data):
324: (12)              size = 0
325: (12)              _fields_ = []
326: (12)              for offset, name, ctype in field_data:
327: (16)                _fields_.append((name, ctype))
328: (16)                size = max(size, ctypes.sizeof(ctype))
329: (12)              if dtype.itemsize != size:
330: (16)                _fields_.append(('', ctypes.c_char * dtype.itemsize))
331: (12)              return type('union', (ctypes.Union,), dict(
332: (16)                _fields_=_fields_,
333: (16)                _pack_=1,
334: (16)                __module__=None,
335: (12)              ))
336: (8)            else:
337: (12)              last_offset = 0
338: (12)              _fields_ = []
339: (12)              for offset, name, ctype in field_data:
340: (16)                padding = offset - last_offset
341: (16)                if padding < 0:
342: (20)                  raise NotImplementedError("Overlapping fields")
343: (16)                if padding > 0:
344: (20)                  _fields_.append(('', ctypes.c_char * padding))
345: (16)                  _fields_.append((name, ctype))
346: (16)                  last_offset = offset + ctypes.sizeof(ctype)
347: (12)                  padding = dtype.itemsize - last_offset
348: (12)                  if padding > 0:
349: (16)                    _fields_.append(('', ctypes.c_char * padding))
350: (12)                  return type('struct', (ctypes.Structure,), dict(
351: (16)                    _fields_=_fields_,
352: (16)                    _pack_=1,
353: (16)                    __module__=None,
354: (12)                  ))
355: (4)          def _ctype_from_dtype(dtype):
356: (8)            if dtype.fields is not None:
357: (12)              return _ctype_from_dtype_structured(dtype)
358: (8)            elif dtype.subdtype is not None:
359: (12)              return _ctype_from_dtype_subarray(dtype)
360: (8)            else:
361: (12)              return _ctype_from_dtype_scalar(dtype)
362: (4)          def as_ctypes_type(dtype):
363: (8)            r"""
364: (8)              Convert a dtype into a ctypes type.
365: (8)              Parameters
366: (8)              -----
367: (8)              dtype : dtype
368: (12)                The dtype to convert

```

```

369: (8)             Returns
370: (8)             -----
371: (8)             ctype
372: (12)            A ctype scalar, union, array, or struct
373: (8)             Raises
374: (8)             -----
375: (8)             NotImplemented
376: (12)            If the conversion is not possible
377: (8)             Notes
378: (8)             -----
379: (8)             This function does not losslessly round-trip in either direction.
380: (8)             ``np.dtype(as_ctypes_type(dt))`` will:
381: (9)             - insert padding fields
382: (9)             - reorder fields to be sorted by offset
383: (9)             - discard field titles
384: (8)             ``as_ctypes_type(np.dtype(ctype))`` will:
385: (9)             - discard the class names of `ctypes.Structure`\ s and
386: (11)            `ctypes.Union`\ s
387: (9)             - convert single-element `ctypes.Union`\ s into single-element
388: (11)            `ctypes.Structure`\ s
389: (9)             - insert padding fields
390: (8)             """
391: (8)             return _ctype_from_dtype(_dtype(dtype))
392: (4)             def as_array(obj, shape=None):
393: (8)             """
394: (8)                 Create a numpy array from a ctypes array or POINTER.
395: (8)                 The numpy array shares the memory with the ctypes object.
396: (8)                 The shape parameter must be given if converting from a ctypes POINTER.
397: (8)                 The shape parameter is ignored if converting from a ctypes array
398: (8)                 """
399: (8)                 if isinstance(obj, ctypes._Pointer):
400: (12)                     if shape is None:
401: (16)                         raise TypeError(
402: (20)                             'as_array() requires a shape argument when called on a '
403: (20)                             'pointer')
404: (12)                     p_arr_type = ctypes.POINTER(_ctype_ndarray(obj._type_, shape))
405: (12)                     obj = ctypes.cast(obj, p_arr_type).contents
406: (8)                     return asarray(obj)
407: (4)             def as_ctypes(obj):
408: (8)                 """Create and return a ctypes object from a numpy array. Actually
409: (8)                 anything that exposes the __array_interface__ is accepted."""
410: (8)                 ai = obj.__array_interface__
411: (8)                 if ai["strides"]:
412: (12)                     raise TypeError("strided arrays not supported")
413: (8)                 if ai["version"] != 3:
414: (12)                     raise TypeError("only __array_interface__ version 3 supported")
415: (8)                 addr, readonly = ai["data"]
416: (8)                 if readonly:
417: (12)                     raise TypeError("readonly arrays unsupported")
418: (8)                 ctype_scalar = as_ctypes_type(ai["typestr"])
419: (8)                 result_type = _ctype_ndarray(ctype_scalar, ai["shape"])
420: (8)                 result = result_type.from_address(addr)
421: (8)                 result.__keep__ = obj
422: (8)                 return result

```

-----

## File 3 - dtypes.py:

```

1: (0)             """
2: (0)             DType classes and utility (:mod:`numpy.dtypes`)
3: (0)             =====
4: (0)             This module is home to specific dtypes related functionality and their
classes.
5: (0)             For more general information about dtypes, also see `numpy.dtype` and
6: (0)             :ref:`arrays.dtypes`.
7: (0)             Similar to the builtin ``types`` module, this submodule defines types
(classes)
8: (0)             that are not widely used directly.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

9: (0) .. versionadded:: NumPy 1.25
10: (4)     The dtypes module is new in NumPy 1.25. Previously Dtype classes were
11: (4)     only accessible indirectly.
12: (0) Dtype classes
13: (0) -----
14: (0)     The following are the classes of the corresponding NumPy dtype instances and
15: (0)     NumPy scalar types. The classes can be used in ``isinstance`` checks and can
16: (0)     also be instantiated or used directly. Direct use of these classes is not
17: (0)     typical, since their scalar counterparts (e.g. ``np.float64``) or strings
18: (0)     like ``"float64"`` can be used.
19: (0) .. list-table::
20: (4)     :header-rows: 1
21: (4)     * - Group
22: (6)         - Dtype class
23: (4)     * - Boolean
24: (6)         - ``BoolDType``
25: (4)     * - Bit-sized integers
26: (6)         - ``Int8DType``, ``UInt8DType``, ``Int16DType``, ``UInt16DType``,
27: (8)         - ``Int32DType``, ``UInt32DType``, ``Int64DType``, ``UInt64DType``
28: (4)     * - C-named integers (may be aliases)
29: (6)         - ``ByteDType``, ``UByteDType``, ``ShortDType``, ``UShortDType``,
30: (8)         - ``IntDType``, ``UIntDType``, ``LongDType``, ``ULongDType``,
31: (8)         - ``LongLongDType``, ``ULongLongDType``
32: (4)     * - Floating point
33: (6)         - ``Float16DType``, ``Float32DType``, ``Float64DType``,
34: (8)         - ``LongDoubleDType``
35: (4)     * - Complex
36: (6)         - ``Complex64DType``, ``Complex128DType``, ``CLongDoubleDType``
37: (4)     * - Strings
38: (6)         - ``BytesDType``, ``BytesDType``
39: (4)     * - Times
40: (6)         - ``DateTime64DType``, ``TimeDelta64DType``
41: (4)     * - Others
42: (6)         - ``ObjectDType``, ``VoidDType``
43: (0) """
44: (0)     __all__ = []
45: (0) def __add_dtype_helper(Dtype, alias):
46: (4)     from numpy import dtypes
47: (4)     setattr(dtypes, Dtype.__name__, Dtype)
48: (4)     __all__.append(Dtype.__name__)
49: (4)     if alias:
50: (8)         alias = alias.removeprefix("numpy.dtypes.")
51: (8)         setattr(dtypes, alias, Dtype)
52: (8)         __all__.append(alias)

```

---

#### File 4 - exceptions.py:

```

1: (0) """
2: (0) Exceptions and Warnings (:mod:`numpy.exceptions`)
3: (0) -----
4: (0) General exceptions used by NumPy. Note that some exceptions may be module
5: (0) specific, such as linear algebra errors.
6: (0) .. versionadded:: NumPy 1.25
7: (4)     The exceptions module is new in NumPy 1.25. Older exceptions remain
8: (4)     available through the main NumPy namespace for compatibility.
9: (0) .. currentmodule:: numpy.exceptions
10: (0) Warnings
11: (0) -----
12: (0) .. autosummary::
13: (3)     :toctree: generated/
14: (3)     ComplexWarning           Given when converting complex to real.
15: (3)     VisibleDeprecationWarning Same as a DeprecationWarning, but more visible.
16: (0) Exceptions
17: (0) -----
18: (0) .. autosummary::
19: (3)     :toctree: generated/
20: (4)     AxisError                Given when an axis was invalid.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

21: (4)          DtypePromotionError  Given when no common dtype could be found.
22: (4)          TooHardError      Error specific to `numpy.shares_memory`.
23: (0)
24: (0)
25: (4)          __all__ = [
26: (4)            "ComplexWarning", "VisibleDeprecationWarning", "ModuleDeprecationWarning",
27: (4)            "TooHardError", "AxisError", "DTypePromotionError"]
28: (0)          if '_is_loaded' in globals():
29: (4)            raise RuntimeError('Reloading numpy._globals is not allowed')
30: (0)          _is_loaded = True
31: (0)          class ComplexWarning(RuntimeWarning):
32: (4)            """
33: (4)              The warning raised when casting a complex dtype to a real dtype.
34: (4)              As implemented, casting a complex number to a real discards its imaginary
35: (4)              part, but this behavior may not be what the user actually wants.
36: (4)            """
37: (0)            pass
38: (0)          class ModuleDeprecationWarning(DeprecationWarning):
39: (4)            """Module deprecation warning.
40: (4)            .. warning::
41: (8)              This warning should not be used, since nose testing is not relevant
42: (8)              anymore.
43: (4)              The nose tester turns ordinary Deprecation warnings into test failures.
44: (4)              That makes it hard to deprecate whole modules, because they get
45: (4)              imported by default. So this is a special Deprecation warning that the
46: (4)              nose tester will let pass without making tests fail.
47: (4)            """
48: (0)          class VisibleDeprecationWarning(UserWarning):
49: (4)            """Visible deprecation warning.
50: (4)            By default, python will not show deprecation warnings, so this class
51: (4)            can be used when a very visible warning is helpful, for example because
52: (4)            the usage is most likely a user bug.
53: (4)            """
54: (0)          class TooHardError(RuntimeError):
55: (4)            """max_work was exceeded.
56: (4)            This is raised whenever the maximum number of candidate solutions
57: (4)            to consider specified by the ``max_work`` parameter is exceeded.
58: (4)            Assigning a finite number to max_work may have caused the operation
59: (4)            to fail.
56: (4)            """
57: (0)            pass
58: (0)          class AxisError(ValueError, IndexError):
59: (4)            """Axis supplied was invalid.
60: (4)            This is raised whenever an ``axis`` parameter is specified that is larger
61: (4)            than the number of array dimensions.
62: (4)            For compatibility with code written against older numpy versions, which
63: (4)            raised a mixture of `ValueError` and `IndexError` for this situation, this
64: (4)            exception subclasses both to ensure that ``except ValueError`` and
65: (4)            ``except IndexError`` statements continue to catch `AxisError`.
66: (4)            .. versionadded:: 1.13
67: (4)            Parameters
68: (4)            -----
69: (4)            axis : int or str
70: (8)              The out of bounds axis or a custom exception message.
71: (8)              If an axis is provided, then `ndim` should be specified as well.
72: (4)            ndim : int, optional
73: (8)              The number of array dimensions.
74: (4)            msg_prefix : str, optional
75: (8)              A prefix for the exception message.
76: (4)            Attributes
77: (4)            -----
78: (4)            axis : int, optional
79: (8)              The out of bounds axis or ``None`` if a custom exception
80: (8)              message was provided. This should be the axis as passed by
81: (8)              the user, before any normalization to resolve negative indices.
82: (4)              .. versionadded:: 1.22
83: (4)            ndim : int, optional
84: (8)              The number of array dimensions or ``None`` if a custom exception
85: (8)              message was provided.
86: (4)              .. versionadded:: 1.22

```

```

90: (4) Examples
91: (4) -----
92: (4) >>> array_1d = np.arange(10)
93: (4) >>> np.cumsum(array_1d, axis=1)
94: (4) Traceback (most recent call last):
95: (6) ...
96: (4) numpy.exceptions.AxisError: axis 1 is out of bounds for array of dimension
1
97: (4) Negative axes are preserved:
98: (4) >>> np.cumsum(array_1d, axis=-2)
99: (4) Traceback (most recent call last):
100: (6) ...
101: (4) numpy.exceptions.AxisError: axis -2 is out of bounds for array of
dimension 1
102: (4) The class constructor generally takes the axis and arrays'
103: (4) dimensionality as arguments:
104: (4) >>> print(np.AxisError(2, 1, msg_prefix='error'))
105: (4) error: axis 2 is out of bounds for array of dimension 1
106: (4) Alternatively, a custom exception message can be passed:
107: (4) >>> print(np.AxisError('Custom error message'))
108: (4) Custom error message
109: (4) """
110: (4) __slots__ = ("axis", "ndim", "_msg")
111: (4) def __init__(self, axis, ndim=None, msg_prefix=None):
112: (8)     if ndim is msg_prefix is None:
113: (12)         self._msg = axis
114: (12)         self.axis = None
115: (12)         self.ndim = None
116: (8)     else:
117: (12)         self._msg = msg_prefix
118: (12)         self.axis = axis
119: (12)         self.ndim = ndim
120: (4) def __str__(self):
121: (8)     axis = self.axis
122: (8)     ndim = self.ndim
123: (8)     if axis is ndim is None:
124: (12)         return self._msg
125: (8)     else:
126: (12)         msg = f"axis {axis} is out of bounds for array of dimension
{ndim}"
127: (12)         if self._msg is not None:
128: (16)             msg = f"{self._msg}: {msg}"
129: (12)         return msg
130: (0) class DtypePromotionError(TypeError):
131: (4)     """Multiple DTypes could not be converted to a common one.
132: (4)     This exception derives from ``TypeError`` and is raised whenever dtypes
133: (4)     cannot be converted to a single common one. This can be because they
134: (4)     are of a different category/class or incompatible instances of the same
135: (4)     one (see Examples).
136: (4) Notes
137: (4) -----
138: (4) Many functions will use promotion to find the correct result and
139: (4) implementation. For these functions the error will typically be chained
140: (4) with a more specific error indicating that no implementation was found
141: (4) for the input dtypes.
142: (4) Typically promotion should be considered "invalid" between the dtypes of
143: (4) two arrays when `arr1 == arr2` can safely return all ``False`` because the
144: (4) dtypes are fundamentally different.
145: (4) Examples
146: (4) -----
147: (4) Datetimes and complex numbers are incompatible classes and cannot be
148: (4) promoted:
149: (4) >>> np.result_type(np.dtype("M8[s]"), np.complex128)
150: (4) DtypePromotionError: The Dtype <class 'numpy.dtype[datetime64]'> could not
151: (4) be promoted by <class 'numpy.dtype[complex128]'>. This means that no
common
152: (4) Dtype exists for the given inputs. For example they cannot be stored in a
153: (4) single array unless the dtype is `object`. The full list of DTypes is:
154: (4) (<class 'numpy.dtype[datetime64]'>, <class 'numpy.dtype[complex128]'>)

```

```

155: (4)          For example for structured dtypes, the structure can mismatch and the
156: (4)          same ``DTypePromotionError`` is given when two structured dtypes with
157: (4)          a mismatch in their number of fields is given:
158: (4)          >>> dtype1 = np.dtype([("field1", np.float64), ("field2", np.int64)])
159: (4)          >>> dtype2 = np.dtype([("field1", np.float64)])
160: (4)          >>> np.promote_types(dtype1, dtype2)
161: (4)          DTypePromotionError: field names `('field1', 'field2')` and `('field1',)`
162: (4)          mismatch.
163: (4)          """
164: (4)          pass
-----
```

## File 5 - matlib.py:

```

1: (0)          import warnings
2: (0)          warnings.warn("Importing from numpy.matlib is deprecated since 1.19.0. "
3: (14)          "The matrix subclass is not the recommended way to represent "
4: (14)          "matrices or deal with linear algebra (see "
5: (14)          "https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-
users.html). "
6: (14)          "Please adjust your code to use regular ndarray. ",
7: (14)          PendingDeprecationWarning, stacklevel=2)
8: (0)          import numpy as np
9: (0)          from numpy.matrixlib.defmatrix import matrix, asmatrix
10: (0)          from numpy import * # noqa: F403
11: (0)          __version__ = np.__version__
12: (0)          __all__ = np.__all__[ :] # copy numpy namespace
13: (0)          __all__ += ['rand', 'randn', 'rppmat']
14: (0)          def empty(shape, dtype=None, order='C'):
15: (4)          """Return a new matrix of given shape and type, without initializing
entries.
16: (4)          Parameters
17: (4)          -----
18: (4)          shape : int or tuple of int
19: (8)          Shape of the empty matrix.
20: (4)          dtype : data-type, optional
21: (8)          Desired output data-type.
22: (4)          order : {'C', 'F'}, optional
23: (8)          Whether to store multi-dimensional data in row-major
24: (8)          (C-style) or column-major (Fortran-style) order in
25: (8)          memory.
26: (4)          See Also
27: (4)          -----
28: (4)          empty_like, zeros
29: (4)          Notes
30: (4)          -----
31: (4)          `empty`, unlike `zeros`, does not set the matrix values to zero,
32: (4)          and may therefore be marginally faster. On the other hand, it requires
33: (4)          the user to manually set all the values in the array, and should be
34: (4)          used with caution.
35: (4)          Examples
36: (4)          -----
37: (4)          >>> import numpy.matlib
38: (4)          >>> np.matlib.empty((2, 2))    # filled with random data
39: (4)          matrix([[ 6.76425276e-320,   9.79033856e-307], # random
40: (12)          [ 7.39337286e-309,   3.22135945e-309]])
41: (4)          >>> np.matlib.empty((2, 2), dtype=int)
42: (4)          matrix([[ 6600475,           0], # random
43: (12)          [ 6586976, 22740995]])
44: (4)          """
45: (4)          return ndarray.__new__(matrix, shape, dtype, order=order)
46: (0)          def ones(shape, dtype=None, order='C'):
47: (4)          """
48: (4)          Matrix of ones.
49: (4)          Return a matrix of given shape and type, filled with ones.
50: (4)          Parameters
51: (4)          -----
52: (4)          shape : {sequence of ints, int}
```

```

53: (8)           Shape of the matrix
54: (4)           dtype : data-type, optional
55: (8)           The desired data-type for the matrix, default is np.float64.
56: (4)           order : {'C', 'F'}, optional
57: (8)           Whether to store matrix in C- or Fortran-contiguous order,
58: (8)           default is 'C'.
59: (4)           Returns
60: (4)           -----
61: (4)           out : matrix
62: (8)           Matrix of ones of given shape, dtype, and order.
63: (4)           See Also
64: (4)           -----
65: (4)           ones : Array of ones.
66: (4)           matlib.zeros : Zero matrix.
67: (4)           Notes
68: (4)           -----
69: (4)           If `shape` has length one i.e. `` $(N,)$ ```, or is a scalar `` $N$ ```,
70: (4)           `out` becomes a single row matrix of shape `` $(1,N)$ ```.
71: (4)           Examples
72: (4)           -----
73: (4)           >>> np.matlib.ones((2,3))
74: (4)           matrix([[1., 1., 1.],
75: (12)             [1., 1., 1.]])
76: (4)           >>> np.matlib.ones(2)
77: (4)           matrix([[1., 1.]])
78: (4)           """
79: (4)           a = ndarray.__new__(matrix, shape, dtype, order=order)
80: (4)           a.fill(1)
81: (4)           return a
82: (0)           def zeros(shape, dtype=None, order='C'):
83: (4)           """
84: (4)           Return a matrix of given shape and type, filled with zeros.
85: (4)           Parameters
86: (4)           -----
87: (4)           shape : int or sequence of ints
88: (8)             Shape of the matrix
89: (4)           dtype : data-type, optional
90: (8)             The desired data-type for the matrix, default is float.
91: (4)           order : {'C', 'F'}, optional
92: (8)             Whether to store the result in C- or Fortran-contiguous order,
93: (8)             default is 'C'.
94: (4)           Returns
95: (4)           -----
96: (4)           out : matrix
97: (8)           Zero matrix of given shape, dtype, and order.
98: (4)           See Also
99: (4)           -----
100: (4)          numpy.zeros : Equivalent array function.
101: (4)          matlib.ones : Return a matrix of ones.
102: (4)          Notes
103: (4)          -----
104: (4)          If `shape` has length one i.e. `` $(N,)$ ```, or is a scalar `` $N$ ```,
105: (4)          `out` becomes a single row matrix of shape `` $(1,N)$ ```.
106: (4)          Examples
107: (4)          -----
108: (4)          >>> import numpy.matlib
109: (4)          >>> np.matlib.zeros((2, 3))
110: (4)          matrix([[0., 0., 0.],
111: (12)            [0., 0., 0.]])
112: (4)          >>> np.matlib.zeros(2)
113: (4)          matrix([[0., 0.]])
114: (4)          """
115: (4)          a = ndarray.__new__(matrix, shape, dtype, order=order)
116: (4)          a.fill(0)
117: (4)          return a
118: (0)          def identity(n,dtype=None):
119: (4)          """
120: (4)          Returns the square identity matrix of given size.
121: (4)          Parameters

```

```

122: (4)
123: (4)
124: (8)
125: (4)
126: (8)
127: (4)
128: (4)
129: (4)
130: (8)
131: (8)
132: (4)
133: (4)
134: (4)
135: (4)
136: (4)
137: (4)
138: (4)
139: (4)
140: (4)
141: (12)
142: (12)
143: (4)
144: (4)
145: (4)
146: (4)
147: (4)
148: (0)
149: (4)
150: (4)
151: (4)
152: (4)
153: (4)
154: (8)
155: (4)
156: (8)
157: (4)
158: (8)
159: (8)
160: (8)
161: (4)
162: (8)
163: (4)
164: (8)
165: (8)
166: (8)
167: (4)
168: (4)
169: (4)
170: (8)
171: (8)
172: (4)
173: (4)
174: (4)
175: (4)
176: (4)
177: (4)
178: (4)
179: (4)
180: (4)
181: (12)
182: (12)
183: (4)
184: (4)
185: (0)
186: (4)
187: (4)
188: (4)
189: (4)
190: (4)

-----  

n : int  

    Size of the returned identity matrix.  

dtype : data-type, optional  

    Data-type of the output. Defaults to ``float``.  

Returns  

-----  

out : matrix  

    `n` x `n` matrix with its main diagonal set to one,  

    and all other elements zero.  

See Also  

-----  

numpy.identity : Equivalent array function.  

matlib.eye : More general matrix identity function.  

Examples  

-----  

>>> import numpy.matlib  

>>> np.matlib.identity(3, dtype=int)  

matrix([[1, 0, 0],  

       [0, 1, 0],  

       [0, 0, 1]])  

"""  

a = array([1]+n*[0], dtype=dtype)  

b = empty((n, n), dtype=dtype)  

b.flat = a  

return b  

def eye(n,M=None, k=0, dtype=float, order='C'):  

"""
    Return a matrix with ones on the diagonal and zeros elsewhere.  

Parameters  

-----  

n : int  

    Number of rows in the output.  

M : int, optional  

    Number of columns in the output, defaults to `n`.  

k : int, optional  

    Index of the diagonal: 0 refers to the main diagonal,  

    a positive value refers to an upper diagonal,  

    and a negative value to a lower diagonal.  

dtype : dtype, optional  

    Data-type of the returned matrix.  

order : {'C', 'F'}, optional  

    Whether the output should be stored in row-major (C-style) or  

    column-major (Fortran-style) order in memory.  

.. versionadded:: 1.14.0  

Returns  

-----  

I : matrix  

    A `n` x `M` matrix where all elements are equal to zero,  

    except for the `k`-th diagonal, whose values are equal to one.  

See Also  

-----  

numpy.eye : Equivalent array function.  

identity : Square identity matrix.  

Examples  

-----  

>>> import numpy.matlib  

>>> np.matlib.eye(3, k=1, dtype=float)  

matrix([[0., 1., 0.],  

       [0., 0., 1.],  

       [0., 0., 0.]])  

"""  

return asmatrix(np.eye(n, M=M, k=k, dtype=dtype, order=order))  

def rand(*args):  

"""
    Return a matrix of random values with given shape.  

Create a matrix of the given shape and propagate it with  

random samples from a uniform distribution over ``[0, 1)``.  

Parameters

```



SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

260: (4)          """
261: (4)          if isinstance(args[0], tuple):
262: (8)            args = args[0]
263: (4)          return asmatrix(np.random.randn(*args))
264: (0)      def repmat(a, m, n):
265: (4)          """
266: (4)          Repeat a 0-D to 2-D array or matrix MxN times.
267: (4)          Parameters
268: (4)          -----
269: (4)          a : array_like
270: (8)            The array or matrix to be repeated.
271: (4)          m, n : int
272: (8)            The number of times `a` is repeated along the first and second axes.
273: (4)          Returns
274: (4)          -----
275: (4)          out : ndarray
276: (8)            The result of repeating `a`.
277: (4)          Examples
278: (4)          -----
279: (4)          >>> import numpy.matlib
280: (4)          >>> a0 = np.array(1)
281: (4)          >>> np.matlib.repmat(a0, 2, 3)
282: (4)          array([[1, 1, 1],
283: (11)            [1, 1, 1]])
284: (4)          >>> a1 = np.arange(4)
285: (4)          >>> np.matlib.repmat(a1, 2, 2)
286: (4)          array([[0, 1, 2, 3, 0, 1, 2, 3],
287: (11)            [0, 1, 2, 3, 0, 1, 2, 3]])
288: (4)          >>> a2 = np.asmatrix(np.arange(6).reshape(2, 3))
289: (4)          >>> np.matlib.repmat(a2, 2, 3)
290: (4)          matrix([[0, 1, 2, 0, 1, 2, 0, 1, 2],
291: (12)            [3, 4, 5, 3, 4, 5, 3, 4, 5],
292: (12)            [0, 1, 2, 0, 1, 2, 0, 1, 2],
293: (12)            [3, 4, 5, 3, 4, 5, 3, 4, 5]])
294: (4)          """
295: (4)          a = asanyarray(a)
296: (4)          ndim = a.ndim
297: (4)          if ndim == 0:
298: (8)            origrows, origcols = (1, 1)
299: (4)          elif ndim == 1:
300: (8)            origrows, origcols = (1, a.shape[0])
301: (4)          else:
302: (8)            origrows, origcols = a.shape
303: (4)          rows = origrows * m
304: (4)          cols = origcols * n
305: (4)          c = a.reshape(1, a.size).repeat(m, 0).reshape(rows, origcols).repeat(n, 0)
306: (4)          return c.reshape(rows, cols)

```

---

**File 6 - version.py:**

```

1: (0)          version = "1.26.4"
2: (0)          __version__ = version
3: (0)          full_version = version
4: (0)          git_revision = "9815c16f449e12915ef35a8255329ba26dacd5c0"
5: (0)          release = 'dev' not in version and '+' not in version
6: (0)          short_version = version.split("+")[0]

```

---

**File 7 - \_distributor\_init.py:**

```

1: (0)          """ Distributor init file
2: (0)          Distributors: you can add custom code here to support particular distributions
3: (0)            of numpy.
4: (0)          For example, this is a good place to put any BLAS/LAPACK initialization code.
5: (0)          The numpy standard source distribution will not put code in this file, so you
6: (0)            can safely replace this file with your own version.

```

```

7: (0)
8: (0)
9: (4)
10: (0)
11: (4)
-----
```

## File 8 - \_globals.py:

```

1: (0)
2: (0)      """
3: (0)      Module defining global singleton classes.
4: (0)      This module raises a RuntimeError if an attempt to reload it is made. In that
5: (0)      way the identities of the classes defined here are fixed and will remain so
6: (0)      even if numpy itself is reloaded. In particular, a function like the following
7: (0)      will still work correctly after numpy is reloaded::
8: (4)          def foo(arg=np._NoValue):
9: (8)              if arg is np._NoValue:
10: (12)                  ...
11: (0)          That was not the case when the singleton classes were defined in the numpy
12: (0)          ``__init__.py`` file. See gh-7844 for a discussion of the reload problem that
13: (0)          motivated this module.
14: (0)      """
15: (0)      import enum
16: (0)      from .utils import set_module as _set_module
17: (0)      __all__ = ['_NoValue', '_CopyMode']
18: (0)      if '_is_loaded' in globals():
19: (4)          raise RuntimeError('Reloading numpy._globals is not allowed')
20: (0)      _is_loaded = True
21: (0)      class _NoValueType:
22: (4)          """Special keyword value.
23: (4)          The instance of this class may be used as the default value assigned to a
24: (4)          keyword if no other obvious default (e.g., `None`) is suitable,
25: (4)          Common reasons for using this keyword are:
26: (6)          - A new keyword is added to a function, and that function forwards its
27: (6)          inputs to another function or method which can be defined outside of
28: ``keepdims``
29: (6)          NumPy. For example, ``np.std(x)`` calls ``x.std``, so when a
30: (6)          keyword was added that could only be forwarded if the user explicitly
31: (6)          specified ``keepdims``; downstream array libraries may not have added
32: (6)          the same keyword, so adding ``x.std(..., keepdims=keepdims)``
33: (6)          unconditionally could have broken previously working code.
34: (4)          - A keyword is being deprecated, and a deprecation warning must only be
35: (4)          emitted when the keyword is used.
36: (4)          """
37: (8)          __instance = None
38: (12)         def __new__(cls):
39: (8)             if not cls.__instance:
40: (4)                 cls.__instance = super().__new__(cls)
41: (8)             return cls.__instance
42: (0)         def __repr__(self):
43: (8)             return "<no value>"
44: (0)         _NoValue = _NoValueType()
45: (0)         @_set_module("numpy")
46: (0)         class _CopyMode(enum.Enum):
47: (4)             """
48: (4)             An enumeration for the copy modes supported
49: (4)             by numpy.copy() and numpy.array(). The following three modes are
50: (4)
51: (17)             - ALWAYS: This means that a deep copy of the input
52: (4)                 array will always be taken.
53: (13)             - IF_NEEDED: This means that a deep copy of the input
54: (4)                 array will be taken only if necessary.
55: (13)             - NEVER: This means that the deep copy will never be taken.
56: (4)                 If a copy cannot be avoided then a `ValueError` will be
57: (4)                 raised.
58: (4)             Note that the buffer-protocol could in theory do copies. NumPy currently
59: (4)             assumes an object exporting the buffer protocol will never do this.
59: (4)             """

```

```

58: (4)             ALWAYS = True
59: (4)             IF_NEEDED = False
60: (4)             NEVER = 2
61: (4)             def __bool__(self):
62: (8)                 if self == _CopyMode.ALWAYS:
63: (12)                     return True
64: (8)                 if self == _CopyMode.IF_NEEDED:
65: (12)                     return False
66: (8)                 raise ValueError(f"{self} is neither True nor False.")
-----
```

## File 9 - \_pytesttester.py:

```

1: (0)             """
2: (0)             Pytest test running.
3: (0)             This module implements the ``test()`` function for NumPy modules. The usual
4: (0)             boiler plate for doing that is to put the following in the module
5: (0)             ``__init__.py`` file::
6: (4)                 from numpy._pytesttester import PytestTester
7: (4)                 test = PytestTester(__name__)
8: (4)                 del PytestTester
9: (0)             Warnings filtering and other runtime settings should be dealt with in the
10: (0)             ``pytest.ini`` file in the numpy repo root. The behavior of the test depends
on
11: (0)             whether or not that file is found as follows:
12: (0)             * ``pytest.ini`` is present (develop mode)
13: (4)                 All warnings except those explicitly filtered out are raised as error.
14: (0)             * ``pytest.ini`` is absent (release mode)
15: (4)                 DeprecationWarnings and PendingDeprecationWarnings are ignored, other
16: (4)                 warnings are passed through.
17: (0)             In practice, tests run from the numpy repo are run in develop mode. That
18: (0)             includes the standard ``python runtests.py`` invocation.
19: (0)             This module is imported by every numpy subpackage, so lies at the top level to
20: (0)             simplify circular import issues. For the same reason, it contains no numpy
21: (0)             imports at module scope, instead importing numpy within function calls.
22: (0)
23: (0)             import sys
24: (0)             import os
25: (0)             __all__ = ['PytestTester']
26: (0)             def _show_numpy_info():
27: (4)                 import numpy as np
28: (4)                 print("NumPy version %s" % np.__version__)
29: (4)                 relaxed_strides = np.ones((10, 1), order="C").flags.f_contiguous
30: (4)                 print("NumPy relaxed strides checking option:", relaxed_strides)
31: (4)                 info = np.lib.utils._opt_info()
32: (4)                 print("NumPy CPU features: ", (info if info else 'nothing enabled'))
33: (0)             class PytestTester:
34: (4)                 """
35: (4)                 Pytest test runner.
36: (4)                 A test function is typically added to a package's __init__.py like so::
37: (6)                     from numpy._pytesttester import PytestTester
38: (6)                     test = PytestTester(__name__).test
39: (6)                     del PytestTester
40: (4)                 Calling this test function finds and runs all tests associated with the
41: (4)                 module and all its sub-modules.
42: (4)                 Attributes
43: (4)                 -----
44: (4)                 module_name : str
45: (8)                     Full path to the package to test.
46: (4)                 Parameters
47: (4)                 -----
48: (4)                 module_name : module name
49: (8)                     The name of the module to test.
50: (4)                 Notes
51: (4)                 -----
52: (4)                 Unlike the previous ``nose``-based implementation, this class is not
53: (4)                 publicly exposed as it performs some ``numpy``-specific warning
54: (4)                 suppression.
```

```

55: (4)
56: (4)
57: (8)
58: (4)
59: (17)
60: (8)
61: (8)
62: (8)
63: (8)
64: (8)
65: (12)
66: (12)
67: (12)
68: (8)
69: (12)
70: (8)
71: (12)
72: (8)
73: (12)
74: (8)
75: (12)
76: (12)
77: (8)
78: (12)
79: (12)
80: (8)
81: (12)
82: (8)
83: (8)
84: (8)
85: (12)
86: (8)
87: (8)
88: (8)
89: (8)
90: (8)
91: (8)
92: (8)
93: (8)
94: (8)
95: (8)
96: (8)
97: (8)
98: (8)
99: (8)
100: (8)
101: (8)
102: (8)
103: (8)
104: (8)
105: (8)
106: (12)
107: (16)
108: (16)
109: (8)
110: (12)
111: (8)
112: (12)
113: (12)
114: (12)
115: (12)
116: (12)
117: (8)
118: (12)
119: (12)
120: (12)
121: (8)
122: (12)
123: (8)

        """
        def __init__(self, module_name):
            self.module_name = module_name
        def __call__(self, label='fast', verbose=1, extra_argv=None,
                    doctests=False, coverage=False, durations=-1, tests=None):
            """
            Run tests for module using pytest.

            Parameters
            -----
            label : {'fast', 'full'}, optional
                Identifies the tests to run. When set to 'fast', tests decorated
                with `pytest.mark.slow` are skipped, when 'full', the slow marker
                is ignored.
            verbose : int, optional
                Verbosity value for test outputs, in the range 1-3. Default is 1.
            extra_argv : list, optional
                List with any extra arguments to pass to pytests.
            doctests : bool, optional
                .. note:: Not supported
            coverage : bool, optional
                If True, report coverage of NumPy code. Default is False.
                Requires installation of (pip) pytest-cov.
            durations : int, optional
                If < 0, do nothing, If 0, report time of all tests, if > 0,
                report the time of the slowest `timer` tests. Default is -1.
            tests : test or list of tests
                Tests to be executed with pytest '--pyargs'
            Returns
            -----
            result : bool
                Return True on success, false otherwise.
            Notes
            -----
            Each NumPy module exposes `test` in its namespace to run all tests for
            it. For example, to run all tests for numpy.lib:
            >>> np.lib.test() #doctest: +SKIP
            Examples
            -----
            >>> result = np.lib.test() #doctest: +SKIP
            ...
            1023 passed, 2 skipped, 6 deselected, 1 xfailed in 10.39 seconds
            >>> result
            True
            """
            import pytest
            import warnings
            module = sys.modules[self.module_name]
            module_path = os.path.abspath(module.__path__[0])
            pytest_args = ["-l"]
            pytest_args += ["-q"]
            if sys.version_info < (3, 12):
                with warnings.catch_warnings():
                    warnings.simplefilter("always")
                    from numpy.distutils import cpuinfo
            with warnings.catch_warnings(record=True):
                import numpy.array_api
            pytest_args += [
                "-W ignore:Not importing directory",
                "-W ignore:numpy.dtype size changed",
                "-W ignore:numpy.ufunc size changed",
                "-W ignore::UserWarning:cpuinfo",
            ]
            pytest_args += [
                "-W ignore:the matrix subclass is not",
                "-W ignore:Importing from numpy.matlib is",
            ]
            if doctests:
                pytest_args += ["--doctest-modules"]
            if extra_argv:

```

```

124: (12)           pytest_args += list(extra_argv)
125: (8)            if verbose > 1:
126: (12)              pytest_args += ["-" + "v"*(verbose - 1)]
127: (8)            if coverage:
128: (12)              pytest_args += ["--cov=" + module_path]
129: (8)            if label == "fast":
130: (12)              from numpy.testing import IS_PYPY
131: (12)              if IS_PYPY:
132: (16)                  pytest_args += ["-m", "not slow and not slow_pypy"]
133: (12)              else:
134: (16)                  pytest_args += ["-m", "not slow"]
135: (8)            elif label != "full":
136: (12)              pytest_args += ["-m", label]
137: (8)            if durations >= 0:
138: (12)              pytest_args += ["--durations=%s" % durations]
139: (8)            if tests is None:
140: (12)              tests = [self.module_name]
141: (8)            pytest_args += ["--pyargs"] + list(tests)
142: (8)            _show_numpy_info()
143: (8)            try:
144: (12)                code = pytest.main(pytest_args)
145: (8)            except SystemExit as exc:
146: (12)                code = exc.code
147: (8)            return code == 0

```

---

## File 10 - \_\_config\_\_.py:

```

1: (0)          from enum import Enum
2: (0)          from numpy.core._multiarray_umath import (
3: (4)              __cpu_features__,
4: (4)              __cpu_baseline__,
5: (4)              __cpu_dispatch__,
6: (0)          )
7: (0)          __all__ = ["show"]
8: (0)          _built_with_meson = True
9: (0)          class DisplayModes(Enum):
10: (4)             stdout = "stdout"
11: (4)             dicts = "dicts"
12: (0)             def _cleanup(d):
13: (4)                 """
14: (4)                     Removes empty values in a `dict` recursively
15: (4)                     This ensures we remove values that Meson could not provide to CONFIG
16: (4)                 """
17: (4)                 if isinstance(d, dict):
18: (8)                     return {k: _cleanup(v) for k, v in d.items() if v and _cleanup(v)}
19: (4)                 else:
20: (8)                     return d
21: (0)             CONFIG = _cleanup(
22: (4)             {
23: (8)                 "Compilers": {
24: (12)                     "c": {
25: (16)                         "name": "msvc",
26: (16)                         "linker": r"link",
27: (16)                         "version": "19.29.30153",
28: (16)                         "commands": r"cl",
29: (16)                         "args": r"",
30: (16)                         "linker args": r"",
31: (12)                     },
32: (12)                     "cython": {
33: (16)                         "name": "cython",
34: (16)                         "linker": r"cython",
35: (16)                         "version": "3.0.8",
36: (16)                         "commands": r"cython",
37: (16)                         "args": r"",
38: (16)                         "linker args": r"",
39: (12)                     },
40: (12)                     "c++": {

```

```

41: (16)                     "name": "msvc",
42: (16)                     "linker": r"link",
43: (16)                     "version": "19.29.30153",
44: (16)                     "commands": r"cl",
45: (16)                     "args": r"",
46: (16)                     "linker args": r"",
47: (12)                     },
48: (8)                     },
49: (8)             "Machine Information": {
50: (12)                 "host": {
51: (16)                     "cpu": "x86_64",
52: (16)                     "family": "x86_64",
53: (16)                     "endian": "little",
54: (16)                     "system": "windows",
55: (12)                 },
56: (12)                 "build": {
57: (16)                     "cpu": "x86_64",
58: (16)                     "family": "x86_64",
59: (16)                     "endian": "little",
60: (16)                     "system": "windows",
61: (12)                 },
62: (12)                 "cross-compiled": bool("False".lower().replace("false", "")),
63: (8)             },
64: (8)             "Build Dependencies": {
65: (12)                 "blas": {
66: (16)                     "name": "openblas64",
67: (16)                     "found": bool("True".lower().replace("false", "")),
68: (16)                     "version": "0.3.23.dev",
69: (16)                     "detection method": "pkgconfig",
70: (16)                     "include directory": r"/c/opt/64/include",
71: (16)                     "lib directory": r"/c/opt/64/lib",
72: (16)                     "openblas configuration": r"USE_64BITINT=1 DYNAMIC_ARCH=1
DYNAMIC_OLDER= NO_CBLAS= NO_LAPACK= NO_LAPACKE= NO_AFFINITY=1 USE_OPENMP= SKYLAKE MAX_THREADS=2",
73: (16)                     "pc file directory": r"C:/opt/64/lib/pkgconfig",
74: (12)                 },
75: (12)                 "lapack": {
76: (16)                     "name": "dep3179274605568",
77: (16)                     "found": bool("True".lower().replace("false", "")),
78: (16)                     "version": "1.26.4",
79: (16)                     "detection method": "internal",
80: (16)                     "include directory": r"unknown",
81: (16)                     "lib directory": r"unknown",
82: (16)                     "openblas configuration": r"unknown",
83: (16)                     "pc file directory": r"unknown",
84: (12)                 },
85: (8)             },
86: (8)             "Python Information": {
87: (12)                 "path": r"C:\Users\runneradmin\AppData\Local\Temp\cibw-run-
g7slqwov\cp312-win_amd64\build\venv\Scripts\python.exe",
88: (12)                 "version": "3.12",
89: (8)             },
90: (8)             "SIMD Extensions": {
91: (12)                 "baseline": __cpu_baseline__,
92: (12)                 "found": [
93: (16)                     feature for feature in __cpu_dispatch__ if
__cpu_features__[feature]
94: (12)                 ],
95: (12)                 "not found": [
96: (16)                     feature for feature in __cpu_dispatch__ if not
__cpu_features__[feature]
97: (12)                 ],
98: (8)             },
99: (4)         },
100: (0)     }
101: (0)     def _check_pyyaml():
102: (4)         import yaml
103: (4)         return yaml
104: (0)     def show(mode=DisplayModes.stdout.value):
105: (4)         """

```

```

106: (4)             Show libraries and system information on which NumPy was built
107: (4)             and is being used
108: (4)             Parameters
109: (4)
110: (4)             -----
111: (8)             mode : {`stdout`, `dicts`}, optional.
112: (8)             Indicates how to display the config information.
113: (8)             `stdout` prints to console, `dicts` returns a dictionary
114: (4)             of the configuration.
115: (4)
116: (4)             Returns
117: (4)
118: (4)             -----
119: (4)             out : {`dict`, `None`}
120: (8)             If mode is `dicts`, a dict is returned, else None
121: (4)             See Also
122: (4)
123: (4)             -----
124: (4)             1. The `stdout` mode will give more readable
125: (7)             output if ``pyyaml`` is installed
126: (4)
127: (4)             """
128: (8)             if mode == DisplayModes.stdout.value:
129: (12)             try: # Non-standard library, check import
130: (12)                 yaml = _check_pyyaml()
131: (8)                 print(yaml.dump(CONFIG))
132: (12)             except ModuleNotFoundError:
133: (12)                 import warnings
134: (12)                 import json
135: (12)                 warnings.warn("Install `pyyaml` for better output", stacklevel=1)
136: (12)                 print(json.dumps(CONFIG, indent=2))
137: (4)             elif mode == DisplayModes_dicts.value:
138: (8)                 return CONFIG
139: (4)
140: (8)             else:
141: (12)                 raise AttributeError(
142: (12)                     f"Invalid `mode`, use one of: {', '.join([e.value for e in
DisplayModes])}")
143: (8)             )

```

---

**File 11 - \_\_init\_\_.py:**

```

1: (0)             """
2: (0)             NumPy
3: (0)             ====
4: (0)             Provides
5: (2)             1. An array object of arbitrary homogeneous items
6: (2)             2. Fast mathematical operations over arrays
7: (2)             3. Linear Algebra, Fourier Transforms, Random Number Generation
8: (0)             How to use the documentation
9: (0)
10: (0)             -----
11: (0)             Documentation is available in two forms: docstrings provided
12: (0)             with the code, and a loose standing reference guide, available from
13: (0)             `the NumPy homepage <https://numpy.org>`.
14: (0)             We recommend exploring the docstrings using
15: (0)             `IPython <https://ipython.org>`, an advanced Python shell with
16: (0)             TAB-completion and introspection capabilities. See below for further
17: (0)             instructions.
18: (2)             The docstring examples assume that `numpy` has been imported as ``np``::
19: (0)                 >>> import numpy as np
20: (2)             Code snippets are indicated by three greater-than signs::
21: (2)                 >>> x = 42
22: (2)                 >>> x = x + 1
23: (0)             Use the built-in ``help`` function to view a function's docstring::
24: (2)                 >>> help(np.sort)
25: (2)                 ... # doctest: +SKIP
26: (0)             For some objects, ``np.info(obj)`` may provide additional help. This is
27: (0)             particularly true if you see the line "Help on ufunc object:" at the top
28: (0)             of the help() page. Ufuncs are implemented in C, not Python, for speed.

```

```

28: (0)          The native Python help() does not know how to view their help, but our
29: (0)          np.info() function does.
30: (0)          To search for documents containing a keyword, do::
31: (2)            >>> np.lookfor('keyword')
32: (2)            ... # doctest: +SKIP
33: (0)          General-purpose documents like a glossary and help on the basic concepts
34: (0)          of numpy are available under the ``doc`` sub-module::
35: (2)            >>> from numpy import doc
36: (2)            >>> help(doc)
37: (2)            ... # doctest: +SKIP
38: (0)          Available subpackages
39: (0)          -----
40: (0)          lib
41: (4)            Basic functions used by several sub-packages.
42: (0)          random
43: (4)            Core Random Tools
44: (0)          linalg
45: (4)            Core Linear Algebra Tools
46: (0)          fft
47: (4)            Core FFT routines
48: (0)          polynomial
49: (4)            Polynomial tools
50: (0)          testing
51: (4)            NumPy testing tools
52: (0)          distutils
53: (4)            Enhancements to distutils with support for
54: (4)            Fortran compilers support and more (for Python <= 3.11).
55: (0)          Utilities
56: (0)          -----
57: (0)          test
58: (4)            Run numpy unittests
59: (0)          show_config
60: (4)            Show numpy build configuration
61: (0)          matlib
62: (4)            Make everything matrices.
63: (0)          __version__
64: (4)            NumPy version string
65: (0)          Viewing documentation using IPython
66: (0)          -----
67: (0)          Start IPython and import `numpy` usually under the alias ``np``: `import
68: (0)          numpy as np`. Then, directly past or use the ``%cpaste`` magic to paste
69: (0)          examples into the shell. To see which functions are available in `numpy`,
70: (0)          type ``np.<TAB>`` (where ``<TAB>`` refers to the TAB key), or use
71: (0)          ``np.*cos?<ENTER>`` (where ``<ENTER>`` refers to the ENTER key) to narrow
72: (0)          down the list. To view the docstring for a function, use
73: (0)          ``np.cos?<ENTER>`` (to view the docstring) and ``np.cos??<ENTER>`` (to view
74: (0)          the source code).
75: (0)          Copies vs. in-place operation
76: (0)          -----
77: (0)          Most of the functions in `numpy` return a copy of the array argument
78: (0)          (e.g., `np.sort`). In-place versions of these functions are often
79: (0)          available as array methods, i.e. ``x = np.array([1,2,3]); x.sort()``.
80: (0)          Exceptions to this rule are documented.
81: (0)          """
82: (0)          def _delvewheel_patch_1_5_2():
83: (4)            import os
84: (4)            libs_dir = os.path.abspath(os.path.join(os.path.dirname(__file__),
os.pardir, 'numpy.libs'))
85: (4)            if os.path.isdir(libs_dir):
86: (8)              os.add_dll_directory(libs_dir)
87: (0)            _delvewheel_patch_1_5_2()
88: (0)            del _delvewheel_patch_1_5_2
89: (0)            import sys
90: (0)            import warnings
91: (0)            from .globals import _NoValue, _CopyMode
92: (0)            from .exceptions import (
93: (4)              ComplexWarning, ModuleDeprecationWarning, VisibleDeprecationWarning,
94: (4)              TooHardError, AxisError)
95: (0)            from . import version

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

96: (0)
97: (0)
98: (4)
99: (0)
100: (4)
101: (0)
102: (4)
103: (0)
104: (4)
105: (4)
106: (8)
107: (4)
108: (8)
from
109: (8)
110: (8)
111: (8)
112: (4)
113: (8)
114: (8)
115: (4)
116: (4)
117: (4)
118: (4)
119: (4)
120: (4)
121: (4)
122: (4)
123: (4)
124: (4)
125: (4)
126: (4)
127: (4)
128: (4)
129: (4)
130: (4)
131: (4)
132: (4)
_extended_msg}\n"
133: (8)
134: (8)
135: (8)
136: (8)
_msg = (
    "module 'numpy' has no attribute '{n}'.\n"
    "`np.{n}` was a deprecated alias for the builtin `{n}`. "
    "To avoid this error in existing code, use `{n}` by itself. "
    "Doing this will not modify any behavior and is safe."
notes.html#deprecations")
137: (8)
138: (8)
139: (8)
notes.html#deprecations")
140: (4)
_specific_msg = (
    "If you specifically wanted the numpy scalar type, use `np.{}` here.")
_int_extended_msg = (
    "When replacing `np.{}`, you may wish to use e.g. `np.int64` "
    "or `np.int32` to specify the precision. If you wish to review "
    "your current use, check the release note link for "
    "additional information.")
_type_info = [
    ("object", ""), # The NumPy scalar only exists by name.
    ("bool", _specific_msg.format("bool_")),
    ("float", _specific_msg.format("float64")),
    ("complex", _specific_msg.format("complex128")),
    ("str", _specific_msg.format("str_")),
    ("int", _int_extended_msg.format("int"))]
_former_attrs__ = {
    n: _msg.format(n=n, extended_msg=extended_msg)
    for n, extended_msg in _type_info
}
_msg = (
    "`np.{n}` is a deprecated alias for `{an}`. (Deprecated NumPy 1.24)")
_type_info = [
    ("bool8", bool_, "np.bool_"),

```

```

162: (8) ("int0", intp, "np.intp"),
163: (8) ("uint0", uintp, "np.uintp"),
164: (8) ("str0", str_, "np.str_"),
165: (8) ("bytes0", bytes_, "np.bytes_"),
166: (8) ("void0", void, "np void"),
167: (8) ("object0", object_,
168: (12)         "`np.object0` is a deprecated alias for `np.object_`. "
169: (12)         "`object` can be used instead. (Deprecated NumPy 1.24)")]
170: (4) __future_scalars__ = {"bool", "long", "ulong", "str", "bytes", "object"}
171: (4) __deprecated_attrs__.update({
172: (8)     n: (alias, _msg.format(n=n, an=an)) for n, alias, an in __type_info__)
173: (4) import math
174: (4) __deprecated_attrs__['math'] = (math,
175: (8)         "`np.math` is a deprecated alias for the standard library `math` "
176: (8)         "module (Deprecated NumPy 1.25). Replace usages of `np.math` with "
177: (8)         "`math`")
178: (4) del math, _msg, __type_info__
179: (4) from .core import abs
180: (4) core.getlimits._register_known_types()
181: (4) __all__.extend(['__version__', 'show_config'])
182: (4) __all__.extend(core.__all__)
183: (4) __all__.extend(_mat.__all__)
184: (4) __all__.extend(lib.__all__)
185: (4) __all__.extend(['linalg', 'fft', 'random', 'ctypeslib', 'ma'])
186: (4) __all__.remove('min')
187: (4) __all__.remove('max')
188: (4) __all__.remove('round')
189: (4) __all__.remove('issubdtype')
190: (4) del long, unicode
191: (4) __all__.remove('long')
192: (4) __all__.remove('unicode')
193: (4) __all__.remove('Arrayterator')
194: (4) del Arrayterator
195: (4) __financial_names = ['fv', 'ipmt', 'irr', 'mirr', 'nper', 'npv', 'pmt',
196: (24)                 'ppmt', 'pv', 'rate']
197: (4) __expired_functions__ = {
198: (8)     name: (f'In accordance with NEP 32, the function {name} was removed '
199: (15)             'from NumPy version 1.20. A replacement for this function '
200: (15)             'is available in the numpy_financial library: '
201: (15)             'https://pypi.org/project/numpy-financial')
202: (8)     for name in __financial_names}
203: (4) warnings.filterwarnings("ignore", message="numpy.dtype size changed")
204: (4) warnings.filterwarnings("ignore", message="numpy.ufunc size changed")
205: (4) warnings.filterwarnings("ignore", message="numpy.ndarray size changed")
206: (4) oldnumeric = 'removed'
207: (4) numarray = 'removed'
208: (4) def __getattr__(attr):
209: (8)     import warnings
210: (8)     import math
211: (8)     try:
212: (12)         msg = __expired_functions__[attr]
213: (8)     except KeyError:
214: (12)         pass
215: (8)     else:
216: (12)         warnings.warn(msg, DeprecationWarning, stacklevel=2)
217: (12)         def __expired(*args, **kwds):
218: (16)             raise RuntimeError(msg)
219: (12)             return __expired
220: (8)     try:
221: (12)         val, msg = __deprecated_attrs__[attr]
222: (8)     except KeyError:
223: (12)         pass
224: (8)     else:
225: (12)         warnings.warn(msg, DeprecationWarning, stacklevel=2)
226: (12)         return val
227: (8)     if attr in __future_scalars__:
228: (12)         warnings.warn(
229: (16)             f"In the future `np.{attr}` will be defined as the "
230: (16)             "corresponding NumPy scalar.", FutureWarning, stacklevel=2)

```

```

231: (8)
232: (12)
233: (8)
234: (12)
235: (12)
236: (8)
237: (12)
238: (12)
239: (8)
240: (29)
241: (4)
242: (8)
243: (8)
244: (12)
245: (12)
246: (12)
247: (8)
248: (8)
249: (4)
250: (4)
251: (4)
252: (4)
253: (8)
254: (8)
255: (8)
256: (8)
257: (8)
early.
258: (8)
259: (8)
260: (8)
261: (8)
262: (12)
263: (12)
264: (16)
265: (8)
266: (12)
267: (19)
"
268: (19)
269: (19)
270: (19)
271: (12)
272: (4)
273: (4)
274: (4)
275: (8)
276: (8)
277: (8)
278: (8)
279: (8)
280: (12)
281: (12)
282: (12)
283: (12)
284: (8)
285: (12)
286: (4)
287: (8)
288: (8)
289: (12)
290: (12)
291: (16)
292: (20)
293: (24)
{str(_wn.message)}"
294: (24)
295: (28)
likely due "

```

```

        if attr in __former_attrs__:
            raise AttributeError(__former_attrs__[attr])
        if attr == 'testing':
            import numpy.testing as testing
            return testing
        elif attr == 'Tester':
            "Removed in NumPy 1.25.0"
            raise RuntimeError("Tester was removed in NumPy 1.25.")
        raise AttributeError("module {!r} has no attribute "
                            "{!r}".format(__name__, attr))
    def __dir__():
        public_symbols = globals().keys() | {'testing'}
        public_symbols -= {
            "core", "matrixlib",
            "ModuleDeprecationWarning", "VisibleDeprecationWarning",
            "ComplexWarning", "TooHardError", "AxisError"
        }
        return list(public_symbols)
    from numpy._pytesttester import PytestTester
    test = PytestTester(__name__)
    del PytestTester
    def _sanity_check():
        """
        Quick sanity checks for common bugs caused by environment.
        There are some cases e.g. with wrong BLAS ABI that cause wrong
        results under specific runtime conditions that are not necessarily
        achieved during test suite runs, and it is useful to catch those
        early.
        See https://github.com/numpy/numpy/issues/8577 and other
        similar bug reports.
        """
        try:
            x = ones(2, dtype=float32)
            if not abs(x.dot(x) - float32(2.0)) < 1e-5:
                raise AssertionError()
        except AssertionError:
            msg = ("The current Numpy installation ({!r}) fails to "
                   "pass simple sanity checks. This can be caused for example
                   "by incorrect BLAS library being linked in, or by mixing "
                   "package managers (pip, conda, apt, ...). Search closed "
                   "numpy issues for similar problems.")
            raise RuntimeError(msg.format(__file__)) from None
    _sanity_check()
    del _sanity_check
    def _mac_os_check():
        """
        Quick Sanity check for Mac OS look for accelerate build bugs.
        Testing numpy polyfit calls init_dgelsd(LAPACK)
        """
        try:
            c = array([3., 2., 1.])
            x = linspace(0, 2, 5)
            y = polyval(c, x)
            _ = polyfit(x, y, 2, cov=True)
        except ValueError:
            pass
    if sys.platform == "darwin":
        from . import exceptions
        with warnings.catch_warnings(record=True) as w:
            _mac_os_check()
            if len(w) > 0:
                for _wn in w:
                    if _wn.category is exceptions.RankWarning:
                        error_message = f"({_wn.category.__name__}):"
                        msg = (
                            "Polyfit sanity test emitted a warning, most"

```

```

296: (28)                                "to using a buggy Accelerate backend."
297: (28)                                "\nIf you compiled yourself, more information is
available at:"
298: (28)                                "\nhttps://numpy.org/devdocs/building/index.html"
299: (28)                                "\nOtherwise report this to the vendor"
300: (28)                                "that provided
NumPy.\n\n{}\n".format(error_message))
301: (24)                                raise RuntimeError(msg)
302: (16)                                del _wn
303: (12)                                del w
304: (4)                                 del _mac_os_check
305: (4)                                 import os
306: (4)                                 use_hugepage = os.environ.get("NUMPY_MADVISE_HUGEPAGE", None)
307: (4)                                 if sys.platform == "linux" and use_hugepage is None:
308: (8)                                  try:
309: (12)                                    use_hugepage = 1
310: (12)                                    kernel_version = os.uname().release.split(".")[:2]
311: (12)                                    kernel_version = tuple(int(v) for v in kernel_version)
312: (12)                                    if kernel_version < (4, 6):
313: (16)                                      use_hugepage = 0
314: (8)                                     except ValueError:
315: (12)                                       use_hugepages = 0
316: (4)                                     elif use_hugepage is None:
317: (8)                                       use_hugepage = 1
318: (4)                                     else:
319: (8)                                       use_hugepage = int(use_hugepage)
320: (4)                                     core.multiarray._set_madvise_hugepage(use_hugepage)
321: (4)                                     del use_hugepage
322: (4)                                     core.multiarray._multiarray_umath._reload_guard()
323: (4)                                     core._set_promotion_state(
324: (8)                                       os.environ.get("NPY_PROMOTION_STATE",
325: (23)                                         "weak" if _using_numpy2_behavior() else "legacy"))
326: (4)                                     def _pyinstaller_hooks_dir():
327: (8)                                       from pathlib import Path
328: (8)                                       return [str(Path(__file__).with_name("_pyinstaller").resolve())]
329: (4)                                     del os
330: (0)                                     del sys, warnings
-----
```

## File 12 - setup.py:

```

1: (0)          def configuration(parent_package="", top_path=None):
2: (4)            from numpy.distutils.misc_util import Configuration
3: (4)            config = Configuration("array_api", parent_package, top_path)
4: (4)            config.add_subpackage("tests")
5: (4)            return config
6: (0)          if __name__ == "__main__":
7: (4)            from numpy.distutils.core import setup
8: (4)            setup(configuration=configuration)
-----
```

## File 13 - linalg.py:

```

1: (0)          from __future__ import annotations
2: (0)          from ._dtypes import (
3: (4)            _floating_dtotypes,
4: (4)            _numeric_dtotypes,
5: (4)            float32,
6: (4)            float64,
7: (4)            complex64,
8: (4)            complex128
9: (0)          )
10: (0)         from ._manipulation_functions import reshape
11: (0)         from ._elementwise_functions import conj
12: (0)         from ._array_object import Array
13: (0)         from ..core.numeric import normalize_axis_tuple
14: (0)         from typing import TYPE_CHECKING
-----
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

15: (0)
16: (4) if TYPE_CHECKING:
17: (0)     from ._typing import Literal, Optional, Sequence, Tuple, Union, Dtype
18: (0)     from typing import NamedTuple
19: (0)     import numpy.linalg
20: (0)     import numpy as np
21: (4)     class EighResult(NamedTuple):
22: (4)         eigenvalues: Array
23: (0)         eigenvectors: Array
24: (4)     class QRResult(NamedTuple):
25: (4)         Q: Array
26: (0)         R: Array
27: (4)     class SlogdetResult(NamedTuple):
28: (4)         sign: Array
29: (0)         logabsdet: Array
30: (4)     class SVDResult(NamedTuple):
31: (4)         U: Array
32: (4)         S: Array
33: (4)         Vh: Array
34: (0)     def cholesky(x: Array, /, *, upper: bool = False) -> Array:
35: (4)         """
36: (4)             Array API compatible wrapper for :py:func:`np.linalg.cholesky`
<numpy.linalg.cholesky>.
37: (4)             See its docstring for more information.
38: (4)             """
39: (8)             if x.dtype not in _floating_dtypes:
40: (4)                 raise TypeError('Only floating-point dtypes are allowed in cholesky')
41: (4)             L = np.linalg.cholesky(x._array)
42: (8)             if upper:
43: (8)                 U = Array._new(L).mT
44: (12)                 if U.dtype in [complex64, complex128]:
45: (8)                     U = conj(U)
46: (4)                 return U
47: (0)             return Array._new(L)
48: (4)     def cross(x1: Array, x2: Array, /, *, axis: int = -1) -> Array:
49: (4)         """
50: (4)             Array API compatible wrapper for :py:func:`np.cross <numpy.cross>` .
51: (4)             See its docstring for more information.
52: (4)             """
53: (8)             if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:
54: (8)                 raise TypeError('Only numeric dtypes are allowed in cross')
55: (4)             if x1.shape != x2.shape:
56: (8)                 raise ValueError('x1 and x2 must have the same shape')
57: (4)             if x1.ndim == 0:
58: (8)                 raise ValueError('cross() requires arrays of dimension at least 1')
59: (4)             if x1.shape[axis] != 3:
60: (8)                 raise ValueError('cross() dimension must equal 3')
61: (4)             return Array._new(np.cross(x1._array, x2._array, axis=axis))
62: (0)     def det(x: Array, /) -> Array:
63: (4)         """
64: (4)             Array API compatible wrapper for :py:func:`np.linalg.det
<numpy.linalg.det>`.
65: (4)             See its docstring for more information.
66: (4)             """
67: (8)             if x.dtype not in _floating_dtypes:
68: (8)                 raise TypeError('Only floating-point dtypes are allowed in det')
69: (4)             return Array._new(np.linalg.det(x._array))
70: (0)     def diagonal(x: Array, /, *, offset: int = 0) -> Array:
71: (4)         """
72: (4)             Array API compatible wrapper for :py:func:`np.diagonal <numpy.diagonal>` .
73: (4)             See its docstring for more information.
74: (4)             """
75: (0)             return Array._new(np.diagonal(x._array, offset=offset, axis1=-2,
axis2=-1))
76: (4)     def eigh(x: Array, /) -> EighResult:
77: (4)         """
78: (4)             Array API compatible wrapper for :py:func:`np.linalg.eigh
<numpy.linalg.eigh>`.
79: (4)             See its docstring for more information.
    """

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

80: (4)             if x.dtype not in _floating_dtypes:
81: (8)                 raise TypeError('Only floating-point dtypes are allowed in eigh')
82: (4)             return EighResult(*map(Array._new, np.linalg.eigh(x._array)))
83: (0)         def eigvalsh(x: Array, /) -> Array:
84: (4)             """
85: (4)                 Array API compatible wrapper for :py:func:`np.linalg.eigvalsh
<numpy.linalg.eigvalsh>`.
86: (4)                 See its docstring for more information.
87: (4)             """
88: (4)             if x.dtype not in _floating_dtypes:
89: (8)                 raise TypeError('Only floating-point dtypes are allowed in eigvalsh')
90: (4)             return Array._new(np.linalg.eigvalsh(x._array))
91: (0)         def inv(x: Array, /) -> Array:
92: (4)             """
93: (4)                 Array API compatible wrapper for :py:func:`np.linalg.inv
<numpy.linalg.inv>`.
94: (4)                 See its docstring for more information.
95: (4)             """
96: (4)             if x.dtype not in _floating_dtypes:
97: (8)                 raise TypeError('Only floating-point dtypes are allowed in inv')
98: (4)                 return Array._new(np.linalg.inv(x._array))
99: (0)         def matmul(x1: Array, x2: Array, /) -> Array:
100: (4)             """
101: (4)                 Array API compatible wrapper for :py:func:`np.matmul <numpy.matmul>`.
102: (4)                 See its docstring for more information.
103: (4)             """
104: (4)             if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:
105: (8)                 raise TypeError('Only numeric dtypes are allowed in matmul')
106: (4)                 return Array._new(np.matmul(x1._array, x2._array))
107: (0)         def matrix_norm(x: Array, /, *, keepdims: bool = False, ord:
Optional[Union[int, float, Literal['fro', 'nuc']]] = 'fro') -> Array:
108: (4)             """
109: (4)                 Array API compatible wrapper for :py:func:`np.linalg.norm
<numpy.linalg.norm>`.
110: (4)                 See its docstring for more information.
111: (4)             """
112: (4)             if x.dtype not in _floating_dtypes:
113: (8)                 raise TypeError('Only floating-point dtypes are allowed in
matrix_norm')
114: (4)                 return Array._new(np.linalg.norm(x._array, axis=(-2, -1),
keepdims=keepdims, ord=ord))
115: (0)         def matrix_power(x: Array, n: int, /) -> Array:
116: (4)             """
117: (4)                 Array API compatible wrapper for :py:func:`np.matrix_power
<numpy.matrix_power>`.
118: (4)                 See its docstring for more information.
119: (4)             """
120: (4)             if x.dtype not in _floating_dtypes:
121: (8)                 raise TypeError('Only floating-point dtypes are allowed for the first
argument of matrix_power')
122: (4)                 return Array._new(np.linalg.matrix_power(x._array, n))
123: (0)         def matrix_rank(x: Array, /, *, rtol: Optional[Union[float, Array]] = None) ->
Array:
124: (4)             """
125: (4)                 Array API compatible wrapper for :py:func:`np.matrix_rank
<numpy.matrix_rank>`.
126: (4)                 See its docstring for more information.
127: (4)             """
128: (4)             if x.ndim < 2:
129: (8)                 raise np.linalg.LinAlgError("1-dimensional array given. Array must be
at least two-dimensional")
130: (4)                 S = np.linalg.svd(x._array, compute_uv=False)
131: (4)                 if rtol is None:
132: (8)                     tol = S.max(axis=-1, keepdims=True) * max(x.shape[-2:]) *
np.finfo(S.dtype).eps
133: (4)                 else:
134: (8)                     if isinstance(rtol, Array):
135: (12)                         rtol = rtol._array
136: (8)                     tol = S.max(axis=-1, keepdims=True)*np.asarray(rtol)[..., np.newaxis]

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

137: (4)             return Array._new(np.count_nonzero(S > tol, axis=-1))
138: (0)             def matrix_transpose(x: Array, /) -> Array:
139: (4)                 if x.ndim < 2:
140: (8)                     raise ValueError("x must be at least 2-dimensional for
matrix_transpose")
141: (4)             return Array._new(np.swapaxes(x._array, -1, -2))
142: (0)             def outer(x1: Array, x2: Array, /) -> Array:
143: (4)                 """
144: (4)                     Array API compatible wrapper for :py:func:`np.outer <numpy.outer>` .
145: (4)                     See its docstring for more information.
146: (4)                 """
147: (4)                 if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:
148: (8)                     raise TypeError('Only numeric dtypes are allowed in outer')
149: (4)                 if x1.ndim != 1 or x2.ndim != 1:
150: (8)                     raise ValueError('The input arrays to outer must be 1-dimensional')
151: (4)                 return Array._new(np.outer(x1._array, x2._array))
152: (0)             def pinv(x: Array, /, *, rtol: Optional[Union[float, Array]] = None) -> Array:
153: (4)                 """
154: (4)                     Array API compatible wrapper for :py:func:`np.linalg.pinv
<numpy.linalg.pinv>` .
155: (4)                     See its docstring for more information.
156: (4)                 """
157: (4)                 if x.dtype not in _floating_dtypes:
158: (8)                     raise TypeError('Only floating-point dtypes are allowed in pinv')
159: (4)                 if rtol is None:
160: (8)                     rtol = max(x.shape[-2:]) * np.finfo(x.dtype).eps
161: (4)                 return Array._new(np.linalg.pinv(x._array, rcond=rtol))
162: (0)             def qr(x: Array, /, *, mode: Literal['reduced', 'complete'] = 'reduced') ->
QRResult:
163: (4)                 """
164: (4)                     Array API compatible wrapper for :py:func:`np.linalg.qr
<numpy.linalg.qr>` .
165: (4)                     See its docstring for more information.
166: (4)                 """
167: (4)                 if x.dtype not in _floating_dtypes:
168: (8)                     raise TypeError('Only floating-point dtypes are allowed in qr')
169: (4)                 return QRResult(*map(Array._new, np.linalg.qr(x._array, mode=mode)))
170: (0)             def slogdet(x: Array, /) -> SlogdetResult:
171: (4)                 """
172: (4)                     Array API compatible wrapper for :py:func:`np.linalg.slogdet
<numpy.linalg.slogdet>` .
173: (4)                     See its docstring for more information.
174: (4)                 """
175: (4)                 if x.dtype not in _floating_dtypes:
176: (8)                     raise TypeError('Only floating-point dtypes are allowed in slogdet')
177: (4)                 return SlogdetResult(*map(Array._new, np.linalg.slogdet(x._array)))
178: (0)             def _solve(a, b):
179: (4)                 from ..linalg.linalg import (_makearray, _assert_stacked_2d,
180: (33)                               _assert_stacked_square, _commonType,
181: (33)                               isComplexType, get_linalg_error_extobj,
182: (33)                               _raise_linalgerror_singular)
183: (4)                 from ..linalg import _umath_linalg
184: (4)                 a, _ = _makearray(a)
185: (4)                 _assert_stacked_2d(a)
186: (4)                 _assert_stacked_square(a)
187: (4)                 b, wrap = _makearray(b)
188: (4)                 t, result_t = _commonType(a, b)
189: (4)                 if b.ndim == 1:
190: (8)                     gufunc = _umath_linalg.solve1
191: (4)                 else:
192: (8)                     gufunc = _umath_linalg.solve
193: (4)                     signature = 'DD->D' if isComplexType(t) else 'dd->d'
194: (4)                     with np.errstate(call=_raise_linalgerror_singular, invalid='call',
195: (21)                           over='ignore', divide='ignore', under='ignore'):
196: (8)                         r = gufunc(a, b, signature=signature)
197: (4)                         return wrap(r.astype(result_t, copy=False))
198: (0)             def solve(x1: Array, x2: Array, /) -> Array:
199: (4)                 """
200: (4)                     Array API compatible wrapper for :py:func:`np.linalg.solve

```

```

<numpy.linalg.solve>`.
201: (4)             See its docstring for more information.
202: (4)
203: (4)
204: (8)             if x1.dtype not in _floating_dtypes or x2.dtype not in _floating_dtypes:
205: (4)                 raise TypeError('Only floating-point dtypes are allowed in solve')
206: (0)             return Array._new(_solve(x1._array, x2._array))
207: (4)
208: (4)             def svd(x: Array, /, *, full_matrices: bool = True) -> SVDResult:
209: (4)                 """
210: (4)                     Array API compatible wrapper for :py:func:`np.linalg.svd`
<numpy.linalg.svd>`.
211: (4)             See its docstring for more information.
212: (4)
213: (4)             if x.dtype not in _floating_dtypes:
214: (8)                 raise TypeError('Only floating-point dtypes are allowed in svd')
215: (4)             return SVDResult(*map(Array._new, np.linalg.svd(x._array,
216: (4)                         full_matrices=full_matrices)))
217: (0)             def svdvals(x: Array, /) -> Union[Array, Tuple[Array, ...]]:
218: (4)                 if x.dtype not in _floating_dtypes:
219: (8)                     raise TypeError('Only floating-point dtypes are allowed in svdvals')
220: (4)                 return Array._new(np.linalg.svd(x._array, compute_uv=False))
221: (4)             def tensordot(x1: Array, x2: Array, /, *, axes: Union[int,
222: (0)                             Tuple[Sequence[int], Sequence[int]]] = 2) -> Array:
223: (4)                 if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:
224: (8)                     raise TypeError('Only numeric dtypes are allowed in tensordot')
225: (4)                     return Array._new(np.tensordot(x1._array, x2._array, axes=axes))
226: (0)             def trace(x: Array, /, *, offset: int = 0, dtype: Optional[Dtype] = None) ->
227: (4)             Array:
228: (4)                 """
229: (4)                     Array API compatible wrapper for :py:func:`np.trace <numpy.trace>`.
230: (4)                     See its docstring for more information.
231: (4)
232: (4)                     if x.dtype not in _numeric_dtypes:
233: (8)                         raise TypeError('Only numeric dtypes are allowed in trace')
234: (4)                     if dtype is None:
235: (8)                         if x.dtype == float32:
236: (12)                             dtype = float64
237: (8)                         elif x.dtype == complex64:
238: (12)                             dtype = complex128
239: (4)                     return Array._new(np.asarray(np.trace(x._array, offset=offset, axis1=-2,
240: (0)                         axis2=-1, dtype=dtype)))
241: (0)             def vecdot(x1: Array, x2: Array, /, *, axis: int = -1) -> Array:
242: (4)                 if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:
243: (8)                     raise TypeError('Only numeric dtypes are allowed in vecdot')
244: (4)                 ndim = max(x1.ndim, x2.ndim)
245: (4)                 x1_shape = (1,)*(ndim - x1.ndim) + tuple(x1.shape)
246: (4)                 x2_shape = (1,)*(ndim - x2.ndim) + tuple(x2.shape)
247: (4)                 if x1_shape[axis] != x2_shape[axis]:
248: (8)                     raise ValueError("x1 and x2 must have the same size along the given
249: (4)                         axis")
250: (4)                 x1_, x2_ = np.broadcast_arrays(x1._array, x2._array)
251: (4)                 x1_ = np.moveaxis(x1_, axis, -1)
252: (4)                 x2_ = np.moveaxis(x2_, axis, -1)
253: (4)                 res = x1_[..., None, :] @ x2_[..., None]
254: (4)                 return Array._new(res[..., 0, 0])
255: (0)             def vector_norm(x: Array, /, *, axis: Optional[Union[int, Tuple[int, ...]]] =
256: (4)                         None, keepdims: bool = False, ord: Optional[Union[int, float]] = 2) -> Array:
257: (4)                 """
258: (4)                     Array API compatible wrapper for :py:func:`np.linalg.norm
<numpy.linalg.norm>`.
259: (4)                     See its docstring for more information.
260: (4)
261: (4)                     if x.dtype not in _floating_dtypes:
262: (8)                         raise TypeError('Only floating-point dtypes are allowed in norm')
263: (4)                     a = x._array
264: (4)                     if axis is None:
265: (8)                         a = a.ravel()
266: (8)                         _axis = 0
267: (4)                     elif isinstance(axis, tuple):
268: (8)                         normalized_axis = normalize_axis_tuple(axis, x.ndim)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

261: (8)             rest = tuple(i for i in range(a.ndim) if i not in normalized_axis)
262: (8)             newshape = axis + rest
263: (8)             a = np.transpose(a, newshape).reshape(
264: (12)                 (np.prod([a.shape[i] for i in axis], dtype=int), *[a.shape[i] for
i in rest]))
265: (8)             _axis = 0
266: (4)         else:
267: (8)             _axis = axis
268: (4)         res = Array._new(np.linalg.norm(a, axis=_axis, ord=ord))
269: (4)         if keepdims:
270: (8)             shape = list(x.shape)
271: (8)             _axis = normalize_axis_tuple(range(x.ndim)) if axis is None else axis,
x.ndim)
272: (8)             for i in _axis:
273: (12)                 shape[i] = 1
274: (8)             res = reshape(res, tuple(shape))
275: (4)         return res
276: (0)     __all__ = ['cholesky', 'cross', 'det', 'diagonal', 'eigh', 'eigvalsh', 'inv',
'matmul', 'matrix_norm', 'matrix_power', 'matrix_rank', 'matrix_transpose', 'outer', 'pinv', 'qr',
'slogdet', 'solve', 'svd', 'svdvals', 'tensordot', 'trace', 'vecdot', 'vector_norm']
-----
```

#### File 14 - \_array\_object.py:

```

1: (0) """
2: (0)     Wrapper class around the ndarray object for the array API standard.
3: (0)     The array API standard defines some behaviors differently than ndarray, in
4: (0)     particular, type promotion rules are different (the standard has no
5: (0)     value-based casting). The standard also specifies a more limited subset of
6: (0)     array methods and functionalities than are implemented on ndarray. Since the
7: (0)     goal of the array_api namespace is to be a minimal implementation of the array
8: (0)     API standard, we need to define a separate wrapper class for the array_api
9: (0)     namespace.
10: (0)    The standard compliant class is only a wrapper class. It is *not* a subclass
11: (0)     of ndarray.
12: (0) """
13: (0) from __future__ import annotations
14: (0) import operator
15: (0) from enum import IntEnum
16: (0) from ._creation_functions import asarray
17: (0) from ._dtypes import (
18: (4)     _all_dtypes,
19: (4)     _boolean_dtypes,
20: (4)     _integer_dtypes,
21: (4)     _integer_or_boolean_dtypes,
22: (4)     _floating_dtypes,
23: (4)     _complex_floating_dtypes,
24: (4)     _numeric_dtypes,
25: (4)     _result_type,
26: (4)     _dtype_categories,
27: (0) )
28: (0) from typing import TYPE_CHECKING, Optional, Tuple, Union, Any, SupportsIndex
29: (0) import types
30: (0) if TYPE_CHECKING:
31: (4)     from ._typing import Any, PyCapsule, Device, Dtype
32: (4)     import numpy.typing as npt
33: (0) import numpy as np
34: (0) from numpy import array_api
35: (0) class Array:
36: (4) """
37: (4)     n-d array object for the array API namespace.
38: (4)     See the docstring of :py:obj:`np.ndarray <numpy.ndarray>` for more
39: (4)     information.
40: (4)     This is a wrapper around numpy.ndarray that restricts the usage to only
41: (4)     those things that are required by the array API namespace. Note,
42: (4)     attributes on this object that start with a single underscore are not part
43: (4)     of the API specification and should only be used internally. This object
44: (4)     should not be constructed directly. Rather, use one of the creation
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

45: (4)             functions, such as asarray().
46: (4)
47: (4)             """
48: (4)             _array: np.ndarray[Any, Any]
49: (4)             @classmethod
50: (8)             def __new__(cls, x, /):
51: (8)                 """
52: (8)                 This is a private method for initializing the array API Array
53: (8)                 object.
54: (8)                 Functions outside of the array_api submodule should not use this
55: (8)                 method. Use one of the creation functions instead, such as
56: (8)                 ``asarray``.
57: (8)
58: (8)
59: (12)             """
60: (8)             obj = super().__new__(cls)
61: (12)             if isinstance(x, np.generic):
62: (8)                 x = np.asarray(x)
63: (12)             if x.dtype not in _all_dtypes:
64: (8)                 raise TypeError(
65: (8)                     f"The array_api namespace does not support the dtype
66: ({x.dtype})"
67: (8)
68: (12)             )
69: (8)
70: (4)             def __new__(cls, *args, **kwargs):
71: (8)                 raise TypeError(
72: (8)                     "The array_api Array object should not be instantiated directly.
73: (8)                     Use an array creation function, such as asarray(), instead."
74: (8)
75: (4)             def __str__(self: Array, /) -> str:
76: (8)                 """
77: (8)                 Performs the operation __str__.
78: (8)
79: (8)                 return self._array.__str__().replace("array", "Array")
80: (4)             def __repr__(self: Array, /) -> str:
81: (8)
82: (8)
83: (8)
84: (12)
85: (12)             suffix = f", dtype={self.dtype.name})"
86: (8)
87: (4)             if 0 in self.shape:
88: (8)                 prefix = "empty("
89: (8)                 mid = str(self.shape)
90: (8)
91: (8)
92: (8)
93: (8)
94: (4)             else:
95: (8)                 prefix = "Array("
96: (8)                 mid = np.array2string(self._array, separator=', ', prefix=prefix,
97: (8)                     suffix=suffix)
98: (8)
99: (4)             return prefix + mid + suffix
100: (4)             def __array__(self, dtype: None | np.dtype[Any] = None) ->
101: (8)                 """
102: (8)                     Warning: this method is NOT part of the array API spec. Implementers
103: (8)                     of other libraries need not include it, and users should not assume it
104: (8)                     will be present in other implementations.
105: (8)
106: (8)
107: (12)                     """
108: (8)                     Helper function for operators to only allow specific input dtypes
109: (8)                     Use like
110: (12)                     other = self._check_allowed_dtypes(other, 'numeric', '__add__')
111: (12)                     if other is NotImplemented:
112: (8)                         return other
113: (8)
114: (8)                     if self.dtype not in _dtype_categories[dtype_category]:
115: (8)                         raise TypeError(f"Only {dtype_category} dtypes are allowed in
116: ({op})")
117: (8)
118: (8)                     if isinstance(other, (int, complex, float, bool)):
119: (8)                         other = self._promote_scalar(other)
120: (8)
121: (8)                     elif isinstance(other, Array):
122: (8)                         if other.dtype not in _dtype_categories[dtype_category]:
123: (8)
124: (8)
125: (8)
126: (8)
127: (8)
128: (8)
129: (8)
130: (8)
131: (8)
132: (8)
133: (8)
134: (8)
135: (8)
136: (8)
137: (8)
138: (8)
139: (8)
140: (8)
141: (8)
142: (8)
143: (8)
144: (8)
145: (8)
146: (8)
147: (8)
148: (8)
149: (8)
150: (8)
151: (8)
152: (8)
153: (8)
154: (8)
155: (8)
156: (8)
157: (8)
158: (8)
159: (8)
160: (8)
161: (8)
162: (8)
163: (8)
164: (8)
165: (8)
166: (8)
167: (8)
168: (8)
169: (8)
170: (8)
171: (8)
172: (8)
173: (8)
174: (8)
175: (8)
176: (8)
177: (8)
178: (8)
179: (8)
180: (8)
181: (8)
182: (8)
183: (8)
184: (8)
185: (8)
186: (8)
187: (8)
188: (8)
189: (8)
190: (8)
191: (8)
192: (8)
193: (8)
194: (8)
195: (8)
196: (8)
197: (8)
198: (8)
199: (8)
200: (8)
201: (8)
202: (8)
203: (8)
204: (8)
205: (8)
206: (8)
207: (8)
208: (8)
209: (8)
210: (8)
211: (8)
212: (8)
213: (8)
214: (8)
215: (8)
216: (8)
217: (8)
218: (8)
219: (8)
220: (8)
221: (8)
222: (8)
223: (8)
224: (8)
225: (8)
226: (8)
227: (8)
228: (8)
229: (8)
230: (8)
231: (8)
232: (8)
233: (8)
234: (8)
235: (8)
236: (8)
237: (8)
238: (8)
239: (8)
240: (8)
241: (8)
242: (8)
243: (8)
244: (8)
245: (8)
246: (8)
247: (8)
248: (8)
249: (8)
250: (8)
251: (8)
252: (8)
253: (8)
254: (8)
255: (8)
256: (8)
257: (8)
258: (8)
259: (8)
260: (8)
261: (8)
262: (8)
263: (8)
264: (8)
265: (8)
266: (8)
267: (8)
268: (8)
269: (8)
270: (8)
271: (8)
272: (8)
273: (8)
274: (8)
275: (8)
276: (8)
277: (8)
278: (8)
279: (8)
280: (8)
281: (8)
282: (8)
283: (8)
284: (8)
285: (8)
286: (8)
287: (8)
288: (8)
289: (8)
290: (8)
291: (8)
292: (8)
293: (8)
294: (8)
295: (8)
296: (8)
297: (8)
298: (8)
299: (8)
300: (8)
301: (8)
302: (8)
303: (8)
304: (8)
305: (8)
306: (8)
307: (8)
308: (8)
309: (8)
310: (8)
311: (8)
312: (8)
313: (8)
314: (8)
315: (8)
316: (8)
317: (8)
318: (8)
319: (8)
320: (8)
321: (8)
322: (8)
323: (8)
324: (8)
325: (8)
326: (8)
327: (8)
328: (8)
329: (8)
330: (8)
331: (8)
332: (8)
333: (8)
334: (8)
335: (8)
336: (8)
337: (8)
338: (8)
339: (8)
340: (8)
341: (8)
342: (8)
343: (8)
344: (8)
345: (8)
346: (8)
347: (8)
348: (8)
349: (8)
350: (8)
351: (8)
352: (8)
353: (8)
354: (8)
355: (8)
356: (8)
357: (8)
358: (8)
359: (8)
360: (8)
361: (8)
362: (8)
363: (8)
364: (8)
365: (8)
366: (8)
367: (8)
368: (8)
369: (8)
370: (8)
371: (8)
372: (8)
373: (8)
374: (8)
375: (8)
376: (8)
377: (8)
378: (8)
379: (8)
380: (8)
381: (8)
382: (8)
383: (8)
384: (8)
385: (8)
386: (8)
387: (8)
388: (8)
389: (8)
390: (8)
391: (8)
392: (8)
393: (8)
394: (8)
395: (8)
396: (8)
397: (8)
398: (8)
399: (8)
400: (8)
401: (8)
402: (8)
403: (8)
404: (8)
405: (8)
406: (8)
407: (8)
408: (8)
409: (8)
410: (8)
411: (8)
412: (8)
413: (8)
414: (8)
415: (8)
416: (8)
417: (8)
418: (8)
419: (8)
420: (8)
421: (8)
422: (8)
423: (8)
424: (8)
425: (8)
426: (8)
427: (8)
428: (8)
429: (8)
430: (8)
431: (8)
432: (8)
433: (8)
434: (8)
435: (8)
436: (8)
437: (8)
438: (8)
439: (8)
440: (8)
441: (8)
442: (8)
443: (8)
444: (8)
445: (8)
446: (8)
447: (8)
448: (8)
449: (8)
450: (8)
451: (8)
452: (8)
453: (8)
454: (8)
455: (8)
456: (8)
457: (8)
458: (8)
459: (8)
460: (8)
461: (8)
462: (8)
463: (8)
464: (8)
465: (8)
466: (8)
467: (8)
468: (8)
469: (8)
470: (8)
471: (8)
472: (8)
473: (8)
474: (8)
475: (8)
476: (8)
477: (8)
478: (8)
479: (8)
480: (8)
481: (8)
482: (8)
483: (8)
484: (8)
485: (8)
486: (8)
487: (8)
488: (8)
489: (8)
490: (8)
491: (8)
492: (8)
493: (8)
494: (8)
495: (8)
496: (8)
497: (8)
498: (8)
499: (8)
500: (8)
501: (8)
502: (8)
503: (8)
504: (8)
505: (8)
506: (8)
507: (8)
508: (8)
509: (8)
510: (8)
511: (8)
512: (8)
513: (8)
514: (8)
515: (8)
516: (8)
517: (8)
518: (8)
519: (8)
520: (8)
521: (8)
522: (8)
523: (8)
524: (8)
525: (8)
526: (8)
527: (8)
528: (8)
529: (8)
530: (8)
531: (8)
532: (8)
533: (8)
534: (8)
535: (8)
536: (8)
537: (8)
538: (8)
539: (8)
540: (8)
541: (8)
542: (8)
543: (8)
544: (8)
545: (8)
546: (8)
547: (8)
548: (8)
549: (8)
550: (8)
551: (8)
552: (8)
553: (8)
554: (8)
555: (8)
556: (8)
557: (8)
558: (8)
559: (8)
560: (8)
561: (8)
562: (8)
563: (8)
564: (8)
565: (8)
566: (8)
567: (8)
568: (8)
569: (8)
570: (8)
571: (8)
572: (8)
573: (8)
574: (8)
575: (8)
576: (8)
577: (8)
578: (8)
579: (8)
580: (8)
581: (8)
582: (8)
583: (8)
584: (8)
585: (8)
586: (8)
587: (8)
588: (8)
589: (8)
590: (8)
591: (8)
592: (8)
593: (8)
594: (8)
595: (8)
596: (8)
597: (8)
598: (8)
599: (8)
600: (8)
601: (8)
602: (8)
603: (8)
604: (8)
605: (8)
606: (8)
607: (8)
608: (8)
609: (8)
610: (8)
611: (8)
612: (8)
613: (8)
614: (8)
615: (8)
616: (8)
617: (8)
618: (8)
619: (8)
620: (8)
621: (8)
622: (8)
623: (8)
624: (8)
625: (8)
626: (8)
627: (8)
628: (8)
629: (8)
630: (8)
631: (8)
632: (8)
633: (8)
634: (8)
635: (8)
636: (8)
637: (8)
638: (8)
639: (8)
640: (8)
641: (8)
642: (8)
643: (8)
644: (8)
645: (8)
646: (8)
647: (8)
648: (8)
649: (8)
650: (8)
651: (8)
652: (8)
653: (8)
654: (8)
655: (8)
656: (8)
657: (8)
658: (8)
659: (8)
660: (8)
661: (8)
662: (8)
663: (8)
664: (8)
665: (8)
666: (8)
667: (8)
668: (8)
669: (8)
670: (8)
671: (8)
672: (8)
673: (8)
674: (8)
675: (8)
676: (8)
677: (8)
678: (8)
679: (8)
680: (8)
681: (8)
682: (8)
683: (8)
684: (8)
685: (8)
686: (8)
687: (8)
688: (8)
689: (8)
690: (8)
691: (8)
692: (8)
693: (8)
694: (8)
695: (8)
696: (8)
697: (8)
698: (8)
699: (8)
700: (8)
701: (8)
702: (8)
703: (8)
704: (8)
705: (8)
706: (8)
707: (8)
708: (8)
709: (8)
710: (8)
711: (8)
712: (8)
713: (8)
714: (8)
715: (8)
716: (8)
717: (8)
718: (8)
719: (8)
720: (8)
721: (8)
722: (8)
723: (8)
724: (8)
725: (8)
726: (8)
727: (8)
728: (8)
729: (8)
730: (8)
731: (8)
732: (8)
733: (8)
734: (8)
735: (8)
736: (8)
737: (8)
738: (8)
739: (8)
740: (8)
741: (8)
742: (8)
743: (8)
744: (8)
745: (8)
746: (8)
747: (8)
748: (8)
749: (8)
750: (8)
751: (8)
752: (8)
753: (8)
754: (8)
755: (8)
756: (8)
757: (8)
758: (8)
759: (8)
760: (8)
761: (8)
762: (8)
763: (8)
764: (8)
765: (8)
766: (8)
767: (8)
768: (8)
769: (8)
770: (8)
771: (8)
772: (8)
773: (8)
774: (8)
775: (8)
776: (8)
777: (8)
778: (8)
779: (8)
780: (8)
781: (8)
782: (8)
783: (8)
784: (8)
785: (8)
786: (8)
787: (8)
788: (8)
789: (8)
790: (8)
791: (8)
792: (8)
793: (8)
794: (8)
795: (8)
796: (8)
797: (8)
798: (8)
799: (8)
800: (8)
801: (8)
802: (8)
803: (8)
804: (8)
805: (8)
806: (8)
807: (8)
808: (8)
809: (8)
810: (8)
811: (8)
812: (8)
813: (8)
814: (8)
815: (8)
816: (8)
817: (8)
818: (8)
819: (8)
820: (8)
821: (8)
822: (8)
823: (8)
824: (8)
825: (8)
826: (8)
827: (8)
828: (8)
829: (8)
830: (8)
831: (8)
832: (8)
833: (8)
834: (8)
835: (8)
836: (8)
837: (8)
838: (8)
839: (8)
840: (8)
841: (8)
842: (8)
843: (8)
844: (8)
845: (8)
846: (8)
847: (8)
848: (8)
849: (8)
850: (8)
851: (8)
852: (8)
853: (8)
854: (8)
855: (8)
856: (8)
857: (8)
858: (8)
859: (8)
860: (8)
861: (8)
862: (8)
863: (8)
864: (8)
865: (8)
866: (8)
867: (8)
868: (8)
869: (8)
870: (8)
871: (8)
872: (8)
873: (8)
874: (8)
875: (8)
876: (8)
877: (8)
878: (8)
879: (8)
880: (8)
881: (8)
882: (8)
883: (8)
884: (8)
885: (8)
886: (8)
887: (8)
888: (8)
889: (8)
890: (8)
891: (8)
892: (8)
893: (8)
894: (8)
895: (8)
896: (8)
897: (8)
898: (8)
899: (8)
900: (8)
901: (8)
902: (8)
903: (8)
904: (8)
905: (8)
906: (8)
907: (8)
908: (8)
909: (8)
910: (8)
911: (8)
912: (8)
913: (8)
914: (8)
915: (8)
916: (8)
917: (8)
918: (8)
919: (8)
920: (8)
921: (8)
922: (8)
923: (8)
924: (8)
925: (8)
926: (8)
927: (8)
928: (8)
929: (8)
930: (8)
931: (8)
932: (8)
933: (8)
934: (8)
935: (8)
936: (8)
937: (8)
938: (8)
939: (8)
940: (8)
941: (8)
942: (8)
943: (8)
944: (8)
945: (8)
946: (8)
947: (8)
948: (8)
949: (8)
950: (8)
951: (8)
952: (8)
953: (8)
954: (8)
955: (8)
956: (8)
957: (8)
958: (8)
959: (8)
960: (8)
961: (8)
962: (8)
963: (8)
964: (8)
965: (8)
966: (8)
967: (8)
968: (8)
969: (8)
970: (8)
971: (8)
972: (8)
973: (8)
974: (8)
975: (8)
976: (8)
977: (8)
978: (8)
979: (8)
980: (8)
981: (8)
982: (8)
983: (8)
984: (8)
985: (8)
986: (8)
987: (8)
988: (8)
989: (8)
990: (8)
991: (8)
992: (8)
993: (8)
994: (8)
995: (8)
996: (8)
997: (8)
998: (8)
999: (8)
1000: (8)
1001: (8)
1002: (8)
1003: (8)
1004: (8)
1005: (8)
1006: (8)
1007: (8)
1008: (8)
1009: (8)
1010: (8)
1011: (8)
1012: (8)
1013: (8)
1014: (8)
1015: (8)
1016: (8)
1017: (8)
1018: (8)
1019: (8)
1020: (8)
1021: (8)
1022: (8)
1023: (8)
1024: (8)
1025: (8)
1026: (8)
1027: (8)
1028: (8)
1029: (8)
1030: (8)
1031: (8)
1032: (8)
1033: (8)
1034: (8)
1035: (8)
1036: (8)
1037: (8)
1038: (8)
1039: (8)
1040: (8)
1041: (8)
1042: (8)
1043: (8)
1044: (8)
1045: (8)
1046: (8)
1047: (8)
1048: (8)
1049: (8)
1050: (8)
1051: (8)
1052: (8)
1053: (8)
1054: (8)
1055: (8)
1056: (8)
1057: (8)
1058: (8)
1059: (8)
1060: (8)
1061: (8)
1062: (8)
1063: (8)
1064: (8)
1065: (8)
1066: (8)
1067: (8)
1068: (8)
1069: (8)
1070: (8)
1071: (8)
1072: (8)
1073: (8)
1074: (8)
1075: (8)
1076: (8)
1077: (8)
1078: (8)
1079: (8)
1080: (8)
1081: (8)
1082: (8)
1083: (8)
1084: (8)
1085: (8)
1086: (8)
1087: (8)
1088: (8)
1089: (8)
1090: (8)
1091: (8)
1092: (8)
1093: (8)
1094: (8)
1095: (8)
1096: (8)
1097: (8)
1098: (8)
1099: (8)
1100: (8)
1101: (8)
1102: (8)
1103: (8)
1104: (8)
1105: (8)
1106: (8)
1107: (8)
1108: (8)
1109: (8)
1110: (8)
1111: (8)
1112: (8)
1113: (8)
1114: (8)
1115: (8)
1116: (8)
1117: (8)
1118: (8)
1119: (8)
1120: (8)
1121: (8)
1122: (8)
1123: (8)
1124: (8)
1125: (8)
1126: (8)
1127: (8)
1128: (8)
1129: (8)
1130: (8)
1131: (8)
1132: (8)
1133: (8)
1134: (8)
1135: (8)
1136: (8)
1137: (8)
1138: (8)
1139: (8)
1140: (8)
1141: (8)
1142: (8)
1143: (8)
1144: (8)
1145: (8)
1146: (8)
1147: (8)
1148: (8)
1149: (8)
1150: (8)
1151: (8)
1152: (8)
1153: (8)
1154: (8)
1155: (8)
1156: (8)
1157: (8)
1158: (8)
1159: (8)
1160: (8)
1161: (8)
1162: (8)
1163: (8)
1164: (8)
1165: (8)
1166: (8)
1167: (8)
1168: (8)
1169: (8)
1170: (8)
1171: (8)
1172: (8)
1173: (8)
1174: (8)
1175: (8)
1176: (8)
1177: (8)
1178: (8)
1179: (8)
1180: (8)
1181: (8)
1182: (8)
1183: (8)
1184: (8)
1185: (8)
1186: (8)
1187: (8)
1188: (8)
1189: (8)
1190: (8)
1191: (8)
1192: (8)
1193: (8)
1194: (8)
1195: (8)
1196: (8)
1197: (8)
1198: (8)
1199: (8)
1200: (8)
1201: (8)
1202: (8)
1203: (8)
1204: (8)
1205: (8)
1206: (8)
1207: (8)
1208: (8)
1209: (8)
1210: (8)
1211: (8)
1212: (8)
1213: (8)
1214: (8)
1215: (8)
1216: (8)
1217: (8)
1218: (8)
1219: (8)
1220: (8)
1221: (8)
1222: (8)
1223: (8)
1224: (8)
1225: (8)
1226: (8)
1227: (8)
1228: (8)
1229: (8)
1230: (8)
1231: (8)
1232: (8)
1233: (8)
1234: (8)
1235: (8)
1236: (8)
1237: (8)
1238: (8)
1239: (8)
1240: (8)
1241: (8)
1242: (8)
1243: (8)
1244: (8)
1245: (8)
1246: (8)
1247: (8)
1248: (8)
1249: (8)
1250: (8)
1251: (8)
1252: (8)
1253: (8)
1254: (8)
1255: (8)
1256: (8)
1257: (8)
1258: (8)
1259: (8)
1260: (8)
1261: (8)
1262: (8)
1263: (8)
1264: (8)
1265: (8)
1266: (8)
1267: (8)
1268: (8)
1269: (8)
1270: (8)
1271: (8)
1272: (8)
1273: (8)
1274: (8)
1275: (8)
1276: (8)
1277: (8)
1278: (8)
1279: (8)
1280: (8)
1281: (8)
1282: (8)
1283: (8)
1284: (8)
1285: (8)
1286: (8)
1287: (8)
1288: (8)
1289: (8)
1290: (8)
1291: (8)
1292: (8)
1293: (8)
1294: (8)
1295: (8)
1296: (8)
1297: (8)
1298: (8)
1299: (8)
1300: (8)
1301: (8)
1302: (8)
1303: (8)
1304: (8)
1305: (8)
1306: (8)
1307: (8)
1308: (8)
1309: (8)
1310: (8)
1311: (8)
1312: (8)
1313: (8)
1314: (8)
1315: (8)
1316: (8)
1317: (8)
1318: (8)
1319: (8)
1320: (8)
1321: (8)
1322: (8)
1323: (8)
1324: (8)
1325: (8)
1326: (8)
1327: (8)
1328: (8)
1329: (8)
1330: (8)
1331: (8)
1332: (8)
1333: (8)
1334: (8)
1335: (8)
1336: (8)
1337: (8)
1338: (8)
1339: (8)
1340: (8)
1341: (8)
1342: (8)
1343: (8)
1344: (8)
1345: (8)
1346: (8)
1347: (8)
1348: (8)
1349: (8)
1350: (8)
1351: (8)
1352: (8)
1353: (8)
1354: (8)
1355: (8)
1356: (8)
1357: (8)
1358: (8)
1359: (8)
1360: (8)
1361: (8)
1362: (8)
1363: (8)
1364: (8)
1365: (8)
1366: (8)
1367: (8)
1368: (8)
1369: (8)
1370: (8)
1371: (8)
1372: (8)
1373: (8)
1374: (8)
1375: (8)
1376: (8)
1377: (8)
1378: (8)
1379: (8)
1380: (8)
1381: (8)
1382: (8)
1383: (8)
1384: (8)
1385: (8)
1386: (8)
1387: (8)
1388: (8)
1389: (8)
1390: (8)
1391: (8)
1392: (8)
1393: (8)
1394: (8)
1395: (8)
1396: (8)
1397: (8)
1398: (8)
1399: (8)
1400: (8)
1401: (8)
1402: (8)
1403: (8)
1404: (8)
1405: (8)
1406: (8)
1407: (8)
1408: (8)
1409: (8)
1410: (8)
1411: (8)
1412: (8)
1413: (8)
1414: (8)
1415: (8)
1416: (8)
1417: (8)
1418: (8)
1419: (8)
1420: (8)
1421: (8)
1422: (8)
1423: (8)
1424: (8)
1425: (8)
1426: (8)
1427: (8)
1428: (8)
1429: (8)
1430: (8)
1431: (8)
1432: (8)
1433: (8)
1434: (8)
1435: (8)
1436: (8)
1437: (8)
1438: (8)
1439: (8)
1440: (8)
1441: (8)
1442: (8)
1443: (8)
1444: (8)
1445: (8)
1446: (8)
1447: (8)
1448: (8)
1449: (8)
1450: (8)
1451: (8)
1452: (8)
1453: (8)
1454: (8)
1455: (8)
1456: (8)
1457: (8)
1458: (8)
1459: (8)
1460: (8)
1461: (8)
1462: (8)
1463: (8)
1464: (8)
1465: (8)
1466: (8)
1467: (8)
1468: (8)
1469: (8)
1470: (8)
1471: (8)
1472: (8)
1473: (8)
1474: (8)
1475: (8)
1476: (8)
1477: (8)
1478: (8)
1479: (8)
1480: (8)
1481: (8)
1482: (8)
1483: (8)
1484: (8)
1485: (8)
1486: (8)
1487: (8)
1488: (8)
1489: (8)
1490: (8)
1491: (8)

```

```

108: (16)                     raise TypeError(f"Only {dtype_category} dtypes are allowed in
{op}")
109: (8)
110: (12)                 else:
111: (8)                     return NotImplemented
112: (8)                 res_dtype = _result_type(self.dtype, other.dtype)
113: (12)                 if op.startswith("__i"):
114: (16)                     if res_dtype != self.dtype:
115: (20)                         raise TypeError(
116: (24)                             f"Cannot perform {op} with dtypes {self.dtype} and
117: (28)                             {other.dtype}")
118: (8)
119: (4)             def _promote_scalar(self, scalar):
120: (8)                 """
121: (8)                     Returns a promoted version of a Python scalar appropriate for use with
122: (8)                     operations on self.
123: (8)                     This may raise an OverflowError in cases where the scalar is an
124: (8)                     integer that is too large to fit in a NumPy integer dtype, or
125: (8)                     TypeError when the scalar type is incompatible with the dtype of self.
126: (8)                 """
127: (12)                 if isinstance(scalar, bool):
128: (16)                     if self.dtype not in _boolean_dtypes:
129: (20)                         raise TypeError(
130: (24)                             "Python bool scalars can only be promoted with bool
131: (28)                             arrays")
132: (8)
133: (12)                 elif isinstance(scalar, int):
134: (16)                     if self.dtype in _boolean_dtypes:
135: (20)                         raise TypeError(
136: (24)                             "Python int scalars cannot be promoted with bool arrays")
137: (8)
138: (12)                     if self.dtype in _integer_dtypes:
139: (16)                         info = np.iinfo(self.dtype)
140: (20)                         if not (info.min <= scalar <= info.max):
141: (24)                             raise OverflowError(
142: (28)                             "Python int scalars must be within the bounds of the
143: (32)                             dtype for integer arrays")
144: (8)
145: (12)                     elif isinstance(scalar, float):
146: (16)                         if self.dtype not in _floating_dtypes:
147: (20)                             raise TypeError(
148: (24)                             "Python float scalars can only be promoted with floating-
149: (28)                             point arrays.")
150: (8)
151: (12)                     elif isinstance(scalar, complex):
152: (16)                         if self.dtype not in _complex_floating_dtypes:
153: (20)                             raise TypeError(
154: (24)                             "Python complex scalars can only be promoted with complex
155: (28)                             floating-point arrays.")
156: (8)
157: (4)             else:
158: (8)                 raise TypeError("'scalar' must be a Python scalar")
159: (8)             return Array._new(np.array(scalar, self.dtype))
160: (4)         @staticmethod
161: (4)         def _normalize_two_args(x1, x2) -> Tuple[Array, Array]:
162: (8)             """
163: (8)                 Normalize inputs to two arg functions to fix type promotion rules
164: (8)                 NumPy deviates from the spec type promotion rules in cases where one
165: (8)                 argument is 0-dimensional and the other is not. For example:
166: (8)                 >>> import numpy as np
167: (8)                 >>> a = np.array([1.0], dtype=np.float32)
168: (8)                 >>> b = np.array(1.0, dtype=np.float64)
169: (8)                 >>> np.add(a, b) # The spec says this should be float64
170: (8)                             array([2.], dtype=float32)
171: (8)                 To fix this, we add a dimension to the 0-dimension array before
172: (8)                 passing it
173: (8)                 through. This works because a dimension would be added anyway from
174: (8)                 broadcasting, so the resulting shape is the same, but this prevents

```

```

169: (8)         from not promoting the dtype.
170: (8)
171: (8)
172: (12)        """
173: (8)        if x1.ndim == 0 and x2.ndim != 0:
174: (12)            x1 = Array._new(x1._array[None])
175: (8)        elif x2.ndim == 0 and x1.ndim != 0:
176: (12)            x2 = Array._new(x2._array[None])
177: (8)        return (x1, x2)
178: (4)    def _validate_index(self, key):
179: (8)        """
180: (8)        Validate an index according to the array API.
181: (8)        The array API specification only requires a subset of indices that are
182: (8)        supported by NumPy. This function will reject any index that is
183: (8)        allowed by NumPy but not required by the array API specification. We
184: (8)        always raise ``IndexError`` on such indices (the spec does not require
185: (8)        any specific behavior on them, but this makes the NumPy array API
186: (8)        namespace a minimal implementation of the spec). See
187: (8)        https://data-apis.org/array-api/latest/API\_specification/indexing.html
188: (8)        for the full list of required indexing behavior
189: (8)        This function raises IndexError if the index ``key`` is invalid. It
190: (8)        only raises ``IndexError`` on indices that are not already rejected by
191: (8)        NumPy, as NumPy will already raise the appropriate error on such
192: (8)        indices. ``shape`` may be None, in which case, only cases that are
193: (8)        independent of the array shape are checked.
194: (8)        The following cases are allowed by NumPy, but not specified by the
195: (10)       array
196: (10)       API specification:
197: (8)
198: (10)       - Indices to not include an implicit ellipsis at the end. That is,
199: (10)       every axis of an array must be explicitly indexed or an ellipsis
200: (10)       included. This behaviour is sometimes referred to as flat indexing.
201: (10)       - The start and stop of a slice may not be out of bounds. In
202: (10)       particular, for a slice ``i:j:k`` on an axis of size ``n``, only the
203: (10)       following are allowed:
204: (8)           - ``i`` or ``j`` omitted (``None``).
205: (8)           - ``-n <= i <= max(0, n - 1)``.
206: (8)           - For ``k > 0`` or ``k`` omitted (``None``), ``-n <= j <= n``.
207: (8)           - For ``k < 0``, ``-n - 1 <= j <= max(0, n - 1)``.
208: (8)       - Boolean array indices are not allowed as part of a larger tuple
209: (8)       index.
210: (8)       - Integer array indices are not allowed (with the exception of 0-D
211: (8)       arrays, which are treated the same as scalars).
212: (8)
213: (8)       Additionally, it should be noted that indices that would return a
214: (8)       scalar in NumPy will return a 0-D array. Array scalars are not allowed
215: (12)       in the specification, only 0-D arrays. This is done in the
216: (16)       ``Array._new`` constructor, not this function.
217: (16)
218: (16)
219: (16)
220: (16)
221: (16)
222: (12)
223: (16)
224: (20)
225: (20)
226: (20)
227: (20)
228: (16)
229: (8)
230: (8)
231: (8)
232: (8)
233: (8)
234: (12)
235: (16)
236: (16)
237: (16)
238: (16)
239: (16)
240: (16)
241: (16)
242: (16)
243: (16)
244: (16)
245: (16)
246: (16)
247: (16)
248: (16)
249: (16)
250: (16)
251: (16)
252: (16)
253: (16)
254: (16)
255: (16)
256: (16)
257: (16)
258: (16)
259: (16)
260: (16)
261: (16)
262: (16)
263: (16)
264: (16)
265: (16)
266: (16)
267: (16)
268: (16)
269: (16)
270: (16)
271: (16)
272: (16)
273: (16)
274: (16)
275: (16)
276: (16)
277: (16)
278: (16)
279: (16)
280: (16)
281: (16)
282: (16)
283: (16)
284: (16)
285: (16)
286: (16)
287: (16)
288: (16)
289: (16)
290: (16)
291: (16)
292: (16)
293: (16)
294: (16)
295: (16)
296: (16)
297: (16)
298: (16)
299: (16)
300: (16)
301: (16)
302: (16)
303: (16)
304: (16)
305: (16)
306: (16)
307: (16)
308: (16)
309: (16)
310: (16)
311: (16)
312: (16)
313: (16)
314: (16)
315: (16)
316: (16)
317: (16)
318: (16)
319: (16)
320: (16)
321: (16)
322: (16)
323: (16)
324: (16)
325: (16)
326: (16)
327: (16)
328: (16)
329: (16)
330: (16)
331: (16)
332: (16)
333: (16)
334: (16)
335: (16)
336: (16)
337: (16)
338: (16)
339: (16)
340: (16)
341: (16)
342: (16)
343: (16)
344: (16)
345: (16)
346: (16)
347: (16)
348: (16)
349: (16)
350: (16)
351: (16)
352: (16)
353: (16)
354: (16)
355: (16)
356: (16)
357: (16)
358: (16)
359: (16)
360: (16)
361: (16)
362: (16)
363: (16)
364: (16)
365: (16)
366: (16)
367: (16)
368: (16)
369: (16)
370: (16)
371: (16)
372: (16)
373: (16)
374: (16)
375: (16)
376: (16)
377: (16)
378: (16)
379: (16)
380: (16)
381: (16)
382: (16)
383: (16)
384: (16)
385: (16)
386: (16)
387: (16)
388: (16)
389: (16)
390: (16)
391: (16)
392: (16)
393: (16)
394: (16)
395: (16)
396: (16)
397: (16)
398: (16)
399: (16)
400: (16)
401: (16)
402: (16)
403: (16)
404: (16)
405: (16)
406: (16)
407: (16)
408: (16)
409: (16)
410: (16)
411: (16)
412: (16)
413: (16)
414: (16)
415: (16)
416: (16)
417: (16)
418: (16)
419: (16)
420: (16)
421: (16)
422: (16)
423: (16)
424: (16)
425: (16)
426: (16)
427: (16)
428: (16)
429: (16)
430: (16)
431: (16)
432: (16)
433: (16)
434: (16)
435: (16)
436: (16)
437: (16)
438: (16)
439: (16)
440: (16)
441: (16)
442: (16)
443: (16)
444: (16)
445: (16)
446: (16)
447: (16)
448: (16)
449: (16)
450: (16)
451: (16)
452: (16)
453: (16)
454: (16)
455: (16)
456: (16)
457: (16)
458: (16)
459: (16)
460: (16)
461: (16)
462: (16)
463: (16)
464: (16)
465: (16)
466: (16)
467: (16)
468: (16)
469: (16)
470: (16)
471: (16)
472: (16)
473: (16)
474: (16)
475: (16)
476: (16)
477: (16)
478: (16)
479: (16)
480: (16)
481: (16)
482: (16)
483: (16)
484: (16)
485: (16)
486: (16)
487: (16)
488: (16)
489: (16)
490: (16)
491: (16)
492: (16)
493: (16)
494: (16)
495: (16)
496: (16)
497: (16)
498: (16)
499: (16)
500: (16)
501: (16)
502: (16)
503: (16)
504: (16)
505: (16)
506: (16)
507: (16)
508: (16)
509: (16)
510: (16)
511: (16)
512: (16)
513: (16)
514: (16)
515: (16)
516: (16)
517: (16)
518: (16)
519: (16)
520: (16)
521: (16)
522: (16)
523: (16)
524: (16)
525: (16)
526: (16)
527: (16)
528: (16)
529: (16)
530: (16)
531: (16)
532: (16)
533: (16)
534: (16)
535: (16)
536: (16)
537: (16)
538: (16)
539: (16)
540: (16)
541: (16)
542: (16)
543: (16)
544: (16)
545: (16)
546: (16)
547: (16)
548: (16)
549: (16)
550: (16)
551: (16)
552: (16)
553: (16)
554: (16)
555: (16)
556: (16)
557: (16)
558: (16)
559: (16)
560: (16)
561: (16)
562: (16)
563: (16)
564: (16)
565: (16)
566: (16)
567: (16)
568: (16)
569: (16)
570: (16)
571: (16)
572: (16)
573: (16)
574: (16)
575: (16)
576: (16)
577: (16)
578: (16)
579: (16)
580: (16)
581: (16)
582: (16)
583: (16)
584: (16)
585: (16)
586: (16)
587: (16)
588: (16)
589: (16)
590: (16)
591: (16)
592: (16)
593: (16)
594: (16)
595: (16)
596: (16)
597: (16)
598: (16)
599: (16)
600: (16)
601: (16)
602: (16)
603: (16)
604: (16)
605: (16)
606: (16)
607: (16)
608: (16)
609: (16)
610: (16)
611: (16)
612: (16)
613: (16)
614: (16)
615: (16)
616: (16)
617: (16)
618: (16)
619: (16)
620: (16)
621: (16)
622: (16)
623: (16)
624: (16)
625: (16)
626: (16)
627: (16)
628: (16)
629: (16)
630: (16)
631: (16)
632: (16)
633: (16)
634: (16)
635: (16)
636: (16)
637: (16)
638: (16)
639: (16)
640: (16)
641: (16)
642: (16)
643: (16)
644: (16)
645: (16)
646: (16)
647: (16)
648: (16)
649: (16)
650: (16)
651: (16)
652: (16)
653: (16)
654: (16)
655: (16)
656: (16)
657: (16)
658: (16)
659: (16)
660: (16)
661: (16)
662: (16)
663: (16)
664: (16)
665: (16)
666: (16)
667: (16)
668: (16)
669: (16)
670: (16)
671: (16)
672: (16)
673: (16)
674: (16)
675: (16)
676: (16)
677: (16)
678: (16)
679: (16)
680: (16)
681: (16)
682: (16)
683: (16)
684: (16)
685: (16)
686: (16)
687: (16)
688: (16)
689: (16)
690: (16)
691: (16)
692: (16)
693: (16)
694: (16)
695: (16)
696: (16)
697: (16)
698: (16)
699: (16)
700: (16)
701: (16)
702: (16)
703: (16)
704: (16)
705: (16)
706: (16)
707: (16)
708: (16)
709: (16)
710: (16)
711: (16)
712: (16)
713: (16)
714: (16)
715: (16)
716: (16)
717: (16)
718: (16)
719: (16)
720: (16)
721: (16)
722: (16)
723: (16)
724: (16)
725: (16)
726: (16)
727: (16)
728: (16)
729: (16)
730: (16)
731: (16)
732: (16)
733: (16)
734: (16)
735: (16)
736: (16)
737: (16)
738: (16)
739: (16)
740: (16)
741: (16)
742: (16)
743: (16)
744: (16)
745: (16)
746: (16)
747: (16)
748: (16)
749: (16)
750: (16)
751: (16)
752: (16)
753: (16)
754: (16)
755: (16)
756: (16)
757: (16)
758: (16)
759: (16)
760: (16)
761: (16)
762: (16)
763: (16)
764: (16)
765: (16)
766: (16)
767: (16)
768: (16)
769: (16)
770: (16)
771: (16)
772: (16)
773: (16)
774: (16)
775: (16)
776: (16)
777: (16)
778: (16)
779: (16)
780: (16)
781: (16)
782: (16)
783: (16)
784: (16)
785: (16)
786: (16)
787: (16)
788: (16)
789: (16)
790: (16)
791: (16)
792: (16)
793: (16)
794: (16)
795: (16)
796: (16)
797: (16)
798: (16)
799: (16)
800: (16)
801: (16)
802: (16)
803: (16)
804: (16)
805: (16)
806: (16)
807: (16)
808: (16)
809: (16)
810: (16)
811: (16)
812: (16)
813: (16)
814: (16)
815: (16)
816: (16)
817: (16)
818: (16)
819: (16)
820: (16)
821: (16)
822: (16)
823: (16)
824: (16)
825: (16)
826: (16)
827: (16)
828: (16)
829: (16)
830: (16)
831: (16)
832: (16)
833: (16)
834: (16)
835: (16)
836: (16)
837: (16)
838: (16)
839: (16)
840: (16)
841: (16)
842: (16)
843: (16)
844: (16)
845: (16)
846: (16)
847: (16)
848: (16)
849: (16)
850: (16)
851: (16)
852: (16)
853: (16)
854: (16)
855: (16)
856: (16)
857: (16)
858: (16)
859: (16)
860: (16)
861: (16)
862: (16)
863: (16)
864: (16)
865: (16)
866: (16)
867: (16)
868: (16)
869: (16)
870: (16)
871: (16)
872: (16)
873: (16)
874: (16)
875: (16)
876: (16)
877: (16)
878: (16)
879: (16)
880: (16)
881: (16)
882: (16)
883: (16)
884: (16)
885: (16)
886: (16)
887: (16)
888: (16)
889: (16)
890: (16)
891: (16)
892: (16)
893: (16)
894: (16)
895: (16)
896: (16)
897: (16)
898: (16)
899: (16)
900: (16)
901: (16)
902: (16)
903: (16)
904: (16)
905: (16)
906: (16)
907: (16)
908: (16)
909: (16)
910: (16)
911: (16)
912: (16)
913: (16)
914: (16)
915: (16)
916: (16)
917: (16)
918: (16)
919: (16)
920: (16)
921: (16)
922: (16)
923: (16)
924: (16)
925: (16)
926: (16)
927: (16)
928: (16)
929: (16)
930: (16)
931: (16)
932: (16)
933: (16)
934: (16)
935: (16)
936: (16)
937: (16)
938: (16)
939: (16)
940: (16)
941: (16)
942: (16)
943: (16)
944: (16)
945: (16)
946: (16)
947: (16)
948: (16)
949: (16)
950: (16)
951: (16)
952: (16)
953: (16)
954: (16)
955: (16)
956: (16)
957: (16)
958: (16)
959: (16)
960: (16)
961: (16)
962: (16)
963: (16)
964: (16)
965: (16)
966: (16)
967: (16)
968: (16)
969: (16)
970: (16)
971: (16)
972: (16)
973: (16)
974: (16)
975: (16)
976: (16)
977: (16)
978: (16)
979: (16)
980: (16)
981: (16)
982: (16)
983: (16)
984: (16)
985: (16)
986: (16)
987: (16)
988: (16)
989: (16)
990: (16)
991: (16)
992: (16)
993: (16)
994: (16)
995: (16)
996: (16)
997: (16)
998: (16)
999: (16)
1000: (16)
1001: (16)
1002: (16)
1003: (16)
1004: (16)
1005: (16)
1006: (16)
1007: (16)
1008: (16)
1009: (16)
1010: (16)
1011: (16)
1012: (16)
1013: (16)
1014: (16)
1015: (16)
1016: (16)
1017: (16)
1018: (16)
1019: (16)
1020: (16)
1021: (16)
1022: (16)
1023: (16)
1024: (16)
1025: (16)
1026: (16)
1027: (16)
1028: (16)
1029: (16)
1030: (16)
1031: (16)
1032: (16)
1033: (16)
1034: (16)
1035: (16)
1036: (16)
1037: (16)
1038: (16)
1039: (16)
1040: (16)
1041: (16)
1042: (16)
1043: (16)
1044: (16)
1045: (16)
1046: (16)
1047: (16)
1048: (16)
1049: (16)
1050: (16)
1051: (16)
1052: (16)
1053: (16)
1054: (16)
1055: (16)
1056: (16)
1057: (16)
1058: (16)
1059: (16)
1060: (16)
1061: (16)
1062: (16)
1063: (16)
1064: (16)
1065: (16)
1066: (16)
1067: (16)
1068: (16)
1069: (16)
1070: (16)
1071: (16)
1072: (16)
1073: (16)
1074: (16)
1075: (16)
1076: (16)
1077: (16)
1078: (16)
1079: (16)
1080: (16)
1081: (16)
1082: (16)
1083: (16)
1084: (16)
1085: (16)
1086: (16)
1087: (16)
1088: (16)
1089: (16)
1090: (16)
1091: (16)
1092: (16)
1093: (16)
1094: (16)
1095: (16)
1096: (16)
1097: (16)
1098: (16)
1099: (16)
1100: (16)
1101: (16)
1102: (16)
1103: (16)
1104: (16)
1105: (16)
1106: (16)
1107: (16)
1108: (16)
1109: (16)
1110: (16)
1111: (16)
1112: (16)
1113: (16)
1114: (16)
1115: (16)
1116: (16)
1117: (16)
1118: (16)
1119: (16)
1120: (16)
1121: (16)
1122: (16)
1123: (16)
1124: (16)
1125: (16)
1126: (16)
1127: (16)
1128: (16)
1129: (16)
1130: (16)
1131: (16)
1132: (16)
1133: (16)
1134: (16)
1135: (16)
1136: (16)
1137: (16)
1138: (16)
1139: (16)
1140: (16)
1141: (16)
1142: (16)
1143: (16)
1144: (16)
1145: (16)
1146: (16)
1147: (16)
1148: (16)
1149: (16)
1150: (16)
1151: (16)
1152: (16)
1153: (16)
1154: (16)
1155: (16)
1156: (16)
1157: (16)
1158: (16)
1159: (16)
1160: (16)
1161: (16)
1162: (16)
1163: (16)
1164: (16)
1165: (16)
1166: (16)
1167: (16)
1168: (16)
1169: (16)
1170: (16)
1171: (16)
1172: (16)
1173: (16)
1174: (16)
1175: (16)
1176: (16)
1177: (16)
1178: (16)
1179: (16)
1180: (16)
1181: (16)
1182: (16)
1183: (16)
1184: (16)
1185: (16)
1186: (16)
1187: (16)
1188: (16)
1189: (16)
1190: (16)
1191: (16)
1192: (16)
1193: (16)
1194: (16)
1195: (16)
1196: (16)
1197: (16)
1198: (16)
1199: (16)
1200: (16)
1201: (16)
1202: (16)
1203: (16)
1204: (16)
1205: (16)
1206: (16)
1207: (16)
1208: (16)
1209: (16)
1210: (16)
1211: (16)
1212: (16)
1213: (16)
1214: (16)
1215: (16)
1216: (16)
1217: (16)
1218: (16)
1219: (16)
1220: (16)
1221: (16)
1222: (16)
1223: (16)
1224: (16)
1225: (16)
1226: (16)
1227: (16)
1228: (16)
1229: (16)
1230: (16)
1231: (16)
1232: (16)
1233: (16)
1234: (16)
1235: (16)
1236: (16)
1237: (16)
1238: (16)
1239: (16)
1240: (16)
1241: (16)
1242: (16)
1243: (16)
1244: (16)
1245: (16)
1246: (16)
1247: (16)
1248: (16)
1249: (16)
1250: (16)
1251: (16)
1252: (16)
1253: (16)
1254: (16)
1255: (16)
1256: (16)
1257: (16)
1258: (16)
1259: (16)
1260: (16)
1261: (16)
1262: (16)
1263: (16)
1264: (16)
1265: (16)
1266: (16)
1267: (16)
1268: (16)
1269: (16)
1270: (16)
1271: (16)
1272: (16)
1273: (16)
1274: (16)
1275: (16)
1276: (16)
1277: (16)
1278: (16)
1279: (16)
1280: (16)
1281: (16)
1282: (16)
1283: (16)
1284: (16)
1285: (16)
1286: (16)
1287: (16)
1288: (16)
1289: (16)
1290: (16)
1291: (16)
1292: (16)
1293: (16)
1294: (16)
1295: (16)
1296: (16)
1297: (16)
1298: (16)
1299: (16)
1300: (16)
1301: (16)
1302: (16)
1303: (16)
1304: (16)
1305: (16)
1306: (16)
1307: (16)
1308: (16)
1309: (16)
1310: (16)
1311: (16)
1312: (16)
1313: (16)
1314: (16)
1315: (16)
1316: (16)
1317: (16)
1318: (16)
1319: (16)
1320: (16)
1321: (16)
1322: (16)
1323: (16)
1324: (16)
1325: (16)
1326: (16)
1327: (16)
1328: (16)
1329: (16)
1330: (16)
1331: (16)
1332: (16)
1333: (16)
1334: (16)
1335: (16)
1336: (16)
1337: (16)
1338: (16)
1339: (16)
1340: (16)
1341: (16)
1342: (16)
1343: (16)
1344: (16)
1345: (16)
1346: (16)
1347: (16)
1348: (16)
1349: (16)
1350: (16)
1351: (16)
1352: (16)
1353: (16)
1354: (16)
1355: (16)
1356: (16)
1357: (16)
1358: (16)
1359: (16)
1360: (16)
1361: (16)
1362: (16)
1363: (16)
1364: (16)
1365: (16)
1366: (16)
1367: (16)
1368: (16)
1369: (16)
1370: (16)
1371: (16)
1372: (16)
1373: (16)
1374: (16)
1375: (16)
1376: (16)
1377: (16)
1378: (16)
1379: (16)
1380: (16)
1381: (16)
1382: (16)
1383: (16)
1384: (16)
1385: (16)
1386: (16)
1387: (16)
1388: (16)
1389: (16)
1390: (16)
1391: (16)
1392: (16)
1393: (16)
1394: (16)
1395: (16)
1396: (16)
1397: (16)
1398: (16)
1399: (16)
1400: (16)
1401: (16)
1402: (16)
1403: (16)
1404: (16)
1405: (16)
1406: (16)
1407: (16)
1408: (16)
1409: (16)
1410: (16)
1411: (16)
1412: (16)
1413: (16)
1414: (16)
1415: (16)
1416: (16)
1417: (16)
1418: (16)
1419: (16)
1420: (16)
1421: (16)
142
```

```

237: (20)
238: (24)
239: (20)
240: (16)
241: (20)
242: (24)
243: (20)
244: (24)
245: (8)
246: (8)
247: (12)
248: (8)
249: (12)
250: (16)
251: (20)
252: (20)
253: (20)
"
254: (20)
"
255: (20)
256: (20)
257: (16)
258: (8)
259: (12)
260: (8)
261: (12)
262: (12)
263: (16)
264: (20)
265: (24)
266: (24)
267: (12)
268: (12)
269: (12)
270: (16)
271: (12)
272: (8)
273: (12)
274: (16)
275: (20)
276: (16)
277: (20)
278: (16)
279: (20)
280: (24)
281: (20)
282: (24)
283: (20)
284: (24)
285: (28)
286: (32)
287: (32)
288: (32)
289: (32)
290: (28)
291: (16)
292: (20)
293: (24)
294: (20)
295: (24)
296: (20)
297: (24)
298: (28)
299: (32)
300: (32)
301: (32)
302: (32)
303: (28)

                if i.dtype in _boolean_dtypes:
                    key_has_mask = True
                    single_axes.append(i)
                else:
                    if i == Ellipsis:
                        n_ellipsis += 1
                    else:
                        single_axes.append(i)
            n_single_axes = len(single_axes)
            if n_ellipsis > 1:
                return # handled by ndarray
            elif n_ellipsis == 0:
                if not key_has_mask and n_single_axes < self.ndim:
                    raise IndexError(
                        f"{self.ndim}, but the multi-axes index only specifies "
                        f"{n_single_axes} dimensions. If this was intentional, "
                        "add a trailing ellipsis (...) which expands into as many "
                        "slices () as necessary - this is what np.ndarray arrays "
                        "implicitly do, but such flat indexing behaviour is not "
                        "specified in the Array API."
                    )
            if n_ellipsis == 0:
                indexed_shape = self.shape
            else:
                ellipsis_start = None
                for pos, i in enumerate(nonexpanding_key):
                    if not (isinstance(i, Array) or isinstance(i, np.ndarray)):
                        if i == Ellipsis:
                            ellipsis_start = pos
                            break
                assert ellipsis_start is not None # sanity check
                ellipsis_end = self.ndim - (n_single_axes - ellipsis_start)
                indexed_shape = (
                    self.shape[:ellipsis_start] + self.shape[ellipsis_end:]
                )
            for i, side in zip(single_axes, indexed_shape):
                if isinstance(i, slice):
                    if side == 0:
                        f_range = "0 (or None)"
                    else:
                        f_range = f"between -{side} and {side - 1} (or None)"
                    if i.start is not None:
                        try:
                            start = operator.index(i.start)
                        except TypeError:
                            pass # handled by ndarray
                    else:
                        if not (-side <= start <= side):
                            raise IndexError(
                                f"Slice {i} contains {start=}, but should be "
                                f"{f_range} for an axis of size {side} "
                                "(out-of-bounds starts are not specified in "
                                "the Array API)"
                            )
                    if i.stop is not None:
                        try:
                            stop = operator.index(i.stop)
                        except TypeError:
                            pass # handled by ndarray
                    else:
                        if not (-side <= stop <= side):
                            raise IndexError(
                                f"Slice {i} contains {stop=}, but should be "
                                f"{f_range} for an axis of size {side} "
                                "(out-of-bounds stops are not specified in "
                                "the Array API)"
                            )

```

```

304: (12)
305: (16)
306: (20)
307: (20)
308: (24)
309: (24)
310: (24)
311: (20)
312: (16)
313: (20)
314: (24)
315: (24)
316: (24)
317: (20)
318: (12)
319: (16)
320: (20)
321: (20)
322: (16)
323: (4)
324: (8)
325: (8)
326: (8)
327: (8)
328: (12)
329: (8)
330: (8)
331: (4)
332: (8)
333: (8)
334: (8)
335: (8)
336: (8)
337: (12)
338: (8)
339: (8)
340: (8)
341: (4)
342: (8)
343: (8)
344: (8)
345: (8)
"__and__")
346: (8)
347: (12)
348: (8)
349: (8)
350: (8)
351: (4)
352: (8)
353: (4)
354: (8)
355: (12)
{api_version!r}")
356: (8)
357: (4)
358: (8)
359: (8)
360: (8)
361: (8)
362: (12)
dimensions")
363: (8)
364: (8)
365: (4)
366: (8)
367: (8)
368: (8)
369: (8)

        elif isinstance(i, Array):
            if i.dtype in _boolean_dtypes and len(_key) != 1:
                assert isinstance(key, tuple) # sanity check
                raise IndexError(
                    f"Single-axes index {i} is a boolean array and "
                    f"{len(key)}), but masking is only specified in the "
                    "Array API when the array is the sole index."
                )
            elif i.dtype in _integer_dtypes and i.ndim != 0:
                raise IndexError(
                    f"Single-axes index {i} is a non-zero-dimensional "
                    "integer array, but advanced integer indexing is not "
                    "specified in the Array API."
                )
        elif isinstance(i, tuple):
            raise IndexError(
                f"Single-axes index {i} is a tuple, but nested tuple "
                "indices are not specified in the Array API."
            )
    )

def __abs__(self: Array, /) -> Array:
    """
    Performs the operation __abs__.
    """
    if self.dtype not in _numeric_dtypes:
        raise TypeError("Only numeric dtypes are allowed in __abs__")
    res = self._array.__abs__()
    return self.__class__.new(res)

def __add__(self: Array, other: Union[int, float, Array], /) -> Array:
    """
    Performs the operation __add__.
    """
    other = self._check_allowed_dtypes(other, "numeric", "__add__")
    if other is NotImplemented:
        return other
    self, other = self._normalize_two_args(self, other)
    res = self._array.__add__(other._array)
    return self.__class__.new(res)

def __and__(self: Array, other: Union[int, bool, Array], /) -> Array:
    """
    Performs the operation __and__.
    """
    other = self._check_allowed_dtypes(other, "integer or boolean",
"__and__")
    if other is NotImplemented:
        return other
    self, other = self._normalize_two_args(self, other)
    res = self._array.__and__(other._array)
    return self.__class__.new(res)

def __array_namespace__(
    self: Array, /, *, api_version: Optional[str] = None
) -> types.ModuleType:
    if api_version is not None and not api_version.startswith("2021."):
        raise ValueError(f"Unrecognized array API version: {api_version!r}")
    return array_api

def __bool__(self: Array, /) -> bool:
    """
    Performs the operation __bool__.
    """
    if self._array.ndim != 0:
        raise TypeError("bool is only allowed on arrays with 0 "
dimensions")
    res = self._array.__bool__()
    return res

def __complex__(self: Array, /) -> complex:
    """
    Performs the operation __complex__.
    """
    if self._array.ndim != 0:

```

```

370: (12)                                raise TypeError("complex is only allowed on arrays with 0
dimensions")
371: (8)                                 dimensions")
372: (8)                                 dimensions")
373: (4)                                 def __dlpack__(self: Array, /, *, stream: None = None) -> PyCapsule:
374: (8)                                 """
375: (8)                                 Performs the operation __dlpack__.
376: (8)                                 """
377: (8)                                 return self._array.__dlpack__(stream=stream)
378: (4)                                 def __dlpack_device__(self: Array, /) -> Tuple[IntEnum, int]:
379: (8)                                 """
380: (8)                                 Performs the operation __dlpack_device__.
381: (8)                                 """
382: (8)                                 return self._array.__dlpack_device__()
383: (4)                                 def __eq__(self: Array, other: Union[int, float, bool, Array], /) ->
Array:
384: (8)                                 """
385: (8)                                 Performs the operation __eq__.
386: (8)                                 """
387: (8)                                 other = self._check_allowed_dtypes(other, "all", "__eq__")
388: (8)                                 if other is NotImplemented:
389: (12)                                     return other
390: (8)                                 self, other = self._normalize_two_args(self, other)
391: (8)                                 res = self._array.__eq__(other._array)
392: (8)                                 return self.__class__.new(res)
393: (4)                                 def __float__(self: Array, /) -> float:
394: (8)                                 """
395: (8)                                 Performs the operation __float__.
396: (8)                                 """
397: (8)                                 if self._array.ndim != 0:
398: (12)                                     raise TypeError("float is only allowed on arrays with 0
dimensions")
399: (8)                                 if self.dtype in _complex_floating_dtypes:
400: (12)                                     raise TypeError("float is not allowed on complex floating-point
arrays")
401: (8)                                 res = self._array.__float__()
402: (8)                                 return res
403: (4)                                 def __floordiv__(self: Array, other: Union[int, float, Array], /) ->
Array:
404: (8)                                 """
405: (8)                                 Performs the operation __floordiv__.
406: (8)                                 """
407: (8)                                 other = self._check_allowed_dtypes(other, "real numeric",
"__floordiv__")
408: (8)                                 if other is NotImplemented:
409: (12)                                     return other
410: (8)                                 self, other = self._normalize_two_args(self, other)
411: (8)                                 res = self._array.__floordiv__(other._array)
412: (8)                                 return self.__class__.new(res)
413: (4)                                 def __ge__(self: Array, other: Union[int, float, Array], /) -> Array:
414: (8)                                 """
415: (8)                                 Performs the operation __ge__.
416: (8)                                 """
417: (8)                                 other = self._check_allowed_dtypes(other, "real numeric", "__ge__")
418: (8)                                 if other is NotImplemented:
419: (12)                                     return other
420: (8)                                 self, other = self._normalize_two_args(self, other)
421: (8)                                 res = self._array.__ge__(other._array)
422: (8)                                 return self.__class__.new(res)
423: (4)                                 def __getitem__(
424: (8)                                 self: Array,
425: (8)                                 key: Union[
426: (12)                                     int,
427: (12)                                     slice,
428: (12)                                     ellipsis,
429: (12)                                     Tuple[Union[int, slice, ellipsis, None], ...],
430: (12)                                     Array,
431: (8)                                     ],
432: (8)                                     /,

```

```

433: (4)           ) -> Array:
434: (8)           """
435: (8)           Performs the operation __getitem__.
436: (8)           """
437: (8)           self._validate_index(key)
438: (8)           if isinstance(key, Array):
439: (12)             key = key._array
440: (8)             res = self._array.__getitem__(key)
441: (8)             return self._new(res)
442: (4)           def __gt__(self: Array, other: Union[int, float, Array], /) -> Array:
443: (8)           """
444: (8)           Performs the operation __gt__.
445: (8)           """
446: (8)           other = self._check_allowed_dtypes(other, "real numeric", "__gt__")
447: (8)           if other is NotImplemented:
448: (12)             return other
449: (8)             self, other = self._normalize_two_args(self, other)
450: (8)             res = self._array.__gt__(other._array)
451: (8)             return self.__class__.__new__(res)
452: (4)           def __int__(self: Array, /) -> int:
453: (8)           """
454: (8)           Performs the operation __int__.
455: (8)           """
456: (8)           if self._array.ndim != 0:
457: (12)             raise TypeError("int is only allowed on arrays with 0 dimensions")
458: (8)           if self.dtype in _complex_floating_dtypes:
459: (12)             raise TypeError("int is not allowed on complex floating-point
arrays")
460: (8)           res = self._array.__int__()
461: (8)           return res
462: (4)           def __index__(self: Array, /) -> int:
463: (8)           """
464: (8)           Performs the operation __index__.
465: (8)           """
466: (8)           res = self._array.__index__()
467: (8)           return res
468: (4)           def __invert__(self: Array, /) -> Array:
469: (8)           """
470: (8)           Performs the operation __invert__.
471: (8)           """
472: (8)           if self.dtype not in _integer_or_boolean_dtypes:
473: (12)             raise TypeError("Only integer or boolean dtypes are allowed in
__invert__")
474: (8)           res = self._array.__invert__()
475: (8)           return self.__class__.__new__(res)
476: (4)           def __le__(self: Array, other: Union[int, float, Array], /) -> Array:
477: (8)           """
478: (8)           Performs the operation __le__.
479: (8)           """
480: (8)           other = self._check_allowed_dtypes(other, "real numeric", "__le__")
481: (8)           if other is NotImplemented:
482: (12)             return other
483: (8)             self, other = self._normalize_two_args(self, other)
484: (8)             res = self._array.__le__(other._array)
485: (8)             return self.__class__.__new__(res)
486: (4)           def __lshift__(self: Array, other: Union[int, Array], /) -> Array:
487: (8)           """
488: (8)           Performs the operation __lshift__.
489: (8)           """
490: (8)           other = self._check_allowed_dtypes(other, "integer", "__lshift__")
491: (8)           if other is NotImplemented:
492: (12)             return other
493: (8)             self, other = self._normalize_two_args(self, other)
494: (8)             res = self._array.__lshift__(other._array)
495: (8)             return self.__class__.__new__(res)
496: (4)           def __lt__(self: Array, other: Union[int, float, Array], /) -> Array:
497: (8)           """
498: (8)           Performs the operation __lt__.
499: (8)           """

```

```

500: (8)          other = self._check_allowed_dtypes(other, "real numeric", "__lt__")
501: (8)          if other is NotImplemented:
502: (12)            return other
503: (8)          self, other = self._normalize_two_args(self, other)
504: (8)          res = self._array.__lt__(other._array)
505: (8)          return self.__class__.new(res)
506: (4)          def __matmul__(self: Array, other: Array, /) -> Array:
507: (8)            """
508: (8)              Performs the operation __matmul__.
509: (8)            """
510: (8)          other = self._check_allowed_dtypes(other, "numeric", "__matmul__")
511: (8)          if other is NotImplemented:
512: (12)            return other
513: (8)          res = self._array.__matmul__(other._array)
514: (8)          return self.__class__.new(res)
515: (4)          def __mod__(self: Array, other: Union[int, float, Array], /) -> Array:
516: (8)            """
517: (8)              Performs the operation __mod__.
518: (8)            """
519: (8)          other = self._check_allowed_dtypes(other, "real numeric", "__mod__")
520: (8)          if other is NotImplemented:
521: (12)            return other
522: (8)          self, other = self._normalize_two_args(self, other)
523: (8)          res = self._array.__mod__(other._array)
524: (8)          return self.__class__.new(res)
525: (4)          def __mul__(self: Array, other: Union[int, float, Array], /) -> Array:
526: (8)            """
527: (8)              Performs the operation __mul__.
528: (8)            """
529: (8)          other = self._check_allowed_dtypes(other, "numeric", "__mul__")
530: (8)          if other is NotImplemented:
531: (12)            return other
532: (8)          self, other = self._normalize_two_args(self, other)
533: (8)          res = self._array.__mul__(other._array)
534: (8)          return self.__class__.new(res)
535: (4)          def __ne__(self: Array, other: Union[int, float, bool, Array], /) ->
Array:
536: (8)            """
537: (8)              Performs the operation __ne__.
538: (8)            """
539: (8)          other = self._check_allowed_dtypes(other, "all", "__ne__")
540: (8)          if other is NotImplemented:
541: (12)            return other
542: (8)          self, other = self._normalize_two_args(self, other)
543: (8)          res = self._array.__ne__(other._array)
544: (8)          return self.__class__.new(res)
545: (4)          def __neg__(self: Array, /) -> Array:
546: (8)            """
547: (8)              Performs the operation __neg__.
548: (8)            """
549: (8)          if self.dtype not in _numeric_dtypes:
550: (12)            raise TypeError("Only numeric dtypes are allowed in __neg__")
551: (8)          res = self._array.__neg__()
552: (8)          return self.__class__.new(res)
553: (4)          def __or__(self: Array, other: Union[int, bool, Array], /) -> Array:
554: (8)            """
555: (8)              Performs the operation __or__.
556: (8)            """
557: (8)          other = self._check_allowed_dtypes(other, "integer or boolean",
"__or__")
558: (8)          if other is NotImplemented:
559: (12)            return other
560: (8)          self, other = self._normalize_two_args(self, other)
561: (8)          res = self._array.__or__(other._array)
562: (8)          return self.__class__.new(res)
563: (4)          def __pos__(self: Array, /) -> Array:
564: (8)            """
565: (8)              Performs the operation __pos__.
566: (8)            """

```

```

567: (8)             if self.dtype not in _numeric_dtypes:
568: (12)             raise TypeError("Only numeric dtypes are allowed in __pos__")
569: (8)             res = self._array.__pos__()
570: (8)             return self.__class__.__new__(res)
571: (4)         def __pow__(self: Array, other: Union[int, float, Array], /) -> Array:
572: (8)             """
573: (8)             Performs the operation __pow__.
574: (8)             """
575: (8)             from ._elementwise_functions import pow
576: (8)             other = self._check_allowed_dtypes(other, "numeric", "__pow__")
577: (8)             if other is NotImplemented:
578: (12)                 return other
579: (8)             return pow(self, other)
580: (4)         def __rshift__(self: Array, other: Union[int, Array], /) -> Array:
581: (8)             """
582: (8)             Performs the operation __rshift__.
583: (8)             """
584: (8)             other = self._check_allowed_dtypes(other, "integer", "__rshift__")
585: (8)             if other is NotImplemented:
586: (12)                 return other
587: (8)             self, other = self._normalize_two_args(self, other)
588: (8)             res = self._array.__rshift__(other._array)
589: (8)             return self.__class__.__new__(res)
590: (4)         def __setitem__(
591: (8)             self,
592: (8)             key: Union[
593: (12)                 int, slice, Ellipsis, Tuple[Union[int, slice, Ellipsis], ...],
Array
594: (8)                     ],
595: (8)                     value: Union[int, float, bool, Array],
596: (8)                     /,
597: (4)             ) -> None:
598: (8)             """
599: (8)             Performs the operation __setitem__.
600: (8)             """
601: (8)             self._validate_index(key)
602: (8)             if isinstance(key, Array):
603: (12)                 key = key._array
604: (8)             self._array.__setitem__(key, asarray(value)._array)
605: (4)         def __sub__(self: Array, other: Union[int, float, Array], /) -> Array:
606: (8)             """
607: (8)             Performs the operation __sub__.
608: (8)             """
609: (8)             other = self._check_allowed_dtypes(other, "numeric", "__sub__")
610: (8)             if other is NotImplemented:
611: (12)                 return other
612: (8)             self, other = self._normalize_two_args(self, other)
613: (8)             res = self._array.__sub__(other._array)
614: (8)             return self.__class__.__new__(res)
615: (4)         def __truediv__(self: Array, other: Union[float, Array], /) -> Array:
616: (8)             """
617: (8)             Performs the operation __truediv__.
618: (8)             """
619: (8)             other = self._check_allowed_dtypes(other, "floating-point",
"__truediv__")
620: (8)             if other is NotImplemented:
621: (12)                 return other
622: (8)             self, other = self._normalize_two_args(self, other)
623: (8)             res = self._array.__truediv__(other._array)
624: (8)             return self.__class__.__new__(res)
625: (4)         def __xor__(self: Array, other: Union[int, bool, Array], /) -> Array:
626: (8)             """
627: (8)             Performs the operation __xor__.
628: (8)             """
629: (8)             other = self._check_allowed_dtypes(other, "integer or boolean",
"__xor__")
630: (8)             if other is NotImplemented:
631: (12)                 return other
632: (8)             self, other = self._normalize_two_args(self, other)

```

```

633: (8)             res = self._array.__xor__(other._array)
634: (8)             return self.__class__.new(res)
635: (4)             def __iadd__(self: Array, other: Union[int, float, Array], /) -> Array:
636: (8)                 """
637: (8)                     Performs the operation __iadd__.
638: (8)                 """
639: (8)                 other = self._check_allowed_dtypes(other, "numeric", "__iadd__")
640: (8)                 if other is NotImplemented:
641: (12)                     return other
642: (8)                 self._array.__iadd__(other._array)
643: (8)                 return self
644: (4)             def __radd__(self: Array, other: Union[int, float, Array], /) -> Array:
645: (8)                 """
646: (8)                     Performs the operation __radd__.
647: (8)                 """
648: (8)                 other = self._check_allowed_dtypes(other, "numeric", "__radd__")
649: (8)                 if other is NotImplemented:
650: (12)                     return other
651: (8)                 self, other = self._normalize_two_args(self, other)
652: (8)                 res = self._array.__radd__(other._array)
653: (8)                 return self.__class__.new(res)
654: (4)             def __iand__(self: Array, other: Union[int, bool, Array], /) -> Array:
655: (8)                 """
656: (8)                     Performs the operation __iand__.
657: (8)                 """
658: (8)                 other = self._check_allowed_dtypes(other, "integer or boolean",
659: ("__iand__"))
660: (8)                 if other is NotImplemented:
661: (12)                     return other
662: (8)                 self._array.__iand__(other._array)
663: (8)                 return self
664: (4)             def __rand__(self: Array, other: Union[int, bool, Array], /) -> Array:
665: (8)                 """
666: (8)                     Performs the operation __rand__.
667: (8)                 """
668: ("__rand__"))
669: (8)                 other = self._check_allowed_dtypes(other, "integer or boolean",
670: (8)                 if other is NotImplemented:
671: (12)                     return other
672: (8)                 self, other = self._normalize_two_args(self, other)
673: (8)                 res = self._array.__rand__(other._array)
674: (4)             def __ifloordiv__(self: Array, other: Union[int, float, Array], /) ->
675: (8)                 """
676: (8)                     Performs the operation __ifloordiv__.
677: (8)                 """
678: ("__ifloordiv__"))
679: (8)                 other = self._check_allowed_dtypes(other, "real numeric",
680: (8)                 if other is NotImplemented:
681: (12)                     return other
682: (8)                 self._array.__ifloordiv__(other._array)
683: (4)             def __rfloordiv__(self: Array, other: Union[int, float, Array], /) ->
684: (8)                 """
685: (8)                     Performs the operation __rfloordiv__.
686: (8)                 """
687: ("__rfloordiv__"))
688: (8)                 other = self._check_allowed_dtypes(other, "real numeric",
689: (8)                 if other is NotImplemented:
690: (12)                     return other
691: (8)                 self, other = self._normalize_two_args(self, other)
692: (4)             def __ilshift__(self: Array, other: Union[int, Array], /) -> Array:
693: (8)                 """
694: (8)                     Performs the operation __ilshift__.
695: (8)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

696: (8)          other = self._check_allowed_dtypes(other, "integer", "__ilshift__")
697: (8)          if other is NotImplemented:
698: (12)            return other
699: (8)          self._array.__ilshift__(other._array)
700: (8)          return self
701: (4)          def __rlshift__(self: Array, other: Union[int, Array], /) -> Array:
702: (8)            """
703: (8)              Performs the operation __rlshift__.
704: (8)            """
705: (8)          other = self._check_allowed_dtypes(other, "integer", "__rlshift__")
706: (8)          if other is NotImplemented:
707: (12)            return other
708: (8)          self, other = self._normalize_two_args(self, other)
709: (8)          res = self._array.__rlshift__(other._array)
710: (8)          return self.__class__.new(res)
711: (4)          def __imatmul__(self: Array, other: Array, /) -> Array:
712: (8)            """
713: (8)              Performs the operation __imatmul__.
714: (8)            """
715: (8)          other = self._check_allowed_dtypes(other, "numeric", "__imatmul__")
716: (8)          if other is NotImplemented:
717: (12)            return other
718: (8)          res = self._array.__imatmul__(other._array)
719: (8)          return self.__class__.new(res)
720: (4)          def __rmatmul__(self: Array, other: Array, /) -> Array:
721: (8)            """
722: (8)              Performs the operation __rmatmul__.
723: (8)            """
724: (8)          other = self._check_allowed_dtypes(other, "numeric", "__rmatmul__")
725: (8)          if other is NotImplemented:
726: (12)            return other
727: (8)          res = self._array.__rmatmul__(other._array)
728: (8)          return self.__class__.new(res)
729: (4)          def __imod__(self: Array, other: Union[int, float, Array], /) -> Array:
730: (8)            """
731: (8)              Performs the operation __imod__.
732: (8)            """
733: (8)          other = self._check_allowed_dtypes(other, "real numeric", "__imod__")
734: (8)          if other is NotImplemented:
735: (12)            return other
736: (8)          self._array.__imod__(other._array)
737: (8)          return self
738: (4)          def __rmod__(self: Array, other: Union[int, float, Array], /) -> Array:
739: (8)            """
740: (8)              Performs the operation __rmod__.
741: (8)            """
742: (8)          other = self._check_allowed_dtypes(other, "real numeric", "__rmod__")
743: (8)          if other is NotImplemented:
744: (12)            return other
745: (8)          self, other = self._normalize_two_args(self, other)
746: (8)          res = self._array.__rmod__(other._array)
747: (8)          return self.__class__.new(res)
748: (4)          def __imul__(self: Array, other: Union[int, float, Array], /) -> Array:
749: (8)            """
750: (8)              Performs the operation __imul__.
751: (8)            """
752: (8)          other = self._check_allowed_dtypes(other, "numeric", "__imul__")
753: (8)          if other is NotImplemented:
754: (12)            return other
755: (8)          self._array.__imul__(other._array)
756: (8)          return self
757: (4)          def __rmul__(self: Array, other: Union[int, float, Array], /) -> Array:
758: (8)            """
759: (8)              Performs the operation __rmul__.
760: (8)            """
761: (8)          other = self._check_allowed_dtypes(other, "numeric", "__rmul__")
762: (8)          if other is NotImplemented:
763: (12)            return other
764: (8)          self, other = self._normalize_two_args(self, other)

```

```

765: (8)             res = self._array.__rmul__(other._array)
766: (8)             return self.__class__.new(res)
767: (4)             def __ior__(self: Array, other: Union[int, bool, Array], /) -> Array:
768: (8)                 """
769: (8)                 Performs the operation __ior__.
770: (8)                 """
771: (8)                 other = self._check_allowed_dtypes(other, "integer or boolean",
772: (8)                 if other is NotImplemented:
773: (12)                     return other
774: (8)                 self._array.__ior__(other._array)
775: (8)                 return self
776: (4)             def __ror__(self: Array, other: Union[int, bool, Array], /) -> Array:
777: (8)                 """
778: (8)                 Performs the operation __ror__.
779: (8)                 """
780: (8)                 other = self._check_allowed_dtypes(other, "integer or boolean",
781: (8)                 if other is NotImplemented:
782: (12)                     return other
783: (8)                 self, other = self._normalize_two_args(self, other)
784: (8)                 res = self._array.__ror__(other._array)
785: (8)                 return self.__class__.new(res)
786: (4)             def __ipow__(self: Array, other: Union[int, float, Array], /) -> Array:
787: (8)                 """
788: (8)                 Performs the operation __ipow__.
789: (8)                 """
790: (8)                 other = self._check_allowed_dtypes(other, "numeric", "__ipow__")
791: (8)                 if other is NotImplemented:
792: (12)                     return other
793: (8)                 self._array.__ipow__(other._array)
794: (8)                 return self
795: (4)             def __rpow__(self: Array, other: Union[int, float, Array], /) -> Array:
796: (8)                 """
797: (8)                 Performs the operation __rpow__.
798: (8)                 """
799: (8)                 from ._elementwise_functions import pow
800: (8)                 other = self._check_allowed_dtypes(other, "numeric", "__rpow__")
801: (8)                 if other is NotImplemented:
802: (12)                     return other
803: (8)                 return pow(other, self)
804: (4)             def __irshift__(self: Array, other: Union[int, Array], /) -> Array:
805: (8)                 """
806: (8)                 Performs the operation __irshift__.
807: (8)                 """
808: (8)                 other = self._check_allowed_dtypes(other, "integer", "__irshift__")
809: (8)                 if other is NotImplemented:
810: (12)                     return other
811: (8)                 self._array.__irshift__(other._array)
812: (8)                 return self
813: (4)             def __rrshift__(self: Array, other: Union[int, Array], /) -> Array:
814: (8)                 """
815: (8)                 Performs the operation __rrshift__.
816: (8)                 """
817: (8)                 other = self._check_allowed_dtypes(other, "integer", "__rrshift__")
818: (8)                 if other is NotImplemented:
819: (12)                     return other
820: (8)                 self, other = self._normalize_two_args(self, other)
821: (8)                 res = self._array.__rrshift__(other._array)
822: (8)                 return self.__class__.new(res)
823: (4)             def __isub__(self: Array, other: Union[int, float, Array], /) -> Array:
824: (8)                 """
825: (8)                 Performs the operation __isub__.
826: (8)                 """
827: (8)                 other = self._check_allowed_dtypes(other, "numeric", "__isub__")
828: (8)                 if other is NotImplemented:
829: (12)                     return other
830: (8)                 self._array.__isub__(other._array)
831: (8)                 return self

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

832: (4) def __rsub__(self: Array, other: Union[int, float, Array], /) -> Array:
833: (8)     """
834: (8)     Performs the operation __rsub__.
835: (8)
836: (8)     other = self._check_allowed_dtypes(other, "numeric", "__rsub__")
837: (8)     if other is NotImplemented:
838: (12)         return other
839: (8)     self, other = self._normalize_two_args(self, other)
840: (8)     res = self._array.__rsub__(other._array)
841: (8)     return self.__class__.new(res)
842: (4) def __itruediv__(self: Array, other: Union[float, Array], /) -> Array:
843: (8)
844: (8)     """
845: (8)     Performs the operation __itruediv__.
846: (8)
847: (8)     other = self._check_allowed_dtypes(other, "floating-point",
848: (8)     "__itruediv__")
849: (8)
850: (8)     if other is NotImplemented:
851: (12)         return other
852: (8)     self._array.__itruediv__(other._array)
853: (8)     return self
854: (8)
855: (8) def __rtruediv__(self: Array, other: Union[float, Array], /) -> Array:
856: (8)
857: (12)
858: (8)
859: (8)
860: (8)
861: (4) def __ixor__(self: Array, other: Union[int, bool, Array], /) -> Array:
862: (8)
863: (8)
864: (8)
865: (8)
866: (8)     """
867: (12)
868: (8)
869: (8)
870: (4) def __rxor__(self: Array, other: Union[int, bool, Array], /) -> Array:
871: (8)
872: (8)
873: (8)
874: (8)
875: (8)
876: (12)
877: (8)
878: (8)
879: (8)
880: (4) def to_device(self: Array, device: Device, /, stream: None = None) ->
Array:
881: (8)     if stream is not None:
882: (12)         raise ValueError("The stream argument to to_device() is not
supported")
883: (8)
884: (12)
885: (8)
886: (4)
887: (4) @property
888: (8)
889: (8)     def dtype(self) -> Dtype:
890: (8)         """
891: (8)         Array API compatible wrapper for :py:meth:`np.ndarray.dtype
<numpy.ndarray.dtype>`.
892: (8)         See its docstring for more information.
893: (4)         """
894: (8)         return self._array.dtype
895: (4)     @property

```

```

894: (4)             def device(self) -> Device:
895: (8)                 return "cpu"
896: (4)             @property
897: (4)                 def mT(self) -> Array:
898: (8)                     from .linalg import matrix_transpose
899: (8)                     return matrix_transpose(self)
900: (4)             @property
901: (4)                 def ndim(self) -> int:
902: (8)                     """
903: (8)                         Array API compatible wrapper for :py:meth:`np.ndarray.ndim
<numpy.ndarray.ndim>`.
904: (8)                         See its docstring for more information.
905: (8)                     """
906: (8)                         return self._array.ndim
907: (4)             @property
908: (4)                 def shape(self) -> Tuple[int, ...]:
909: (8)                     """
910: (8)                         Array API compatible wrapper for :py:meth:`np.ndarray.shape
<numpy.ndarray.shape>`.
911: (8)                         See its docstring for more information.
912: (8)                     """
913: (8)                         return self._array.shape
914: (4)             @property
915: (4)                 def size(self) -> int:
916: (8)                     """
917: (8)                         Array API compatible wrapper for :py:meth:`np.ndarray.size
<numpy.ndarray.size>`.
918: (8)                         See its docstring for more information.
919: (8)                     """
920: (8)                         return self._array.size
921: (4)             @property
922: (4)                 def T(self) -> Array:
923: (8)                     """
924: (8)                         Array API compatible wrapper for :py:meth:`np.ndarray.T
<numpy.ndarray.T>`.
925: (8)                         See its docstring for more information.
926: (8)                     """
927: (8)                         if self.ndim != 2:
928: (12)                             raise ValueError("x.T requires x to have 2 dimensions. Use x.mT to
transpose stacks of matrices and permute_dims() to permute dimensions.")
929: (8)                         return self.__class__.__new__(self._array.T)

-----

```

## File 15 - \_constants.py:

```

1: (0)             import numpy as np
2: (0)             e = np.e
3: (0)             inf = np.inf
4: (0)             nan = np.nan
5: (0)             pi = np.pi
6: (0)             newaxis = np.newaxis

-----

```

## File 16 - \_creation\_functions.py:

```

1: (0)             from __future__ import annotations
2: (0)             from typing import TYPE_CHECKING, List, Optional, Tuple, Union
3: (0)             if TYPE_CHECKING:
4: (4)                 from ._typing import (
5: (8)                     Array,
6: (8)                     Device,
7: (8)                     Dtype,
8: (8)                     NestedSequence,
9: (8)                     SupportsBufferProtocol,
10: (4)                )
11: (4)                 from collections.abc import Sequence
12: (0)             from ._dtypes import _all_dtotypes


```

```

13: (0) import numpy as np
14: (0) def _check_valid_dtype(dtype):
15: (4)     for d in (None,) + _all_dtypes:
16: (8)         if dtype is d:
17: (12)             return
18: (4)     raise ValueError("dtype must be one of the supported dtypes")
19: (0) def asarray(
20: (4)     obj: Union[
21: (8)         Array,
22: (8)         bool,
23: (8)         int,
24: (8)         float,
25: (8)         NestedSequence[bool | int | float],
26: (8)         SupportsBufferProtocol,
27: (4)     ],
28: (4)     /,
29: (4)     *,
30: (4)     dtype: Optional[Dtype] = None,
31: (4)     device: Optional[Device] = None,
32: (4)     copy: Optional[Union[bool, np._CopyMode]] = None,
33: (0) ) -> Array:
34: (4)     """
35: (4)     Array API compatible wrapper for :py:func:`np.asarray <numpy.asarray>`.
36: (4)     See its docstring for more information.
37: (4)     """
38: (4)     from ._array_object import Array
39: (4)     _check_valid_dtype(dtype)
40: (4)     if device not in ["cpu", None]:
41: (8)         raise ValueError(f"Unsupported device {device!r}")
42: (4)     if copy in (False, np._CopyMode.IF_NEEDED):
43: (8)         raise NotImplementedError("copy=False is not yet implemented")
44: (4)     if isinstance(obj, Array):
45: (8)         if dtype is not None and obj.dtype != dtype:
46: (12)             copy = True
47: (8)         if copy in (True, np._CopyMode.ALWAYS):
48: (12)             return Array._new(np.array(obj._array, copy=True, dtype=dtype))
49: (8)         return obj
50: (4)     if dtype is None and isinstance(obj, int) and (obj > 2 ** 64 or obj < -(2
** 63)):
51: (8)         raise OverflowError("Integer out of bounds for array dtypes")
52: (4)     res = np.asarray(obj, dtype=dtype)
53: (4)     return Array._new(res)
54: (0) def arange(
55: (4)     start: Union[int, float],
56: (4)     /,
57: (4)     stop: Optional[Union[int, float]] = None,
58: (4)     step: Union[int, float] = 1,
59: (4)     *,
60: (4)     dtype: Optional[Dtype] = None,
61: (4)     device: Optional[Device] = None,
62: (0) ) -> Array:
63: (4)     """
64: (4)     Array API compatible wrapper for :py:func:`np.arange <numpy.arange>`.
65: (4)     See its docstring for more information.
66: (4)     """
67: (4)     from ._array_object import Array
68: (4)     _check_valid_dtype(dtype)
69: (4)     if device not in ["cpu", None]:
70: (8)         raise ValueError(f"Unsupported device {device!r}")
71: (4)     return Array._new(np.arange(start, stop=stop, step=step, dtype=dtype))
72: (0) def empty(
73: (4)     shape: Union[int, Tuple[int, ...]],
74: (4)     *,
75: (4)     dtype: Optional[Dtype] = None,
76: (4)     device: Optional[Device] = None,
77: (0) ) -> Array:
78: (4)     """
79: (4)     Array API compatible wrapper for :py:func:`np.empty <numpy.empty>`.
80: (4)     See its docstring for more information.

```

```

81: (4)
82: (4)
83: (4)
84: (4)
85: (8)
86: (4)
87: (0)
88: (4)
None
89: (0)
90: (4)
91: (4)
<numpy.empty_like` .
92: (4)
93: (4)
94: (4)
95: (4)
96: (4)
97: (8)
98: (4)
99: (0)
100: (4)
101: (4)
102: (4)
103: (4)
104: (4)
105: (4)
106: (4)
device: Optional[Device] =
107: (0)
) -> Array:
"""
Array API compatible wrapper for :py:func:`np.empty_like
<numpy.empty_like` .
See its docstring for more information.
"""
from ._array_object import Array
_check_valid_dtype(dtype)
if device not in ["cpu", None]:
    raise ValueError(f"Unsupported device {device!r}")
return Array._new(np.empty(shape, dtype=dtype))
def empty_like(
    x: Array, /, *, dtype: Optional[Dtype] = None, device: Optional[Device] =
None
) -> Array:
"""
Array API compatible wrapper for :py:func:`np.empty_like` .
See its docstring for more information.
"""
from ._array_object import Array
_check_valid_dtype(dtype)
if device not in ["cpu", None]:
    raise ValueError(f"Unsupported device {device!r}")
return Array._new(np.empty_like(x._array, dtype=dtype))
def eye(
    n_rows: int,
    n_cols: Optional[int] = None,
    /,
    *,
    k: int = 0,
    dtype: Optional[Dtype] = None,
    device: Optional[Device] = None,
) -> Array:
"""
Array API compatible wrapper for :py:func:`np.eye <numpy.eye` .
See its docstring for more information.
"""
from ._array_object import Array
_check_valid_dtype(dtype)
if device not in ["cpu", None]:
    raise ValueError(f"Unsupported device {device!r}")
return Array._new(np.eye(n_rows, M=n_cols, k=k, dtype=dtype))
def from_dlpark(x: object, /) -> Array:
    from ._array_object import Array
    return Array._new(np.from_dlpark(x))
def full(
    shape: Union[int, Tuple[int, ...]],
    fill_value: Union[int, float],
    *,
    dtype: Optional[Dtype] = None,
    device: Optional[Device] = None,
) -> Array:
"""
Array API compatible wrapper for :py:func:`np.full <numpy.full` .
See its docstring for more information.
"""
from ._array_object import Array
_check_valid_dtype(dtype)
if device not in ["cpu", None]:
    raise ValueError(f"Unsupported device {device!r}")
if isinstance(fill_value, Array) and fill_value.ndim == 0:
    fill_value = fill_value._array
res = np.full(shape, fill_value, dtype=dtype)
if res.dtype not in _all_dtypes:
    raise TypeError("Invalid input to full")
return Array._new(res)
def full_like(
    x: Array,
    /,
    fill_value: Union[int, float],
    *,
    dtype: Optional[Dtype] = None,
    device: Optional[Device] = None,
) -> Array:
"""
Array API compatible wrapper for :py:func:`np.full_like <numpy.full_like` .
See its docstring for more information.
"""
from ._array_object import Array
_check_valid_dtype(dtype)
if device not in ["cpu", None]:
    raise ValueError(f"Unsupported device {device!r}")
if isinstance(fill_value, Array) and fill_value.ndim == 0:
    fill_value = fill_value._array
res = np.full_like(x, fill_value, dtype=dtype)
if res.dtype not in _all_dtypes:
    raise TypeError("Invalid input to full_like")
return Array._new(res)

```

```

148: (0) ) -> Array:
149: (4) """
150: (4)     Array API compatible wrapper for :py:func:`np.full_like
<numpy.full_like>`.
151: (4)     See its docstring for more information.
152: (4) """
153: (4)     from ._array_object import Array
154: (4)     _check_valid_dtype(dtype)
155: (4)     if device not in ["cpu", None]:
156: (8)         raise ValueError(f"Unsupported device {device!r}")
157: (4)     res = np.full_like(x._array, fill_value, dtype=dtype)
158: (4)     if res.dtype not in _all_dtypes:
159: (8)         raise TypeError("Invalid input to full_like")
160: (4)     return Array._new(res)
161: (0) def linspace(
162: (4)     start: Union[int, float],
163: (4)     stop: Union[int, float],
164: (4)     /,
165: (4)     num: int,
166: (4)     *,
167: (4)     dtype: Optional[Dtype] = None,
168: (4)     device: Optional[Device] = None,
169: (4)     endpoint: bool = True,
170: (0) ) -> Array:
171: (4) """
172: (4)     Array API compatible wrapper for :py:func:`np.linspace <numpy.linspace>`.
173: (4)     See its docstring for more information.
174: (4) """
175: (4)     from ._array_object import Array
176: (4)     _check_valid_dtype(dtype)
177: (4)     if device not in ["cpu", None]:
178: (8)         raise ValueError(f"Unsupported device {device!r}")
179: (4)     return Array._new(np.linspace(start, stop, num, dtype=dtype,
180: (0)     endpoint=endpoint))
181: (4) def meshgrid(*arrays: Array, indexing: str = "xy") -> List[Array]:
182: (4) """
183: (4)     Array API compatible wrapper for :py:func:`np.meshgrid <numpy.meshgrid>`.
184: (4)     See its docstring for more information.
185: (4) """
186: (4)     from ._array_object import Array
187: (8)     if len({a.dtype for a in arrays}) > 1:
188: (4)         raise ValueError("meshgrid inputs must all have the same dtype")
189: (8)     return [
190: (4)         Array._new(array)
191: (4)         for array in np.meshgrid(*[a._array for a in arrays],
192: (0)         indexing=indexing)
193: (4)     ]
194: (4) def ones(
195: (4)     shape: Union[int, Tuple[int, ...]],
196: (4)     *,
197: (4)     dtype: Optional[Dtype] = None,
198: (4)     device: Optional[Device] = None,
199: (0) ) -> Array:
200: (4) """
201: (4)     Array API compatible wrapper for :py:func:`np.ones <numpy.ones>`.
202: (4)     See its docstring for more information.
203: (4) """
204: (4)     from ._array_object import Array
205: (4)     _check_valid_dtype(dtype)
206: (4)     if device not in ["cpu", None]:
207: (8)         raise ValueError(f"Unsupported device {device!r}")
208: (4)     return Array._new(np.ones(shape, dtype=dtype))
209: (0) def ones_like(
210: (4)     x: Array, /, *, dtype: Optional[Dtype] = None, device: Optional[Device] =
None
211: (4) ) -> Array:
212: (4) """
213: (4)     Array API compatible wrapper for :py:func:`np.ones_like
<numpy.ones_like>`.

```

```

212: (4)             See its docstring for more information.
213: (4)
214: (4)             from ._array_object import Array
215: (4)             _check_valid_dtype(dtype)
216: (4)             if device not in ["cpu", None]:
217: (8)                 raise ValueError(f"Unsupported device {device!r}")
218: (4)             return Array._new(np.ones_like(x._array, dtype=dtype))
219: (0)             def tril(x: Array, /, *, k: int = 0) -> Array:
220: (4)
221: (4)                 """
222: (4)                     Array API compatible wrapper for :py:func:`np.tril <numpy.tril>`.
223: (4)                     See its docstring for more information.
224: (4)
225: (4)                     from ._array_object import Array
226: (4)                     if x.ndim < 2:
227: (8)                         raise ValueError("x must be at least 2-dimensional for tril")
228: (4)                     return Array._new(np.tril(x._array, k=k))
229: (0)             def triu(x: Array, /, *, k: int = 0) -> Array:
230: (4)
231: (4)                 """
232: (4)                     Array API compatible wrapper for :py:func:`np.triu <numpy.triu>`.
233: (4)                     See its docstring for more information.
234: (4)
235: (4)                     from ._array_object import Array
236: (4)                     if x.ndim < 2:
237: (8)                         raise ValueError("x must be at least 2-dimensional for triu")
238: (4)                     return Array._new(np.triu(x._array, k=k))
239: (0)             def zeros(
240: (4)                 shape: Union[int, Tuple[int, ...]],
241: (4)                 *,
242: (4)                 dtype: Optional[Dtype] = None,
243: (4)                 device: Optional[Device] = None,
244: (0)             ) -> Array:
245: (4)
246: (4)                 """
247: (4)                     Array API compatible wrapper for :py:func:`np.zeros <numpy.zeros>`.
248: (4)                     See its docstring for more information.
249: (4)
250: (4)                     from ._array_object import Array
251: (4)                     _check_valid_dtype(dtype)
252: (4)                     if device not in ["cpu", None]:
253: (8)                         raise ValueError(f"Unsupported device {device!r}")
254: (4)                     return Array._new(np.zeros(shape, dtype=dtype))
255: (0)             def zeros_like(
256: (4)                 x: Array, /, *, dtype: Optional[Dtype] = None, device: Optional[Device] =
None
257: (0)             ) -> Array:
258: (4)
259: (4)                 """
260: (4)                     Array API compatible wrapper for :py:func:`np.zeros_like
<numpy.zeros_like>`.
261: (4)                     See its docstring for more information.
262: (4)
263: (4)                     from ._array_object import Array
264: (4)                     _check_valid_dtype(dtype)
265: (4)                     if device not in ["cpu", None]:
266: (8)                         raise ValueError(f"Unsupported device {device!r}")
267: (4)                     return Array._new(np.zeros_like(x._array, dtype=dtype))

```

---

## File 17 - \_dtypes.py:

```

1: (0)             import numpy as np
2: (0)             int8 = np.dtype("int8")
3: (0)             int16 = np.dtype("int16")
4: (0)             int32 = np.dtype("int32")
5: (0)             int64 = np.dtype("int64")
6: (0)             uint8 = np.dtype("uint8")
7: (0)             uint16 = np.dtype("uint16")
8: (0)             uint32 = np.dtype("uint32")
9: (0)             uint64 = np.dtype("uint64")
10: (0)            float32 = np.dtype("float32")

```

```
11: (0)          float64 = np.dtype("float64")
12: (0)          complex64 = np.dtype("complex64")
13: (0)          complex128 = np.dtype("complex128")
14: (0)          bool = np.dtype("bool")
15: (0)          _all_dtypes = (
16: (4)            int8,
17: (4)            int16,
18: (4)            int32,
19: (4)            int64,
20: (4)            uint8,
21: (4)            uint16,
22: (4)            uint32,
23: (4)            uint64,
24: (4)            float32,
25: (4)            float64,
26: (4)            complex64,
27: (4)            complex128,
28: (4)            bool,
29: (0)
30: (0)          _boolean_dtypes = (bool,)
31: (0)          _real_floating_dtypes = (float32, float64)
32: (0)          _floating_dtypes = (float32, float64, complex64, complex128)
33: (0)          _complex_floating_dtypes = (complex64, complex128)
34: (0)          _integer_dtypes = (int8, int16, int32, int64, uint8, uint16, uint32, uint64)
35: (0)          _signed_integer_dtypes = (int8, int16, int32, int64)
36: (0)          _unsigned_integer_dtypes = (uint8, uint16, uint32, uint64)
37: (0)          _integer_or_boolean_dtypes = (
38: (4)            bool,
39: (4)            int8,
40: (4)            int16,
41: (4)            int32,
42: (4)            int64,
43: (4)            uint8,
44: (4)            uint16,
45: (4)            uint32,
46: (4)            uint64,
47: (0)
48: (0)          _real_numeric_dtypes = (
49: (4)            float32,
50: (4)            float64,
51: (4)            int8,
52: (4)            int16,
53: (4)            int32,
54: (4)            int64,
55: (4)            uint8,
56: (4)            uint16,
57: (4)            uint32,
58: (4)            uint64,
59: (0)
60: (0)          _numeric_dtypes = (
61: (4)            float32,
62: (4)            float64,
63: (4)            complex64,
64: (4)            complex128,
65: (4)            int8,
66: (4)            int16,
67: (4)            int32,
68: (4)            int64,
69: (4)            uint8,
70: (4)            uint16,
71: (4)            uint32,
72: (4)            uint64,
73: (0)
74: (0)          _dtype_categories = {
75: (4)            "all": _all_dtypes,
76: (4)            "real numeric": _real_numeric_dtypes,
77: (4)            "numeric": _numeric_dtypes,
78: (4)            "integer": _integer_dtypes,
79: (4)            "integer or boolean": _integer_or_boolean_dtypes,
```

```
80: (4)          "boolean": _boolean_dtypes,
81: (4)          "real floating-point": _floating_dtypes,
82: (4)          "complex floating-point": _complex_floating_dtypes,
83: (4)          "floating-point": _floating_dtypes,
84: (0)
85: (0)          _promotion_table = {
86: (4)            (int8, int8): int8,
87: (4)            (int8, int16): int16,
88: (4)            (int8, int32): int32,
89: (4)            (int8, int64): int64,
90: (4)            (int16, int8): int16,
91: (4)            (int16, int16): int16,
92: (4)            (int16, int32): int32,
93: (4)            (int16, int64): int64,
94: (4)            (int32, int8): int32,
95: (4)            (int32, int16): int32,
96: (4)            (int32, int32): int32,
97: (4)            (int32, int64): int64,
98: (4)            (int64, int8): int64,
99: (4)            (int64, int16): int64,
100: (4)            (int64, int32): int64,
101: (4)            (int64, int64): int64,
102: (4)            (uint8, uint8): uint8,
103: (4)            (uint8, uint16): uint16,
104: (4)            (uint8, uint32): uint32,
105: (4)            (uint8, uint64): uint64,
106: (4)            (uint16, uint8): uint16,
107: (4)            (uint16, uint16): uint16,
108: (4)            (uint16, uint32): uint32,
109: (4)            (uint16, uint64): uint64,
110: (4)            (uint32, uint8): uint32,
111: (4)            (uint32, uint16): uint32,
112: (4)            (uint32, uint32): uint32,
113: (4)            (uint32, uint64): uint64,
114: (4)            (uint64, uint8): uint64,
115: (4)            (uint64, uint16): uint64,
116: (4)            (uint64, uint32): uint64,
117: (4)            (uint64, uint64): uint64,
118: (4)            (int8, uint8): int16,
119: (4)            (int8, uint16): int32,
120: (4)            (int8, uint32): int64,
121: (4)            (int16, uint8): int16,
122: (4)            (int16, uint16): int32,
123: (4)            (int16, uint32): int64,
124: (4)            (int32, uint8): int32,
125: (4)            (int32, uint16): int32,
126: (4)            (int32, uint32): int64,
127: (4)            (int64, uint8): int64,
128: (4)            (int64, uint16): int64,
129: (4)            (int64, uint32): int64,
130: (4)            (uint8, int8): int16,
131: (4)            (uint16, int8): int32,
132: (4)            (uint32, int8): int64,
133: (4)            (uint8, int16): int16,
134: (4)            (uint16, int16): int32,
135: (4)            (uint32, int16): int64,
136: (4)            (uint8, int32): int32,
137: (4)            (uint16, int32): int32,
138: (4)            (uint32, int32): int64,
139: (4)            (uint8, int64): int64,
140: (4)            (uint16, int64): int64,
141: (4)            (uint32, int64): int64,
142: (4)            (float32, float32): float32,
143: (4)            (float32, float64): float64,
144: (4)            (float64, float32): float64,
145: (4)            (float64, float64): float64,
146: (4)            (complex64, complex64): complex64,
147: (4)            (complex64, complex128): complex128,
148: (4)            (complex128, complex64): complex128,
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

149: (4)             (complex128, complex128): complex128,
150: (4)             (float32, complex64): complex64,
151: (4)             (float32, complex128): complex128,
152: (4)             (float64, complex64): complex128,
153: (4)             (float64, complex128): complex128,
154: (4)             (complex64, float32): complex64,
155: (4)             (complex64, float64): complex128,
156: (4)             (complex128, float32): complex128,
157: (4)             (complex128, float64): complex128,
158: (4)             (bool, bool): bool,
159: (0)
160: (0)
161: (4) def _result_type(type1, type2):
162: (8)     if (type1, type2) in _promotion_table:
163: (4)         return _promotion_table[type1, type2]
164: (4)     raise TypeError(f"{type1} and {type2} cannot be type promoted together")
-----
```

## File 18 - \_data\_type\_functions.py:

```

1: (0) from __future__ import annotations
2: (0) from .array_object import Array
3: (0) from .dtypes import (
4: (4)     _all_dtypes,
5: (4)     _boolean_dtypes,
6: (4)     _signed_integer_dtypes,
7: (4)     _unsigned_integer_dtypes,
8: (4)     _integer_dtypes,
9: (4)     _real_floating_dtypes,
10: (4)    _complex_floating_dtypes,
11: (4)    _numeric_dtypes,
12: (4)    _result_type,
13: (0)
14: (0)
15: (0) from typing import TYPE_CHECKING, List, Tuple, Union
16: (0) if TYPE_CHECKING:
17: (4)     from .typing import Dtype
18: (4)     from collections.abc import Sequence
19: (0) import numpy as np
20: (0) def astype(x: Array, dtype: Dtype, /, *, copy: bool = True) -> Array:
21: (4)     if not copy and dtype == x.dtype:
22: (8)         return x
23: (4)     return Array._new(x._array.astype(dtype=dtype, copy=copy))
24: (0) def broadcast_arrays(*arrays: Array) -> List[Array]:
25: (4)     """
26: (4)         Array API compatible wrapper for :py:func:`np.broadcast_arrays
<numpy.broadcast_arrays>`.
27: (4)         See its docstring for more information.
28: (4)     """
29: (4)     from .array_object import Array
30: (4)     return [
31: (8)         Array._new(array) for array in np.broadcast_arrays(*[a._array for a in
32: (4) arrays])
33: (4)     ]
34: (0)     def broadcast_to(x: Array, /, shape: Tuple[int, ...]) -> Array:
35: (4)         """
36: (4)             Array API compatible wrapper for :py:func:`np.broadcast_to
<numpy.broadcast_to>`.
37: (4)             See its docstring for more information.
38: (4)         """
39: (4)         from .array_object import Array
40: (0)         return Array._new(np.broadcast_to(x._array, shape))
41: (4)     def can_cast(from_: Union[Dtype, Array], to: Dtype, /) -> bool:
42: (4)         """
43: (4)             Array API compatible wrapper for :py:func:`np.can_cast <numpy.can_cast>`.
44: (4)             See its docstring for more information.
45: (4)         """
46: (8)         if isinstance(from_, Array):
47: (8)             from_ = from_.dtype

```

```

47: (4)           elif from_ not in _all_dtypes:
48: (8)             raise TypeError(f"{from_=}, but should be an array_api array or"
49: (4)               dtype")
50: (8)             if to not in _all_dtypes:
51: (4)               raise TypeError(f"{to=}, but should be a dtype")
52: (8)             try:
53: (8)               dtype = _result_type(from_, to)
54: (4)             except TypeError:
55: (8)               return False
56: (0)           @dataclass
57: (0)           class finfo_object:
58: (4)             bits: int
59: (4)             eps: float
60: (4)             max: float
61: (4)             min: float
62: (4)             smallest_normal: float
63: (4)             dtype: Dtype
64: (0)           @dataclass
65: (0)           class iinfo_object:
66: (4)             bits: int
67: (4)             max: int
68: (4)             min: int
69: (4)             dtype: Dtype
70: (0)           def finfo(type: Union[Dtype, Array], /) -> finfo_object:
71: (4)             """
72: (4)               Array API compatible wrapper for :py:func:`np.finfo <numpy.finfo>`.
73: (4)               See its docstring for more information.
74: (4)             """
75: (4)             fi = np.finfo(type)
76: (4)             return finfo_object(
77: (8)               fi.bits,
78: (8)               float(fi.eps),
79: (8)               float(fi.max),
80: (8)               float(fi.min),
81: (8)               float(fi.smallest_normal),
82: (8)               fi.dtype,
83: (4)             )
84: (0)           def iinfo(type: Union[Dtype, Array], /) -> iinfo_object:
85: (4)             """
86: (4)               Array API compatible wrapper for :py:func:`np.iinfo <numpy.iinfo>`.
87: (4)               See its docstring for more information.
88: (4)             """
89: (4)             ii = np.iinfo(type)
90: (4)             return iinfo_object(ii.bits, ii.max, ii.min, ii.dtype)
91: (0)           def isdtype(
92: (4)             dtype: Dtype, kind: Union[Dtype, str, Tuple[Union[Dtype, str], ...]]
93: (0)           ) -> bool:
94: (4)             """
95: (4)               Returns a boolean indicating whether a provided dtype is of a specified
data type ``kind``.
96: (4)             See
97: (4)               https://data-apis.org/array-
api/latest/API\_specification/generated/array\_api.isdtype.html
98: (4)               for more details
99: (4)             """
100: (4)             if isinstance(kind, tuple):
101: (8)               if any(isinstance(k, tuple) for k in kind):
102: (12)                 raise TypeError("'kind' must be a dtype, str, or tuple of dtypes
and strs")
103: (8)               return any(isdtype(dtype, k) for k in kind)
104: (4)             elif isinstance(kind, str):
105: (8)               if kind == 'bool':
106: (12)                 return dtype in _boolean_dtypes
107: (8)               elif kind == 'signed integer':
108: (12)                 return dtype in _signed_integer_dtypes
109: (8)               elif kind == 'unsigned integer':
110: (12)                 return dtype in _unsigned_integer_dtypes
111: (8)               elif kind == 'integral':
```

```

112: (12)             return dtype in _integer_dtotypes
113: (8)              elif kind == 'real floating':
114: (12)                return dtype in _real_floating_dtotypes
115: (8)              elif kind == 'complex floating':
116: (12)                return dtype in _complex_floating_dtotypes
117: (8)              elif kind == 'numeric':
118: (12)                return dtype in _numeric_dtotypes
119: (8)            else:
120: (12)                raise ValueError(f"Unrecognized data type kind: {kind!r}")
121: (4)            elif kind in _all_dtotypes:
122: (8)              return dtype == kind
123: (4)            else:
124: (8)                raise TypeError(f"'kind' must be a dtype, str, or tuple of dtypes and
strs, not {type(kind).__name__}")
125: (0)        def result_type(*arrays_and_dtotypes: Union[Array, Dtype]) -> Dtype:
126: (4)            """
127: (4)                Array API compatible wrapper for :py:func:`np.result_type
<numpy.result_type>`.
128: (4)            See its docstring for more information.
129: (4)            """
130: (4)            A = []
131: (4)            for a in arrays_and_dtotypes:
132: (8)                if isinstance(a, Array):
133: (12)                  a = a.dtype
134: (8)                elif isinstance(a, np.ndarray) or a not in _all_dtotypes:
135: (12)                  raise TypeError("result_type() inputs must be array_api arrays or
dtypes")
136: (8)            A.append(a)
137: (4)            if len(A) == 0:
138: (8)                raise ValueError("at least one array or dtype is required")
139: (4)            elif len(A) == 1:
140: (8)                return A[0]
141: (4)            else:
142: (8)                t = A[0]
143: (8)                for t2 in A[1:]:
144: (12)                  t = _result_type(t, t2)
145: (8)            return t

```

---

## File 19 - \_elementwise\_functions.py:

```

1: (0)      from __future__ import annotations
2: (0)      from ._dtypes import (
3: (4)          _boolean_dtotypes,
4: (4)          _floating_dtotypes,
5: (4)          _real_floating_dtotypes,
6: (4)          _complex_floating_dtotypes,
7: (4)          _integer_dtotypes,
8: (4)          _integer_or_boolean_dtotypes,
9: (4)          _real_numeric_dtotypes,
10: (4)          _numeric_dtotypes,
11: (4)          _result_type,
12: (0)      )
13: (0)      from ._array_object import Array
14: (0)      import numpy as np
15: (0)      def abs(x: Array, /) -> Array:
16: (4)          """
17: (4)              Array API compatible wrapper for :py:func:`np.abs <numpy.abs>`.
18: (4)              See its docstring for more information.
19: (4)              """
20: (4)              if x.dtype not in _numeric_dtotypes:
21: (8)                  raise TypeError("Only numeric dtypes are allowed in abs")
22: (4)              return Array._new(np.abs(x._array))
23: (0)      def acos(x: Array, /) -> Array:
24: (4)          """
25: (4)              Array API compatible wrapper for :py:func:`np.arccos <numpy.arccos>`.
26: (4)              See its docstring for more information.
27: (4)              """

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

28: (4)           if x.dtype not in _floating_dtypes:
29: (8)             raise TypeError("Only floating-point dtypes are allowed in acos")
30: (4)             return Array._new(np.arccos(x._array))
31: (0)         def acosh(x: Array, /) -> Array:
32: (4)             """
33: (4)               Array API compatible wrapper for :py:func:`np.arccosh <numpy.arccosh>` .
34: (4)               See its docstring for more information.
35: (4)
36: (4)               if x.dtype not in _floating_dtypes:
37: (8)                 raise TypeError("Only floating-point dtypes are allowed in acosh")
38: (4)                 return Array._new(np.arccosh(x._array))
39: (0)         def add(x1: Array, x2: Array, /) -> Array:
40: (4)             """
41: (4)               Array API compatible wrapper for :py:func:`np.add <numpy.add>` .
42: (4)               See its docstring for more information.
43: (4)
44: (4)               if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:
45: (8)                 raise TypeError("Only numeric dtypes are allowed in add")
46: (4)                 _result_type(x1.dtype, x2.dtype)
47: (4)                 x1, x2 = Array._normalize_two_args(x1, x2)
48: (4)                 return Array._new(np.add(x1._array, x2._array))
49: (0)         def asin(x: Array, /) -> Array:
50: (4)             """
51: (4)               Array API compatible wrapper for :py:func:`np.arcsin <numpy.arcsin>` .
52: (4)               See its docstring for more information.
53: (4)
54: (4)               if x.dtype not in _floating_dtypes:
55: (8)                 raise TypeError("Only floating-point dtypes are allowed in asin")
56: (4)                 return Array._new(np.arcsin(x._array))
57: (0)         def asinh(x: Array, /) -> Array:
58: (4)             """
59: (4)               Array API compatible wrapper for :py:func:`np.arcsinh <numpy.arcsinh>` .
60: (4)               See its docstring for more information.
61: (4)
62: (4)               if x.dtype not in _floating_dtypes:
63: (8)                 raise TypeError("Only floating-point dtypes are allowed in asinh")
64: (4)                 return Array._new(np.arcsinh(x._array))
65: (0)         def atan(x: Array, /) -> Array:
66: (4)             """
67: (4)               Array API compatible wrapper for :py:func:`np.arctan <numpy.arctan>` .
68: (4)               See its docstring for more information.
69: (4)
70: (4)               if x.dtype not in _floating_dtypes:
71: (8)                 raise TypeError("Only floating-point dtypes are allowed in atan")
72: (4)                 return Array._new(np.arctan(x._array))
73: (0)         def atan2(x1: Array, x2: Array, /) -> Array:
74: (4)             """
75: (4)               Array API compatible wrapper for :py:func:`np.arctan2 <numpy.arctan2>` .
76: (4)               See its docstring for more information.
77: (4)
78: (4)               if x1.dtype not in _real_floating_dtypes or x2.dtype not in
79: (8)                 _real_floating_dtypes:
80: (4)                   raise TypeError("Only real floating-point dtypes are allowed in
81: (4)                     atan2")
82: (4)                   _result_type(x1.dtype, x2.dtype)
83: (0)                   x1, x2 = Array._normalize_two_args(x1, x2)
84: (4)                   return Array._new(np.arctan2(x1._array, x2._array))
85: (0)         def atanh(x: Array, /) -> Array:
86: (4)             """
87: (4)               Array API compatible wrapper for :py:func:`np.arctanh <numpy.arctanh>` .
88: (4)               See its docstring for more information.
89: (4)
90: (4)               if x.dtype not in _floating_dtypes:
91: (8)                 raise TypeError("Only floating-point dtypes are allowed in atanh")
92: (4)                 return Array._new(np.arctanh(x._array))
93: (0)         def bitwise_and(x1: Array, x2: Array, /) -> Array:
94: (4)             """
95: (4)               Array API compatible wrapper for :py:func:`np.bitwise_and
<numpy.bitwise_and>` .

```

```

94: (4)           See its docstring for more information.
95: (4)
96: (4)
97: (8)
98: (8)
99: (4)
100: (8)
bitwise_and")
101: (4)
102: (4)
103: (4)
104: (0)
105: (4)
106: (4)
<numpy.left_shift>`.
107: (4)
108: (4)
109: (4)
110: (8)
bitwise_left_shift")
111: (4)
112: (4)
113: (4)
114: (8)
0")
115: (4)
116: (0)
117: (4)
118: (4)
119: (4)
120: (4)
121: (4)
122: (8)
bitwise_invert")
123: (4)
124: (0)
125: (4)
126: (4)
<numpy.bitwise_or>`.
127: (4)
128: (4)
129: (4)
130: (8)
131: (8)
132: (4)
133: (8)
bitwise_or")
134: (4)
135: (4)
136: (4)
137: (0)
138: (4)
139: (4)
<numpy.right_shift>`.
140: (4)
141: (4)
142: (4)
143: (8)
bitwise_right_shift")
144: (4)
145: (4)
146: (4)
147: (8)
0")
148: (4)
149: (0)
150: (4)
151: (4)
<numpy.bitwise_xor>`.

See its docstring for more information.
"""
if (
    x1.dtype not in _integer_or_boolean_dtypes
    or x2.dtype not in _integer_or_boolean_dtypes
):
    raise TypeError("Only integer or boolean dtypes are allowed in
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
return Array._new(np.bitwise_and(x1._array, x2._array))
def bitwise_left_shift(x1: Array, x2: Array, /) -> Array:
"""
Array API compatible wrapper for :py:func:`np.left_shift
See its docstring for more information.
"""
if x1.dtype not in _integer_dtypes or x2.dtype not in _integer_dtypes:
    raise TypeError("Only integer dtypes are allowed in
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
if np.any(x2._array < 0):
    raise ValueError("bitwise_left_shift(x1, x2) is only defined for x2 >=
0")
return Array._new(np.left_shift(x1._array, x2._array))
def bitwise_invert(x: Array, /) -> Array:
"""
Array API compatible wrapper for :py:func:`np.invert <numpy.invert>`.
See its docstring for more information.
"""
if x.dtype not in _integer_or_boolean_dtypes:
    raise TypeError("Only integer or boolean dtypes are allowed in
return Array._new(np.invert(x._array))
def bitwise_or(x1: Array, x2: Array, /) -> Array:
"""
Array API compatible wrapper for :py:func:`np.bitwise_or
See its docstring for more information.
"""
if (
    x1.dtype not in _integer_or_boolean_dtypes
    or x2.dtype not in _integer_or_boolean_dtypes
):
    raise TypeError("Only integer or boolean dtypes are allowed in
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
return Array._new(np.bitwise_or(x1._array, x2._array))
def bitwise_right_shift(x1: Array, x2: Array, /) -> Array:
"""
Array API compatible wrapper for :py:func:`np.right_shift
See its docstring for more information.
"""
if x1.dtype not in _integer_dtypes or x2.dtype not in _integer_dtypes:
    raise TypeError("Only integer dtypes are allowed in
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
if np.any(x2._array < 0):
    raise ValueError("bitwise_right_shift(x1, x2) is only defined for x2 >=
0")
return Array._new(np.right_shift(x1._array, x2._array))
def bitwise_xor(x1: Array, x2: Array, /) -> Array:
"""
Array API compatible wrapper for :py:func:`np.bitwise_xor

```

```

152: (4)                                     See its docstring for more information.
153: (4)
154: (4)
155: (8)
156: (8)
157: (4)
158: (8)
bitwise_xor")
159: (4)
160: (4)
161: (4)
162: (0)
163: (4)
164: (4)
165: (4)
166: (4)
167: (4)
168: (8)
169: (4)
170: (8)
171: (4)
172: (0)
173: (4)
174: (4)
175: (4)
176: (4)
177: (4)
178: (8)
conj")
179: (4)
180: (0)
181: (4)
182: (4)
183: (4)
184: (4)
185: (4)
186: (8)
187: (4)
188: (0)
189: (4)
190: (4)
191: (4)
192: (4)
193: (4)
194: (8)
195: (4)
196: (0)
197: (4)
198: (4)
199: (4)
200: (4)
201: (4)
202: (8)
203: (4)
204: (4)
205: (4)
206: (0)
207: (4)
208: (4)
209: (4)
210: (4)
211: (4)
212: (4)
213: (4)
214: (0)
215: (4)
216: (4)
217: (4)
218: (4)

    if (
        x1.dtype not in _integer_or_boolean_dtypes
        or x2.dtype not in _integer_or_boolean_dtypes
    ):
        raise TypeError("Only integer or boolean dtypes are allowed in
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
return Array._new(np.bitwise_xor(x1._array, x2._array))

def ceil(x: Array, /) -> Array:
"""
    Array API compatible wrapper for :py:func:`np.ceil <numpy.ceil>`.
    See its docstring for more information.
"""
    if x.dtype not in _real_numeric_dtypes:
        raise TypeError("Only real numeric dtypes are allowed in ceil")
    if x.dtype in _integer_dtypes:
        return x
    return Array._new(np.ceil(x._array))

def conj(x: Array, /) -> Array:
"""
    Array API compatible wrapper for :py:func:`np.conj <numpy.conj>`.
    See its docstring for more information.
"""
    if x.dtype not in _complex_floating_dtypes:
        raise TypeError("Only complex floating-point dtypes are allowed in
conj")
    return Array._new(np.conj(x))

def cos(x: Array, /) -> Array:
"""
    Array API compatible wrapper for :py:func:`np.cos <numpy.cos>`.
    See its docstring for more information.
"""
    if x.dtype not in _floating_dtypes:
        raise TypeError("Only floating-point dtypes are allowed in cos")
    return Array._new(np.cos(x._array))

def cosh(x: Array, /) -> Array:
"""
    Array API compatible wrapper for :py:func:`np.cosh <numpy.cosh>`.
    See its docstring for more information.
"""
    if x.dtype not in _floating_dtypes:
        raise TypeError("Only floating-point dtypes are allowed in cosh")
    return Array._new(np.cosh(x._array))

def divide(x1: Array, x2: Array, /) -> Array:
"""
    Array API compatible wrapper for :py:func:`np.divide <numpy.divide>`.
    See its docstring for more information.
"""
    if x1.dtype not in _floating_dtypes or x2.dtype not in _floating_dtypes:
        raise TypeError("Only floating-point dtypes are allowed in divide")
    _result_type(x1.dtype, x2.dtype)
    x1, x2 = Array._normalize_two_args(x1, x2)
    return Array._new(np.divide(x1._array, x2._array))

def equal(x1: Array, x2: Array, /) -> Array:
"""
    Array API compatible wrapper for :py:func:`np.equal <numpy.equal>`.
    See its docstring for more information.
"""
    _result_type(x1.dtype, x2.dtype)
    x1, x2 = Array._normalize_two_args(x1, x2)
    return Array._new(np.equal(x1._array, x2._array))

def exp(x: Array, /) -> Array:
"""
    Array API compatible wrapper for :py:func:`np.exp <numpy.exp>`.
    See its docstring for more information.
"""

```

```

219: (4)             if x.dtype not in _floating_dtypes:
220: (8)                 raise TypeError("Only floating-point dtypes are allowed in exp")
221: (4)             return Array._new(np.exp(x._array))
222: (0)         def expm1(x: Array, /) -> Array:
223: (4)             """
224: (4)                 Array API compatible wrapper for :py:func:`np.expm1 <numpy.expm1>`.
225: (4)                 See its docstring for more information.
226: (4)
227: (4)                 if x.dtype not in _floating_dtypes:
228: (8)                     raise TypeError("Only floating-point dtypes are allowed in expm1")
229: (4)                 return Array._new(np.expm1(x._array))
230: (0)         def floor(x: Array, /) -> Array:
231: (4)             """
232: (4)                 Array API compatible wrapper for :py:func:`np.floor <numpy.floor>`.
233: (4)                 See its docstring for more information.
234: (4)
235: (4)                 if x.dtype not in _real_numeric_dtypes:
236: (8)                     raise TypeError("Only real numeric dtypes are allowed in floor")
237: (4)                 if x.dtype in _integer_dtypes:
238: (8)                     return x
239: (4)                 return Array._new(np.floor(x._array))
240: (0)         def floor_divide(x1: Array, x2: Array, /) -> Array:
241: (4)             """
242: (4)                 Array API compatible wrapper for :py:func:`np.floor_divide
<numpy.floor_divide>`.
243: (4)                 See its docstring for more information.
244: (4)
245: (4)                 if x1.dtype not in _real_numeric_dtypes or x2.dtype not in
246: (8)                     raise TypeError("Only real numeric dtypes are allowed in
floor_divide")
247: (4)
248: (4)                 _result_type(x1.dtype, x2.dtype)
249: (4)                 x1, x2 = Array._normalize_two_args(x1, x2)
250: (0)                 return Array._new(np.floor_divide(x1._array, x2._array))
251: (4)         def greater(x1: Array, x2: Array, /) -> Array:
252: (4)             """
253: (4)                 Array API compatible wrapper for :py:func:`np.greater <numpy.greater>`.
254: (4)                 See its docstring for more information.
255: (4)
256: (4)                 if x1.dtype not in _real_numeric_dtypes or x2.dtype not in
257: (8)                     raise TypeError("Only real numeric dtypes are allowed in greater")
258: (4)                     _result_type(x1.dtype, x2.dtype)
259: (4)                     x1, x2 = Array._normalize_two_args(x1, x2)
260: (0)                     return Array._new(np.greater(x1._array, x2._array))
261: (4)         def greater_equal(x1: Array, x2: Array, /) -> Array:
262: (4)             """
263: (4)                 Array API compatible wrapper for :py:func:`np.greater_equal
<numpy.greater_equal>`.
264: (4)                 See its docstring for more information.
265: (4)
266: (4)                 if x1.dtype not in _real_numeric_dtypes or x2.dtype not in
267: (8)                     raise TypeError("Only real numeric dtypes are allowed in
greater_equal")
268: (4)                     _result_type(x1.dtype, x2.dtype)
269: (4)                     x1, x2 = Array._normalize_two_args(x1, x2)
270: (0)                     return Array._new(np.greater_equal(x1._array, x2._array))
271: (4)         def imag(x: Array, /) -> Array:
272: (4)             """
273: (4)                 Array API compatible wrapper for :py:func:`np.imag <numpy.imag>`.
274: (4)                 See its docstring for more information.
275: (4)
276: (4)                 if x.dtype not in _complex_floating_dtypes:
277: (8)                     raise TypeError("Only complex floating-point dtypes are allowed in
imag")
278: (0)                 return Array._new(np.imag(x))
279: (4)         def isfinite(x: Array, /) -> Array:
279: (4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

280: (4)           Array API compatible wrapper for :py:func:`np.isfinite <numpy.isfinite>` .
281: (4)           See its docstring for more information.
282: (4)
283: (4)           if x.dtype not in _numeric_dtypes:
284: (8)             raise TypeError("Only numeric dtypes are allowed in isfinite")
285: (4)             return Array._new(np.isfinite(x._array))
286: (0) def isinf(x: Array, /) -> Array:
287: (4)
288: (4)           Array API compatible wrapper for :py:func:`np.isinf <numpy.isinf>` .
289: (4)           See its docstring for more information.
290: (4)
291: (4)           if x.dtype not in _numeric_dtypes:
292: (8)             raise TypeError("Only numeric dtypes are allowed in isinf")
293: (4)             return Array._new(np.isinf(x._array))
294: (0) def isnan(x: Array, /) -> Array:
295: (4)
296: (4)           Array API compatible wrapper for :py:func:`np.isnan <numpy.isnan>` .
297: (4)           See its docstring for more information.
298: (4)
299: (4)           if x.dtype not in _numeric_dtypes:
300: (8)             raise TypeError("Only numeric dtypes are allowed in isnan")
301: (4)             return Array._new(np.isnan(x._array))
302: (0) def less(x1: Array, x2: Array, /) -> Array:
303: (4)
304: (4)           Array API compatible wrapper for :py:func:`np.less <numpy.less>` .
305: (4)           See its docstring for more information.
306: (4)
307: (4)           if x1.dtype not in _real_numeric_dtypes or x2.dtype not in
308: (8)             _real_numeric_dtypes:
309: (4)               raise TypeError("Only real numeric dtypes are allowed in less")
310: (4)               _result_type(x1.dtype, x2.dtype)
311: (4)               x1, x2 = Array._normalize_two_args(x1, x2)
312: (0)               return Array._new(np.less(x1._array, x2._array))
313: (4) def less_equal(x1: Array, x2: Array, /) -> Array:
314: (4)
315: (4)           Array API compatible wrapper for :py:func:`np.less_equal
316: (4)             See its docstring for more information.
317: (4)
318: (4)           if x1.dtype not in _real_numeric_dtypes or x2.dtype not in
319: (8)             _real_numeric_dtypes:
320: (4)               raise TypeError("Only real numeric dtypes are allowed in less_equal")
321: (4)               _result_type(x1.dtype, x2.dtype)
322: (0)               x1, x2 = Array._normalize_two_args(x1, x2)
323: (4)               return Array._new(np.less_equal(x1._array, x2._array))
324: (4) def log(x: Array, /) -> Array:
325: (4)
326: (4)           Array API compatible wrapper for :py:func:`np.log <numpy.log>` .
327: (4)           See its docstring for more information.
328: (4)
329: (4)           if x.dtype not in _floating_dtypes:
330: (8)             raise TypeError("Only floating-point dtypes are allowed in log")
331: (4)             return Array._new(np.log(x._array))
332: (0) def log1p(x: Array, /) -> Array:
333: (4)
334: (4)           Array API compatible wrapper for :py:func:`np.log1p <numpy.log1p>` .
335: (4)           See its docstring for more information.
336: (4)
337: (4)           if x.dtype not in _floating_dtypes:
338: (8)             raise TypeError("Only floating-point dtypes are allowed in log1p")
339: (4)             return Array._new(np.log1p(x._array))
340: (0) def log2(x: Array, /) -> Array:
341: (4)
342: (4)           Array API compatible wrapper for :py:func:`np.log2 <numpy.log2>` .
343: (4)           See its docstring for more information.
344: (4)
345: (4)           if x.dtype not in _floating_dtypes:
346: (8)             raise TypeError("Only floating-point dtypes are allowed in log2")
347: (4)             return Array._new(np.log2(x._array))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

346: (0)
347: (4)
348: (4)
349: (4)
350: (4)
351: (4)
352: (8)
353: (4)
354: (0)
355: (4)
356: (4)
<numpy.logaddexp>`.
357: (4)
358: (4)
359: (4)
_real_floating_dtypes:
360: (8)
logaddexp")
361: (4)
362: (4)
363: (4)
364: (0)
365: (4)
366: (4)
<numpy.logical_and>`.
367: (4)
368: (4)
369: (4)
370: (8)
371: (4)
372: (4)
373: (4)
374: (0)
375: (4)
376: (4)
<numpy.logical_not>`.
377: (4)
378: (4)
379: (4)
380: (8)
381: (4)
382: (0)
383: (4)
384: (4)
<numpy.logical_or>`.
385: (4)
386: (4)
387: (4)
388: (8)
389: (4)
390: (4)
391: (4)
392: (0)
393: (4)
394: (4)
<numpy.logical_xor>`.
395: (4)
396: (4)
397: (4)
398: (8)
399: (4)
400: (4)
401: (4)
402: (0)
403: (4)
404: (4)
405: (4)
406: (4)
407: (4)

def log10(x: Array, /) -> Array:
    """
        Array API compatible wrapper for :py:func:`np.log10 <numpy.log10>`.
        See its docstring for more information.
    """
    if x.dtype not in _floating_dtypes:
        raise TypeError("Only floating-point dtypes are allowed in log10")
    return Array._new(np.log10(x._array))

def logaddexp(x1: Array, x2: Array) -> Array:
    """
        Array API compatible wrapper for :py:func:`np.logaddexp
        See its docstring for more information.
    """
    if x1.dtype not in _real_floating_dtypes or x2.dtype not in
_real_floating_dtypes:
        raise TypeError("Only real floating-point dtypes are allowed in
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
return Array._new(np.logaddexp(x1._array, x2._array))

def logical_and(x1: Array, x2: Array, /) -> Array:
    """
        Array API compatible wrapper for :py:func:`np.logical_and
        See its docstring for more information.
    """
    if x1.dtype not in _boolean_dtypes or x2.dtype not in _boolean_dtypes:
        raise TypeError("Only boolean dtypes are allowed in logical_and")
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
return Array._new(np.logical_and(x1._array, x2._array))

def logical_not(x: Array, /) -> Array:
    """
        Array API compatible wrapper for :py:func:`np.logical_not
        See its docstring for more information.
    """
    if x.dtype not in _boolean_dtypes:
        raise TypeError("Only boolean dtypes are allowed in logical_not")
return Array._new(np.logical_not(x._array))

def logical_or(x1: Array, x2: Array, /) -> Array:
    """
        Array API compatible wrapper for :py:func:`np.logical_or
        See its docstring for more information.
    """
    if x1.dtype not in _boolean_dtypes or x2.dtype not in _boolean_dtypes:
        raise TypeError("Only boolean dtypes are allowed in logical_or")
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
return Array._new(np.logical_or(x1._array, x2._array))

def logical_xor(x1: Array, x2: Array, /) -> Array:
    """
        Array API compatible wrapper for :py:func:`np.logical_xor
        See its docstring for more information.
    """
    if x1.dtype not in _boolean_dtypes or x2.dtype not in _boolean_dtypes:
        raise TypeError("Only boolean dtypes are allowed in logical_xor")
_result_type(x1.dtype, x2.dtype)
x1, x2 = Array._normalize_two_args(x1, x2)
return Array._new(np.logical_xor(x1._array, x2._array))

def multiply(x1: Array, x2: Array, /) -> Array:
    """
        Array API compatible wrapper for :py:func:`np.multiply <numpy.multiply>`.
        See its docstring for more information.
    """
    if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:

```

```

408: (8)           raise TypeError("Only numeric dtypes are allowed in multiply")
409: (4)           _result_type(x1.dtype, x2.dtype)
410: (4)           x1, x2 = Array._normalize_two_args(x1, x2)
411: (4)           return Array._new(np.multiply(x1._array, x2._array))
412: (0)           def negative(x: Array, /) -> Array:
413: (4)               """
414: (4)                   Array API compatible wrapper for :py:func:`np.negative` <numpy.negative>.
415: (4)                   See its docstring for more information.
416: (4)
417: (4)               if x.dtype not in _numeric_dtypes:
418: (8)                   raise TypeError("Only numeric dtypes are allowed in negative")
419: (4)                   return Array._new(np.negative(x._array))
420: (0)           def not_equal(x1: Array, x2: Array, /) -> Array:
421: (4)               """
422: (4)                   Array API compatible wrapper for :py:func:`np.not_equal`
<numpy.not_equal>.
423: (4)                   See its docstring for more information.
424: (4)
425: (4)               _result_type(x1.dtype, x2.dtype)
426: (4)               x1, x2 = Array._normalize_two_args(x1, x2)
427: (4)               return Array._new(np.not_equal(x1._array, x2._array))
428: (0)           def positive(x: Array, /) -> Array:
429: (4)               """
430: (4)                   Array API compatible wrapper for :py:func:`np.positive` <numpy.positive>.
431: (4)                   See its docstring for more information.
432: (4)
433: (4)               if x.dtype not in _numeric_dtypes:
434: (8)                   raise TypeError("Only numeric dtypes are allowed in positive")
435: (4)                   return Array._new(np.positive(x._array))
436: (0)           def pow(x1: Array, x2: Array, /) -> Array:
437: (4)               """
438: (4)                   Array API compatible wrapper for :py:func:`np.power` <numpy.power>.
439: (4)                   See its docstring for more information.
440: (4)
441: (4)               if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:
442: (8)                   raise TypeError("Only numeric dtypes are allowed in pow")
443: (4)               _result_type(x1.dtype, x2.dtype)
444: (4)               x1, x2 = Array._normalize_two_args(x1, x2)
445: (4)               return Array._new(np.power(x1._array, x2._array))
446: (0)           def real(x: Array, /) -> Array:
447: (4)               """
448: (4)                   Array API compatible wrapper for :py:func:`np.real` <numpy.real>.
449: (4)                   See its docstring for more information.
450: (4)
451: (4)               if x.dtype not in _complex_floating_dtypes:
452: (8)                   raise TypeError("Only complex floating-point dtypes are allowed in
real")
453: (4)               return Array._new(np.real(x))
454: (0)           def remainder(x1: Array, x2: Array, /) -> Array:
455: (4)
456: (4)               Array API compatible wrapper for :py:func:`np.remainder
<numpy.remainder>`.
457: (4)               See its docstring for more information.
458: (4)
459: (4)               if x1.dtype not in _real_numeric_dtypes or x2.dtype not in
_real_numeric_dtypes:
460: (8)                   raise TypeError("Only real numeric dtypes are allowed in remainder")
461: (4)               _result_type(x1.dtype, x2.dtype)
462: (4)               x1, x2 = Array._normalize_two_args(x1, x2)
463: (4)               return Array._new(np.remainder(x1._array, x2._array))
464: (0)           def round(x: Array, /) -> Array:
465: (4)               """
466: (4)                   Array API compatible wrapper for :py:func:`np.round` <numpy.round>.
467: (4)                   See its docstring for more information.
468: (4)
469: (4)               if x.dtype not in _numeric_dtypes:
470: (8)                   raise TypeError("Only numeric dtypes are allowed in round")
471: (4)                   return Array._new(np.round(x._array))
472: (0)           def sign(x: Array, /) -> Array:

```

```

473: (4)
474: (4)    """
475: (4)        Array API compatible wrapper for :py:func:`np.sign <numpy.sign>` .
476: (4)        See its docstring for more information.
477: (4)    """
478: (8)        if x.dtype not in _numeric_dtypes:
479: (4)            raise TypeError("Only numeric dtypes are allowed in sign")
480: (0)        return Array._new(np.sign(x._array))
481: (4)    def sin(x: Array, /) -> Array:
482: (4)        """
483: (4)            Array API compatible wrapper for :py:func:`np.sin <numpy.sin>` .
484: (4)            See its docstring for more information.
485: (4)        """
486: (8)            if x.dtype not in _floating_dtypes:
487: (4)                raise TypeError("Only floating-point dtypes are allowed in sin")
488: (0)            return Array._new(np.sin(x._array))
489: (4)    def sinh(x: Array, /) -> Array:
490: (4)        """
491: (4)            Array API compatible wrapper for :py:func:`np.sinh <numpy.sinh>` .
492: (4)            See its docstring for more information.
493: (4)        """
494: (8)            if x.dtype not in _floating_dtypes:
495: (4)                raise TypeError("Only floating-point dtypes are allowed in sinh")
496: (0)            return Array._new(np.sinh(x._array))
497: (4)    def square(x: Array, /) -> Array:
498: (4)        """
499: (4)            Array API compatible wrapper for :py:func:`np.square <numpy.square>` .
500: (4)            See its docstring for more information.
501: (4)        """
502: (8)            if x.dtype not in _numeric_dtypes:
503: (4)                raise TypeError("Only numeric dtypes are allowed in square")
504: (0)            return Array._new(np.square(x._array))
505: (4)    def sqrt(x: Array, /) -> Array:
506: (4)        """
507: (4)            Array API compatible wrapper for :py:func:`np.sqrt <numpy.sqrt>` .
508: (4)            See its docstring for more information.
509: (4)        """
510: (8)            if x.dtype not in _floating_dtypes:
511: (4)                raise TypeError("Only floating-point dtypes are allowed in sqrt")
512: (0)            return Array._new(np.sqrt(x._array))
513: (4)    def subtract(x1: Array, x2: Array, /) -> Array:
514: (4)        """
515: (4)            Array API compatible wrapper for :py:func:`np.subtract <numpy.subtract>` .
516: (4)            See its docstring for more information.
517: (4)        """
518: (8)            if x1.dtype not in _numeric_dtypes or x2.dtype not in _numeric_dtypes:
519: (4)                raise TypeError("Only numeric dtypes are allowed in subtract")
520: (4)            _result_type(x1.dtype, x2.dtype)
521: (4)            x1, x2 = Array._normalize_two_args(x1, x2)
522: (0)            return Array._new(np.subtract(x1._array, x2._array))
523: (4)    def tan(x: Array, /) -> Array:
524: (4)        """
525: (4)            Array API compatible wrapper for :py:func:`np.tan <numpy.tan>` .
526: (4)            See its docstring for more information.
527: (4)        """
528: (8)            if x.dtype not in _floating_dtypes:
529: (4)                raise TypeError("Only floating-point dtypes are allowed in tan")
530: (0)            return Array._new(np.tan(x._array))
531: (4)    def tanh(x: Array, /) -> Array:
532: (4)        """
533: (4)            Array API compatible wrapper for :py:func:`np.tanh <numpy.tanh>` .
534: (4)            See its docstring for more information.
535: (4)        """
536: (8)            if x.dtype not in _floating_dtypes:
537: (4)                raise TypeError("Only floating-point dtypes are allowed in tanh")
538: (0)            return Array._new(np.tanh(x._array))
539: (4)    def trunc(x: Array, /) -> Array:
540: (4)        """
541: (4)            Array API compatible wrapper for :py:func:`np.trunc <numpy.trunc>` .

```

```

542: (4)
543: (4)        """
544: (8)        if x.dtype not in _real_numeric_dtypes:
545: (4)            raise TypeError("Only real numeric dtypes are allowed in trunc")
546: (8)        if x.dtype in _integer_dtypes:
547: (4)            return x
548: (4)        return Array._new(np.trunc(x._array))
-----
```

## File 20 - \_indexing\_functions.py:

```

1: (0)          from __future__ import annotations
2: (0)          from .array_object import Array
3: (0)          from ._dtypes import _integer_dtypes
4: (0)          import numpy as np
5: (0)          def take(x: Array, indices: Array, /, *, axis: Optional[int] = None) -> Array:
6: (4)            """
7: (4)                Array API compatible wrapper for :py:func:`np.take` <numpy.take>.
8: (4)                See its docstring for more information.
9: (4)            """
10: (4)           if axis is None and x.ndim != 1:
11: (8)             raise ValueError("axis must be specified when ndim > 1")
12: (4)           if indices.dtype not in _integer_dtypes:
13: (8)             raise TypeError("Only integer dtypes are allowed in indexing")
14: (4)           if indices.ndim != 1:
15: (8)             raise ValueError("Only 1-dim indices array is supported")
16: (4)           return Array._new(np.take(x._array, indices._array, axis=axis))
-----
```

## File 21 - \_manipulation\_functions.py:

```

1: (0)          from __future__ import annotations
2: (0)          from .array_object import Array
3: (0)          from ._data_type_functions import result_type
4: (0)          from typing import List, Optional, Tuple, Union
5: (0)          import numpy as np
6: (0)          def concat(
7: (4)            arrays: Union[Tuple[Array, ...], List[Array]], /, *, axis: Optional[int] = 0
8: (0)          ) -> Array:
9: (4)            """
10: (4)                Array API compatible wrapper for :py:func:`np.concatenate` <numpy.concatenate>.
11: (4)                See its docstring for more information.
12: (4)            """
13: (4)            dtype = result_type(*arrays)
14: (4)            arrays = tuple(a._array for a in arrays)
15: (4)            return Array._new(np.concatenate(arrays, axis=axis, dtype=dtype))
16: (0)          def expand_dims(x: Array, /, *, axis: int) -> Array:
17: (4)            """
18: (4)                Array API compatible wrapper for :py:func:`np.expand_dims` <numpy.expand_dims>.
19: (4)                See its docstring for more information.
20: (4)            """
21: (4)            return Array._new(np.expand_dims(x._array, axis))
22: (0)          def flip(x: Array, /, *, axis: Optional[Union[int, Tuple[int, ...]]] = None) -> Array:
23: (4)            """
24: (4)                Array API compatible wrapper for :py:func:`np.flip` <numpy.flip>.
25: (4)                See its docstring for more information.
26: (4)            """
27: (4)            return Array._new(np.flip(x._array, axis=axis))
28: (0)          def permute_dims(x: Array, /, axes: Tuple[int, ...]) -> Array:
29: (4)            """
30: (4)                Array API compatible wrapper for :py:func:`np.transpose` <numpy.transpose>.
31: (4)                See its docstring for more information.
32: (4)            """
-----
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

33: (4)          return Array._new(np.transpose(x._array, axes))
34: (0)          def reshape(x: Array,
35: (12)          '/',
36: (12)          shape: Tuple[int, ...],
37: (12)          *,
38: (12)          copy: Optional[Bool] = None) -> Array:
39: (4)          """
40: (4)          Array API compatible wrapper for :py:func:`np.reshape <numpy.reshape>` .
41: (4)          See its docstring for more information.
42: (4)          """
43: (4)          data = x._array
44: (4)          if copy:
45: (8)              data = np.copy(data)
46: (4)          reshaped = np.reshape(data, shape)
47: (4)          if copy is False and not np.shares_memory(data, reshaped):
48: (8)              raise AttributeError("Incompatible shape for in-place modification.")
49: (4)          return Array._new(reshaped)
50: (0)          def roll(
51: (4)          x: Array,
52: (4)          '/',
53: (4)          shift: Union[int, Tuple[int, ...]],
54: (4)          *,
55: (4)          axis: Optional[Union[int, Tuple[int, ...]]] = None,
56: (0)          ) -> Array:
57: (4)          """
58: (4)          Array API compatible wrapper for :py:func:`np.roll <numpy.roll>` .
59: (4)          See its docstring for more information.
60: (4)          """
61: (4)          return Array._new(np.roll(x._array, shift, axis=axis))
62: (0)          def squeeze(x: Array, /, axis: Union[int, Tuple[int, ...]]) -> Array:
63: (4)          """
64: (4)          Array API compatible wrapper for :py:func:`np.squeeze <numpy.squeeze>` .
65: (4)          See its docstring for more information.
66: (4)          """
67: (4)          return Array._new(np.squeeze(x._array, axis=axis))
68: (0)          def stack(arrays: Union[Tuple[Array, ...], List[Array]], /, *, axis: int = 0)
-> Array:
69: (4)          """
70: (4)          Array API compatible wrapper for :py:func:`np.stack <numpy.stack>` .
71: (4)          See its docstring for more information.
72: (4)          """
73: (4)          result_type(*arrays)
74: (4)          arrays = tuple(a._array for a in arrays)
75: (4)          return Array._new(np.stack(arrays, axis=axis))

```

---

File 22 - \_searching\_functions.py:

```

1: (0)          from __future__ import annotations
2: (0)          from ._array_object import Array
3: (0)          from ._dtypes import _result_type, _real_numeric_dtypes
4: (0)          from typing import Optional, Tuple
5: (0)          import numpy as np
6: (0)          def argmax(x: Array, /, *, axis: Optional[int] = None, keepdims: bool = False)
-> Array:
7: (4)          """
8: (4)          Array API compatible wrapper for :py:func:`np.argmax <numpy.argmax>` .
9: (4)          See its docstring for more information.
10: (4)          """
11: (4)          if x.dtype not in _real_numeric_dtypes:
12: (8)              raise TypeError("Only real numeric dtypes are allowed in argmax")
13: (4)          return Array._new(np.asarray(np.argmax(x._array, axis=axis,
keepdims=keepdims)))
14: (0)          def argmin(x: Array, /, *, axis: Optional[int] = None, keepdims: bool = False)
-> Array:
15: (4)          """
16: (4)          Array API compatible wrapper for :py:func:`np.argmin <numpy.argmin>` .
17: (4)          See its docstring for more information.

```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

18: (4)
19: (4)
20: (8)
21: (4)
keepdims=keepdims)))
22: (0)    def nonzero(x: Array, /) -> Tuple[Array, ...]:
23: (4)
24: (4)        """
25: (4)            Array API compatible wrapper for :py:func:`np.nonzero <numpy.nonzero>` .
26: (4)            See its docstring for more information.
27: (4)        """
28: (0)    def where(condition: Array, x1: Array, x2: Array, /) -> Array:
29: (4)
30: (4)        """
31: (4)            Array API compatible wrapper for :py:func:`np.where <numpy.where>` .
32: (4)            See its docstring for more information.
33: (4)        """
34: (4)        _result_type(x1.dtype, x2.dtype)
35: (4)        x1, x2 = Array._normalize_two_args(x1, x2)
36: (4)        return Array._new(np.where(condition._array, x1._array, x2._array))
```

-----  
File 23 - \_set\_functions.py:

```
1: (0)    from __future__ import annotations
2: (0)    from ._array_object import Array
3: (0)    from typing import NamedTuple
4: (0)    import numpy as np
5: (0)    class UniqueAllResult(NamedTuple):
6: (4)        values: Array
7: (4)        indices: Array
8: (4)        inverse_indices: Array
9: (4)        counts: Array
10: (0)    class UniqueCountsResult(NamedTuple):
11: (4)        values: Array
12: (4)        counts: Array
13: (0)    class UniqueInverseResult(NamedTuple):
14: (4)        values: Array
15: (4)        inverse_indices: Array
16: (0)    def unique_all(x: Array, /) -> UniqueAllResult:
17: (4)
18: (4)        """
19: (4)            Array API compatible wrapper for :py:func:`np.unique <numpy.unique>` .
20: (4)            See its docstring for more information.
21: (4)        """
22: (8)        values, indices, inverse_indices, counts = np.unique(
23: (8)            x._array,
24: (8)            return_counts=True,
25: (8)            return_index=True,
26: (8)            return_inverse=True,
27: (8)            equal_nan=False,
28: (4)        )
29: (4)        inverse_indices = inverse_indices.reshape(x.shape)
30: (8)        return UniqueAllResult(
31: (8)            Array._new(values),
32: (8)            Array._new(indices),
33: (8)            Array._new(inverse_indices),
34: (8)            Array._new(counts),
35: (4)        )
36: (0)    def unique_counts(x: Array, /) -> UniqueCountsResult:
37: (4)        res = np.unique(
38: (8)            x._array,
39: (8)            return_counts=True,
40: (8)            return_index=False,
41: (8)            return_inverse=False,
42: (8)            equal_nan=False,
43: (4)        )
44: (0)        return UniqueCountsResult(*[Array._new(i) for i in res])
45: (4)    def unique_inverse(x: Array, /) -> UniqueInverseResult:
46: (4)        """
```

```

46: (4)             Array API compatible wrapper for :py:func:`np.unique <numpy.unique>` .
47: (4)             See its docstring for more information.
48: (4)
49: (4)             """
50: (8)             values, inverse_indices = np.unique(
51: (8)                 x._array,
52: (8)                 return_counts=False,
53: (8)                 return_index=False,
54: (8)                 return_inverse=True,
55: (8)                 equal_nan=False,
56: (4)             )
57: (4)             inverse_indices = inverse_indices.reshape(x.shape)
58: (4)             return UniqueInverseResult(Array._new(values),
59: (4)             Array._new(inverse_indices))
59: (0)             def unique_values(x: Array, /) -> Array:
60: (4)                 """
61: (4)                     Array API compatible wrapper for :py:func:`np.unique <numpy.unique>` .
62: (4)                     See its docstring for more information.
63: (4)
64: (4)                     """
65: (4)                     res = np.unique(
66: (8)                         x._array,
67: (8)                         return_counts=False,
68: (8)                         return_index=False,
69: (8)                         return_inverse=False,
70: (8)                         equal_nan=False,
70: (4)                     )
70: (4)                     return Array._new(res)

```

-----

## File 24 - \_sorting\_functions.py:

```

1: (0)             from __future__ import annotations
2: (0)             from .array_object import Array
3: (0)             from .dtypes import _real_numeric_dtypes
4: (0)             import numpy as np
5: (0)             def argsort(
6: (4)                 x: Array, /, *, axis: int = -1, descending: bool = False, stable: bool =
True
7: (0)             ) -> Array:
8: (4)                 """
9: (4)                     Array API compatible wrapper for :py:func:`np.argsort <numpy.argsort>` .
10: (4)                     See its docstring for more information.
11: (4)                     """
12: (4)                     if x.dtype not in _real_numeric_dtypes:
13: (8)                         raise TypeError("Only real numeric dtypes are allowed in argsort")
14: (4)                     kind = "stable" if stable else "quicksort"
15: (4)                     if not descending:
16: (8)                         res = np.argsort(x._array, axis=axis, kind=kind)
17: (4)                     else:
18: (8)                         res = np.flip(
19: (12)                             np.argsort(np.flip(x._array, axis=axis), axis=axis, kind=kind),
20: (12)                             axis=axis,
21: (8)                         )
22: (8)                         normalised_axis = axis if axis >= 0 else x.ndim + axis
23: (8)                         max_i = x.shape[normalised_axis] - 1
24: (8)                         res = max_i - res
25: (4)                     return Array._new(res)
26: (0)             def sort(
27: (4)                 x: Array, /, *, axis: int = -1, descending: bool = False, stable: bool =
True
28: (0)             ) -> Array:
29: (4)                 """
30: (4)                     Array API compatible wrapper for :py:func:`np.sort <numpy.sort>` .
31: (4)                     See its docstring for more information.
32: (4)                     """
33: (4)                     if x.dtype not in _real_numeric_dtypes:
34: (8)                         raise TypeError("Only real numeric dtypes are allowed in sort")
35: (4)                     kind = "stable" if stable else "quicksort"
36: (4)                     res = np.sort(x._array, axis=axis, kind=kind)

```

```

37: (4)         if descending:
38: (8)             res = np.flip(res, axis=axis)
39: (4)         return Array._new(res)
-----
```

## File 25 - \_typing.py:

```

1: (0)         """
2: (0)             This file defines the types for type annotations.
3: (0)             These names aren't part of the module namespace, but they are used in the
4: (0)             annotations in the function signatures. The functions in the module are only
5: (0)             valid for inputs that match the given type annotations.
6: (0)         """
7: (0)         from __future__ import annotations
8: (0)         __all__ = [
9: (4)             "Array",
10: (4)             "Device",
11: (4)             "Dtype",
12: (4)             "SupportsDLPack",
13: (4)             "SupportsBufferProtocol",
14: (4)             "PyCapsule",
15: (0)         ]
16: (0)         import sys
17: (0)         from typing import (
18: (4)             Any,
19: (4)             Literal,
20: (4)             Sequence,
21: (4)             Type,
22: (4)             Union,
23: (4)             TypeVar,
24: (4)             Protocol,
25: (0)         )
26: (0)         from .array_object import Array
27: (0)         from numpy import (
28: (4)             dtype,
29: (4)             int8,
30: (4)             int16,
31: (4)             int32,
32: (4)             int64,
33: (4)             uint8,
34: (4)             uint16,
35: (4)             uint32,
36: (4)             uint64,
37: (4)             float32,
38: (4)             float64,
39: (0)         )
40: (0)         _T_co = TypeVar("_T_co", covariant=True)
41: (0)         class NestedSequence(Protocol[_T_co]):
42: (4)             def __getitem__(self, key: int, /) -> _T_co | NestedSequence[_T_co]: ...
43: (4)             def __len__(self, /) -> int: ...
44: (0)             Device = Literal["cpu"]
45: (0)             Dtype = dtype[Union[
46: (4)                 int8,
47: (4)                 int16,
48: (4)                 int32,
49: (4)                 int64,
50: (4)                 uint8,
51: (4)                 uint16,
52: (4)                 uint32,
53: (4)                 uint64,
54: (4)                 float32,
55: (4)                 float64,
56: (0)             ]]
57: (0)             if sys.version_info >= (3, 12):
58: (4)                 from collections.abc import Buffer as SupportsBufferProtocol
59: (0)             else:
60: (4)                 SupportsBufferProtocol = Any
61: (0)             PyCapsule = Any

```

```
62: (0)         class SupportsDLPack(Protocol):
63: (4)             def __delpack__(self, /, *, stream: None = ...) -> PyCapsule: ...
-----
```

## File 26 - \_statistical\_functions.py:

```
1: (0)             from __future__ import annotations
2: (0)             from ._dtypes import (
3: (4)                 _real_floating_dtypes,
4: (4)                 _real_numeric_dtypes,
5: (4)                 _numeric_dtypes,
6: (0)             )
7: (0)             from ._array_object import Array
8: (0)             from ._dtypes import float32, float64, complex64, complex128
9: (0)             from typing import TYPE_CHECKING, Optional, Tuple, Union
10: (0)            if TYPE_CHECKING:
11: (4)                from ._typing import Dtype
12: (0)            import numpy as np
13: (0)            def max(
14: (4)                x: Array,
15: (4)                /,
16: (4)                *,
17: (4)                axis: Optional[Union[int, Tuple[int, ...]]] = None,
18: (4)                keepdims: bool = False,
19: (0)            ) -> Array:
20: (4)                if x.dtype not in _real_numeric_dtypes:
21: (8)                    raise TypeError("Only real numeric dtypes are allowed in max")
22: (4)                return Array._new(np.max(x._array, axis=axis, keepdims=keepdims))
23: (0)            def mean(
24: (4)                x: Array,
25: (4)                /,
26: (4)                *,
27: (4)                axis: Optional[Union[int, Tuple[int, ...]]] = None,
28: (4)                keepdims: bool = False,
29: (0)            ) -> Array:
30: (4)                if x.dtype not in _real_floating_dtypes:
31: (8)                    raise TypeError("Only real floating-point dtypes are allowed in mean")
32: (4)                return Array._new(np.mean(x._array, axis=axis, keepdims=keepdims))
33: (0)            def min(
34: (4)                x: Array,
35: (4)                /,
36: (4)                *,
37: (4)                axis: Optional[Union[int, Tuple[int, ...]]] = None,
38: (4)                keepdims: bool = False,
39: (0)            ) -> Array:
40: (4)                if x.dtype not in _real_numeric_dtypes:
41: (8)                    raise TypeError("Only real numeric dtypes are allowed in min")
42: (4)                return Array._new(np.min(x._array, axis=axis, keepdims=keepdims))
43: (0)            def prod(
44: (4)                x: Array,
45: (4)                /,
46: (4)                *,
47: (4)                axis: Optional[Union[int, Tuple[int, ...]]] = None,
48: (4)                dtype: Optional[Dtype] = None,
49: (4)                keepdims: bool = False,
50: (0)            ) -> Array:
51: (4)                if x.dtype not in _numeric_dtypes:
52: (8)                    raise TypeError("Only numeric dtypes are allowed in prod")
53: (4)                if dtype is None:
54: (8)                    if x.dtype == float32:
55: (12)                        dtype = float64
56: (8)                    elif x.dtype == complex64:
57: (12)                        dtype = complex128
58: (4)                    return Array._new(np.prod(x._array, dtype=dtype, axis=axis,
59: (0)                        keepdims=keepdims))
60: (4)            def std(
61: (4)                x: Array,
62: (4)                /,
```

```

62: (4)          *,
63: (4)          axis: Optional[Union[int, Tuple[int, ...]]] = None,
64: (4)          correction: Union[int, float] = 0.0,
65: (4)          keepdims: bool = False,
66: (0)          ) -> Array:
67: (4)          if x.dtype not in _real_floating_dtypes:
68: (8)              raise TypeError("Only real floating-point dtypes are allowed in std")
69: (4)          return Array._new(np.std(x._array, axis=axis, ddof=correction,
keepdims=keepdims))
70: (0)      def sum(
71: (4)          x: Array,
72: (4)          /,
73: (4)          *,
74: (4)          axis: Optional[Union[int, Tuple[int, ...]]] = None,
75: (4)          dtype: Optional[Dtype] = None,
76: (4)          keepdims: bool = False,
77: (0)          ) -> Array:
78: (4)          if x.dtype not in _numeric_dtotypes:
79: (8)              raise TypeError("Only numeric dtypes are allowed in sum")
80: (4)          if dtype is None:
81: (8)              if x.dtype == float32:
82: (12)                  dtype = float64
83: (8)              elif x.dtype == complex64:
84: (12)                  dtype = complex128
85: (4)          return Array._new(np.sum(x._array, axis=axis, dtype=dtype,
keepdims=keepdims))
86: (0)      def var(
87: (4)          x: Array,
88: (4)          /,
89: (4)          *,
90: (4)          axis: Optional[Union[int, Tuple[int, ...]]] = None,
91: (4)          correction: Union[int, float] = 0.0,
92: (4)          keepdims: bool = False,
93: (0)          ) -> Array:
94: (4)          if x.dtype not in _real_floating_dtypes:
95: (8)              raise TypeError("Only real floating-point dtypes are allowed in var")
96: (4)          return Array._new(np.var(x._array, axis=axis, ddof=correction,
keepdims=keepdims))
-----
```

## File 27 - \_\_init\_\_.py:

```

1: (0)      """
2: (0)      A NumPy sub-namespace that conforms to the Python array API standard.
3: (0)      This submodule accompanies NEP 47, which proposes its inclusion in NumPy. It
4: (0)      is still considered experimental, and will issue a warning when imported.
5: (0)      This is a proof-of-concept namespace that wraps the corresponding NumPy
6: (0)      functions to give a conforming implementation of the Python array API standard
7: (0)      (https://data-apis.github.io/array-api/latest/). The standard is currently in
8: (0)      an RFC phase and comments on it are both welcome and encouraged. Comments
9: (0)      should be made either at https://github.com/data-apis/array-api or at
10: (0)      https://github.com/data-apis/consortium-feedback/discussions.
11: (0)      NumPy already follows the proposed spec for the most part, so this module
12: (0)      serves mostly as a thin wrapper around it. However, NumPy also implements a
13: (0)      lot of behavior that is not included in the spec, so this serves as a
14: (0)      restricted subset of the API. Only those functions that are part of the spec
15: (0)      are included in this namespace, and all functions are given with the exact
16: (0)      signature given in the spec, including the use of position-only arguments, and
17: (0)      omitting any extra keyword arguments implemented by NumPy but not part of the
18: (0)      spec. The behavior of some functions is also modified from the NumPy behavior
19: (0)      to conform to the standard. Note that the underlying array object itself is
20: (0)      wrapped in a wrapper Array() class, but is otherwise unchanged. This submodule
21: (0)      is implemented in pure Python with no C extensions.
22: (0)      The array API spec is designed as a "minimal API subset" and explicitly allows
23: (0)      libraries to include behaviors not specified by it. But users of this module
24: (0)      that intend to write portable code should be aware that only those behaviors
25: (0)      that are listed in the spec are guaranteed to be implemented across libraries.
26: (0)      Consequently, the NumPy implementation was chosen to be both conforming and
```

```

27: (0) minimal, so that users can use this implementation of the array API namespace
28: (0) and be sure that behaviors that it defines will be available in conforming
29: (0) namespaces from other libraries.
30: (0) A few notes about the current state of this submodule:
31: (0) - There is a test suite that tests modules against the array API standard at
32: (2) https://github.com/data-apis/array-api-tests. The test suite is still a work
33: (2) in progress, but the existing tests pass on this module, with a few
34: (2) exceptions:
35: (2) - DLPack support (see https://github.com/data-apis/array-api/pull/106) is
36: (4) not included here, as it requires a full implementation in NumPy proper
37: (4) first.
38: (2) The test suite is not yet complete, and even the tests that exist are not
39: (2) guaranteed to give a comprehensive coverage of the spec. Therefore, when
40: (2) reviewing and using this submodule, you should refer to the standard
41: (2) documents themselves. There are some tests in numpy.array_api.tests, but
42: (2) they primarily focus on things that are not tested by the official array API
43: (2) test suite.
44: (0) - There is a custom array object, numpy.array_api.Array, which is returned by
45: (2) all functions in this module. All functions in the array API namespace
46: (2) implicitly assume that they will only receive this object as input. The only
47: (2) way to create instances of this object is to use one of the array creation
48: (2) functions. It does not have a public constructor on the object itself. The
49: (2) object is a small wrapper class around numpy.ndarray. The main purpose of it
50: (2) is to restrict the namespace of the array object to only those dtypes and
51: (2) only those methods that are required by the spec, as well as to limit/change
52: (2) certain behavior that differs in the spec. In particular:
53: (2) - The array API namespace does not have scalar objects, only 0-D arrays.
54: (4) Operations on Array that would create a scalar in NumPy create a 0-D
55: (4) array.
56: (2) - Indexing: Only a subset of indices supported by NumPy are required by the
57: (4) spec. The Array object restricts indexing to only allow those types of
58: (4) indices that are required by the spec. See the docstring of the
59: (4) numpy.array_api.Array._validate_indices helper function for more
60: (4) information.
61: (2) - Type promotion: Some type promotion rules are different in the spec. In
62: (4) particular, the spec does not have any value-based casting. The spec also
63: (4) does not require cross-kind casting, like integer -> floating-point. Only
64: (4) those promotions that are explicitly required by the array API
65: (4) specification are allowed in this module. See NEP 47 for more info.
66: (2) - Functions do not automatically call asarray() on their input, and will not
67: (4) work if the input type is not Array. The exception is array creation
68: (4) functions, and Python operators on the Array object, which accept Python
69: (4) scalars of the same type as the array dtype.
70: (0) - All functions include type annotations, corresponding to those given in the
71: (2) spec (see _typing.py for definitions of some custom types). These do not
72: (2) currently fully pass mypy due to some limitations in mypy.
73: (0) - Dtype objects are just the NumPy dtype objects, e.g., float64 =
74: (2) np.dtype('float64'). The spec does not require any behavior on these dtype
75: (2) objects other than that they be accessible by name and be comparable by
76: (2) equality, but it was considered too much extra complexity to create custom
77: (2) objects to represent dtypes.
78: (0) - All places where the implementations in this submodule are known to deviate
79: (2) from their corresponding functions in NumPy are marked with "# Note:"
80: (2) comments.
81: (0) Still TODO in this module are:
82: (0) - DLPack support for numpy.ndarray is still in progress. See
83: (2) https://github.com/numpy/numpy/pull/19083.
84: (0) - The copy=False keyword argument to asarray() is not yet implemented. This
85: (2) requires support in numpy.asarray() first.
86: (0) - Some functions are not yet fully tested in the array API test suite, and may
87: (2) require updates that are not yet known until the tests are written.
88: (0) - The spec is still in an RFC phase and may still have minor updates, which
89: (2) will need to be reflected here.
90: (0) - Complex number support in array API spec is planned but not yet finalized,
91: (2) as are the fft extension and certain linear algebra functions such as eig
92: (2) that require complex dtypes.
93: (0)
94: (0) """
95: (0) import warnings
         warnings.warn(

```

```
96: (4)          "The numpy.array_api submodule is still experimental. See NEP 47.",  
stacklevel=2  
97: (0)          )  
98: (0)          __array_api_version__ = "2022.12"  
99: (0)          __all__ = ["__array_api_version__"]  
100: (0)         from .constants import e, inf, nan, pi, newaxis  
101: (0)         __all__ += ["e", "inf", "nan", "pi", "newaxis"]  
102: (0)         from .creation_functions import (  
103: (4)           asarray,  
104: (4)           arange,  
105: (4)           empty,  
106: (4)           empty_like,  
107: (4)           eye,  
108: (4)           from_dlpack,  
109: (4)           full,  
110: (4)           full_like,  
111: (4)           linspace,  
112: (4)           meshgrid,  
113: (4)           ones,  
114: (4)           ones_like,  
115: (4)           tril,  
116: (4)           triu,  
117: (4)           zeros,  
118: (4)           zeros_like,  
119: (0)          )  
120: (0)          __all__ += [  
121: (4)            "asarray",  
122: (4)            "arange",  
123: (4)            "empty",  
124: (4)            "empty_like",  
125: (4)            "eye",  
126: (4)            "from_dlpack",  
127: (4)            "full",  
128: (4)            "full_like",  
129: (4)            "linspace",  
130: (4)            "meshgrid",  
131: (4)            "ones",  
132: (4)            "ones_like",  
133: (4)            "tril",  
134: (4)            "triu",  
135: (4)            "zeros",  
136: (4)            "zeros_like",  
137: (0)          ]  
138: (0)          from ._data_type_functions import (  
139: (4)            astype,  
140: (4)            broadcast_arrays,  
141: (4)            broadcast_to,  
142: (4)            can_cast,  
143: (4)            finfo,  
144: (4)            isdtype,  
145: (4)            iinfo,  
146: (4)            result_type,  
147: (0)          )  
148: (0)          __all__ += [  
149: (4)            "astype",  
150: (4)            "broadcast_arrays",  
151: (4)            "broadcast_to",  
152: (4)            "can_cast",  
153: (4)            "finfo",  
154: (4)            "iinfo",  
155: (4)            "result_type",  
156: (0)          ]  
157: (0)          from ._dtypes import (  
158: (4)            int8,  
159: (4)            int16,  
160: (4)            int32,  
161: (4)            int64,  
162: (4)            uint8,  
163: (4)            uint16,
```

```
164: (4)          uint32,
165: (4)          uint64,
166: (4)          float32,
167: (4)          float64,
168: (4)          complex64,
169: (4)          complex128,
170: (4)          bool,
171: (0)
172: (0)      __all__ += [
173: (4)          "int8",
174: (4)          "int16",
175: (4)          "int32",
176: (4)          "int64",
177: (4)          "uint8",
178: (4)          "uint16",
179: (4)          "uint32",
180: (4)          "uint64",
181: (4)          "float32",
182: (4)          "float64",
183: (4)          "bool",
184: (0)
185: (0)      from ._elementwise_functions import (
186: (4)          abs,
187: (4)          acos,
188: (4)          acosh,
189: (4)          add,
190: (4)          asin,
191: (4)          asinh,
192: (4)          atan,
193: (4)          atan2,
194: (4)          atanh,
195: (4)          bitwise_and,
196: (4)          bitwise_left_shift,
197: (4)          bitwise_invert,
198: (4)          bitwise_or,
199: (4)          bitwise_right_shift,
200: (4)          bitwise_xor,
201: (4)          ceil,
202: (4)          conj,
203: (4)          cos,
204: (4)          cosh,
205: (4)          divide,
206: (4)          equal,
207: (4)          exp,
208: (4)          expm1,
209: (4)          floor,
210: (4)          floor_divide,
211: (4)          greater,
212: (4)          greater_equal,
213: (4)          imag,
214: (4)          isfinite,
215: (4)          isnan,
216: (4)          isnan,
217: (4)          less,
218: (4)          less_equal,
219: (4)          log,
220: (4)          log1p,
221: (4)          log2,
222: (4)          log10,
223: (4)          logaddexp,
224: (4)          logical_and,
225: (4)          logical_not,
226: (4)          logical_or,
227: (4)          logical_xor,
228: (4)          multiply,
229: (4)          negative,
230: (4)          not_equal,
231: (4)          positive,
232: (4)          pow,
```

```
233: (4)           real,
234: (4)           remainder,
235: (4)           round,
236: (4)           sign,
237: (4)           sin,
238: (4)           sinh,
239: (4)           square,
240: (4)           sqrt,
241: (4)           subtract,
242: (4)           tan,
243: (4)           tanh,
244: (4)           trunc,
245: (0)
246: (0)       __all__ += [
247: (4)           "abs",
248: (4)           "acos",
249: (4)           "acosh",
250: (4)           "add",
251: (4)           "asin",
252: (4)           "asinh",
253: (4)           "atan",
254: (4)           "atan2",
255: (4)           "atanh",
256: (4)           "bitwise_and",
257: (4)           "bitwise_left_shift",
258: (4)           "bitwise_invert",
259: (4)           "bitwise_or",
260: (4)           "bitwise_right_shift",
261: (4)           "bitwise_xor",
262: (4)           "ceil",
263: (4)           "cos",
264: (4)           "cosh",
265: (4)           "divide",
266: (4)           "equal",
267: (4)           "exp",
268: (4)           "expm1",
269: (4)           "floor",
270: (4)           "floor_divide",
271: (4)           "greater",
272: (4)           "greater_equal",
273: (4)           "isfinite",
274: (4)           "isinf",
275: (4)           "isnan",
276: (4)           "less",
277: (4)           "less_equal",
278: (4)           "log",
279: (4)           "log1p",
280: (4)           "log2",
281: (4)           "log10",
282: (4)           "logaddexp",
283: (4)           "logical_and",
284: (4)           "logical_not",
285: (4)           "logical_or",
286: (4)           "logical_xor",
287: (4)           "multiply",
288: (4)           "negative",
289: (4)           "not_equal",
290: (4)           "positive",
291: (4)           "pow",
292: (4)           "remainder",
293: (4)           "round",
294: (4)           "sign",
295: (4)           "sin",
296: (4)           "sinh",
297: (4)           "square",
298: (4)           "sqrt",
299: (4)           "subtract",
300: (4)           "tan",
301: (4)           "tanh",
```

```

302: (4)           "trunc",
303: (0)
304: (0)           ]
305: (0)           from ._indexing_functions import take
306: (0)           __all__ += ["take"]
307: (0)           from . import linalg
308: (0)           __all__ += ["linalg"]
309: (0)           from .linalg import matmul, tensordot, matrix_transpose, vecdot
310: (0)           __all__ += ["matmul", "tensordot", "matrix_transpose", "vecdot"]
311: (4)           from ._manipulation_functions import (
312: (4)             concat,
313: (4)             expand_dims,
314: (4)             flip,
315: (4)             permute_dims,
316: (4)             reshape,
317: (4)             roll,
318: (4)             squeeze,
319: (0)             stack,
320: (0)         )
321: (0)         __all__ += ["concat", "expand_dims", "flip", "permute_dims", "reshape",
322: (0)         "roll", "squeeze", "stack"]
323: (0)         from ._searching_functions import argmax, argmin, nonzero, where
324: (0)         __all__ += ["argmax", "argmin", "nonzero", "where"]
325: (0)         from ._set_functions import unique_all, unique_counts, unique_inverse,
326: (0)         unique_values
327: (0)         __all__ += ["unique_all", "unique_counts", "unique_inverse", "unique_values"]
328: (0)         from ._sorting_functions import argsort, sort
329: (0)         __all__ += ["argsort", "sort"]
330: (0)         from ._statistical_functions import max, mean, min, prod, std, sum, var
331: (0)         __all__ += ["max", "mean", "min", "prod", "std", "sum", "var"]
332: (0)         from ._utility_functions import all, any
333: (0)         __all__ += ["all", "any"]

```

-----

## File 28 - \_utility\_functions.py:

```

1: (0)             from __future__ import annotations
2: (0)             from ._array_object import Array
3: (0)             from typing import Optional, Tuple, Union
4: (0)             import numpy as np
5: (0)             def all(
6: (4)               x: Array,
7: (4)               /,
8: (4)               *,
9: (4)               axis: Optional[Union[int, Tuple[int, ...]]] = None,
10: (4)              keepdims: bool = False,
11: (0)            ) -> Array:
12: (4)              """
13: (4)                  Array API compatible wrapper for :py:func:`np.all <numpy.all>` .
14: (4)                  See its docstring for more information.
15: (4)              """
16: (4)              return Array._new(np.asarray(np.all(x._array, axis=axis,
keepdims=keepdims)))
17: (0)             def any(
18: (4)               x: Array,
19: (4)               /,
20: (4)               *,
21: (4)               axis: Optional[Union[int, Tuple[int, ...]]] = None,
22: (4)              keepdims: bool = False,
23: (0)            ) -> Array:
24: (4)              """
25: (4)                  Array API compatible wrapper for :py:func:`np.any <numpy.any>` .
26: (4)                  See its docstring for more information.
27: (4)              """
28: (4)              return Array._new(np.asarray(np.any(x._array, axis=axis,
keepdims=keepdims)))

```

-----

## File 29 - test\_array\_object.py:

```

1: (0)          import operator
2: (0)          from numpy.testing import assert_raises, suppress_warnings
3: (0)          import numpy as np
4: (0)          import pytest
5: (0)          from .. import ones, asarray, reshape, result_type, all, equal
6: (0)          from .._array_object import Array
7: (0)          from .._dtypes import (
8: (4)            _all_dtypes,
9: (4)            _boolean_dtypes,
10: (4)            _real_floating_dtypes,
11: (4)            _floating_dtypes,
12: (4)            _complex_floating_dtypes,
13: (4)            _integer_dtypes,
14: (4)            _integer_or_boolean_dtypes,
15: (4)            _real_numeric_dtypes,
16: (4)            _numeric_dtypes,
17: (4)            int8,
18: (4)            int16,
19: (4)            int32,
20: (4)            int64,
21: (4)            uint64,
22: (4)            bool as bool_,
23: (0)
24: (0)        def test_validate_index():
25: (4)            a = ones((3, 4))
26: (4)            assert_raises(IndexError, lambda: a[:4])
27: (4)            assert_raises(IndexError, lambda: a[:-4])
28: (4)            assert_raises(IndexError, lambda: a[3:-1])
29: (4)            assert_raises(IndexError, lambda: a[-5:-1])
30: (4)            assert_raises(IndexError, lambda: a[4:])
31: (4)            assert_raises(IndexError, lambda: a[-4:])
32: (4)            assert_raises(IndexError, lambda: a[4::-1])
33: (4)            assert_raises(IndexError, lambda: a[-4::-1])
34: (4)            assert_raises(IndexError, lambda: a[...,:5])
35: (4)            assert_raises(IndexError, lambda: a[...,:-5])
36: (4)            assert_raises(IndexError, lambda: a[...,:5:-1])
37: (4)            assert_raises(IndexError, lambda: a[...,:-6:-1])
38: (4)            assert_raises(IndexError, lambda: a[...,:5:])
39: (4)            assert_raises(IndexError, lambda: a[...,:-5:])
40: (4)            assert_raises(IndexError, lambda: a[...,:5::-1])
41: (4)            assert_raises(IndexError, lambda: a[...,:-5::-1])
42: (4)            assert_raises(IndexError, lambda: a[a[:,0]==1,0])
43: (4)            assert_raises(IndexError, lambda: a[a[:,0]==1,...])
44: (4)            assert_raises(IndexError, lambda: a[..., a[0]==1])
45: (4)            assert_raises(IndexError, lambda: a[[True, True, True]]))
46: (4)            assert_raises(IndexError, lambda: a[(True, True, True),]))
47: (4)            idx = asarray([[0, 1]])
48: (4)            assert_raises(IndexError, lambda: a[idx])
49: (4)            assert_raises(IndexError, lambda: a[idx,])
50: (4)            assert_raises(IndexError, lambda: a[[0, 1]])
51: (4)            assert_raises(IndexError, lambda: a[(0, 1), (0, 1)])
52: (4)            assert_raises(IndexError, lambda: a[[0, 1]])
53: (4)            assert_raises(IndexError, lambda: a[np.array([[0, 1]])))
54: (4)            assert_raises(IndexError, lambda: a[()])
55: (4)            assert_raises(IndexError, lambda: a[0,])
56: (4)            assert_raises(IndexError, lambda: a[0])
57: (4)            assert_raises(IndexError, lambda: a[:])
58: (0)
59: (4)        def test_operators():
60: (8)            binary_op_dtotypes = {
61: (8)              "__add__": "numeric",
62: (8)              "__and__": "integer_or_boolean",
63: (8)              "__eq__": "all",
64: (8)              "__floordiv__": "real numeric",
65: (8)              "__ge__": "real numeric",
66: (8)              "__gt__": "real numeric",
67: (8)              "__le__": "real numeric",
68: (8)              "__lshift__": "integer",

```

```

68: (8)             "__lt__": "real numeric",
69: (8)             "__mod__": "real numeric",
70: (8)             "__mul__": "numeric",
71: (8)             "__ne__": "all",
72: (8)             "__or__": "integer_or_boolean",
73: (8)             "__pow__": "numeric",
74: (8)             "__rshift__": "integer",
75: (8)             "__sub__": "numeric",
76: (8)             "__truediv__": "floating",
77: (8)             "__xor__": "integer_or_boolean",
78: (4)
79: (4)         }
80: (8)     def _array_vals():
81: (12)         for d in _integer_dtypes:
82: (8)             yield asarray(1, dtype=d)
83: (12)         for d in _boolean_dtypes:
84: (8)             yield asarray(False, dtype=d)
85: (12)         for d in _floating_dtypes:
86: (8)             yield asarray(1.0, dtype=d)
87: (4)         BIG_INT = int(1e30)
88: (4)         for op, dtypes in binary_op_dtypes.items():
89: (8)             ops = [op]
90: (12)             if op not in ["__eq__", "__ne__", "__le__", "__ge__", "__lt__",
91: (12)                 "__gt__"]:
92: (12)                 rop = "__r" + op[2:]
93: (8)                 iop = "__i" + op[2:]
94: (12)                 ops += [rop, iop]
95: (16)             for s in [1, 1.0, 1j, BIG_INT, False]:
96: (20)                 for _op in ops:
97: (25)                     for a in _array_vals():
98: (25)                         if ((dtypes == "all"
99: (25)                             or dtypes == "numeric" and a.dtype in _numeric_dtypes
100: (25)                            or dtypes == "real numeric" and a.dtype in
101: (25)                            _real_numeric_dtypes
102: (25)                            or dtypes == "integer" and a.dtype in _integer_dtypes
103: (24)                            or dtypes == "integer_or_boolean" and a.dtype in
104: (24)                            _integer_or_boolean_dtypes
105: (29)                            or dtypes == "boolean" and a.dtype in _boolean_dtypes
106: (29)                            or dtypes == "floating" and a.dtype in
107: (29)                            _floating_dtypes
108: (24)
109: (24)
110: (28)                         )
111: (24)                         and (a.dtype in _boolean_dtypes and type(s) == bool
112: (28)                             or a.dtype in _integer_dtypes and type(s) == int
113: (32)                             or a.dtype in _real_floating_dtypes and type(s)
114: (43)                             or a.dtype in _complex_floating_dtypes and
115: (32)                             type(s) in [complex, float, int]
116: (20)                         )):
117: (24)                         if a.dtype in _integer_dtypes and s == BIG_INT:
118: (16)                             assert_raises(OverflowError, lambda: getattr(a,
119: (20)                               _op)(s))
120: (24)
121: (28)
122: (32)                         else:
123: (32)                             with suppress_warnings() as sup:
124: (32)                               sup.filter(RuntimeWarning,
125: (32)                                   "invalid value encountered in
126: (32)                                   getattr(a, _op))(s)
127: (32)                         else:
128: (32)                             assert_raises(TypeError, lambda: getattr(a, _op))(s))
129: (32)                         for _op in ops:
130: (32)                             for x in _array_vals():
131: (32)                                 for y in _array_vals():
132: (32)                                     if (x.dtype == uint64 and y.dtype in [int8, int16,
133: (32)                                         or y.dtype == uint64 and x.dtype in [int8,
134: (32)                                         or x.dtype in _integer_dtypes and y.dtype not
135: (32)                                         or y.dtype in _integer_dtypes and x.dtype not
136: (32)                                         in _integer_dtypes
137: (32)
138: (32)
139: (32)
140: (32)
141: (32)
142: (32)
143: (32)
144: (32)
145: (32)
146: (32)
147: (32)
148: (32)
149: (32)
150: (32)
151: (32)
152: (32)
153: (32)
154: (32)
155: (32)
156: (32)
157: (32)
158: (32)
159: (32)
160: (32)
161: (32)
162: (32)
163: (32)
164: (32)
165: (32)
166: (32)
167: (32)
168: (32)
169: (32)
170: (32)
171: (32)
172: (32)
173: (32)
174: (32)
175: (32)
176: (32)
177: (32)
178: (32)
179: (32)
180: (32)
181: (32)
182: (32)
183: (32)
184: (32)
185: (32)
186: (32)
187: (32)
188: (32)
189: (32)
190: (32)
191: (32)
192: (32)
193: (32)
194: (32)
195: (32)
196: (32)
197: (32)
198: (32)
199: (32)
200: (32)
201: (32)
202: (32)
203: (32)
204: (32)
205: (32)
206: (32)
207: (32)
208: (32)
209: (32)
210: (32)
211: (32)
212: (32)
213: (32)
214: (32)
215: (32)
216: (32)
217: (32)
218: (32)
219: (32)
220: (32)
221: (32)
222: (32)
223: (32)
224: (32)
225: (32)
226: (32)
227: (32)
228: (32)
229: (32)
230: (32)
231: (32)
232: (32)
233: (32)
234: (32)
235: (32)
236: (32)
237: (32)
238: (32)
239: (32)
240: (32)
241: (32)
242: (32)
243: (32)
244: (32)
245: (32)
246: (32)
247: (32)
248: (32)
249: (32)
250: (32)
251: (32)
252: (32)
253: (32)
254: (32)
255: (32)
256: (32)
257: (32)
258: (32)
259: (32)
260: (32)
261: (32)
262: (32)
263: (32)
264: (32)
265: (32)
266: (32)
267: (32)
268: (32)
269: (32)
270: (32)
271: (32)
272: (32)
273: (32)
274: (32)
275: (32)
276: (32)
277: (32)
278: (32)
279: (32)
280: (32)
281: (32)
282: (32)
283: (32)
284: (32)
285: (32)
286: (32)
287: (32)
288: (32)
289: (32)
290: (32)
291: (32)
292: (32)
293: (32)
294: (32)
295: (32)
296: (32)
297: (32)
298: (32)
299: (32)
300: (32)
301: (32)
302: (32)
303: (32)
304: (32)
305: (32)
306: (32)
307: (32)
308: (32)
309: (32)
310: (32)
311: (32)
312: (32)
313: (32)
314: (32)
315: (32)
316: (32)
317: (32)
318: (32)
319: (32)
320: (32)
321: (32)
322: (32)
323: (32)
324: (32)
325: (32)
326: (32)
327: (32)
328: (32)
329: (32)
330: (32)
331: (32)
332: (32)
333: (32)
334: (32)
335: (32)
336: (32)
337: (32)
338: (32)
339: (32)
340: (32)
341: (32)
342: (32)
343: (32)
344: (32)
345: (32)
346: (32)
347: (32)
348: (32)
349: (32)
350: (32)
351: (32)
352: (32)
353: (32)
354: (32)
355: (32)
356: (32)
357: (32)
358: (32)
359: (32)
360: (32)
361: (32)
362: (32)
363: (32)
364: (32)
365: (32)
366: (32)
367: (32)
368: (32)
369: (32)
370: (32)
371: (32)
372: (32)
373: (32)
374: (32)
375: (32)
376: (32)
377: (32)
378: (32)
379: (32)
380: (32)
381: (32)
382: (32)
383: (32)
384: (32)
385: (32)
386: (32)
387: (32)
388: (32)
389: (32)
390: (32)
391: (32)
392: (32)
393: (32)
394: (32)
395: (32)
396: (32)
397: (32)
398: (32)
399: (32)
400: (32)
401: (32)
402: (32)
403: (32)
404: (32)
405: (32)
406: (32)
407: (32)
408: (32)
409: (32)
410: (32)
411: (32)
412: (32)
413: (32)
414: (32)
415: (32)
416: (32)
417: (32)
418: (32)
419: (32)
420: (32)
421: (32)
422: (32)
423: (32)
424: (32)
425: (32)
426: (32)
427: (32)
428: (32)
429: (32)
430: (32)
431: (32)
432: (32)
433: (32)
434: (32)
435: (32)
436: (32)
437: (32)
438: (32)
439: (32)
440: (32)
441: (32)
442: (32)
443: (32)
444: (32)
445: (32)
446: (32)
447: (32)
448: (32)
449: (32)
450: (32)
451: (32)
452: (32)
453: (32)
454: (32)
455: (32)
456: (32)
457: (32)
458: (32)
459: (32)
460: (32)
461: (32)
462: (32)
463: (32)
464: (32)
465: (32)
466: (32)
467: (32)
468: (32)
469: (32)
470: (32)
471: (32)
472: (32)
473: (32)
474: (32)
475: (32)
476: (32)
477: (32)
478: (32)
479: (32)
480: (32)
481: (32)
482: (32)
483: (32)
484: (32)
485: (32)
486: (32)
487: (32)
488: (32)
489: (32)
490: (32)
491: (32)
492: (32)
493: (32)
494: (32)
495: (32)
496: (32)
497: (32)
498: (32)
499: (32)
500: (32)
501: (32)
502: (32)
503: (32)
504: (32)
505: (32)
506: (32)
507: (32)
508: (32)
509: (32)
510: (32)
511: (32)
512: (32)
513: (32)
514: (32)
515: (32)
516: (32)
517: (32)
518: (32)
519: (32)
520: (32)
521: (32)
522: (32)
523: (32)
524: (32)
525: (32)
526: (32)
527: (32)
528: (32)
529: (32)
530: (32)
531: (32)
532: (32)
533: (32)
534: (32)
535: (32)
536: (32)
537: (32)
538: (32)
539: (32)
540: (32)
541: (32)
542: (32)
543: (32)
544: (32)
545: (32)
546: (32)
547: (32)
548: (32)
549: (32)
550: (32)
551: (32)
552: (32)
553: (32)
554: (32)
555: (32)
556: (32)
557: (32)
558: (32)
559: (32)
560: (32)
561: (32)
562: (32)
563: (32)
564: (32)
565: (32)
566: (32)
567: (32)
568: (32)
569: (32)
570: (32)
571: (32)
572: (32)
573: (32)
574: (32)
575: (32)
576: (32)
577: (32)
578: (32)
579: (32)
580: (32)
581: (32)
582: (32)
583: (32)
584: (32)
585: (32)
586: (32)
587: (32)
588: (32)
589: (32)
590: (32)
591: (32)
592: (32)
593: (32)
594: (32)
595: (32)
596: (32)
597: (32)
598: (32)
599: (32)
600: (32)
601: (32)
602: (32)
603: (32)
604: (32)
605: (32)
606: (32)
607: (32)
608: (32)
609: (32)
610: (32)
611: (32)
612: (32)
613: (32)
614: (32)
615: (32)
616: (32)
617: (32)
618: (32)
619: (32)
620: (32)
621: (32)
622: (32)
623: (32)
624: (32)
625: (32)
626: (32)
627: (32)
628: (32)
629: (32)
630: (32)
631: (32)
632: (32)
633: (32)
634: (32)
635: (32)
636: (32)
637: (32)
638: (32)
639: (32)
640: (32)
641: (32)
642: (32)
643: (32)
644: (32)
645: (32)
646: (32)
647: (32)
648: (32)
649: (32)
650: (32)
651: (32)
652: (32)
653: (32)
654: (32)
655: (32)
656: (32)
657: (32)
658: (32)
659: (32)
660: (32)
661: (32)
662: (32)
663: (32)
664: (32)
665: (32)
666: (32)
667: (32)
668: (32)
669: (32)
670: (32)
671: (32)
672: (32)
673: (32)
674: (32)
675: (32)
676: (32)
677: (32)
678: (32)
679: (32)
680: (32)
681: (32)
682: (32)
683: (32)
684: (32)
685: (32)
686: (32)
687: (32)
688: (32)
689: (32)
690: (32)
691: (32)
692: (32)
693: (32)
694: (32)
695: (32)
696: (32)
697: (32)
698: (32)
699: (32)
700: (32)
701: (32)
702: (32)
703: (32)
704: (32)
705: (32)
706: (32)
707: (32)
708: (32)
709: (32)
710: (32)
711: (32)
712: (32)
713: (32)
714: (32)
715: (32)
716: (32)
717: (32)
718: (32)
719: (32)
720: (32)
721: (32)
722: (32)
723: (32)
724: (32)
725: (32)
726: (32)
727: (32)
728: (32)
729: (32)
730: (32)
731: (32)
732: (32)
733: (32)
734: (32)
735: (32)
736: (32)
737: (32)
738: (32)
739: (32)
740: (32)
741: (32)
742: (32)
743: (32)
744: (32)
745: (32)
746: (32)
747: (32)
748: (32)
749: (32)
750: (32)
751: (32)
752: (32)
753: (32)
754: (32)
755: (32)
756: (32)
757: (32)
758: (32)
759: (32)
760: (32)
761: (32)
762: (32)
763: (32)
764: (32)
765: (32)
766: (32)
767: (32)
768: (32)
769: (32)
770: (32)
771: (32)
772: (32)
773: (32)
774: (32)
775: (32)
776: (32)
777: (32)
778: (32)
779: (32)
780: (32)
781: (32)
782: (32)
783: (32)
784: (32)
785: (32)
786: (32)
787: (32)
788: (32)
789: (32)
790: (32)
791: (32)
792: (32)
793: (32)
794: (32)
795: (32)
796: (32)
797: (32)
798: (32)
799: (32)
800: (32)
801: (32)
802: (32)
803: (32)
804: (32)
805: (32)
806: (32)
807: (32)
808: (32)
809: (32)
810: (32)
811: (32)
812: (32)
813: (32)
814: (32)
815: (32)
816: (32)
817: (32)
818: (32)
819: (32)
820: (32)
821: (32)
822: (32)
823: (32)
824: (32)
825: (32)
826: (32)
827: (32)
828: (32)
829: (32)
830: (32)
831: (32)
832: (32)
833: (32)
834: (32)
835: (32)
836: (32)
837: (32)
838: (32)
839: (32)
840: (32)
841: (32)
842: (32)
843: (32)
844: (32)
845: (32)
846: (32)
847: (32)
848: (32)
849: (32)
850: (32)
851: (32)
852: (32)
853: (32)
854: (32)
855: (32)
856: (32)
857: (32)
858: (32)
859: (32)
860: (32)
861: (32)
862: (32)
863: (32)
864: (32)
865: (32)
866: (32)
867: (32)
868: (32)
869: (32)
870: (32)
871: (32)
872: (32)
873: (32)
874: (32)
875: (32)
876: (32)
877: (32)
878: (32)
879: (32)
880: (32)
881: (32)
882: (32)
883: (32)
884: (32)
885: (32)
886: (32)
887: (32)
888: (32)
889: (32)
890: (32)
891: (32)
892: (32)
893: (32)
894: (32)
895: (32)
896: (32)
897: (32)
898: (32)
899: (32)
900: (32)
901: (32)
902: (32)
903: (32)
904: (32)
905: (32)
906: (32)
907: (32)
908: (32)
909: (32)
910: (32)
911: (32)
912: (32)
913: (32)
914: (32)
915: (32)
916: (32)
917: (32)
918: (32)
919: (32)
920: (32)
921: (32)
922: (32)
923: (32)
924: (32)
925: (32)
926: (32)
927: (32)
928: (32)
929: (32)
930: (32)
931: (32)
932: (32)
933: (32)
934: (32)
935: (32)
936: (32)
937: (32)
938: (32)
939: (32)
940: (32)
941: (32)
942: (32)
943: (32)
944: (32)
945: (32)
946: (32)
947: (32)
948: (32)
949: (32)
950: (32)
951: (32)
952: (32)
953: (32)
954: (32)
955: (32)
956: (32)
957: (32)
958: (32)
959: (32)
960: (32)
961: (32)
962: (32)
963: (32)
964: (32)
965: (32)
966: (32)
967: (32)
968: (32)
969: (32)
970: (32)
971: (32)
972: (32)
973: (32)
974: (32)
975: (32)
976: (32)
977: (32)
978: (32)
979: (32)
980: (32)
981: (32)
982: (32)
983: (32)
984: (32)
985: (32)
986: (32)
987: (32)
988: (32)
989: (32)
990: (32)
991: (32)
992: (32)
993: (32)
994: (32)
995: (32)
996: (32)
997: (32)
998: (32)
999: (32)
1000: (32)
1001: (32)
1002: (32)
1003: (32)
1004: (32)
1005: (32)
1006: (32)
1007: (32)
1008: (32)
1009: (32)
1010: (32)
1011: (32)
1012: (32)
1013: (32)
1014: (32)
1015: (32)
1016: (32)
1017: (32)
1018: (32)
1019: (32)
1020: (32)
1021: (32)
1022: (32)
1023: (32)
1024: (32)
1025: (32)
1026: (32)
1027: (32)
1028: (32)
1029: (32)
1030: (32)
1031: (32)
1032: (32)
1033: (32)
1034: (32)
1035: (32)
1036: (32)
1037: (32)
1038: (32)
1039: (32)
1040: (32)
1041: (32)
1042: (32)
1043: (32)
1044: (32)
1045: (32)
1046: (32)
1047: (32)
1048: (32)
1049: (32)
1050: (32)
1051: (32)
1052: (32)
1053: (32)
1054: (32)
1055: (32)
1056: (32)
1057: (32)
1058: (32)
1059: (32)
1060: (32)
1061: (32)
1062: (32)
1063: (32)
1064: (32)
1065: (32)
1066: (32)
1067: (32)
1068: (32)
1069: (32)
1070: (32)
1071: (32)
1072: (32)
1073: (32)
1074: (32)
1075: (32)
1076: (32)
1077: (32)
1078: (32)
1079: (32)
1080: (32)
1081: (32)
1082: (32)
1083: (32)
1084: (32)
1085: (32)
1086: (32)
1087: (32)
1088: (32)
1089: (32)
1090: (32)
1091: (32)
1092: (32)
1093: (32)
1094: (32)
1095: (32)
1096: (32)
1097: (32)
1098: (32)
1099: (32)
1100: (32)
1101: (32)
1102: (32)
1103: (32)
1104: (32)
1105: (32)
1106: (32)
1107: (32)
1108: (32)
1109: (32)
1110: (32)
1111: (32)
1112: (32)
1113: (32)
1114: (32)
1115: (32)
1116: (32)
1117: (32)
1118: (32)
1119: (32)
1120: (32)
1121: (32)
1122: (32)
1123: (32)
1124: (32)
1125: (32)
1126: (32)
1127: (32)
1128: (32)
1129: (32)
1130: (32)
1131: (32)
1132: (32)
1133: (32)
1134: (32)
1135: (32)
1136: (32)
1137: (32)
1138: (32)
1139: (32)
1140: (32)
1141: (32)
1142: (32)
1143: (32)
1144: (32)
1145: (32)
1146: (32)
1147: (32)
1148: (32)
1149: (32)
1150: (32)
1151: (32)
1152: (32)
1153: (32)
1154: (32)
1155: (32)
1156: (32)
1157: (32)
1158: (32)
1159: (32)
1160: (32)
1161: (32)
1162: (32)
1163: (32)
1164: (32)
1165: (32)
1166: (32)
1167: (32)
1168: (32)
1169: (32)
1170: (32)
1171: (32)
1172: (32)
1173: (32)
1174: (32)
1175: (32)
1176: (32)
1177: (32)
1178: (32)
1179: (32)
1180: (32)
1181: (32)
1182: (32)
1183: (32)
1184: (32)
1185: (32)
1186: (32)
1187: (32)
1188: (32)
1189: (32)
1190: (32)
1191: (32)
1192: (32)
1193: (32)
1194: (32)
1195: (32)
1196: (32)
1197: (32)
1198: (32)
1199: (32)
1200: (32)
1201: (32)
1202: (32)
1203: (32)
1204: (32)
1205: (32)
1206: (32)
1207: (32)
1208: (32)
1209: (32)
1210: (32)
1211: (32)
1212: (32)
1213: (32)
1214: (32)
1215: (32)
1216: (32)
1217: (32)
1218: (32)
1219: (32)
1220: (32)
1221: (32)
1222: (32)
1223: (32)
1224: (32)
1225: (32)
1226: (32)
1227: (32)
1228: (32)
1229: (32)
1230: (32)
1231: (32)
1232: (32)
1233: (32)
1234: (32)
1235: (32)
1236: (32)
1237: (32)
1238: (32)
1239: (32)
1240: (32)
1241: (32)
1242: (32)
1243: (32)
1244: (32)
1245: (32)
1246: (32)
1247: (32)
1248: (32)
1249: (32)
1250: (32)
1251: (32)
1252: (32)
1253: (32)
1254: (32)
1255: (32)
1256: (32)
1257: (32)
1258: (32)
1259: (32)
1260: (32)
1261: (32)
1262: (32)
1263: (32)
1264: (32)
1265: (32)
1266: (32)
1267: (32)
1268: (32)
1269: (32)
1270: (32)
1271: (32)
1272: (32)
1273: (32)
1274: (32)
1275: (32)
1276: (32)
1277: (32)
1278: (32)
1279: (32)
1280: (32)
1281: (32)
1282: (32)
1283: (32)
1284: (32)
1285: (32)
1286: (32)
1287: (32)
1288: (32)
1289: (32)
1290: (32)
1291: (32)
1292: (32)
1293: (32)
1294: (32)
1295: (32)
1296: (32)
1297: (32)
1298: (32)
1299: (32)
1300: (32)
1301: (32)
1302: (32)
1303: (32)
1304: (32)
1305: (32)
1306: (32)
1307: (32)
1308: (32)
1309: (32)
1310: (32)
1311: (32)
1312: (32)
1313: (32)
1314: (32)
1315: (32)
1316: (32)
1317: (32)
1318: (32)
1319: (32)
1320: (32)
1321: (32)
1322: (32)
1323: (32)
1324: (32)
1325: (32)
1326: (32)
1327: (32)
1328: (32)
1329: (32)
1330: (32)
1331: (32)
13
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

125: (32)                                     or x.dtype in _boolean_dtypes and y.dtype not
in _boolean_dtypes                           or y.dtype in _boolean_dtypes and x.dtype not
126: (32)                                     or x.dtype in _floating_dtypes and y.dtype not
in _boolean_dtypes                           or y.dtype in _floating_dtypes and x.dtype not
127: (32)                                     ):
in _floating_dtypes
128: (32)                                     assert_raises(TypeError, lambda: getattr(x,
in _floating_dtypes
129: (32)
130: (32)                                     _op)(y))
131: (28)                                     )
132: (32)                                     elif (
133: (32)                                         _op.startswith("__i")
134: (28)                                         and result_type(x.dtype, y.dtype) != x.dtype
135: (32)                                     ):
136: (28)                                         assert_raises(TypeError, lambda: getattr(x,
in _boolean_dtypes and y.dtype in _boolean_dtypes
137: (54)                                         elif (dtypes == "all" and (x.dtype in
in _numeric_dtypes and y.dtype in _numeric_dtypes
138: (32)                                         or x.dtype in
in _real_numeric_dtypes and y.dtype in _real_numeric_dtypes
139: (32)                                         or (dtypes == "real numeric" and x.dtype in
in _numeric_dtypes and y.dtype in _numeric_dtypes
140: (32)                                         or (dtypes == "numeric" and x.dtype in
in _integer_dtypes and y.dtype in _integer_dtypes
141: (32)                                         or dtypes == "integer" and x.dtype in
in _integer_dtypes and y.dtype in _integer_dtypes
142: (71)                                         or
in _integer_dtypes and y.dtype in _integer_dtypes
143: (32)                                         or dtypes == "boolean" and x.dtype in
in _boolean_dtypes and y.dtype in _boolean_dtypes
144: (32)                                         or dtypes == "floating" and x.dtype in
in _floating_dtypes and y.dtype in _floating_dtypes
145: (28)                                     ):
146: (32)                                         setattr(x, _op)(y)
147: (28)
148: (32)                                     else:
in _op)(y))
149: (4)                                         assert_raises(TypeError, lambda: setattr(x,
unary_op_dtypes = {
150: (8)                                         "__abs__": "numeric",
151: (8)                                         "__invert__": "integer_or_boolean",
152: (8)                                         "__neg__": "numeric",
153: (8)                                         "__pos__": "numeric",
154: (4)                                         }
155: (4)                                         for op, dtypes in unary_op_dtypes.items():
156: (8)                                         for a in _array_vals():
157: (12)                                         if (
158: (16)                                             dtypes == "numeric"
159: (16)                                             and a.dtype in _numeric_dtypes
160: (16)                                             or dtypes == "integer_or_boolean"
161: (16)                                             and a.dtype in _integer_or_boolean_dtypes
162: (12)
163: (16)
164: (12)
165: (16)                                         getattribute(a, op)()
166: (4)                                         else:
167: (8)                                         assert_raises(TypeError, lambda: getattribute(a, op)())
def _matmul_array_vals():
168: (12)                                         for a in _array_vals():
169: (8)                                         yield a
170: (12)                                         for d in _all_dtypes:
171: (12)                                             yield ones((3, 4), dtype=d)
172: (12)                                             yield ones((4, 2), dtype=d)
173: (4)                                             yield ones((4, 4), dtype=d)
174: (8)                                         for _op in ["__matmul__", "__rmatmul__", "__imatmul__"]:
175: (12)                                         for s in [1, 1.0, False]:
176: (16)                                         for a in _matmul_array_vals():
177: (20)                                             if (type(s) in [float, int] and a.dtype in _floating_dtypes
                                                 or type(s) == int and a.dtype in _integer_dtypes):

```

```

178: (20)                         assert_raises(ValueError, lambda: getattr(a, _op)(s))
179: (16)                         else:
180: (20)                         assert_raises(TypeError, lambda: setattr(a, _op)(s))
181: (4)                          for x in _matmul_array_vals():
182: (8)                            for y in _matmul_array_vals():
183: (12)                              if (x.dtype == uint64 and y.dtype in [int8, int16, int32, int64]
184: (16)                                or y.dtype == uint64 and x.dtype in [int8, int16, int32,
185: (16)                                  int64])
186: (16)                                or x.dtype in _integer_dtypes and y.dtype not in
187: (16)                                  _integer_dtypes and x.dtype not in
188: (16)                                  or x.dtype in _floating_dtypes and y.dtype not in
189: (16)                                  or y.dtype in _floating_dtypes and x.dtype not in
190: (16)                                  or x.dtype in _boolean_dtypes
191: (16)                                  or y.dtype in _boolean_dtypes
192: (16)                            ):
193: (16)                            assert_raises(TypeError, lambda: x.__matmul__(y))
194: (16)                            assert_raises(TypeError, lambda: y.__rmatmul__(x))
195: (12)                            assert_raises(TypeError, lambda: x.__imatmul__(y))
196: (16)                            elif x.shape == () or y.shape == () or x.shape[1] != y.shape[0]:
197: (16)                            assert_raises(ValueError, lambda: x.__matmul__(y))
198: (16)                            assert_raises(ValueError, lambda: y.__rmatmul__(x))
199: (20)                            if result_type(x.dtype, y.dtype) != x.dtype:
200: (16)                                assert_raises(TypeError, lambda: x.__imatmul__(y))
201: (20)                            else:
202: (12)                                assert_raises(ValueError, lambda: x.__imatmul__(y))
203: (16)                            else:
204: (16)                                x.__matmul__(y)
205: (16)                                y.__rmatmul__(x)
206: (20)                                if result_type(x.dtype, y.dtype) != x.dtype:
207: (16)                                    assert_raises(TypeError, lambda: x.__imatmul__(y))
208: (20)                                    elif y.shape[0] != y.shape[1]:
209: (16)                                        assert_raises(ValueError, lambda: x.__imatmul__(y))
210: (20)                                    else:
211: (0)                                        x.__imatmul__(y)
def test_python_scalar_constructors():
212: (4)    b = asarray(False)
213: (4)    i = asarray(0)
214: (4)    f = asarray(0.0)
215: (4)    c = asarray(0j)
216: (4)    assert bool(b) == False
217: (4)    assert int(i) == 0
218: (4)    assert float(f) == 0.0
219: (4)    assert operator.index(i) == 0
220: (4)    assert_raises(TypeError, lambda: bool(asarray([False])))
221: (4)    assert_raises(TypeError, lambda: int(asarray([0])))
222: (4)    assert_raises(TypeError, lambda: float(asarray([0.0])))
223: (4)    assert_raises(TypeError, lambda: complex(asarray([0j])))
224: (4)    assert_raises(TypeError, lambda: operator.index(asarray([0])))
225: (4)    assert bool(b) is bool(i) is bool(f) is bool(c) is False
226: (4)    assert int(b) == int(i) == int(f) == 0
227: (4)    assert_raises(TypeError, lambda: int(c))
228: (4)    assert float(b) == float(i) == float(f) == 0.0
229: (4)    assert_raises(TypeError, lambda: float(c))
230: (4)    assert complex(b) == complex(i) == complex(f) == complex(c) == 0j
231: (4)    assert operator.index(i) == 0
232: (4)    assert_raises(TypeError, lambda: operator.index(b))
233: (4)    assert_raises(TypeError, lambda: operator.index(f))
234: (4)    assert_raises(TypeError, lambda: operator.index(c))
235: (0)    def test_device_property():
236: (4)        a = ones((3, 4))
237: (4)        assert a.device == 'cpu'
238: (4)        assert all(equal(a.to_device('cpu'), a))
239: (4)        assert_raises(ValueError, lambda: a.to_device('gpu'))
240: (4)        assert all(equal(asarray(a, device='cpu'), a))
241: (4)        assert_raises(ValueError, lambda: asarray(a, device='gpu'))
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

242: (0) def test_array_properties():
243: (4)     a = ones((1, 2, 3))
244: (4)     b = ones((2, 3))
245: (4)     assert_raises(ValueError, lambda: a.T)
246: (4)     assert isinstance(b.T, Array)
247: (4)     assert b.T.shape == (3, 2)
248: (4)     assert isinstance(a.mT, Array)
249: (4)     assert a.mT.shape == (1, 3, 2)
250: (4)     assert isinstance(b.mT, Array)
251: (4)     assert b.mT.shape == (3, 2)
252: (0) def test__array__():
253: (4)     a = ones((2, 3), dtype=int16)
254: (4)     assert np.asarray(a) is a._array
255: (4)     b = np.asarray(a, dtype=np.float64)
256: (4)     assert np.all(np.equal(b, np.ones((2, 3), dtype=np.float64)))
257: (4)     assert b.dtype == np.float64
258: (0) def test_allow_newaxis():
259: (4)     a = ones(5)
260: (4)     indexed_a = a[None, :]
261: (4)     assert indexed_a.shape == (1, 5)
262: (0) def test_disallow_flat_indexing_with_newaxis():
263: (4)     a = ones((3, 3, 3))
264: (4)     with pytest.raises(IndexError):
265: (8)         a[None, 0, 0]
266: (0) def test_disallow_mask_with_newaxis():
267: (4)     a = ones((3, 3, 3))
268: (4)     with pytest.raises(IndexError):
269: (8)         a[None, asarray(True)]
270: (0) @pytest.mark.parametrize("shape", [(), (5,), (3, 3, 3)])
271: (0) @pytest.mark.parametrize("index", ["string", False, True])
272: (0) def test_error_on_invalid_index(shape, index):
273: (4)     a = ones(shape)
274: (4)     with pytest.raises(IndexError):
275: (8)         a[index]
276: (0) def test_mask_0d_array_without_errors():
277: (4)     a = ones(())
278: (4)     a[asarray(True)]
279: (0) @pytest.mark.parametrize(
280: (4)     "i", [slice(5), slice(5, 0), asarray(True), asarray([0, 1])]
281: (0)
282: (0) def test_error_on_invalid_index_with_ellipsis(i):
283: (4)     a = ones((3, 3, 3))
284: (4)     with pytest.raises(IndexError):
285: (8)         a[..., i]
286: (4)     with pytest.raises(IndexError):
287: (8)         a[i, ...]
288: (0) def test_array_keys_use_private_array():
289: (4)     """
290: (4)         Indexing operations convert array keys before indexing the internal array
291: (4)         Fails when array_api array keys are not converted into NumPy-proper arrays
292: (4)         in __getitem__(). This is achieved by passing array_api arrays with 0-
293: (4)         sized
294: (4)         dimensions, which NumPy-proper treats erroneously - not sure why!
295: (4)         TODO: Find and use appropriate __setitem__() case.
296: (4)         """
297: (4)         a = ones((0, 0), dtype=bool_)
298: (4)         assert a[a].shape == (0,)
299: (4)         a = ones((0,), dtype=bool_)
300: (4)         key = ones((0, 0), dtype=bool_)
301: (8)         with pytest.raises(IndexError):
302: (8)             a[key]

```

---

#### File 30 - test\_creation\_functions.py:

```

1: (0)         from numpy.testing import assert_raises
2: (0)         import numpy as np
3: (0)         from .. import all

```

```

4: (0)
5: (4)
6: (4)
7: (4)
8: (4)
9: (4)
10: (4)
11: (4)
12: (4)
13: (4)
14: (4)
15: (4)
16: (4)
17: (4)
18: (0)
19: (0)
20: (0)
21: (0)
22: (4)
23: (4)
24: (4)
25: (4)
26: (4)
27: (4)
28: (4)
29: (4)
30: (4)
31: (0)
32: (4)
33: (4)
34: (4)
35: (4)
36: (4)
37: (4)
38: (4)
39: (4)
40: (4)
41: (4)
42: (4)
43: (4)
44: (4)
45: (4)
46: (4)
47: (4)
48: (18)
49: (0)
50: (4)
51: (4)
52: (4)
53: (4)
54: (0)
55: (4)
56: (4)
57: (4)
58: (4)
59: (0)
60: (4)
61: (4)
62: (4)
63: (4)
64: (0)
65: (4)
66: (4)
67: (4)
68: (4)
69: (0)
70: (4)
71: (4)
72: (4)

from .._creation_functions import (
    asarray,
    arange,
    empty,
    empty_like,
    eye,
    full,
    full_like,
    linspace,
    meshgrid,
    ones,
    ones_like,
    zeros,
    zeros_like,
)
from .._dtypes import float32, float64
from .._array_object import Array
def test_asarray_errors():
    assert_raises(TypeError, lambda: Array([1]))
    assert_raises(TypeError, lambda: asarray(["a"]))
    assert_raises(ValueError, lambda: asarray([1.0], dtype=np.float16))
    assert_raises(OverflowError, lambda: asarray(2**100))
    assert_raises(TypeError, lambda: asarray([2**100]))
    asarray([1], device="cpu") # Doesn't error
    assert_raises(ValueError, lambda: asarray([1], device="gpu"))
    assert_raises(ValueError, lambda: asarray([1], dtype=int))
    assert_raises(ValueError, lambda: asarray([1], dtype="i"))

def test_asarray_copy():
    a = asarray([1])
    b = asarray(a, copy=True)
    a[0] = 0
    assert all(b[0] == 1)
    assert all(a[0] == 0)
    a = asarray([1])
    b = asarray(a, copy=np._CopyMode.ALWAYS)
    a[0] = 0
    assert all(b[0] == 1)
    assert all(a[0] == 0)
    a = asarray([1])
    b = asarray(a, copy=np._CopyMode.NEVER)
    a[0] = 0
    assert all(b[0] == 0)
    assert_raises(NotImplementedError, lambda: asarray(a, copy=False))
    assert_raises(NotImplementedError,
                 lambda: asarray(a, copy=np._CopyMode.IF_NEEDED))

def test_arange_errors():
    arange(1, device="cpu") # Doesn't error
    assert_raises(ValueError, lambda: arange(1, device="gpu"))
    assert_raises(ValueError, lambda: arange(1, dtype=int))
    assert_raises(ValueError, lambda: arange(1, dtype="i"))

def test_empty_errors():
    empty((1,), device="cpu") # Doesn't error
    assert_raises(ValueError, lambda: empty((1,), device="gpu"))
    assert_raises(ValueError, lambda: empty((1,), dtype=int))
    assert_raises(ValueError, lambda: empty((1,), dtype="i"))

def test_empty_like_errors():
    empty_like(asarray(1), device="cpu") # Doesn't error
    assert_raises(ValueError, lambda: empty_like(asarray(1), device="gpu"))
    assert_raises(ValueError, lambda: empty_like(asarray(1), dtype=int))
    assert_raises(ValueError, lambda: empty_like(asarray(1), dtype="i"))

def test_eye_errors():
    eye(1, device="cpu") # Doesn't error
    assert_raises(ValueError, lambda: eye(1, device="gpu"))
    assert_raises(ValueError, lambda: eye(1, dtype=int))
    assert_raises(ValueError, lambda: eye(1, dtype="i"))

def test_full_errors():
    full((1,), 0, device="cpu") # Doesn't error
    assert_raises(ValueError, lambda: full((1,), 0, device="gpu"))
    assert_raises(ValueError, lambda: full((1,), 0, dtype=int))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

73: (4)             assert_raises(ValueError, lambda: full((1,), 0, dtype="i"))
74: (0)
75: (4)             def test_full_like_errors():
76: (4)                 full_like(asarray(1), 0, device="cpu") # Doesn't error
77: (4)                 assert_raises(ValueError, lambda: full_like(asarray(1), 0, device="gpu"))
78: (4)                 assert_raises(ValueError, lambda: full_like(asarray(1), 0, dtype=int))
79: (0)                 assert_raises(ValueError, lambda: full_like(asarray(1), 0, dtype="i"))
80: (4)             def test_linspace_errors():
81: (4)                 linspace(0, 1, 10, device="cpu") # Doesn't error
82: (4)                 assert_raises(ValueError, lambda: linspace(0, 1, 10, device="gpu"))
83: (4)                 assert_raises(ValueError, lambda: linspace(0, 1, 10, dtype=float))
84: (0)                 assert_raises(ValueError, lambda: linspace(0, 1, 10, dtype="f"))
85: (4)             def test_ones_errors():
86: (4)                 ones((1,), device="cpu") # Doesn't error
87: (4)                 assert_raises(ValueError, lambda: ones((1,), device="gpu"))
88: (4)                 assert_raises(ValueError, lambda: ones((1,), dtype=int))
89: (0)                 assert_raises(ValueError, lambda: ones((1,), dtype="i"))
90: (4)             def test_ones_like_errors():
91: (4)                 ones_like(asarray(1), device="cpu") # Doesn't error
92: (4)                 assert_raises(ValueError, lambda: ones_like(asarray(1), device="gpu"))
93: (4)                 assert_raises(ValueError, lambda: ones_like(asarray(1), dtype=int))
94: (0)                 assert_raises(ValueError, lambda: ones_like(asarray(1), dtype="i"))
95: (4)             def test_zeros_errors():
96: (4)                 zeros((1,), device="cpu") # Doesn't error
97: (4)                 assert_raises(ValueError, lambda: zeros((1,), device="gpu"))
98: (4)                 assert_raises(ValueError, lambda: zeros((1,), dtype=int))
99: (0)                 assert_raises(ValueError, lambda: zeros((1,), dtype="i"))
100: (4)            def test_zeros_like_errors():
101: (4)                zeros_like(asarray(1), device="cpu") # Doesn't error
102: (4)                assert_raises(ValueError, lambda: zeros_like(asarray(1), device="gpu"))
103: (4)                assert_raises(ValueError, lambda: zeros_like(asarray(1), dtype=int))
104: (0)                assert_raises(ValueError, lambda: zeros_like(asarray(1), dtype="i"))
105: (4)            def test_meshgrid_dtype_errors():
106: (4)                meshgrid()
107: (4)                meshgrid(asarray([1.], dtype=float32))
108: (4)                meshgrid(asarray([1.], dtype=float32), asarray([1.], dtype=float32))
109: (0)                assert_raises(ValueError, lambda: meshgrid(asarray([1.], dtype=float32),
asarray([1.], dtype=float64)))

```

---

File 31 - test\_data\_type\_functions.py:

```

1: (0)             import pytest
2: (0)             from numpy.testing import assert_raises
3: (0)             from numpy import array_api as xp
4: (0)             import numpy as np
5: (0)             @pytest.mark.parametrize(
6: (4)                 "from_, to, expected",
7: (4)                 [
8: (8)                     (xp.int8, xp.int16, True),
9: (8)                     (xp.int16, xp.int8, False),
10: (8)                     (xp.bool, xp.int8, False),
11: (8)                     (xp.asarray(0, dtype=xp.uint8), xp.int8, False),
12: (4)                 ],
13: (0)
14: (0)             def test_can_cast(from_, to, expected):
15: (4)                 """
16: (4)                     can_cast() returns correct result
17: (4)                 """
18: (4)                 assert xp.can_cast(from_, to) == expected
19: (0)             def test_isdtype_strictness():
20: (4)                 assert_raises(TypeError, lambda: xp.isdtype(xp.float64, 64))
21: (4)                 assert_raises(ValueError, lambda: xp.isdtype(xp.float64, 'f8'))
22: (4)                 assert_raises(TypeError, lambda: xp.isdtype(xp.float64, (('integral',),)))
23: (4)                 assert_raises(TypeError, lambda: xp.isdtype(xp.float64, np.object_))

```

---

File 32 - test\_elementwise\_functions.py:

```
1: (0)     from inspect import getfullargspec
2: (0)     from numpy.testing import assert_raises
3: (0)     from .. import asarray, _elementwise_functions
4: (0)     from .._elementwise_functions import bitwise_left_shift, bitwise_right_shift
5: (0)     from ..dtypes import (
6: (4)         _dtype_categories,
7: (4)         _boolean_dtypes,
8: (4)         _floating_dtypes,
9: (4)         _integer_dtypes,
10: (0)
11: (0)    )
12: (4)    def nargs(func):
13: (0)        return len(getfullargspec(func).args)
14: (4)    def test_function_types():
15: (8)        elementwise_function_input_types = {
16: (8)            "abs": "numeric",
17: (8)            "acos": "floating-point",
18: (8)            "acosh": "floating-point",
19: (8)            "add": "numeric",
20: (8)            "asin": "floating-point",
21: (8)            "asinh": "floating-point",
22: (8)            "atan": "floating-point",
23: (8)            "atan2": "real floating-point",
24: (8)            "atanh": "floating-point",
25: (8)            "bitwise_and": "integer or boolean",
26: (8)            "bitwise_invert": "integer or boolean",
27: (8)            "bitwise_left_shift": "integer",
28: (8)            "bitwise_or": "integer or boolean",
29: (8)            "bitwise_right_shift": "integer",
30: (8)            "bitwise_xor": "integer or boolean",
31: (8)            "ceil": "real numeric",
32: (8)            "conj": "complex floating-point",
33: (8)            "cos": "floating-point",
34: (8)            "cosh": "floating-point",
35: (8)            "divide": "floating-point",
36: (8)            "equal": "all",
37: (8)            "exp": "floating-point",
38: (8)            "expm1": "floating-point",
39: (8)            "floor": "real numeric",
40: (8)            "floor_divide": "real numeric",
41: (8)            "greater": "real numeric",
42: (8)            "greater_equal": "real numeric",
43: (8)            "imag": "complex floating-point",
44: (8)            "isfinite": "numeric",
45: (8)            "isinf": "numeric",
46: (8)            "isnan": "numeric",
47: (8)            "less": "real numeric",
48: (8)            "less_equal": "real numeric",
49: (8)            "log": "floating-point",
50: (8)            "logaddexp": "real floating-point",
51: (8)            "log10": "floating-point",
52: (8)            "log1p": "floating-point",
53: (8)            "log2": "floating-point",
54: (8)            "logical_and": "boolean",
55: (8)            "logical_not": "boolean",
56: (8)            "logical_or": "boolean",
57: (8)            "logical_xor": "boolean",
58: (8)            "multiply": "numeric",
59: (8)            "negative": "numeric",
60: (8)            "not_equal": "all",
61: (8)            "positive": "numeric",
62: (8)            "pow": "numeric",
63: (8)            "real": "complex floating-point",
64: (8)            "remainder": "real numeric",
65: (8)            "round": "numeric",
66: (8)            "sign": "numeric",
67: (8)            "sin": "floating-point",
68: (8)            "sinh": "floating-point",
69: (8)            "sqrt": "floating-point",
```

```

69: (8)                 "square": "numeric",
70: (8)                 "subtract": "numeric",
71: (8)                 "tan": "floating-point",
72: (8)                 "tanh": "floating-point",
73: (8)                 "trunc": "real numeric",
74: (4)
75: (4)             def _array_vals():
76: (8)                 for d in _integer_dtypes:
77: (12)                     yield asarray(1, dtype=d)
78: (8)                 for d in _boolean_dtypes:
79: (12)                     yield asarray(False, dtype=d)
80: (8)                 for d in _floating_dtypes:
81: (12)                     yield asarray(1.0, dtype=d)
82: (4)             for x in _array_vals():
83: (8)                 for func_name, types in elementwise_function_input_types.items():
84: (12)                     dtypes = _dtype_categories[types]
85: (12)                     func = getattr(_elementwise_functions, func_name)
86: (12)                     if nargs(func) == 2:
87: (16)                         for y in _array_vals():
88: (20)                             if x.dtype not in dtypes or y.dtype not in dtypes:
89: (24)                                 assert_raises(TypeError, lambda: func(x, y))
90: (12)                     else:
91: (16)                         if x.dtype not in dtypes:
92: (20)                             assert_raises(TypeError, lambda: func(x))
93: (0)             def test_bitwise_shift_error():
94: (4)                 assert_raises(
95: (8)                     ValueError, lambda: bitwise_left_shift(asarray([1, 1]), asarray([1,
-1]))
96: (4)
97: (4)                 assert_raises(
98: (8)                     ValueError, lambda: bitwise_right_shift(asarray([1, 1]), asarray([1,
-1]))
99: (4)

```

-----

## File 33 - test\_indexing\_functions.py:

```

1: (0)         import pytest
2: (0)         from numpy import array_api as xp
3: (0)         @pytest.mark.parametrize(
4: (4)             "x, indices, axis, expected",
5: (4)             [
6: (8)                 ([2, 3], [1, 1, 0], 0, [3, 3, 2]),
7: (8)                 ([2, 3], [1, 1, 0], -1, [3, 3, 2]),
8: (8)                 ([[2, 3]], [1], -1, [[3]]),
9: (8)                 ([[2, 3]], [0, 0], 0, [[2, 3], [2, 3]]),
10: (4)                ],
11: (0)
12: (0)             def test_take_function(x, indices, axis, expected):
13: (4)                 """
14: (4)                     Indices respect relative order of a descending stable-sort
15: (4)                     See https://github.com/numpy/numpy/issues/20778
16: (4)                 """
17: (4)                 x = xp.asarray(x)
18: (4)                 indices = xp.asarray(indices)
19: (4)                 out = xp.take(x, indices, axis=axis)
20: (4)                 assert xp.all(out == xp.asarray(expected))

```

-----

## File 34 - test\_manipulation\_functions.py:

```

1: (0)         from numpy.testing import assert_raises
2: (0)         import numpy as np
3: (0)         from .. import all_
4: (0)         from .._creation_functions import asarray
5: (0)         from .._dtypes import float64, int8
6: (0)         from .._manipulation_functions import (

```

```

7: (8)           concat,
8: (8)           reshape,
9: (8)           stack
10: (0)
11: (0)         )
12: (4)         def test_concat_errors():
13: (4)             assert_raises(TypeError, lambda: concat((1, 1), axis=None))
14: (45)             assert_raises(TypeError, lambda: concat([asarray([1], dtype=int8),
15: (0)                             asarray([1], dtype=float64)]))
16: (4)
17: (44)         def test_stack_errors():
18: (0)             assert_raises(TypeError, lambda: stack([asarray([1, 1], dtype=int8),
19: (4)                             asarray([2, 2], dtype=float64)]))
20: (4)
21: (4)         def test_reshape_copy():
22: (4)             a = asarray(np.ones((2, 3)))
23: (4)             b = reshape(a, (3, 2), copy=True)
24: (4)             assert not np.shares_memory(a._array, b._array)
25: (4)             a = asarray(np.ones((2, 3)))
26: (4)             b = reshape(a, (3, 2), copy=False)
27: (4)             assert np.shares_memory(a._array, b._array)
28: (4)             a = asarray(np.ones((2, 3)).T)
29: (4)             b = reshape(a, (3, 2), copy=True)
30: (4)             assert_raises(AttributeError, lambda: reshape(a, (2, 3), copy=False))

-----

```

## File 35 - test\_set\_functions.py:

```

1: (0)           import pytest
2: (0)           from hypothesis import given
3: (0)           from hypothesis.extra.array_api import make_strategies_namespace
4: (0)           from numpy import array_api as xp
5: (0)           xps = make_strategies_namespace(xp)
6: (0)           @pytest.mark.parametrize("func", [xp.unique_all, xp.unique_inverse])
7: (0)           @given(xps.arrays(dtype=xps.scalar_dtypes(), shape=xps.array_shapes()))
8: (0)           def test_inverse_indices_shape(func, x):
9: (4)               """
10: (4)                   Inverse indices share shape of input array
11: (4)                   See https://github.com/numpy/numpy/issues/20638
12: (4)               """
13: (4)               out = func(x)
14: (4)               assert out.inverse_indices.shape == x.shape

-----

```

## File 36 - test\_sorting\_functions.py:

```

1: (0)           import pytest
2: (0)           from numpy import array_api as xp
3: (0)           @pytest.mark.parametrize(
4: (4)               "obj, axis, expected",
5: (4)               [
6: (8)                   ([[0, 0], -1, [0, 1]],
7: (8)                   ([[0, 1, 0], -1, [1, 0, 2]]),
8: (8)                   ([[0, 1], [1, 1]], 0, [[1, 0], [0, 1]]),
9: (8)                   ([[0, 1], [1, 1]], 1, [[1, 0], [0, 1]]),
10: (4)               ],
11: (0)           )
12: (0)           def test_stable_desc_argsort(obj, axis, expected):
13: (4)               """
14: (4)                   Indices respect relative order of a descending stable-sort
15: (4)                   See https://github.com/numpy/numpy/issues/20778
16: (4)               """
17: (4)               x = xp.asarray(obj)
18: (4)               out = xp.argsort(x, axis=axis, stable=True, descending=True)
19: (4)               assert xp.all(out == xp.asarray(expected))

-----

```

## File 37 - test\_validation.py:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1: (0)     from typing import Callable
2: (0)     import pytest
3: (0)     from numpy import array_api as xp
4: (0)     def p(func: Callable, *args, **kwargs):
5: (4)         f_sig = ", ".join(
6: (8)             [str(a) for a in args] + [f"{k}={v}" for k, v in kwargs.items()])
7: (4)     )
8: (4)     id_ = f"{func.__name__}({f_sig})"
9: (4)     return pytest.param(func, args, kwargs, id=id_)
10: (0)    @pytest.mark.parametrize(
11: (4)        "func, args, kwargs",
12: (4)        [
13: (8)            p(xp.can_cast, 42, xp.int8),
14: (8)            p(xp.can_cast, xp.int8, 42),
15: (8)            p(xp.result_type, 42),
16: (4)        ],
17: (0)
18: (0)    def test.raises_on_invalid_types(func, args, kwargs):
19: (4)        """Function raises TypeError when passed invalidly-typed inputs"""
20: (4)        with pytest.raises(TypeError):
21: (8)            func(*args, **kwargs)

```

-----  
File 38 - py3k.py:

```

1: (0) """
2: (0) Python 3.X compatibility tools.
3: (0) While this file was originally intended for Python 2 -> 3 transition,
4: (0) it is now used to create a compatibility layer between different
5: (0) minor versions of Python 3.
6: (0) While the active version of numpy may not support a given version of python,
we
7: (0)     allow downstream libraries to continue to use these shims for forward
8: (0)     compatibility with numpy while they transition their code to newer versions of
9: (0)     Python.
10: (0) """
11: (0) __all__ = ['bytes', 'asbytes', 'isfileobj', 'getexception', 'strchar',
12: (11)         'unicode', 'asunicode', 'asbytes_nested', 'asunicode_nested',
13: (11)         'asstr', 'open_latin1', 'long', 'basestring', 'sixu',
14: (11)         'integer_types', 'is_pathlib_path', 'npy_load_module', 'Path',
15: (11)         'pickle', 'contextlib_nullcontext', 'os_fspath', 'os_PathLike']
16: (0) import sys
17: (0) import os
18: (0) from pathlib import Path
19: (0) import io
20: (0) try:
21: (4)     import pickle5 as pickle
22: (0) except ImportError:
23: (4)     import pickle
24: (0) long = int
25: (0) integer_types = (int,)
26: (0) basestring = str
27: (0) unicode = str
28: (0) bytes = bytes
29: (0) def asunicode(s):
30: (4)     if isinstance(s, bytes):
31: (8)         return s.decode('latin1')
32: (4)     return str(s)
33: (0) def asbytes(s):
34: (4)     if isinstance(s, bytes):
35: (8)         return s
36: (4)     return str(s).encode('latin1')
37: (0) def asstr(s):
38: (4)     if isinstance(s, bytes):
39: (8)         return s.decode('latin1')
40: (4)     return str(s)
41: (0) def isfileobj(f):
42: (4)     if not isinstance(f, (io.FileIO, io.BufferedReader, io.BufferedWriter)):

```

```

43: (8)             return False
44: (4)         try:
45: (8)             f.fileno()
46: (8)             return True
47: (4)         except OSError:
48: (8)             return False
49: (0)     def open_latin1(filename, mode='r'):
50: (4)         return open(filename, mode=mode, encoding='iso-8859-1')
51: (0)
52: (4)     def sixu(s):
53: (0)         return s
54: (0)         strchar = 'U'
55: (4)     def getexception():
56: (0)         return sys.exc_info()[1]
57: (4)     def asbytes_nested(x):
58: (8)         if hasattr(x, '__iter__') and not isinstance(x, (bytes, unicode)):
59: (8)             return [asbytes_nested(y) for y in x]
60: (4)         else:
61: (8)             return asbytes(x)
62: (0)     def asunicode_nested(x):
63: (4)         if hasattr(x, '__iter__') and not isinstance(x, (bytes, unicode)):
64: (8)             return [asunicode_nested(y) for y in x]
65: (4)         else:
66: (8)             return asunicode(x)
67: (0)     def is_pathlib_path(obj):
68: (4)         """
69: (4)             Check whether obj is a `pathlib.Path` object.
70: (4)             Prefer using ``isinstance(obj, os.PathLike)`` instead of this function.
71: (4)         """
72: (0)         return isinstance(obj, Path)
73: (4)     class contextlib_nullcontext:
74: (4)         """
75: (4)             Context manager that does no additional processing.
76: (4)             Used as a stand-in for a normal context manager, when a particular
77: (4)             block of code is only sometimes used with a normal context manager:
78: (4)             cm = optional_cm if condition else nullcontext()
79: (8)             with cm:
80: (4)                 .. note::
81: (8)                     Prefer using `contextlib.nullcontext` instead of this context manager.
82: (4)             def __init__(self, enter_result=None):
83: (8)                 self.enter_result = enter_result
84: (4)             def __enter__(self):
85: (8)                 return self.enter_result
86: (4)             def __exit__(self, *excinfo):
87: (8)                 pass
88: (0)         def npy_load_module(name, fn, info=None):
89: (4)             """
90: (4)                 Load a module. Uses ``load_module`` which will be deprecated in python
91: (4)                 3.12. An alternative that uses ``exec_module`` is in
92: (4)                 numpy.distutils.misc_util.exec_mod_from_location
93: (4)                 .. versionadded:: 1.11.2
94: (4)                 Parameters
95: (4)                 -----
96: (8)                 name : str
97: (8)                     Full module name.
98: (4)                 fn : str
98: (8)                     Path to module file.
99: (4)                 info : tuple, optional
100: (8)                     Only here for backward compatibility with Python 2.*.
101: (4)                 Returns
102: (4)                 -----
103: (4)                 mod : module
104: (4)             """
105: (4)                 from importlib.machinery import SourceFileLoader
106: (4)                 return SourceFileLoader(name, fn).load_module()
107: (0)                 os_fspath = os.fspath
108: (0)                 os_PathLike = os.PathLike

```

## File 39 - \_\_init\_\_.py:

```

1: (0)      """
2: (0)      Tests for the array API namespace.
3: (0)      Note, full compliance with the array API can be tested with the official array
API test
4: (0)      suite https://github.com/data-apis/array-api-tests. This test suite primarily
5: (0)      focuses on those things that are not tested by the official test suite.
6: (0)      """
-----
```

## File 40 - setup.py:

```

1: (0)      def configuration(parent_package='', top_path=None):
2: (4)          from numpy.distutils.misc_util import Configuration
3: (4)          config = Configuration('compat', parent_package, top_path)
4: (4)          config.add_subpackage('tests')
5: (4)          return config
6: (0)      if __name__ == '__main__':
7: (4)          from numpy.distutils.core import setup
8: (4)          setup(configuration=configuration)
-----
```

## File 41 - \_\_init\_\_.py:

```

1: (0)      """
2: (0)      Compatibility module.
3: (0)      This module contains duplicated code from Python itself or 3rd party
4: (0)      extensions, which may be included for the following reasons:
5: (2)          * compatibility
6: (2)          * we may only need a small subset of the copied library/module
7: (0)      """
8: (0)      from .._utils import _inspect
9: (0)      from .._utils._inspect import getargspec, formatargspec
10: (0)      from . import py3k
11: (0)      from .py3k import *
12: (0)      __all__ = []
13: (0)      __all__.extend(_inspect.__all__)
14: (0)      __all__.extend(py3k.__all__)
-----
```

## File 42 - test\_compat.py:

```

1: (0)      from os.path import join
2: (0)      from io import BufferedReader, BytesIO
3: (0)      from numpy.compat import isfileobj
4: (0)      from numpy.testing import assert_
5: (0)      from numpy.testing import tempdir
6: (0)      def test_isfileobj():
7: (4)          with tempdir(prefix="numpy_test_compat_") as folder:
8: (8)              filename = join(folder, 'a.bin')
9: (8)              with open(filename, 'wb') as f:
10: (12)                  assert_(isfileobj(f))
11: (8)                  with open(filename, 'ab') as f:
12: (12)                      assert_(isfileobj(f))
13: (8)                      with open(filename, 'rb') as f:
14: (12)                          assert_(isfileobj(f))
15: (8)                          assert_(isfileobj(BufferedReader(BytesIO()))) is False)
-----
```

## File 43 - \_\_init\_\_.py:

```

1: (0)
-----
```

## File 44 - arrayprint.py:

```

1: (0)          """Array printing function
2: (0)          $Id: arrayprint.py,v 1.9 2005/09/13 13:58:44 teoliphant Exp $
3: (0)
4: (0)          __all__ = ["array2string", "array_str", "array_repr", "set_string_function",
5: (11)           "set_printoptions", "get_printoptions", "printoptions",
6: (11)           "format_float_positional", "format_float_scientific"]
7: (0)          __docformat__ = 'restructuredtext'
8: (0)          import functools
9: (0)          import numbers
10: (0)         import sys
11: (0)         try:
12: (4)             from _thread import get_ident
13: (0)         except ImportError:
14: (4)             from _dummy_thread import get_ident
15: (0)         import numpy as np
16: (0)         from . import numeric_types as _nt
17: (0)         from .umath import absolute, isnan, isinf, isnan
18: (0)         from . import multiarray
19: (0)         from .multiarray import (array, dragon4_positional, dragon4_scientific,
20: (25)             datetime_as_string, datetime_data, ndarray,
21: (25)             set_legacy_print_mode)
22: (0)         from .fromnumeric import any
23: (0)         from .numeric import concatenate, asarray, errstate
24: (0)         from .numeric_types import (longlong, intc, int_, float_, complex_, bool_,
25: (27)             flexible)
26: (0)         from .overrides import array_function_dispatch, set_module
27: (0)         import operator
28: (0)         import warnings
29: (0)         import contextlib
30: (0)         _format_options = {
31: (4)             'edgeitems': 3, # repr N leading and trailing items of each dimension
32: (4)             'threshold': 1000, # total items > triggers array summarization
33: (4)             'floatmode': 'maxprec',
34: (4)             'precision': 8, # precision of floating point representations
35: (4)             'suppress': False, # suppress printing small floating values in exp
format
36: (4)             'linewidth': 75,
37: (4)             'nanstr': 'nan',
38: (4)             'infstr': 'inf',
39: (4)             'sign': '-',
40: (4)             'formatter': None,
41: (4)             'legacy': sys.maxsize}
42: (0)         def _make_options_dict(precision=None, threshold=None, edgeitems=None,
43: (23)                         linewidth=None, suppress=None, nanstr=None,
44: (23)                         sign=None, formatter=None, floatmode=None,
45: (4)                         ):
46: (4)             """
47: (4)             Make a dictionary out of the non-None arguments, plus conversion of
48: (4)             *legacy* and sanity checks.
49: (4)             """
50: (4)             options = {k: v for k, v in locals().items() if v is not None}
51: (8)             if suppress is not None:
52: (4)                 options['suppress'] = bool(suppress)
53: (4)             modes = ['fixed', 'unique', 'maxprec', 'maxprec_equal']
54: (8)             if floatmode not in modes + [None]:
55: (25)                 raise ValueError("floatmode option must be one of " +
56: (4)                     ", ".join('{}'.format(m) for m in modes))
57: (8)             if sign not in [None, '-', '+', '']:
58: (4)                 raise ValueError("sign option must be one of ' ', '+', or '-'")
59: (8)             if legacy == False:
60: (4)                 options['legacy'] = sys.maxsize
61: (8)             elif legacy == '1.13':
62: (4)                 options['legacy'] = 113
63: (8)             elif legacy == '1.21':
64: (4)                 options['legacy'] = 121

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

64: (4)      elif legacy is None:
65: (8)          pass # OK, do nothing.
66: (4)      else:
67: (8)          warnings.warn(
68: (12)              "legacy printing option can currently only be '1.13', '1.21', or "
69: (12)              "'False'", stacklevel=3)
70: (4)      if threshold is not None:
71: (8)          if not isinstance(threshold, numbers.Number):
72: (12)              raise TypeError("threshold must be numeric")
73: (8)          if np.isnan(threshold):
74: (12)              raise ValueError("threshold must be non-NAN, try "
75: (29)                  "sys.maxsize for untruncated representation")
76: (4)      if precision is not None:
77: (8)          try:
78: (12)              options['precision'] = operator.index(precision)
79: (8)          except TypeError as e:
80: (12)              raise TypeError('precision must be an integer') from e
81: (4)      return options
82: (0) @set_module('numpy')
83: (0) def set_printoptions(precision=None, threshold=None, edgeitems=None,
84: (21)                 linewidth=None, suppress=None, nanstr=None, infstr=None,
85: (21)                 formatter=None, sign=None, floatmode=None, *,
86: (4) legacy=None):
87: (4)     """
88: (4)         Set printing options.
89: (4)         These options determine the way floating point numbers, arrays and
90: (4)         other NumPy objects are displayed.
91: (4)         Parameters
92: (4)         -----
93: (8)             precision : int or None, optional
94: (8)                 Number of digits of precision for floating point output (default 8).
95: (8)                 May be None if `floatmode` is not `fixed`, to print as many digits as
96: (4)                 necessary to uniquely specify the value.
97: (8)             threshold : int, optional
98: (8)                 Total number of array elements which trigger summarization
99: (8)                 rather than full repr (default 1000).
100: (4)                To always use the full repr without summarization, pass `sys.maxsize`.
101: (8)             edgeitems : int, optional
102: (8)                 Number of array items in summary at beginning and end of
103: (4)                 each dimension (default 3).
104: (8)             linewidth : int, optional
105: (8)                 The number of characters per line for the purpose of inserting
106: (4)                 line breaks (default 75).
107: (8)             suppress : bool, optional
108: (8)                 If True, always print floating point numbers using fixed point
109: (8)                 notation, in which case numbers equal to zero in the current precision
110: (8)                 will print as zero. If False, then scientific notation is used when
111: (8)                 absolute value of the smallest number is < 1e-4 or the ratio of the
112: (4)                 maximum absolute value to the minimum is > 1e3. The default is False.
113: (8)             nanstr : str, optional
114: (4)                 String representation of floating point not-a-number (default nan).
115: (8)             infstr : str, optional
116: (4)                 String representation of floating point infinity (default inf).
117: (8)             sign : string, either '-', '+', or ' ', optional
118: (8)                 Controls printing of the sign of floating-point types. If '+', always
119: (8)                 print the sign of positive values. If ' ', always prints a space
120: (8)                 (whitespace character) in the sign position of positive values. If
121: (4)                 '-', omit the sign character of positive values. (default '-')
122: (8)             formatter : dict of callables, optional
123: (8)                 If not None, the keys should indicate the type(s) that the respective
124: (8)                 formatting function applies to. Callables should return a string.
125: (8)                 Types that are not specified (by their corresponding keys) are handled
126: (8)                 by the default formatters. Individual types for which a formatter
127: (8)                 can be set are:
128: (8)                     - 'bool'
129: (8)                     - 'int'
130: (8)                     - 'timedelta' : a `numpy.timedelta64`
131: (8)                     - 'datetime' : a `numpy.datetime64`

```

```

132: (8)           - 'longfloat' : 128-bit floats
133: (8)           - 'complexfloat'
134: (8)           - 'longcomplexfloat' : composed of two 128-bit floats
135: (8)           - 'numpystr' : types `numpy.bytes_` and `numpy.str_`
136: (8)           - 'object' : `np.object_` arrays
137: (8)           Other keys that can be used to set a group of types at once are:
138: (8)           - 'all' : sets all types
139: (8)           - 'int_kind' : sets 'int'
140: (8)           - 'float_kind' : sets 'float' and 'longfloat'
141: (8)           - 'complex_kind' : sets 'complexfloat' and 'longcomplexfloat'
142: (8)           - 'str_kind' : sets 'numpystr'
143: (4)           floatmode : str, optional
144: (8)             Controls the interpretation of the `precision` option for
145: (8)             floating-point types. Can take the following values
146: (8)             (default maxprec_equal):
147: (8)               * 'fixed': Always print exactly `precision` fractional digits,
148: (16)                 even if this would print more or fewer digits than
149: (16)                 necessary to specify the value uniquely.
150: (8)               * 'unique': Print the minimum number of fractional digits necessary
151: (16)                 to represent each value uniquely. Different elements may
152: (16)                 have a different number of digits. The value of the
153: (16)                 `precision` option is ignored.
154: (8)               * 'maxprec': Print at most `precision` fractional digits, but if
155: (16)                 an element can be uniquely represented with fewer digits
156: (16)                 only print it with that many.
157: (8)               * 'maxprec_equal': Print at most `precision` fractional digits,
158: (16)                 but if every element in the array can be uniquely
159: (16)                 represented with an equal number of fewer digits, use that
160: (16)                 many digits for all elements.
161: (4)           legacy : string or `False`, optional
162: (8)             If set to the string ``1.13`` enables 1.13 legacy printing mode. This
163: (8)             approximates numpy 1.13 print output by including a space in the sign
164: (8)             position of floats and different behavior for 0d arrays. This also
165: (8)             enables 1.21 legacy printing mode (described below).
166: (8)             If set to the string ``1.21`` enables 1.21 legacy printing mode. This
167: (8)             approximates numpy 1.21 print output of complex structured dtypes
168: (8)             by not inserting spaces after commas that separate fields and after
169: (8)             colons.
170: (8)             If set to `False`, disables legacy mode.
171: (8)             Unrecognized strings will be ignored with a warning for forward
172: (8)             compatibility.
173: (8)             .. versionadded:: 1.14.0
174: (8)             .. versionchanged:: 1.22.0
175: (4)           See Also
176: (4)           -----
177: (4)             get_printoptions, printoptions, set_string_function, array2string
178: (4)           Notes
179: (4)           -----
180: (4)             `formatter` is always reset with a call to `set_printoptions`.
181: (4)             Use `printoptions` as a context manager to set the values temporarily.
182: (4)           Examples
183: (4)           -----
184: (4)             Floating point precision can be set:
185: (4)             >>> np.set_printoptions(precision=4)
186: (4)             >>> np.array([1.123456789])
187: (4)             [1.1235]
188: (4)             Long arrays can be summarised:
189: (4)             >>> np.set_printoptions(threshold=5)
190: (4)             >>> np.arange(10)
191: (4)             array([0, 1, 2, ..., 7, 8, 9])
192: (4)             Small results can be suppressed:
193: (4)             >>> eps = np.finfo(float).eps
194: (4)             >>> x = np.arange(4.)
195: (4)             >>> x**2 - (x + eps)**2
196: (4)             array([-4.9304e-32, -4.4409e-16,  0.0000e+00,  0.0000e+00])
197: (4)             >>> np.set_printoptions(suppress=True)
198: (4)             >>> x**2 - (x + eps)**2
199: (4)             array([-0., -0.,  0.,  0.])
200: (4)             A custom formatter can be used to display array elements as desired:

```

```

201: (4)          >>> np.set_printoptions(formatter={'all':lambda x: 'int: '+str(-x)})
202: (4)          >>> x = np.arange(3)
203: (4)          >>> x
204: (4)          array([int: 0, int: -1, int: -2])
205: (4)          >>> np.set_printoptions() # formatter gets reset
206: (4)          >>> x
207: (4)          array([0, 1, 2])
208: (4)          To put back the default options, you can use:
209: (4)          >>> np.set_printoptions(edgeitems=3, infstr='inf',
210: (4)          ... linewidth=75, nanstr='nan', precision=8,
211: (4)          ... suppress=False, threshold=1000, formatter=None)
212: (4)          Also to temporarily override options, use `printoptions` as a context
manager:
213: (4)          >>> with np.printoptions(precision=2, suppress=True, threshold=5):
214: (4)          ...     np.linspace(0, 10, 10)
215: (4)          array([ 0. ,  1.1,  2.2, ...,  7.78,  8.89, 10. ])
216: (4)          """
217: (4)          opt = _make_options_dict(precision, threshold, edgeitems, linewidth,
218: (29)          ...             suppress, nanstr, infstr, sign, formatter,
219: (29)          ...             floatmode, legacy)
220: (4)          opt['formatter'] = formatter
221: (4)          _format_options.update(opt)
222: (4)          if _format_options['legacy'] == 113:
223: (8)          set_legacy_print_mode(113)
224: (8)          _format_options['sign'] = '-'
225: (4)          elif _format_options['legacy'] == 121:
226: (8)          set_legacy_print_mode(121)
227: (4)          elif _format_options['legacy'] == sys.maxsize:
228: (8)          set_legacy_print_mode(0)
229: (0)          @set_module('numpy')
230: (0)          def get_printoptions():
231: (4)          """
232: (4)          Return the current print options.
233: (4)          Returns
234: (4)          -----
235: (4)          print_opts : dict
236: (8)          Dictionary of current print options with keys
237: (10)          - precision : int
238: (10)          - threshold : int
239: (10)          - edgeitems : int
240: (10)          - linewidth : int
241: (10)          - suppress : bool
242: (10)          - nanstr : str
243: (10)          - infstr : str
244: (10)          - formatter : dict of callables
245: (10)          - sign : str
246: (8)          For a full description of these options, see `set_printoptions`.
247: (4)          See Also
248: (4)          -----
249: (4)          set_printoptions, printoptions, set_string_function
250: (4)          """
251: (4)          opts = _format_options.copy()
252: (4)          opts['legacy'] = {
253: (8)          113: '1.13', 121: '1.21', sys.maxsize: False,
254: (4)          }[opts['legacy']]
255: (4)          return opts
256: (0)          def _get_legacy_print_mode():
257: (4)          """Return the legacy print mode as an int."""
258: (4)          return _format_options['legacy']
259: (0)          @set_module('numpy')
260: (0)          @contextlib.contextmanager
261: (0)          def printoptions(*args, **kwargs):
262: (4)          """Context manager for setting print options.
263: (4)          Set print options for the scope of the `with` block, and restore the old
264: (4)          options at the end. See `set_printoptions` for the full description of
265: (4)          available options.
266: (4)          Examples
267: (4)          -----
268: (4)          >>> from numpy.testing import assert_equal

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

269: (4)
270: (4)
271: (4)
272: (4)
273: (4)
274: (4)
275: (4)
276: (4)
277: (4)
278: (4)
279: (4)
280: (4)
281: (8)
282: (8)
283: (4)
284: (8)
285: (0)
286: (4)
287: (4)
288: (4)
289: (4)
290: (4)
291: (4)
292: (4)
293: (8)
294: (4)
295: (8)
296: (12)
:edgeitems],
297: (12)
298: (8)
299: (4)
300: (8)
301: (0)
302: (4)
303: (4)
304: (8)
305: (4)
306: (8)
307: (4)
308: (0)
309: (4)
310: (0)
311: (4)
312: (0)
313: (20)
314: (4)
315: (8)
316: (8)
317: (8)
318: (12)
319: (8)
320: (12)
321: (8)
322: (12)
323: (8)
324: (12)
325: (8)
326: (8)
327: (8)
328: (8)
329: (8)
330: (4)
331: (8)
332: (4)
333: (8)
334: (8)
335: (12)
336: (16)

    >>> with np.printoptions(precision=2):
        ...     np.array([2.0]) / 3
    array([0.67])
    The `as`-clause of the `with`-statement gives the current print options:
    >>> with np.printoptions(precision=2) as opts:
        ...     assert_equal(opts, np.get_printoptions())
    See Also
    -----
    set_printoptions, get_printoptions
    """
    opts = np.get_printoptions()
    try:
        np.set_printoptions(*args, **kwargs)
        yield np.get_printoptions()
    finally:
        np.set_printoptions(**opts)
def _leading_trailing(a, edgeitems, index=()):
    """
    Keep only the N-D corners (leading and trailing edges) of an array.
    Should be passed a base-class ndarray, since it makes no guarantees about
    preserving subclasses.
    """
    axis = len(index)
    if axis == a.ndim:
        return a[index]
    if a.shape[axis] > 2*edgeitems:
        return concatenate((
            _leading_trailing(a, edgeitems, index + np.index_exp[
                _leading_trailing(a, edgeitems, index + np.index_exp[-edgeitems:])
            ], axis=axis)
        ), axis=axis)
    else:
        return _leading_trailing(a, edgeitems, index + np.index_exp[:])
def _object_format(o):
    """
    Object arrays containing lists should be printed unambiguously """
    if type(o) is list:
        fmt = 'list({!r})'
    else:
        fmt = '{!r}'
    return fmt.format(o)
def repr_format(x):
    return repr(x)
def str_format(x):
    return str(x)
def _get_formatdict(data, *, precision, floatmode, suppress, sign, legacy,
                   formatter, **kwargs):
    formatdict = {
        'bool': lambda: BoolFormat(data),
        'int': lambda: IntegerFormat(data),
        'float': lambda: FloatingFormat(
            data, precision, floatmode, suppress, sign, legacy=legacy),
        'longfloat': lambda: FloatingFormat(
            data, precision, floatmode, suppress, sign, legacy=legacy),
        'complexfloat': lambda: ComplexFloatingFormat(
            data, precision, floatmode, suppress, sign, legacy=legacy),
        'longcomplexfloat': lambda: ComplexFloatingFormat(
            data, precision, floatmode, suppress, sign, legacy=legacy),
        'datetime': lambda: DatetimeFormat(data, legacy=legacy),
        'timedelta': lambda: TimedeltaFormat(data),
        'object': lambda: _object_format,
        'void': lambda: str_format,
        'numpystr': lambda: repr_format}
    def indirect(x):
        return lambda: x
    if formatter is not None:
        fkeys = [k for k in formatter.keys() if formatter[k] is not None]
        if 'all' in fkeys:
            for key in formatdict.keys():
                formatdict[key] = indirect(formatter['all'])

```

```

337: (8)             if 'int_kind' in fkeys:
338: (12)            for key in ['int']:
339: (16)              formatdict[key] = indirect(formatter['int_kind'])
340: (8)            if 'float_kind' in fkeys:
341: (12)              for key in ['float', 'longfloat']:
342: (16)                formatdict[key] = indirect(formatter['float_kind'])
343: (8)            if 'complex_kind' in fkeys:
344: (12)              for key in ['complexfloat', 'longcomplexfloat']:
345: (16)                formatdict[key] = indirect(formatter['complex_kind'])
346: (8)            if 'str_kind' in fkeys:
347: (12)              formatdict['numpystr'] = indirect(formatter['str_kind'])
348: (8)            for key in formatdict.keys():
349: (12)              if key in fkeys:
350: (16)                formatdict[key] = indirect(formatter[key])
351: (4)            return formatdict
352: (0)          def _get_format_function(data, **options):
353: (4)            """
354: (4)              find the right formatting function for the dtype_
355: (4)            """
356: (4)            dtype_ = data.dtype
357: (4)            dtypeobj = dtype_.type
358: (4)            formatdict = _get_formatdict(data, **options)
359: (4)            if dtypeobj is None:
360: (8)              return formatdict["numpystr"]()
361: (4)            elif issubclass(dtypeobj, _nt.bool_):
362: (8)              return formatdict['bool']()
363: (4)            elif issubclass(dtypeobj, _nt.integer):
364: (8)              if issubclass(dtypeobj, _nt.timedelta64):
365: (12)                return formatdict['timedelta']()
366: (8)              else:
367: (12)                return formatdict['int']()
368: (4)            elif issubclass(dtypeobj, _nt.floating):
369: (8)              if issubclass(dtypeobj, _nt.longfloat):
370: (12)                return formatdict['longfloat']()
371: (8)              else:
372: (12)                return formatdict['float']()
373: (4)            elif issubclass(dtypeobj, _nt.complexfloating):
374: (8)              if issubclass(dtypeobj, _nt.clongfloat):
375: (12)                return formatdict['longcomplexfloat']()
376: (8)              else:
377: (12)                return formatdict['complexfloat']()
378: (4)            elif issubclass(dtypeobj, (_nt.str_, _nt.bytes_)):
379: (8)              return formatdict['numpystr']()
380: (4)            elif issubclass(dtypeobj, _nt.datetime64):
381: (8)              return formatdict['datetime']()
382: (4)            elif issubclass(dtypeobj, _nt.object_):
383: (8)              return formatdict['object']()
384: (4)            elif issubclass(dtypeobj, _nt void):
385: (8)              if dtype_.names is not None:
386: (12)                return StructuredVoidFormat.from_data(data, **options)
387: (8)              else:
388: (12)                return formatdict['void']()
389: (4)            else:
390: (8)              return formatdict['numpystr']()
391: (0)          def _recursive_guard(fillvalue='...'):
392: (4)            """
393: (4)              Like the python 3.2 reprlib.recursive_repr, but forwards *args and
394: (4)              **kwargs
395: (4)              Decorates a function such that if it calls itself with the same first
396: (4)              argument, it returns `fillvalue` instead of recursing.
397: (4)              Largely copied from reprlib.recursive_repr
398: (4)            """
399: (8)            def decorating_function(f):
400: (8)              repr_running = set()
401: (8)              @functools.wraps(f)
402: (12)              def wrapper(self, *args, **kwargs):
403: (12)                key = id(self), get_ident()
404: (16)                if key in repr_running:

```

```

405: (12)           repr_running.add(key)
406: (12)           try:
407: (16)             return f(self, *args, **kwargs)
408: (12)           finally:
409: (16)             repr_running.discard(key)
410: (8)           return wrapper
411: (4)           return decorating_function
412: (0) @_recursive_guard()
413: (0) def _array2string(a, options, separator=' ', prefix ""):
414: (4)   data = asarray(a)
415: (4)   if a.shape == ():
416: (8)     a = data
417: (4)   if a.size > options['threshold']:
418: (8)     summary_insert = "..."
419: (8)     data = _leading_trailing(data, options['edgeitems'])
420: (4)   else:
421: (8)     summary_insert = ""
422: (4)   format_function = _get_format_function(data, **options)
423: (4)   next_line_prefix = " "
424: (4)   next_line_prefix += " " *len(prefix)
425: (4)   lst = _formatArray(a, format_function, options['linewidth'],
426: (23)               next_line_prefix, separator, options['edgeitems'],
427: (23)               summary_insert, options['legacy'])
428: (4)   return lst
429: (0) def _array2string_dispatcher(
430: (8)   a, max_line_width=None, precision=None,
431: (8)   suppress_small=None, separator=None, prefix=None,
432: (8)   style=None, formatter=None, threshold=None,
433: (8)   edgeitems=None, sign=None, floatmode=None, suffix=None,
434: (8)   *, legacy=None):
435: (4)   return (a,)
436: (0) @array_function_dispatch(_array2string_dispatcher, module='numpy')
437: (0) def array2string(a, max_line_width=None, precision=None,
438: (17)           suppress_small=None, separator=' ', prefix="",
439: (17)           style=np._NoValue, formatter=None, threshold=None,
440: (17)           edgeitems=None, sign=None, floatmode=None, suffix="",
441: (17)           *, legacy=None):
442: (4) """
443: (4)     Return a string representation of an array.
444: (4) Parameters
445: (4) -----
446: (4) a : ndarray
447: (8)     Input array.
448: (4) max_line_width : int, optional
449: (8)     Inserts newlines if text is longer than `max_line_width`.
450: (8)     Defaults to ``numpy.get_printoptions()['linewidth']``.
451: (4) precision : int or None, optional
452: (8)     Floating point precision.
453: (8)     Defaults to ``numpy.get_printoptions()['precision']``.
454: (4) suppress_small : bool, optional
455: (8)     Represent numbers "very close" to zero as zero; default is False.
456: (8)     Very close is defined by precision: if the precision is 8, e.g.,
457: (8)     numbers smaller (in absolute value) than 5e-9 are represented as
458: (8)     zero.
459: (8)     Defaults to ``numpy.get_printoptions()['suppress']``.
460: (4) separator : str, optional
461: (8)     Inserted between elements.
462: (4) prefix : str, optional
463: (4) suffix : str, optional
464: (8)     The length of the prefix and suffix strings are used to respectively
465: (8)     align and wrap the output. An array is typically printed as::
466: (10)       prefix + array2string(a) + suffix
467: (8)     The output is left-padded by the length of the prefix string, and
468: (8)     wrapping is forced at the column ``max_line_width - len(suffix)``.
469: (8)     It should be noted that the content of prefix and suffix strings are
470: (8)     not included in the output.
471: (4) style : _NoValue, optional
472: (8)     Has no effect, do not use.
473: (8)     .. deprecated:: 1.14.0

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

474: (4) formatter : dict of callables, optional
475: (8) If not None, the keys should indicate the type(s) that the respective
476: (8) formatting function applies to. Callables should return a string.
477: (8) Types that are not specified (by their corresponding keys) are handled
478: (8) by the default formatters. Individual types for which a formatter
479: (8) can be set are:
480: (8) - 'bool'
481: (8) - 'int'
482: (8) - 'timedelta' : a `numpy.timedelta64`
483: (8) - 'datetime' : a `numpy.datetime64`
484: (8) - 'float'
485: (8) - 'longfloat' : 128-bit floats
486: (8) - 'complexfloat'
487: (8) - 'longcomplexfloat' : composed of two 128-bit floats
488: (8) - 'void' : type `numpy.void`
489: (8) - 'numpystr' : types `numpy.bytes_` and `numpy.str_`
490: (8) Other keys that can be used to set a group of types at once are:
491: (8) - 'all' : sets all types
492: (8) - 'int_kind' : sets 'int'
493: (8) - 'float_kind' : sets 'float' and 'longfloat'
494: (8) - 'complex_kind' : sets 'complexfloat' and 'longcomplexfloat'
495: (8) - 'str_kind' : sets 'numpystr'
496: (4) threshold : int, optional
497: (8) Total number of array elements which trigger summarization
498: (8) rather than full repr.
499: (8) Defaults to ``numpy.get_printoptions()['threshold']``.
500: (4) edgeitems : int, optional
501: (8) Number of array items in summary at beginning and end of
502: (8) each dimension.
503: (8) Defaults to ``numpy.get_printoptions()['edgeitems']``.
504: (4) sign : string, either '-', '+', or ' ', optional
505: (8) Controls printing of the sign of floating-point types. If '+', always
506: (8) print the sign of positive values. If ' ', always prints a space
507: (8) (whitespace character) in the sign position of positive values. If
508: (8) '-', omit the sign character of positive values.
509: (8) Defaults to ``numpy.get_printoptions()['sign']``.
510: (4) floatmode : str, optional
511: (8) Controls the interpretation of the `precision` option for
512: (8) floating-point types.
513: (8) Defaults to ``numpy.get_printoptions()['floatmode']``.
514: (8) Can take the following values:
515: (8) - 'fixed': Always print exactly `precision` fractional digits,
516: (10) even if this would print more or fewer digits than
517: (10) necessary to specify the value uniquely.
518: (8) - 'unique': Print the minimum number of fractional digits necessary
519: (10) to represent each value uniquely. Different elements may
520: (10) have a different number of digits. The value of the
521: (10) `precision` option is ignored.
522: (8) - 'maxprec': Print at most `precision` fractional digits, but if
523: (10) an element can be uniquely represented with fewer digits
524: (10) only print it with that many.
525: (8) - 'maxprec_equal': Print at most `precision` fractional digits,
526: (10) but if every element in the array can be uniquely
527: (10) represented with an equal number of fewer digits, use that
528: (10) many digits for all elements.
529: (4) legacy : string or `False`, optional
530: (8) If set to the string ``1.13`` enables 1.13 legacy printing mode. This
531: (8) approximates numpy 1.13 print output by including a space in the sign
532: (8) position of floats and different behavior for 0d arrays. If set to
533: (8) `False`, disables legacy mode. Unrecognized strings will be ignored
534: (8) with a warning for forward compatibility.
535: (8) .. versionadded:: 1.14.0
536: (4) Returns
537: (4) -----
538: (4) array_str : str
539: (8) String representation of the array.
540: (4) Raises
541: (4) -----
542: (4) TypeError

```

```

543: (8)             if a callable in `formatter` does not return a string.
544: (4)             See Also
545: (4)
546: (4)             array_str, array_repr, set_printoptions, get_printoptions
547: (4)             Notes
548: (4)
549: (4)             If a formatter is specified for a certain type, the `precision` keyword is
550: (4)             ignored for that type.
551: (4)             This is a very flexible function; `array_repr` and `array_str` are using
552: (4)             `array2string` internally so keywords with the same name should work
553: (4)             identically in all three functions.
554: (4)             Examples
555: (4)
556: (4)             >>> x = np.array([1e-16,1,2,3])
557: (4)             >>> np.array2string(x, precision=2, separator=',',
558: (4)                           ...                           suppress_small=True)
559: (4)             '[0.,1.,2.,3.]'
560: (4)             >>> x = np.arange(3.)
561: (4)             >>> np.array2string(x, formatter={'float_kind':lambda x: "% .2f" % x})
562: (4)             '[0.00 1.00 2.00]'
563: (4)             >>> x = np.arange(3)
564: (4)             >>> np.array2string(x, formatter={'int':lambda x: hex(x)})
565: (4)             '[0x0 0x1 0x2]'

566: (4)
567: (4)             overrides = _make_options_dict(precision, threshold, edgeitems,
568: (35)                           max_line_width, suppress_small, None, None,
569: (35)                           sign, formatter, floatmode, legacy)
570: (4)             options = _format_options.copy()
571: (4)             options.update(overrides)
572: (4)             if options['legacy'] <= 113:
573: (8)                 if style is np._NoValue:
574: (12)                     style = repr
575: (8)                     if a.shape == () and a.dtype.names is None:
576: (12)                         return style(a.item())
577: (4)                     elif style is not np._NoValue:
578: (8)                         warnings.warn("'style' argument is deprecated and no longer
functional"
579: (22)                               " except in 1.13 'legacy' mode",
580: (22)                               DeprecationWarning, stacklevel=2)
581: (4)             if options['legacy'] > 113:
582: (8)                 options['linewidth'] -= len(suffix)
583: (4)             if a.size == 0:
584: (8)                 return "["
585: (4)             return _array2string(a, options, separator, prefix)
586: (0)             def _extendLine(s, line, word, line_width, next_line_prefix, legacy):
587: (4)                 needs_wrap = len(line) + len(word) > line_width
588: (4)                 if legacy > 113:
589: (8)                     if len(line) <= len(next_line_prefix):
590: (12)                         needs_wrap = False
591: (4)                 if needs_wrap:
592: (8)                     s += line.rstrip() + "\n"
593: (8)                     line = next_line_prefix
594: (4)                     line += word
595: (4)                     return s, line
596: (0)             def _extendLine_pretty(s, line, word, line_width, next_line_prefix, legacy):
597: (4)
598: (4)                 Extends line with nicely formatted (possibly multi-line) string ``word``.
599: (4)
600: (4)                 words = word.splitlines()
601: (4)                 if len(words) == 1 or legacy <= 113:
602: (8)                     return _extendLine(s, line, word, line_width, next_line_prefix,
legacy)
603: (4)                     max_word_length = max(len(word) for word in words)
604: (4)                     if (len(line) + max_word_length > line_width and
605: (12)                         len(line) > len(next_line_prefix)):
606: (8)                         s += line.rstrip() + '\n'
607: (8)                         line = next_line_prefix + words[0]
608: (8)                         indent = next_line_prefix
609: (4)                     else:
```

```

610: (8)           indent = len(line)*' '
611: (8)           line += words[0]
612: (4)           for word in words[1::]:
613: (8)             s += line.rstrip() + '\n'
614: (8)             line = indent + word
615: (4)           suffix_length = max_word_length - len(words[-1])
616: (4)           line += suffix_length*' '
617: (4)           return s, line
618: (0)           def _formatArray(a, format_function, line_width, next_line_prefix,
619: (17)                     separator, edge_items, summary_insert, legacy):
620: (4)             """formatArray is designed for two modes of operation:
621: (4)               1. Full output
622: (4)               2. Summarized output
623: (4) """
624: (4)           def recruser(index, hanging_indent, curr_width):
625: (8)             """
626: (8)               By using this local function, we don't need to recurse with all the
627: (8)               arguments. Since this function is not created recursively, the cost is
628: (8)               not significant
629: (8)
630: (8)               axis = len(index)
631: (8)               axes_left = a.ndim - axis
632: (8)               if axes_left == 0:
633: (12)                 return format_function(a[index])
634: (8)               next_hanging_indent = hanging_indent + ' '
635: (8)               if legacy <= 113:
636: (12)                 next_width = curr_width
637: (8)               else:
638: (12)                 next_width = curr_width - len(']')
639: (8)               a_len = a.shape[axis]
640: (8)               show_summary = summary_insert and 2*edge_items < a_len
641: (8)               if show_summary:
642: (12)                 leading_items = edge_items
643: (12)                 trailing_items = edge_items
644: (8)               else:
645: (12)                 leading_items = 0
646: (12)                 trailing_items = a_len
647: (8)               s = ''
648: (8)               if axes_left == 1:
649: (12)                 if legacy <= 113:
650: (16)                   elem_width = curr_width - len(separator.rstrip())
651: (12)                 else:
652: (16)                   elem_width = curr_width - max(len(separator.rstrip()),
len(']'))
653: (12)                   line = hanging_indent
654: (12)                   for i in range(leading_items):
655: (16)                     word = recruser(index + (i,), next_hanging_indent, next_width)
656: (16)                     s, line = _extendLine_pretty(
657: (20)                       s, line, word, elem_width, hanging_indent, legacy)
658: (16)                   line += separator
659: (12)                   if show_summary:
660: (16)                     s, line = _extendLine(
661: (20)                       s, line, summary_insert, elem_width, hanging_indent,
legacy)
662: (16)                     if legacy <= 113:
663: (20)                       line += ","
664: (16)                     else:
665: (20)                       line += separator
666: (12)                     for i in range(trailing_items, 1, -1):
667: (16)                       word = recruser(index + (-i,), next_hanging_indent,
next_width)
668: (16)                       s, line = _extendLine_pretty(
669: (20)                         s, line, word, elem_width, hanging_indent, legacy)
670: (16)                       line += separator
671: (12)                     if legacy <= 113:
672: (16)                       elem_width = curr_width
673: (12)                       word = recruser(index + (-1,), next_hanging_indent, next_width)
674: (12)                       s, line = _extendLine_pretty(
675: (16)                         s, line, word, elem_width, hanging_indent, legacy)

```

```

676: (12)           s += line
677: (8)            else:
678: (12)              s = ''
679: (12)              line_sep = separator.rstrip() + '\n'*(axes_left - 1)
680: (12)              for i in range(leading_items):
681: (16)                  nested = recurser(index + (i,), next_hanging_indent,
next_width)
682: (16)                      s += hanging_indent + nested + line_sep
683: (12)                  if show_summary:
684: (16)                      if legacy <= 113:
685: (20)                          s += hanging_indent + summary_insert + ", \n"
686: (16)                      else:
687: (20)                          s += hanging_indent + summary_insert + line_sep
688: (12)                  for i in range(trailing_items, 1, -1):
689: (16)                      nested = recurser(index + (-i,), next_hanging_indent,
next_width)
690: (34)                          s += hanging_indent + nested + line_sep
691: (16)                      nested = recurser(index + (-1,), next_hanging_indent, next_width)
692: (12)                      s += hanging_indent + nested
693: (12)                  s = '[' + s[len(hanging_indent):] + ']'
694: (8)                  return s
695: (8)
696: (4)          try:
697: (8)              return recurser(index=(),
698: (24)                  hanging_indent=next_line_prefix,
699: (24)                  curr_width=line_width)
700: (4)          finally:
701: (8)              recurser = None
702: (0)      def _none_or_positive_arg(x, name):
703: (4)          if x is None:
704: (8)              return -1
705: (4)          if x < 0:
706: (8)              raise ValueError("{} must be >= 0".format(name))
707: (4)          return x
708: (0)      class FloatingFormat:
709: (4)          """ Formatter for subtypes of np.floating """
710: (4)          def __init__(self, data, precision, floatmode, suppress_small, sign=False,
711: (17)                  *, legacy=None):
712: (8)              if isinstance(sign, bool):
713: (12)                  sign = '+' if sign else '-'
714: (8)              self._legacy = legacy
715: (8)              if self._legacy <= 113:
716: (12)                  if data.shape != () and sign == '-':
717: (16)                      sign = ' '
718: (8)                  self.floatmode = floatmode
719: (8)                  if floatmode == 'unique':
720: (12)                      self.precision = None
721: (8)                  else:
722: (12)                      self.precision = precision
723: (8)                  self.precision = _none_or_positive_arg(self.precision, 'precision')
724: (8)                  self.suppress_small = suppress_small
725: (8)                  self.sign = sign
726: (8)                  self.exp_format = False
727: (8)                  self.large_exponent = False
728: (8)                  self.fillFormat(data)
729: (4)          def fillFormat(self, data):
730: (8)              finite_vals = data[isfinite(data)]
731: (8)              abs_non_zero = absolute(finite_vals[finite_vals != 0])
732: (8)              if len(abs_non_zero) != 0:
733: (12)                  max_val = np.max(abs_non_zero)
734: (12)                  min_val = np.min(abs_non_zero)
735: (12)                  with errstate(over='ignore'): # division can overflow
736: (16)                      if max_val >= 1.e8 or (not self.suppress_small and
737: (24)                          (min_val < 0.0001 or max_val/min_val > 1000.)):
738: (20)                          self.exp_format = True
739: (8)              if len(finite_vals) == 0:
740: (12)                  self.pad_left = 0
741: (12)                  self.pad_right = 0
742: (12)                  self.trim = '.'
743: (12)                  self.exp_size = -1

```

```

744: (12)                     self.unique = True
745: (12)                     self.min_digits = None
746: (8)                      elif self.exp_format:
747: (12)                        trim, unique = '.', True
748: (12)                        if self.floatmode == 'fixed' or self._legacy <= 113:
749: (16)                          trim, unique = 'k', False
750: (12)                          strs = (dragon4_scientific(x, precision=self.precision,
751: (31)                                            unique=unique, trim=trim, sign=self.sign ==
752: ('+')
753: (20)
754: (12)                      for x in finite_vals)
755: (12)                      frac_strs, _, exp_strs = zip(*(s.partition('e') for s in strs))
756: (12)                      int_part, frac_part = zip(*(s.split('.') for s in frac_strs))
757: (12)                      self.exp_size = max(len(s) for s in exp_strs) - 1
758: (12)                      self.trim = 'k'
759: (12)                      self.precision = max(len(s) for s in frac_part)
760: (12)                      self.min_digits = self.precision
761: (16)                      self.unique = unique
762: (12)                      if self._legacy <= 113:
763: (16)                        self.pad_left = 3
764: (12)                      else:
765: (8)                        self.pad_left = max(len(s) for s in int_part)
766: (12)                        self.pad_right = self.exp_size + 2 + self.precision
767: (12)                      else:
768: (16)                        trim, unique = '.', True
769: (12)                        if self.floatmode == 'fixed':
770: (39)                          trim, unique = 'k', False
771: (39)                          strs = (dragon4_positional(x, precision=self.precision,
772: (39)                                            fractional=True,
773: (20)                                              unique=unique, trim=trim,
774: (12)                                              sign=self.sign == '+')
775: (12)                                              for x in finite_vals)
776: (16)                                              int_part, frac_part = zip(*(s.split('.') for s in strs))
777: (12)                                              if self._legacy <= 113:
778: (16)                                                self.pad_left = 1 + max(len(s.lstrip('-+')) for s in int_part)
779: (12)                                              else:
780: (12)                                                self.pad_left = max(len(s) for s in int_part)
781: (12)                                              self.pad_right = max(len(s) for s in frac_part)
782: (12)                                              self.exp_size = -1
783: (16)                                              self.unique = unique
784: (16)                                              if self.floatmode in ['fixed', 'maxprec_equal']:
785: (12)                                                self.precision = self.min_digits = self.pad_right
786: (16)                                                self.trim = 'k'
787: (16)                                              else:
788: (8)                                                self.trim = '.'
789: (12)                                                self.min_digits = 0
790: (16)                                              if self._legacy > 113:
791: (8)                                                if self.sign == ' ' and not any(np.signbit(finite_vals)):
792: (12)                                                  self.pad_left += 1
793: (12)                                              if data.size != finite_vals.size:
794: (12)                                                neginf = self.sign != '-' or any(data[isinf(data)] < 0)
795: (12)                                                nanlen = len(_format_options['nanstr'])
796: (12)                                                inflen = len(_format_options['infstr']) + neginf
797: (4)                                                offset = self.pad_right + 1 # +1 for decimal pt
798: (8)                                                self.pad_left = max(self.pad_left, nanlen - offset, inflen -
799: (12)                                              offset)
800: (16)                                              def __call__(self, x):
801: (20)                                                if not np.isfinite(x):
802: (20)                                                  with errstate(invalid='ignore'):
803: (16)                                                    if np.isnan(x):
804: (20)                                                      sign = '+' if self.sign == '+' else ''
805: (20)                                                      ret = sign + _format_options['nanstr']
806: (16)                                                    else: # isinf
807: (8)                                                      sign = '-' if x < 0 else '+' if self.sign == '+' else ''
808: (12)                                                      ret = sign + _format_options['infstr']
809: (38)                                                      return ' '*(self.pad_left + self.pad_right + 1 - len(ret)) +
ret
if self.exp_format:
    return dragon4_scientific(x,
                               precision=self.precision,

```

```

810: (38) min_digits=self.min_digits,
811: (38) unique=self.unique,
812: (38) trim=self.trim,
813: (38) sign=self.sign == '+',
814: (38) pad_left=self.pad_left,
815: (38) exp_digits=self.exp_size)
816: (8) else:
817: (12)     return dragon4_positional(x,
818: (38)             precision=self.precision,
819: (38)             min_digits=self.min_digits,
820: (38)             unique=self.unique,
821: (38)             fractional=True,
822: (38)             trim=self.trim,
823: (38)             sign=self.sign == '+',
824: (38)             pad_left=self.pad_left,
825: (38)             pad_right=self.pad_right)
826: (0) @set_module('numpy')
827: (0) def format_float_scientific(x, precision=None, unique=True, trim='k',
828: (28)             sign=False, pad_left=None, exp_digits=None,
829: (28)             min_digits=None):
830: (4) """
831: (4) Format a floating-point scalar as a decimal string in scientific notation.
832: (4) Provides control over rounding, trimming and padding. Uses and assumes
833: (4) IEEE unbiased rounding. Uses the "Dragon4" algorithm.
834: (4) Parameters
835: (4) -----
836: (4) x : python float or numpy floating scalar
837: (8)     Value to format.
838: (4) precision : non-negative integer or None, optional
839: (8)     Maximum number of digits to print. May be None if `unique` is
840: (8)     `True`, but must be an integer if unique is `False`.
841: (4) unique : boolean, optional
842: (8)     If `True`, use a digit-generation strategy which gives the shortest
843: (8)     representation which uniquely identifies the floating-point number
from
844: (8) other values of the same type, by judicious rounding. If `precision`
845: (8) is given fewer digits than necessary can be printed. If `min_digits`
846: (8) is given more can be printed, in which cases the last digit is rounded
847: (8) with unbiased rounding.
848: (8) If `False`, digits are generated as if printing an infinite-precision
849: (8) value and stopping after `precision` digits, rounding the remaining
850: (8) value with unbiased rounding
851: (4) trim : one of 'k', '.', '0', '-', optional
852: (8)     Controls post-processing trimming of trailing digits, as follows:
853: (8)     * 'k' : keep trailing zeros, keep decimal point (no trimming)
854: (8)     * '.' : trim all trailing zeros, leave decimal point
855: (8)     * '0' : trim all but the zero before the decimal point. Insert the
856: (10)      zero if it is missing.
857: (8)     * '-' : trim trailing zeros and any trailing decimal point
858: (4) sign : boolean, optional
859: (8)     Whether to show the sign for positive values.
860: (4) pad_left : non-negative integer, optional
861: (8)     Pad the left side of the string with whitespace until at least that
862: (8)     many characters are to the left of the decimal point.
863: (4) exp_digits : non-negative integer, optional
864: (8)     Pad the exponent with zeros until it contains at least this many
digits.
865: (8)     If omitted, the exponent will be at least 2 digits.
866: (4) min_digits : non-negative integer or None, optional
867: (8)     Minimum number of digits to print. This only has an effect for
868: (8)     `unique=True`. In that case more digits than necessary to uniquely
869: (8)     identify the value may be printed and rounded unbiased.
870: (8)     -- versionadded:: 1.21.0
871: (4) Returns
872: (4) -----
873: (4) rep : string
874: (8)     The string representation of the floating point value
875: (4) See Also
876: (4) -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

877: (4)           format_float_positional
878: (4)           Examples
879: (4)           -----
880: (4)           >>> np.format_float_scientific(np.float32(np.pi))
881: (4)           '3.1415927e+00'
882: (4)           >>> s = np.float32(1.23e24)
883: (4)           >>> np.format_float_scientific(s, unique=False, precision=15)
884: (4)           '1.230000071797338e+24'
885: (4)           >>> np.format_float_scientific(s, exp_digits=4)
886: (4)           '1.23e+0024'
887: (4)           """
888: (4)           precision = _none_or_positive_arg(precision, 'precision')
889: (4)           pad_left = _none_or_positive_arg(pad_left, 'pad_left')
890: (4)           exp_digits = _none_or_positive_arg(exp_digits, 'exp_digits')
891: (4)           min_digits = _none_or_positive_arg(min_digits, 'min_digits')
892: (4)           if min_digits > 0 and precision > 0 and min_digits > precision:
893: (8)             raise ValueError("min_digits must be less than or equal to precision")
894: (4)           return dragon4_scientific(x, precision=precision, unique=unique,
895: (30)                     trim=trim, sign=sign, pad_left=pad_left,
896: (30)                     exp_digits=exp_digits, min_digits=min_digits)
897: (0)           @set_module('numpy')
898: (0)           def format_float_positional(x, precision=None, unique=True,
899: (28)                     fractional=True, trim='k', sign=False,
900: (28)                     pad_left=None, pad_right=None, min_digits=None):
901: (4)           """
902: (4)           Format a floating-point scalar as a decimal string in positional notation.
903: (4)           Provides control over rounding, trimming and padding. Uses and assumes
904: (4)           IEEE unbiased rounding. Uses the "Dragon4" algorithm.
905: (4)           Parameters
906: (4)           -----
907: (4)           x : python float or numpy floating scalar
908: (8)             Value to format.
909: (4)           precision : non-negative integer or None, optional
910: (8)             Maximum number of digits to print. May be None if `unique` is
911: (8)             `True`, but must be an integer if unique is `False`.
912: (4)           unique : boolean, optional
913: (8)             If `True`, use a digit-generation strategy which gives the shortest
914: (8)             representation which uniquely identifies the floating-point number
from
915: (8)           other values of the same type, by judicious rounding. If `precision`
916: (8)             is given fewer digits than necessary can be printed, or if
`min_digits`
917: (8)             is given more can be printed, in which cases the last digit is rounded
918: (8)             with unbiased rounding.
919: (8)             If `False`, digits are generated as if printing an infinite-precision
920: (8)             value and stopping after `precision` digits, rounding the remaining
921: (8)             value with unbiased rounding
922: (4)           fractional : boolean, optional
923: (8)             If `True`, the cutoffs of `precision` and `min_digits` refer to the
924: (8)             total number of digits after the decimal point, including leading
925: (8)             zeros.
926: (8)             If `False`, `precision` and `min_digits` refer to the total number of
927: (8)             significant digits, before or after the decimal point, ignoring
leading
928: (8)             zeros.
929: (4)           trim : one of 'k', '.', '0', '-', optional
930: (8)             Controls post-processing trimming of trailing digits, as follows:
931: (8)             * 'k' : keep trailing zeros, keep decimal point (no trimming)
932: (8)             * '.' : trim all trailing zeros, leave decimal point
933: (8)             * '0' : trim all but the zero before the decimal point. Insert the
934: (10)            zero if it is missing.
935: (8)             * '-' : trim trailing zeros and any trailing decimal point
936: (4)           sign : boolean, optional
937: (8)             Whether to show the sign for positive values.
938: (4)           pad_left : non-negative integer, optional
939: (8)             Pad the left side of the string with whitespace until at least that
940: (8)             many characters are to the left of the decimal point.
941: (4)           pad_right : non-negative integer, optional
942: (8)             Pad the right side of the string with whitespace until at least that

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

943: (8)           many characters are to the right of the decimal point.
944: (4)           min_digits : non-negative integer or None, optional
945: (8)           Minimum number of digits to print. Only has an effect if `unique=True`
946: (8)           in which case additional digits past those necessary to uniquely
947: (8)           identify the value may be printed, rounding the last additional digit.
948: (8)           -- versionadded:: 1.21.0
949: (4)           Returns
950: (4)
951: (4)           rep : string
952: (8)           The string representation of the floating point value
953: (4)           See Also
954: (4)
955: (4)           -----
956: (4)           format_float_scientific
957: (4)           Examples
958: (4)           -----
959: (4)           >>> np.format_float_positional(np.float32(np.pi))
960: (4)           '3.1415927'
961: (4)           >>> np.format_float_positional(np.float16(np.pi))
962: (4)           '3.14'
963: (4)           >>> np.format_float_positional(np.float16(0.3))
964: (4)           '0.3'
965: (4)           >>> np.format_float_positional(np.float16(0.3), unique=False,
precision=10)
966: (4)           '0.3000488281'
967: (4)           """
968: (4)           precision = _none_or_positive_arg(precision, 'precision')
969: (4)           pad_left = _none_or_positive_arg(pad_left, 'pad_left')
970: (4)           pad_right = _none_or_positive_arg(pad_right, 'pad_right')
971: (4)           min_digits = _none_or_positive_arg(min_digits, 'min_digits')
972: (4)           if not fractional and precision == 0:
973: (8)               raise ValueError("precision must be greater than 0 if "
974: (4)                   "fractional=False")
975: (8)           if min_digits > 0 and precision > 0 and min_digits > precision:
976: (4)               raise ValueError("min_digits must be less than or equal to precision")
977: (30)          return dragon4_positional(x, precision=precision, unique=unique,
978: (30)                  fractional=fractional, trim=trim,
979: (30)                  sign=sign, pad_left=pad_left,
979: (30)                  pad_right=pad_right, min_digits=min_digits)

980: (0)           class IntegerFormat:
981: (4)             def __init__(self, data):
982: (8)               if data.size > 0:
983: (12)                 max_str_len = max(len(str(np.max(data))), len(str(np.min(data))))
984: (30)
985: (8)
986: (12)
987: (8)               self.format = '%{}d'.format(max_str_len)
988: (4)             def __call__(self, x):
989: (8)               return self.format % x
990: (0)           class BoolFormat:
991: (4)             def __init__(self, data, **kwargs):
992: (8)               self.truestr = ' True' if data.shape != () else 'True'
993: (4)             def __call__(self, x):
994: (8)               return self.truestr if x else "False"
995: (0)           class ComplexFloatingFormat:
996: (4)             """ Formatter for subtypes of np.complexfloating """
997: (4)             def __init__(self, x, precision, floatmode, suppress_small,
998: (17)                 sign=False, *, legacy=None):
999: (8)
1000: (12)               if isinstance(sign, bool):
1001: (8)                 sign = '+' if sign else '-'
1002: (8)               floatmode_real = floatmode_imag = floatmode
1003: (12)               if legacy <= 113:
1004: (12)                 floatmode_real = 'maxprec_equal'
1005: (8)                 floatmode_imag = 'maxprec'
1006: (12)                 self.real_format = FloatingFormat(
1007: (12)                   x.real, precision, floatmode_real, suppress_small,
1008: (8)                   sign=sign, legacy=legacy
1009: (8)
1010: (12)                 self.imag_format = FloatingFormat(
1010: (12)                   x.imag, precision, floatmode_imag, suppress_small,
```

```

1011: (12)                     sign='+', legacy=legacy
1012: (8)
1013: (4)
1014: (8)
1015: (8)
1016: (8)
1017: (8)
1018: (8)
1019: (0)
1020: (4)
1021: (8)
1022: (8)
1023: (12)
1024: (30)
1025: (8)
1026: (12)
1027: (8)
1028: (12)
1029: (8)
1030: (8)
1031: (4)
1032: (8)
1033: (4)
1034: (8)
1035: (12)
1036: (8)
1037: (12)
1038: (0)
1039: (4)
1040: (17)
1041: (8)
1042: (12)
1043: (16)
1044: (12)
1045: (16)
1046: (8)
1047: (12)
1048: (8)
1049: (8)
1050: (8)
1051: (8)
1052: (8)
1053: (4)
1054: (8)
1055: (12)
1056: (8)
1057: (4)
1058: (8)
1059: (36)
1060: (36)
1061: (36)
1062: (0)
1063: (4)
1064: (8)
1065: (0)
1066: (4)
1067: (8)
1068: (8)
1069: (8)
1070: (4)
1071: (8)
1072: (8)
1073: (4)
1074: (8)
1075: (12)
1076: (8)
1077: (12)
1078: (16)
1079: (16)

        )
        def __call__(self, x):
            r = self.real_format(x.real)
            i = self.imag_format(x.imag)
            sp = len(i.rstrip())
            i = i[:sp] + 'j' + i[sp:]
            return r + i

    class _TimelikeFormat:
        def __init__(self, data):
            non_nat = data[~isnat(data)]
            if len(non_nat) > 0:
                max_str_len = max(len(self._format_non_nat(np.max(non_nat))), len(self._format_non_nat(np.min(non_nat))))
            else:
                max_str_len = 0
            if len(non_nat) < data.size:
                max_str_len = max(max_str_len, 5)
            self._format = '%{}s'.format(max_str_len)
            self._nat = "'NaT'".rjust(max_str_len)
        def _format_non_nat(self, x):
            raise NotImplementedError
        def __call__(self, x):
            if isnat(x):
                return self._nat
            else:
                return self._format % self._format_non_nat(x)

    class DatetimeFormat(_TimelikeFormat):
        def __init__(self, x, unit=None, timezone=None, casting='same_kind',
                     legacy=False):
            if unit is None:
                if x.dtype.kind == 'M':
                    unit = datetime_data(x.dtype)[0]
                else:
                    unit = 's'
            if timezone is None:
                timezone = 'naive'
            self.timezone = timezone
            self.unit = unit
            self.casting = casting
            self.legacy = legacy
            super().__init__(x)
        def __call__(self, x):
            if self.legacy <= 113:
                return self._format_non_nat(x)
            return super().__call__(x)
        def _format_non_nat(self, x):
            return "%s" % datetime_as_string(x,
                                              unit=self.unit,
                                              timezone=self.timezone,
                                              casting=self.casting)

    class TimedeltaFormat(_TimelikeFormat):
        def _format_non_nat(self, x):
            return str(x.astype('i8'))

    class SubArrayFormat:
        def __init__(self, format_function, **options):
            self.format_function = format_function
            self.threshold = options['threshold']
            self.edge_items = options['edgeitems']
        def __call__(self, a):
            self.summary_insert = "..." if a.size > self.threshold else ""
            return self.format_array(a)
        def format_array(self, a):
            if np.ndim(a) == 0:
                return self.format_function(a)
            if self.summary_insert and a.shape[0] > 2*self.edge_items:
                formatted = (
                    [self.format_array(a_) for a_ in a[:self.edge_items]]
                    + [self.summary_insert]

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1080: (16) + [self.format_array(a_) for a_ in a[-self.edge_items:]]
1081: (12)
1082: (8)
1083: (12)     )
1084: (8)     else:
1085: (0)         formatted = [self.format_array(a_) for a_ in a]
1086: (4)     return "[" + ", ".join(formatted) + "]"
1087: (4) class StructuredVoidFormat:
1088: (4)     """
1089: (4)         Formatter for structured np.void objects.
1090: (4)         This does not work on structured alias types like np.dtype(('i4',
1091: (4)             'i2,i2')),,
1092: (4)         as alias scalars lose their field information, and the implementation
1093: (8)             relies upon np.void.__getitem__.
1094: (4)             """
1095: (4)     def __init__(self, format_functions):
1096: (8)         self.format_functions = format_functions
1097: (8)     @classmethod
1098: (4)     def from_data(cls, data, **options):
1099: (8)         """
1100: (8)             This is a second way to initialize StructuredVoidFormat, using the raw
1101: (8)             data
1102: (12)             as input. Added to avoid changing the signature of __init__.
1103: (8)             """
1104: (16)     format_functions = []
1105: (12)     for field_name in data.dtype.names:
1106: (8)         format_function = _get_format_function(data[field_name],
1107: (4)             **options)
1108: (8)     if data.dtype[field_name].shape != ():
1109: (12)         format_function = SubArrayFormat(format_function, **options)
1110: (12)         format_functions.append(format_function)
1111: (8)     return cls(format_functions)
1112: (4)     def __call__(self, x):
1113: (8)         str_fields = [
1114: (12)             format_function(field)
1115: (12)             for field, format_function in zip(x, self.format_functions)
1116: (8)         ]
1117: (4)         if len(str_fields) == 1:
1118: (8)             return "({},)".format(str_fields[0])
1119: (4)         else:
1120: (8)             return "({})".format(", ".join(str_fields))
1121: (4)     def _void_scalar_repr(x):
1122: (8)         """
1123: (4)             Implements the repr for structured-void scalars. It is called from the
1124: (4)             scalartypes.c.src code, and is placed here because it uses the elementwise
1125: (4)             formatters defined above.
1126: (4)             """
1127: (4)         return StructuredVoidFormat.from_data(array(x), **_format_options)(x)
1128: (0) _typelessdata = [int_, float_, complex_, bool_]
1129: (0) def dtype_is_implied(dtype):
1130: (4)     """
1131: (4)         Determine if the given dtype is implied by the representation of its
1132: (4)         values.
1133: (4)         Parameters
1134: (8)             -----
1135: (4)             dtype : dtype
1136: (8)                 Data type
1137: (4)             Returns
1138: (8)                 -----
1139: (4)             implied : bool
1140: (8)                 True if the dtype is implied by the representation of its values.
1141: (4)             Examples
1142: (8)                 -----
1143: (4)                 >>> np.core.arrayprint.dtype_is_implied(int)
1144: (4)                 True

```

```

1145: (4)
1146: (4)
1147: (4)
1148: (8)
1149: (4)
1150: (8)
1151: (4)
1152: (8)
1153: (4)
1154: (0)
1155: (4)
1156: (4)
1157: (4)
1158: (4)
1159: (4)
1160: (4)
1161: (4)
1162: (4)
1163: (8)
1164: (4)
1165: (8)
1166: (4)
1167: (8)
1168: (4)
1169: (4)
1170: (8)
1171: (4)
1172: (8)
1173: (4)
1174: (0)
1175: (8)
1176: (8)
1177: (4)
array2string."""
1178: (4)
1179: (8)
1180: (4)
1181: (8)
1182: (4)
1183: (8)
1184: (4)
1185: (4)
1186: (4)
1187: (4)
1188: (12)
1189: (8)
1190: (4)
1191: (8)
1192: (27)
1193: (4)
1194: (8)
1195: (4)
1196: (4)
1197: (8)
1198: (4)
1199: (4)
1200: (4)
1201: (4)
1202: (8)
1203: (12)
1204: (4)
1205: (8)
1206: (4)
1207: (0)
1208: (8)
1209: (4)
1210: (0)
1211: (0)
1212: (4)

    """
    dtype = np.dtype(dtype)
    if _format_options['legacy'] <= 113 and dtype.type == bool_:
        return False
    if dtype.names is not None:
        return False
    if not dtype.isnative:
        return False
    return dtype.type in _typelessdata
def dtype_short_repr(dtype):
    """
    Convert a dtype to a short form which evaluates to the same dtype.
    The intent is roughly that the following holds
    >>> from numpy import *
    >>> dt = np.int64([1, 2]).dtype
    >>> assert eval(dtype_short_repr(dt)) == dt
    """
    if type(dtype).__repr__ != np.dtype.__repr__:
        return repr(dtype)
    if dtype.names is not None:
        return str(dtype)
    elif issubclass(dtype.type, flexible):
        return "%s" % str(dtype)
    typename = dtype.name
    if not dtype.isnative:
        return "%s" % str(dtype)
    if typename and not (typename[0].isalpha() and typename.isalnum()):
        typename = repr(typename)
    return typename
def _array_repr_implementation(
    arr, max_line_width=None, precision=None, suppress_small=None,
    array2string=array2string):
    """Internal version of array_repr() that allows overriding
array2string."""
    if max_line_width is None:
        max_line_width = _format_options['linewidth']
    if type(arr) is not ndarray:
        class_name = type(arr).__name__
    else:
        class_name = "array"
    skipdtype = dtype_is_implied(arr.dtype) and arr.size > 0
    prefix = class_name + "("
    suffix = ")" if skipdtype else ","
    if (_format_options['legacy'] <= 113 and
        arr.shape == () and not arr.dtype.names):
        lst = repr(arr.item())
    elif arr.size > 0 or arr.shape == (0,):
        lst = array2string(arr, max_line_width, precision, suppress_small,
                           ', ', prefix, suffix=suffix)
    else: # show zero-length shape unless it is (0,)
        lst = "[], shape=%s" % (repr(arr.shape),)
    arr_str = prefix + lst + suffix
    if skipdtype:
        return arr_str
    dtype_str = "dtype={}".format(dtype_short_repr(arr.dtype))
    last_line_len = len(arr_str) - (arr_str.rfind('\n') + 1)
    spacer = " "
    if _format_options['legacy'] <= 113:
        if issubclass(arr.dtype.type, flexible):
            spacer = '\n' + '*'len(class_name + "(")
        elif last_line_len + len(dtype_str) + 1 > max_line_width:
            spacer = '\n' + '*'len(class_name + "(")
    return arr_str + spacer + dtype_str
def _array_repr_dispatcher(
    arr, max_line_width=None, precision=None, suppress_small=None):
    return (arr,)
@array_function_dispatch(_array_repr_dispatcher, module='numpy')
def array_repr(arr, max_line_width=None, precision=None, suppress_small=None):
    """

```

```

1213: (4)           Return the string representation of an array.
1214: (4)           Parameters
1215: (4)           -----
1216: (4)           arr : ndarray
1217: (8)             Input array.
1218: (4)           max_line_width : int, optional
1219: (8)             Inserts newlines if text is longer than `max_line_width`.
1220: (8)             Defaults to ``numpy.get_printoptions()['linewidth']``.
1221: (4)           precision : int, optional
1222: (8)             Floating point precision.
1223: (8)             Defaults to ``numpy.get_printoptions()['precision']``.
1224: (4)           suppress_small : bool, optional
1225: (8)             Represent numbers "very close" to zero as zero; default is False.
1226: (8)             Very close is defined by precision: if the precision is 8, e.g.,
1227: (8)               numbers smaller (in absolute value) than 5e-9 are represented as
1228: (8)               zero.
1229: (8)             Defaults to ``numpy.get_printoptions()['suppress']``.
1230: (4)           Returns
1231: (4)
1232: (4)           string : str
1233: (6)             The string representation of an array.
1234: (4)           See Also
1235: (4)
1236: (4)           array_str, array2string, set_printoptions
1237: (4)           Examples
1238: (4)
1239: (4)           >>> np.array_repr(np.array([1,2]))
1240: (4)           'array([1, 2])'
1241: (4)           >>> np.array_repr(np.ma.array([0.]))
1242: (4)           'MaskedArray([0.])'
1243: (4)           >>> np.array_repr(np.array([], np.int32))
1244: (4)           'array([], dtype=int32)'
1245: (4)           >>> x = np.array([1e-6, 4e-7, 2, 3])
1246: (4)           >>> np.array_repr(x, precision=6, suppress_small=True)
1247: (4)           'array([0.000001,  0.        ,  2.        ,  3.        ])'
1248: (4)           """
1249: (4)           return _array_repr_implementation(
1250: (8)             arr, max_line_width, precision, suppress_small)
1251: (0)           @_recursive_guard()
1252: (0)           def _guarded_repr_or_str(v):
1253: (4)             if isinstance(v, bytes):
1254: (8)               return repr(v)
1255: (4)             return str(v)
1256: (0)           def _array_str_implementation(
1257: (8)             a, max_line_width=None, precision=None, suppress_small=None,
1258: (8)             array2string=array2string):
1259: (4)             """Internal version of array_str() that allows overriding array2string."""
1260: (4)             if (_format_options['legacy'] <= 113 and
1261: (12)               a.shape == () and not a.dtype.names):
1262: (8)               return str(a.item())
1263: (4)             if a.shape == ():
1264: (8)               return _guarded_repr_or_str(np.ndarray.__getitem__(a, ()))
1265: (4)             return array2string(a, max_line_width, precision, suppress_small, ' ', '')
1266: (0)           def _array_str_dispatcher(
1267: (8)             a, max_line_width=None, precision=None, suppress_small=None):
1268: (4)             return (a,)
1269: (0)           @array_function_dispatch(_array_str_dispatcher, module='numpy')
1270: (0)           def array_str(a, max_line_width=None, precision=None, suppress_small=None):
1271: (4)             """
1272: (4)               Return a string representation of the data in an array.
1273: (4)               The data in the array is returned as a single string. This function is
1274: (4)               similar to `array_repr`, the difference being that `array_repr` also
1275: (4)               returns information on the kind of array and its data type.
1276: (4)               Parameters
1277: (4)               -----
1278: (4)               a : ndarray
1279: (8)                 Input array.
1280: (4)               max_line_width : int, optional
1281: (8)                 Inserts newlines if text is longer than `max_line_width`.

```

```

1282: (8)             Defaults to ``numpy.get_printoptions()['linewidth']``.
1283: (4)             precision : int, optional
1284: (8)                 Floating point precision.
1285: (8)                 Defaults to ``numpy.get_printoptions()['precision']``.
1286: (4)             suppress_small : bool, optional
1287: (8)                 Represent numbers "very close" to zero as zero; default is False.
1288: (8)                 Very close is defined by precision: if the precision is 8, e.g.,
1289: (8)                     numbers smaller (in absolute value) than 5e-9 are represented as
1290: (8)                     zero.
1291: (8)                 Defaults to ``numpy.get_printoptions()['suppress']``.
1292: (4)             See Also
1293: (4)             -----
1294: (4)                 array2string, array_repr, set_printoptions
1295: (4)             Examples
1296: (4)             -----
1297: (4)                 >>> np.array_str(np.arange(3))
1298: (4)                 '[0 1 2]'
1299: (4)                 """
1300: (4)                 return _array_str_implementation(
1301: (8)                     a, max_line_width, precision, suppress_small)
1302: (0)             _array2string_impl = getattr(array2string, '__wrapped__', array2string)
1303: (0)             _default_array_str = functools.partial(_array_str_implementation,
1304: (39)                         array2string=_array2string_impl)
1305: (0)             _default_array_repr = functools.partial(_array_repr_implementation,
1306: (40)                         array2string=_array2string_impl)
1307: (0)             def set_string_function(f, repr=True):
1308: (4)                 """
1309: (4)                 Set a Python function to be used when pretty printing arrays.
1310: (4)             Parameters
1311: (4)             -----
1312: (4)                 f : function or None
1313: (8)                 Function to be used to pretty print arrays. The function should expect
1314: (8)                 a single array argument and return a string of the representation of
1315: (8)                 the array. If None, the function is reset to the default NumPy
function
1316: (8)                 to print arrays.
1317: (4)             repr : bool, optional
1318: (8)                 If True (default), the function for pretty printing (`__repr__`)
1319: (8)                 is set, if False the function that returns the default string
1320: (8)                 representation (`__str__`) is set.
1321: (4)             See Also
1322: (4)             -----
1323: (4)                 set_printoptions, get_printoptions
1324: (4)             Examples
1325: (4)             -----
1326: (4)                 >>> def pprint(arr):
1327: (4)                     ...     return 'HA! - What are you going to do now?'
1328: (4)                     ...
1329: (4)                 >>> np.set_string_function(pprint)
1330: (4)                 >>> a = np.arange(10)
1331: (4)                 >>> a
1332: (4)                 HA! - What are you going to do now?
1333: (4)                 >>> _ = a
1334: (4)                 >>> # [0 1 2 3 4 5 6 7 8 9]
1335: (4)                 We can reset the function to the default:
1336: (4)                 >>> np.set_string_function(None)
1337: (4)                 >>> a
1338: (4)                 array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
`repr` affects either pretty printing or normal string representation.
1339: (4)                 Note that `__repr__` is still affected by setting `__str__`
1340: (4)                 because the width of each array element in the returned string becomes
1341: (4)                 equal to the length of the result of `__str__()` .
1342: (4)                 >>> x = np.arange(4)
1343: (4)                 >>> np.set_string_function(lambda x:'random', repr=False)
1344: (4)                 >>> x.__str__()
1345: (4)                 'random'
1346: (4)                 >>> x.__repr__()
1347: (4)                 'array([0, 1, 2, 3])'
1348: (4)                 """
1349: (4)

```

```

1350: (4)         if f is None:
1351: (8)             if repr:
1352: (12)                 return multiarray.set_string_function(_default_array_repr, 1)
1353: (8)             else:
1354: (12)                 return multiarray.set_string_function(_default_array_str, 0)
1355: (4)         else:
1356: (8)             return multiarray.set_string_function(f, repr)

```

---

## File 45 - cversions.py:

```

1: (0) """Simple script to compute the api hash of the current API.
2: (0) The API has is defined by numpy_api_order and ufunc_api_order.
3: (0)
4: (0) from os.path import dirname
5: (0) from code_generators.genapi import fullapi_hash
6: (0) from code_generators.numpy_api import full_api
7: (0) if __name__ == '__main__':
8: (4)     curdir = dirname(__file__)
9: (4)     print(fullapi_hash(full_api))

```

---

## File 46 - defchararray.py:

```

1: (0) """
2: (0) This module contains a set of functions for vectorized string
3: (0) operations and methods.
4: (0) .. note::
5: (3)     The `chararray` class exists for backwards compatibility with
6: (3)     Numarray, it is not recommended for new development. Starting from numpy
7: (3)     1.4, if one needs arrays of strings, it is recommended to use arrays of
8: (3)     `dtype` `object_`, `bytes_` or `str_`, and use the free functions
9: (3)     in the `numpy.char` module for fast vectorized string operations.
10: (0) Some methods will only be available if the corresponding string method is
11: (0) available in your version of Python.
12: (0) The preferred alias for `defchararray` is `numpy.char` .
13: (0)
14: (0) import functools
15: (0) from ..utils import set_module
16: (0) from .numericatypes import (
17: (4)     bytes_, str_, integer, int_, object_, bool_, character)
18: (0) from .numeric import ndarray, compare_chararrays
19: (0) from .numeric import array as narray
20: (0) from numpy.core.multiarray import _vec_string
21: (0) from numpy.core import overrides
22: (0) from numpy.compat import asbytes
23: (0) import numpy
24: (0) __all__ = [
25: (4)     'equal', 'not_equal', 'greater_equal', 'less_equal',
26: (4)     'greater', 'less', 'str_len', 'add', 'multiply', 'mod', 'capitalize',
27: (4)     'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
28: (4)     'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
29: (4)     'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
30: (4)     'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
31: (4)     'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
32: (4)     'title', 'translate', 'upper', 'zfill', 'isnumeric', 'isdecimal',
33: (4)     'array', 'asarray'
34: (4) ]
35: (0) _globalvar = 0
36: (0) array_function_dispatch = functools.partial(
37: (4)     overrides.array_function_dispatch, module='numpy.char')
38: (0) def _is_unicode(arr):
39: (4)     """Returns True if arr is a string or a string array with a dtype that
40: (4)     represents a unicode string, otherwise returns False.
41: (4)     """
42: (4)     if (isinstance(arr, str) or
43: (12)         issubclass(numpy.asarray(arr).dtype.type, str)):

```

```

44: (8)           return True
45: (4)           return False
46: (0) def _to_bytes_or_str_array(result, output_dtype_like=None):
47: (4)     """
48: (4)         Helper function to cast a result back into an array
49: (4)         with the appropriate dtype if an object array must be used
50: (4)         as an intermediary.
51: (4)     """
52: (4)     ret = numpy.asarray(result.tolist())
53: (4)     dtype = getattr(output_dtype_like, 'dtype', None)
54: (4)     if dtype is not None:
55: (8)         return ret.astype(type(dtype)(_get_num_chars(ret)), copy=False)
56: (4)     return ret
57: (0) def _clean_args(*args):
58: (4)     """
59: (4)         Helper function for delegating arguments to Python string
60: (4)         functions.
61: (4)         Many of the Python string operations that have optional arguments
62: (4)         do not use 'None' to indicate a default value. In these cases,
63: (4)         we need to remove all None arguments, and those following them.
64: (4)     """
65: (4)     newargs = []
66: (4)     for chk in args:
67: (8)         if chk is None:
68: (12)             break
69: (8)         newargs.append(chk)
70: (4)     return newargs
71: (0) def _get_num_chars(a):
72: (4)     """
73: (4)         Helper function that returns the number of characters per field in
74: (4)         a string or unicode array. This is to abstract out the fact that
75: (4)         for a unicode array this is itemsize / 4.
76: (4)     """
77: (4)     if issubclass(a.dtype.type, str_):
78: (8)         return a.itemsize // 4
79: (4)     return a.itemsize
80: (0) def _binary_op_dispatcher(x1, x2):
81: (4)     return (x1, x2)
82: (0) @array_function_dispatch(_binary_op_dispatcher)
83: (0) def equal(x1, x2):
84: (4)     """
85: (4)         Return (x1 == x2) element-wise.
86: (4)         Unlike `numpy.equal`, this comparison is performed by first
87: (4)         stripping whitespace characters from the end of the string. This
88: (4)         behavior is provided for backward-compatibility with numarray.
89: (4)         Parameters
90: (4)         -----
91: (4)         x1, x2 : array_like of str or unicode
92: (8)             Input arrays of the same shape.
93: (4)         Returns
94: (4)         -----
95: (4)         out : ndarray
96: (8)             Output array of bools.
97: (4)         See Also
98: (4)         -----
99: (4)         not_equal, greater_equal, less_equal, greater, less
100: (4)     """
101: (4)     return compare_chararrays(x1, x2, '==', True)
102: (0) @array_function_dispatch(_binary_op_dispatcher)
103: (0) def not_equal(x1, x2):
104: (4)     """
105: (4)         Return (x1 != x2) element-wise.
106: (4)         Unlike `numpy.not_equal`, this comparison is performed by first
107: (4)         stripping whitespace characters from the end of the string. This
108: (4)         behavior is provided for backward-compatibility with numarray.
109: (4)         Parameters
110: (4)         -----
111: (4)         x1, x2 : array_like of str or unicode
112: (8)             Input arrays of the same shape.

```

```
113: (4)             Returns
114: (4)             -----
115: (4)             out : ndarray
116: (8)             Output array of bools.
117: (4)             See Also
118: (4)             -----
119: (4)             equal, greater_equal, less_equal, greater, less
120: (4)             """
121: (4)             return compare_chararrays(x1, x2, '!=', True)
122: (0)             @array_function_dispatch(_binary_op_dispatcher)
123: (0)             def greater_equal(x1, x2):
124: (4)             """
125: (4)             Return (x1 >= x2) element-wise.
126: (4)             Unlike `numpy.greater_equal`, this comparison is performed by
127: (4)             first stripping whitespace characters from the end of the string.
128: (4)             This behavior is provided for backward-compatibility with
129: (4)             numarray.
130: (4)             Parameters
131: (4)             -----
132: (4)             x1, x2 : array_like of str or unicode
133: (8)             Input arrays of the same shape.
134: (4)             Returns
135: (4)             -----
136: (4)             out : ndarray
137: (8)             Output array of bools.
138: (4)             See Also
139: (4)             -----
140: (4)             equal, not_equal, less_equal, greater, less
141: (4)             """
142: (4)             return compare_chararrays(x1, x2, '>=', True)
143: (0)             @array_function_dispatch(_binary_op_dispatcher)
144: (0)             def less_equal(x1, x2):
145: (4)             """
146: (4)             Return (x1 <= x2) element-wise.
147: (4)             Unlike `numpy.less_equal`, this comparison is performed by first
148: (4)             stripping whitespace characters from the end of the string. This
149: (4)             behavior is provided for backward-compatibility with numarray.
150: (4)             Parameters
151: (4)             -----
152: (4)             x1, x2 : array_like of str or unicode
153: (8)             Input arrays of the same shape.
154: (4)             Returns
155: (4)             -----
156: (4)             out : ndarray
157: (8)             Output array of bools.
158: (4)             See Also
159: (4)             -----
160: (4)             equal, not_equal, greater_equal, greater, less
161: (4)             """
162: (4)             return compare_chararrays(x1, x2, '<=', True)
163: (0)             @array_function_dispatch(_binary_op_dispatcher)
164: (0)             def greater(x1, x2):
165: (4)             """
166: (4)             Return (x1 > x2) element-wise.
167: (4)             Unlike `numpy.greater`, this comparison is performed by first
168: (4)             stripping whitespace characters from the end of the string. This
169: (4)             behavior is provided for backward-compatibility with numarray.
170: (4)             Parameters
171: (4)             -----
172: (4)             x1, x2 : array_like of str or unicode
173: (8)             Input arrays of the same shape.
174: (4)             Returns
175: (4)             -----
176: (4)             out : ndarray
177: (8)             Output array of bools.
178: (4)             See Also
179: (4)             -----
180: (4)             equal, not_equal, greater_equal, less_equal, less
181: (4)             """
```

```

182: (4)           return compare_chararrays(x1, x2, '>', True)
183: (0)           @array_function_dispatch(_binary_op_dispatcher)
184: (0)           def less(x1, x2):
185: (4)             """
186: (4)               Return (x1 < x2) element-wise.
187: (4)               Unlike `numpy.greater`, this comparison is performed by first
188: (4)               stripping whitespace characters from the end of the string. This
189: (4)               behavior is provided for backward-compatibility with numarray.
190: (4)               Parameters
191: (4)                 -----
192: (4)                 x1, x2 : array_like of str or unicode
193: (8)                   Input arrays of the same shape.
194: (4)               Returns
195: (4)                 -----
196: (4)                 out : ndarray
197: (8)                   Output array of bools.
198: (4)               See Also
199: (4)                 -----
200: (4)                 equal, not_equal, greater_equal, less_equal, greater
201: (4)               """
202: (4)           return compare_chararrays(x1, x2, '<', True)
203: (0)           def _unary_op_dispatcher(a):
204: (4)             return (a,)
205: (0)           @array_function_dispatch(_unary_op_dispatcher)
206: (0)           def str_len(a):
207: (4)             """
208: (4)               Return len(a) element-wise.
209: (4)               Parameters
210: (4)                 -----
211: (4)                 a : array_like of str or unicode
212: (4)               Returns
213: (4)                 -----
214: (4)                 out : ndarray
215: (8)                   Output array of integers
216: (4)               See Also
217: (4)                 -----
218: (4)                 len
219: (4)               Examples
220: (4)                 -----
221: (4)                 >>> a = np.array(['Grace Hopper Conference', 'Open Source Day'])
222: (4)                 >>> np.char.str_len(a)
223: (4)                 array([23, 15])
224: (4)                 >>> a = np.array([u'\u0420', u'\u043e'])
225: (4)                 >>> np.char.str_len(a)
226: (4)                 array([1, 1])
227: (4)                 >>> a = np.array([[['hello', 'world'], [u'\u0420', u'\u043e']]])
228: (4)                 >>> np.char.str_len(a)
229: (4)                 array([[5, 5], [1, 1]])
230: (4)               """
231: (4)           return _vec_string(a, int_, '__len__')
232: (0)           @array_function_dispatch(_binary_op_dispatcher)
233: (0)           def add(x1, x2):
234: (4)             """
235: (4)               Return element-wise string concatenation for two arrays of str or unicode.
236: (4)               Arrays `x1` and `x2` must have the same shape.
237: (4)               Parameters
238: (4)                 -----
239: (4)                 x1 : array_like of str or unicode
240: (8)                   Input array.
241: (4)                 x2 : array_like of str or unicode
242: (8)                   Input array.
243: (4)               Returns
244: (4)                 -----
245: (4)                 add : ndarray
246: (8)                   Output array of `bytes_` or `str_`, depending on input types
247: (8)                   of the same shape as `x1` and `x2`.
248: (4)               """
249: (4)               arr1 = numpy.asarray(x1)
250: (4)               arr2 = numpy.asarray(x2)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

251: (4)          out_size = _get_num_chars(arr1) + _get_num_chars(arr2)
252: (4)          if type(arr1.dtype) != type(arr2.dtype):
253: (8)              raise TypeError(
254: (12)                  "np.char.add() requires both arrays of the same dtype kind, but "
255: (12)                  f"got dtypes: '{arr1.dtype}' and '{arr2.dtype}' (the few cases "
256: (12)                  "where this used to work often lead to incorrect results).")
257: (4)          return _vec_string(arr1, type(arr1.dtype)(out_size), '__add__', (arr2,))
258: (0)          def _multiply_dispatcher(a, i):
259: (4)              return (a,)
260: (0)          @array_function_dispatch(_multiply_dispatcher)
261: (0)          def multiply(a, i):
262: (4)              """
263: (4)                  Return (a * i), that is string multiple concatenation,
264: (4)                  element-wise.
265: (4)                  Values in `i` of less than 0 are treated as 0 (which yields an
266: (4)                  empty string).
267: (4)                  Parameters
268: (4)                      -----
269: (4)                      a : array_like of str or unicode
270: (4)                      i : array_like of ints
271: (4)                  Returns
272: (4)                      -----
273: (4)                      out : ndarray
274: (8)                          Output array of str or unicode, depending on input types
275: (4)                  Examples
276: (4)                      -----
277: (4)                      >>> a = np.array(["a", "b", "c"])
278: (4)                      >>> np.char.multiply(x, 3)
279: (4)                      array(['aaa', 'bbb', 'ccc'], dtype='<U3')
280: (4)                      >>> i = np.array([1, 2, 3])
281: (4)                      >>> np.char.multiply(a, i)
282: (4)                      array(['a', 'bb', 'ccc'], dtype='<U3')
283: (4)                      >>> np.char.multiply(np.array(['a']), i)
284: (4)                      array(['a', 'aa', 'aaa'], dtype='<U3')
285: (4)                      >>> a = np.array(['a', 'b', 'c', 'd', 'e', 'f']).reshape((2, 3))
286: (4)                      >>> np.char.multiply(a, 3)
287: (4)                      array([['aaa', 'bbb', 'ccc'],
288: (11)                          ['ddd', 'eee', 'fff']], dtype='<U3')
289: (4)                      >>> np.char.multiply(a, i)
290: (4)                      array([['a', 'bb', 'ccc'],
291: (11)                          ['d', 'ee', 'fff']], dtype='<U3')
292: (4)                      """
293: (4)                      a_arr = numpy.asarray(a)
294: (4)                      i_arr = numpy.asarray(i)
295: (4)                      if not issubclass(i_arr.dtype.type, integer):
296: (8)                          raise ValueError("Can only multiply by integers")
297: (4)                      out_size = _get_num_chars(a_arr) * max(int(i_arr.max()), 0)
298: (4)                      return _vec_string(
299: (8)                          a_arr, type(a_arr.dtype)(out_size), '__mul__', (i_arr,))
300: (0)          def _mod_dispatcher(a, values):
301: (4)              return (a, values)
302: (0)          @array_function_dispatch(_mod_dispatcher)
303: (0)          def mod(a, values):
304: (4)              """
305: (4)                  Return (a % i), that is pre-Python 2.6 string formatting
306: (4)                  (interpolation), element-wise for a pair of array_likes of str
307: (4)                  or unicode.
308: (4)                  Parameters
309: (4)                      -----
310: (4)                      a : array_like of str or unicode
311: (4)                      values : array_like of values
312: (7)                          These values will be element-wise interpolated into the string.
313: (4)                  Returns
314: (4)                      -----
315: (4)                      out : ndarray
316: (8)                          Output array of str or unicode, depending on input types
317: (4)                  See Also
318: (4)                      -----
319: (4)                      str.__mod__

```

```

320: (4)
321: (4)
322: (8)
323: (0)
324: (0)
325: (4)
326: (4)
327: (4)
328: (4)
329: (4)
330: (4)
331: (4)
332: (4)
333: (8)
334: (4)
335: (4)
336: (4)
337: (8)
338: (8)
339: (4)
340: (4)
341: (4)
342: (4)
343: (4)
344: (4)
345: (4)
346: (8)
347: (4)
348: (4)
349: (8)
350: (4)
351: (4)
352: (4)
353: (0)
354: (4)
355: (0)
356: (0)
357: (4)
358: (4)
359: (4)
360: (4)
361: (4)
362: (4)
363: (4)
364: (4)
365: (8)
366: (4)
367: (8)
368: (4)
369: (4)
370: (4)
371: (8)
372: (8)
373: (4)
374: (4)
375: (4)
376: (4)
377: (4)
378: (4)
379: (4)
380: (4)
381: (4)
382: (4)
383: (4)
384: (4)
385: (4)
386: (4)
387: (4)
388: (4)

        """
        return _to_bytes_or_str_array(
            _vec_string(a, object_, '__mod__', (values,)), a)
    @array_function_dispatch(_unary_op_dispatcher)
def capitalize(a):
    """
    Return a copy of `a` with only the first character of each element
    capitalized.
    Calls `str.capitalize` element-wise.
    For 8-bit strings, this method is locale-dependent.
    Parameters
    -----
    a : array_like of str or unicode
        Input array of strings to capitalize.
    Returns
    -----
    out : ndarray
        Output array of str or unicode, depending on input
        types
    See Also
    -----
    str.capitalize
    Examples
    -----
    >>> c = np.array(['a1b2','1b2a','b2a1','2a1b'],'S4'); c
    array(['a1b2', '1b2a', 'b2a1', '2a1b'],
          dtype='|S4')
    >>> np.char.capitalize(c)
    array(['A1b2', '1b2a', 'B2a1', '2a1b'],
          dtype='|S4')
    """
    a_arr = numpy.asarray(a)
    return _vec_string(a_arr, a_arr.dtype, 'capitalize')
def _center_dispatcher(a, width, fillchar=None):
    return (a,)
@array_function_dispatch(_center_dispatcher)
def center(a, width, fillchar=' '):
    """
    Return a copy of `a` with its elements centered in a string of
    length `width`.
    Calls `str.center` element-wise.
    Parameters
    -----
    a : array_like of str or unicode
    width : int
        The length of the resulting strings
    fillchar : str or unicode, optional
        The padding character to use (default is space).
    Returns
    -----
    out : ndarray
        Output array of str or unicode, depending on input
        types
    See Also
    -----
    str.center
    Notes
    -----
    This function is intended to work with arrays of strings. The
    fill character is not applied to numeric types.
    Examples
    -----
    >>> c = np.array(['a1b2','1b2a','b2a1','2a1b']); c
    array(['a1b2', '1b2a', 'b2a1', '2a1b'], dtype='<U4')
    >>> np.char.center(c, width=9)
    array(['  a1b2 ', ' 1b2a ', ' b2a1 ', ' 2a1b '], dtype='<U9')
    >>> np.char.center(c, width=9, fillchar='*')
    array(['***a1b2***', '***1b2a***', '***b2a1***', '***2a1b***'], dtype='<U9')
    >>> np.char.center(c, width=1)

```

```

389: (4)             array(['a', '1', 'b', '2'], dtype='<U1')
390: (4)
391: (4)             """
392: (4)             a_arr = numpy.asarray(a)
393: (4)             width_arr = numpy.asarray(width)
394: (4)             size = int(numpy.max(width_arr.flat))
395: (8)             if numpy.issubdtype(a_arr.dtype, numpy.bytes_):
396: (4)                 fillchar = asbytes(fillchar)
397: (8)             return _vec_string(
398: (0)                 a_arr, type(a_arr.dtype)(size), 'center', (width_arr, fillchar))
399: (4)
400: (0)             def _count_dispatcher(a, sub, start=None, end=None):
401: (0)                 return (a,)
402: (4)             @array_function_dispatch(_count_dispatcher)
403: (0)             def count(a, sub, start=0, end=None):
404: (4)                 """
405: (4)                 Returns an array with the number of non-overlapping occurrences of
406: (4)                 substring `sub` in the range [`start`, `end`].
407: (4)                 Calls `str.count` element-wise.
408: (4)                 Parameters
409: (4)                 -----
410: (4)                 a : array_like of str or unicode
411: (4)                 sub : str or unicode
412: (7)                     The substring to search for.
413: (7)                 start, end : int, optional
414: (4)                     Optional arguments `start` and `end` are interpreted as slice
415: (4)                     notation to specify the range in which to count.
416: (4)             Returns
417: (4)             -----
418: (4)             out : ndarray
419: (4)                 Output array of ints.
420: (4)             See Also
421: (4)             -----
422: (4)             Examples
423: (4)             -----
424: (4)             >>> c = np.array(['aAaAaA', ' aA ', 'abBABba'])
425: (4)             >>> c
426: (4)             array(['aAaAaA', ' aA ', 'abBABba'], dtype='<U7')
427: (4)             >>> np.char.count(c, 'A')
428: (4)             array([3, 1, 1])
429: (4)             >>> np.char.count(c, 'aA')
430: (4)             array([3, 1, 0])
431: (4)             >>> np.char.count(c, 'A', start=1, end=4)
432: (4)             array([2, 1, 1])
433: (4)             >>> np.char.count(c, 'A', start=1, end=3)
434: (4)             array([1, 0, 0])
435: (4)             """
436: (0)             return _vec_string(a, int_, 'count', [sub, start] + _clean_args(end))
437: (4)
438: (0)             def _code_dispatcher(a, encoding=None, errors=None):
439: (0)                 return (a,)
440: (4)             @array_function_dispatch(_code_dispatcher)
441: (4)             def decode(a, encoding=None, errors=None):
442: (4)                 """
443: (4)                 Calls ``bytes.decode`` element-wise.
444: (4)                 The set of available codecs comes from the Python standard library,
445: (4)                 and may be extended at runtime. For more information, see the
446: (4)                 :mod:`codecs` module.
447: (4)                 Parameters
448: (4)                 -----
449: (7)                 a : array_like of str or unicode
450: (4)                 encoding : str, optional
451: (7)                     The name of an encoding
452: (4)                 errors : str, optional
453: (7)                     Specifies how to handle encoding errors
454: (4)             Returns
455: (4)             -----
456: (4)             out : ndarray
457: (4)             See Also
458: (4)             -----
459: (4)             :py:meth:`bytes.decode`
```

```

458: (4)          Notes
459: (4)          -----
460: (4)          The type of the result will depend on the encoding specified.
461: (4)          Examples
462: (4)          -----
463: (4)          >>> c = np.array([b'\x81\xc1\x81\xc1\x81\xc1', b'@@\x81\xc1@@',
464: (4)          ...           b'\x81\x82\xc2\xc1\xc2\x82\x81'])
465: (4)          >>> c
466: (4)          array([b'\x81\xc1\x81\xc1\x81\xc1', b'@@\x81\xc1@@',
467: (4)          ...           b'\x81\x82\xc2\xc1\xc2\x82\x81'], dtype='|S7')
468: (4)          >>> np.char.decode(c, encoding='cp037')
469: (4)          array(['aAaAaA', 'aA ', 'abBABba'], dtype='<U7')
470: (4)          """
471: (4)          return _to_bytes_or_str_array(
472: (8)            _vec_string(a, object_, 'decode', _clean_args(encoding, errors)))
473: (0)          @array_function_dispatch(_code_dispatcher)
474: (0)          def encode(a, encoding=None, errors=None):
475: (4)          """
476: (4)          Calls `str.encode` element-wise.
477: (4)          The set of available codecs comes from the Python standard library,
478: (4)          and may be extended at runtime. For more information, see the codecs
479: (4)          module.
480: (4)          Parameters
481: (4)          -----
482: (4)          a : array_like of str or unicode
483: (4)          encoding : str, optional
484: (7)            The name of an encoding
485: (4)          errors : str, optional
486: (7)            Specifies how to handle encoding errors
487: (4)          Returns
488: (4)          -----
489: (4)          out : ndarray
490: (4)          See Also
491: (4)          -----
492: (4)          str.encode
493: (4)          Notes
494: (4)          -----
495: (4)          The type of the result will depend on the encoding specified.
496: (4)          """
497: (4)          return _to_bytes_or_str_array(
498: (8)            _vec_string(a, object_, 'encode', _clean_args(encoding, errors)))
499: (0)          @array_function_dispatch(_endswith_dispatcher)
500: (4)          def _endswith_dispatcher(a, suffix, start=None, end=None):
501: (4)            return (a,)
502: (0)          @array_function_dispatch(_endswith_dispatcher)
503: (0)          def endswith(a, suffix, start=0, end=None):
504: (4)          """
505: (4)          Returns a boolean array which is `True` where the string element
506: (4)          in `a` ends with `suffix`, otherwise `False`.
507: (4)          Calls `str.endswith` element-wise.
508: (4)          Parameters
509: (4)          -----
510: (4)          a : array_like of str or unicode
511: (4)          suffix : str
512: (4)          start, end : int, optional
513: (8)            With optional `start`, test beginning at that position. With
514: (8)            optional `end`, stop comparing at that position.
515: (4)          Returns
516: (4)          -----
517: (8)          out : ndarray
518: (4)            Outputs an array of bools.
519: (4)          See Also
520: (4)          -----
521: (4)          str.endswith
522: (4)          Examples
523: (4)          >>> s = np.array(['foo', 'bar'])
524: (4)          >>> s[0] = 'foo'
525: (4)          >>> s[1] = 'bar'
526: (4)          >>> s

```

```

527: (4)             array(['foo', 'bar'], dtype='<U3')
528: (4)             >>> np.char.endswith(s, 'ar')
529: (4)             array([False, True])
530: (4)             >>> np.char.endswith(s, 'a', start=1, end=2)
531: (4)             array([False, True])
532: (4)             """
533: (4)             return _vec_string(
534: (8)                 a, bool_, 'endswith', [suffix, start] + _clean_args(end))
535: (0)             def _expandtabs_dispatcher(a, tabsize=None):
536: (4)                 return (a,)
537: (0)             @array_function_dispatch(_expandtabs_dispatcher)
538: (0)             def expandtabs(a, tabsize=8):
539: (4)                 """
540: (4)                 Return a copy of each string element where all tab characters are
541: (4)                 replaced by one or more spaces.
542: (4)                 Calls `str.expandtabs` element-wise.
543: (4)                 Return a copy of each string element where all tab characters are
544: (4)                 replaced by one or more spaces, depending on the current column
545: (4)                 and the given `tabsize`. The column number is reset to zero after
546: (4)                 each newline occurring in the string. This doesn't understand other
547: (4)                 non-printing characters or escape sequences.
548: (4)             Parameters
549: (4)             -----
550: (4)             a : array_like of str or unicode
551: (8)                 Input array
552: (4)             tabsize : int, optional
553: (8)                 Replace tabs with `tabsize` number of spaces. If not given defaults
554: (8)                 to 8 spaces.
555: (4)             Returns
556: (4)             -----
557: (4)             out : ndarray
558: (8)                 Output array of str or unicode, depending on input type
559: (4)             See Also
560: (4)             -----
561: (4)             str.expandtabs
562: (4)             """
563: (4)             return _to_bytes_or_str_array(
564: (8)                 _vec_string(a, object_, 'expandtabs', (tabsize,)), a)
565: (0)             @array_function_dispatch(_count_dispatcher)
566: (0)             def find(a, sub, start=0, end=None):
567: (4)                 """
568: (4)                 For each element, return the lowest index in the string where
569: (4)                 substring `sub` is found.
570: (4)                 Calls `str.find` element-wise.
571: (4)                 For each element, return the lowest index in the string where
572: (4)                 substring `sub` is found, such that `sub` is contained in the
573: (4)                 range [`start`, `end`].
574: (4)             Parameters
575: (4)             -----
576: (4)             a : array_like of str or unicode
577: (4)             sub : str or unicode
578: (4)             start, end : int, optional
579: (8)                 Optional arguments `start` and `end` are interpreted as in
580: (8)                 slice notation.
581: (4)             Returns
582: (4)             -----
583: (4)             out : ndarray or int
584: (8)                 Output array of ints. Returns -1 if `sub` is not found.
585: (4)             See Also
586: (4)             -----
587: (4)             str.find
588: (4)             Examples
589: (4)             -----
590: (4)             >>> a = np.array(["NumPy is a Python library"])
591: (4)             >>> np.char.find(a, "Python", start=0, end=None)
592: (4)             array([11])
593: (4)             """
594: (4)             return _vec_string(
595: (8)                 a, int_, 'find', [sub, start] + _clean_args(end))

```

```
596: (0)
597: (0)
598: (4)
599: (4)
600: (4)
601: (4)
602: (4)
603: (4)
604: (4)
605: (4)
606: (4)
607: (4)
608: (4)
609: (8)
610: (4)
611: (4)
612: (4)
613: (4)
614: (4)
615: (4)
616: (4)
617: (4)
618: (4)
619: (4)
620: (8)
621: (0)
622: (0)
def index(a, sub, start=0, end=None):
    """
        Like `find`, but raises `ValueError` when the substring is not found.
        Calls `str.index` element-wise.
    Parameters
    -----
        a : array_like of str or unicode
        sub : str or unicode
        start, end : int, optional
    Returns
    -----
        out : ndarray
            Output array of ints. Returns -1 if `sub` is not found.
    See Also
    -----
        find, str.find
    Examples
    -----
        >>> a = np.array(["Computer Science"])
        >>> np.char.index(a, "Science", start=0, end=None)
        array([9])
    """
    return _vec_string(
        a, int_, 'index', [sub, start] + _clean_args(end))
@array_function_dispatch(_unary_op_dispatcher)
def isalnum(a):
    """
        Returns true for each element if all characters in the string are
        alphanumeric and there is at least one character, false otherwise.
        Calls `str.isalnum` element-wise.
        For 8-bit strings, this method is locale-dependent.
    Parameters
    -----
        a : array_like of str or unicode
    Returns
    -----
        out : ndarray
            Output array of str or unicode, depending on input type
    See Also
    -----
        str.isalnum
    """
    return _vec_string(a, bool_, 'isalnum')
@array_function_dispatch(_unary_op_dispatcher)
def isalpha(a):
    """
        Returns true for each element if all characters in the string are
        alphabetic and there is at least one character, false otherwise.
        Calls `str.isalpha` element-wise.
        For 8-bit strings, this method is locale-dependent.
    Parameters
    -----
        a : array_like of str or unicode
    Returns
    -----
        out : ndarray
            Output array of bools
    See Also
    -----
        str.isalpha
    """
    return _vec_string(a, bool_, 'isalpha')
@array_function_dispatch(_unary_op_dispatcher)
def isdigit(a):
    """
        Returns true for each element if all characters in the string are
        digits and there is at least one character, false otherwise.
        Calls `str.isdigit` element-wise.
    
```

```

665: (4)             For 8-bit strings, this method is locale-dependent.
666: (4)             Parameters
667: (4)             -----
668: (4)             a : array_like of str or unicode
669: (4)             Returns
670: (4)             -----
671: (4)             out : ndarray
672: (8)               Output array of bools
673: (4)             See Also
674: (4)             -----
675: (4)             str.isdigit
676: (4)             Examples
677: (4)             -----
678: (4)             >>> a = np.array(['a', 'b', '0'])
679: (4)             >>> np.char.isdigit(a)
680: (4)             array([False, False, True])
681: (4)             >>> a = np.array([('a', 'b', '0'), ('c', '1', '2')])
682: (4)             >>> np.char.isdigit(a)
683: (4)             array([[False, False, True], [False, True, True]])
684: (4)             """
685: (4)             return _vec_string(a, bool_, 'isdigit')
686: (0)             @array_function_dispatch(_unary_op_dispatcher)
687: (0)             def islower(a):
688: (4)               """
689: (4)               Returns true for each element if all cased characters in the
690: (4)               string are lowercase and there is at least one cased character,
691: (4)               false otherwise.
692: (4)               Calls `str.islower` element-wise.
693: (4)               For 8-bit strings, this method is locale-dependent.
694: (4)               Parameters
695: (4)               -----
696: (4)               a : array_like of str or unicode
697: (4)               Returns
698: (4)               -----
699: (4)               out : ndarray
700: (8)                 Output array of bools
701: (4)               See Also
702: (4)               -----
703: (4)               str.islower
704: (4)               """
705: (4)               return _vec_string(a, bool_, 'islower')
706: (0)             @array_function_dispatch(_unary_op_dispatcher)
707: (0)             def isspace(a):
708: (4)               """
709: (4)               Returns true for each element if there are only whitespace
710: (4)               characters in the string and there is at least one character,
711: (4)               false otherwise.
712: (4)               Calls `str.isspace` element-wise.
713: (4)               For 8-bit strings, this method is locale-dependent.
714: (4)               Parameters
715: (4)               -----
716: (4)               a : array_like of str or unicode
717: (4)               Returns
718: (4)               -----
719: (4)               out : ndarray
720: (8)                 Output array of bools
721: (4)               See Also
722: (4)               -----
723: (4)               str.isspace
724: (4)               """
725: (4)               return _vec_string(a, bool_, 'isspace')
726: (0)             @array_function_dispatch(_unary_op_dispatcher)
727: (0)             def istitle(a):
728: (4)               """
729: (4)               Returns true for each element if the element is a titlecased
730: (4)               string and there is at least one character, false otherwise.
731: (4)               Call `str.istitle` element-wise.
732: (4)               For 8-bit strings, this method is locale-dependent.
733: (4)               Parameters

```

```

734: (4)      -----
735: (4)      a : array_like of str or unicode
736: (4)      Returns
737: (4)      -----
738: (4)      out : ndarray
739: (8)      Output array of bools
740: (4)      See Also
741: (4)      -----
742: (4)      str.istitle
743: (4)      """
744: (4)      return _vec_string(a, bool_, 'istitle')
745: (0)      @array_function_dispatch(_unary_op_dispatcher)
746: (0)      def isupper(a):
747: (4)      """
748: (4)      Return true for each element if all cased characters in the
749: (4)      string are uppercase and there is at least one character, false
750: (4)      otherwise.
751: (4)      Call `str.isupper` element-wise.
752: (4)      For 8-bit strings, this method is locale-dependent.
753: (4)      Parameters
754: (4)      -----
755: (4)      a : array_like of str or unicode
756: (4)      Returns
757: (4)      -----
758: (4)      out : ndarray
759: (8)      Output array of bools
760: (4)      See Also
761: (4)      -----
762: (4)      str.isupper
763: (4)      Examples
764: (4)      -----
765: (4)      >>> str = "GHC"
766: (4)      >>> np.char.isupper(str)
767: (4)      array(True)
768: (4)      >>> a = np.array(["hello", "HELLO", "Hello"])
769: (4)      >>> np.char.isupper(a)
770: (4)      array([False, True, False])
771: (4)      """
772: (4)      return _vec_string(a, bool_, 'isupper')
773: (0)      def _join_dispatcher(sep, seq):
774: (4)      return (sep, seq)
775: (0)      @array_function_dispatch(_join_dispatcher)
776: (0)      def join(sep, seq):
777: (4)      """
778: (4)      Return a string which is the concatenation of the strings in the
779: (4)      sequence `seq`.
780: (4)      Calls `str.join` element-wise.
781: (4)      Parameters
782: (4)      -----
783: (4)      sep : array_like of str or unicode
784: (4)      seq : array_like of str or unicode
785: (4)      Returns
786: (4)      -----
787: (4)      out : ndarray
788: (8)      Output array of str or unicode, depending on input types
789: (4)      See Also
790: (4)      -----
791: (4)      str.join
792: (4)      Examples
793: (4)      -----
794: (4)      >>> np.char.join('-', 'osd')
795: (4)      array('o-s-d', dtype='<U5')
796: (4)      >>> np.char.join(['-', '.'], ['ghc', 'osd'])
797: (4)      array(['g-h-c', 'o.s.d'], dtype='<U5')
798: (4)      """
799: (4)      return _to_bytes_or_str_array(
800: (8)          _vec_string(sep, object_, 'join', (seq,)), seq)
801: (0)      def _just_dispatcher(a, width, fillchar=None):
802: (4)      return (a,)
```

```

803: (0)
804: (0)
805: (4)
806: (4)
807: (4)
808: (4)
809: (4)
810: (4)
811: (4)
812: (4)
813: (8)
814: (4)
815: (8)
816: (4)
817: (4)
818: (4)
819: (8)
820: (4)
821: (4)
822: (4)
823: (4)
824: (4)
825: (4)
826: (4)
827: (4)
828: (8)
829: (4)
830: (8)
831: (0)
832: (0)
833: (4)
834: (4)
835: (4)
836: (4)
837: (4)
838: (4)
839: (4)
840: (8)
841: (4)
842: (4)
843: (4)
844: (8)
845: (4)
846: (4)
847: (4)
848: (4)
849: (4)
850: (4)
851: (4)
852: (4)
853: (4)
854: (4)
855: (4)
856: (4)
857: (0)
858: (4)
859: (0)
860: (0)
861: (4)
862: (4)
863: (4)
864: (4)
865: (4)
866: (4)
867: (4)
868: (8)
869: (4)
870: (8)
871: (8)

@array_function_dispatch(_just_dispatcher)
def ljust(a, width, fillchar=' '):
    """
        Return an array with the elements of `a` left-justified in a
        string of length `width`.
        Calls `str.ljust` element-wise.
    Parameters
    -----
        a : array_like of str or unicode
        width : int
            The length of the resulting strings
        fillchar : str or unicode, optional
            The character to use for padding
    Returns
    -----
        out : ndarray
            Output array of str or unicode, depending on input type
    See Also
    -----
        str.ljust
    """
    a_arr = numpy.asarray(a)
    width_arr = numpy.asarray(width)
    size = int(numpy.max(width_arr.flat))
    if numpy.issubdtype(a_arr.dtype, numpy.bytes_):
        fillchar = asbytes(fillchar)
    return _vec_string(
        a_arr, type(a_arr.dtype)(size), 'ljust', (width_arr, fillchar))

@array_function_dispatch(_unary_op_dispatcher)
def lower(a):
    """
        Return an array with the elements converted to lowercase.
        Call `str.lower` element-wise.
        For 8-bit strings, this method is locale-dependent.
    Parameters
    -----
        a : array_like, {str, unicode}
            Input array.
    Returns
    -----
        out : ndarray, {str, unicode}
            Output array of str or unicode, depending on input type
    See Also
    -----
        str.lower
    Examples
    -----
        >>> c = np.array(['A1B C', '1BCA', 'BCA1']); c
        array(['A1B C', '1BCA', 'BCA1'], dtype='<U5')
        >>> np.char.lower(c)
        array(['a1b c', '1bca', 'bca1'], dtype='<U5')
    """
    a_arr = numpy.asarray(a)
    return _vec_string(a_arr, a_arr.dtype, 'lower')

def _strip_dispatcher(a, chars=None):
    return (a,)

@array_function_dispatch(_strip_dispatcher)
def lstrip(a, chars=None):
    """
        For each element in `a`, return a copy with the leading characters
        removed.
        Calls `str.lstrip` element-wise.
    Parameters
    -----
        a : array-like, {str, unicode}
            Input array.
        chars : {str, unicode}, optional
            The `chars` argument is a string specifying the set of
            characters to be removed. If omitted or None, the `chars`-

```

```

872: (8)             argument defaults to removing whitespace. The `chars` argument
873: (8)             is not a prefix; rather, all combinations of its values are
874: (8)             stripped.
875: (4)             Returns
876: (4)             -----
877: (4)             out : ndarray, {str, unicode}
878: (8)                 Output array of str or unicode, depending on input type
879: (4)             See Also
880: (4)             -----
881: (4)             str.lstrip
882: (4)             Examples
883: (4)             -----
884: (4)             >>> c = np.array(['aAaAaA', ' aA ', 'abBABba'])
885: (4)             >>> c
886: (4)             array(['aAaAaA', ' aA ', 'abBABba'], dtype='<U7')
887: (4)             The 'a' variable is unstripped from c[1] because whitespace leading.
888: (4)             >>> np.char.lstrip(c, 'a')
889: (4)             array(['AaAaA', ' aA ', 'bBABba'], dtype='<U7')
890: (4)             >>> np.char.lstrip(c, 'A') # leaves c unchanged
891: (4)             array(['aAaAaA', ' aA ', 'abBABba'], dtype='<U7')
892: (4)             >>> (np.char.lstrip(c, ' ') == np.char.lstrip(c, '')).all()
893: (4)             ... # XXX: is this a regression? This used to return True
894: (4)             ... # np.char.lstrip(c,'') does not modify c at all.
895: (4)             False
896: (4)             >>> (np.char.lstrip(c, ' ') == np.char.lstrip(c, None)).all()
897: (4)             True
898: (4)             """
899: (4)             a_arr = numpy.asarray(a)
900: (4)             return _vec_string(a_arr, a_arr.dtype, 'lstrip', (chars,))
def _partition_dispatcher(a, sep):
901: (0)             return (a,)
902: (4)
903: (0)             @array_function_dispatch(_partition_dispatcher)
904: (0)
905: (4)
906: (4)             def partition(a, sep):
907: (4)                 """
908: (4)                     Partition each element in `a` around `sep`.
909: (4)                     Calls `str.partition` element-wise.
910: (4)                     For each element in `a`, split the element as the first
911: (4)                     occurrence of `sep`, and return 3 strings containing the part
912: (4)                     before the separator, the separator itself, and the part after
913: (4)                     the separator. If the separator is not found, return 3 strings
914: (4)                     containing the string itself, followed by two empty strings.
915: (4)                     Parameters
916: (4)                     -----
917: (4)                     a : array_like, {str, unicode}
918: (4)                         Input array
919: (4)                     sep : {str, unicode}
920: (4)                         Separator to split each string element in `a`.
921: (4)                     Returns
922: (4)                     -----
923: (4)                     out : ndarray, {str, unicode}
924: (4)                         Output array of str or unicode, depending on input type.
925: (4)                         The output array will have an extra dimension with 3
926: (4)                         elements per input element.
927: (4)             See Also
928: (4)             -----
929: (4)             str.partition
930: (4)             """
931: (4)             def _replace_dispatcher(a, old, new, count=None):
932: (4)                 return (a,)
933: (4)             @array_function_dispatch(_replace_dispatcher)
934: (4)
935: (4)             def replace(a, old, new, count=None):
936: (4)                 """
937: (4)                     For each element in `a`, return a copy of the string with all
938: (4)                     occurrences of substring `old` replaced by `new`.
939: (4)                     Calls `str.replace` element-wise.
940: (4)                     Parameters
941: (4)                     -----

```

```

941: (4)             a : array-like of str or unicode
942: (4)             old, new : str or unicode
943: (4)             count : int, optional
944: (8)                 If the optional argument `count` is given, only the first
945: (8)                 `count` occurrences are replaced.
946: (4)             Returns
947: (4)             -----
948: (4)             out : ndarray
949: (8)                 Output array of str or unicode, depending on input type
950: (4)             See Also
951: (4)             -----
952: (4)             str.replace
953: (4)             Examples
954: (4)             -----
955: (4)             >>> a = np.array(["That is a mango", "Monkeys eat mangos"])
956: (4)             >>> np.char.replace(a, 'mango', 'banana')
957: (4)             array(['That is a banana', 'Monkeys eat bananas'], dtype='<U19')
958: (4)             >>> a = np.array(["The dish is fresh", "This is it"])
959: (4)             >>> np.char.replace(a, 'is', 'was')
960: (4)             array(['The dwash was fresh', 'Thwas was it'], dtype='<U19')
961: (4)             """
962: (4)             return _to_bytes_or_str_array(
963: (8)                 _vec_string(a, object_, 'replace', [old, new] + _clean_args(count)),
964: (0)             a)
965: (0)             @array_function_dispatch(_count_dispatcher)
966: (4)             def rfind(a, sub, start=0, end=None):
967: (4)                 """
968: (4)                     For each element in `a`, return the highest index in the string
969: (4)                     where substring `sub` is found, such that `sub` is contained
970: (4)                     within [`start`, `end`].
971: (4)                     Calls `str.rfind` element-wise.
972: (4)                     Parameters
973: (4)                     -----
974: (4)                     a : array-like of str or unicode
975: (4)                     sub : str or unicode
976: (4)                     start, end : int, optional
977: (8)                         Optional arguments `start` and `end` are interpreted as in
978: (4)                         slice notation.
979: (4)                     Returns
980: (4)                     -----
981: (7)                     out : ndarray
982: (4)                         Output array of ints. Return -1 on failure.
983: (4)                     See Also
984: (4)                     -----
985: (4)                     str.rfind
986: (4)                     """
987: (8)                     return _vec_string(
988: (0)                         a, int_, 'rfind', [sub, start] + _clean_args(end))
989: (0)             @array_function_dispatch(_count_dispatcher)
990: (4)             def rindex(a, sub, start=0, end=None):
991: (4)                 """
992: (4)                     Like `rfind`, but raises `ValueError` when the substring `sub` is
993: (4)                     not found.
994: (4)                     Calls `str.rindex` element-wise.
995: (4)                     Parameters
996: (4)                     -----
997: (4)                     a : array-like of str or unicode
998: (4)                     sub : str or unicode
999: (4)                     start, end : int, optional
1000: (4)                     Returns
1001: (4)                     -----
1002: (7)                     out : ndarray
1003: (4)                         Output array of ints.
1004: (4)                     See Also
1005: (4)                     -----
1006: (4)                     rfind, str.rindex
1007: (4)                     """
1008: (8)                     return _vec_string(
1009: (0)                         a, int_, 'rindex', [sub, start] + _clean_args(end))

```

```

1009: (0)
1010: (0)
1011: (4)
1012: (4)
1013: (4)
1014: (4)
1015: (4)
1016: (4)
1017: (4)
1018: (4)
1019: (8)
1020: (4)
1021: (8)
1022: (4)
1023: (4)
1024: (4)
1025: (8)
1026: (4)
1027: (4)
1028: (4)
1029: (4)
1030: (4)
1031: (4)
1032: (4)
1033: (4)
1034: (8)
1035: (4)
1036: (8)
1037: (0)
1038: (0)
1039: (4)
1040: (4)
1041: (4)
1042: (4)
1043: (4)
1044: (4)
1045: (4)
1046: (4)
1047: (4)
1048: (4)
1049: (4)
1050: (8)
1051: (4)
1052: (8)
1053: (4)
1054: (4)
1055: (4)
1056: (8)
1057: (8)
1058: (8)
1059: (4)
1060: (4)
1061: (4)
1062: (4)
1063: (4)
1064: (8)
1065: (0)
1066: (4)
1067: (0)
1068: (0)
1069: (4)
1070: (4)
1071: (4)
1072: (4)
1073: (4)
1074: (4)
1075: (4)
1076: (4)
1077: (4)

@array_function_dispatch(_just_dispatcher)
def rjust(a, width, fillchar=' '):
    """
        Return an array with the elements of `a` right-justified in a
        string of length `width`.
        Calls `str.rjust` element-wise.
    Parameters
    -----
        a : array_like of str or unicode
        width : int
            The length of the resulting strings
        fillchar : str or unicode, optional
            The character to use for padding
    Returns
    -----
        out : ndarray
            Output array of str or unicode, depending on input type
    See Also
    -----
        str.rjust
    """
    a_arr = numpy.asarray(a)
    width_arr = numpy.asarray(width)
    size = int(numpy.max(width_arr.flat))
    if numpy.issubdtype(a_arr.dtype, numpy.bytes_):
        fillchar = asbytes(fillchar)
    return _vec_string(
        a_arr, type(a_arr.dtype)(size), 'rjust', (width_arr, fillchar))

@array_function_dispatch(_partition_dispatcher)
def rpartition(a, sep):
    """
        Partition (split) each element around the right-most separator.
        Calls `str.rpartition` element-wise.
        For each element in `a`, split the element as the last
        occurrence of `sep`, and return 3 strings containing the part
        before the separator, the separator itself, and the part after
        the separator. If the separator is not found, return 3 strings
        containing the string itself, followed by two empty strings.
    Parameters
    -----
        a : array_like of str or unicode
            Input array
        sep : str or unicode
            Right-most separator to split each element in array.
    Returns
    -----
        out : ndarray
            Output array of string or unicode, depending on input
            type. The output array will have an extra dimension with
            3 elements per input element.
    See Also
    -----
        str.rpartition
    """
    return _to_bytes_or_str_array(
        _vec_string(a, object_, 'rpartition', (sep,)), a)

def _split_dispatcher(a, sep=None, maxsplit=None):
    return (a,)

@array_function_dispatch(_split_dispatcher)
def rsplit(a, sep=None, maxsplit=None):
    """
        For each element in `a`, return a list of the words in the
        string, using `sep` as the delimiter string.
        Calls `str.rsplit` element-wise.
        Except for splitting from the right, `rsplit`
        behaves like `split`.
    Parameters
    -----
        a : array_like of str or unicode
    """

```

```

1078: (4)           sep : str or unicode, optional
1079: (8)             If `sep` is not specified or None, any whitespace string
1080: (8)               is a separator.
1081: (4)           maxsplit : int, optional
1082: (8)             If `maxsplit` is given, at most `maxsplit` splits are done,
1083: (8)               the rightmost ones.
1084: (4)           Returns
1085: (4)           -----
1086: (4)           out : ndarray
1087: (7)             Array of list objects
1088: (4)           See Also
1089: (4)           -----
1090: (4)           str.rsplit, split
1091: (4)           """
1092: (4)           return _vec_string(
1093: (8)             a, object_, 'rsplit', [sep] + _clean_args(maxsplit))
1094: (0)           def _strip_dispatcher(a, chars=None):
1095: (4)             return (a,)
1096: (0)           @array_function_dispatch(_strip_dispatcher)
1097: (0)           def rstrip(a, chars=None):
1098: (4)             """
1099: (4)             For each element in `a`, return a copy with the trailing
1100: (4)               characters removed.
1101: (4)             Calls `str.rstrip` element-wise.
1102: (4)           Parameters
1103: (4)           -----
1104: (4)           a : array-like of str or unicode
1105: (4)           chars : str or unicode, optional
1106: (7)             The `chars` argument is a string specifying the set of
1107: (7)               characters to be removed. If omitted or None, the `chars`-
1108: (7)               argument defaults to removing whitespace. The `chars` argument
1109: (7)               is not a suffix; rather, all combinations of its values are
1110: (7)               stripped.
1111: (4)           Returns
1112: (4)           -----
1113: (4)           out : ndarray
1114: (8)             Output array of str or unicode, depending on input type
1115: (4)           See Also
1116: (4)           -----
1117: (4)           str.rstrip
1118: (4)           Examples
1119: (4)           -----
1120: (4)           >>> c = np.array(['aAaAaA', 'abBABba'], dtype='S7'); c
1121: (4)           array(['aAaAaA', 'abBABba'],
1122: (8)             dtype='|S7')
1123: (4)           >>> np.char.rstrip(c, b'a')
1124: (4)           array(['aAaAaA', 'abBABb'],
1125: (8)             dtype='|S7')
1126: (4)           >>> np.char.rstrip(c, b'A')
1127: (4)           array(['aAaAa', 'abBABba'],
1128: (8)             dtype='|S7')
1129: (4)           """
1130: (4)           a_arr = numpy.asarray(a)
1131: (4)           return _vec_string(a_arr, a_arr.dtype, 'rstrip', (chars,))
1132: (0)           @array_function_dispatch(_split_dispatcher)
1133: (0)           def split(a, sep=None, maxsplit=None):
1134: (4)             """
1135: (4)             For each element in `a`, return a list of the words in the
1136: (4)               string, using `sep` as the delimiter string.
1137: (4)             Calls `str.split` element-wise.
1138: (4)           Parameters
1139: (4)           -----
1140: (4)           a : array-like of str or unicode
1141: (4)           sep : str or unicode, optional
1142: (7)             If `sep` is not specified or None, any whitespace string is a
1143: (7)               separator.
1144: (4)           maxsplit : int, optional
1145: (8)             If `maxsplit` is given, at most `maxsplit` splits are done.
1146: (4)           Returns

```

```

1147: (4)
1148: (4)
1149: (8)
1150: (4)
1151: (4)
1152: (4)
1153: (4)
1154: (4)
1155: (8)
1156: (0)
1157: (4)
1158: (0)
1159: (0)
1160: (4)
1161: (4)
1162: (4)
1163: (4)
1164: (4)
1165: (4)
1166: (4)
1167: (4)
1168: (8)
1169: (8)
1170: (4)
1171: (4)
1172: (4)
1173: (8)
1174: (4)
1175: (4)
1176: (4)
1177: (4)
1178: (4)
1179: (8)
1180: (0)
1181: (4)
1182: (0)
1183: (0)
1184: (4)
1185: (4)
1186: (4)
1187: (4)
1188: (4)
1189: (4)
1190: (4)
1191: (4)
1192: (4)
1193: (8)
1194: (8)
1195: (4)
1196: (4)
1197: (4)
1198: (8)
1199: (4)
1200: (4)
1201: (4)
1202: (4)
1203: (4)
1204: (8)
1205: (0)
1206: (0)
1207: (4)
1208: (4)
1209: (4)
1210: (4)
1211: (4)
1212: (4)
1213: (4)
1214: (4)
1215: (7)

-----  

1148: (4)  

1149: (8)  

1150: (4)  

1151: (4)  

1152: (4)  

1153: (4)  

1154: (4)  

1155: (8)  

1156: (0)  

1157: (4)  

1158: (0)  

1159: (0)  

1160: (4)  

1161: (4)  

1162: (4)  

1163: (4)  

1164: (4)  

1165: (4)  

1166: (4)  

1167: (4)  

1168: (8)  

1169: (8)  

1170: (4)  

1171: (4)  

1172: (4)  

1173: (8)  

1174: (4)  

1175: (4)  

1176: (4)  

1177: (4)  

1178: (4)  

1179: (8)  

1180: (0)  

1181: (4)  

1182: (0)  

1183: (0)  

1184: (4)  

1185: (4)  

1186: (4)  

1187: (4)  

1188: (4)  

1189: (4)  

1190: (4)  

1191: (4)  

1192: (4)  

1193: (8)  

1194: (8)  

1195: (4)  

1196: (4)  

1197: (4)  

1198: (8)  

1199: (4)  

1200: (4)  

1201: (4)  

1202: (4)  

1203: (4)  

1204: (8)  

1205: (0)  

1206: (0)  

1207: (4)  

1208: (4)  

1209: (4)  

1210: (4)  

1211: (4)  

1212: (4)  

1213: (4)  

1214: (4)  

1215: (7)

-----  

out : ndarray  

    Array of list objects  

See Also  

-----  

str.split, rsplit  

"""  

    return _vec_string(  

        a, object_, 'split', [sep] + _clean_args(maxsplit))  

def _splitlines_dispatcher(a, keepends=None):  

    return (a,)  

@array_function_dispatch(_splitlines_dispatcher)  

def splitlines(a, keepends=None):  

    """  

        For each element in `a`, return a list of the lines in the  

        element, breaking at line boundaries.  

        Calls `str.splitlines` element-wise.  

Parameters  

-----  

a : array_like of str or unicode  

keepends : bool, optional  

    Line breaks are not included in the resulting list unless  

    keepends is given and true.  

Returns  

-----  

out : ndarray  

    Array of list objects  

See Also  

-----  

str.splitlines  

"""  

    return _vec_string(  

        a, object_, 'splitlines', _clean_args(keepends))  

def _startswith_dispatcher(a, prefix, start=None, end=None):  

    return (a,)  

@array_function_dispatch(_startswith_dispatcher)  

def startswith(a, prefix, start=0, end=None):  

    """  

        Returns a boolean array which is `True` where the string element  

        in `a` starts with `prefix`, otherwise `False`.  

        Calls `str.startswith` element-wise.  

Parameters  

-----  

a : array_like of str or unicode  

prefix : str  

start, end : int, optional  

    With optional `start`, test beginning at that position. With  

    optional `end`, stop comparing at that position.  

Returns  

-----  

out : ndarray  

    Array of booleans  

See Also  

-----  

str.startswith  

"""  

    return _vec_string(  

        a, bool_, 'startswith', [prefix, start] + _clean_args(end))  

@array_function_dispatch(_strip_dispatcher)  

def strip(a, chars=None):  

    """  

        For each element in `a`, return a copy with the leading and  

        trailing characters removed.  

        Calls `str.strip` element-wise.  

Parameters  

-----  

a : array-like of str or unicode  

chars : str or unicode, optional  

    The `chars` argument is a string specifying the set of

```

```

1216: (7)           characters to be removed. If omitted or None, the `chars` argument
1217: (7)           defaults to removing whitespace. The `chars` argument
1218: (7)           is not a prefix or suffix; rather, all combinations of its
1219: (7)           values are stripped.
1220: (4)           Returns
1221: (4)           -----
1222: (4)           out : ndarray
1223: (8)             Output array of str or unicode, depending on input type
1224: (4)           See Also
1225: (4)           -----
1226: (4)           str.strip
1227: (4)           Examples
1228: (4)           -----
1229: (4)             >>> c = np.array(['aAaAaA', ' aA ', 'abBABba'])
1230: (4)             >>> c
1231: (4)             array(['aAaAaA', ' aA ', 'abBABba'], dtype='<U7')
1232: (4)             >>> np.char.strip(c)
1233: (4)             array(['aAaAaA', 'aA', 'abBABba'], dtype='<U7')
1234: (4)             >>> np.char.strip(c, 'a') # 'a' unstripped from c[1] because whitespace
leads
1235: (4)             array(['AaAaA', ' aA ', 'bBABB'], dtype='<U7')
1236: (4)             >>> np.char.strip(c, 'A') # 'A' unstripped from c[1] because (unprinted)
ws trails
1237: (4)             array(['aAaAa', ' aA ', 'abBABba'], dtype='<U7')
1238: (4)             """
1239: (4)             a_arr = numpy.asarray(a)
1240: (4)             return _vec_string(a_arr, a_arr.dtype, 'strip', _clean_args(chars))
@array_function_dispatch(_unary_op_dispatcher)
1241: (0)           def swapcase(a):
1242: (0)             """
1243: (4)               Return element-wise a copy of the string with
1244: (4)               uppercase characters converted to lowercase and vice versa.
1245: (4)               Calls `str.swapcase` element-wise.
1246: (4)               For 8-bit strings, this method is locale-dependent.
1247: (4)               Parameters
1248: (4)               -----
1249: (4)               a : array_like, {str, unicode}
1250: (4)                 Input array.
1251: (8)           Returns
1252: (4)           -----
1253: (4)           out : ndarray, {str, unicode}
1254: (4)             Output array of str or unicode, depending on input type
1255: (8)           See Also
1256: (4)           -----
1257: (4)           str.swapcase
1258: (4)           Examples
1259: (4)           -----
1260: (4)             >>> c=np.array(['a1B c','1b Ca','b Ca1','cA1b'],'S5'); c
1261: (4)             array(['a1B c', '1b Ca', 'b Ca1', 'cA1b'],
1262: (4)               dtype='|S5')
1263: (8)             >>> np.char.swapcase(c)
1264: (4)             array(['A1b C', '1B cA', 'B ca1', 'Ca1B'],
1265: (4)               dtype='|S5')
1266: (8)             """
1267: (4)             a_arr = numpy.asarray(a)
1268: (4)             return _vec_string(a_arr, a_arr.dtype, 'swapcase')
@array_function_dispatch(_unary_op_dispatcher)
1269: (4)           def title(a):
1270: (0)             """
1271: (0)               Return element-wise title cased version of string or unicode.
1272: (4)               Title case words start with uppercase characters, all remaining cased
1273: (4)               characters are lowercase.
1274: (4)               Calls `str.title` element-wise.
1275: (4)               For 8-bit strings, this method is locale-dependent.
1276: (4)               Parameters
1277: (4)               -----
1278: (4)               a : array_like, {str, unicode}
1279: (4)                 Input array.
1280: (4)           Returns

```

```

1283: (4)
1284: (4)
1285: (8)
1286: (4)
1287: (4)
1288: (4)
1289: (4)
1290: (4)
1291: (4)
1292: (4)
1293: (8)
1294: (4)
1295: (4)
1296: (8)
1297: (4)
1298: (4)
1299: (4)
1300: (0)
1301: (4)
1302: (0)
1303: (0)
1304: (4)
1305: (4)
1306: (4)
1307: (4)
1308: (4)
1309: (4)
1310: (4)
1311: (4)
1312: (4)
1313: (4)
1314: (4)
1315: (4)
1316: (4)
1317: (4)
1318: (8)
1319: (4)
1320: (4)
1321: (4)
1322: (4)
1323: (4)
1324: (4)
1325: (8)
1326: (12)
1327: (4)
1328: (8)
1329: (12)
1330: (0)
1331: (0)
1332: (4)
1333: (4)
1334: (4)
1335: (4)
1336: (4)
1337: (4)
1338: (4)
1339: (8)
1340: (4)
1341: (4)
1342: (4)
1343: (8)
1344: (4)
1345: (4)
1346: (4)
1347: (4)
1348: (4)
1349: (4)
1350: (4)

-----  

1283: (4)
1284: (4)
1285: (8)
1286: (4)
1287: (4)
1288: (4)
1289: (4)
1290: (4)
1291: (4)
1292: (4)
1293: (8)
1294: (4)
1295: (4)
1296: (8)
1297: (4)
1298: (4)
1299: (4)
1300: (0)
1301: (4)
1302: (0)
1303: (0)
1304: (4)
1305: (4)
1306: (4)
1307: (4)
1308: (4)
1309: (4)
1310: (4)
1311: (4)
1312: (4)
1313: (4)
1314: (4)
1315: (4)
1316: (4)
1317: (4)
1318: (8)
1319: (4)
1320: (4)
1321: (4)
1322: (4)
1323: (4)
1324: (4)
1325: (8)
1326: (12)
1327: (4)
1328: (8)
1329: (12)
1330: (0)
1331: (0)
1332: (4)
1333: (4)
1334: (4)
1335: (4)
1336: (4)
1337: (4)
1338: (4)
1339: (8)
1340: (4)
1341: (4)
1342: (4)
1343: (8)
1344: (4)
1345: (4)
1346: (4)
1347: (4)
1348: (4)
1349: (4)
1350: (4)

-----  

out : ndarray
    Output array of str or unicode, depending on input type
See Also
-----
str.title
Examples
-----
>>> c=np.array(['a1b c','1b ca','b ca1','ca1b'],'S5'); c
array(['a1b c', '1b ca', 'b ca1', 'ca1b'],
      dtype='|S5')
>>> np.char.title(c)
array(['A1B C', '1B Ca', 'B Ca1', 'Ca1B'],
      dtype='|S5')
"""
a_arr = numpy.asarray(a)
return _vec_string(a_arr, a_arr.dtype, 'title')
def _translate_dispatcher(a, table, deletechars=None):
    return (a,)
@array_function_dispatch(_translate_dispatcher)
def translate(a, table, deletechars=None):
    """
    For each element in `a`, return a copy of the string where all
    characters occurring in the optional argument `deletechars` are
    removed, and the remaining characters have been mapped through the
    given translation table.
    Calls `str.translate` element-wise.
    Parameters
    -----
    a : array-like of str or unicode
    table : str of length 256
    deletechars : str
    Returns
    -----
    out : ndarray
        Output array of str or unicode, depending on input type
See Also
-----
str.translate
"""
a_arr = numpy.asarray(a)
if issubclass(a_arr.dtype.type, str_):
    return _vec_string(
        a_arr, a_arr.dtype, 'translate', (table,))
else:
    return _vec_string(
        a_arr, a_arr.dtype, 'translate', [table] +
_clean_args(deletechars))
@array_function_dispatch(_unary_op_dispatcher)
def upper(a):
    """
    Return an array with the elements converted to uppercase.
    Calls `str.upper` element-wise.
    For 8-bit strings, this method is locale-dependent.
    Parameters
    -----
    a : array_like, {str, unicode}
        Input array.
    Returns
    -----
    out : ndarray, {str, unicode}
        Output array of str or unicode, depending on input type
See Also
-----
str.upper
Examples
-----
>>> c = np.array(['a1b c', '1bca', 'bca1']); c
array(['a1b c', '1bca', 'bca1'], dtype='<U5')

```

```

1351: (4)             >>> np.char.upper(c)
1352: (4)             array(['A1B C', '1BCA', 'BCA1'], dtype='<U5')
1353: (4)             """
1354: (4)             a_arr = numpy.asarray(a)
1355: (4)             return _vec_string(a_arr, a_arr.dtype, 'upper')
1356: (0)             def _zfill_dispatcher(a, width):
1357: (4)                 return (a,)
1358: (0)             @array_function_dispatch(_zfill_dispatcher)
1359: (0)             def zfill(a, width):
1360: (4)                 """
1361: (4)                 Return the numeric string left-filled with zeros
1362: (4)                 Calls `str.zfill` element-wise.
1363: (4)                 Parameters
1364: (4)                 -----
1365: (4)                 a : array_like, {str, unicode}
1366: (8)                   Input array.
1367: (4)                 width : int
1368: (8)                   Width of string to left-fill elements in `a`.
1369: (4)                 Returns
1370: (4)                 -----
1371: (4)                 out : ndarray, {str, unicode}
1372: (8)                   Output array of str or unicode, depending on input type
1373: (4)                 See Also
1374: (4)                 -----
1375: (4)                 str.zfill
1376: (4)                 """
1377: (4)                 a_arr = numpy.asarray(a)
1378: (4)                 width_arr = numpy.asarray(width)
1379: (4)                 size = int(numpy.max(width_arr.flat))
1380: (4)                 return _vec_string(
1381: (8)                     a_arr, type(a_arr.dtype)(size), 'zfill', (width_arr,))
1382: (0)             @array_function_dispatch(_unary_op_dispatcher)
1383: (0)             def isnumeric(a):
1384: (4)                 """
1385: (4)                 For each element, return True if there are only numeric
1386: (4)                 characters in the element.
1387: (4)                 Calls `str.isnumeric` element-wise.
1388: (4)                 Numeric characters include digit characters, and all characters
1389: (4)                 that have the Unicode numeric value property, e.g. ``U+2155,
1390: (4)                 VULGAR FRACTION ONE FIFTH``.
1391: (4)                 Parameters
1392: (4)                 -----
1393: (4)                 a : array_like, unicode
1394: (8)                   Input array.
1395: (4)                 Returns
1396: (4)                 -----
1397: (4)                 out : ndarray, bool
1398: (8)                   Array of booleans of same shape as `a`.
1399: (4)                 See Also
1400: (4)                 -----
1401: (4)                 str.isnumeric
1402: (4)                 Examples
1403: (4)                 -----
1404: (4)                 >>> np.char.isnumeric(['123', '123abc', '9.0', '1/4', 'VIII'])
1405: (4)                 array([ True, False, False, False, False])
1406: (4)                 """
1407: (4)                 if not _is_unicode(a):
1408: (8)                     raise TypeError("isnumeric is only available for Unicode strings and
arrays")
1409: (4)                     return _vec_string(a, bool_, 'isnumeric')
1410: (0)             @array_function_dispatch(_unary_op_dispatcher)
1411: (0)             def isdecimal(a):
1412: (4)                 """
1413: (4)                 For each element, return True if there are only decimal
1414: (4)                 characters in the element.
1415: (4)                 Calls `str.isdecimal` element-wise.
1416: (4)                 Decimal characters include digit characters, and all characters
1417: (4)                 that can be used to form decimal-radix numbers,
e.g. ``U+0660, ARABIC-INDIC DIGIT ZERO``.

```

```

1419: (4)          Parameters
1420: (4)          -----
1421: (4)          a : array_like, unicode
1422: (8)          Input array.
1423: (4)          Returns
1424: (4)          -----
1425: (4)          out : ndarray, bool
1426: (8)          Array of booleans identical in shape to `a`.
1427: (4)          See Also
1428: (4)          -----
1429: (4)          str.isdecimal
1430: (4)          Examples
1431: (4)          -----
1432: (4)          >>> np.char.isdecimal(['12345', '4.99', '123ABC', ''])
1433: (4)          array([ True, False, False, False])
1434: (4)
1435: (4)          """
1436: (8)          if not _is_unicode(a):
1437: (12)          raise TypeError(
1438: (4)                  "isdecimal is only available for Unicode strings and arrays")
1439: (0)          return _vec_string(a, bool_, 'isdecimal')
1440: (0)          @set_module('numpy')
1441: (4)          class chararray(ndarray):
1442: (4)          """
1443: (14)          chararray(shape, itemsize=1, unicode=False, buffer=None, offset=0,
1444: (4)                      strides=None, order=None)
1445: (4)          Provides a convenient view on arrays of string and unicode values.
1446: (4)          .. note::
1447: (7)          The `chararray` class exists for backwards compatibility with
1448: (7)          Numarray, it is not recommended for new development. Starting from
1449: (7)          1.4, if one needs arrays of strings, it is recommended to use arrays of
1450: (7)          `dtype` `object_`, `bytes_` or `str_`, and use the free functions
1451: (4)          in the `numpy.char` module for fast vectorized string operations.
1452: (4)          Versus a regular NumPy array of type `str` or `unicode`, this
1453: (6)          class adds the following functionality:
1454: (9)          1) values automatically have whitespace removed from the end
1455: (6)          when indexed
1456: (9)          2) comparison operators automatically remove whitespace from the
1457: (6)          end when comparing values
1458: (9)          3) vectorized string operations are provided as methods
1459: (4)          (e.g. `endswith`) and infix operators (e.g. `"+", "*", "%"`)
1460: (4)          chararrays should be created using `numpy.char.array` or
1461: (4)          `numpy.char.asarray`, rather than this constructor directly.
1462: (4)          This constructor creates the array, using `buffer` (with `offset`
1463: (4)          and `strides`) if it is not ``None``. If `buffer` is ``None``, then
1464: (4)          constructs a new array with `strides` in "C order", unless both
1465: (4)          ``len(shape) >= 2`` and ``order='F'``, in which case `strides`
1466: (4)          is in "Fortran order".
1467: (4)          Methods
1468: (4)          -----
1469: (4)          astype
1470: (4)          argsort
1471: (4)          copy
1472: (4)          count
1473: (4)          decode
1474: (4)          dump
1475: (4)          dumps
1476: (4)          encode
1477: (4)          endswith
1478: (4)          expandtabs
1479: (4)          fill
1480: (4)          find
1481: (4)          flatten
1482: (4)          getfield
1483: (4)          index
1484: (4)          isalnum
1485: (4)          isalpha
1486: (4)          isdecimal
1487: (4)          isdigit

```

```
1487: (4)           islower
1488: (4)           isnumeric
1489: (4)           isspace
1490: (4)           istitle
1491: (4)           isupper
1492: (4)           item
1493: (4)           join
1494: (4)           ljust
1495: (4)           lower
1496: (4)           lstrip
1497: (4)           nonzero
1498: (4)           put
1499: (4)           ravel
1500: (4)           repeat
1501: (4)           replace
1502: (4)           reshape
1503: (4)           resize
1504: (4)           rfind
1505: (4)           rindex
1506: (4)           rjust
1507: (4)           rsplit
1508: (4)           rstrip
1509: (4)           searchsorted
1510: (4)           setfield
1511: (4)           setflags
1512: (4)           sort
1513: (4)           split
1514: (4)           splitlines
1515: (4)           squeeze
1516: (4)           startswith
1517: (4)           strip
1518: (4)           swapaxes
1519: (4)           swapcase
1520: (4)           take
1521: (4)           title
1522: (4)           tofile
1523: (4)           tolist
1524: (4)           tostring
1525: (4)           translate
1526: (4)           transpose
1527: (4)           upper
1528: (4)           view
1529: (4)           zfill
1530: (4)           Parameters
1531: (4)           -----
1532: (4)           shape : tuple
1533: (8)             Shape of the array.
1534: (4)           itemsize : int, optional
1535: (8)             Length of each array element, in number of characters. Default is 1.
1536: (4)           unicode : bool, optional
1537: (8)             Are the array elements of type unicode (True) or string (False).
1538: (8)             Default is False.
1539: (4)           buffer : object exposing the buffer interface or str, optional
1540: (8)             Memory address of the start of the array data. Default is None,
1541: (8)               in which case a new array is created.
1542: (4)           offset : int, optional
1543: (8)             Fixed stride displacement from the beginning of an axis?
1544: (8)             Default is 0. Needs to be >=0.
1545: (4)           strides : array_like of ints, optional
1546: (8)             Strides for the array (see `ndarray.strides` for full description).
1547: (8)             Default is None.
1548: (4)           order : {'C', 'F'}, optional
1549: (8)             The order in which the array data is stored in memory: 'C' ->
1550: (8)               "row major" order (the default), 'F' -> "column major"
1551: (8)               (Fortran) order.
1552: (4)           Examples
1553: (4)           -----
1554: (4)           >>> charar = np.chararray((3, 3))
1555: (4)           >>> charar[:] = 'a'
```

```

1556: (4)          >>> charar
1557: (4)          chararray([[b'a', b'a', b'a'],
1558: (15)            [b'a', b'a', b'a'],
1559: (15)            [b'a', b'a', b'a']], dtype='|S1')
1560: (4)          >>> charar = np.chararray(charar.shape, itemsize=5)
1561: (4)          >>> charar[:] = 'abc'
1562: (4)          >>> charar
1563: (4)          chararray([[b'abc', b'abc', b'abc'],
1564: (15)            [b'abc', b'abc', b'abc'],
1565: (15)            [b'abc', b'abc', b'abc']], dtype='|S5')
1566: (4)          """
1567: (4)          def __new__(subtype, shape, itemsize=1, unicode=False, buffer=None,
1568: (16)            offset=0, strides=None, order='C'):
1569: (8)            global _globalvar
1570: (8)            if unicode:
1571: (12)              dtype = str_
1572: (8)            else:
1573: (12)              dtype = bytes_
1574: (8)            itemsize = int(itemsize)
1575: (8)            if isinstance(buffer, str):
1576: (12)              filler = buffer
1577: (12)              buffer = None
1578: (8)            else:
1579: (12)              filler = None
1580: (8)            _globalvar = 1
1581: (8)            if buffer is None:
1582: (12)              self = ndarray.__new__(subtype, shape, (dtype, itemsize),
1583: (35)                order=order)
1584: (8)            else:
1585: (12)              self = ndarray.__new__(subtype, shape, (dtype, itemsize),
1586: (35)                buffer=buffer,
1587: (35)                offset=offset, strides=strides,
1588: (35)                order=order)
1589: (8)            if filler is not None:
1590: (12)              self[...] = filler
1591: (8)            _globalvar = 0
1592: (8)            return self
1593: (4)          def __array_finalize__(self, obj):
1594: (8)            if not _globalvar and self.dtype.char not in 'SUbc':
1595: (12)              raise ValueError("Can only create a chararray from string data.")
1596: (4)          def __getitem__(self, obj):
1597: (8)            val = ndarray.__getitem__(self, obj)
1598: (8)            if isinstance(val, character):
1599: (12)              temp = val.rstrip()
1600: (12)              if len(temp) == 0:
1601: (16)                val = ''
1602: (12)              else:
1603: (16)                val = temp
1604: (8)            return val
1605: (4)          def __eq__(self, other):
1606: (8)          """
1607: (8)            Return (self == other) element-wise.
1608: (8)            See Also
1609: (8)            -----
1610: (8)            equal
1611: (8)            """
1612: (8)            return equal(self, other)
1613: (4)          def __ne__(self, other):
1614: (8)          """
1615: (8)            Return (self != other) element-wise.
1616: (8)            See Also
1617: (8)            -----
1618: (8)            not_equal
1619: (8)            """
1620: (8)            return not_equal(self, other)
1621: (4)          def __ge__(self, other):
1622: (8)          """
1623: (8)            Return (self >= other) element-wise.
1624: (8)            See Also

```

```
1625: (8)          -----
1626: (8)          greater_equal
1627: (8)          """
1628: (8)          return greater_equal(self, other)
1629: (4)          def __le__(self, other):
1630: (8)          """
1631: (8)          Return (self <= other) element-wise.
1632: (8)          See Also
1633: (8)          -----
1634: (8)          less_equal
1635: (8)          """
1636: (8)          return less_equal(self, other)
1637: (4)          def __gt__(self, other):
1638: (8)          """
1639: (8)          Return (self > other) element-wise.
1640: (8)          See Also
1641: (8)          -----
1642: (8)          greater
1643: (8)          """
1644: (8)          return greater(self, other)
1645: (4)          def __lt__(self, other):
1646: (8)          """
1647: (8)          Return (self < other) element-wise.
1648: (8)          See Also
1649: (8)          -----
1650: (8)          less
1651: (8)          """
1652: (8)          return less(self, other)
1653: (4)          def __add__(self, other):
1654: (8)          """
1655: (8)          Return (self + other), that is string concatenation,
1656: (8)          element-wise for a pair of array_likes of str or unicode.
1657: (8)          See Also
1658: (8)          -----
1659: (8)          add
1660: (8)          """
1661: (8)          return asarray(add(self, other))
1662: (4)          def __radd__(self, other):
1663: (8)          """
1664: (8)          Return (other + self), that is string concatenation,
1665: (8)          element-wise for a pair of array_likes of `bytes_` or `str_`.
1666: (8)          See Also
1667: (8)          -----
1668: (8)          add
1669: (8)          """
1670: (8)          return asarray(add(numpy.asarray(other), self))
1671: (4)          def __mul__(self, i):
1672: (8)          """
1673: (8)          Return (self * i), that is string multiple concatenation,
1674: (8)          element-wise.
1675: (8)          See Also
1676: (8)          -----
1677: (8)          multiply
1678: (8)          """
1679: (8)          return asarray(multiply(self, i))
1680: (4)          def __rmul__(self, i):
1681: (8)          """
1682: (8)          Return (self * i), that is string multiple concatenation,
1683: (8)          element-wise.
1684: (8)          See Also
1685: (8)          -----
1686: (8)          multiply
1687: (8)          """
1688: (8)          return asarray(multiply(self, i))
1689: (4)          def __mod__(self, i):
1690: (8)          """
1691: (8)          Return (self % i), that is pre-Python 2.6 string formatting
1692: (8)          (interpolation), element-wise for a pair of array_likes of `bytes_`
1693: (8)          or `str_`.
```

```

1694: (8)             See Also
1695: (8)             -----
1696: (8)             mod
1697: (8)             """
1698: (8)             return asarray(mod(self, i))
1699: (4)             def __rmod__(self, other):
1700: (8)                 return NotImplemented
1701: (4)             def argsort(self, axis=-1, kind=None, order=None):
1702: (8)                 """
1703: (8)                 Return the indices that sort the array lexicographically.
1704: (8)                 For full documentation see `numpy.argsort`, for which this method is
1705: (8)                 in fact merely a "thin wrapper."
1706: (8)             Examples
1707: (8)             -----
1708: (8)             >>> c = np.array(['a1b c', '1b ca', 'b ca1', 'Ca1b'], '|S5')
1709: (8)             >>> c = c.view(np.chararray); c
1710: (8)                 chararray(['a1b c', '1b ca', 'b ca1', 'Ca1b'],
1711: (14)                   dtype='|S5')
1712: (8)             >>> c[c.argsort()]
1713: (8)                 chararray(['1b ca', 'Ca1b', 'a1b c', 'b ca1'],
1714: (14)                   dtype='|S5')
1715: (8)                 """
1716: (8)             return self.__array__().argsort(axis, kind, order)
1717: (4)             argsort.__doc__ = ndarray.argsort.__doc__
1718: (4)             def capitalize(self):
1719: (8)                 """
1720: (8)                 Return a copy of `self` with only the first character of each element
1721: (8)                 capitalized.
1722: (8)             See Also
1723: (8)             -----
1724: (8)             char.capitalize
1725: (8)             """
1726: (8)             return asarray(capitalize(self))
1727: (4)             def center(self, width, fillchar=' '):
1728: (8)                 """
1729: (8)                 Return a copy of `self` with its elements centered in a
1730: (8)                 string of length `width`.
1731: (8)             See Also
1732: (8)             -----
1733: (8)             center
1734: (8)             """
1735: (8)             return asarray(center(self, width, fillchar))
1736: (4)             def count(self, sub, start=0, end=None):
1737: (8)                 """
1738: (8)                 Returns an array with the number of non-overlapping occurrences of
1739: (8)                 substring `sub` in the range [`start`, `end`].
1740: (8)             See Also
1741: (8)             -----
1742: (8)             char.count
1743: (8)             """
1744: (8)             return count(self, sub, start, end)
1745: (4)             def decode(self, encoding=None, errors=None):
1746: (8)                 """
1747: (8)                 Calls ``bytes.decode`` element-wise.
1748: (8)             See Also
1749: (8)             -----
1750: (8)             char.decode
1751: (8)             """
1752: (8)             return decode(self, encoding, errors)
1753: (4)             def encode(self, encoding=None, errors=None):
1754: (8)                 """
1755: (8)                 Calls `str.encode` element-wise.
1756: (8)             See Also
1757: (8)             -----
1758: (8)             char.encode
1759: (8)             """
1760: (8)             return encode(self, encoding, errors)
1761: (4)             def endswith(self, suffix, start=0, end=None):
1762: (8)                 """

```

```

1763: (8)           Returns a boolean array which is `True` where the string element
1764: (8)           in `self` ends with `suffix`, otherwise `False`.
1765: (8)           See Also
1766: (8)           -----
1767: (8)           char.endswith
1768: (8)           """
1769: (8)           return endswith(self, suffix, start, end)
1770: (4) def expandtabs(self, tabsize=8):
1771: (8)           """
1772: (8)           Return a copy of each string element where all tab characters are
1773: (8)           replaced by one or more spaces.
1774: (8)           See Also
1775: (8)           -----
1776: (8)           char.expandtabs
1777: (8)           """
1778: (8)           return asarray(expandtabs(self, tabsize))
1779: (4) def find(self, sub, start=0, end=None):
1780: (8)           """
1781: (8)           For each element, return the lowest index in the string where
1782: (8)           substring `sub` is found.
1783: (8)           See Also
1784: (8)           -----
1785: (8)           char.find
1786: (8)           """
1787: (8)           return find(self, sub, start, end)
1788: (4) def index(self, sub, start=0, end=None):
1789: (8)           """
1790: (8)           Like `find`, but raises `ValueError` when the substring is not found.
1791: (8)           See Also
1792: (8)           -----
1793: (8)           char.index
1794: (8)           """
1795: (8)           return index(self, sub, start, end)
1796: (4) def isalnum(self):
1797: (8)           """
1798: (8)           Returns true for each element if all characters in the string
1799: (8)           are alphanumeric and there is at least one character, false
1800: (8)           otherwise.
1801: (8)           See Also
1802: (8)           -----
1803: (8)           char.isalnum
1804: (8)           """
1805: (8)           return isalnum(self)
1806: (4) def isalpha(self):
1807: (8)           """
1808: (8)           Returns true for each element if all characters in the string
1809: (8)           are alphabetic and there is at least one character, false
1810: (8)           otherwise.
1811: (8)           See Also
1812: (8)           -----
1813: (8)           char.isalpha
1814: (8)           """
1815: (8)           return isalpha(self)
1816: (4) def isdigit(self):
1817: (8)           """
1818: (8)           Returns true for each element if all characters in the string are
1819: (8)           digits and there is at least one character, false otherwise.
1820: (8)           See Also
1821: (8)           -----
1822: (8)           char.isdigit
1823: (8)           """
1824: (8)           return isdigit(self)
1825: (4) def islower(self):
1826: (8)           """
1827: (8)           Returns true for each element if all cased characters in the
1828: (8)           string are lowercase and there is at least one cased character,
1829: (8)           false otherwise.
1830: (8)           See Also
1831: (8)           -----

```

```
1832: (8)             char.islower
1833: (8)             """
1834: (8)             return islower(self)
1835: (4)              def isspace(self):
1836: (8)              """
1837: (8)              Returns true for each element if there are only whitespace
1838: (8)              characters in the string and there is at least one character,
1839: (8)              false otherwise.
1840: (8)              See Also
1841: (8)              -----
1842: (8)              charisspace
1843: (8)              """
1844: (8)              return isspace(self)
1845: (4)              def istitle(self):
1846: (8)              """
1847: (8)              Returns true for each element if the element is a titlecased
1848: (8)              string and there is at least one character, false otherwise.
1849: (8)              See Also
1850: (8)              -----
1851: (8)              char.istitle
1852: (8)              """
1853: (8)              return istitle(self)
1854: (4)              def isupper(self):
1855: (8)              """
1856: (8)              Returns true for each element if all cased characters in the
1857: (8)              string are uppercase and there is at least one character, false
1858: (8)              otherwise.
1859: (8)              See Also
1860: (8)              -----
1861: (8)              char.isupper
1862: (8)              """
1863: (8)              return isupper(self)
1864: (4)              def join(self, seq):
1865: (8)              """
1866: (8)              Return a string which is the concatenation of the strings in the
1867: (8)              sequence `seq`.
1868: (8)              See Also
1869: (8)              -----
1870: (8)              char.join
1871: (8)              """
1872: (8)              return join(self, seq)
1873: (4)              def ljust(self, width, fillchar=' '):
1874: (8)              """
1875: (8)              Return an array with the elements of `self` left-justified in a
1876: (8)              string of length `width`.
1877: (8)              See Also
1878: (8)              -----
1879: (8)              char.ljust
1880: (8)              """
1881: (8)              return asarray(ljust(self, width, fillchar))
1882: (4)              def lower(self):
1883: (8)              """
1884: (8)              Return an array with the elements of `self` converted to
1885: (8)              lowercase.
1886: (8)              See Also
1887: (8)              -----
1888: (8)              char.lower
1889: (8)              """
1890: (8)              return asarray(lower(self))
1891: (4)              def lstrip(self, chars=None):
1892: (8)              """
1893: (8)              For each element in `self`, return a copy with the leading characters
1894: (8)              removed.
1895: (8)              See Also
1896: (8)              -----
1897: (8)              char.lstrip
1898: (8)              """
1899: (8)              return asarray(lstrip(self, chars))
1900: (4)              def partition(self, sep):
```

```
1901: (8)          """
1902: (8)          Partition each element in `self` around `sep`.
1903: (8)          See Also
1904: (8)          -----
1905: (8)          partition
1906: (8)          """
1907: (8)          return asarray(partition(self, sep))
1908: (4)          def replace(self, old, new, count=None):
1909: (8)          """
1910: (8)          For each element in `self`, return a copy of the string with all
1911: (8)          occurrences of substring `old` replaced by `new`.
1912: (8)          See Also
1913: (8)          -----
1914: (8)          char.replace
1915: (8)          """
1916: (8)          return asarray(replace(self, old, new, count))
1917: (4)          def rfind(self, sub, start=0, end=None):
1918: (8)          """
1919: (8)          For each element in `self`, return the highest index in the string
1920: (8)          where substring `sub` is found, such that `sub` is contained
1921: (8)          within [`start`, `end`].
1922: (8)          See Also
1923: (8)          -----
1924: (8)          char.rfind
1925: (8)          """
1926: (8)          return rfind(self, sub, start, end)
1927: (4)          def rindex(self, sub, start=0, end=None):
1928: (8)          """
1929: (8)          Like `rfind`, but raises `ValueError` when the substring `sub` is
1930: (8)          not found.
1931: (8)          See Also
1932: (8)          -----
1933: (8)          char.rindex
1934: (8)          """
1935: (8)          return rindex(self, sub, start, end)
1936: (4)          def rjust(self, width, fillchar=' '):
1937: (8)          """
1938: (8)          Return an array with the elements of `self`
1939: (8)          right-justified in a string of length `width`.
1940: (8)          See Also
1941: (8)          -----
1942: (8)          char.rjust
1943: (8)          """
1944: (8)          return asarray(rjust(self, width, fillchar))
1945: (4)          def rpartition(self, sep):
1946: (8)          """
1947: (8)          Partition each element in `self` around `sep`.
1948: (8)          See Also
1949: (8)          -----
1950: (8)          rpartition
1951: (8)          """
1952: (8)          return asarray(rpartition(self, sep))
1953: (4)          def rsplit(self, sep=None, maxsplit=None):
1954: (8)          """
1955: (8)          For each element in `self`, return a list of the words in
1956: (8)          the string, using `sep` as the delimiter string.
1957: (8)          See Also
1958: (8)          -----
1959: (8)          char.rsplit
1960: (8)          """
1961: (8)          return rsplit(self, sep, maxsplit)
1962: (4)          def rstrip(self, chars=None):
1963: (8)          """
1964: (8)          For each element in `self`, return a copy with the trailing
1965: (8)          characters removed.
1966: (8)          See Also
1967: (8)          -----
1968: (8)          char.rstrip
1969: (8)          """
```

```

1970: (8)           return asarray(rstrip(self, chars))
1971: (4)           def split(self, sep=None, maxsplit=None):
1972: (8)             """
1973: (8)               For each element in `self`, return a list of the words in the
1974: (8)               string, using `sep` as the delimiter string.
1975: (8)               See Also
1976: (8)                 -----
1977: (8)                 char.split
1978: (8)                 """
1979: (8)               return split(self, sep, maxsplit)
1980: (4)           def splitlines(self, keepends=None):
1981: (8)             """
1982: (8)               For each element in `self`, return a list of the lines in the
1983: (8)               element, breaking at line boundaries.
1984: (8)               See Also
1985: (8)                 -----
1986: (8)                 char.splitlines
1987: (8)                 """
1988: (8)               return splitlines(self, keepends)
1989: (4)           def startswith(self, prefix, start=0, end=None):
1990: (8)             """
1991: (8)               Returns a boolean array which is `True` where the string element
1992: (8)               in `self` starts with `prefix`, otherwise `False`.
1993: (8)               See Also
1994: (8)                 -----
1995: (8)                 char.startswith
1996: (8)                 """
1997: (8)               return startswith(self, prefix, start, end)
1998: (4)           def strip(self, chars=None):
1999: (8)             """
2000: (8)               For each element in `self`, return a copy with the leading and
2001: (8)               trailing characters removed.
2002: (8)               See Also
2003: (8)                 -----
2004: (8)                 char.strip
2005: (8)                 """
2006: (8)               return asarray(strip(self, chars))
2007: (4)           def swapcase(self):
2008: (8)             """
2009: (8)               For each element in `self`, return a copy of the string with
2010: (8)               uppercase characters converted to lowercase and vice versa.
2011: (8)               See Also
2012: (8)                 -----
2013: (8)                 char.swapcase
2014: (8)                 """
2015: (8)               return asarray(swapcase(self))
2016: (4)           def title(self):
2017: (8)             """
2018: (8)               For each element in `self`, return a titlecased version of the
2019: (8)               string: words start with uppercase characters, all remaining cased
2020: (8)               characters are lowercase.
2021: (8)               See Also
2022: (8)                 -----
2023: (8)                 char.title
2024: (8)                 """
2025: (8)               return asarray(title(self))
2026: (4)           def translate(self, table, deletechars=None):
2027: (8)             """
2028: (8)               For each element in `self`, return a copy of the string where
2029: (8)               all characters occurring in the optional argument
2030: (8)               `deletechars` are removed, and the remaining characters have
2031: (8)               been mapped through the given translation table.
2032: (8)               See Also
2033: (8)                 -----
2034: (8)                 char.translate
2035: (8)                 """
2036: (8)               return asarray(translate(self, table, deletechars))
2037: (4)           def upper(self):
2038: (8)             """

```

```

2039: (8)             Return an array with the elements of `self` converted to
2040: (8)             uppercase.
2041: (8)             See Also
2042: (8)
2043: (8)             -----
2044: (8)             char.upper
2045: (8)             """
2046: (4)             return asarray(upper(self))
def zfill(self, width):
    """
2048: (8)             Return the numeric string left-filled with zeros in a string of
2049: (8)             length `width`.
2050: (8)             See Also
2051: (8)
2052: (8)             -----
2053: (8)             char.zfill
2054: (8)             """
2055: (4)             return asarray(zfill(self, width))
def isnumeric(self):
    """
2057: (8)             For each element in `self`, return True if there are only
2058: (8)             numeric characters in the element.
2059: (8)             See Also
2060: (8)
2061: (8)             -----
2062: (8)             char.isnumeric
2063: (8)             """
2064: (4)             return isnumeric(self)
def isdecimal(self):
    """
2066: (8)             For each element in `self`, return True if there are only
2067: (8)             decimal characters in the element.
2068: (8)             See Also
2069: (8)
2070: (8)             -----
2071: (8)             char.isdecimal
2072: (8)             """
2073: (0)             @set_module("numpy.char")
2074: (0)             def array(obj, itemsize=None, copy=True, unicode=None, order=None):
    """
2076: (4)             Create a `chararray`.
2077: (4)             .. note::
2078: (7)                 This class is provided for numarray backward-compatibility.
2079: (7)                 New code (not concerned with numarray compatibility) should use
2080: (7)                 arrays of type `bytes_` or `str_` and use the free functions
2081: (7)                 in :mod:`numpy.char <numpy.core.defchararray>` for fast
2082: (7)                 vectorized string operations instead.
2083: (4)                 Versus a regular NumPy array of type `str` or `unicode`, this
2084: (4)                 class adds the following functionality:
2085: (6)                     1) values automatically have whitespace removed from the end
2086: (9)                         when indexed
2087: (6)                     2) comparison operators automatically remove whitespace from the
2088: (9)                         end when comparing values
2089: (6)                     3) vectorized string operations are provided as methods
2090: (9)                         (e.g. `str.endswith`) and infix operators (e.g. ``+``, `*`, `%``)
2091: (4)             Parameters
2092: (4)
2093: (4)             -----
2094: (4)             obj : array of str or unicode-like
itemsize : int, optional
2095: (8)                 `itemsize` is the number of characters per scalar in the
2096: (8)                 resulting array. If `itemsize` is None, and `obj` is an
2097: (8)                 object array or a Python list, the `itemsize` will be
2098: (8)                 automatically determined. If `itemsize` is provided and `obj`
2099: (8)                 is of type str or unicode, then the `obj` string will be
2100: (8)                 chunked into `itemsize` pieces.
2101: (4)             copy : bool, optional
2102: (8)                 If true (default), then the object is copied. Otherwise, a copy
2103: (8)                 will only be made if __array__ returns a copy, if obj is a
2104: (8)                 nested sequence, or if a copy is needed to satisfy any of the other
2105: (8)                 requirements (`itemsize`, unicode, `order`, etc.).
2106: (4)             unicode : bool, optional
2107: (8)                 When true, the resulting `chararray` can contain Unicode

```

```

2108: (8)           characters, when false only 8-bit characters. If unicode is
2109: (8)           None and `obj` is one of the following:
2110: (10)          - a `chararray`,
2111: (10)          - an ndarray of type `str` or `unicode`
2112: (10)          - a Python str or unicode object,
2113: (8)          then the unicode setting of the output array will be
2114: (8)          automatically determined.
2115: (4)          order : {'C', 'F', 'A'}, optional
2116: (8)          Specify the order of the array. If order is 'C' (default), then the
2117: (8)          array will be in C-contiguous order (last-index varies the
2118: (8)          fastest). If order is 'F', then the returned array
2119: (8)          will be in Fortran-contiguous order (first-index varies the
2120: (8)          fastest). If order is 'A', then the returned array may
2121: (8)          be in any order (either C-, Fortran-contiguous, or even
2122: (8)          discontiguous).
2123: (4)
2124: (4)      """
2125: (8)      if isinstance(obj, (bytes, str)):
2126: (12)          if unicode is None:
2127: (16)              if isinstance(obj, str):
2128: (12)                  unicode = True
2129: (16)              else:
2130: (8)                  unicode = False
2131: (12)          if itemsize is None:
2132: (8)              itemsize = len(obj)
2133: (8)          shape = len(obj) // itemsize
2134: (25)          return chararray(shape, itemsize=itemsize, unicode=unicode,
2135: (4)                      buffer=obj, order=order)
2136: (8)      if isinstance(obj, (list, tuple)):
2137: (4)          obj = numpy.asarray(obj)
2138: (8)      if isinstance(obj, ndarray) and issubclass(obj.dtype.type, character):
2139: (12)          if not isinstance(obj, chararray):
2140: (8)              obj = obj.view(chararray)
2141: (12)          if itemsize is None:
2142: (12)              itemsize = obj.itemsize
2143: (16)              if issubclass(obj.dtype.type, str_):
2144: (8)                  itemsize //= 4
2145: (12)          if unicode is None:
2146: (16)              if issubclass(obj.dtype.type, str_):
2147: (12)                  unicode = True
2148: (16)              else:
2149: (8)                  unicode = False
2150: (12)          if unicode:
2151: (8)              dtype = str_
2152: (12)          else:
2153: (8)              dtype = bytes_
2154: (12)          if order is not None:
2155: (8)              obj = numpy.asarray(obj, order=order)
2156: (16)          if (copy or
2157: (16)              (itemsize != obj.itemsize) or
2158: (16)              (not unicode and isinstance(obj, str_)) or
2159: (12)              (unicode and isinstance(obj, bytes_))):
2160: (8)              obj = obj.astype((dtype, int(itemsize)))
2161: (4)          return obj
2162: (8)      if isinstance(obj, ndarray) and issubclass(obj.dtype.type, object):
2163: (12)          if itemsize is None:
2164: (8)              obj = obj.tolist()
2165: (4)          if unicode:
2166: (8)              dtype = str_
2167: (4)          else:
2168: (8)              dtype = bytes_
2169: (8)          if itemsize is None:
2170: (4)              val = narray(obj, dtype=dtype, order=order, subok=True)
2171: (8)          else:
2172: (4)              val = narray(obj, dtype=(dtype, itemsize), order=order, subok=True)
2173: (0)          return val.view(chararray)
2174: (0)      @set_module("numpy.char")
2175: (4)      def asarray(obj, itemsize=None, unicode=None, order=None):
2176: (4)          """
2177: (4)          Convert the input to a `chararray`, copying the data only if

```

```

2177: (4)             necessary.
2178: (4)             Versus a regular NumPy array of type `str` or `unicode`, this
2179: (4)             class adds the following functionality:
2180: (6)               1) values automatically have whitespace removed from the end
2181: (9)                 when indexed
2182: (6)               2) comparison operators automatically remove whitespace from the
2183: (9)                 end when comparing values
2184: (6)               3) vectorized string operations are provided as methods
2185: (9)                 (e.g. `str.endswith`) and infix operators (e.g. ``+``, ``*``, ``%``)
2186: (4)             Parameters
2187: (4)             -----
2188: (4)             obj : array of str or unicode-like
2189: (4)             itemsize : int, optional
2190: (8)               `itemsize` is the number of characters per scalar in the
2191: (8)               resulting array. If `itemsize` is None, and `obj` is an
2192: (8)               object array or a Python list, the `itemsize` will be
2193: (8)               automatically determined. If `itemsize` is provided and `obj`
2194: (8)               is of type str or unicode, then the `obj` string will be
2195: (8)               chunked into `itemsize` pieces.
2196: (4)             unicode : bool, optional
2197: (8)               When true, the resulting `chararray` can contain Unicode
2198: (8)               characters, when false only 8-bit characters. If unicode is
2199: (8)               None and `obj` is one of the following:
2200: (10)              - a `chararray`,
2201: (10)              - an ndarray of type `str` or `unicode`
2202: (10)              - a Python str or unicode object,
2203: (8)               then the unicode setting of the output array will be
2204: (8)               automatically determined.
2205: (4)             order : {'C', 'F'}, optional
2206: (8)               Specify the order of the array. If order is 'C' (default), then the
2207: (8)               array will be in C-contiguous order (last-index varies the
2208: (8)               fastest). If order is 'F', then the returned array
2209: (8)               will be in Fortran-contiguous order (first-index varies the
2210: (8)               fastest).
2211: (4)             """
2212: (4)             return array(obj, itemsize, copy=False,
2213: (17)                           unicode=unicode, order=order)

```

---

File 47 - einsumfunc.py:

```

1: (0)             """
2: (0)             Implementation of optimized einsum.
3: (0)             """
4: (0)             import itertools
5: (0)             import operator
6: (0)             from numpy.core.multiarray import c_einsum
7: (0)             from numpy.core.numeric import asanyarray, tensordot
8: (0)             from numpy.core.overrides import array_function_dispatch
9: (0)             __all__ = ['einsum', 'einsum_path']
10: (0)            einsum_symbols = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
11: (0)            einsum_symbols_set = set(einsum_symbols)
12: (0)            def _flop_count(idx_contraction, inner, num_terms, size_dictionary):
13: (4)             """
14: (4)               Computes the number of FLOPS in the contraction.
15: (4)             Parameters
16: (4)             -----
17: (4)             idx_contraction : iterable
18: (8)               The indices involved in the contraction
19: (4)             inner : bool
20: (8)               Does this contraction require an inner product?
21: (4)             num_terms : int
22: (8)               The number of terms in a contraction
23: (4)             size_dictionary : dict
24: (8)               The size of each of the indices in idx_contraction
25: (4)             Returns
26: (4)             -----
27: (4)             flop_count : int

```

```

28: (8)          The total number of FLOPS required for the contraction.
29: (4)          Examples
30: (4)          -----
31: (4)          >>> _flop_count('abc', False, 1, {'a': 2, 'b':3, 'c':5})
32: (4)          30
33: (4)          >>> _flop_count('abc', True, 2, {'a': 2, 'b':3, 'c':5})
34: (4)          60
35: (4)          """
36: (4)          overall_size = _compute_size_by_dict(idx_contraction, size_dictionary)
37: (4)          op_factor = max(1, num_terms - 1)
38: (4)          if inner:
39: (8)              op_factor += 1
40: (4)          return overall_size * op_factor
41: (0)          def _compute_size_by_dict(indices, idx_dict):
42: (4)              """
43: (4)              Computes the product of the elements in indices based on the dictionary
44: (4)              idx_dict.
45: (4)              Parameters
46: (4)              -----
47: (4)              indices : iterable
48: (8)                  Indices to base the product on.
49: (4)              idx_dict : dictionary
50: (8)                  Dictionary of index sizes
51: (4)              Returns
52: (4)              -----
53: (4)              ret : int
54: (8)                  The resulting product.
55: (4)              Examples
56: (4)              -----
57: (4)              >>> _compute_size_by_dict('abbc', {'a': 2, 'b':3, 'c':5})
58: (4)              90
59: (4)              """
60: (4)              ret = 1
61: (4)              for i in indices:
62: (8)                  ret *= idx_dict[i]
63: (4)              return ret
64: (0)          def _find_contraction(positions, input_sets, output_set):
65: (4)              """
66: (4)              Finds the contraction for a given set of input and output sets.
67: (4)              Parameters
68: (4)              -----
69: (4)              positions : iterable
70: (8)                  Integer positions of terms used in the contraction.
71: (4)              input_sets : list
72: (8)                  List of sets that represent the lhs side of the einsum subscript
73: (4)              output_set : set
74: (8)                  Set that represents the rhs side of the overall einsum subscript
75: (4)              Returns
76: (4)              -----
77: (4)              new_result : set
78: (8)                  The indices of the resulting contraction
79: (4)              remaining : list
80: (8)                  List of sets that have not been contracted, the new set is appended to
81: (8)                  the end of this list
82: (4)              idx_removed : set
83: (8)                  Indices removed from the entire contraction
84: (4)              idx_contraction : set
85: (8)                  The indices used in the current contraction
86: (4)              Examples
87: (4)              -----
88: (4)              >>> pos = (0, 1)
89: (4)              >>> isets = [set('ab'), set('bc')]
90: (4)              >>> oset = set('ac')
91: (4)              >>> _find_contraction(pos, isets, oset)
92: (4)              ({'a', 'c'}, [{('a', 'c')}, {'b'}, {'a', 'b', 'c'}])
93: (4)              >>> pos = (0, 2)
94: (4)              >>> isets = [set('abd'), set('ac'), set('bdc')]
95: (4)              >>> oset = set('ac')
96: (4)              >>> _find_contraction(pos, isets, oset)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

97: (4)         ({{'a', 'c'}, [{('a', 'c'), {'a', 'c'}}, {'b', 'd'}, {'a', 'b', 'c', 'd'}])
98: (4)         """
99: (4)         idx_contract = set()
100: (4)        idx_remain = output_set.copy()
101: (4)        remaining = []
102: (4)        for ind, value in enumerate(input_sets):
103: (8)          if ind in positions:
104: (12)            idx_contract |= value
105: (8)          else:
106: (12)            remaining.append(value)
107: (12)            idx_remain |= value
108: (4)        new_result = idx_remain & idx_contract
109: (4)        idx_removed = (idx_contract - new_result)
110: (4)        remaining.append(new_result)
111: (4)        return (new_result, remaining, idx_removed, idx_contract)
112: (0)      def _optimal_path(input_sets, output_set, idx_dict, memory_limit):
113: (4)      """
114: (4)      Computes all possible pair contractions, sieves the results based
115: (4)      on ``memory_limit`` and returns the lowest cost path. This algorithm
116: (4)      scales factorial with respect to the elements in the list ``input_sets``.
117: (4)      Parameters
118: (4)      -----
119: (4)      input_sets : list
120: (8)        List of sets that represent the lhs side of the einsum subscript
121: (4)      output_set : set
122: (8)        Set that represents the rhs side of the overall einsum subscript
123: (4)      idx_dict : dictionary
124: (8)        Dictionary of index sizes
125: (4)      memory_limit : int
126: (8)        The maximum number of elements in a temporary array
127: (4)      Returns
128: (4)      -----
129: (4)      path : list
130: (8)        The optimal contraction order within the memory limit constraint.
131: (4)      Examples
132: (4)      -----
133: (4)      >>> isets = [set('abd'), set('ac'), set('bdc')]
134: (4)      >>> oset = set()
135: (4)      >>> idx_sizes = {'a': 1, 'b':2, 'c':3, 'd':4}
136: (4)      >>> _optimal_path(isets, oset, idx_sizes, 5000)
137: (4)      [(0, 2), (0, 1)]
138: (4)      """
139: (4)      full_results = [(0, [], input_sets)]
140: (4)      for iteration in range(len(input_sets) - 1):
141: (8)        iter_results = []
142: (8)        for curr in full_results:
143: (12)          cost, positions, remaining = curr
144: (12)          for con in itertools.combinations(range(len(input_sets)) -
iteration), 2):
145: (16)            cont = _find_contraction(con, remaining, output_set)
146: (16)            new_result, new_input_sets, idx_removed, idx_contract = cont
147: (16)            new_size = _compute_size_by_dict(new_result, idx_dict)
148: (16)            if new_size > memory_limit:
149: (20)              continue
150: (16)            total_cost = cost + _flop_count(idx_contract, idx_removed,
len(con), idx_dict)
151: (16)            new_pos = positions + [con]
152: (16)            iter_results.append((total_cost, new_pos, new_input_sets))
153: (8)
154: (12)
155: (8)
156: (12)
157: (12)
158: (12)
159: (4)
160: (8)
161: (4)
162: (4)
163: (0)      def _parse_possible_contraction(positions, input_sets, output_set, idx_dict,

```

```

memory_limit, path_cost, naive_cost):
164: (4)          """Compute the cost (removed size + flops) and resultant indices for
165: (4)          performing the contraction specified by ``positions``.
166: (4)          Parameters
167: (4)          -----
168: (4)          positions : tuple of int
169: (8)            The locations of the proposed tensors to contract.
170: (4)          input_sets : list of sets
171: (8)            The indices found on each tensors.
172: (4)          output_set : set
173: (8)            The output indices of the expression.
174: (4)          idx_dict : dict
175: (8)            Mapping of each index to its size.
176: (4)          memory_limit : int
177: (8)            The total allowed size for an intermediary tensor.
178: (4)          path_cost : int
179: (8)            The contraction cost so far.
180: (4)          naive_cost : int
181: (8)            The cost of the unoptimized expression.
182: (4)          Returns
183: (4)          -----
184: (4)          cost : (int, int)
185: (8)            A tuple containing the size of any indices removed, and the flop cost.
186: (4)          positions : tuple of int
187: (8)            The locations of the proposed tensors to contract.
188: (4)          new_input_sets : list of sets
189: (8)            The resulting new list of indices if this proposed contraction is
performed.
190: (4)          """
191: (4)          contract = _find_contraction(positions, input_sets, output_set)
192: (4)          idx_result, new_input_sets, idx_removed, idx_contract = contract
193: (4)          new_size = _compute_size_by_dict(idx_result, idx_dict)
194: (4)          if new_size > memory_limit:
195: (8)            return None
196: (4)          old_sizes = (_compute_size_by_dict(input_sets[p], idx_dict) for p in
positions)
197: (4)          removed_size = sum(old_sizes) - new_size
198: (4)          cost = _flop_count(idx_contract, idx_removed, len(positions), idx_dict)
199: (4)          sort = (-removed_size, cost)
200: (4)          if (path_cost + cost) > naive_cost:
201: (8)            return None
202: (4)          return [sort, positions, new_input_sets]
def _update_other_results(results, best):
203: (0)          """Update the positions and provisional input_sets of ``results`` based on
204: (4)          performing the contraction result ``best``. Remove any involving the
205: (4)          tensors
206: (4)          contracted.
207: (4)          Parameters
208: (4)          -----
209: (4)          results : list
210: (8)            List of contraction results produced by
``_parse_possible_contraction``.
211: (4)          best : list
212: (8)            The best contraction of ``results`` i.e. the one that will be
performed.
213: (4)          Returns
214: (4)          -----
215: (4)          mod_results : list
216: (8)            The list of modified results, updated with outcome of ``best``
contraction.
217: (4)          """
218: (4)          best_con = best[1]
219: (4)          bx, by = best_con
220: (4)          mod_results = []
221: (4)          for cost, (x, y), con_sets in results:
222: (8)            if x in best_con or y in best_con:
223: (12)              continue
224: (8)              del con_sets[by - int(by > x) - int(by > y)]
225: (8)              del con_sets[bx - int(bx > x) - int(bx > y)]

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

226: (8)           con_sets.insert(-1, best[2][-1])
227: (8)           mod_con = x - int(x > bx) - int(x > by), y - int(y > bx) - int(y > by)
228: (8)           mod_results.append((cost, mod_con, con_sets))
229: (4)           return mod_results
230: (0) def _greedy_path(input_sets, output_set, idx_dict, memory_limit):
231: (4)     """
232: (4)     Finds the path by contracting the best pair until the input list is
233: (4)     exhausted. The best pair is found by minimizing the tuple
234: (4)     ``(-prod(indices_removed), cost)``. What this amounts to is prioritizing
235: (4)     matrix multiplication or inner product operations, then Hadamard like
236: (4)     operations, and finally outer operations. Outer products are limited by
237: (4)     ``memory_limit``. This algorithm scales cubically with respect to the
238: (4)     number of elements in the list ``input_sets``.
239: (4)     Parameters
240: (4)     -----
241: (4)     input_sets : list
242: (8)       List of sets that represent the lhs side of the einsum subscript
243: (4)     output_set : set
244: (8)       Set that represents the rhs side of the overall einsum subscript
245: (4)     idx_dict : dictionary
246: (8)       Dictionary of index sizes
247: (4)     memory_limit : int
248: (8)       The maximum number of elements in a temporary array
249: (4)     Returns
250: (4)     -----
251: (4)     path : list
252: (8)       The greedy contraction order within the memory limit constraint.
253: (4)     Examples
254: (4)     -----
255: (4)     >>> isets = [set('abd'), set('ac'), set('bdc')]
256: (4)     >>> oset = set()
257: (4)     >>> idx_sizes = {'a': 1, 'b':2, 'c':3, 'd':4}
258: (4)     >>> _greedy_path(isets, oset, idx_sizes, 5000)
259: (4)     [(0, 2), (0, 1)]
260: (4)     """
261: (4)     if len(input_sets) == 1:
262: (8)       return [(0,)]
263: (4)     elif len(input_sets) == 2:
264: (8)       return [(0, 1)]
265: (4)     contract = _find_contraction(range(len(input_sets)), input_sets,
266: (4)     output_set)
267: (4)     idx_result, new_input_sets, idx_removed, idx_contract = contract
268: (4)     naive_cost = _flop_count(idx_contract, idx_removed, len(input_sets),
269: (4)     idx_dict)
270: (4)     comb_iter = itertools.combinations(range(len(input_sets)), 2)
271: (4)     known_contractions = []
272: (4)     path_cost = 0
273: (8)     path = []
274: (12)    for iteration in range(len(input_sets) - 1):
275: (16)      for positions in comb_iter:
276: (12)        if input_sets[positions[0]].isdisjoint(input_sets[positions[1]]):
277: (16)          continue
278: (12)        result = _parse_possible_contraction(positions, input_sets,
279: (16)        output_set, idx_dict, memory_limit, path_cost,
280: (12)                                              naive_cost)
281: (12)        if result is not None:
282: (16)          known_contractions.append(result)
283: (12)        if len(known_contractions) == 0:
284: (16)          for positions in itertools.combinations(range(len(input_sets)),
285: (12)            2):
286: (16)            result = _parse_possible_contraction(positions, input_sets,
287: (12)            output_set, idx_dict, memory_limit,
288: (16)                                              path_cost, naive_cost)
289: (16)            if result is not None:
285: (20)              known_contractions.append(result)
286: (12)            if len(known_contractions) == 0:
287: (16)              path.append(tuple(range(len(input_sets))))
288: (16)              break
289: (8)            best = min(known_contractions, key=lambda x: x[0])

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

290: (8)           known_contractions = _update_other_results(known_contractions, best)
291: (8)           input_sets = best[2]
292: (8)           new_tensor_pos = len(input_sets) - 1
293: (8)           comb_iter = ((i, new_tensor_pos) for i in range(new_tensor_pos))
294: (8)           path.append(best[1])
295: (8)           path_cost += best[0][1]
296: (4)           return path
297: (0) def _can_dot(inputs, result, idx_removed):
298: (4)     """
299: (4)         Checks if we can use BLAS (np.tensordot) call and its beneficial to do so.
300: (4)         Parameters
301: (4)             -----
302: (4)             inputs : list of str
303: (8)                 Specifies the subscripts for summation.
304: (4)             result : str
305: (8)                 Resulting summation.
306: (4)             idx_removed : set
307: (8)                 Indices that are removed in the summation
308: (4)             Returns
309: (4)             -----
310: (4)             type : bool
311: (8)                 Returns true if BLAS should and can be used, else False
312: (4)             Notes
313: (4)             -----
314: (4)             If the operations is BLAS level 1 or 2 and is not already aligned
315: (4)             we default back to einsum as the memory movement to copy is more
316: (4)             costly than the operation itself.
317: (4)             Examples
318: (4)             -----
319: (4)             >>> _can_dot(['ij', 'jk'], 'ik', set('j'))
320: (4)             True
321: (4)             >>> _can_dot(['ijj', 'jk'], 'ik', set('j'))
322: (4)             False
323: (4)             >>> _can_dot(['ijk', 'ikj'], '', set('ijk'))
324: (4)             False
325: (4)             """
326: (4)             if len(idx_removed) == 0:
327: (8)                 return False
328: (4)             if len(inputs) != 2:
329: (8)                 return False
330: (4)             input_left, input_right = inputs
331: (4)             for c in set(input_left + input_right):
332: (8)                 nl, nr = input_left.count(c), input_right.count(c)
333: (8)                 if (nl > 1) or (nr > 1) or (nl + nr > 2):
334: (12)                     return False
335: (8)                 if nl + nr - 1 == int(c in result):
336: (12)                     return False
337: (4)                 set_left = set(input_left)
338: (4)                 set_right = set(input_right)
339: (4)                 keep_left = set_left - idx_removed
340: (4)                 keep_right = set_right - idx_removed
341: (4)                 rs = len(idx_removed)
342: (4)                 if input_left == input_right:
343: (8)                     return True
344: (4)                 if set_left == set_right:
345: (8)                     return False
346: (4)                 if input_left[-rs:] == input_right[:rs]:
347: (8)                     return True
348: (4)                 if input_left[:rs] == input_right[-rs:]:
349: (8)                     return True
350: (4)                 if input_left[-rs:] == input_right[-rs:]:
351: (8)                     return True
352: (4)                 if input_left[:rs] == input_right[:rs]:
353: (8)                     return True
354: (4)                 if not keep_left or not keep_right:
355: (8)                     return False
356: (4)                 return True
357: (0) def _parse_einsum_input(operands):
358: (4)     """

```

```

359: (4)                                A reproduction of einsum c side einsum parsing in python.
360: (4)                                Returns
361: (4)
362: (4)                                -----
363: (8)                                input_strings : str
364: (4)                                Parsed input strings
365: (8)                                output_string : str
366: (4)                                Parsed output string
367: (8)                                operands : list of array_like
368: (4)                                The operands to use in the numpy contraction
369: (4)
370: (4)                                Examples
371: (4)
372: (4)                                -----
373: (4)                                The operand list is simplified to reduce printing:
374: (4)                                >>> np.random.seed(123)
375: (4)                                >>> a = np.random.rand(4, 4)
376: (4)                                >>> b = np.random.rand(4, 4, 4)
377: (4)                                >>> _parse_einsum_input('...a,...a->...', a, b))
378: (4)                                ('za,xza', 'xz', [a, b]) # may vary
379: (4)                                >>> _parse_einsum_input([Ellipsis, 0], b, [Ellipsis, 0]))
380: (8)                                ('za,xza', 'xz', [a, b]) # may vary
381: (4)
382: (8)                                """
383: (8)                                if len(operands) == 0:
384: (8)                                    raise ValueError("No input operands")
385: (12)                                if isinstance(operands[0], str):
386: (16)                                    subscripts = operands[0].replace(" ", "")
387: (12)                                    operands = [asanyarray(v) for v in operands[1:]]
388: (16)                                    for s in subscripts:
389: (4)                                        if s in '.',->':
390: (8)                                            continue
391: (8)                                        if s not in einsum_symbols:
392: (8)                                            raise ValueError("Character %s is not a valid symbol." % s)
393: (8)
394: (12)                                else:
395: (12)                                    tmp_operands = list(operands)
396: (8)                                    operand_list = []
397: (8)                                    subscript_list = []
398: (8)
399: (8)                                    for p in range(len(operands) // 2):
400: (8)                                        operand_list.append(tmp_operands.pop(0))
401: (12)                                        subscript_list.append(tmp_operands.pop(0))
402: (16)                                    output_list = tmp_operands[-1] if len(tmp_operands) else None
403: (20)                                    operands = [asanyarray(v) for v in operand_list]
404: (16)                                    subscripts = ""
405: (20)                                    last = len(subscript_list) - 1
406: (24)                                    for num, sub in enumerate(subscript_list):
407: (20)                                        for s in sub:
408: (24)                                            if s is Ellipsis:
409: (40)                                                subscripts += "..."
410: (20)                                            else:
411: (12)                                                try:
412: (16)                                                    s = operator.index(s)
413: (8)                                                except TypeError as e:
414: (12)                                                    raise TypeError("For this input type lists must
415: (12)                                                        contain "
416: (16)                                                        "either int or Ellipsis") from e
417: (20)                                                subscripts += einsum_symbols[s]
418: (16)
419: (20)
420: (24)
421: (20)
422: (24)
423: (40)
424: (20)
425: (4)                                if ("-" in subscripts) or (">" in subscripts):

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

426: (8)           invalid = (subscripts.count("-") > 1) or (subscripts.count(">") > 1)
427: (8)           if invalid or (subscripts.count(">") != 1):
428: (12)             raise ValueError("Subscripts can only contain one '->'")
429: (4)           if "." in subscripts:
430: (8)             used = subscripts.replace(".", "").replace(", ", "").replace(">", "")
431: (8)             unused = list(einsum_symbols_set - set(used))
432: (8)             ellipse_inds = "".join(unused)
433: (8)             longest = 0
434: (8)             if ">" in subscripts:
435: (12)               input_tmp, output_sub = subscripts.split(">")
436: (12)               split_subscripts = input_tmp.split(",")
437: (12)               out_sub = True
438: (8)           else:
439: (12)             split_subscripts = subscripts.split(',')
440: (12)             out_sub = False
441: (8)           for num, sub in enumerate(split_subscripts):
442: (12)             if "." in sub:
443: (16)               if (sub.count(".") != 3) or (sub.count("...") != 1):
444: (20)                 raise ValueError("Invalid Ellipses.")
445: (16)               if operands[num].shape == ():
446: (20)                 ellipse_count = 0
447: (16)               else:
448: (20)                 ellipse_count = max(operands[num].ndim, 1)
449: (20)                 ellipse_count -= (len(sub) - 3)
450: (16)               if ellipse_count > longest:
451: (20)                 longest = ellipse_count
452: (16)               if ellipse_count < 0:
453: (20)                 raise ValueError("Ellipses lengths do not match.")
454: (16)               elif ellipse_count == 0:
455: (20)                 split_subscripts[num] = sub.replace('...', '')
456: (16)               else:
457: (20)                 rep_inds = ellipse_inds[-ellipse_count:]
458: (20)                 split_subscripts[num] = sub.replace('...', rep_inds)
459: (8)           subscripts = ",".join(split_subscripts)
460: (8)           if longest == 0:
461: (12)             out_ellipse = ""
462: (8)           else:
463: (12)             out_ellipse = ellipse_inds[-longest:]
464: (8)           if out_sub:
465: (12)             subscripts += ">" + output_sub.replace("...", out_ellipse)
466: (8)           else:
467: (12)             output_subscript = ""
468: (12)             tmp_subscripts = subscripts.replace(", ", "")
469: (12)             for s in sorted(set(tmp_subscripts)):
470: (16)               if s not in (einsum_symbols):
471: (20)                 raise ValueError("Character %s is not a valid symbol." % s)
472: (16)                 if tmp_subscripts.count(s) == 1:
473: (20)                   output_subscript += s
474: (12)             normal_inds = ''.join(sorted(set(output_subscript) -
475: (41)                           set(out_ellipse)))
476: (12)             subscripts += ">" + out_ellipse + normal_inds
477: (4)           if ">" in subscripts:
478: (8)             input_subscripts, output_subscript = subscripts.split(">")
479: (4)           else:
480: (8)             input_subscripts = subscripts
481: (8)             tmp_subscripts = subscripts.replace(", ", "")
482: (8)             output_subscript = ""
483: (8)             for s in sorted(set(tmp_subscripts)):
484: (12)               if s not in einsum_symbols:
485: (16)                 raise ValueError("Character %s is not a valid symbol." % s)
486: (12)                 if tmp_subscripts.count(s) == 1:
487: (16)                   output_subscript += s
488: (4)             for char in output_subscript:
489: (8)               if char not in input_subscripts:
490: (12)                 raise ValueError("Output character %s did not appear in the input" %
491: (29)                               "% char")
492: (4)             if len(input_subscripts.split(',')) != len(operands):
493: (8)               raise ValueError("Number of einsum subscripts must be equal to the "

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

494: (25)                                     "number of operands.")
495: (4)                                      return (input_subscripts, output_subscript, operands)
496: (0)                                      def _einsum_path_dispatcher(*operands, optimize=None, einsum_call=None):
497: (4)                                      return operands
498: (0)                                      @array_function_dispatch(_einsum_path_dispatcher, module='numpy')
499: (0)                                      def einsum_path(*operands, optimize='greedy', einsum_call=False):
500: (4)                                      """
501: (4)                                          einsum_path(subscripts, *operands, optimize='greedy')
502: (4)                                          Evaluates the lowest cost contraction order for an einsum expression by
503: (4)                                          considering the creation of intermediate arrays.
504: (4)                                          Parameters
505: (4)                                          -----
506: (4)                                          subscripts : str
507: (8)                                          Specifies the subscripts for summation.
508: (4)                                          *operands : list of array_like
509: (8)                                          These are the arrays for the operation.
510: (4)                                          optimize : {bool, list, tuple, 'greedy', 'optimal'}
511: (8)                                          Choose the type of path. If a tuple is provided, the second argument
is
512: (8)                                          assumed to be the maximum intermediate size created. If only a single
513: (8)                                          argument is provided the largest input or output array size is used
514: (8)                                          as a maximum intermediate size.
515: (8)                                          * if a list is given that starts with ``einsum_path``, uses this as
the
516: (10)                                         contraction path
517: (8)                                         * if False no optimization is taken
518: (8)                                         * if True defaults to the 'greedy' algorithm
519: (8)                                         * 'optimal' An algorithm that combinatorially explores all possible
520: (10)                                         ways of contracting the listed tensors and chooses the least costly
521: (10)                                         path. Scales exponentially with the number of terms in the
522: (10)                                         contraction.
523: (8)                                         * 'greedy' An algorithm that chooses the best pair contraction
524: (10)                                         at each step. Effectively, this algorithm searches the largest
inner,
525: (10)                                         Hadamard, and then outer products at each step. Scales cubically
with
526: (10)                                         the number of terms in the contraction. Equivalent to the 'optimal'
527: (10)                                         path for most contractions.
528: (8)                                         Default is 'greedy'.
529: (4)                                         Returns
530: (4)                                         -----
531: (4)                                         path : list of tuples
532: (8)                                         A list representation of the einsum path.
533: (4)                                         string_repr : str
534: (8)                                         A printable representation of the einsum path.
535: (4)                                         Notes
536: (4)                                         -----
537: (4)                                         The resulting path indicates which terms of the input contraction should
be
538: (4)                                         contracted first, the result of this contraction is then appended to the
539: (4)                                         end of the contraction list. This list can then be iterated over until all
540: (4)                                         intermediate contractions are complete.
541: (4)                                         See Also
542: (4)                                         -----
543: (4)                                         einsum, linalg.multi_dot
544: (4)                                         Examples
545: (4)                                         -----
546: (4)                                         We can begin with a chain dot example. In this case, it is optimal to
547: (4)                                         contract the ``b`` and ``c`` tensors first as represented by the first
548: (4)                                         element of the path ``(1, 2)``. The resulting tensor is added to the end
549: (4)                                         of the contraction and the remaining contraction ``(0, 1)`` is then
550: (4)                                         completed.
551: (4)                                         >>> np.random.seed(123)
552: (4)                                         >>> a = np.random.rand(2, 2)
553: (4)                                         >>> b = np.random.rand(2, 5)
554: (4)                                         >>> c = np.random.rand(5, 2)
555: (4)                                         >>> path_info = np.einsum_path('ij,jk,kl->il', a, b, c, optimize='greedy')
556: (4)                                         >>> print(path_info[0])
557: (4)                                         ['einsum_path', (1, 2), (0, 1)]

```

```

558: (4)          >>> print(path_info[1])
559: (6)          Complete contraction: ij,jk,kl->il # may vary
560: (13)         Naive scaling: 4
561: (9)          Optimized scaling: 3
562: (10)         Naive FLOP count: 1.600e+02
563: (6)          Optimized FLOP count: 5.600e+01
564: (7)          Theoretical speedup: 2.857
565: (6)          Largest intermediate: 4.000e+00 elements
566: (4)
567: (4)          -----
568: (4)          scaling           current           remaining
569: (7)          3                 kl,jk->jl          ij,jl->il
570: (7)          3                 jl,ij->il          il->il
571: (4)          A more complex index transformation example.
572: (4)          >>> I = np.random.rand(10, 10, 10, 10)
573: (4)          >>> C = np.random.rand(10, 10)
574: (4)          >>> path_info = np.einsum_path('ea,fb,abcd,gc,hd->efgh', C, C, I, C, C,
575: (4)          ...                           optimize='greedy')
576: (4)          >>> print(path_info[0])
577: (4)          ['einsum_path', (0, 2), (0, 3), (0, 2), (0, 1)]
578: (4)          >>> print(path_info[1])
579: (6)          Complete contraction: ea,fb,abcd,gc,hd->efgh # may vary
580: (13)         Naive scaling: 8
581: (9)          Optimized scaling: 5
582: (10)         Naive FLOP count: 8.000e+08
583: (6)          Optimized FLOP count: 8.000e+05
584: (7)          Theoretical speedup: 1000.000
585: (6)          Largest intermediate: 1.000e+04 elements
586: (4)
587: (4)          -----
588: (4)          scaling           current           remaining
589: (7)          5                 abcd,ea->bcde      fb,gc,hd,bcde->efgh
590: (7)          5                 bcde,fb->cdef      gc,hd,cdef->efgh
591: (7)          5                 cdef,gc->defg      hd,defg->efgh
592: (7)          5                 defg,hd->efgh     efgh->efgh
593: (4)          """
594: (4)          path_type = optimize
595: (4)          if path_type is True:
596: (8)              path_type = 'greedy'
597: (4)          if path_type is None:
598: (8)              path_type = False
599: (4)          explicit_einsum_path = False
600: (4)          memory_limit = None
601: (4)          if (path_type is False) or isinstance(path_type, str):
602: (8)              pass
603: (4)          elif len(path_type) and (path_type[0] == 'einsum_path'):
604: (8)              explicit_einsum_path = True
605: (4)          elif ((len(path_type) == 2) and isinstance(path_type[0], str) and
606: (12)              isinstance(path_type[1], (int, float))):
607: (8)              memory_limit = int(path_type[1])
608: (8)              path_type = path_type[0]
609: (4)          else:
610: (8)              raise TypeError("Did not understand the path: %s" % str(path_type))
611: (4)          einsum_call_arg = einsum_call
612: (4)          input_subscripts, output_subscript, operands =
_parse_einsum_input(operands)
613: (4)          input_list = input_subscripts.split(',')
614: (4)          input_sets = [set(x) for x in input_list]
615: (4)          output_set = set(output_subscript)
616: (4)          indices = set(input_subscripts.replace(',', ' '))
617: (4)          dimension_dict = {}
618: (4)          broadcast_indices = [[] for x in range(len(input_list))]
619: (4)          for tnum, term in enumerate(input_list):
620: (8)              sh = operands[tnum].shape
621: (8)              if len(sh) != len(term):
622: (12)                  raise ValueError("Einstein sum subscript %s does not contain the "
623: (29)                      "correct number of indices for operand %d."
624: (29)                      % (input_subscripts[tnum], tnum))
625: (8)              for cnum, char in enumerate(term):

```

```

626: (12)           dim = sh[cnum]
627: (12)           if dim == 1:
628: (16)             broadcast_indices[tnum].append(char)
629: (12)           if char in dimension_dict.keys():
630: (16)             if dimension_dict[char] == 1:
631: (20)               dimension_dict[char] = dim
632: (16)             elif dim not in (1, dimension_dict[char]):
633: (20)               raise ValueError("Size of label '%s' for operand %d (%d) "
634: (37)                   "does not match previous terms (%d)."
635: (37)                   % (char, tnum, dimension_dict[char],
dim))
636: (12)           else:
637: (16)             dimension_dict[char] = dim
638: (4)           broadcast_indices = [set(x) for x in broadcast_indices]
639: (4)           size_list = [_compute_size_by_dict(term, dimension_dict)
640: (17)                   for term in input_list + [output_subscript]]
641: (4)           max_size = max(size_list)
642: (4)           if memory_limit is None:
643: (8)             memory_arg = max_size
644: (4)           else:
645: (8)             memory_arg = memory_limit
646: (4)           inner_product = (sum(len(x) for x in input_sets) - len(indices)) > 0
647: (4)           naive_cost = _flop_count(indices, inner_product, len(input_list),
dimension_dict)
648: (4)           if explicit_einsum_path:
649: (8)             path = path_type[1:]
650: (4)           elif (
651: (8)             (path_type is False)
652: (8)             or (len(input_list) in [1, 2])
653: (8)             or (indices == output_set)
654: (4)           ):
655: (8)             path = [tuple(range(len(input_list)))]
656: (4)           elif path_type == "greedy":
657: (8)             path = _greedy_path(input_sets, output_set, dimension_dict,
memory_arg)
658: (4)           elif path_type == "optimal":
659: (8)             path = _optimal_path(input_sets, output_set, dimension_dict,
memory_arg)
660: (4)           else:
661: (8)             raise KeyError("Path name %s not found", path_type)
662: (4)           cost_list, scale_list, size_list, contraction_list = [], [], [], []
663: (4)           for cnum, contract_inds in enumerate(path):
664: (8)             contract_inds = tuple(sorted(list(contract_inds), reverse=True))
665: (8)             contract = _find_contraction(contract_inds, input_sets, output_set)
666: (8)             out_inds, input_sets, idx_removed, idx_contract = contract
667: (8)             cost = _flop_count(idx_contract, idx_removed, len(contract_inds),
dimension_dict)
668: (8)             cost_list.append(cost)
669: (8)             scale_list.append(len(idx_contract))
670: (8)             size_list.append(_compute_size_by_dict(out_inds, dimension_dict))
671: (8)             bcast = set()
672: (8)             tmp_inputs = []
673: (8)             for x in contract_inds:
674: (12)               tmp_inputs.append(input_list.pop(x))
675: (12)               bcast |= broadcast_indices.pop(x)
676: (8)             new_bcast_inds = bcast - idx_removed
677: (8)             if not len(idx_removed) & bcast:
678: (12)               do blas = _can_dot(tmp_inputs, out_inds, idx_removed)
679: (8)             else:
680: (12)               do blas = False
681: (8)             if (cnum - len(path)) == -1:
682: (12)               idx_result = output_subscript
683: (8)             else:
684: (12)               sort_result = [(dimension_dict[ind], ind) for ind in out_inds]
685: (12)               idx_result = "".join([x[1] for x in sorted(sort_result)])
686: (8)             input_list.append(idx_result)
687: (8)             broadcast_indices.append(new_bcast_inds)
688: (8)             einsum_str = ",".join(tmp_inputs) + ">" + idx_result
689: (8)             contraction = (contract_inds, idx_removed, einsum_str, input_list[:],
```

```

do_blas)
690: (8)           contraction_list.append(contraction)
691: (4)           opt_cost = sum(cost_list) + 1
692: (4)           if len(input_list) != 1:
693: (8)             raise RuntimeError(
694: (12)               "Invalid einsum_path is specified: {} more operands has to be "
695: (12)               "contracted.".format(len(input_list) - 1))
696: (4)           if einsum_call_arg:
697: (8)             return (operands, contraction_list)
overall_contraction = input_subscripts + "->" + output_subscript
698: (4)           header = ("scaling", "current", "remaining")
699: (4)           speedup = naive_cost / opt_cost
700: (4)           max_i = max(size_list)
701: (4)           path_print = " Complete contraction: %s\n" % overall_contraction
702: (4)           path_print += "      Naive scaling: %d\n" % len(indices)
703: (4)           path_print += "      Optimized scaling: %d\n" % max(scale_list)
704: (4)           path_print += "      Naive FLOP count: %.3e\n" % naive_cost
705: (4)           path_print += "      Optimized FLOP count: %.3e\n" % opt_cost
706: (4)           path_print += "      Theoretical speedup: %3.3f\n" % speedup
707: (4)           path_print += "      Largest intermediate: %.3e elements\n" % max_i
708: (4)           path_print += "-" * 74 + "\n"
709: (4)           path_print += "%6s %24s %40s\n" % header
710: (4)           path_print += "-" * 74
711: (4)           for n, contraction in enumerate(contraction_list):
712: (4)             inds, idx_rm, einsum_str, remaining, blas = contraction
713: (8)             remaining_str = ",".join(remaining) + "->" + output_subscript
714: (8)             path_run = (scale_list[n], einsum_str, remaining_str)
715: (8)             path_print += "\n%4d    %24s %40s" % path_run
716: (8)             path = ['einsum_path'] + path
717: (4)             return (path, path_print)
718: (4)         def _einsum_dispatcher(*operands, out=None, optimize=None, **kwargs):
719: (0)             yield from operands
720: (4)             yield out
721: (4)         @array_function_dispatch(_einsum_dispatcher, module='numpy')
722: (0)         def einsum(*operands, out=None, optimize=False, **kwargs):
723: (0)             """
724: (4)             einsum(subscripts, *operands, out=None, dtype=None, order='K',
725: (11)               casting='safe', optimize=False)
726: (4)             Evaluates the Einstein summation convention on the operands.
727: (4)             Using the Einstein summation convention, many common multi-dimensional,
728: (4)             linear algebraic array operations can be represented in a simple fashion.
729: (4)             In *implicit* mode `einsum` computes these values.
730: (4)             In *explicit* mode, `einsum` provides further flexibility to compute
731: (4)             other array operations that might not be considered classical Einstein
732: (4)             summation operations, by disabling, or forcing summation over specified
733: (4)             subscript labels.
734: (4)             See the notes and examples for clarification.
735: (4)             Parameters
736: (4)             -----
737: (4)             subscripts : str
738: (4)               Specifies the subscripts for summation as comma separated list of
739: (8)               subscript labels. An implicit (classical Einstein summation)
740: (8)               calculation is performed unless the explicit indicator '->' is
741: (8)               included as well as subscript labels of the precise output form.
742: (8)             operands : list of array_like
743: (4)               These are the arrays for the operation.
744: (8)             out : ndarray, optional
745: (4)               If provided, the calculation is done into this array.
746: (8)             dtype : {data-type, None}, optional
747: (4)               If provided, forces the calculation to use the data type specified.
748: (8)               Note that you may have to also give a more liberal `casting`
749: (8)               parameter to allow the conversions. Default is None.
750: (8)             order : {'C', 'F', 'A', 'K'}, optional
751: (4)               Controls the memory layout of the output. 'C' means it should
752: (8)               be C contiguous. 'F' means it should be Fortran contiguous,
753: (8)               'A' means it should be 'F' if the inputs are all 'F', 'C' otherwise.
754: (8)               'K' means it should be as close to the layout as the inputs as
755: (8)               is possible, including arbitrarily permuted axes.
756: (8)               Default is 'K'.
757: (8)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

758: (4) casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
759: (8)     Controls what kind of data casting may occur. Setting this to
760: (8)     'unsafe' is not recommended, as it can adversely affect accumulations.
761: (10)    * 'no' means the data types should not be cast at all.
762: (10)    * 'equiv' means only byte-order changes are allowed.
763: (10)    * 'safe' means only casts which can preserve values are allowed.
764: (10)    * 'same_kind' means only safe casts or casts within a kind,
765: (12)        like float64 to float32, are allowed.
766: (10)    * 'unsafe' means any data conversions may be done.
767: (8)        Default is 'safe'.
768: (4) optimize : {False, True, 'greedy', 'optimal'}, optional
769: (8)     Controls if intermediate optimization should occur. No optimization
770: (8)     will occur if False and True will default to the 'greedy' algorithm.
771: (8)     Also accepts an explicit contraction list from the ``np.einsum_path``
772: (8)     function. See ``np.einsum_path`` for more details. Defaults to False.
773: (4) Returns
774: (4) -----
775: (4) output : ndarray
776: (8)     The calculation based on the Einstein summation convention.
777: (4) See Also
778: (4) -----
779: (4) einsum_path, dot, inner, outer, tensordot, linalg.multi_dot
780: (4) einops :
781: (8)     similar verbose interface is provided by
782: (8)     `einops <https://github.com/arogozhnikov/einops>`_ package to cover
783: (8)     additional operations: transpose, reshape/flatten, repeat/tile,
784: (8)     squeeze/unsqueeze and reductions.
785: (4) opt_einsum :
786: (8)     `opt_einsum <https://optimized-einsum.readthedocs.io/en/stable/>`_
787: (8)     optimizes contraction order for einsum-like expressions
788: (8)     in backend-agnostic manner.
789: (4) Notes
790: (4) -----
791: (4) .. versionadded:: 1.6.0
792: (4) The Einstein summation convention can be used to compute
793: (4) many multi-dimensional, linear algebraic array operations. `einsum`
794: (4) provides a succinct way of representing these.
795: (4) A non-exhaustive list of these operations,
796: (4) which can be computed by `einsum`, is shown below along with examples:
797: (4) * Trace of an array, :py:func:`numpy.trace`.
798: (4) * Return a diagonal, :py:func:`numpy.diag`.
799: (4) * Array axis summations, :py:func:`numpy.sum`.
800: (4) * Transpositions and permutations, :py:func:`numpy.transpose`.
801: (4) * Matrix multiplication and dot product, :py:func:`numpy.matmul`  

:py:func:`numpy.dot`.
802: (4) * Vector inner and outer products, :py:func:`numpy.inner`  

:py:func:`numpy.outer`.
803: (4) * Broadcasting, element-wise and scalar multiplication,
:py:func:`numpy.multiply`.
804: (4) * Tensor contractions, :py:func:`numpy.tensordot`.
805: (4) * Chained array operations, in efficient calculation order,  

:py:func:`numpy.einsum_path`.
806: (4) The subscripts string is a comma-separated list of subscript labels,
807: (4) where each label refers to a dimension of the corresponding operand.
808: (4) Whenever a label is repeated it is summed, so ``np.einsum('i,i', a, b)``
809: (4) is equivalent to :py:func:`np.inner(a,b) <numpy.inner>`. If a label
810: (4) appears only once, it is not summed, so ``np.einsum('i', a)`` produces a
811: (4) view of ``a`` with no changes. A further example ``np.einsum('ij,jk', a,
b)``
812: (4) describes traditional matrix multiplication and is equivalent to
813: (4) :py:func:`np.matmul(a,b) <numpy.matmul>`. Repeated subscript labels in one
814: (4) operand take the diagonal. For example, ``np.einsum('ii', a)`` is
equivalent
815: (4) to :py:func:`np.trace(a) <numpy.trace>`.
816: (4) In *implicit mode*, the chosen subscripts are important
817: (4) since the axes of the output are reordered alphabetically. This
818: (4) means that ``np.einsum('ij', a)`` doesn't affect a 2D array, while
819: (4) ``np.einsum('ji', a)`` takes its transpose. Additionally,
820: (4) ``np.einsum('ij,jk', a, b)`` returns a matrix multiplication, while,
```

```

821: (4) ``np.einsum('ij,jh', a, b)`` returns the transpose of the
822: (4) multiplication since subscript 'h' precedes subscript 'i'.
823: (4) In *explicit mode* the output can be directly controlled by
824: (4) specifying output subscript labels. This requires the
825: (4) identifier ' $\rightarrow$ ' as well as the list of output subscript labels.
826: (4) This feature increases the flexibility of the function since
827: (4) summing can be disabled or forced when required. The call
828: (4) ``np.einsum('i $\rightarrow$ ', a)`` is like :py:func:`np.sum(a, axis=-1) <numpy.sum>`,
829: (4) and ``np.einsum('ii $\rightarrow$ i', a)`` is like :py:func:`np.diag(a) <numpy.diag>`.
830: (4) The difference is that `einsum` does not allow broadcasting by default.
831: (4) Additionally ``np.einsum('ij,jh $\rightarrow$ ih', a, b)`` directly specifies the
832: (4) order of the output subscript labels and therefore returns matrix
833: (4) multiplication, unlike the example above in implicit mode.
834: (4) To enable and control broadcasting, use an ellipsis. Default
835: (4) NumPy-style broadcasting is done by adding an ellipsis
836: (4) to the left of each term, like ``np.einsum('...ii $\rightarrow$ ...i', a)``.
837: (4) To take the trace along the first and last axes,
838: (4) you can do ``np.einsum('i...i', a)``, or to do a matrix-matrix
839: (4) product with the left-most indices instead of rightmost, one can do
840: (4) ``np.einsum('ij...,jk... $\rightarrow$ ik...', a, b)``.
841: (4) When there is only one operand, no axes are summed, and no output
842: (4) parameter is provided, a view into the operand is returned instead
843: (4) of a new array. Thus, taking the diagonal as ``np.einsum('ii $\rightarrow$ i', a)``
844: (4) produces a view (changed in version 1.10.0).
845: (4) `einsum` also provides an alternative way to provide the subscripts
846: (4) and operands as ``einsum(op0, sublist0, op1, sublist1, ...,
[sublistout])``.
847: (4) If the output shape is not provided in this format `einsum` will be
848: (4) calculated in implicit mode, otherwise it will be performed explicitly.
849: (4) The examples below have corresponding `einsum` calls with the two
850: (4) parameter methods.
851: (4) .. versionadded:: 1.10.0
852: (4) Views returned from einsum are now writeable whenever the input array
853: (4) is writeable. For example, ``np.einsum('ijk... $\rightarrow$ kji...', a)`` will now
854: (4) have the same effect as :py:func:`np.swapaxes(a, 0, 2) <numpy.swapaxes>`-
855: (4) and ``np.einsum('ii $\rightarrow$ i', a)`` will return a writeable view of the diagonal
856: (4) of a 2D array.
857: (4) .. versionadded:: 1.12.0
858: (4) Added the ``optimize`` argument which will optimize the contraction order
859: (4) of an einsum expression. For a contraction with three or more operands
this
860: (4) can greatly increase the computational efficiency at the cost of a larger
861: (4) memory footprint during computation.
862: (4) Typically a 'greedy' algorithm is applied which empirical tests have shown
863: (4) returns the optimal path in the majority of cases. In some cases 'optimal'
864: (4) will return the superlative path through a more expensive, exhaustive
search.
865: (4) For iterative calculations it may be advisable to calculate the optimal
path
866: (4) once and reuse that path by supplying it as an argument. An example is
given
867: (4) below.
868: (4) See :py:func:`numpy.einsum_path` for more details.
869: (4) Examples
870: (4) -----
871: (4) >>> a = np.arange(25).reshape(5,5)
872: (4) >>> b = np.arange(5)
873: (4) >>> c = np.arange(6).reshape(2,3)
874: (4) Trace of a matrix:
875: (4) >>> np.einsum('ii', a)
876: (4) 60
877: (4) >>> np.einsum(a, [0,0])
878: (4) 60
879: (4) >>> np.trace(a)
880: (4) 60
881: (4) Extract the diagonal (requires explicit form):
882: (4) >>> np.einsum('ii $\rightarrow$ i', a)
883: (4) array([ 0,  6, 12, 18, 24])
884: (4) >>> np.einsum(a, [0,0], [0])

```

```

885: (4) array([ 0,  6, 12, 18, 24])
886: (4) >>> np.diag(a)
887: (4) array([ 0,  6, 12, 18, 24])
888: (4) Sum over an axis (requires explicit form):
889: (4) >>> np.einsum('ij->i', a)
890: (4) array([ 10, 35, 60, 85, 110])
891: (4) >>> np.einsum(a, [0,1], [0])
892: (4) array([ 10, 35, 60, 85, 110])
893: (4) >>> np.sum(a, axis=1)
894: (4) array([ 10, 35, 60, 85, 110])
895: (4) For higher dimensional arrays summing a single axis can be done with
ellipsis:
896: (4) >>> np.einsum('...j->...', a)
897: (4) array([ 10, 35, 60, 85, 110])
898: (4) >>> np.einsum(a, [Ellipsis,1], [Ellipsis])
899: (4) array([ 10, 35, 60, 85, 110])
900: (4) Compute a matrix transpose, or reorder any number of axes:
901: (4) >>> np.einsum('ji', c)
902: (4) array([[0, 3],
903: (11)     [1, 4],
904: (11)     [2, 5]])
905: (4) >>> np.einsum('ij->ji', c)
906: (4) array([[0, 3],
907: (11)     [1, 4],
908: (11)     [2, 5]])
909: (4) >>> np.einsum(c, [1,0])
910: (4) array([[0, 3],
911: (11)     [1, 4],
912: (11)     [2, 5]])
913: (4) >>> np.transpose(c)
914: (4) array([[0, 3],
915: (11)     [1, 4],
916: (11)     [2, 5]])
917: (4) Vector inner products:
918: (4) >>> np.einsum('i,i', b, b)
919: (4) 30
920: (4) >>> np.einsum(b, [0], b, [0])
921: (4) 30
922: (4) >>> np.inner(b,b)
923: (4) 30
924: (4) Matrix vector multiplication:
925: (4) >>> np.einsum('ij,j', a, b)
926: (4) array([ 30, 80, 130, 180, 230])
927: (4) >>> np.einsum(a, [0,1], b, [1])
928: (4) array([ 30, 80, 130, 180, 230])
929: (4) >>> np.dot(a, b)
930: (4) array([ 30, 80, 130, 180, 230])
931: (4) >>> np.einsum('...j,j', a, b)
932: (4) array([ 30, 80, 130, 180, 230])
933: (4) Broadcasting and scalar multiplication:
934: (4) >>> np.einsum('..., ...', 3, c)
935: (4) array([[ 0,  3,  6],
936: (11)     [ 9, 12, 15]])
937: (4) >>> np.einsum(',ij', 3, c)
938: (4) array([[ 0,  3,  6],
939: (11)     [ 9, 12, 15]])
940: (4) >>> np.einsum(3, [Ellipsis], c, [Ellipsis])
941: (4) array([[ 0,  3,  6],
942: (11)     [ 9, 12, 15]])
943: (4) >>> np.multiply(3, c)
944: (4) array([[ 0,  3,  6],
945: (11)     [ 9, 12, 15]])
946: (4) Vector outer product:
947: (4) >>> np.einsum('i,j', np.arange(2)+1, b)
948: (4) array([[ 0,  1,  2,  3,  4],
949: (11)     [ 0,  2,  4,  6,  8]])
950: (4) >>> np.einsum(np.arange(2)+1, [0], b, [1])
951: (4) array([[ 0,  1,  2,  3,  4],
952: (11)     [ 0,  2,  4,  6,  8]])

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

953: (4)          >>> np.outer(np.arange(2)+1, b)
954: (4)          array([[0, 1, 2, 3, 4],
955: (11)           [0, 2, 4, 6, 8]])
956: (4)          Tensor contraction:
957: (4)          >>> a = np.arange(60.).reshape(3,4,5)
958: (4)          >>> b = np.arange(24.).reshape(4,3,2)
959: (4)          >>> np.einsum('ijk,jil->kl', a, b)
960: (4)          array([[4400., 4730.],
961: (11)           [4532., 4874.],
962: (11)           [4664., 5018.],
963: (11)           [4796., 5162.],
964: (11)           [4928., 5306.]])
965: (4)          >>> np.einsum(a, [0,1,2], b, [1,0,3], [2,3])
966: (4)          array([[4400., 4730.],
967: (11)           [4532., 4874.],
968: (11)           [4664., 5018.],
969: (11)           [4796., 5162.],
970: (11)           [4928., 5306.]])
971: (4)          >>> np.tensordot(a,b, axes=([1,0],[0,1]))
972: (4)          array([[4400., 4730.],
973: (11)           [4532., 4874.],
974: (11)           [4664., 5018.],
975: (11)           [4796., 5162.],
976: (11)           [4928., 5306.]])
977: (4)          Writeable returned arrays (since version 1.10.0):
978: (4)          >>> a = np.zeros((3, 3))
979: (4)          >>> np.einsum('ii->i', a)[:] = 1
980: (4)          >>> a
981: (4)          array([[1., 0., 0.],
982: (11)           [0., 1., 0.],
983: (11)           [0., 0., 1.]])
984: (4)          Example of ellipsis use:
985: (4)          >>> a = np.arange(6).reshape((3,2))
986: (4)          >>> b = np.arange(12).reshape((4,3))
987: (4)          >>> np.einsum('ki,jk->ij', a, b)
988: (4)          array([[10, 28, 46, 64],
989: (11)           [13, 40, 67, 94]])
990: (4)          >>> np.einsum('ki,...k->i...', a, b)
991: (4)          array([[10, 28, 46, 64],
992: (11)           [13, 40, 67, 94]])
993: (4)          >>> np.einsum('k...,jk', a, b)
994: (4)          array([[10, 28, 46, 64],
995: (11)           [13, 40, 67, 94]])
996: (4)          Chained array operations. For more complicated contractions, speed ups
997: (4)          might be achieved by repeatedly computing a 'greedy' path or pre-computing
the
998: (4)          'optimal' path and repeatedly applying it, using an
999: (4)          `einsum_path` insertion (since version 1.12.0). Performance improvements
can be
1000: (4)          particularly significant with larger arrays:
1001: (4)          >>> a = np.ones(64).reshape(2,4,8)
1002: (4)          Basic `einsum`: ~1520ms (benchmarked on 3.1GHz Intel i5.)
1003: (4)          >>> for iteration in range(500):
1004: (4)              ...      _ = np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a,a)
1005: (4)          Sub-optimal `einsum` (due to repeated path calculation time): ~330ms
1006: (4)          >>> for iteration in range(500):
1007: (4)              ...      _ = np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a,a,
optimize='optimal')
1008: (4)          Greedy `einsum` (faster optimal path approximation): ~160ms
1009: (4)          >>> for iteration in range(500):
1010: (4)              ...      _ = np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a,a,
optimize='greedy')
1011: (4)          Optimal `einsum` (best usage pattern in some use cases): ~110ms
1012: (4)          >>> path = np.einsum_path('ijk,ilm,njm,nlk,abc->',a,a,a,a,a,a,
optimize='optimal')[0]
1013: (4)          >>> for iteration in range(500):
1014: (4)              ...      _ = np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a,a, optimize=path)
1015: (4)          """"
1016: (4)          specified_out = out is not None

```

```

1017: (4)         if optimize is False:
1018: (8)             if specified_out:
1019: (12)                 kwargs['out'] = out
1020: (8)             return c_einsum(*operands, **kwargs)
1021: (4)         valid_einsum_kwargs = ['dtype', 'order', 'casting']
1022: (4)         unknown_kwargs = [k for (k, v) in kwargs.items() if
1023: (22)                     k not in valid_einsum_kwargs]
1024: (4)         if len(unknown_kwargs):
1025: (8)             raise TypeError("Did not understand the following kwargs: %s"
1026: (24)                         % unknown_kwargs)
1027: (4)         operands, contraction_list = einsum_path(*operands, optimize=optimize,
1028: (45)                         einsum_call=True)
1029: (4)         output_order = kwargs.pop('order', 'K')
1030: (4)         if output_order.upper() == 'A':
1031: (8)             if all(arr.flags.f_contiguous for arr in operands):
1032: (12)                 output_order = 'F'
1033: (8)             else:
1034: (12)                 output_order = 'C'
1035: (4)         for num, contraction in enumerate(contraction_list):
1036: (8)             inds, idx_rm, einsum_str, remaining, blas = contraction
1037: (8)             tmp_operands = [operands.pop(x) for x in inds]
1038: (8)             handle_out = specified_out and ((num + 1) == len(contraction_list))
1039: (8)             if blas:
1040: (12)                 input_str, results_index = einsum_str.split('->')
1041: (12)                 input_left, input_right = input_str.split(',')
1042: (12)                 tensor_result = input_left + input_right
1043: (12)                 for s in idx_rm:
1044: (16)                     tensor_result = tensor_result.replace(s, "")
1045: (12)                 left_pos, right_pos = [], []
1046: (12)                 for s in sorted(idx_rm):
1047: (16)                     left_pos.append(input_left.find(s))
1048: (16)                     right_pos.append(input_right.find(s))
1049: (12)                 new_view = tensordot(*tmp_operands, axes=(tuple(left_pos),
tuple(right_pos)))
1050: (12)                 if (tensor_result != results_index) or handle_out:
1051: (16)                     if handle_out:
1052: (20)                         kwargs["out"] = out
1053: (16)                     new_view = c_einsum(tensor_result + '->' + results_index,
new_view, **kwargs)
1054: (8)                 else:
1055: (12)                     if handle_out:
1056: (16)                         kwargs["out"] = out
1057: (12)                         new_view = c_einsum(einsum_str, *tmp_operands, **kwargs)
1058: (8)                         operands.append(new_view)
1059: (8)                         del tmp_operands, new_view
1060: (4)             if specified_out:
1061: (8)                 return out
1062: (4)             else:
1063: (8)                 return asanyarray(operands[0], order=output_order)

```

-----  
File 48 - fromnumeric.py:

```

1: (0)         """Module containing non-deprecated functions borrowed from Numeric.
2: (0)         """
3: (0)         import functools
4: (0)         import types
5: (0)         import warnings
6: (0)         import numpy as np
7: (0)         from .._utils import set_module
8: (0)         from . import multiarray as mu
9: (0)         from . import overrides
10: (0)         from . import umath as um
11: (0)         from . import numerictypes as nt
12: (0)         from .multiarray import asarray, array, asanyarray, concatenate
13: (0)         from . import _methods
14: (0)         _dt_ = nt.sctype2char
15: (0)         __all__ = [

```

```

16: (4)          'all', 'alltrue', 'amax', 'amin', 'any', 'argmax',
17: (4)          'argmin', 'argpartition', 'argsort', 'around', 'choose', 'clip',
18: (4)          'compress', 'cumprod', 'cumproduct', 'cumsum', 'diagonal', 'mean',
19: (4)          'max', 'min',
20: (4)          'ndim', 'nonzero', 'partition', 'prod', 'product', 'ptp', 'put',
21: (4)          'ravel', 'repeat', 'reshape', 'resize', 'round', 'round_',
22: (4)          'searchsorted', 'shape', 'size', 'sometrue', 'sort', 'squeeze',
23: (4)          'std', 'sum', 'swapaxes', 'take', 'trace', 'transpose', 'var',
24: (0)
25: (0)          ]
26: (0)          _gentype = types.GeneratorType
27: (0)          _sum_ = sum
28: (4)          array_function_dispatch = functools.partial(
29: (4)              overrides.array_function_dispatch, module='numpy')
def _wrapit(obj, method, *args, **kwds):
    try:
        wrap = obj.__array_wrap__
    except AttributeError:
        wrap = None
    result = getattr(asarray(obj), method)(*args, **kwds)
    if wrap:
        if not isinstance(result, mu.ndarray):
            result = asarray(result)
        result = wrap(result)
    return result
def _wrapfunc(obj, method, *args, **kwds):
    bound = getattr(obj, method, None)
    if bound is None:
        return _wrapit(obj, method, *args, **kwds)
    try:
        return bound(*args, **kwds)
    except TypeError:
        return _wrapit(obj, method, *args, **kwds)
def _wrapreduction(obj, ufunc, method, axis, dtype, out, **kwargs):
    passkwargs = {k: v for k, v in kwargs.items()
                  if v is not np._NoValue}
    if type(obj) is not mu.ndarray:
        try:
            reduction = getattr(obj, method)
        except AttributeError:
            pass
        else:
            if dtype is not None:
                return reduction(axis=axis, dtype=dtype, out=out,
                                 **passkwargs)
            else:
                return reduction(axis=axis, out=out, **passkwargs)
    return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
def _take_dispatcher(a, indices, axis=None, out=None, mode=None):
    return (a, out)
@array_function_dispatch(_take_dispatcher)
def take(a, indices, axis=None, out=None, mode='raise'):
    """
    Take elements from an array along an axis.
    When axis is not None, this function does the same thing as "fancy"
    indexing (indexing arrays using arrays); however, it can be easier to use
    if you need elements along a given axis. A call such as
    ``np.take(arr, indices, axis=3)`` is equivalent to
    ``arr[:, :, :, indices, ...]``.
    Explained without fancy indexing, this is equivalent to the following use
    of `ndindex`, which sets each of ``ii``, ``jj``, and ``kk`` to a tuple of
    indices::
        Ni, Nk = a.shape[:axis], a.shape[axis+1:]
        Nj = indices.shape
        for ii in ndindex(Ni):
            for jj in ndindex(Nj):
                for kk in ndindex(Nk):
                    out[ii + jj + kk] = a[ii + (indices[jj],) + kk]
    Parameters
    -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

84: (4)     a : array_like (Ni..., M, Nk...)
85: (8)       The source array.
86: (4)     indices : array_like (Nj...)
87: (8)       The indices of the values to extract.
88: (8)       .. versionadded:: 1.8.0
89: (8)       Also allow scalars for indices.
90: (4)     axis : int, optional
91: (8)       The axis over which to select values. By default, the flattened
92: (8)       input array is used.
93: (4)     out : ndarray, optional (Ni..., Nj..., Nk...)
94: (8)       If provided, the result will be placed in this array. It should
95: (8)       be of the appropriate shape and dtype. Note that `out` is always
96: (8)       buffered if `mode='raise'`; use other modes for better performance.
97: (4)     mode : {'raise', 'wrap', 'clip'}, optional
98: (8)       Specifies how out-of-bounds indices will behave.
99: (8)       * 'raise' -- raise an error (default)
100: (8)      * 'wrap' -- wrap around
101: (8)      * 'clip' -- clip to the range
102: (8)       'clip' mode means that all indices that are too large are replaced
103: (8)       by the index that addresses the last element along that axis. Note
104: (8)       that this disables indexing with negative numbers.
105: (4)   Returns
106: (4)   -----
107: (4)   out : ndarray (Ni..., Nj..., Nk...)
108: (8)       The returned array has the same type as `a`.
109: (4)   See Also
110: (4)   -----
111: (4)   compress : Take elements using a boolean mask
112: (4)   ndarray.take : equivalent method
113: (4)   take_along_axis : Take elements by matching the array and the index arrays
114: (4)
115: (4)
116: (4)   Notes
117: (4)   -----
118: (4)   By eliminating the inner loop in the description above, and using `s_` to
119: (8)   build simple slice objects, `take` can be expressed in terms of applying
120: (8)   fancy indexing to each 1-d slice:::
121: (12)  Ni, Nk = a.shape[:axis], a.shape[axis+1:]
122: (16)  for ii in ndindex(Ni):
123: (4)    for kk in ndindex(Nj):
124: (8)      out[ii + s_[...,] + kk] = a[ii + s_[:,] + kk][indices]
125: (4)   For this reason, it is equivalent to (but faster than) the following use
126: (4)   of `apply_along_axis`::
127: (4)   Examples
128: (4)   -----
129: (4)   >>> a = [4, 3, 5, 7, 6, 8]
130: (4)   >>> indices = [0, 1, 4]
131: (4)   >>> np.take(a, indices)
132: (4)   array([4, 3, 6])
133: (4)   In this example if `a` is an ndarray, "fancy" indexing can be used.
134: (4)   >>> a = np.array(a)
135: (4)   >>> a[indices]
136: (4)   array([4, 3, 6])
137: (4)   If `indices` is not one dimensional, the output also has these dimensions.
138: (4)   >>> np.take(a, [[0, 1], [2, 3]])
139: (11)  array([[4, 3],
140: (4)           [5, 7]])
141: (4)   """
142: (0)   return _wrapfunc(a, 'take', indices, axis=axis, out=out, mode=mode)
143: (4) def _reshape_dispatcher(a, newshape, order=None):
144: (0)   return (a,)
145: (0) @array_function_dispatch(_reshape_dispatcher)
146: (4) def reshape(a, newshape, order='C'):
147: (4)   """
148: (4)   Gives a new shape to an array without changing its data.
149: (4)   Parameters
150: (4)   -----
151: (8)   a : array_like
152: (4)       Array to be reshaped.

```

newshape : int or tuple of ints

```

153: (8)           The new shape should be compatible with the original shape. If
154: (8)           an integer, then the result will be a 1-D array of that length.
155: (8)           One shape dimension can be -1. In this case, the value is
156: (8)           inferred from the length of the array and remaining dimensions.
157: (4)           order : {'C', 'F', 'A'}, optional
158: (8)           Read the elements of `a` using this index order, and place the
159: (8)           elements into the reshaped array using this index order. 'C'
160: (8)           means to read / write the elements using C-like index order,
161: (8)           with the last axis index changing fastest, back to the first
162: (8)           axis index changing slowest. 'F' means to read / write the
163: (8)           elements using Fortran-like index order, with the first index
164: (8)           changing fastest, and the last index changing slowest. Note that
165: (8)           the 'C' and 'F' options take no account of the memory layout of
166: (8)           the underlying array, and only refer to the order of indexing.
167: (8)           'A' means to read / write the elements in Fortran-like index
168: (8)           order if `a` is Fortran *contiguous* in memory, C-like order
169: (8)           otherwise.

170: (4)           Returns
171: (4)
172: (4)           -----
173: (8)           reshaped_array : ndarray
174: (8)           This will be a new view object if possible; otherwise, it will
175: (8)           be a copy. Note there is no guarantee of the *memory layout* (C- or
176: (8)           Fortran- contiguous) of the returned array.

177: (4)           See Also
178: (4)           -----
179: (4)           ndarray.reshape : Equivalent method.

180: (4)           Notes
181: (4)           -----
182: (4)           It is not always possible to change the shape of an array without copying
183: (4)           the data.
184: (4)           The `order` keyword gives the index ordering both for *fetching* the
185: (4)           values
186: (4)           from `a`, and then *placing* the values into the output array.
187: (4)           For example, let's say you have an array:
188: (4)           >>> a = np.arange(6).reshape((3, 2))
189: (4)           >>> a
190: (11)          array([[0, 1],
191: (4)           [2, 3],
192: (4)           [4, 5]])
193: (4)           You can think of reshaping as first raveling the array (using the given
194: (4)           index order), then inserting the elements from the raveled array into the
195: (4)           new array using the same kind of index ordering as was used for the
196: (4)           raveling.
197: (4)           >>> np.reshape(a, (2, 3)) # C-like index ordering
198: (4)           array([[0, 1, 2],
199: (4)           [3, 4, 5]])
200: (11)          >>> np.reshape(np.ravel(a), (2, 3)) # equivalent to C ravel then C reshape
201: (4)           array([[0, 1, 2],
202: (4)           [3, 4, 5]])
203: (11)          >>> np.reshape(a, (2, 3), order='F') # Fortran-like index ordering
204: (4)           array([[0, 4, 3],
205: (4)           [2, 1, 5]])
206: (11)          >>> np.reshape(np.ravel(a, order='F'), (2, 3), order='F')
207: (4)           array([[0, 4, 3],
208: (4)           [2, 1, 5]])
209: (4)           Examples
210: (4)
211: (4)           >>> a = np.array([[1,2,3], [4,5,6]])
212: (4)           >>> np.reshape(a, 6)
213: (4)           array([1, 2, 3, 4, 5, 6])
214: (4)           >>> np.reshape(a, 6, order='F')
215: (4)           array([1, 4, 2, 5, 3, 6])
216: (4)           >>> np.reshape(a, (3,-1))      # the unspecified value is inferred to be
217: (4)           array([[1, 2],
218: (4)           [3, 4],
219: (4)           [5, 6]])
219: (4)           """
219: (4)           return _wrapfunc(a, 'reshape', newshape, order=order)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

220: (0)
221: (4)     def _choose_dispatcher(a, choices, out=None, mode=None):
222: (4)         yield a
223: (4)         yield from choices
224: (4)         yield out
225: (0)     @array_function_dispatch(_choose_dispatcher)
226: (4)     def choose(a, choices, out=None, mode='raise'):
227: (4)         """
228: (4)             Construct an array from an index array and a list of arrays to choose
229: (4)             from.
230: (4)             First of all, if confused or uncertain, definitely look at the Examples -
231: (4)             in its full generality, this function is less simple than it might
232: (4)             seem from the following code description (below ndi =
233: (4)             `numpy.lib.index_tricks`):
234: (4)             ``np.choose(a,c) == np.array([c[a[I]][I] for I in
235: (4)             ndi.ndindex(a.shape)])``.
236: (4)             But this omits some subtleties. Here is a fully general summary:
237: (4)             Given an "index" array (`a`) of integers and a sequence of ``n`` arrays
238: (4)             (`choices`), `a` and each choice array are first broadcast, as necessary,
239: (4)             to arrays of a common shape; calling these *Ba* and *Bchoices[i], i =
240: (4)             0,...,n-1* we have that, necessarily, ``Ba.shape == Bchoices[i].shape``
241: (4)             for each ``i``. Then, a new array with shape ``Ba.shape`` is created as
242: (4)             follows:
243: (4)             * if ``mode='raise'`` (the default), then, first of all, each element of
244: (4)             ``a`` (and thus ``Ba``) must be in the range ``[0, n-1]``; now, suppose
245: (4)             that ``i`` (in that range) is the value at the ``(j0, j1, ..., jm)``
246: (4)             position in ``Ba`` - then the value at the same position
247: (4)             is the value in ``Bchoices[i]`` at that same position;
248: (4)             * if ``mode='wrap'``, values in `a` (and thus `Ba`) may be any (signed)
249: (4)             integer; modular arithmetic is used to map integers outside the range
250: (4)             `[0, n-1]` back into that range; and then the new array is constructed
251: (4)             as above;
252: (4)             * if ``mode='clip'``, values in `a` (and thus `Ba`) may be any (signed)
253: (4)             integer; negative integers are mapped to 0; values greater than ``n-1``
254: (4)             are mapped to ``n-1``; and then the new array is constructed as above.
255: (4)             Parameters
256: (4)             -----
257: (4)             a : int array
258: (4)                 This array must contain integers in ``[0, n-1]``, where ``n`` is the
259: (4)                 number of choices, unless ``mode=wrap`` or ``mode=clip``, in which
260: (4)                 cases any integers are permissible.
261: (4)             choices : sequence of arrays
262: (4)                 Choice arrays. `a` and all of the choices must be broadcastable to the
263: (4)                 same shape. If `choices` is itself an array (not recommended), then
264: (4)                 its outermost dimension (i.e., the one corresponding to
265: (4)                 ``choices.shape[0]``) is taken as defining the "sequence".
266: (4)             out : array, optional
267: (4)                 If provided, the result will be inserted into this array. It should
268: (4)                 be of the appropriate shape and dtype. Note that `out` is always
269: (4)                 buffered if ``mode='raise'``; use other modes for better performance.
270: (4)             mode : {'raise' (default), 'wrap', 'clip'}, optional
271: (4)                 Specifies how indices outside ``[0, n-1]`` will be treated:
272: (4)                     * 'raise' : an exception is raised
273: (4)                     * 'wrap' : value becomes value mod ``n``
274: (4)                     * 'clip' : values < 0 are mapped to 0, values > n-1 are mapped to n-
275: (4)                     1
276: (4)             Returns
277: (4)             -----
278: (4)             merged_array : array
279: (4)                 The merged result.
280: (4)             Raises
281: (4)             -----
282: (4)             ValueError: shape mismatch
283: (4)                 If `a` and each choice array are not all broadcastable to the same
284: (4)                 shape.
285: (4)             See Also
286: (4)             -----
287: (4)             ndarray.choose : equivalent method
288: (4)             numpy.take_along_axis : Preferable if `choices` is an array

```

```

285: (4)          Notes
286: (4)          -----
287: (4)          To reduce the chance of misinterpretation, even though the following
288: (4)          "abuse" is nominally supported, `choices` should neither be, nor be
289: (4)          thought of as, a single array, i.e., the outermost sequence-like container
290: (4)          should be either a list or a tuple.
291: (4)          Examples
292: (4)          -----
293: (4)          >>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
294: (4)          ... [20, 21, 22, 23], [30, 31, 32, 33]]
295: (4)          >>> np.choose([2, 3, 1, 0], choices
296: (4)          ... # the first element of the result will be the first element of the
297: (4)          ... # third (2+1) "array" in choices, namely, 20; the second element
298: (4)          ... # will be the second element of the fourth (3+1) choice array, i.e.,
299: (4)          ... # 31, etc.
300: (4)          ...
301: (4)          array([20, 31, 12, 3])
302: (4)          >>> np.choose([2, 4, 1, 0], choices, mode='clip') # 4 goes to 3 (4-1)
303: (4)          array([20, 31, 12, 3])
304: (4)          >>> # because there are 4 choice arrays
305: (4)          >>> np.choose([2, 4, 1, 0], choices, mode='wrap') # 4 goes to (4 mod 4)
306: (4)          array([20, 1, 12, 3])
307: (4)          >>> # i.e., 0
308: (4)          A couple examples illustrating how choose broadcasts:
309: (4)          >>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
310: (4)          >>> choices = [-10, 10]
311: (4)          >>> np.choose(a, choices)
312: (4)          array([[ 10, -10,  10],
313: (11)          ... [-10,  10, -10],
314: (11)          ... [ 10, -10,  10]])
315: (4)          >>> # With thanks to Anne Archibald
316: (4)          >>> a = np.array([0, 1]).reshape((2,1,1))
317: (4)          >>> c1 = np.array([1, 2, 3]).reshape((1,3,1))
318: (4)          >>> c2 = np.array([-1, -2, -3, -4, -5]).reshape((1,1,5))
319: (4)          >>> np.choose(a, (c1, c2)) # result is 2x3x5, res[0,:,:]=c1, res[1,:,:]=c2
320: (4)          array([[[ 1,  1,  1,  1,  1],
321: (12)          ... [ 2,  2,  2,  2,  2],
322: (12)          ... [ 3,  3,  3,  3,  3]],
323: (11)          ... [[-1, -2, -3, -4, -5],
324: (12)          ... [-1, -2, -3, -4, -5],
325: (12)          ... [-1, -2, -3, -4, -5]]])
326: (4)          """
327: (4)          return _wrapfunc(a, 'choose', choices, out=out, mode=mode)
328: (0)          def _repeat_dispatcher(a, repeats, axis=None):
329: (4)          return (a,)
330: (0)          @array_function_dispatch(_repeat_dispatcher)
331: (0)          def repeat(a, repeats, axis=None):
332: (4)          """
333: (4)          Repeat each element of an array after themselves
334: (4)          Parameters
335: (4)          -----
336: (4)          a : array_like
337: (8)          Input array.
338: (4)          repeats : int or array of ints
339: (8)          The number of repetitions for each element. `repeats` is broadcasted
340: (8)          to fit the shape of the given axis.
341: (4)          axis : int, optional
342: (8)          The axis along which to repeat values. By default, use the
343: (8)          flattened input array, and return a flat output array.
344: (4)          Returns
345: (4)          -----
346: (4)          repeated_array : ndarray
347: (8)          Output array which has the same shape as `a`, except along
348: (8)          the given axis.
349: (4)          See Also
350: (4)          -----
351: (4)          tile : Tile an array.
352: (4)          unique : Find the unique elements of an array.
353: (4)          Examples

```

```

354: (4)      -----
355: (4)      >>> np.repeat(3, 4)
356: (4)      array([3, 3, 3, 3])
357: (4)      >>> x = np.array([[1,2],[3,4]])
358: (4)      >>> np.repeat(x, 2)
359: (4)      array([1, 1, 2, 2, 3, 3, 4, 4])
360: (4)      >>> np.repeat(x, 3, axis=1)
361: (4)      array([[1, 1, 1, 2, 2, 2],
362: (11)      [3, 3, 3, 4, 4, 4]])
363: (4)      >>> np.repeat(x, [1, 2], axis=0)
364: (4)      array([[1, 2],
365: (11)      [3, 4],
366: (11)      [3, 4]])
367: (4)      """
368: (4)      return _wrapfunc(a, 'repeat', repeats, axis=axis)
369: (0)      def _put_dispatcher(a, ind, v, mode=None):
370: (4)          return (a, ind, v)
371: (0)      @array_function_dispatch(_put_dispatcher)
372: (0)      def put(a, ind, v, mode='raise'):
373: (4)          """
374: (4)          Replaces specified elements of an array with given values.
375: (4)          The indexing works on the flattened target array. `put` is roughly
376: (4)          equivalent to:
377: (4)          ::
378: (8)              a.flat[ind] = v
379: (4)      Parameters
380: (4)      -----
381: (4)      a : ndarray
382: (8)          Target array.
383: (4)      ind : array_like
384: (8)          Target indices, interpreted as integers.
385: (4)      v : array_like
386: (8)          Values to place in `a` at target indices. If `v` is shorter than
387: (8)          `ind` it will be repeated as necessary.
388: (4)      mode : {'raise', 'wrap', 'clip'}, optional
389: (8)          Specifies how out-of-bounds indices will behave.
390: (8)          * 'raise' -- raise an error (default)
391: (8)          * 'wrap' -- wrap around
392: (8)          * 'clip' -- clip to the range
393: (8)          'clip' mode means that all indices that are too large are replaced
394: (8)          by the index that addresses the last element along that axis. Note
395: (8)          that this disables indexing with negative numbers. In 'raise' mode,
396: (8)          if an exception occurs the target array may still be modified.
397: (4)      See Also
398: (4)      -----
399: (4)      putmask, place
400: (4)      put_along_axis : Put elements by matching the array and the index arrays
401: (4)      Examples
402: (4)      -----
403: (4)      >>> a = np.arange(5)
404: (4)      >>> np.put(a, [0, 2], [-44, -55])
405: (4)      >>> a
406: (4)      array([-44, 1, -55, 3, 4])
407: (4)      >>> a = np.arange(5)
408: (4)      >>> np.put(a, 22, -5, mode='clip')
409: (4)      >>> a
410: (4)      array([ 0, 1, 2, 3, -5])
411: (4)      """
412: (4)      try:
413: (8)          put = a.put
414: (4)      except AttributeError as e:
415: (8)          raise TypeError("argument 1 must be numpy.ndarray, "
416: (24)                      "not {name}".format(name=type(a).__name__)) from e
417: (4)      return put(ind, v, mode=mode)
418: (0)      def _swapaxes_dispatcher(a, axis1, axis2):
419: (4)          return (a,)
420: (0)      @array_function_dispatch(_swapaxes_dispatcher)
421: (0)      def swapaxes(a, axis1, axis2):
422: (4)          """

```

```

423: (4)           Interchange two axes of an array.
424: (4)           Parameters
425: (4)           -----
426: (4)           a : array_like
427: (8)           Input array.
428: (4)           axis1 : int
429: (8)           First axis.
430: (4)           axis2 : int
431: (8)           Second axis.
432: (4)           Returns
433: (4)           -----
434: (4)           a_swapped : ndarray
435: (8)           For NumPy >= 1.10.0, if `a` is an ndarray, then a view of `a` is
436: (8)           returned; otherwise a new array is created. For earlier NumPy
437: (8)           versions a view of `a` is returned only if the order of the
438: (8)           axes is changed, otherwise the input array is returned.
439: (4)           Examples
440: (4)           -----
441: (4)           >>> x = np.array([[1,2,3]])
442: (4)           >>> np.swapaxes(x,0,1)
443: (4)           array([[1],
444: (11)          [2],
445: (11)          [3]])
446: (4)           >>> x = np.array([[[0,1],[2,3]],[[4,5],[6,7]]])
447: (4)           >>> x
448: (4)           array([[[0, 1],
449: (12)          [2, 3]],
450: (11)          [[4, 5],
451: (12)          [6, 7]]])
452: (4)           >>> np.swapaxes(x,0,2)
453: (4)           array([[[0, 4],
454: (12)          [2, 6]],
455: (11)          [[1, 5],
456: (12)          [3, 7]]])
457: (4)           """
458: (4)           return _wrapfunc(a, 'swapaxes', axis1, axis2)
459: (0)           def _transpose_dispatcher(a, axes=None):
460: (4)           return (a,)
461: (0)           @array_function_dispatch(_transpose_dispatcher)
462: (0)           def transpose(a, axes=None):
463: (4)           """
464: (4)           Returns an array with axes transposed.
465: (4)           For a 1-D array, this returns an unchanged view of the original array, as
a
466: (4)           transposed vector is simply the same vector.
467: (4)           To convert a 1-D array into a 2-D column vector, an additional dimension
468: (4)           must be added, e.g., ``np.atleast2d(a).T`` achieves this, as does
469: (4)           ``a[:, np.newaxis]``.
470: (4)           For a 2-D array, this is the standard matrix transpose.
471: (4)           For an n-D array, if axes are given, their order indicates how the
472: (4)           axes are permuted (see Examples). If axes are not provided, then
473: (4)           ``transpose(a).shape == a.shape[::-1]``.
474: (4)           Parameters
475: (4)           -----
476: (4)           a : array_like
477: (8)           Input array.
478: (4)           axes : tuple or list of ints, optional
479: (8)           If specified, it must be a tuple or list which contains a permutation
480: (8)           of [0,1,...,N-1] where N is the number of axes of `a`. The `i`'th axis
481: (8)           of the returned array will correspond to the axis numbered ``axes[i]``
482: (8)           of the input. If not specified, defaults to ``range(a.ndim)[::-1]``,
483: (8)           which reverses the order of the axes.
484: (4)           Returns
485: (4)           -----
486: (4)           p : ndarray
487: (8)           `a` with its axes permuted. A view is returned whenever possible.
488: (4)           See Also
489: (4)           -----
490: (4)           ndarray.transpose : Equivalent method.

```

```

491: (4) moveaxis : Move axes of an array to new positions.
492: (4) argsort : Return the indices that would sort an array.
493: (4) Notes -----
495: (4) Use ``transpose(a, argsort(axes))`` to invert the transposition of tensors
496: (4) when using the `axes` keyword argument.
497: (4) Examples -----
499: (4) >>> a = np.array([[1, 2], [3, 4]])
500: (4) >>> a
501: (4) array([[1, 2],
502: (11)      [3, 4]])
503: (4) >>> np.transpose(a)
504: (4) array([[1, 3],
505: (11)      [2, 4]])
506: (4) >>> a = np.array([1, 2, 3, 4])
507: (4) >>> a
508: (4) array([1, 2, 3, 4])
509: (4) >>> np.transpose(a)
510: (4) array([1, 2, 3, 4])
511: (4) >>> a = np.ones((1, 2, 3))
512: (4) >>> np.transpose(a, (1, 0, 2)).shape
513: (4) (2, 1, 3)
514: (4) >>> a = np.ones((2, 3, 4, 5))
515: (4) >>> np.transpose(a).shape
516: (4) (5, 4, 3, 2)
517: (4) """
518: (4)     return _wrapfunc(a, 'transpose', axes)
519: (0) def _partition_dispatcher(a, kth, axis=None, kind=None, order=None):
520: (4)     return (a,)
521: (0) @array_function_dispatch(_partition_dispatcher)
522: (0) def partition(a, kth, axis=-1, kind='introselect', order=None):
523: (4) """
524: (4)     Return a partitioned copy of an array.
525: (4)     Creates a copy of the array with its elements rearranged in such a
526: (4)     way that the value of the element in k-th position is in the position
527: (4)     the value would be in a sorted array. In the partitioned array, all
528: (4)     elements before the k-th element are less than or equal to that
529: (4)     element, and all the elements after the k-th element are greater than
530: (4)     or equal to that element. The ordering of the elements in the two
531: (4)     partitions is undefined.
532: (4) .. versionadded:: 1.8.0
533: (4) Parameters -----
534: (4) a : array_like
535: (4)     Array to be sorted.
536: (8) kth : int or sequence of ints
537: (4)     Element index to partition by. The k-th value of the element
538: (8)     will be in its final sorted position and all smaller elements
539: (8)     will be moved before it and all equal or greater elements behind
540: (8)     it. The order of all elements in the partitions is undefined. If
541: (8)     provided with a sequence of k-th it will partition all elements
542: (8)     indexed by k-th of them into their sorted position at once.
543: (8) .. deprecated:: 1.22.0
544: (8)     Passing booleans as index is deprecated.
545: (12) axis : int or None, optional
546: (4)     Axis along which to sort. If None, the array is flattened before
547: (8)     sorting. The default is -1, which sorts along the last axis.
548: (8) kind : {'introselect'}, optional
549: (4)     Selection algorithm. Default is 'introselect'.
550: (8) order : str or list of str, optional
551: (4)     When `a` is an array with fields defined, this argument
552: (8)     specifies which fields to compare first, second, etc. A single
553: (8)     field can be specified as a string. Not all fields need be
554: (8)     specified, but unspecified fields will still be used, in the
555: (8)     order in which they come up in the dtype, to break ties.
556: (8) Returns -----
557: (4)     partitioned_array : ndarray

```

```

560: (8)                                Array of the same type and shape as `a` .
561: (4)                                See Also
562: (4)
563: (4)                                ndarray.partition : Method to sort an array in-place.
564: (4)                                argpartition : Indirect partition.
565: (4)                                sort : Full sorting
566: (4)                                Notes
567: (4)
568: (4)                                The various selection algorithms are characterized by their average
569: (4)                                speed, worst case performance, work space size, and whether they are
570: (4)                                stable. A stable sort keeps items with the same key in the same
571: (4)                                relative order. The available algorithms have the following
572: (4)                                properties:
573: (4)                                ===== ===== ===== ===== ===== =====
574: (7)          kind      speed   worst case   work space   stable
575: (4)
576: (4)          'introselect'    1        O(n)        0        no
577: (4)
578: (4)                                All the partition algorithms make temporary copies of the data when
579: (4)                                partitioning along any but the last axis. Consequently,
580: (4)                                partitioning along the last axis is faster and uses less space than
581: (4)                                partitioning along any other axis.
582: (4)                                The sort order for complex numbers is lexicographic. If both the
583: (4)                                real and imaginary parts are non-nan then the order is determined by
584: (4)                                the real parts except when they are equal, in which case the order
585: (4)                                is determined by the imaginary parts.
586: (4)                                Examples
587: (4)
588: (4)                                >>> a = np.array([7, 1, 7, 7, 1, 5, 7, 2, 3, 2, 6, 2, 3, 0])
589: (4)                                >>> p = np.partition(a, 4)
590: (4)                                >>> p
591: (4)                                array([0, 1, 2, 1, 2, 5, 2, 3, 3, 6, 7, 7, 7, 7])
592: (4)                                ``p[4]`` is 2; all elements in ``p[:4]`` are less than or equal
593: (4)                                to ``p[4]``, and all elements in ``p[5:]`` are greater than or
594: (4)                                equal to ``p[4]``. The partition is::
595: (8)                                [0, 1, 2, 1], [2], [5, 2, 3, 3, 6, 7, 7, 7, 7]
596: (4)                                The next example shows the use of multiple values passed to `kth` .
597: (4)                                >>> p2 = np.partition(a, (4, 8))
598: (4)                                >>> p2
599: (4)                                array([0, 1, 2, 1, 2, 3, 3, 2, 5, 6, 7, 7, 7, 7])
600: (4)                                ``p2[4]`` is 2 and ``p2[8]`` is 5. All elements in ``p2[:4]``
601: (4)                                are less than or equal to ``p2[4]``, all elements in ``p2[5:8]``
602: (4)                                are greater than or equal to ``p2[4]`` and less than or equal to
603: (4)                                ``p2[8]``, and all elements in ``p2[9:]`` are greater than or
604: (4)                                equal to ``p2[8]``. The partition is::
605: (8)                                [0, 1, 2, 1], [2], [3, 3, 2], [5], [6, 7, 7, 7, 7]
606: (4)                                """
607: (4)                                if axis is None:
608: (8)                                    a = asanyarray(a).flatten()
609: (8)                                    axis = -1
610: (4)                                else:
611: (8)                                    a = asanyarray(a).copy(order="K")
612: (4)                                    a.partition(kth, axis=axis, kind=kind, order=order)
613: (4)                                    return a
614: (0)                                def _argpartition_dispatcher(a, kth, axis=None, kind=None, order=None):
615: (4)                                    return (a,)
616: (0)                                @array_function_dispatch(_argpartition_dispatcher)
617: (0)                                def argpartition(a, kth, axis=-1, kind='introselect', order=None):
618: (4)                                    """
619: (4)                                    Perform an indirect partition along the given axis using the
620: (4)                                    algorithm specified by the `kind` keyword. It returns an array of
621: (4)                                    indices of the same shape as `a` that index data along the given
622: (4)                                    axis in partitioned order.
623: (4)                                    .. versionadded:: 1.8.0
624: (4)                                    Parameters
625: (4)
626: (4)                                    a : array_like
627: (8)                                    Array to sort.
628: (4)                                    kth : int or sequence of ints

```

```

629: (8)           Element index to partition by. The k-th element will be in its
630: (8)           final sorted position and all smaller elements will be moved
631: (8)           before it and all larger elements behind it. The order of all
632: (8)           elements in the partitions is undefined. If provided with a
633: (8)           sequence of k-th it will partition all of them into their sorted
634: (8)           position at once.
635: (8)           .. deprecated:: 1.22.0
636: (12)          Passing booleans as index is deprecated.
637: (4)           axis : int or None, optional
638: (8)           Axis along which to sort. The default is -1 (the last axis). If
639: (8)           None, the flattened array is used.
640: (4)           kind : {'introselect'}, optional
641: (8)           Selection algorithm. Default is 'introselect'
642: (4)           order : str or list of str, optional
643: (8)           When `a` is an array with fields defined, this argument
644: (8)           specifies which fields to compare first, second, etc. A single
645: (8)           field can be specified as a string, and not all fields need be
646: (8)           specified, but unspecified fields will still be used, in the
647: (8)           order in which they come up in the dtype, to break ties.
648: (4)           Returns
649: (4)           -----
650: (4)           index_array : ndarray, int
651: (8)           Array of indices that partition `a` along the specified axis.
652: (8)           If `a` is one-dimensional, ``a[index_array]`` yields a partitioned
`a`.
653: (8)           More generally, ``np.take_along_axis(a, index_array, axis=axis)``
654: (8)           always yields the partitioned `a`, irrespective of dimensionality.
655: (4)           See Also
656: (4)           -----
657: (4)           partition : Describes partition algorithms used.
658: (4)           ndarray.partition : Inplace partition.
659: (4)           argsort : Full indirect sort.
660: (4)           take_along_axis : Apply ``index_array`` from argpartition
661: (22)          to an array as if by calling partition.
662: (4)           Notes
663: (4)           -----
664: (4)           See `partition` for notes on the different selection algorithms.
665: (4)           Examples
666: (4)           -----
667: (4)           One dimensional array:
668: (4)           >>> x = np.array([3, 4, 2, 1])
669: (4)           >>> x[np.argpartition(x, 3)]
670: (4)           array([2, 1, 3, 4])
671: (4)           >>> x[np.argpartition(x, (1, 3))]
672: (4)           array([1, 2, 3, 4])
673: (4)           >>> x = [3, 4, 2, 1]
674: (4)           >>> np.array(x)[np.argpartition(x, 3)]
675: (4)           array([2, 1, 3, 4])
676: (4)           Multi-dimensional array:
677: (4)           >>> x = np.array([[3, 4, 2], [1, 3, 1]])
678: (4)           >>> index_array = np.argpartition(x, kth=1, axis=-1)
679: (4)           >>> np.take_along_axis(x, index_array, axis=-1) # same as np.partition(x,
kth=1)
680: (4)           array([[2, 3, 4],
681: (11)             [1, 1, 3]])
682: (4)           """
683: (4)           return _wrapfunc(a, 'argpartition', kth, axis=axis, kind=kind,
order=order)
684: (0)           def _sort_dispatcher(a, axis=None, kind=None, order=None):
685: (4)               return (a,)
686: (0)           @array_function_dispatch(_sort_dispatcher)
687: (0)           def sort(a, axis=-1, kind=None, order=None):
688: (4)               """
689: (4)               Return a sorted copy of an array.
690: (4)               Parameters
691: (4)               -----
692: (4)               a : array_like
693: (8)                   Array to be sorted.
694: (4)               axis : int or None, optional

```

```

695: (8) Axis along which to sort. If None, the array is flattened before
696: (8) sorting. The default is -1, which sorts along the last axis.
697: (4) kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
698: (8) Sorting algorithm. The default is 'quicksort'. Note that both 'stable'
699: (8) and 'mergesort' use timsort or radix sort under the covers and, in
general,
700: (8) the actual implementation will vary with data type. The 'mergesort'
option
701: (8) is retained for backwards compatibility.
702: (8) .. versionchanged:: 1.15.0.
703: (11) The 'stable' option was added.
704: (4) order : str or list of str, optional
705: (8) When `a` is an array with fields defined, this argument specifies
706: (8) which fields to compare first, second, etc. A single field can
707: (8) be specified as a string, and not all fields need be specified,
708: (8) but unspecified fields will still be used, in the order in which
709: (8) they come up in the dtype, to break ties.
710: (4) Returns
711: (4)
712: (4) -----
713: (8) sorted_array : ndarray
714: (4)     Array of the same type and shape as `a`.
See Also
715: (4) -----
716: (4) ndarray.sort : Method to sort an array in-place.
717: (4) argsort : Indirect sort.
718: (4) lexsort : Indirect stable sort on multiple keys.
719: (4) searchsorted : Find elements in a sorted array.
720: (4) partition : Partial sort.
721: (4) Notes
722: (4) -----
723: (4) The various sorting algorithms are characterized by their average speed,
724: (4) worst case performance, work space size, and whether they are stable. A
725: (4) stable sort keeps items with the same key in the same relative
726: (4) order. The four algorithms implemented in NumPy have the following
727: (4) properties:
728: (4) ====== ====== ====== ====== ====== ======
729: (7) kind      speed    worst case   work space   stable
730: (4) ====== ====== ====== ====== ====== ======
731: (4) 'quicksort'   1      O(n^2)          0        no
732: (4) 'heapsort'    3      O(n*log(n))    0        no
733: (4) 'mergesort'   2      O(n*log(n))  ~n/2      yes
734: (4) 'timsort'     2      O(n*log(n))  ~n/2      yes
735: (4) ====== ====== ====== ====== ====== ======
736: (4) .. note:: The datatype determines which of 'mergesort' or 'timsort'
737: (7) is actually used, even if 'mergesort' is specified. User selection
738: (7) at a finer scale is not currently available.
All the sort algorithms make temporary copies of the data when
739: (4) sorting along any but the last axis. Consequently, sorting along
740: (4) the last axis is faster and uses less space than sorting along
741: (4) any other axis.
742: (4) The sort order for complex numbers is lexicographic. If both the real
743: (4) and imaginary parts are non-nan then the order is determined by the
744: (4) real parts except when they are equal, in which case the order is
745: (4) determined by the imaginary parts.
746: (4) Previous to numpy 1.4.0 sorting real and complex arrays containing nan
747: (4) values led to undefined behaviour. In numpy versions >= 1.4.0 nan
748: (4) values are sorted to the end. The extended sort order is:
749: (4)     * Real: [R, nan]
750: (6)     * Complex: [R + Rj, R + nanj, nan + Rj, nan + nanj]
751: (6) where R is a non-nan real value. Complex values with the same nan
752: (4) placements are sorted according to the non-nan part if it exists.
753: (4) Non-nan values are sorted as before.
754: (4) .. versionadded:: 1.12.0
755: (4) quicksort has been changed to `introsort
<https://en.wikipedia.org/wiki/Introsort>`_.
756: (4) When sorting does not make enough progress it switches to
757: (4) `heapsort <https://en.wikipedia.org/wiki/Heapsort>`_.
758: (4) This implementation makes quicksort O(n*log(n)) in the worst case.
759: (4) 'stable' automatically chooses the best stable sorting algorithm
760: (4)

```

```

761: (4)           for the data type being sorted.
762: (4)           It, along with 'mergesort' is currently mapped to
763: (4)           `timsort <https://en.wikipedia.org/wiki/Timsort>`_
764: (4)           or `radix sort <https://en.wikipedia.org/wiki/Radix\_sort>`_
765: (4)           depending on the data type.
766: (4)           API forward compatibility currently limits the
767: (4)           ability to select the implementation and it is hardwired for the different
768: (4)           data types.
769: (4)           .. versionadded:: 1.17.0
770: (4)           Timsort is added for better performance on already or nearly
771: (4)           sorted data. On random data timsort is almost identical to
772: (4)           mergesort. It is now used for stable sort while quicksort is still the
773: (4)           default sort if none is chosen. For timsort details, refer to
774: (4)           `CPython listsort.txt
<https://github.com/python/cpython/blob/3.7/Objects/listsort.txt>`.
775: (4)           'mergesort' and 'stable' are mapped to radix sort for integer data types.

Radix sort is an
776: (4)           O(n) sort instead of O(n log n).
777: (4)           .. versionchanged:: 1.18.0
778: (4)           NaT now sorts to the end of arrays for consistency with NaN.
779: (4)           Examples
780: (4)           -----
781: (4)           >>> a = np.array([[1,4],[3,1]])
782: (4)           >>> np.sort(a)                      # sort along the last axis
783: (4)           array([[1, 4],
784: (11)             [1, 3]])
785: (4)           >>> np.sort(a, axis=None)      # sort the flattened array
786: (4)           array([1, 1, 3, 4])
787: (4)           >>> np.sort(a, axis=0)        # sort along the first axis
788: (4)           array([[1, 1],
789: (11)             [3, 4]])
790: (4)           Use the `order` keyword to specify a field to use when sorting a
791: (4)           structured array:
792: (4)           >>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
793: (4)           >>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38),
794: (4)                         ...          ('Galahad', 1.7, 38)]
795: (4)           >>> a = np.array(values, dtype=dtype)      # create a structured array
796: (4)           >>> np.sort(a, order='height')            # doctest: +SKIP
797: (4)           array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
798: (11)               ('Lancelot', 1.8999999999999999, 38)],
799: (10)                 dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
800: (4)           Sort by age, then height if ages are equal:
801: (4)           >>> np.sort(a, order=['age', 'height'])       # doctest: +SKIP
802: (4)           array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38),
803: (11)                     ('Arthur', 1.8, 41)],
804: (10)                   dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
805: (4)           """
806: (4)           if axis is None:
807: (8)               a = asanyarray(a).flatten()
808: (8)               axis = -1
809: (4)           else:
810: (8)               a = asanyarray(a).copy(order="K")
811: (4)               a.sort(axis=axis, kind=kind, order=order)
812: (4)               return a
813: (0)           def _argsort_dispatcher(a, axis=None, kind=None, order=None):
814: (4)               return (a,)
815: (0)           @array_function_dispatch(_argsort_dispatcher)
816: (0)           def argsort(a, axis=-1, kind=None, order=None):
817: (4)               """
818: (4)               Returns the indices that would sort an array.
819: (4)               Perform an indirect sort along the given axis using the algorithm
specified
820: (4)               by the `kind` keyword. It returns an array of indices of the same shape as
821: (4)               `a` that index data along the given axis in sorted order.
822: (4)               Parameters
823: (4)               -----
824: (4)               a : array_like
825: (8)                   Array to sort.
826: (4)               axis : int or None, optional

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

827: (8) Axis along which to sort. The default is -1 (the last axis). If None,
828: (8) the flattened array is used.
829: (4) kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
830: (8) Sorting algorithm. The default is 'quicksort'. Note that both 'stable'
831: (8) and 'mergesort' use timsort under the covers and, in general, the
832: (8) actual implementation will vary with data type. The 'mergesort' option
833: (8) is retained for backwards compatibility.
834: (8) .. versionchanged:: 1.15.0.
835: (11) The 'stable' option was added.
836: (4) order : str or list of str, optional
837: (8) When `a` is an array with fields defined, this argument specifies
838: (8) which fields to compare first, second, etc. A single field can
839: (8) be specified as a string, and not all fields need be specified,
840: (8) but unspecified fields will still be used, in the order in which
841: (8) they come up in the dtype, to break ties.
842: (4) Returns
843: (4)
844: (4) index_array : ndarray, int
845: (8) Array of indices that sort `a` along the specified `axis`.
846: (8) If `a` is one-dimensional, ``a[index_array]`` yields a sorted `a`.
847: (8) More generally, ``np.take_along_axis(a, index_array, axis=axis)``
848: (8) always yields the sorted `a`, irrespective of dimensionality.
849: (4) See Also
850: (4)
851: (4) sort : Describes sorting algorithms used.
852: (4) lexsort : Indirect stable sort with multiple keys.
853: (4) ndarray.sort : Inplace sort.
854: (4) argpartition : Indirect partial sort.
855: (4) take_along_axis : Apply ``index_array`` from argsort
856: (22) to an array as if by calling sort.
857: (4) Notes
858: (4)
859: (4) See `sort` for notes on the different sorting algorithms.
860: (4) As of NumPy 1.4.0 `argsort` works with real/complex arrays containing
861: (4) nan values. The enhanced sort order is documented in `sort`.
862: (4) Examples
863: (4)
864: (4) One dimensional array:
865: (4) >>> x = np.array([3, 1, 2])
866: (4) >>> np.argsort(x)
867: (4) array([1, 2, 0])
868: (4) Two-dimensional array:
869: (4) >>> x = np.array([[0, 3], [2, 2]])
870: (4) >>> x
871: (4) array([[0, 3],
872: (11)           [2, 2]])
873: (4) >>> ind = np.argsort(x, axis=0) # sorts along first axis (down)
874: (4) >>> ind
875: (4) array([[0, 1],
876: (11)           [1, 0]])
877: (4) >>> np.take_along_axis(x, ind, axis=0) # same as np.sort(x, axis=0)
878: (4) array([[0, 2],
879: (11)           [2, 3]])
880: (4) >>> ind = np.argsort(x, axis=1) # sorts along last axis (across)
881: (4) >>> ind
882: (4) array([[0, 1],
883: (11)           [0, 1]])
884: (4) >>> np.take_along_axis(x, ind, axis=1) # same as np.sort(x, axis=1)
885: (4) array([[0, 3],
886: (11)           [2, 2]])
887: (4) Indices of the sorted elements of a N-dimensional array:
888: (4) >>> ind = np.unravel_index(np.argsort(x, axis=None), x.shape)
889: (4) >>> ind
890: (4) (array([0, 1, 1, 0]), array([0, 0, 1, 1]))
891: (4) >>> x[ind] # same as np.sort(x, axis=None)
892: (4) array([0, 2, 2, 3])
893: (4) Sorting with keys:
894: (4) >>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
895: (4) >>> x

```

```

896: (4)
897: (10)
898: (4)
899: (4)
900: (4)
901: (4)
902: (4)
903: (4)
904: (0)
905: (4)
906: (0)
907: (0)
908: (4)
909: (4)
910: (4)
911: (4)
912: (4)
913: (8)
914: (4)
915: (8)
916: (8)
917: (4)
918: (8)
919: (8)
920: (4)
921: (8)
922: (8)
923: (8)
924: (8)
925: (4)
926: (4)
927: (4)
928: (8)
929: (8)
930: (8)
931: (8)
932: (4)
933: (4)
934: (4)
935: (4)
936: (4)
937: (4)
938: (22)
939: (4)
940: (4)
941: (4)
942: (4)
943: (4)
944: (4)
945: (4)
946: (4)
947: (4)
948: (11)
949: (4)
950: (4)
951: (4)
952: (4)
953: (4)
954: (4)
955: (4)
956: (4)
957: (4)
958: (4)
959: (4)
960: (4)
961: (4)
962: (4)
963: (4)
964: (4)

        array([(1, 0), (0, 1)],
               dtype=[('x', '<i4'), ('y', '<i4')])
    >>> np.argsort(x, order=('x','y'))
    array([1, 0])
    >>> np.argsort(x, order=('y','x'))
    array([0, 1])
    """
    return _wrapfunc(a, 'argsort', axis=axis, kind=kind, order=order)
def _argmax_dispatcher(a, axis=None, out=None, *, keepdims=np._NoValue):
    return (a, out)
@array_function_dispatch(_argmax_dispatcher)
def argmax(a, axis=None, out=None, *, keepdims=np._NoValue):
    """
    Returns the indices of the maximum values along an axis.

    Parameters
    -----
    a : array_like
        Input array.
    axis : int, optional
        By default, the index is into the flattened array, otherwise
        along the specified axis.
    out : array, optional
        If provided, the result will be inserted into this array. It should
        be of the appropriate shape and dtype.
    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the array.
        .. versionadded:: 1.22.0
    Returns
    -----
    index_array : ndarray of ints
        Array of indices into the array. It has the same shape as `a.shape`
        with the dimension along `axis` removed. If `keepdims` is set to True,
        then the size of `axis` will be 1 with the resulting array having same
        shape as `a.shape`.
    See Also
    -----
    ndarray.argmax, argmin
    amax : The maximum value along a given axis.
    unravel_index : Convert a flat index into an index tuple.
    take_along_axis : Apply ``np.expand_dims(index_array, axis)``
                      from argmax to an array as if by calling max.
    Notes
    -----
    In case of multiple occurrences of the maximum values, the indices
    corresponding to the first occurrence are returned.
    Examples
    -----
    >>> a = np.arange(6).reshape(2,3) + 10
    >>> a
    array([[10, 11, 12],
           [13, 14, 15]])
    >>> np.argmax(a)
    5
    >>> np.argmax(a, axis=0)
    array([1, 1, 1])
    >>> np.argmax(a, axis=1)
    array([2, 2])
    Indexes of the maximal elements of a N-dimensional array:
    >>> ind = np.unravel_index(np.argmax(a, axis=None), a.shape)
    >>> ind
    (1, 2)
    >>> a[ind]
    15
    >>> b = np.arange(6)
    >>> b[1] = 5
    >>> b
    array([0, 5, 2, 3, 4, 5])

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

965: (4)             >>> np.argmax(b) # Only the first occurrence is returned.
966: (4)             1
967: (4)             >>> x = np.array([[4,2,3], [1,0,3]])
968: (4)             >>> index_array = np.argmax(x, axis=-1)
969: (4)             >>> # Same as np.amax(x, axis=-1, keepdims=True)
970: (4)             >>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1)
971: (4)             array([[4],
972: (11)             [3]])
973: (4)             >>> # Same as np.amax(x, axis=-1)
974: (4)             >>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1),
axis=-1).squeeze(axis=-1)
975: (4)             array([4, 3])
976: (4)             Setting `keepdims` to `True`,
977: (4)             >>> x = np.arange(24).reshape((2, 3, 4))
978: (4)             >>> res = np.argmax(x, axis=1, keepdims=True)
979: (4)             >>> res.shape
980: (4)             (2, 1, 4)
981: (4)             """
982: (4)             kwds = {'keepdims': keepdims} if keepdims is not np._NoValue else {}
983: (4)             return _wrapfunc(a, 'argmax', axis=axis, out=out, **kwds)
984: (0)             def _argmin_dispatcher(a, axis=None, out=None, *, keepdims=np._NoValue):
985: (4)                 return (a, out)
986: (0)             @array_function_dispatch(_argmin_dispatcher)
987: (0)             def argmin(a, axis=None, out=None, *, keepdims=np._NoValue):
988: (4)                 """
989: (4)                 Returns the indices of the minimum values along an axis.
990: (4)                 Parameters
991: (4)                 -----
992: (4)                 a : array_like
993: (8)                   Input array.
994: (4)                 axis : int, optional
995: (8)                   By default, the index is into the flattened array, otherwise
996: (8)                   along the specified axis.
997: (4)                 out : array, optional
998: (8)                   If provided, the result will be inserted into this array. It should
999: (8)                   be of the appropriate shape and dtype.
1000: (4)                 keepdims : bool, optional
1001: (8)                   If this is set to True, the axes which are reduced are left
1002: (8)                   in the result as dimensions with size one. With this option,
1003: (8)                   the result will broadcast correctly against the array.
1004: (8)                   .. versionadded:: 1.22.0
1005: (4)             Returns
1006: (4)             -----
1007: (4)             index_array : ndarray of ints
1008: (8)               Array of indices into the array. It has the same shape as `a.shape`
1009: (8)               with the dimension along `axis` removed. If `keepdims` is set to True,
1010: (8)               then the size of `axis` will be 1 with the resulting array having same
1011: (8)               shape as `a.shape`.
1012: (4)             See Also
1013: (4)             -----
1014: (4)             ndarray.argmin, argmax
1015: (4)             amin : The minimum value along a given axis.
1016: (4)             unravel_index : Convert a flat index into an index tuple.
1017: (4)             take_along_axis : Apply ``np.expand_dims(index_array, axis)``
1018: (22)               from argmin to an array as if by calling min.
1019: (4)             Notes
1020: (4)             -----
1021: (4)             In case of multiple occurrences of the minimum values, the indices
1022: (4)             corresponding to the first occurrence are returned.
1023: (4)             Examples
1024: (4)             -----
1025: (4)             >>> a = np.arange(6).reshape(2,3) + 10
1026: (4)             >>> a
1027: (4)             array([[10, 11, 12],
1028: (11)             [13, 14, 15]])
1029: (4)             >>> np.argmin(a)
1030: (4)             0
1031: (4)             >>> np.argmin(a, axis=0)
1032: (4)             array([0, 0, 0])

```

```

1033: (4)          >>> np.argmin(a, axis=1)
1034: (4)          array([0, 0])
1035: (4)          Indices of the minimum elements of a N-dimensional array:
1036: (4)          >>> ind = np.unravel_index(np.argmin(a, axis=None), a.shape)
1037: (4)          >>> ind
1038: (4)          (0, 0)
1039: (4)          >>> a[ind]
1040: (4)          10
1041: (4)          >>> b = np.arange(6) + 10
1042: (4)          >>> b[4] = 10
1043: (4)          >>> b
1044: (4)          array([10, 11, 12, 13, 10, 15])
1045: (4)          >>> np.argmin(b) # Only the first occurrence is returned.
1046: (4)          0
1047: (4)          >>> x = np.array([[4,2,3], [1,0,3]])
1048: (4)          >>> index_array = np.argmax(x, axis=-1)
1049: (4)          >>> # Same as np.amin(x, axis=-1, keepdims=True)
1050: (4)          >>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1)
1051: (4)          array([[2],
1052: (11)             [0]])
1053: (4)          >>> # Same as np.amax(x, axis=-1)
1054: (4)          >>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1),
axis=-1).squeeze(axis=-1)
1055: (4)          array([2, 0])
1056: (4)          Setting `keepdims` to `True`,
1057: (4)          >>> x = np.arange(24).reshape((2, 3, 4))
1058: (4)          >>> res = np.argmax(x, axis=1, keepdims=True)
1059: (4)          >>> res.shape
1060: (4)          (2, 1, 4)
1061: (4)          """
1062: (4)          kwds = {'keepdims': keepdims} if keepdims is not np._NoValue else {}
1063: (4)          return _wrapfunc(a, 'argmin', axis=axis, out=out, **kwds)
1064: (0)          def _searchsorted_dispatcher(a, v, side=None, sorter=None):
1065: (4)          return (a, v, sorter)
1066: (0)          @array_function_dispatch(_searchsorted_dispatcher)
1067: (0)          def searchsorted(a, v, side='left', sorter=None):
1068: (4)          """
1069: (4)          Find indices where elements should be inserted to maintain order.
1070: (4)          Find the indices into a sorted array `a` such that, if the
1071: (4)          corresponding elements in `v` were inserted before the indices, the
1072: (4)          order of `a` would be preserved.
1073: (4)          Assuming that `a` is sorted:
1074: (4)          =====
1075: (4)          `side` returned index `i` satisfies
1076: (4)          =====
1077: (4)          left   ``a[i-1] < v <= a[i]``
1078: (4)          right  ``a[i-1] <= v < a[i]``
1079: (4)          =====
1080: (4)          Parameters
1081: (4)          -----
1082: (4)          a : 1-D array_like
1083: (8)          Input array. If `sorter` is None, then it must be sorted in
1084: (8)          ascending order, otherwise `sorter` must be an array of indices
1085: (8)          that sort it.
1086: (4)          v : array_like
1087: (8)          Values to insert into `a`.
1088: (4)          side : {'left', 'right'}, optional
1089: (8)          If 'left', the index of the first suitable location found is given.
1090: (8)          If 'right', return the last such index. If there is no suitable
1091: (8)          index, return either 0 or N (where N is the length of `a`).
1092: (4)          sorter : 1-D array_like, optional
1093: (8)          Optional array of integer indices that sort array a into ascending
1094: (8)          order. They are typically the result of argsort.
1095: (8)          .. versionadded:: 1.7.0
1096: (4)          Returns
1097: (4)          -----
1098: (4)          indices : int or array of ints
1099: (8)          Array of insertion points with the same shape as `v`,
1100: (8)          or an integer if `v` is a scalar.

```

```

1101: (4) See Also
1102: (4)
1103: (4)
1104: (4)
1105: (4)
1106: (4)
1107: (4)
1108: (4)
1109: (4)
1110: (4)
`bisect.bisect_left` -----
sort : Return a sorted copy of an array.
histogram : Produce histogram from 1-D data.
Notes -----
Binary search is used to find the required insertion points.
As of NumPy 1.4.0 `searchsorted` works with real/complex arrays containing
`nan` values. The enhanced sort order is documented in `sort`.
This function uses the same algorithm as the builtin python

(``side='left'``) and `bisect.bisect_right` (``side='right'``) functions,
which is also vectorized in the `v` argument.
Examples -----
>>> np.searchsorted([1,2,3,4,5], 3)
2
>>> np.searchsorted([1,2,3,4,5], 3, side='right')
3
>>> np.searchsorted([1,2,3,4,5], [-10, 10, 2, 3])
array([0, 5, 1, 2])
"""
return _wrapfunc(a, 'searchsorted', v, side=side, sorter=sorter)
def _resize_dispatcher(a, new_shape):
    return (a,)
@array_function_dispatch(_resize_dispatcher)
def resize(a, new_shape):
    """
    Return a new array with the specified shape.
    If the new array is larger than the original array, then the new
    array is filled with repeated copies of `a`. Note that this behavior
    is different from a.resize(new_shape) which fills with zeros instead
    of repeated copies of `a`.
    Parameters
    -----
    a : array_like
        Array to be resized.
    new_shape : int or tuple of int
        Shape of resized array.
    Returns
    -----
    reshaped_array : ndarray
        The new array is formed from the data in the old array, repeated
        if necessary to fill out the required number of elements. The
        data are repeated iterating over the array in C-order.
See Also
-----
numpy.reshape : Reshape an array without changing the total size.
numpy.pad : Enlarge and pad an array.
numpy.repeat : Repeat elements of an array.
ndarray.resize : resize an array in-place.
Notes -----
When the total size of the array does not change `~numpy.reshape` should
be used. In most other cases either indexing (to reduce the size)
or padding (to increase the size) may be a more appropriate solution.
Warning: This functionality does **not** consider axes separately,
i.e. it does not apply interpolation/extrapolation.
It fills the return array with the required number of elements, iterating
over `a` in C-order, disregarding axes (and cycling back from the start if
the new shape is larger). This functionality is therefore not suitable to
resize images, or data where each axis represents a separate and distinct
entity.
Examples -----
>>> a=np.array([[0,1],[2,3]])
>>> np.resize(a,(2,3))
array([[0, 1, 2],
       [3, 0, 1]])

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1169: (4)             >>> np.resize(a,(1,4))
1170: (4)             array([[0, 1, 2, 3]])
1171: (4)             >>> np.resize(a,(2,4))
1172: (4)             array([[0, 1, 2, 3],
1173: (11)                [0, 1, 2, 3]])
1174: (4)             """
1175: (4)             if isinstance(new_shape, (int, nt.integer)):
1176: (8)                 new_shape = (new_shape,)
1177: (4)             a = ravel(a)
1178: (4)             new_size = 1
1179: (4)             for dim_length in new_shape:
1180: (8)                 new_size *= dim_length
1181: (8)                 if dim_length < 0:
1182: (12)                     raise ValueError('all elements of `new_shape` must be non-negative')
1183: (4)             if a.size == 0 or new_size == 0:
1184: (8)                 return np.zeros_like(a, shape=new_shape)
1185: (4)             repeats = -(-new_size // a.size) # ceil division
1186: (4)             a = concatenate((a,) * repeats)[:new_size]
1187: (4)             return reshape(a, new_shape)
1188: (0)             def _squeeze_dispatcher(a, axis=None):
1189: (4)                 return (a,)
1190: (0)             @array_function_dispatch(_squeeze_dispatcher)
1191: (0)             def squeeze(a, axis=None):
1192: (4)                 """
1193: (4)                 Remove axes of length one from `a`.
1194: (4)                 Parameters
1195: (4)                 -----
1196: (4)                 a : array_like
1197: (8)                     Input data.
1198: (4)                 axis : None or int or tuple of ints, optional
1199: (8)                     .. versionadded:: 1.7.0
1200: (8)                     Selects a subset of the entries of length one in the
1201: (8)                     shape. If an axis is selected with shape entry greater than
1202: (8)                     one, an error is raised.
1203: (4)                 Returns
1204: (4)                 -----
1205: (4)                 squeezed : ndarray
1206: (8)                     The input array, but with all or a subset of the
1207: (8)                     dimensions of length 1 removed. This is always `a` itself
1208: (8)                     or a view into `a`. Note that if all axes are squeezed,
1209: (8)                     the result is a 0d array and not a scalar.
1210: (4)                 Raises
1211: (4)                 -----
1212: (4)                 ValueError
1213: (8)                     If `axis` is not None, and an axis being squeezed is not of length 1
1214: (4)                 See Also
1215: (4)                 -----
1216: (4)                 expand_dims : The inverse operation, adding entries of length one
1217: (4)                 reshape : Insert, remove, and combine dimensions, and resize existing ones
1218: (4)                 Examples
1219: (4)                 -----
1220: (4)                 >>> x = np.array([[[0], [1], [2]]])
1221: (4)                 >>> x.shape
1222: (4)                 (1, 3, 1)
1223: (4)                 >>> np.squeeze(x).shape
1224: (4)                 (3,)
1225: (4)                 >>> np.squeeze(x, axis=0).shape
1226: (4)                 (3, 1)
1227: (4)                 >>> np.squeeze(x, axis=1).shape
1228: (4)                 Traceback (most recent call last):
1229: (4)                     ...
1230: (4)                     ValueError: cannot select an axis to squeeze out which has size not equal
to one
1231: (4)                 >>> np.squeeze(x, axis=2).shape
1232: (4)                 (1, 3)
1233: (4)                 >>> x = np.array([[1234]])
1234: (4)                 >>> x.shape
1235: (4)                 (1, 1)

```

```

1236: (4)          >>> np.squeeze(x)
1237: (4)          array(1234)  # 0d array
1238: (4)          >>> np.squeeze(x).shape
1239: (4)          ()
1240: (4)          >>> np.squeeze(x)[()]
1241: (4)          1234
1242: (4)          """
1243: (4)          try:
1244: (8)              squeeze = a.squeeze
1245: (4)          except AttributeError:
1246: (8)              return _wrapit(a, 'squeeze', axis=axis)
1247: (4)          if axis is None:
1248: (8)              return squeeze()
1249: (4)          else:
1250: (8)              return squeeze(axis=axis)
1251: (0)          def _diagonal_dispatcher(a, offset=None, axis1=None, axis2=None):
1252: (4)              return (a,)
1253: (0)          @array_function_dispatch(_diagonal_dispatcher)
1254: (0)          def diagonal(a, offset=0, axis1=0, axis2=1):
1255: (4)              """
1256: (4)              Return specified diagonals.
1257: (4)              If `a` is 2-D, returns the diagonal of `a` with the given offset,
1258: (4)              i.e., the collection of elements of the form ``a[i, i+offset]``. If
1259: (4)              `a` has more than two dimensions, then the axes specified by `axis1`
1260: (4)              and `axis2` are used to determine the 2-D sub-array whose diagonal is
1261: (4)              returned. The shape of the resulting array can be determined by
1262: (4)              removing `axis1` and `axis2` and appending an index to the right equal
1263: (4)              to the size of the resulting diagonals.
1264: (4)              In versions of NumPy prior to 1.7, this function always returned a new,
1265: (4)              independent array containing a copy of the values in the diagonal.
1266: (4)              In NumPy 1.7 and 1.8, it continues to return a copy of the diagonal,
1267: (4)              but depending on this fact is deprecated. Writing to the resulting
1268: (4)              array continues to work as it used to, but a FutureWarning is issued.
1269: (4)              Starting in NumPy 1.9 it returns a read-only view on the original array.
1270: (4)              Attempting to write to the resulting array will produce an error.
1271: (4)              In some future release, it will return a read/write view and writing to
1272: (4)              the returned array will alter your original array. The returned array
1273: (4)              will have the same type as the input array.
1274: (4)              If you don't write to the array returned by this function, then you can
1275: (4)              just ignore all of the above.
1276: (4)              If you depend on the current behavior, then we suggest copying the
1277: (4)              returned array explicitly, i.e., use ``np.diagonal(a).copy()`` instead
1278: (4)              of just ``np.diagonal(a)``. This will work with both past and future
1279: (4)              versions of NumPy.
1280: (4)          Parameters
1281: (4)          -----
1282: (4)          a : array_like
1283: (8)              Array from which the diagonals are taken.
1284: (4)          offset : int, optional
1285: (8)              Offset of the diagonal from the main diagonal. Can be positive or
1286: (8)              negative. Defaults to main diagonal (0).
1287: (4)          axis1 : int, optional
1288: (8)              Axis to be used as the first axis of the 2-D sub-arrays from which
1289: (8)              the diagonals should be taken. Defaults to first axis (0).
1290: (4)          axis2 : int, optional
1291: (8)              Axis to be used as the second axis of the 2-D sub-arrays from
1292: (8)              which the diagonals should be taken. Defaults to second axis (1).
1293: (4)          Returns
1294: (4)          -----
1295: (4)          array_of_diagonals : ndarray
1296: (8)              If `a` is 2-D, then a 1-D array containing the diagonal and of the
1297: (8)              same type as `a` is returned unless `a` is a `matrix`, in which case
1298: (8)              a 1-D array rather than a (2-D) `matrix` is returned in order to
1299: (8)              maintain backward compatibility.
1300: (8)              If ``a.ndim > 2``, then the dimensions specified by `axis1` and
1301: (8)              `axis2`
1302: (8)              are removed, and a new axis inserted at the end corresponding to the
1303: (4)              diagonal.

```

**Raises**

```

1304: (4)
1305: (4)
1306: (8)
1307: (4)
1308: (4)
1309: (4)
1310: (4)
1311: (4)
1312: (4)
1313: (4)
1314: (4)
1315: (4)
1316: (4)
1317: (11)
1318: (4)
1319: (4)
1320: (4)
1321: (4)
1322: (4)
1323: (4)
1324: (4)
1325: (12)
1326: (11)
1327: (12)
1328: (4)
1329: (4)
1330: (4)
1331: (4)
1332: (11)
1333: (4)
1334: (4)
1335: (4)
1336: (4)
1337: (4)
1338: (11)
1339: (4)
1340: (4)
1341: (11)
1342: (4)
1343: (4)
1344: (4)
1345: (4)
1346: (4)
1347: (11)
1348: (11)
1349: (4)
1350: (4)
1351: (4)
1352: (4)
1353: (4)
1354: (4)
1355: (4)
1356: (4)
1357: (8)
1358: (4)
1359: (8)
1360: (0)
1361: (8)
1362: (4)
1363: (0)
1364: (0)
1365: (4)
1366: (4)
1367: (4)
1368: (4)
1369: (4)
1370: (4)
1371: (4)
1372: (4)

-----  

ValueError  

    If the dimension of `a` is less than 2.  

See Also  

-----  

diag : MATLAB work-a-like for 1-D and 2-D arrays.  

diagflat : Create diagonal arrays.  

trace : Sum along diagonals.  

Examples  

-----  

>>> a = np.arange(4).reshape(2,2)  

>>> a  

array([[0, 1],  

       [2, 3]])  

>>> a.diagonal()  

array([0, 3])  

>>> a.diagonal(1)  

array([1])  

A 3-D example:  

>>> a = np.arange(8).reshape(2,2,2); a  

array([[[0, 1],  

        [2, 3]],  

       [[4, 5],  

        [6, 7]]])  

>>> a.diagonal(0, # Main diagonals of two arrays created by skipping  

...          0, # across the outer(left)-most axis last and  

...          1) # the "middle" (row) axis first.  

array([[0, 6],  

       [1, 7]])  

The sub-arrays whose main diagonals we just obtained; note that each  

corresponds to fixing the right-most (column) axis, and that the  

diagonals are "packed" in rows.  

>>> a[:, :, 0] # main diagonal is [0 6]  

array([[0, 2],  

       [4, 6]])  

>>> a[:, :, 1] # main diagonal is [1 7]  

array([[1, 3],  

       [5, 7]])  

The anti-diagonal can be obtained by reversing the order of elements  

using either `numpy.flipud` or `numpy.fliplr`.  

>>> a = np.arange(9).reshape(3, 3)  

>>> a  

array([[0, 1, 2],  

       [3, 4, 5],  

       [6, 7, 8]])  

>>> np.fliplr(a).diagonal() # Horizontal flip  

array([2, 4, 6])  

>>> np.flipud(a).diagonal() # Vertical flip  

array([6, 4, 2])  

Note that the order in which the diagonal is retrieved varies depending  

on the flip function.  

"""  

if isinstance(a, np.matrix):  

    return asarray(a).diagonal(offset=offset, axis1=axis1, axis2=axis2)  

else:  

    return asanyarray(a).diagonal(offset=offset, axis1=axis1, axis2=axis2)  

def _trace_dispatcher(  

    a, offset=None, axis1=None, axis2=None, dtype=None, out=None):  

    return (a, out)  

@array_function_dispatch(_trace_dispatcher)  

def trace(a, offset=0, axis1=0, axis2=1, dtype=None, out=None):  

    """  

    Return the sum along diagonals of the array.  

    If `a` is 2-D, the sum along its diagonal with the given offset  

    is returned, i.e., the sum of elements ``a[i,i+offset]`` for all i.  

    If `a` has more than two dimensions, then the axes specified by axis1 and  

    axis2 are used to determine the 2-D sub-arrays whose traces are returned.  

    The shape of the resulting array is the same as that of `a` with `axis1`  

    and `axis2` removed.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1373: (4)             Parameters
1374: (4)             -----
1375: (4)             a : array_like
1376: (8)               Input array, from which the diagonals are taken.
1377: (4)             offset : int, optional
1378: (8)               Offset of the diagonal from the main diagonal. Can be both positive
1379: (8)               and negative. Defaults to 0.
1380: (4)             axis1, axis2 : int, optional
1381: (8)               Axes to be used as the first and second axis of the 2-D sub-arrays
1382: (8)               from which the diagonals should be taken. Defaults are the first two
1383: (8)               axes of `a`.
1384: (4)             dtype : dtype, optional
1385: (8)               Determines the data-type of the returned array and of the accumulator
1386: (8)               where the elements are summed. If dtype has the value None and `a` is
1387: (8)               of integer type of precision less than the default integer
1388: (8)               precision, then the default integer precision is used. Otherwise,
1389: (8)               the precision is the same as that of `a`.
1390: (4)             out : ndarray, optional
1391: (8)               Array into which the output is placed. Its type is preserved and
1392: (8)               it must be of the right shape to hold the output.
1393: (4)             Returns
1394: (4)             -----
1395: (4)             sum_along_diagonals : ndarray
1396: (8)               If `a` is 2-D, the sum along the diagonal is returned. If `a` has
1397: (8)               larger dimensions, then an array of sums along diagonals is returned.
1398: (4)             See Also
1399: (4)             -----
1400: (4)             diag, diagonal, diagflat
1401: (4)             Examples
1402: (4)             -----
1403: (4)             >>> np.trace(np.eye(3))
1404: (4)             3.0
1405: (4)             >>> a = np.arange(8).reshape((2,2,2))
1406: (4)             >>> np.trace(a)
1407: (4)             array([6, 8])
1408: (4)             >>> a = np.arange(24).reshape((2,2,2,3))
1409: (4)             >>> np.trace(a).shape
1410: (4)             (2, 3)
1411: (4)             """
1412: (4)             if isinstance(a, np.matrix):
1413: (8)                 return asarray(a).trace(offset=offset, axis1=axis1, axis2=axis2,
1414: (4)             dtype=dtype, out=out)
1415: (4)             else:
1416: (8)                 return asanyarray(a).trace(offset=offset, axis1=axis1, axis2=axis2,
1417: (4)             dtype=dtype, out=out)
1418: (0)                 def _ravel_dispatcher(a, order=None):
1419: (4)                     return (a,)
1420: (0)                     @array_function_dispatch(_ravel_dispatcher)
1421: (4)                     def ravel(a, order='C'):
1422: (4)                         """Return a contiguous flattened array.
1423: (4)                         A 1-D array, containing the elements of the input, is returned. A copy is
1424: (4)                         made only if needed.
1425: (4)                         As of NumPy 1.10, the returned array will have the same type as the input
1426: (4)                         array. (for example, a masked array will be returned for a masked array
1427: (4)                         input)
1428: (4)                         Parameters
1429: (4)                         -----
1430: (4)                         a : array_like
1431: (8)                           Input array. The elements in `a` are read in the order specified by
1432: (8)                           `order`, and packed as a 1-D array.
1433: (4)                         order : {'C', 'F', 'A', 'K'}, optional
1434: (8)                           The elements of `a` are read using this index order. 'C' means
1435: (8)                           to index the elements in row-major, C-style order,
1436: (8)                           with the last axis index changing fastest, back to the first
1437: (8)                           axis index changing slowest. 'F' means to index the elements
1438: (8)                           in column-major, Fortran-style order, with the
1439: (8)                           first index changing fastest, and the last index changing
                           slowest. Note that the 'C' and 'F' options take no account of
                           the memory layout of the underlying array, and only refer to

```

```

1440: (8)          the order of axis indexing. 'A' means to read the elements in
1441: (8)          Fortran-like index order if `a` is Fortran *contiguous* in
1442: (8)          memory, C-like order otherwise. 'K' means to read the
1443: (8)          elements in the order they occur in memory, except for
1444: (8)          reversing the data when strides are negative. By default, 'C'
1445: (8)          index order is used.
1446: (4)          Returns
1447: (4)
1448: (4)          -----
1449: (8)          y : array_like
1450: (8)          y is a contiguous 1-D array of the same subtype as `a`,
1451: (8)          with shape ``a.size,``.
1452: (8)          Note that matrices are special cased for backward compatibility,
1453: (8)          if `a` is a matrix, then y is a 1-D ndarray.
1454: (4)          See Also
1455: (4)          -----
1456: (4)          ndarray.flat : 1-D iterator over an array.
1457: (22)          ndarray.flatten : 1-D array copy of the elements of an array
1458: (4)          in row-major order.
1459: (4)          ndarray.reshape : Change the shape of an array without changing its data.
1460: (4)          Notes
1461: (4)          -----
1462: (4)          In row-major, C-style order, in two dimensions, the row index
1463: (4)          varies the slowest, and the column index the quickest. This can
1464: (4)          be generalized to multiple dimensions, where row-major order
1465: (4)          implies that the index along the first axis varies slowest, and
1466: (4)          the index along the last quickest. The opposite holds for
1467: (4)          column-major, Fortran-style index ordering.
1468: (4)          When a view is desired in as many cases as possible, ``arr.reshape(-1)``
1469: (4)          may be preferable. However, ``ravel`` supports ``K`` in the optional
1470: (4)          ``order`` argument while ``reshape`` does not.
1471: (4)          Examples
1472: (4)          -----
1473: (4)          It is equivalent to ``reshape(-1, order=order)``.
1474: (4)          >>> x = np.array([[1, 2, 3], [4, 5, 6]])
1475: (4)          >>> np.ravel(x)
1476: (4)          array([1, 2, 3, 4, 5, 6])
1477: (4)          >>> x.reshape(-1)
1478: (4)          array([1, 2, 3, 4, 5, 6])
1479: (4)          >>> np.ravel(x, order='F')
1480: (4)          array([1, 4, 2, 5, 3, 6])
1481: (4)          When ``order`` is 'A', it will preserve the array's 'C' or 'F' ordering:
1482: (4)          >>> np.ravel(x.T)
1483: (4)          array([1, 4, 2, 5, 3, 6])
1484: (4)          >>> np.ravel(x.T, order='A')
1485: (4)          array([1, 2, 3, 4, 5, 6])
1486: (4)          When ``order`` is 'K', it will preserve orderings that are neither 'C'
1487: (4)          nor 'F', but won't reverse axes:
1488: (4)          >>> a = np.arange(3)[::-1]; a
1489: (4)          array([2, 1, 0])
1490: (4)          >>> a.ravel(order='C')
1491: (4)          array([2, 1, 0])
1492: (4)          >>> a.ravel(order='K')
1493: (4)          array([2, 1, 0])
1494: (4)          >>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
1495: (12)          array([[[ 0,  2,  4],
1496: (11)                  [ 1,  3,  5]],
1497: (12)                  [[ 6,  8, 10],
1498: (4)                  [ 7,  9, 11]]])
1499: (4)          >>> a.ravel(order='C')
1500: (4)          array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
1501: (4)          >>> a.ravel(order='K')
1502: (4)          array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
1503: (4)
1504: (8)          """
1505: (4)          if isinstance(a, np.matrix):
1506: (8)              return asarray(a).ravel(order=order)
1507: (0)          else:
1508: (4)              return asanyarray(a).ravel(order=order)
def _nonzero_dispatcher(a):
    return (a,) 
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1509: (0)
1510: (0)
1511: (4)
1512: (4)
1513: (4)
1514: (4)
1515: (4)
1516: (4)
1517: (4)
1518: (4)
1519: (4)
1520: (7)
1521: (7)
1522: (7)
1523: (10)
1524: (4)
1525: (4)
1526: (4)
1527: (8)
1528: (4)
1529: (4)
1530: (4)
1531: (8)
1532: (4)
1533: (4)
1534: (4)
1535: (8)
1536: (8)
1537: (4)
1538: (8)
1539: (4)
1540: (8)
1541: (4)
1542: (4)
1543: (4)
1544: (4)
1545: (4)
1546: (4)
1547: (4)
1548: (4)
1549: (4)
1550: (4)
1551: (11)
1552: (11)
1553: (4)
1554: (4)
1555: (4)
1556: (4)
1557: (4)
1558: (4)
1559: (11)
1560: (11)
1561: (11)
1562: (4)
1563: (4)
1564: (4)
1565: (4)
1566: (4)
1567: (4)
1568: (4)
1569: (11)
1570: (11)
1571: (4)
1572: (4)
1573: (4)
1574: (4)
1575: (4)
1576: (4)
1577: (4)

@array_function_dispatch(_nonzero_dispatcher)
def nonzero(a):
    """
    Return the indices of the elements that are non-zero.
    Returns a tuple of arrays, one for each dimension of `a`,
    containing the indices of the non-zero elements in that
    dimension. The values in `a` are always tested and returned in
    row-major, C-style order.
    To group the indices by element, rather than dimension, use `argwhere`,
    which returns a row for each non-zero element.
    .. note::
        When called on a zero-d array or scalar, ``nonzero(a)`` is treated
        as ``nonzero(atleast_1d(a))``.
        .. deprecated:: 1.17.0
            Use `atleast_1d` explicitly if this behavior is deliberate.
    Parameters
    -----
    a : array_like
        Input array.
    Returns
    -----
    tuple_of_arrays : tuple
        Indices of elements that are non-zero.
    See Also
    -----
    flatnonzero :
        Return indices that are non-zero in the flattened version of the input
        array.
    ndarray.nonzero :
        Equivalent ndarray method.
    count_nonzero :
        Counts the number of non-zero elements in the input array.
    Notes
    -----
    While the nonzero values can be obtained with ``a[nonzero(a)]``, it is
    recommended to use ``x[x.astype(bool)]`` or ``x[x != 0]`` instead, which
    will correctly handle 0-d arrays.
    Examples
    -----
    >>> x = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
    >>> x
    array([[3, 0, 0],
           [0, 4, 0],
           [5, 6, 0]])
    >>> np.nonzero(x)
    (array([0, 1, 2, 2]), array([0, 1, 0, 1]))
    >>> x[np.nonzero(x)]
    array([3, 4, 5, 6])
    >>> np.transpose(np.nonzero(x))
    array([[0, 0],
           [1, 1],
           [2, 0],
           [2, 1]])
    A common use for ``nonzero`` is to find the indices of an array, where
    a condition is True. Given an array `a`, the condition `a > 3` is a
    boolean array and since False is interpreted as 0, np.nonzero(a > 3)
    yields the indices of the `a` where the condition is true.
    >>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    >>> a > 3
    array([[False, False, False],
           [ True,  True,  True],
           [ True,  True,  True]])
    >>> np.nonzero(a > 3)
    (array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
    Using this result to index `a` is equivalent to using the mask directly:
    >>> a[np.nonzero(a > 3)]
    array([4, 5, 6, 7, 8, 9])
    >>> a[a > 3] # prefer this spelling
    array([4, 5, 6, 7, 8, 9])

```

```

1578: (4)           ``nonzero`` can also be called as a method of the array.
1579: (4)           >>> (a > 3).nonzero()
1580: (4)           (array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
1581: (4)
1582: (4)           """
1583: (0)           return _wrapfunc(a, 'nonzero')
1584: (4)           def _shape_dispatcher(a):
1585: (0)               return (a,)
1586: (0)           @array_function_dispatch(_shape_dispatcher)
1587: (4)           def shape(a):
1588: (4)               """
1589: (4)                   Return the shape of an array.
1590: (4)           Parameters
1591: (4)               -----
1592: (8)               a : array_like
1593: (4)                   Input array.
1594: (4)           Returns
1595: (4)               -----
1596: (4)               shape : tuple of ints
1597: (4)                   The elements of the shape tuple give the lengths of the
1598: (4)                   corresponding array dimensions.
1599: (4)           See Also
1600: (4)               -----
1601: (10)              len : ``len(a)`` is equivalent to ``np.shape(a)[0]`` for N-D arrays with
1602: (4)                  ``N>=1``.
1603: (4)              ndarray.shape : Equivalent array method.
1604: (4)           Examples
1605: (4)               -----
1606: (4)               >>> np.shape(np.eye(3))
1607: (4)               (3, 3)
1608: (4)               >>> np.shape([[1, 3]])
1609: (4)               (1, 2)
1610: (4)               >>> np.shape([0])
1611: (4)               (1,)
1612: (4)               >>> np.shape(0)
1613: (4)               ()
1614: (4)               >>> a = np.array([(1, 2), (3, 4), (5, 6)],
1615: (4)                   ...                      dtype=[('x', 'i4'), ('y', 'i4')])
1616: (4)               >>> np.shape(a)
1617: (4)               (3,)
1618: (4)               >>> a.shape
1619: (4)               (3,)
1620: (4)               """
1621: (8)               try:
1622: (4)                   result = a.shape
1623: (8)               except AttributeError:
1624: (4)                   result = asarray(a).shape
1625: (0)               return result
1626: (4)           def _compress_dispatcher(condition, a, axis=None, out=None):
1627: (0)               return (condition, a, out)
1628: (0)           @array_function_dispatch(_compress_dispatcher)
1629: (4)           def compress(condition, a, axis=None, out=None):
1630: (4)               """
1631: (4)                   Return selected slices of an array along given axis.
1632: (4)                   When working along a given axis, a slice along that axis is returned in
1633: (4)                   `output` for each index where `condition` evaluates to True. When
1634: (4)                   working on a 1-D array, `compress` is equivalent to `extract`.
1635: (4)           Parameters
1636: (4)               -----
1637: (8)               condition : 1-D array of bools
1638: (8)                   Array that selects which entries to return. If len(condition)
1639: (8)                   is less than the size of `a` along the given axis, then output is
1640: (4)                   truncated to the length of the condition array.
1641: (8)               a : array_like
1642: (4)                   Array from which to extract a part.
1643: (8)               axis : int, optional
1644: (8)                   Axis along which to take slices. If None (default), work on the
1645: (4)                   flattened array.
1646: (8)               out : ndarray, optional
1647: (4)                   Output array. Its type is preserved and it must be of the right

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1647: (8)           shape to hold the output.
1648: (4)           Returns
1649: (4)
1650: (4)
1651: (8)           -----
1652: (8)           compressed_array : ndarray
1653: (4)           A copy of `a` without the slices along axis for which `condition`
1654: (4)           is false.
1655: (4)           See Also
1656: (4)
1657: (4)
1658: (4)
1659: (4)           -----
1660: (4)           Examples
1661: (4)           -----
1662: (4)           >>> a = np.array([[1, 2], [3, 4], [5, 6]])
1663: (4)           >>> a
1664: (11)          array([[1, 2],
1665: (11)          [3, 4],
1666: (4)          [5, 6]])
1667: (4)           >>> np.compress([0, 1], a, axis=0)
1668: (4)           array([[3, 4]])
1669: (4)           >>> np.compress([False, True, True], a, axis=0)
1670: (11)          array([[3, 4],
1671: (4)          [5, 6]])
1672: (4)           >>> np.compress([False, True], a, axis=1)
1673: (11)          array([[2],
1674: (11)          [4],
1675: (4)          [6]])
1676: (4)           Working on the flattened array does not return slices along an axis but
1677: (4)           selects elements.
1678: (4)           >>> np.compress([False, True], a)
1679: (4)           array([2])
1680: (4)           """
1681: (0)           return _wrapfunc(a, 'compress', condition, axis=axis, out=out)
def _clip_dispatcher(a, a_min, a_max, out=None, **kwargs):
1682: (4)           return (a, a_min, a_max)
1683: (0)           @array_function_dispatch(_clip_dispatcher)
1684: (0)           def clip(a, a_min, a_max, out=None, **kwargs):
1685: (4)           """
1686: (4)           Clip (limit) the values in an array.
1687: (4)           Given an interval, values outside the interval are clipped to
1688: (4)           the interval edges. For example, if an interval of ``[0, 1]``
1689: (4)           is specified, values smaller than 0 become 0, and values larger
1690: (4)           than 1 become 1.
1691: (4)           Equivalent to but faster than ``np.minimum(a_max, np.maximum(a, a_min))``.
1692: (4)           No check is performed to ensure ``a_min < a_max``.
1693: (4)           Parameters
1694: (4)
1695: (4)           a : array_like
1696: (8)           Array containing elements to clip.
1697: (4)           a_min, a_max : array_like or None
1698: (8)           Minimum and maximum value. If ``None``, clipping is not performed on
1699: (8)           the corresponding edge. Only one of `a_min` and `a_max` may be
1700: (8)           ``None``. Both are broadcast against `a`.
1701: (4)           out : ndarray, optional
1702: (8)           The results will be placed in this array. It may be the input
1703: (8)           array for in-place clipping. `out` must be of the right shape
1704: (8)           to hold the output. Its type is preserved.
1705: (4)           **kwargs
1706: (8)           For other keyword-only arguments, see the
1707: (8)           :ref:`ufunc docs <ufuncs.kwargs>`.
1708: (8)           .. versionadded:: 1.17.0
1709: (4)
1710: (4)
1711: (4)           Returns
1712: (8)
1713: (8)           -----
1714: (8)           clipped_array : ndarray
1715: (4)           An array with the elements of `a`, but where values
1716: (8)           < `a_min` are replaced with `a_min`, and those > `a_max`
1717: (8)           with `a_max`.
1718: (4)           See Also

```

```

1716: (4)
1717: (4)
1718: (4)
1719: (4)
1720: (4)
1721: (4)
1722: (4)
1723: (4)
1724: (4)
1725: (4)
1726: (4)
1727: (4)
1728: (4)
1729: (4)
1730: (4)
1731: (4)
1732: (4)
1733: (4)
1734: (4)
1735: (4)
1736: (4)
1737: (4)
1738: (4)
1739: (4)
1740: (4)
1741: (4)
1742: (4)
1743: (0)
1744: (20)
1745: (4)
1746: (0)
1747: (0)
1748: (8)
1749: (4)
1750: (4)
1751: (4)
1752: (4)
1753: (4)
1754: (8)
1755: (4)
1756: (8)
1757: (8)
1758: (8)
1759: (8)
1760: (8)
1761: (8)
1762: (8)
1763: (4)
1764: (8)
1765: (8)
1766: (8)
1767: (8)
1768: (8)
1769: (8)
1770: (4)
1771: (8)
1772: (8)
1773: (8)
1774: (4)
1775: (8)
1776: (8)
1777: (8)
1778: (8)
1779: (8)
1780: (8)
1781: (8)
1782: (8)
1783: (4)
1784: (8)

-----  

:ref:`ufuncs-output-type`  

Notes  

-----  

When `a_min` is greater than `a_max`, `clip` returns an array in which all values are equal to `a_max`, as shown in the second example.  

Examples  

-----  

>>> a = np.arange(10)  

>>> a  

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])  

>>> np.clip(a, 1, 8)  

array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])  

>>> np.clip(a, 8, 1)  

array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])  

>>> np.clip(a, 3, 6, out=a)  

array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])  

>>> a  

array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])  

>>> a = np.arange(10)  

>>> a  

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])  

>>> np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)  

array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])  

"""  

    return _wrapfunc(a, 'clip', a_min, a_max, out=out, **kwargs)  

def _sum_dispatcher(a, axis=None, dtype=None, out=None, keepdims=None,  

                    initial=None, where=None):  

    return (a, out)  

@array_function_dispatch(_sum_dispatcher)  

def sum(a, axis=None, dtype=None, out=None, keepdims=np._NoValue,  

       initial=np._NoValue, where=np._NoValue):  

    """  

        Sum of array elements over a given axis.  

Parameters  

-----  

a : array_like  

    Elements to sum.  

axis : None or int or tuple of ints, optional  

    Axis or axes along which a sum is performed. The default,  

    axis=None, will sum all of the elements of the input array. If  

    axis is negative it counts from the last to the first axis.  

.. versionadded:: 1.7.0  

    If axis is a tuple of ints, a sum is performed on all of the axes  

    specified in the tuple instead of a single axis or all the axes as  

    before.  

dtype : dtype, optional  

    The type of the returned array and of the accumulator in which the  

    elements are summed. The dtype of `a` is used by default unless `a`  

    has an integer dtype of less precision than the default platform  

    integer. In that case, if `a` is signed then the platform integer  

    is used while if `a` is unsigned then an unsigned integer of the  

    same precision as the platform integer is used.  

out : ndarray, optional  

    Alternative output array in which to place the result. It must have  

    the same shape as the expected output, but the type of the output  

    values will be cast if necessary.  

keepdims : bool, optional  

    If this is set to True, the axes which are reduced are left  

    in the result as dimensions with size one. With this option,  

    the result will broadcast correctly against the input array.  

    If the default value is passed, then `keepdims` will not be  

    passed through to the `sum` method of sub-classes of  

    `ndarray`, however any non-default value will be. If the  

    sub-class' method does not implement `keepdims` any  

    exceptions will be raised.  

initial : scalar, optional  

    Starting value for the sum. See `~numpy.ufunc.reduce` for details.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1785: (8) .. versionadded:: 1.15.0
1786: (4) where : array_like of bool, optional
1787: (8) Elements to include in the sum. See `~numpy.ufunc.reduce` for details.
1788: (8) .. versionadded:: 1.17.0
1789: (4) Returns
1790: (4) -----
1791: (4) sum_along_axis : ndarray
1792: (8) An array with the same shape as `a`, with the specified
1793: (8) axis removed. If `a` is a 0-d array, or if `axis` is None, a scalar
1794: (8) is returned. If an output array is specified, a reference to
1795: (8) `out` is returned.
1796: (4) See Also
1797: (4) -----
1798: (4) ndarray.sum : Equivalent method.
1799: (4) add.reduce : Equivalent functionality of `add`.
1800: (4) cumsum : Cumulative sum of array elements.
1801: (4) trapz : Integration of array values using the composite trapezoidal rule.
1802: (4) mean, average
1803: (4) Notes
1804: (4) -----
1805: (4) Arithmetic is modular when using integer types, and no error is
1806: (4) raised on overflow.
1807: (4) The sum of an empty array is the neutral element 0:
1808: (4) >>> np.sum([])
1809: (4) 0.0
1810: (4) For floating point numbers the numerical precision of sum (and
1811: (4) ``np.add.reduce``) is in general limited by directly adding each number
1812: (4) individually to the result causing rounding errors in every step.
1813: (4) However, often numpy will use a numerically better approach (partial
1814: (4) pairwise summation) leading to improved precision in many use-cases.
1815: (4) This improved precision is always provided when no ``axis`` is given.
1816: (4) When ``axis`` is given, it will depend on which axis is summed.
1817: (4) Technically, to provide the best speed possible, the improved precision
1818: (4) is only used when the summation is along the fast axis in memory.
1819: (4) Note that the exact precision may vary depending on other parameters.
1820: (4) In contrast to NumPy, Python's ``math.fsum`` function uses a slower but
1821: (4) more precise approach to summation.
1822: (4) Especially when summing a large number of lower precision floating point
1823: (4) numbers, such as ``float32``, numerical errors can become significant.
1824: (4) In such cases it can be advisable to use `dtype="float64"` to use a higher
1825: (4) precision for the output.
1826: (4) Examples
1827: (4) -----
1828: (4) >>> np.sum([0.5, 1.5])
1829: (4) 2.0
1830: (4) >>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1831: (4) 1
1832: (4) >>> np.sum([[0, 1], [0, 5]])
1833: (4) 6
1834: (4) >>> np.sum([[0, 1], [0, 5]], axis=0)
1835: (4) array([0, 6])
1836: (4) >>> np.sum([[0, 1], [0, 5]], axis=1)
1837: (4) array([1, 5])
1838: (4) >>> np.sum([[0, 1], [np.nan, 5]], where=[False, True], axis=1)
1839: (4) array([1., 5.])
1840: (4) If the accumulator is too small, overflow occurs:
1841: (4) >>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
1842: (4) -128
1843: (4) You can also start the sum with a value other than zero:
1844: (4) >>> np.sum([10], initial=5)
1845: (4) 15
1846: (4) """
1847: (4)     if isinstance(a, _gentype):
1848: (8)         warnings.warn(
1849: (12)             "Calling np.sum(generator) is deprecated, and in the future will
give a different result. "
1850: (12)             "Use np.sum(np.fromiter(generator)) or the python sum builtin
instead.",
1851: (12)             DeprecationWarning, stacklevel=2)

```

```

1852: (8)             res = _sum_(a)
1853: (8)             if out is not None:
1854: (12)             out[...] = res
1855: (12)             return out
1856: (8)             return res
1857: (4)             return _wrapreduction(a, np.add, 'sum', axis, dtype, out,
keepdims=keepdims,
1858: (26)                     initial=initial, where=where)
1859: (0)             def _any_dispatcher(a, axis=None, out=None, keepdims=None, *,
1860: (20)                         where=np._NoValue):
1861: (4)                         return (a, where, out)
1862: (0) @array_function_dispatch(_any_dispatcher)
1863: (0) def any(a, axis=None, out=None, keepdims=np._NoValue, *, where=np._NoValue):
1864: (4) """
1865: (4)     Test whether any array element along a given axis evaluates to True.
1866: (4)     Returns single boolean if `axis` is ``None``.
1867: (4)     Parameters
1868: (4)     -----
1869: (4)     a : array_like
1870: (8)         Input array or object that can be converted to an array.
1871: (4)     axis : None or int or tuple of ints, optional
1872: (8)         Axis or axes along which a logical OR reduction is performed.
1873: (8)         The default (``axis=None``) is to perform a logical OR over all
1874: (8)         the dimensions of the input array. `axis` may be negative, in
1875: (8)         which case it counts from the last to the first axis.
1876: (8)         .. versionadded:: 1.7.0
1877: (8)         If this is a tuple of ints, a reduction is performed on multiple
1878: (8)         axes, instead of a single axis or all the axes as before.
1879: (4)     out : ndarray, optional
1880: (8)         Alternate output array in which to place the result. It must have
1881: (8)         the same shape as the expected output and its type is preserved
1882: (8)         (e.g., if it is of type float, then it will remain so, returning
1883: (8)         1.0 for True and 0.0 for False, regardless of the type of `a`).
1884: (8)         See :ref:`ufuncs-output-type` for more details.
1885: (4)     keepdims : bool, optional
1886: (8)         If this is set to True, the axes which are reduced are left
1887: (8)         in the result as dimensions with size one. With this option,
1888: (8)         the result will broadcast correctly against the input array.
1889: (8)         If the default value is passed, then `keepdims` will not be
1890: (8)         passed through to the `any` method of sub-classes of
1891: (8)         `ndarray`, however any non-default value will be. If the
1892: (8)         sub-class' method does not implement `keepdims` any
1893: (8)         exceptions will be raised.
1894: (4)     where : array_like of bool, optional
1895: (8)         Elements to include in checking for any `True` values.
1896: (8)         See `~numpy.ufunc.reduce` for details.
1897: (8)         .. versionadded:: 1.20.0
1898: (4)     Returns
1899: (4)     -----
1900: (4)     any : bool or ndarray
1901: (8)         A new boolean or `ndarray` is returned unless `out` is specified,
1902: (8)         in which case a reference to `out` is returned.
1903: (4)     See Also
1904: (4)     -----
1905: (4)     ndarray.any : equivalent method
1906: (4)     all : Test whether all elements along a given axis evaluate to True.
1907: (4)     Notes
1908: (4)     -----
1909: (4)     Not a Number (NaN), positive infinity and negative infinity evaluate
1910: (4)     to `True` because these are not equal to zero.
1911: (4)     Examples
1912: (4)     -----
1913: (4)     >>> np.any([[True, False], [True, True]])
1914: (4)     True
1915: (4)     >>> np.any([[True, False], [False, False]], axis=0)
1916: (4)     array([ True, False])
1917: (4)     >>> np.any([-1, 0, 5])
1918: (4)     True
1919: (4)     >>> np.any(np.nan)

```

```

1920: (4)          True
1921: (4)          >>> np.any([[True, False], [False, False]], where=[[False], [True]])
1922: (4)          False
1923: (4)          >>> o=np.array(False)
1924: (4)          >>> z=np.any([-1, 4, 5], out=o)
1925: (4)          >>> z, o
1926: (4)          (array(True), array(True))
1927: (4)          >>> # Check now that z is a reference to o
1928: (4)          >>> z is o
1929: (4)          True
1930: (4)          >>> id(z), id(o) # identity of z and o           # doctest: +SKIP
1931: (4)          (191614240, 191614240)
1932: (4)          """
1933: (4)          return _wrapreduction(a, np.logical_or, 'any', axis, None, out,
1934: (26)            keepdims=keepdims, where=where)
1935: (0)          def _all_dispatcher(a, axis=None, out=None, keepdims=None, *,
1936: (20)            where=None):
1937: (4)          return (a, where, out)
1938: (0)          @array_function_dispatch(_all_dispatcher)
1939: (0)          def all(a, axis=None, out=None, keepdims=np._NoValue, *, where=np._NoValue):
1940: (4)          """
1941: (4)          Test whether all array elements along a given axis evaluate to True.
1942: (4)          Parameters
1943: (4)          -----
1944: (4)          a : array_like
1945: (8)          Input array or object that can be converted to an array.
1946: (4)          axis : None or int or tuple of ints, optional
1947: (8)          Axis or axes along which a logical AND reduction is performed.
1948: (8)          The default (`axis=None`) is to perform a logical AND over all
1949: (8)          the dimensions of the input array. `axis` may be negative, in
1950: (8)          which case it counts from the last to the first axis.
1951: (8)          .. versionadded:: 1.7.0
1952: (8)          If this is a tuple of ints, a reduction is performed on multiple
1953: (8)          axes, instead of a single axis or all the axes as before.
1954: (4)          out : ndarray, optional
1955: (8)          Alternate output array in which to place the result.
1956: (8)          It must have the same shape as the expected output and its
1957: (8)          type is preserved (e.g., if ``dtype(out)`` is float, the result
1958: (8)          will consist of 0.0's and 1.0's). See :ref:`ufuncs-output-type` for
more
1959: (8)          details.
1960: (4)          keepdims : bool, optional
1961: (8)          If this is set to True, the axes which are reduced are left
1962: (8)          in the result as dimensions with size one. With this option,
1963: (8)          the result will broadcast correctly against the input array.
1964: (8)          If the default value is passed, then `keepdims` will not be
1965: (8)          passed through to the `all` method of sub-classes of
1966: (8)          `ndarray`, however any non-default value will be. If the
1967: (8)          sub-class' method does not implement `keepdims` any
1968: (8)          exceptions will be raised.
1969: (4)          where : array_like of bool, optional
1970: (8)          Elements to include in checking for all `True` values.
1971: (8)          See `~numpy.ufunc.reduce` for details.
1972: (8)          .. versionadded:: 1.20.0
1973: (4)          Returns
1974: (4)          -----
1975: (4)          all : ndarray, bool
1976: (8)          A new boolean or array is returned unless `out` is specified,
1977: (8)          in which case a reference to `out` is returned.
1978: (4)          See Also
1979: (4)          -----
1980: (4)          ndarray.all : equivalent method
1981: (4)          any : Test whether any element along a given axis evaluates to True.
1982: (4)          Notes
1983: (4)          -----
1984: (4)          Not a Number (NaN), positive infinity and negative infinity
1985: (4)          evaluate to `True` because these are not equal to zero.
1986: (4)          Examples
1987: (4)          -----

```

```

1988: (4) >>> np.all([[True, False], [True, True]])
1989: (4) False
1990: (4) >>> np.all([[True, False], [True, True]], axis=0)
1991: (4) array([ True, False])
1992: (4) >>> np.all([-1, 4, 5])
1993: (4) True
1994: (4) >>> np.all([1.0, np.nan])
1995: (4) True
1996: (4) >>> np.all([[True, True], [False, True]], where=[[True], [False]])
1997: (4) True
1998: (4) >>> o=np.array(False)
1999: (4) >>> z=np.all([-1, 4, 5], out=o)
2000: (4) >>> id(z), id(o), z
2001: (4) (28293632, 28293632, array(True)) # may vary
2002: (4) """
2003: (4)     return _wrapreduction(a, np.logical_and, 'all', axis, None, out,
2004: (26)                         keepdims=keepdims, where=where)
2005: (0) def _cumsum_dispatcher(a, axis=None, dtype=None, out=None):
2006: (4)     return (a, out)
2007: (0) @array_function_dispatch(_cumsum_dispatcher)
2008: (0) def cumsum(a, axis=None, dtype=None, out=None):
2009: (4) """
2010: (4)     Return the cumulative sum of the elements along a given axis.
2011: (4) Parameters
2012: (4) -----
2013: (4) a : array_like
2014: (8)     Input array.
2015: (4) axis : int, optional
2016: (8)     Axis along which the cumulative sum is computed. The default
2017: (8)     (None) is to compute the cumsum over the flattened array.
2018: (4) dtype : dtype, optional
2019: (8)     Type of the returned array and of the accumulator in which the
2020: (8)     elements are summed. If `dtype` is not specified, it defaults
2021: (8)     to the dtype of `a`, unless `a` has an integer dtype with a
2022: (8)     precision less than that of the default platform integer. In
2023: (8)     that case, the default platform integer is used.
2024: (4) out : ndarray, optional
2025: (8)     Alternative output array in which to place the result. It must
2026: (8)     have the same shape and buffer length as the expected output
2027: (8)     but the type will be cast if necessary. See :ref:`ufuncs-output-type`
for
2028: (8)     more details.
2029: (4) Returns
2030: (4) -----
2031: (4) cumsum_along_axis : ndarray.
2032: (8)     A new array holding the result is returned unless `out` is
2033: (8)     specified, in which case a reference to `out` is returned. The
2034: (8)     result has the same size as `a`, and the same shape as `a` if
2035: (8)     `axis` is not None or `a` is a 1-d array.
2036: (4) See Also
2037: (4) -----
2038: (4) sum : Sum array elements.
2039: (4) trapz : Integration of array values using the composite trapezoidal rule.
2040: (4) diff : Calculate the n-th discrete difference along given axis.
2041: (4) Notes
2042: (4) -----
2043: (4) Arithmetic is modular when using integer types, and no error is
2044: (4) raised on overflow.
2045: (4) ``cumsum(a)[-1]`` may not be equal to ``sum(a)`` for floating-point
2046: (4) values since ``sum`` may use a pairwise summation routine, reducing
2047: (4) the roundoff-error. See `sum` for more information.
2048: (4) Examples
2049: (4) -----
2050: (4) >>> a = np.array([[1,2,3], [4,5,6]])
2051: (4) >>> a
2052: (4) array([[1, 2, 3],
2053: (11)           [4, 5, 6]])
2054: (4) >>> np.cumsum(a)
2055: (4) array([ 1,  3,  6, 10, 15, 21])

```

```

2056: (4)
2057: (4)
2058: (4)
2059: (4)
2060: (11)
2061: (4)
2062: (4)
2063: (11)
2064: (4)
2065: (4)
2066: (4)
2067: (4)
2068: (4)
2069: (4)
2070: (4)
2071: (4)
2072: (0)
2073: (4)
2074: (0)
2075: (0)
2076: (4)
2077: (4)
2078: (4)
2079: (4)
2080: (8)
2081: (8)
2082: (8)
2083: (8)
2084: (8)
2085: (8)
2086: (4)
2087: (4)
2088: (4)
2089: (8)
2090: (4)
2091: (8)
2092: (8)
2093: (8)
2094: (8)
2095: (8)
2096: (8)
2097: (4)
2098: (8)
2099: (8)
2100: (8)
2101: (4)
2102: (8)
2103: (8)
2104: (8)
2105: (8)
2106: (8)
2107: (8)
2108: (8)
2109: (8)
2110: (4)
2111: (4)
2112: (4)
2113: (8)
2114: (8)
2115: (4)
2116: (4)
2117: (4)
2118: (4)
2119: (4)
2120: (4)
2121: (4)
2122: (4)
2123: (4)
2124: (4)

>>> np.cumsum(a, dtype=float)      # specifies type of output value(s)
array([ 1.,   3.,   6.,  10.,  15.,  21.])
>>> np.cumsum(a, axis=0)         # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> np.cumsum(a, axis=1)         # sum over columns for each of the 2 rows
array([[ 1,   3,   6],
       [ 4,   9,  15]])
```cumsum(b)[-1]`` may not be equal to ``sum(b)``
>>> b = np.array([1, 2e-9, 3e-9] * 1000000)
>>> b.cumsum()[-1]
1000000.0050045159
>>> b.sum()
1000000.0050000029
"""
    return _wrapfunc(a, 'cumsum', axis=axis, dtype=dtype, out=out)
def _ptp_dispatcher(a, axis=None, out=None, keepdims=None):
    return (a, out)
@array_function_dispatch(_ptp_dispatcher)
def ptp(a, axis=None, out=None, keepdims=np._NoValue):
    """
    Range of values (maximum - minimum) along an axis.
    The name of the function comes from the acronym for 'peak to peak'.
    .. warning::
        `ptp` preserves the data type of the array. This means the
        return value for an input of signed integers with n bits
        (e.g. `np.int8`, `np.int16`, etc) is also a signed integer
        with n bits. In that case, peak-to-peak values greater than
        ``2**n-1`` will be returned as negative values. An example
        with a work-around is shown below.
    Parameters
    -----
    a : array_like
        Input values.
    axis : None or int or tuple of ints, optional
        Axis along which to find the peaks. By default, flatten the
        array. `axis` may be negative, in
        which case it counts from the last to the first axis.
        .. versionadded:: 1.15.0
        If this is a tuple of ints, a reduction is performed on multiple
        axes, instead of a single axis or all the axes as before.
    out : array_like
        Alternative output array in which to place the result. It must
        have the same shape and buffer length as the expected output,
        but the type of the output values will be cast if necessary.
    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.
        If the default value is passed, then `keepdims` will not be
        passed through to the `ptp` method of sub-classes of
        `ndarray`, however any non-default value will be. If the
        sub-class' method does not implement `keepdims` any
        exceptions will be raised.
    Returns
    -----
    ptp : ndarray or scalar
        The range of a given array - `scalar` if array is one-dimensional
        or a new array holding the result along the given axis
    Examples
    -----
    >>> x = np.array([[4, 9, 2, 10],
                      ...           [6, 9, 7, 12]])
    >>> np.ptp(x, axis=1)
    array([8, 6])
    >>> np.ptp(x, axis=0)
    array([2, 0, 5, 2])
    >>> np.ptp(x)
    10

```

```

2125: (4) This example shows that a negative value can be returned when
2126: (4) the input is an array of signed integers.
2127: (4) >>> y = np.array([[1, 127],
2128: (4) ... [0, 127],
2129: (4) ... [-1, 127],
2130: (4) ... [-2, 127]], dtype=np.int8)
2131: (4) >>> np.ptp(y, axis=1)
2132: (4) array([ 126, 127, -128, -127], dtype=int8)
2133: (4) A work-around is to use the `view()` method to view the result as
2134: (4) unsigned integers with the same bit width:
2135: (4) >>> np.ptp(y, axis=1).view(np.uint8)
2136: (4) array([126, 127, 128, 129], dtype=uint8)
2137: (4) """
2138: (4)     kwargs = {}
2139: (4)     if keepdims is not np._NoValue:
2140: (8)         kwargs['keepdims'] = keepdims
2141: (4)     if type(a) is not mu.ndarray:
2142: (8)         try:
2143: (12)             ptp = a.ptp
2144: (8)         except AttributeError:
2145: (12)             pass
2146: (8)         else:
2147: (12)             return ptp(axis=axis, out=out, **kwargs)
2148: (4)     return _methods._ptp(a, axis=axis, out=out, **kwargs)
2149: (0) def _max_dispatcher(a, axis=None, out=None, keepdims=None, initial=None,
2150: (20)           where=None):
2151: (4)     return (a, out)
2152: (0) @array_function_dispatch(_max_dispatcher)
2153: (0) @set_module('numpy')
2154: (0) def max(a, axis=None, out=None, keepdims=np._NoValue, initial=np._NoValue,
2155: (9)           where=np._NoValue):
2156: (4) """
2157: (4)     Return the maximum of an array or maximum along an axis.
2158: (4) Parameters
2159: (4) -----
2160: (4)     a : array_like
2161: (8)         Input data.
2162: (4)     axis : None or int or tuple of ints, optional
2163: (8)         Axis or axes along which to operate. By default, flattened input is
2164: (8)         used.
2165: (8)         .. versionadded:: 1.7.0
2166: (8)         If this is a tuple of ints, the maximum is selected over multiple
2167: (8)         axes,
2168: (4)         instead of a single axis or all the axes as before.
2169: (8)     out : ndarray, optional
2170: (8)         Alternative output array in which to place the result. Must
2171: (8)         be of the same shape and buffer length as the expected output.
2172: (8)         See :ref:`ufuncs-output-type` for more details.
2173: (4)     keepdims : bool, optional
2174: (8)         If this is set to True, the axes which are reduced are left
2175: (8)         in the result as dimensions with size one. With this option,
2176: (8)         the result will broadcast correctly against the input array.
2177: (8)         If the default value is passed, then `keepdims` will not be
2178: (8)         passed through to the ``max`` method of sub-classes of
2179: (8)         `ndarray`, however any non-default value will be. If the
2180: (8)         sub-class' method does not implement `keepdims` any
2181: (4)         exceptions will be raised.
2182: (8)     initial : scalar, optional
2183: (8)         The minimum value of an output element. Must be present to allow
2184: (8)         computation on empty slice. See `~numpy.ufunc.reduce` for details.
2185: (4)         .. versionadded:: 1.15.0
2186: (8)     where : array_like of bool, optional
2187: (8)         Elements to compare for the maximum. See `~numpy.ufunc.reduce`
2188: (8)         for details.
2189: (4)         .. versionadded:: 1.17.0
2190: (4) Returns
2191: (4) -----
2192: (8)     max : ndarray or scalar
2193: (8)         Maximum of `a`. If `axis` is None, the result is a scalar value.

```

```

2193: (8)           If `axis` is an int, the result is an array of dimension
2194: (8)           ``a.ndim - 1``. If `axis` is a tuple, the result is an array of
2195: (8)           dimension ``a.ndim - len(axis)``.
2196: (4)           See Also
2197: (4)
2198: (4)
2199: (8)           amin :
2200: (4)           The minimum value of an array along a given axis, propagating any
2201: (8)           NaNs.
2202: (4)
2203: (8)           nanmax :
2204: (4)           The maximum value of an array along a given axis, ignoring any NaNs.
2205: (8)           maximum :
2206: (4)           Element-wise maximum of two arrays, propagating any NaNs.
2207: (4)           fmax :
2208: (4)           Element-wise maximum of two arrays, ignoring any NaNs.
2209: (4)           argmax :
2210: (4)           Return the indices of the maximum values.
2211: (4)           nanmin, minimum, fmin
2212: (4)           Notes
2213: (4)
2214: (4)           NaN values are propagated, that is if at least one item is NaN, the
2215: (4)           corresponding max value will be NaN as well. To ignore NaN values
2216: (4)           (MATLAB behavior), please use nanmax.
2217: (4)           Don't use `~numpy.max` for element-wise comparison of 2 arrays; when
2218: (4)           ``a.shape[0]`` is 2, ``maximum(a[0], a[1])`` is faster than
2219: (4)           ``max(a, axis=0)``.
2220: (4)           Examples
2221: (4)
2222: (11)
2223: (4)           >>> a = np.arange(4).reshape((2,2))
2224: (4)           >>> a
2225: (4)           array([[0, 1],
2226: (4)           [2, 3]])
2227: (4)           >>> np.max(a)           # Maximum of the flattened array
2228: (4)           3
2229: (4)           >>> np.max(a, axis=0)    # Maxima along the first axis
2230: (4)           array([2, 3])
2231: (4)           >>> np.max(a, axis=1)    # Maxima along the second axis
2232: (4)           array([1, 3])
2233: (4)           >>> np.max(a, where=[False, True], initial=-1, axis=0)
2234: (4)           array([-1, 3])
2235: (4)           >>> b = np.arange(5, dtype=float)
2236: (4)           >>> b[2] = np.NaN
2237: (4)           >>> np.max(b)
2238: (4)           nan
2239: (4)           >>> np.max(b, where=~np.isnan(b), initial=-1)
2240: (4)           4.0
2241: (4)           >>> np.nanmax(b)
2242: (4)           4.0
2243: (4)           You can use an initial value to compute the maximum of an empty slice, or
2244: (4)           to initialize it to a different value:
2245: (4)           >>> np.max([-50], [10]), axis=-1, initial=0)
2246: (4)           array([ 0, 10])
2247: (4)           Notice that the initial value is used as one of the elements for which the
2248: (4)           maximum is determined, unlike for the default argument Python's max
2249: (4)           function, which is only used for empty iterables.
2250: (4)           >>> np.max([5], initial=6)
2251: (4)           6
2252: (26)           >>> max([5], default=6)
2253: (0)           5
2254: (0)           """
2255: (9)           return _wrapreduction(a, np.maximum, 'max', axis, None, out,
2256: (4)           keepdims=keepdims, initial=initial, where=where)
2257: (4)           @array_function_dispatch(_max_dispatcher)
2258: (4)           def amax(a, axis=None, out=None, keepdims=np._NoValue, initial=np._NoValue,
2259: (4)           where=np._NoValue):
2260: (4)           """
2261: (4)           Return the maximum of an array or maximum along an axis.
2262: (4)           `amax` is an alias of `~numpy.max`.
2263: (4)           See Also
2264: (4)
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2261: (4)           max : alias of this function
2262: (4)           ndarray.max : equivalent method
2263: (4)
2264: (4)
2265: (26)          """
2266: (0)           return _wrapreduction(a, np.maximum, 'max', axis, None, out,
2267: (20)             keepdims=keepdims, initial=initial, where=where)
2268: (4)           def _min_dispatcher(a, axis=None, out=None, keepdims=None, initial=None,
2269: (0)             where=None):
2270: (0)               return (a, out)
2271: (8)           @array_function_dispatch(_min_dispatcher)
2272: (4)           def min(a, axis=None, out=None, keepdims=np._NoValue, initial=np._NoValue,
2273: (8)             where=np._NoValue):
2274: (4)               """
2275: (4)               Return the minimum of an array or minimum along an axis.
2276: (4)               Parameters
2277: (8)               -----
2278: (4)               a : array_like
2279: (8)                 Input data.
2280: (8)               axis : None or int or tuple of ints, optional
2281: (8)                 Axis or axes along which to operate. By default, flattened input is
2282: (8)                 used.
2283: (8)               .. versionadded:: 1.7.0
2284: (8)               If this is a tuple of ints, the minimum is selected over multiple
2285: (8)               axes,
2286: (8)               instead of a single axis or all the axes as before.
2287: (8)               out : ndarray, optional
2288: (8)                 Alternative output array in which to place the result. Must
2289: (8)                 be of the same shape and buffer length as the expected output.
2290: (8)                 See :ref:`ufuncs-output-type` for more details.
2291: (8)               keepdims : bool, optional
2292: (8)                 If this is set to True, the axes which are reduced are left
2293: (8)                 in the result as dimensions with size one. With this option,
2294: (8)                 the result will broadcast correctly against the input array.
2295: (8)                 If the default value is passed, then `keepdims` will not be
2296: (8)                 passed through to the ``min`` method of sub-classes of
2297: (8)                 `ndarray`, however any non-default value will be. If the
2298: (8)                 sub-class' method does not implement `keepdims` any
2299: (8)                 exceptions will be raised.
2300: (8)               initial : scalar, optional
2301: (8)                 The maximum value of an output element. Must be present to allow
2302: (8)                 computation on empty slice. See `~numpy.ufunc.reduce` for details.
2303: (8)               .. versionadded:: 1.15.0
2304: (8)               where : array_like of bool, optional
2305: (8)                 Elements to compare for the minimum. See `~numpy.ufunc.reduce`
2306: (8)                 for details.
2307: (8)               .. versionadded:: 1.17.0
2308: (4)           Returns
2309: (4)
2310: (4)           -----
2311: (4)           min : ndarray or scalar
2312: (4)             Minimum of `a`. If `axis` is None, the result is a scalar value.
2313: (4)             If `axis` is an int, the result is an array of dimension
2314: (4)               ``a.ndim - 1``. If `axis` is a tuple, the result is an array of
2315: (8)               dimension ``a.ndim - len(axis)``.
2316: (4)           See Also
2317: (4)
2318: (4)           -----
2319: (4)           amax :
2320: (4)             The maximum value of an array along a given axis, propagating any
2321: (4)             NaNs.
2322: (4)           nanmin :
2323: (4)             The minimum value of an array along a given axis, ignoring any NaNs.
2324: (4)           minimum :
2325: (4)             Element-wise minimum of two arrays, propagating any NaNs.
2326: (4)           fmin :
2327: (4)             Element-wise minimum of two arrays, ignoring any NaNs.

```

Notes

-----

NaN values are propagated, that is if at least one item is NaN, the

```

2328: (4) corresponding min value will be NaN as well. To ignore NaN values
2329: (4) (MATLAB behavior), please use nanmin.
2330: (4) Don't use `~numpy.min` for element-wise comparison of 2 arrays; when
2331: (4) ``a.shape[0]`` is 2, ``minimum(a[0], a[1])`` is faster than
2332: (4) ``min(a, axis=0)``.
2333: (4) Examples
2334: (4) -----
2335: (4) >>> a = np.arange(4).reshape((2,2))
2336: (4) >>> a
2337: (4) array([[0, 1],
2338: (11)     [2, 3]])
2339: (4) >>> np.min(a) # Minimum of the flattened array
2340: (4) 0
2341: (4) >>> np.min(a, axis=0) # Minima along the first axis
2342: (4) array([0, 1])
2343: (4) >>> np.min(a, axis=1) # Minima along the second axis
2344: (4) array([0, 2])
2345: (4) >>> np.min(a, where=[False, True], initial=10, axis=0)
2346: (4) array([10,  1])
2347: (4) >>> b = np.arange(5, dtype=float)
2348: (4) >>> b[2] = np.NaN
2349: (4) >>> np.min(b)
2350: (4) nan
2351: (4) >>> np.min(b, where=~np.isnan(b), initial=10)
2352: (4) 0.0
2353: (4) >>> np.nanmin(b)
2354: (4) 0.0
2355: (4) >>> np.min([-50], [10]), axis=-1, initial=0)
2356: (4) array([-50,    0])
2357: (4) Notice that the initial value is used as one of the elements for which the
2358: (4) minimum is determined, unlike for the default argument Python's max
2359: (4) function, which is only used for empty iterables.
2360: (4) Notice that this isn't the same as Python's ``default`` argument.
2361: (4) >>> np.min([6], initial=5)
2362: (4) 5
2363: (4) >>> min([6], default=5)
2364: (4) 6
2365: (4) """
2366: (4)     return _wrapreduction(a, np.minimum, 'min', axis, None, out,
2367: (26)             keepdims=keepdims, initial=initial, where=where)
2368: (0) @array_function_dispatch(_min_dispatcher)
2369: (0) def amin(a, axis=None, out=None, keepdims=np._NoValue, initial=np._NoValue,
2370: (9)     where=np._NoValue):
2371: (4) """
2372: (4)     Return the minimum of an array or minimum along an axis.
2373: (4)     `amin` is an alias of `~numpy.min`.
2374: (4)     See Also
2375: (4)     -----
2376: (4)     min : alias of this function
2377: (4)     ndarray.min : equivalent method
2378: (4)     """
2379: (4)     return _wrapreduction(a, np.minimum, 'min', axis, None, out,
2380: (26)             keepdims=keepdims, initial=initial, where=where)
2381: (0) def _prod_dispatcher(a, axis=None, dtype=None, out=None, keepdims=None,
2382: (21)     initial=None, where=None):
2383: (4)     return (a, out)
2384: (0) @array_function_dispatch(_prod_dispatcher)
2385: (0) def prod(a, axis=None, dtype=None, out=None, keepdims=np._NoValue,
2386: (9)     initial=np._NoValue, where=np._NoValue):
2387: (4) """
2388: (4)     Return the product of array elements over a given axis.
2389: (4)     Parameters
2390: (4)     -----
2391: (4)     a : array_like
2392: (8)         Input data.
2393: (4)     axis : None or int or tuple of ints, optional
2394: (8)         Axis or axes along which a product is performed. The default,
2395: (8)         axis=None, will calculate the product of all the elements in the
2396: (8)         input array. If axis is negative it counts from the last to the

```

```

2397: (8)           first axis.
2398: (8)           .. versionadded:: 1.7.0
2399: (8)           If axis is a tuple of ints, a product is performed on all of the
2400: (8)           axes specified in the tuple instead of a single axis or all the
2401: (8)           axes as before.
2402: (4)           dtype : dtype, optional
2403: (8)           The type of the returned array, as well as of the accumulator in
2404: (8)           which the elements are multiplied. The dtype of `a` is used by
2405: (8)           default unless `a` has an integer dtype of less precision than the
2406: (8)           default platform integer. In that case, if `a` is signed then the
2407: (8)           platform integer is used while if `a` is unsigned then an unsigned
2408: (8)           integer of the same precision as the platform integer is used.
2409: (4)           out : ndarray, optional
2410: (8)           Alternative output array in which to place the result. It must have
2411: (8)           the same shape as the expected output, but the type of the output
2412: (8)           values will be cast if necessary.
2413: (4)           keepdims : bool, optional
2414: (8)           If this is set to True, the axes which are reduced are left in the
2415: (8)           result as dimensions with size one. With this option, the result
2416: (8)           will broadcast correctly against the input array.
2417: (8)           If the default value is passed, then `keepdims` will not be
2418: (8)           passed through to the `prod` method of sub-classes of
2419: (8)           `ndarray`, however any non-default value will be. If the
2420: (8)           sub-class' method does not implement `keepdims` any
2421: (8)           exceptions will be raised.
2422: (4)           initial : scalar, optional
2423: (8)           The starting value for this product. See `~numpy.ufunc.reduce` for
details.
2424: (8)           .. versionadded:: 1.15.0
2425: (4)           where : array_like of bool, optional
2426: (8)           Elements to include in the product. See `~numpy.ufunc.reduce` for
details.
2427: (8)           .. versionadded:: 1.17.0
2428: (4)           Returns
2429: (4)           -----
2430: (4)           product_along_axis : ndarray, see `dtype` parameter above.
2431: (8)           An array shaped as `a` but with the specified axis removed.
2432: (8)           Returns a reference to `out` if specified.
2433: (4)           See Also
2434: (4)           -----
2435: (4)           ndarray.prod : equivalent method
2436: (4)           :ref:`ufuncs-output-type`
2437: (4)           Notes
2438: (4)           -----
2439: (4)           Arithmetic is modular when using integer types, and no error is
2440: (4)           raised on overflow. That means that, on a 32-bit platform:
2441: (4)           >>> x = np.array([536870910, 536870910, 536870910, 536870910])
2442: (4)
2443: (4)           16 # may vary
2444: (4)           The product of an empty array is the neutral element 1:
2445: (4)           >>> np.prod([])
2446: (4)           1.0
2447: (4)           Examples
2448: (4)           -----
2449: (4)           By default, calculate the product of all elements:
2450: (4)           >>> np.prod([1., 2.])
2451: (4)           2.0
2452: (4)           Even when the input array is two-dimensional:
2453: (4)           >>> a = np.array([[1., 2.], [3., 4.]])
2454: (4)           >>> np.prod(a)
2455: (4)           24.0
2456: (4)           But we can also specify the axis over which to multiply:
2457: (4)           >>> np.prod(a, axis=1)
2458: (4)           array([ 2., 12.])
2459: (4)           >>> np.prod(a, axis=0)
2460: (4)           array([3., 8.])
2461: (4)           Or select specific elements to include:
2462: (4)           >>> np.prod([1., np.nan, 3.], where=[True, False, True])
2463: (4)           3.0

```

```

2464: (4)           If the type of `x` is unsigned, then the output type is
2465: (4)           the unsigned platform integer:
2466: (4)           >>> x = np.array([1, 2, 3], dtype=np.uint8)
2467: (4)           >>> np.prod(x).dtype == np.uint
2468: (4)           True
2469: (4)           If `x` is of a signed integer type, then the output type
2470: (4)           is the default platform integer:
2471: (4)           >>> x = np.array([1, 2, 3], dtype=np.int8)
2472: (4)           >>> np.prod(x).dtype == int
2473: (4)           True
2474: (4)           You can also start the product with a value other than one:
2475: (4)           >>> np.prod([1, 2], initial=5)
2476: (4)           10
2477: (4)           """
2478: (4)           return _wrapreduction(a, np.multiply, 'prod', axis, dtype, out,
2479: (26)                 keepdims=keepdims, initial=initial, where=where)
2480: (0)           def _cumprod_dispatcher(a, axis=None, dtype=None, out=None):
2481: (4)               return (a, out)
2482: (0)           @array_function_dispatch(_cumprod_dispatcher)
2483: (0)           def cumprod(a, axis=None, dtype=None, out=None):
2484: (4)               """
2485: (4)               Return the cumulative product of elements along a given axis.
2486: (4)               Parameters
2487: (4)               -----
2488: (4)               a : array_like
2489: (8)                   Input array.
2490: (4)               axis : int, optional
2491: (8)                   Axis along which the cumulative product is computed. By default
2492: (8)                   the input is flattened.
2493: (4)               dtype : dtype, optional
2494: (8)                   Type of the returned array, as well as of the accumulator in which
2495: (8)                   the elements are multiplied. If *dtype* is not specified, it
2496: (8)                   defaults to the dtype of `a`, unless `a` has an integer dtype with
2497: (8)                   a precision less than that of the default platform integer. In
2498: (8)                   that case, the default platform integer is used instead.
2499: (4)               out : ndarray, optional
2500: (8)                   Alternative output array in which to place the result. It must
2501: (8)                   have the same shape and buffer length as the expected output
2502: (8)                   but the type of the resulting values will be cast if necessary.
2503: (4)               Returns
2504: (4)               -----
2505: (4)               cumprod : ndarray
2506: (8)                   A new array holding the result is returned unless `out` is
2507: (8)                   specified, in which case a reference to out is returned.
2508: (4)               See Also
2509: (4)               -----
2510: (4)               :ref:`ufuncs-output-type`
2511: (4)               Notes
2512: (4)               -----
2513: (4)               Arithmetic is modular when using integer types, and no error is
2514: (4)               raised on overflow.
2515: (4)               Examples
2516: (4)               -----
2517: (4)               >>> a = np.array([1,2,3])
2518: (4)               >>> np.cumprod(a) # intermediate results 1, 1*2
2519: (4)               ...
2520: (4)               # total product 1*2*3 = 6
2521: (4)               array([1, 2, 6])
2522: (4)               >>> a = np.array([[1, 2, 3], [4, 5, 6]])
2523: (4)               >>> np.cumprod(a, dtype=float) # specify type of output
2524: (4)               array([[ 1.,   2.,   6.,  24., 120., 720.]])
2525: (4)               The cumulative product for each column (i.e., over the rows) of `a`:
2526: (4)               >>> np.cumprod(a, axis=0)
2527: (4)               array([[ 1,   2,   3],
2528: (11)                  [ 4,  10, 18]])
2529: (4)               The cumulative product for each row (i.e. over the columns) of `a`:
2530: (4)               >>> np.cumprod(a, axis=1)
2531: (4)               array([[ 1,   2,   6],
2532: (11)                  [ 4,  20, 120]])
2533: (4)               """

```

```

2533: (4)             return _wrapfunc(a, 'cumprod', axis=axis, dtype=dtype, out=out)
2534: (0)             def _ndim_dispatcher(a):
2535: (4)                 return (a,)
2536: (0)             @array_function_dispatch(_ndim_dispatcher)
2537: (0)             def ndim(a):
2538: (4)                 """
2539: (4)                     Return the number of dimensions of an array.
2540: (4)             Parameters
2541: (4)                 -----
2542: (4)                 a : array_like
2543: (8)                     Input array. If it is not already an ndarray, a conversion is
2544: (8)                     attempted.
2545: (4)             Returns
2546: (4)                 -----
2547: (4)                 number_of_dimensions : int
2548: (8)                     The number of dimensions in `a`. Scalars are zero-dimensional.
2549: (4)             See Also
2550: (4)                 -----
2551: (4)                 ndarray.ndim : equivalent method
2552: (4)                 shape : dimensions of array
2553: (4)                 ndarray.shape : dimensions of array
2554: (4)             Examples
2555: (4)                 -----
2556: (4)                 >>> np.ndim([[1,2,3],[4,5,6]])
2557: (4)                 2
2558: (4)                 >>> np.ndim(np.array([[1,2,3],[4,5,6]]))
2559: (4)                 2
2560: (4)                 >>> np.ndim(1)
2561: (4)                 0
2562: (4)                 """
2563: (4)             try:
2564: (8)                 return a.ndim
2565: (4)             except AttributeError:
2566: (8)                 return asarray(a).ndim
2567: (0)             def _size_dispatcher(a, axis=None):
2568: (4)                 return (a,)
2569: (0)             @array_function_dispatch(_size_dispatcher)
2570: (0)             def size(a, axis=None):
2571: (4)                 """
2572: (4)                     Return the number of elements along a given axis.
2573: (4)             Parameters
2574: (4)                 -----
2575: (4)                 a : array_like
2576: (8)                     Input data.
2577: (4)                 axis : int, optional
2578: (8)                     Axis along which the elements are counted. By default, give
2579: (8)                     the total number of elements.
2580: (4)             Returns
2581: (4)                 -----
2582: (4)                 element_count : int
2583: (8)                     Number of elements along the specified axis.
2584: (4)             See Also
2585: (4)                 -----
2586: (4)                 shape : dimensions of array
2587: (4)                 ndarray.shape : dimensions of array
2588: (4)                 ndarray.size : number of elements in array
2589: (4)             Examples
2590: (4)                 -----
2591: (4)                 >>> a = np.array([[1,2,3],[4,5,6]])
2592: (4)                 >>> np.size(a)
2593: (4)                 6
2594: (4)                 >>> np.size(a,1)
2595: (4)                 3
2596: (4)                 >>> np.size(a,0)
2597: (4)                 2
2598: (4)                 """
2599: (4)                 if axis is None:
2600: (8)                     try:
2601: (12)                         return a.size

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2602: (8)
2603: (12)
2604: (4)
2605: (8)
2606: (12)
2607: (8)
2608: (12)
2609: (0)
2610: (4)
2611: (0)
2612: (0)
2613: (4)
2614: (4)
2615: (4)
2616: (4)
2617: (4)
2618: (8)
2619: (4)
2620: (8)
2621: (8)
2622: (8)
2623: (4)
2624: (8)
2625: (8)
2626: (8)
more
2627: (8)
2628: (4)
2629: (4)
2630: (4)
2631: (8)
2632: (8)
2633: (8)
2634: (8)
2635: (8)
2636: (4)
2637: (4)
2638: (4)
2639: (4)
2640: (4)
2641: (4)
2642: (4)
2643: (4)
2644: (4)
2645: (4)
2646: (4)
2647: (4)
2648: (4)
2649: (4)
2650: (4)
2651: (4)
2652: (8)
2653: (8)
2654: (4)
is
2655: (4)
2656: (4)
2657: (8)
2658: (8)
2659: (4)
2660: (4)
2661: (4)
2662: (4)
2663: (4)
2664: (8)
2665: (8)
2666: (8)
16.0549999999999997
2667: (8)

```

except AttributeError:  
 return asarray(a).size  
else:  
 try:  
 return a.shape[axis]  
 except AttributeError:  
 return asarray(a).shape[axis]

def \_round\_dispatcher(a, decimals=None, out=None):  
 return (a, out)

@array\_function\_dispatch(\_round\_dispatcher)

def round(a, decimals=0, out=None):  
 """  
 Evenly round to the given number of decimals.  
 Parameters  
 -----  
 a : array\_like  
 Input data.  
 decimals : int, optional  
 Number of decimal places to round to (default: 0). If  
 decimals is negative, it specifies the number of positions to  
 the left of the decimal point.  
 out : ndarray, optional  
 Alternative output array in which to place the result. It must have  
 the same shape as the expected output, but the type of the output  
 values will be cast if necessary. See :ref:`ufuncs-output-type` for  
 more  
 details.  
 Returns  
 -----  
 rounded\_array : ndarray  
 An array of the same type as `a`, containing the rounded values.  
 Unless `out` was specified, a new array is created. A reference to  
 the result is returned.  
 The real and imaginary parts of complex numbers are rounded  
 separately. The result of rounding a float is a float.  
 See Also  
 -----  
 ndarray.round : equivalent method  
 around : an alias for this function  
 ceil, fix, floor, rint, trunc  
 Notes  
 -----  
 For values exactly halfway between rounded decimal values, NumPy  
 rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0,  
 -0.5 and 0.5 round to 0.0, etc.  
 ``np.round`` uses a fast but sometimes inexact algorithm to round  
 floating-point datatypes. For positive `decimals` it is equivalent to  
 ``np.true\_divide(np.rint(a \* 10\*\*decimals), 10\*\*decimals)``  
 , which has  
 error due to the inexact representation of decimal fractions in the IEEE  
 floating point standard [1]\_ and errors introduced when scaling by powers  
 of ten. For instance, note the extra "1" in the following:  
 >>> np.round(56294995342131.5, 3)  
 56294995342131.51  
 If your goal is to print such values with a fixed number of decimals, it  
 is  
 preferable to use numpy's float printing routines to limit the number of  
 printed decimals:  
 >>> np.format\_float\_positional(56294995342131.5, precision=3)  
 '56294995342131.5'  
 The float printing routines use an accurate but much more computationally  
 demanding algorithm to compute the number of digits after the decimal  
 point.  
 Alternatively, Python's builtin `round` function uses a more accurate  
 but slower algorithm for 64-bit floating point values:  
 >>> round(56294995342131.5, 3)  
 56294995342131.5  
 >>> np.round(16.055, 2), round(16.055, 2) # equals  
 (16.06, 16.05)

```

2668: (4)             References
2669: (4)
2670: (4)
2671: (11)
2672: (4)
2673: (4)
2674: (4)
2675: (4)
2676: (4)
2677: (4)
2678: (4)
2679: (4)
2680: (4)
2681: (4)
2682: (4)
2683: (4)
2684: (4)
2685: (4)             .. [1] "Lecture Notes on the Status of IEEE 754", William Kahan,
2686: (0)               https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF
2687: (0)
2688: (4)
2689: (4)             Examples
2690: (4)
2691: (4)
2692: (4)
2693: (4)
2694: (4)
2695: (4)
2696: (4)
2697: (4)
2698: (0)
2699: (21)
2700: (4)
2701: (0)
2702: (0)
2703: (9)
2704: (4)
2705: (4)
2706: (4)
2707: (4)
2708: (4)
2709: (4)
2710: (4)
2711: (4)
2712: (8)
2713: (8)
2714: (4)
2715: (8)
2716: (8)
2717: (8)
2718: (8)
2719: (8)
2720: (4)
2721: (8)
2722: (8)
2723: (8)
2724: (4)
2725: (8)
2726: (8)
2727: (8)
2728: (8)
2729: (4)
2730: (8)
2731: (8)
2732: (8)
2733: (8)
2734: (8)
2735: (8)
2736: (8)

      array([0., 2.])
      array([0.4, 1.6])
      array([0., 2., 2., 4., 4.])
      array([ 1,  2,  3, 11])
      array([ 1,  2,  3, 11])
      array([ 0,  0,  0, 10])

      """
      return _wrapfunc(a, 'round', decimals=decimals, out=out)
@array_function_dispatch(_round_dispatcher)
def around(a, decimals=0, out=None):
    """
    Round an array to the given number of decimals.
    `around` is an alias of `~numpy.round`.
    See Also
    -----
    ndarray.round : equivalent method
    round : alias for this function
    ceil, fix, floor, rint, trunc
    """
    return _wrapfunc(a, 'round', decimals=decimals, out=out)
def _mean_dispatcher(a, axis=None, dtype=None, out=None, *, where=None):
    return (a, where, out)
@array_function_dispatch(_mean_dispatcher)
def mean(a, axis=None, dtype=None, out=None, keepdims=np._NoValue, *, where=np._NoValue):
    """
    Compute the arithmetic mean along the specified axis.
    Returns the average of the array elements. The average is taken over
    the flattened array by default, otherwise over the specified axis.
    `float64` intermediate and return values are used for integer inputs.
    Parameters
    -----
    a : array_like
        Array containing numbers whose mean is desired. If `a` is not an
        array, a conversion is attempted.
    axis : None or int or tuple of ints, optional
        Axis or axes along which the means are computed. The default is to
        compute the mean of the flattened array.
        .. versionadded:: 1.7.0
        If this is a tuple of ints, a mean is performed over multiple axes,
        instead of a single axis or all the axes as before.
    dtype : data-type, optional
        Type to use in computing the mean. For integer inputs, the default
        is `float64`; for floating point inputs, it is the same as the
        input dtype.
    out : ndarray, optional
        Alternate output array in which to place the result. The default
        is ``None``; if provided, it must have the same shape as the
        expected output, but the type will be cast if necessary.
        See :ref:`ufuncs-output-type` for more details.
    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.
        If the default value is passed, then `keepdims` will not be
        passed through to the `mean` method of sub-classes of
        `ndarray`, however any non-default value will be. If the
        sub-class' method does not implement `keepdims` any

```

```

2737: (8)             exceptions will be raised.
2738: (4)             where : array_like of bool, optional
2739: (8)                 Elements to include in the mean. See `~numpy.ufunc.reduce` for
details.
2740: (8)                 .. versionadded:: 1.20.0
2741: (4)             Returns
2742: (4)             -----
2743: (4)             m : ndarray, see dtype parameter above
2744: (8)                 If `out=None`, returns a new array containing the mean values,
2745: (8)                 otherwise a reference to the output array is returned.
2746: (4)             See Also
2747: (4)             -----
2748: (4)             average : Weighted average
2749: (4)             std, var, nanmean, nanstd, nanvar
2750: (4)             Notes
2751: (4)             -----
2752: (4)                 The arithmetic mean is the sum of the elements along the axis divided
2753: (4)                 by the number of elements.
2754: (4)                 Note that for floating-point input, the mean is computed using the
2755: (4)                 same precision the input has. Depending on the input data, this can
2756: (4)                 cause the results to be inaccurate, especially for `float32` (see
2757: (4)                 example below). Specifying a higher-precision accumulator using the
2758: (4)                 `dtype` keyword can alleviate this issue.
2759: (4)                 By default, `float16` results are computed using `float32` intermediates
2760: (4)                 for extra precision.
2761: (4)             Examples
2762: (4)             -----
2763: (4)             >>> a = np.array([[1, 2], [3, 4]])
2764: (4)             >>> np.mean(a)
2765: (4)             2.5
2766: (4)             >>> np.mean(a, axis=0)
2767: (4)             array([2., 3.])
2768: (4)             >>> np.mean(a, axis=1)
2769: (4)             array([1.5, 3.5])
2770: (4)             In single precision, `mean` can be inaccurate:
2771: (4)             >>> a = np.zeros((2, 512*512), dtype=np.float32)
2772: (4)             >>> a[0, :] = 1.0
2773: (4)             >>> a[1, :] = 0.1
2774: (4)             >>> np.mean(a)
2775: (4)             0.54999924
2776: (4)             Computing the mean in float64 is more accurate:
2777: (4)             >>> np.mean(a, dtype=np.float64)
2778: (4)             0.55000000074505806 # may vary
2779: (4)             Specifying a where argument:
2780: (4)             >>> a = np.array([[5, 9, 13], [14, 10, 12], [11, 15, 19]])
2781: (4)             >>> np.mean(a)
2782: (4)             12.0
2783: (4)             >>> np.mean(a, where=[[True], [False], [False]])
2784: (4)             9.0
2785: (4)             """
2786: (4)             kwargs = {}
2787: (4)             if keepdims is not np._NoValue:
2788: (8)                 kwargs['keepdims'] = keepdims
2789: (4)             if where is not np._NoValue:
2790: (8)                 kwargs['where'] = where
2791: (4)             if type(a) is not mu.ndarray:
2792: (8)                 try:
2793: (12)                     mean = a.mean
2794: (8)                 except AttributeError:
2795: (12)                     pass
2796: (8)                 else:
2797: (12)                     return mean(axis=axis, dtype=dtype, out=out, **kwargs)
2798: (4)             return _methods._mean(a, axis=axis, dtype=dtype,
2799: (26)                             out=out, **kwargs)
2800: (0)             def _std_dispatcher(a, axis=None, dtype=None, out=None, ddof=None,
2801: (20)                             keepdims=None, *, where=None):
2802: (4)                 return (a, where, out)
2803: (0)             @array_function_dispatch(_std_dispatcher)
2804: (0)             def std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=np._NoValue, *,

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2805: (8)           where=np._NoValue):
2806: (4)           """
2807: (4)           Compute the standard deviation along the specified axis.
2808: (4)           Returns the standard deviation, a measure of the spread of a distribution,
2809: (4)           of the array elements. The standard deviation is computed for the
2810: (4)           flattened array by default, otherwise over the specified axis.
2811: (4)           Parameters
2812: (4)           -----
2813: (4)           a : array_like
2814: (8)           Calculate the standard deviation of these values.
2815: (4)           axis : None or int or tuple of ints, optional
2816: (8)           Axis or axes along which the standard deviation is computed. The
2817: (8)           default is to compute the standard deviation of the flattened array.
2818: (8)           .. versionadded:: 1.7.0
2819: (8)           If this is a tuple of ints, a standard deviation is performed over
2820: (8)           multiple axes, instead of a single axis or all the axes as before.
2821: (4)           dtype : dtype, optional
2822: (8)           Type to use in computing the standard deviation. For arrays of
2823: (8)           integer type the default is float64, for arrays of float types it is
2824: (8)           the same as the array type.
2825: (4)           out : ndarray, optional
2826: (8)           Alternative output array in which to place the result. It must have
2827: (8)           the same shape as the expected output but the type (of the calculated
2828: (8)           values) will be cast if necessary.
2829: (4)           ddof : int, optional
2830: (8)           Means Delta Degrees of Freedom. The divisor used in calculations
2831: (8)           is ``N - ddof``, where ``N`` represents the number of elements.
2832: (8)           By default `ddof` is zero.
2833: (4)           keepdims : bool, optional
2834: (8)           If this is set to True, the axes which are reduced are left
2835: (8)           in the result as dimensions with size one. With this option,
2836: (8)           the result will broadcast correctly against the input array.
2837: (8)           If the default value is passed, then `keepdims` will not be
2838: (8)           passed through to the `std` method of sub-classes of
2839: (8)           `ndarray`, however any non-default value will be. If the
2840: (8)           sub-class' method does not implement `keepdims` any
2841: (8)           exceptions will be raised.
2842: (4)           where : array_like of bool, optional
2843: (8)           Elements to include in the standard deviation.
2844: (8)           See `~numpy.ufunc.reduce` for details.
2845: (8)           .. versionadded:: 1.20.0
2846: (4)           Returns
2847: (4)           -----
2848: (4)           standard_deviation : ndarray, see dtype parameter above.
2849: (8)           If `out` is None, return a new array containing the standard
2850: (8)           otherwise return a reference to the output array.
2851: (4)           See Also
2852: (4)           -----
2853: (4)           var, mean, nanmean, nanstd, nanvar
2854: (4)           :ref:`ufuncs-output-type`
2855: (4)           Notes
2856: (4)           -----
2857: (4)           The standard deviation is the square root of the average of the squared
2858: (4)           deviations from the mean, i.e., ``std = sqrt(mean(x))``, where
2859: (4)           ``x = abs(a - a.mean())**2``.
2860: (4)           The average squared deviation is typically calculated as ``x.sum() / N``,
2861: (4)           where ``N = len(x)``. If, however, `ddof` is specified, the divisor
2862: (4)           ``N - ddof`` is used instead. In standard statistical practice, ``ddof=1``
2863: (4)           provides an unbiased estimator of the variance of the infinite population.
2864: (4)           ``ddof=0`` provides a maximum likelihood estimate of the variance for
2865: (4)           normally distributed variables. The standard deviation computed in this
2866: (4)           function is the square root of the estimated variance, so even with
2867: (4)           ``ddof=1``, it will not be an unbiased estimate of the standard deviation
2868: (4)           per se.
2869: (4)           Note that, for complex numbers, `std` takes the absolute
2870: (4)           value before squaring, so that the result is always real and nonnegative.
2871: (4)           For floating-point input, the *std* is computed using the same
2872: (4)           precision the input has. Depending on the input data, this can cause

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2873: (4)          the results to be inaccurate, especially for float32 (see example below).
2874: (4)          Specifying a higher-accuracy accumulator using the `dtype` keyword can
2875: (4)          alleviate this issue.
2876: (4)          Examples
2877: (4)          -----
2878: (4)          >>> a = np.array([[1, 2], [3, 4]])
2879: (4)          >>> np.std(a)
2880: (4)          1.1180339887498949 # may vary
2881: (4)          >>> np.std(a, axis=0)
2882: (4)          array([1., 1.])
2883: (4)          >>> np.std(a, axis=1)
2884: (4)          array([0.5, 0.5])
2885: (4)          In single precision, std() can be inaccurate:
2886: (4)          >>> a = np.zeros((2, 512*512), dtype=np.float32)
2887: (4)          >>> a[0, :] = 1.0
2888: (4)          >>> a[1, :] = 0.1
2889: (4)          >>> np.std(a)
2890: (4)          0.45000005
2891: (4)          Computing the standard deviation in float64 is more accurate:
2892: (4)          >>> np.std(a, dtype=np.float64)
2893: (4)          0.44999999925494177 # may vary
2894: (4)          Specifying a where argument:
2895: (4)          >>> a = np.array([[14, 8, 11, 10], [7, 9, 10, 11], [10, 15, 5, 10]])
2896: (4)          >>> np.std(a)
2897: (4)          2.614064523559687 # may vary
2898: (4)          >>> np.std(a, where=[[True], [True], [False]])
2899: (4)          2.0
2900: (4)          """
2901: (4)          kwargs = {}
2902: (4)          if keepdims is not np._NoValue:
2903: (8)              kwargs['keepdims'] = keepdims
2904: (4)          if where is not np._NoValue:
2905: (8)              kwargs['where'] = where
2906: (4)          if type(a) is not mu.ndarray:
2907: (8)              try:
2908: (12)                  std = a.std
2909: (8)              except AttributeError:
2910: (12)                  pass
2911: (8)              else:
2912: (12)                  return std(axis=axis, dtype=dtype, out=out, ddof=ddof, **kwargs)
2913: (4)          return _methods._std(a, axis=axis, dtype=dtype, out=out, ddof=ddof,
2914: (25)                      **kwargs)
2915: (0)          def _var_dispatcher(a, axis=None, dtype=None, out=None, ddof=None,
2916: (20)                      keepdims=None, *, where=None):
2917: (4)              return (a, where, out)
2918: (0)          @array_function_dispatch(_var_dispatcher)
2919: (0)          def var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=np._NoValue, *,
2920: (8)                      where=np._NoValue):
2921: (4)                      """
2922: (4)              Compute the variance along the specified axis.
2923: (4)              Returns the variance of the array elements, a measure of the spread of a
2924: (4)              distribution. The variance is computed for the flattened array by
2925: (4)              default, otherwise over the specified axis.
2926: (4)          Parameters
2927: (4)          -----
2928: (4)          a : array_like
2929: (8)              Array containing numbers whose variance is desired. If `a` is not an
2930: (8)              array, a conversion is attempted.
2931: (4)          axis : None or int or tuple of ints, optional
2932: (8)              Axis or axes along which the variance is computed. The default is to
2933: (8)              compute the variance of the flattened array.
2934: (8)              .. versionadded:: 1.7.0
2935: (8)              If this is a tuple of ints, a variance is performed over multiple
2936: (8)              axes,
2937: (4)              instead of a single axis or all the axes as before.
2938: (8)          dtype : data-type, optional
2939: (8)              Type to use in computing the variance. For arrays of integer type
2940: (8)              the default is `float64`; for arrays of float types it is the same as

```

```

2941: (4)          out : ndarray, optional
2942: (8)            Alternate output array in which to place the result. It must have
2943: (8)            the same shape as the expected output, but the type is cast if
2944: (8)            necessary.
2945: (4)          ddof : int, optional
2946: (8)            "Delta Degrees of Freedom": the divisor used in the calculation is
2947: (8)            ``N - ddof``, where ``N`` represents the number of elements. By
2948: (8)            default `ddof` is zero.
2949: (4)          keepdims : bool, optional
2950: (8)            If this is set to True, the axes which are reduced are left
2951: (8)            in the result as dimensions with size one. With this option,
2952: (8)            the result will broadcast correctly against the input array.
2953: (8)            If the default value is passed, then `keepdims` will not be
2954: (8)            passed through to the `var` method of sub-classes of
2955: (8)            `ndarray`, however any non-default value will be. If the
2956: (8)            sub-class' method does not implement `keepdims` any
2957: (8)            exceptions will be raised.
2958: (4)          where : array_like of bool, optional
2959: (8)            Elements to include in the variance. See `~numpy.ufunc.reduce` for
2960: (8)            details.
2961: (8)            .. versionadded:: 1.20.0
2962: (4)          Returns
2963: (4)          -----
2964: (4)          variance : ndarray, see dtype parameter above
2965: (8)            If ``out=None``, returns a new array containing the variance;
2966: (8)            otherwise, a reference to the output array is returned.
2967: (4)          See Also
2968: (4)          -----
2969: (4)          std, mean, nanmean, nanstd, nanvar
2970: (4)          :ref:`ufuncs-output-type`
2971: (4)          Notes
2972: (4)          -----
2973: (4)          The variance is the average of the squared deviations from the mean,
2974: (4)          i.e., ``var = mean(x)**2``. where ``x = abs(a - a.mean())**2``.
2975: (4)          The mean is typically calculated as ``x.sum() / N``, where ``N = len(x)``.
2976: (4)          If, however, `ddof` is specified, the divisor ``N - ddof`` is used
2977: (4)          instead. In standard statistical practice, ``ddof=1`` provides an
2978: (4)          unbiased estimator of the variance of a hypothetical infinite population.
2979: (4)          ``ddof=0`` provides a maximum likelihood estimate of the variance for
2980: (4)          normally distributed variables.
2981: (4)          Note that for complex numbers, the absolute value is taken before
2982: (4)          squaring, so that the result is always real and nonnegative.
2983: (4)          For floating-point input, the variance is computed using the same
2984: (4)          precision the input has. Depending on the input data, this can cause
2985: (4)          the results to be inaccurate, especially for `float32` (see example
2986: (4)          below). Specifying a higher-accuracy accumulator using the ``dtype``
2987: (4)          keyword can alleviate this issue.
2988: (4)          Examples
2989: (4)          -----
2990: (4)          >>> a = np.array([[1, 2], [3, 4]])
2991: (4)          >>> np.var(a)
2992: (4)          1.25
2993: (4)          >>> np.var(a, axis=0)
2994: (4)          array([1., 1.])
2995: (4)          >>> np.var(a, axis=1)
2996: (4)          array([0.25, 0.25])
2997: (4)          In single precision, var() can be inaccurate:
2998: (4)          >>> a = np.zeros((2, 512*512), dtype=np.float32)
2999: (4)          >>> a[0, :] = 1.0
3000: (4)          >>> a[1, :] = 0.1
3001: (4)          >>> np.var(a)
3002: (4)          0.20250003
3003: (4)          Computing the variance in float64 is more accurate:
3004: (4)          >>> np.var(a, dtype=np.float64)
3005: (4)          0.20249999932944759 # may vary
3006: (4)          >>> ((1-0.55)**2 + (0.1-0.55)**2)/2
3007: (4)          0.2025
3008: (4)          Specifying a where argument:
3009: (4)          >>> a = np.array([[14, 8, 11, 10], [7, 9, 10, 11], [10, 15, 5, 10]])

```

```

3010: (4)
3011: (4)
3012: (4)
3013: (4)
3014: (4)
3015: (4)
3016: (4)
3017: (8)
3018: (4)
3019: (8)
3020: (4)
3021: (8)
3022: (12)
3023: (8)
3024: (12)
3025: (8)
3026: (12)
3027: (4)
3028: (25)
3029: (0)
3030: (4)
3031: (18)
3032: (18)
3033: (4)
3034: (0)
3035: (0)
3036: (4)
3037: (4)
3038: (4)
3039: (4)
3040: (4)
3041: (8)
3042: (8)
3043: (4)
3044: (4)
3045: (4)
3046: (4)
3047: (4)
3048: (0)
3049: (24)
3050: (4)
3051: (18)
3052: (18)
3053: (4)
3054: (0)
3055: (0)
3056: (4)
3057: (4)
3058: (4)
3059: (8)
3060: (8)
3061: (4)
3062: (4)
3063: (4)
3064: (4)
3065: (4)
3066: (0)
3067: (4)
"
3068: (18)
3069: (18)
3070: (4)
3071: (0)
3072: (0)
3073: (4)
3074: (4)
3075: (4)
3076: (8)
3077: (8)

>>> np.var(a)
6.83333333333333 # may vary
>>> np.var(a, where=[[True], [True], [False]])
4.0
"""
kwargs = {}
if keepdims is not np._NoValue:
    kwargs['keepdims'] = keepdims
if where is not np._NoValue:
    kwargs['where'] = where
if type(a) is not mu.ndarray:
    try:
        var = a.var
    except AttributeError:
        pass
    else:
        return var(axis=axis, dtype=dtype, out=out, ddof=ddof, **kwargs)
return _methods._var(a, axis=axis, dtype=dtype, out=out, ddof=ddof,
                     **kwargs)
def _round_dispatcher(a, decimals=None, out=None):
    warnings.warn(`round_` is deprecated as of NumPy 1.25.0, and will be "
                  "removed in NumPy 2.0. Please use `round` instead.",
                  DeprecationWarning, stacklevel=3)
    return (a, out)
@array_function_dispatch(_round_dispatcher)
def round_(a, decimals=0, out=None):
    """
    Round an array to the given number of decimals.
    `~numpy.round_` is a disrecommended backwards-compatibility
    alias of `~numpy.around` and `~numpy.round` .
    .. deprecated:: 1.25.0
        ``round_`` is deprecated as of NumPy 1.25.0, and will be
        removed in NumPy 2.0. Please use `round` instead.
    See Also
    -----
    around : equivalent function; see for details.
    """
    return around(a, decimals=decimals, out=out)
def _product_dispatcher(a, axis=None, dtype=None, out=None, keepdims=None,
                       initial=None, where=None):
    warnings.warn(`product` is deprecated as of NumPy 1.25.0, and will be "
                  "removed in NumPy 2.0. Please use `prod` instead.",
                  DeprecationWarning, stacklevel=3)
    return (a, out)
@array_function_dispatch(_product_dispatcher, verify=False)
def product(*args, **kwargs):
    """
    Return the product of array elements over a given axis.
    .. deprecated:: 1.25.0
        ``product`` is deprecated as of NumPy 1.25.0, and will be
        removed in NumPy 2.0. Please use `prod` instead.
    See Also
    -----
    prod : equivalent function; see for details.
    """
    return prod(*args, **kwargs)
def _cumproduct_dispatcher(a, axis=None, dtype=None, out=None):
    warnings.warn(`cumproduct` is deprecated as of NumPy 1.25.0, and will be "
                  "removed in NumPy 2.0. Please use `cumprod` instead.",
                  DeprecationWarning, stacklevel=3)
    return (a, out)
@array_function_dispatch(_cumproduct_dispatcher, verify=False)
def cumproduct(*args, **kwargs):
    """
    Return the cumulative product over the given axis.
    .. deprecated:: 1.25.0
        ``cumproduct`` is deprecated as of NumPy 1.25.0, and will be
        removed in NumPy 2.0. Please use `cumprod` instead.
    
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3078: (4)           See Also
3079: (4)
3080: (4)
3081: (4)
3082: (4)
3083: (0)
3084: (25)
3085: (4)
3086: (18)
3087: (18)
3088: (4)
3089: (0)
3090: (0)
3091: (4)
3092: (4)
3093: (4)
3094: (4)
3095: (8)
3096: (8)
3097: (4)
3098: (4)
3099: (4)
3100: (4)
3101: (4)
3102: (0)
3103: (4)
3104: (18)
3105: (18)
3106: (4)
3107: (0)
3108: (0)
3109: (4)
3110: (4)
3111: (4)
3112: (8)
3113: (8)
3114: (4)
3115: (4)
3116: (4)
3117: (4)
3118: (4)

-----

```

#### File 49 - function\_base.py:

```

1: (0)           import functools
2: (0)           import warnings
3: (0)           import operator
4: (0)           import types
5: (0)           import numpy as np
6: (0)           from . import numeric as _nx
7: (0)           from .numeric import result_type, NaN, asanyarray, ndim
8: (0)           from numpy.core.multiarray import add_docstring
9: (0)           from numpy.core import overrides
10: (0)          __all__ = ['logspace', 'linspace', 'geomspace']
11: (0)          array_function_dispatch = functools.partial(
12: (4)            overrides.array_function_dispatch, module='numpy')
13: (0)          def _linspace_dispatcher(start, stop, num=None, endpoint=None, retstep=None,
14: (25)              dtype=None, axis=None):
15: (4)            return (start, stop)
16: (0)          @array_function_dispatch(_linspace_dispatcher)
17: (0)          def linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None,
18: (13)              axis=0):
19: (4)            """
20: (4)            Return evenly spaced numbers over a specified interval.
21: (4)            Returns `num` evenly spaced samples, calculated over the
22: (4)            interval [`start`, `stop`].
23: (4)            The endpoint of the interval can optionally be excluded.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

24: (4) .. versionchanged:: 1.16.0
25: (8)     Non-scalar `start` and `stop` are now supported.
26: (4) .. versionchanged:: 1.20.0
27: (8)     Values are rounded towards ``-inf`` instead of ``0`` when an
28: (8)     integer ``dtype`` is specified. The old behavior can
29: (8)     still be obtained with ``np.linspace(start, stop, num).astype(int)``
30: (4) Parameters
31: (4) -----
32: (4) start : array_like
33: (8)     The starting value of the sequence.
34: (4) stop : array_like
35: (8)     The end value of the sequence, unless `endpoint` is set to False.
36: (8)     In that case, the sequence consists of all but the last of ``num + 1``
37: (8)     evenly spaced samples, so that `stop` is excluded. Note that the step
38: (8)     size changes when `endpoint` is False.
39: (4) num : int, optional
40: (8)     Number of samples to generate. Default is 50. Must be non-negative.
41: (4) endpoint : bool, optional
42: (8)     If True, `stop` is the last sample. Otherwise, it is not included.
43: (8)     Default is True.
44: (4) retstep : bool, optional
45: (8)     If True, return ('samples', 'step'), where `step` is the spacing
46: (8)     between samples.
47: (4) dtype : dtype, optional
48: (8)     The type of the output array. If `dtype` is not given, the data type
49: (8)     is inferred from `start` and `stop`. The inferred dtype will never be
50: (8)     an integer; 'float' is chosen even if the arguments would produce an
51: (8)     array of integers.
52: (8)     .. versionadded:: 1.9.0
53: (4) axis : int, optional
54: (8)     The axis in the result to store the samples. Relevant only if start
55: (8)     or stop are array-like. By default (0), the samples will be along a
56: (8)     new axis inserted at the beginning. Use -1 to get an axis at the end.
57: (8)     .. versionadded:: 1.16.0
58: (4) Returns
59: (4) -----
60: (4) samples : ndarray
61: (8)     There are `num` equally spaced samples in the closed interval
62: (8)     ``[start, stop]`` or the half-open interval ``[start, stop)``
63: (8)     (depending on whether `endpoint` is True or False).
64: (4) step : float, optional
65: (8)     Only returned if `retstep` is True
66: (8)     Size of spacing between samples.
67: (4) See Also
68: (4) -----
69: (4) arange : Similar to 'linspace', but uses a step size (instead of the
70: (13)     number of samples).
71: (4) geomspace : Similar to 'linspace', but with numbers spaced evenly on a log
72: (16)     scale (a geometric progression).
73: (4) logspace : Similar to 'geomspace', but with the end points specified as
74: (15)     logarithms.
75: (4) :ref:`how-to-partition`
76: (4) Examples
77: (4) -----
78: (4) >>> np.linspace(2.0, 3.0, num=5)
79: (4) array([2. , 2.25, 2.5 , 2.75, 3. ])
80: (4) >>> np.linspace(2.0, 3.0, num=5, endpoint=False)
81: (4) array([2. , 2.2, 2.4, 2.6, 2.8])
82: (4) >>> np.linspace(2.0, 3.0, num=5, retstep=True)
83: (4) (array([2. , 2.25, 2.5 , 2.75, 3. ]), 0.25)
84: (4) Graphical illustration:
85: (4) >>> import matplotlib.pyplot as plt
86: (4) >>> N = 8
87: (4) >>> y = np.zeros(N)
88: (4) >>> x1 = np.linspace(0, 10, N, endpoint=True)
89: (4) >>> x2 = np.linspace(0, 10, N, endpoint=False)
90: (4) >>> plt.plot(x1, y, 'o')
91: (4) [<matplotlib.lines.Line2D object at 0x...>]
92: (4) >>> plt.plot(x2, y + 0.5, 'o')
```

```

93: (4)          [<matplotlib.lines.Line2D object at 0x...>]
94: (4)          >>> plt.ylim([-0.5, 1])
95: (4)          (-0.5, 1)
96: (4)          >>> plt.show()
97: (4)          """
98: (4)          num = operator.index(num)
99: (4)          if num < 0:
100: (8)             raise ValueError("Number of samples, %s, must be non-negative." % num)
101: (4)          div = (num - 1) if endpoint else num
102: (4)          start = asanyarray(start) * 1.0
103: (4)          stop = asanyarray(stop) * 1.0
104: (4)          dt = result_type(start, stop, float(num))
105: (4)          if dtype is None:
106: (8)              dtype = dt
107: (8)              integer_dtype = False
108: (4)          else:
109: (8)              integer_dtype = _nx.issubdtype(dtype, _nx.integer)
110: (4)          delta = stop - start
111: (4)          y = _nx.arange(0, num, dtype=dt).reshape((-1,) + (1,) * ndim(delta))
112: (4)          if div > 0:
113: (8)              _mult_inplace = _nx.isscalar(delta)
114: (8)              step = delta / div
115: (8)              any_step_zero = (
116: (12)                  step == 0 if _mult_inplace else _nx.asanyarray(step == 0).any())
117: (8)              if any_step_zero:
118: (12)                  y /= div
119: (12)                  if _mult_inplace:
120: (16)                      y *= delta
121: (12)                  else:
122: (16)                      y = y * delta
123: (8)                  else:
124: (12)                      if _mult_inplace:
125: (16)                          y *= step
126: (12)                      else:
127: (16)                          y = y * step
128: (4)                  else:
129: (8)                      step = NaN
130: (8)                      y = y * delta
131: (4)                      y += start
132: (4)                      if endpoint and num > 1:
133: (8)                          y[-1, ...] = stop
134: (4)                      if axis != 0:
135: (8)                          y = _nx.moveaxis(y, 0, axis)
136: (4)                      if integer_dtype:
137: (8)                          _nx.floor(y, out=y)
138: (4)                      if retstep:
139: (8)                          return y.astype(dtype, copy=False), step
140: (4)                      else:
141: (8)                          return y.astype(dtype, copy=False)
142: (0)          def _logspace_dispatcher(start, stop, num=None, endpoint=None, base=None,
143: (25)                                dtype=None, axis=None):
144: (4)              return (start, stop, base)
145: (0)          @array_function_dispatch(_logspace_dispatcher)
146: (0)          def logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None,
147: (13)                                axis=0):
148: (4)              """
149: (4)              Return numbers spaced evenly on a log scale.
150: (4)              In linear space, the sequence starts at ``base ** start```
151: (4)              (`base` to the power of `start`) and ends with ``base ** stop```
152: (4)              (see `endpoint` below).
153: (4)              .. versionchanged:: 1.16.0
154: (8)                  Non-scalar `start` and `stop` are now supported.
155: (4)              .. versionchanged:: 1.25.0
156: (8)                  Non-scalar `base` is now supported
157: (4)          Parameters
158: (-----)
159: (4)          start : array_like
160: (8)              ``base ** start`` is the starting value of the sequence.
161: (4)          stop : array_like

```

```

162: (8)           ``base ** stop`` is the final value of the sequence, unless `endpoint`  

163: (8)           is False. In that case, ``num + 1`` values are spaced over the  

164: (8)           interval in log-space, of which all but the last (a sequence of  

165: (8)           length `num`) are returned.  

166: (4)           num : integer, optional  

167: (8)           Number of samples to generate. Default is 50.  

168: (4)           endpoint : boolean, optional  

169: (8)           If true, `stop` is the last sample. Otherwise, it is not included.  

170: (8)           Default is True.  

171: (4)           base : array_like, optional  

172: (8)           The base of the log space. The step size between the elements in  

173: (8)           ``ln(samples) / ln(base)`` (or ``log_base(samples)``) is uniform.  

174: (8)           Default is 10.0.  

175: (4)           dtype : dtype  

176: (8)           The type of the output array. If `dtype` is not given, the data type  

177: (8)           is inferred from `start` and `stop`. The inferred type will never be  

178: (8)           an integer; `float` is chosen even if the arguments would produce an  

179: (8)           array of integers.  

180: (4)           axis : int, optional  

181: (8)           The axis in the result to store the samples. Relevant only if start,  

182: (8)           stop, or base are array-like. By default (0), the samples will be  

183: (8)           along a new axis inserted at the beginning. Use -1 to get an axis at  

184: (8)           the end.  

185: (8)           .. versionadded:: 1.16.0  

186: (4)           Returns  

187: (4)           -----  

188: (4)           samples : ndarray  

189: (8)           `num` samples, equally spaced on a log scale.  

190: (4)           See Also  

191: (4)           -----  

192: (4)           arange : Similar to linspace, with the step size specified instead of the  

193: (13)          number of samples. Note that, when used with a float endpoint,  

the  

194: (13)          endpoint may or may not be included.  

195: (4)           linspace : Similar to logspace, but with the samples uniformly distributed  

196: (15)          in linear space, instead of log space.  

197: (4)           geomspace : Similar to logspace, but with endpoints specified directly.  

198: (4)           :ref:`how-to-partition`  

199: (4)           Notes  

200: (4)           -----  

201: (4)           If base is a scalar, logspace is equivalent to the code  

202: (4)           >>> y = np.linspace(start, stop, num=num, endpoint=endpoint)  

203: (4)           ... # doctest: +SKIP  

204: (4)           >>> power(base, y).astype(dtype)  

205: (4)           ... # doctest: +SKIP  

206: (4)           Examples  

207: (4)           -----  

208: (4)           >>> np.logspace(2.0, 3.0, num=4)  

209: (4)           array([ 100.          ,  215.443469  ,  464.15888336,  1000.          ])  

210: (4)           >>> np.logspace(2.0, 3.0, num=4, endpoint=False)  

211: (4)           array([100.          ,  177.827941  ,  316.22776602,  562.34132519])  

212: (4)           >>> np.logspace(2.0, 3.0, num=4, base=2.0)  

213: (4)           array([4.          ,  5.0396842 ,  6.34960421,  8.          ])  

214: (4)           >>> np.logspace(2.0, 3.0, num=4, base=[2.0, 3.0], axis=-1)  

215: (4)           array([[ 4.          ,  5.0396842 ,  6.34960421,  8.          ],  

216: (11)          [ 9.          , 12.98024613, 18.72075441, 27.          ]])  

217: (4)           Graphical illustration:  

218: (4)           >>> import matplotlib.pyplot as plt  

219: (4)           >>> N = 10  

220: (4)           >>> x1 = np.logspace(0.1, 1, N, endpoint=True)  

221: (4)           >>> x2 = np.logspace(0.1, 1, N, endpoint=False)  

222: (4)           >>> y = np.zeros(N)  

223: (4)           >>> plt.plot(x1, y, 'o')  

224: (4)           [<matplotlib.lines.Line2D object at 0x...>]  

225: (4)           >>> plt.plot(x2, y + 0.5, 'o')  

226: (4)           [<matplotlib.lines.Line2D object at 0x...>]  

227: (4)           >>> plt.ylim([-0.5, 1])  

228: (4)           (-0.5, 1)  

229: (4)           >>> plt.show()

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

230: (4)
231: (4)
232: (4)
233: (8)
234: (8)
235: (4)
236: (4)
237: (4)
238: (4)
239: (8)
240: (4)
241: (0)
242: (26)
243: (4)
244: (0)
245: (0)
246: (4)
247: (4)
248: (4)
249: (4)
250: (4)
251: (8)
252: (4)
253: (4)
254: (4)
255: (8)
256: (4)
257: (8)
258: (8)
259: (8)
260: (8)
261: (4)
262: (8)
263: (4)
264: (8)
265: (8)
266: (4)
267: (8)
268: (8)
269: (8)
270: (8)
271: (4)
272: (8)
273: (8)
274: (8)
275: (8)
276: (4)
277: (4)
278: (4)
279: (8)
280: (4)
281: (4)
282: (4)
283: (15)
284: (4)
285: (15)
286: (4)
287: (13)
288: (4)
289: (4)
290: (4)
291: (4)
292: (4)
293: (4)
path.)
294: (4)
295: (4)
296: (4)
297: (4)

        """
        ndmax = np.broadcast(start, stop, base).ndim
        start, stop, base = (
            np.array(a, copy=False, subok=True, ndmin=ndmax)
            for a in (start, stop, base)
        )
        y = linspace(start, stop, num=num, endpoint=endpoint, axis=axis)
        base = np.expand_dims(base, axis=axis)
        if dtype is None:
            return _nx.power(base, y)
        return _nx.power(base, y).astype(dtype, copy=False)
    def _geomspace_dispatcher(start, stop, num=None, endpoint=None, dtype=None,
                             axis=None):
        return (start, stop)
    @array_function_dispatch(_geomspace_dispatcher)
    def geomspace(start, stop, num=50, endpoint=True, dtype=None, axis=0):
        """
        Return numbers spaced evenly on a log scale (a geometric progression).
        This is similar to `logspace`, but with endpoints specified directly.
        Each output sample is a constant multiple of the previous.
        .. versionchanged:: 1.16.0
            Non-scalar `start` and `stop` are now supported.

        Parameters
        -----
        start : array_like
            The starting value of the sequence.
        stop : array_like
            The final value of the sequence, unless `endpoint` is False.
            In that case, ``num + 1`` values are spaced over the
            interval in log-space, of which all but the last (a sequence of
            length `num`) are returned.
        num : integer, optional
            Number of samples to generate. Default is 50.
        endpoint : boolean, optional
            If true, `stop` is the last sample. Otherwise, it is not included.
            Default is True.
        dtype : dtype
            The type of the output array. If `dtype` is not given, the data type
            is inferred from `start` and `stop`. The inferred dtype will never be
            an integer; `float` is chosen even if the arguments would produce an
            array of integers.
        axis : int, optional
            The axis in the result to store the samples. Relevant only if start
            or stop are array-like. By default (0), the samples will be along a
            new axis inserted at the beginning. Use -1 to get an axis at the end.
            .. versionadded:: 1.16.0

        Returns
        -----
        samples : ndarray
            `num` samples, equally spaced on a log scale.

        See Also
        -----
        logspace : Similar to geomspace, but with endpoints specified using log
                   and base.
        linspace : Similar to geomspace, but with arithmetic instead of geometric
                   progression.
        arange : Similar to linspace, with the step size specified instead of the
                 number of samples.
        :ref:`how-to-partition`

        Notes
        -----
        If the inputs or dtype are complex, the output will follow a logarithmic
        spiral in the complex plane. (There are an infinite number of spirals
        passing through two points; the output will follow the shortest such
        path.)

        Examples
        -----
        >>> np.geomspace(1, 1000, num=4)
        array([ 1.,   10.,  100., 1000.])

```

```

298: (4)          >>> np.geomspace(1, 1000, num=3, endpoint=False)
299: (4)          array([ 1., 10., 100.])
300: (4)          >>> np.geomspace(1, 1000, num=4, endpoint=False)
301: (4)          array([ 1. , 5.62341325, 31.6227766 , 177.827941 ])
302: (4)          >>> np.geomspace(1, 256, num=9)
303: (4)          array([ 1., 2., 4., 8., 16., 32., 64., 128., 256.])
304: (4)          Note that the above may not produce exact integers:
305: (4)          >>> np.geomspace(1, 256, num=9, dtype=int)
306: (4)          array([ 1, 2, 4, 7, 16, 32, 63, 127, 256])
307: (4)          >>> np.around(np.geomspace(1, 256, num=9)).astype(int)
308: (4)          array([ 1, 2, 4, 8, 16, 32, 64, 128, 256])
309: (4)          Negative, decreasing, and complex inputs are allowed:
310: (4)          >>> np.geomspace(1000, 1, num=4)
311: (4)          array([1000., 100., 10., 1.])
312: (4)          >>> np.geomspace(-1000, -1, num=4)
313: (4)          array([-1000., -100., -10., -1.])
314: (4)          >>> np.geomspace(1j, 1000j, num=4) # Straight line
315: (4)          array([0. +1.j, 0. +10.j, 0. +100.j, 0.+1000.j])
316: (4)          >>> np.geomspace(-1+0j, 1+0j, num=5) # Circle
317: (4)          array([-1.0000000e+00+1.22464680e-16j, -7.07106781e-01+7.07106781e-01j,
318: (12)           6.12323400e-17+1.0000000e+00j, 7.07106781e-01+7.07106781e-01j,
319: (12)           1.0000000e+00+0.0000000e+00j])
320: (4)          Graphical illustration of `endpoint` parameter:
321: (4)          >>> import matplotlib.pyplot as plt
322: (4)          >>> N = 10
323: (4)          >>> y = np.zeros(N)
324: (4)          >>> plt.semilogx(np.geomspace(1, 1000, N, endpoint=True), y + 1, 'o')
325: (4)          [

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

367: (4)           return result.astype(dtype, copy=False)
368: (0)           def _needs_add_docstring(obj):
369: (4)             """
370: (4)               Returns true if the only way to set the docstring of `obj` from python is
371: (4)               via add_docstring.
372: (4)               This function errs on the side of being overly conservative.
373: (4)             """
374: (4)             Py_TPFLAGS_HEAPTYPE = 1 << 9
375: (4)             if isinstance(obj, (types.FunctionType, types.MethodType, property)):
376: (8)                 return False
377: (4)             if isinstance(obj, type) and obj.__flags__ & Py_TPFLAGS_HEAPTYPE:
378: (8)                 return False
379: (4)             return True
380: (0)           def _add_docstring(obj, doc, warn_on_python):
381: (4)             if warn_on_python and not _needs_add_docstring(obj):
382: (8)               warnings.warn(
383: (12)                 "add_newdoc was used on a pure-python object {}. "
384: (12)                 "Prefer to attach it directly to the source."
385: (12)                 .format(obj),
386: (12)                 UserWarning,
387: (12)                 stacklevel=3)
388: (4)             try:
389: (8)               add_docstring(obj, doc)
390: (4)             except Exception:
391: (8)               pass
392: (0)           def add_newdoc(place, obj, doc, warn_on_python=True):
393: (4)             """
394: (4)               Add documentation to an existing object, typically one defined in C
395: (4)               The purpose is to allow easier editing of the docstrings without requiring
396: (4)               a re-compile. This exists primarily for internal use within numpy itself.
397: (4)               Parameters
398: (4)               -----
399: (4)               place : str
400: (8)                 The absolute name of the module to import from
401: (4)               obj : str
402: (8)                 The name of the object to add documentation to, typically a class or
403: (8)                 function name
404: (4)               doc : {str, Tuple[str, str], List[Tuple[str, str]]}
405: (8)                 If a string, the documentation to apply to `obj`
406: (8)                 If a tuple, then the first element is interpreted as an attribute of
407: (8)                 `obj` and the second as the docstring to apply - ``(method,
docstring)``
408: (8)                 If a list, then each element of the list should be a tuple of length
409: (8)                   two - ``[(method1, docstring1), (method2, docstring2), ...]``
410: (4)               warn_on_python : bool
411: (8)                 If True, the default, emit `UserWarning` if this is used to attach
412: (8)                   documentation to a pure-python object.
413: (4)               Notes
414: (4)               -----
415: (4)                 This routine never raises an error if the docstring can't be written, but
416: (4)                 will raise an error if the object being documented does not exist.
417: (4)                 This routine cannot modify read-only docstrings, as appear
418: (4)                 in new-style classes or built-in functions. Because this
419: (4)                 routine never raises an error the caller must check manually
420: (4)                 that the docstrings were changed.
421: (4)                 Since this function grabs the ``char *`` from a c-level str object and
puts
422: (4)                   it into the ``tp_doc`` slot of the type of `obj`, it violates a number of
423: (4)                   C-API best-practices, by:
424: (4)                     - modifying a `PyTypeObject` after calling `PyType_Ready`
425: (4)                     - calling `Py_INCREF` on the str and losing the reference, so the str
426: (6)                       will never be released
427: (4)                     If possible it should be avoided.
428: (4)                     """
429: (4)                     new = getattr(__import__(place, globals(), {}, [obj]), obj)
430: (4)                     if isinstance(doc, str):
431: (8)                       _add_docstring(new, doc.strip(), warn_on_python)
432: (4)                     elif isinstance(doc, tuple):
433: (8)                       attr, docstring = doc

```

```

434: (8)             _add_docstring(getattr(new, attr), docstring.strip(), warn_on_python)
435: (4)             elif isinstance(doc, list):
436: (8)                 for attr, docstring in doc:
437: (12)                     _add_docstring(getattr(new, attr), docstring.strip(),
warn_on_python)

-----

```

## File 50 - getlimits.py:

```

1: (0)             """Machine limits for Float32 and Float64 and (long double) if available...
2: (0)             """
3: (0)             __all__ = ['finfo', 'iinfo']
4: (0)             import warnings
5: (0)             from .._utils import set_module
6: (0)             from ._machar import MachAr
7: (0)             from . import numeric
8: (0)             from . import numerictypes as ntypes
9: (0)             from .numeric import array, inf, NaN
10: (0)            from .umath import log10, exp2, nextafter, isnan
11: (0)            def _fr0(a):
12: (4)                """fix rank-0 --> rank-1"""
13: (4)                if a.ndim == 0:
14: (8)                    a = a.copy()
15: (8)                    a.shape = (1,)
16: (4)                return a
17: (0)            def _fr1(a):
18: (4)                """fix rank > 0 --> rank-0"""
19: (4)                if a.size == 1:
20: (8)                    a = a.copy()
21: (8)                    a.shape = ()
22: (4)                return a
23: (0)            class MachArLike:
24: (4)                """ Object to simulate MachAr instance """
25: (4)                def __init__(self, ftype, *, eps, epsneg, huge, tiny,
26: (17)                    ibeta, smallest_subnormal=None, **kwargs):
27: (8)                    self.params = _MACHAR_PARAMS[ftype]
28: (8)                    self.ftype = ftype
29: (8)                    self.title = self.params['title']
30: (8)                    if not smallest_subnormal:
31: (12)                        self._smallest_subnormal = nextafter(
32: (16)                            self.ftype(0), self.ftype(1), dtype=self.ftype)
33: (8)                    else:
34: (12)                        self._smallest_subnormal = smallest_subnormal
35: (8)                    self.epsilon = self.eps = self._float_to_float(eps)
36: (8)                    self.epsneg = self._float_to_float(epsneg)
37: (8)                    self.xmax = self.huge = self._float_to_float(huge)
38: (8)                    self.xmin = self._float_to_float(tiny)
39: (8)                    self.smallest_normal = self.tiny = self._float_to_float(tiny)
40: (8)                    self.ibeta = self.params['itype'](ibeta)
41: (8)                    self.__dict__.update(kwargs)
42: (8)                    self.precision = int(-log10(self.eps))
43: (8)                    self.resolution = self._float_to_float(
44: (12)                        self._float_conv(10) ** (-self.precision))
45: (8)                    self._str_eps = self._float_to_str(self.eps)
46: (8)                    self._str_epsneg = self._float_to_str(self.epsneg)
47: (8)                    self._str_xmin = self._float_to_str(self.xmin)
48: (8)                    self._str_xmax = self._float_to_str(self.xmax)
49: (8)                    self._str_resolution = self._float_to_str(self.resolution)
50: (8)                    self._str_smallest_normal = self._float_to_str(self.xmin)
51: (4)            @property
52: (4)            def smallest_subnormal(self):
53: (8)                """Return the value for the smallest subnormal.
54: (8)                Returns
55: (8)                -----
56: (8)                smallest_subnormal : float
57: (12)                    value for the smallest subnormal.
58: (8)                Warns
59: (8)                -----
```

```

60: (8)             UserWarning
61: (12)            If the calculated value for the smallest subnormal is zero.
62: (8)
63: (8)            """
64: (8)            value = self._smallest_subnormal
65: (12)            if self.ftype(0) == value:
66: (16)                warnings.warn(
67: (16)                    'The value of the smallest subnormal for {} type '
68: (16)                    'is zero.'.format(self.ftype), UserWarning, stacklevel=2)
69: (8)            return self._float_to_float(value)
70: (4) @property
71: (8) def _str_smallest_subnormal(self):
72: (8)     """Return the string representation of the smallest subnormal."""
73: (4)     return self._float_to_str(self.smallest_subnormal)
74: (8) def _float_to_float(self, value):
75: (8)     """Converts float to float.
76: (8)     Parameters
77: (8)     -----
78: (8)     value : float
79: (12)        value to be converted.
80: (8)     """
81: (4)     return _fr1(self._float_conv(value))
82: (8) def _float_conv(self, value):
83: (8)     """Converts float to conv.
84: (8)     Parameters
85: (8)     -----
86: (12)     value : float
87: (8)        value to be converted.
88: (8)     """
89: (4)     return array([value], self.ftype)
90: (8) def _float_to_str(self, value):
91: (8)     """Converts float to str.
92: (8)     Parameters
93: (8)     -----
94: (12)     value : float
95: (8)        value to be converted.
96: (8)     """
97: (0)     return self.params['fmt'] % array(_fr0(value)[0], self.ftype)
98: (4) _convert_to_float = {
99: (4)     ntypes.csingle: ntypes.single,
100: (4)     ntypes.complex_: ntypes.float_,
101: (4)     ntypes.clongfloat: ntypes.longfloat
102: (0) }
103: (0) _title_fmt = 'numpy {} precision floating point number'
104: (4) _MACHAR_PARAMS = {
105: (8)     ntypes.double: dict(
106: (8)         itype = ntypes.int64,
107: (8)         fmt = '%24.16e',
108: (8)         title = _title_fmt.format('double')),
109: (4)     ntypes.single: dict(
110: (8)         itype = ntypes.int32,
111: (8)         fmt = '%15.7e',
112: (8)         title = _title_fmt.format('single')),
113: (4)     ntypes.longdouble: dict(
114: (8)         itype = ntypes.longlong,
115: (8)         fmt = '%s',
116: (8)         title = _title_fmt.format('long double')),
117: (4)     ntypes.half: dict(
118: (8)         itype = ntypes.int16,
119: (8)         fmt = '%12.5e',
120: (0)         title = _title_fmt.format('half'))}
121: (0) _KNOWN_TYPES = {}
122: (4) def _register_type(machar, bytepat):
123: (0)     _KNOWN_TYPES[bytepat] = machar
124: (0) _float_ma = {}
125: (4) def _register_known_types():
126: (4)     f16 = ntypes.float16
127: (28)    float16_ma = MachArLike(f16,
128: (28)                                macheep=-10,
128: (28)                                negep=-11,

```

```

129: (28)                         minexp=-14,
130: (28)                         maxexp=16,
131: (28)                         it=10,
132: (28)                         iexp=5,
133: (28)                         ibeta=2,
134: (28)                         irnd=5,
135: (28)                         ngrd=0,
136: (28)                         eps=exp2(f16(-10)),
137: (28)                         epsneg=exp2(f16(-11)),
138: (28)                         huge=f16(65504),
139: (28)                         tiny=f16(2 ** -14))
140: (4)                          _register_type(float16_ma, b'f\xae')
141: (4)                          float_ma[16] = float16_ma
142: (4)                          f32 = ntypes.float32
143: (4)                          float32_ma = MachArLike(f32,
144: (28)                            machep=-23,
145: (28)                            negep=-24,
146: (28)                            minexp=-126,
147: (28)                            maxexp=128,
148: (28)                            it=23,
149: (28)                            iexp=8,
150: (28)                            ibeta=2,
151: (28)                            irnd=5,
152: (28)                            ngrd=0,
153: (28)                            eps=exp2(f32(-23)),
154: (28)                            epsneg=exp2(f32(-24)),
155: (28)                            huge=f32((1 - 2 ** -24) * 2**128),
156: (28)                            tiny=exp2(f32(-126)))
157: (4)                          _register_type(float32_ma, b'\xcd\xcc\xcc\xbd')
158: (4)                          float_ma[32] = float32_ma
159: (4)                          f64 = ntypes.float64
160: (4)                          epsneg_f64 = 2.0 ** -53.0
161: (4)                          tiny_f64 = 2.0 ** -1022.0
162: (4)                          float64_ma = MachArLike(f64,
163: (28)                            machep=-52,
164: (28)                            negep=-53,
165: (28)                            minexp=-1022,
166: (28)                            maxexp=1024,
167: (28)                            it=52,
168: (28)                            iexp=11,
169: (28)                            ibeta=2,
170: (28)                            irnd=5,
171: (28)                            ngrd=0,
172: (28)                            eps=2.0 ** -52.0,
173: (28)                            epsneg=epsneg_f64,
174: (28)                            huge=(1.0 - epsneg_f64) / tiny_f64 * f64(4),
175: (28)                            tiny=tiny_f64)
176: (4)                          _register_type(float64_ma, b'\x9a\x99\x99\x99\x99\x99\xb9\xbf')
177: (4)                          float_ma[64] = float64_ma
178: (4)                          ld = ntypes.longdouble
179: (4)                          epsneg_f128 = exp2(ld(-113))
180: (4)                          tiny_f128 = exp2(ld(-16382))
181: (4)                          with numeric.errstate(all='ignore'):
182: (8)                            huge_f128 = (ld(1) - epsneg_f128) / tiny_f128 * ld(4)
183: (4)                          float128_ma = MachArLike(ld,
184: (29)                            machep=-112,
185: (29)                            negep=-113,
186: (29)                            minexp=-16382,
187: (29)                            maxexp=16384,
188: (29)                            it=112,
189: (29)                            iexp=15,
190: (29)                            ibeta=2,
191: (29)                            irnd=5,
192: (29)                            ngrd=0,
193: (29)                            eps=exp2(ld(-112)),
194: (29)                            epsneg=epsneg_f128,
195: (29)                            huge=huge_f128,
196: (29)                            tiny=tiny_f128)
197: (4)                          _register_type(float128_ma,

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

198: (8)           b'\x9a\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\x99\xfb\xbf')
199: (4)           _float_ma[128] = float128_ma
200: (4)           epsneg_f80 = exp2(1d(-64))
201: (4)           tiny_f80 = exp2(1d(-16382))
202: (4)           with numeric.errstate(all='ignore'):
203: (8)             huge_f80 = (1d(1) - epsneg_f80) / tiny_f80 * 1d(4)
204: (4)             float80_ma = MachArLike(1d,
205: (28)               machep=-63,
206: (28)               negep=-64,
207: (28)               minexp=-16382,
208: (28)               maxexp=16384,
209: (28)               it=63,
210: (28)               iexp=15,
211: (28)               ibeta=2,
212: (28)               irnd=5,
213: (28)               ngrd=0,
214: (28)               eps=exp2(1d(-63)),
215: (28)               epsneg=epsneg_f80,
216: (28)               huge=huge_f80,
217: (28)               tiny=tiny_f80)
218: (4)           _register_type(float80_ma, b'\xcd\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xfb\xbf')
219: (4)           _float_ma[80] = float80_ma
220: (4)           huge_dd = nextafter(1d(inf), 1d(0), dtype=1d)
221: (4)           smallest_normal_dd = NaN
222: (4)           smallest_subnormal_dd = 1d(nextafter(0., 1.))
223: (4)           float_dd_ma = MachArLike(1d,
224: (29)             machep=-105,
225: (29)             negep=-106,
226: (29)             minexp=-1022,
227: (29)             maxexp=1024,
228: (29)             it=105,
229: (29)             iexp=11,
230: (29)             ibeta=2,
231: (29)             irnd=5,
232: (29)             ngrd=0,
233: (29)             eps=exp2(1d(-105)),
234: (29)             epsneg=exp2(1d(-106)),
235: (29)             huge=huge_dd,
236: (29)             tiny=smallest_normal_dd,
237: (29)             smallest_subnormal=smallest_subnormal_dd)
238: (4)           _register_type(float_dd_ma,
239: (8)             b'\x9a\x99\x99\x99\x99\x99Y<\x9a\x99\x99\x99\x99\x99\x99\x99\xbf')
240: (4)           _register_type(float_dd_ma,
241: (8)             b'\x9a\x99\x99\x99\x99\x99\x99\x99\xbf\x9a\x99\x99\x99\x99\x99\x99Y<')
242: (4)           _float_ma['dd'] = float_dd_ma
243: (0)           def _get_machar(ftype):
244: (4)             """ Get MachAr instance or MachAr-like instance
245: (4)             Get parameters for floating point type, by first trying signatures of
246: (4)             various known floating point types, then, if none match, attempting to
247: (4)             identify parameters by analysis.
248: (4)             Parameters
249: (4)             -----
250: (4)             ftype : class
251: (8)               Numpy floating point type class (e.g. ``np.float64``)
252: (4)             Returns
253: (4)             -----
254: (4)             ma_like : instance of :class:`MachAr` or :class:`MachArLike`-
255: (8)               Object giving floating point parameters for `ftype`.
256: (4)             Warns
257: (4)             -----
258: (4)             UserWarning
259: (8)               If the binary signature of the float type is not in the dictionary of
260: (8)                 known float types.
261: (4)             """
262: (4)             params = _MACHAR_PARAMS.get(ftype)
263: (4)             if params is None:
264: (8)               raise ValueError(repr(ftype))
265: (4)             key = (ftype(-1.0) / ftype(10.)).newbyteorder('<').tobytes()
266: (4)             ma_like = None

```

```

267: (4)           if ftype == ntypes.longdouble:
268: (8)             ma_like = _KNOWN_TYPES.get(key[:10])
269: (4)           if ma_like is None:
270: (8)             ma_like = _KNOWN_TYPES.get(key)
271: (4)           if ma_like is None and len(key) == 16:
272: (8)             _kt = {k[:10]: v for k, v in _KNOWN_TYPES.items() if len(k) == 16}
273: (8)             ma_like = _kt.get(key[:10])
274: (4)           if ma_like is not None:
275: (8)             return ma_like
276: (4)           warnings.warn(
277: (8)             f'Signature {key} for {ftype} does not match any known type: '
278: (8)             'falling back to type probe function.\n'
279: (8)             'This warning indicates broken support for the dtype!', 
280: (8)             UserWarning, stacklevel=2)
281: (4)           return _discovered_machar(ftype)
282: (0)         def _discovered_machar(ftype):
283: (4)           """ Create MachAr instance with found information on float types
284: (4)           TODO: MachAr should be retired completely ideally. We currently only
285: (10)             ever use it system with broken longdouble (valgrind, WSL).
286: (4)           """
287: (4)           params = _MACHAR_PARAMS[ftype]
288: (4)           return MachAr(lambda v: array([v], ftype),
289: (18)             lambda v:_fr0(v.astype(params['itype']))[0],
290: (18)             lambda v:array(_fr0(v)[0], ftype),
291: (18)             lambda v: params['fmt'] % array(_fr0(v)[0], ftype),
292: (18)             params['title'])
293: (0)         @set_module('numpy')
294: (0)         class finfo:
295: (4)           """
296: (4)             finfo(dtype)
297: (4)               Machine limits for floating point types.
298: (4)             Attributes
299: (4)             -----
300: (4)             bits : int
301: (8)               The number of bits occupied by the type.
302: (4)             dtype : dtype
303: (8)               Returns the dtype for which `finfo` returns information. For complex
304: (8)               input, the returned dtype is the associated ``float`` dtype for its
305: (8)               real and complex components.
306: (4)             eps : float
307: (8)               The difference between 1.0 and the next smallest representable float
308: (8)               larger than 1.0. For example, for 64-bit binary floats in the IEEE-754
309: (8)               standard, ``eps = 2**-52``, approximately 2.22e-16.
310: (4)             epsneg : float
311: (8)               The difference between 1.0 and the next smallest representable float
312: (8)               less than 1.0. For example, for 64-bit binary floats in the IEEE-754
313: (8)               standard, ``epsneg = 2**-53``, approximately 1.11e-16.
314: (4)             iexp : int
315: (8)               The number of bits in the exponent portion of the floating point
316: (8)               representation.
317: (4)             machep : int
318: (8)               The exponent that yields `eps`.
319: (4)             max : floating point number of the appropriate type
320: (8)               The largest representable number.
321: (4)             maxexp : int
322: (8)               The smallest positive power of the base (2) that causes overflow.
323: (4)             min : floating point number of the appropriate type
324: (8)               The smallest representable number, typically ``-max``.
325: (4)             minexp : int
326: (8)               The most negative power of the base (2) consistent with there
327: (8)               being no leading 0's in the mantissa.
328: (4)             negep : int
329: (8)               The exponent that yields `epsneg`.
330: (4)             nexp : int
331: (8)               The number of bits in the exponent including its sign and bias.
332: (4)             nmant : int
333: (8)               The number of bits in the mantissa.
334: (4)             precision : int
335: (8)               The approximate number of decimal digits to which this kind of

```

```

336: (8)           float is precise.
337: (4)           resolution : floating point number of the appropriate type
338: (8)             The approximate decimal resolution of this type, i.e.,
339: (8)               ``10**-precision``.
340: (4)           tiny : float
341: (8)             An alias for `smallest_normal`, kept for backwards compatibility.
342: (4)           smallest_normal : float
343: (8)             The smallest positive floating point number with 1 as leading bit in
344: (8)               the mantissa following IEEE-754 (see Notes).
345: (4)           smallest_subnormal : float
346: (8)             The smallest positive floating point number with 0 as leading bit in
347: (8)               the mantissa following IEEE-754.
348: (4)           Parameters
349: (4)           -----
350: (4)             dtype : float, dtype, or instance
351: (8)               Kind of floating point or complex floating point
352: (8)                 data-type about which to get information.
353: (4)           See Also
354: (4)           -----
355: (4)             iinfo : The equivalent for integer data types.
356: (4)             spacing : The distance between a value and the nearest adjacent number
357: (4)             nextafter : The next floating point value after x1 towards x2
358: (4)           Notes
359: (4)           -----
360: (4)             For developers of NumPy: do not instantiate this at the module level.
361: (4)             The initial calculation of these parameters is expensive and negatively
362: (4)               impacts import times. These objects are cached, so calling ``finfo()``
363: (4)               repeatedly inside your functions is not a problem.
364: (4)             Note that ``smallest_normal`` is not actually the smallest positive
365: (4)               representable value in a NumPy floating point type. As in the IEEE-754
366: (4)               standard [1]_, NumPy floating point types make use of subnormal numbers to
367: (4)               fill the gap between 0 and ``smallest_normal``. However, subnormal numbers
368: (4)               may have significantly reduced precision [2]_.
369: (4)             This function can also be used for complex data types as well. If used,
370: (4)               the output will be the same as the corresponding real float type
371: (4)                 (e.g. numpy.finfo(numpy.csingle) is the same as
numpy.finfo(numpy.single)).
372: (4)               However, the output is true for the real and imaginary components.
373: (4)           References
374: (4)           -----
375: (4)             .. [1] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008,
376: (11)               pp.1-70, 2008, http://www.doi.org/10.1109/IEEEESTD.2008.4610935
377: (4)             .. [2] Wikipedia, "Denormal Numbers",
378: (11)               https://en.wikipedia.org/wiki/Denormal\_number
379: (4)           Examples
380: (4)           -----
381: (4)             >>> np.finfo(np.float64).dtype
382: (4)               dtype('float64')
383: (4)             >>> np.finfo(np.complex64).dtype
384: (4)               dtype('float32')
385: (4)               """
386: (4)               _finfo_cache = {}
387: (4)             def __new__(cls, dtype):
388: (8)               try:
389: (12)                   obj = cls._finfo_cache.get(dtype) # most common path
390: (12)                   if obj is not None:
391: (16)                       return obj
392: (8)                   except TypeError:
393: (12)                       pass
394: (8)                   if dtype is None:
395: (12)                       warnings.warn(
396: (16)                           "finfo() dtype cannot be None. This behavior will "
397: (16)                           "raise an error in the future. (Deprecated in NumPy 1.25)",
398: (16)                           DeprecationWarning,
399: (16)                           stacklevel=2
400: (12)                   )
401: (8)               try:
402: (12)                   dtype = numeric.dtype(dtype)
403: (8)               except TypeError:

```

```

404: (12)           dtype = numeric.dtype(type(dtype))
405: (8)            obj = cls._finfo_cache.get(dtype)
406: (8)            if obj is not None:
407: (12)              return obj
408: (8)            dtypes = [dtype]
409: (8)            newdtype = numeric.obj2sctype(dtype)
410: (8)            if newdtype is not dtype:
411: (12)              dtypes.append(newdtype)
412: (12)              dtype = newdtype
413: (8)            if not issubclass(dtype, numeric.inexact):
414: (12)              raise ValueError("data type %r not inexact" % (dtype))
415: (8)            obj = cls._finfo_cache.get(dtype)
416: (8)            if obj is not None:
417: (12)              return obj
418: (8)            if not issubclass(dtype, numeric.floating):
419: (12)              newdtype = _convert_to_float[dtype]
420: (12)              if newdtype is not dtype:
421: (16)                dtypes.append(newdtype)
422: (16)                dtype = newdtype
423: (16)                obj = cls._finfo_cache.get(dtype, None)
424: (16)                if obj is not None:
425: (20)                  for dt in dtypes:
426: (24)                    cls._finfo_cache[dt] = obj
427: (20)            return obj
428: (8)            obj = object.__new__(cls).__init__(dtype)
429: (8)            for dt in dtypes:
430: (12)              cls._finfo_cache[dt] = obj
431: (8)            return obj
432: (4)            def __init__(self, dtype):
433: (8)              self.dtype = numeric.dtype(dtype)
434: (8)              machar = _get_machar(dtype)
435: (8)              for word in ['precision', 'iexp',
436: (21)                  'maxexp', 'minexp', 'negep',
437: (21)                  'macheep']:
438: (12)                setattr(self, word, getattr(machar, word))
439: (8)              for word in ['resolution', 'epsneg', 'smallest_subnormal']:
440: (12)                setattr(self, word, getattr(machar, word).flat[0])
441: (8)              self.bits = self.dtype.itemsize * 8
442: (8)              self.max = machar.huge.flat[0]
443: (8)              self.min = -self.max
444: (8)              self.eps = machar.eps.flat[0]
445: (8)              self.nexp = machar.iexp
446: (8)              self.nmant = machar.it
447: (8)              self._machar = machar
448: (8)              self._str_tiny = machar._str_xmin.strip()
449: (8)              self._str_max = machar._str_xmax.strip()
450: (8)              self._str_epsneg = machar._str_epsneg.strip()
451: (8)              self._str_eps = machar._str_eps.strip()
452: (8)              self._str_resolution = machar._str_resolution.strip()
453: (8)              self._str_smallest_normal = machar._str_smallest_normal.strip()
454: (8)              self._str_smallest_subnormal = machar._str_smallest_subnormal.strip()
455: (8)              return self
456: (4)            def __str__(self):
457: (8)              fmt = (
458: (12)                  'Machine parameters for %(dtype)s\n'
459: (12)                  '-----\n'
460: (12)                  'precision = %(precision)3s    resolution = %(_str_resolution)s\n'
461: (12)                  'macheep = %(macheep)6s    eps =      %(_str_eps)s\n'
462: (12)                  'negep =  %(negep)6s    epsneg =    %(_str_epsneg)s\n'
463: (12)                  'minexp = %(minexp)6s    tiny =     %(_str_tiny)s\n'
464: (12)                  'maxexp = %(maxexp)6s    max =      %(_str_max)s\n'
465: (12)                  'nexp =   %(nexp)6s    min =      -max\n'
466: (12)                  'smallest_normal = %(_str_smallest_normal)s '
467: (12)                  'smallest_subnormal = %(_str_smallest_subnormal)s\n'
468: (12)                  '-----\n'
469: (12)              )
470: (8)              return fmt % self.__dict__

```

```

471: (4) def __repr__(self):
472: (8)     c = self.__class__.name_
473: (8)     d = self.__dict__.copy()
474: (8)     d['klass'] = c
475: (8)     return ('(%(klass)s(resolution=%(resolution)s, min=-%(__str_max)s,"'
476: (17)             " max=%(_str_max)s, dtype=%(dtype)s)") % d)
477: (4) @property
478: (4) def smallest_normal(self):
479: (8)     """Return the value for the smallest normal.
480: (8)     Returns
481: (8)     -----
482: (8)     smallest_normal : float
483: (12)         Value for the smallest normal.
484: (8)     Warns
485: (8)     -----
486: (8)     UserWarning
487: (12)         If the calculated value for the smallest normal is requested for
488: (12)         double-double.
489: (8)     """
490: (8)     if isnan(self._machar.smallest_normal.flat[0]):
491: (12)         warnings.warn(
492: (16)             'The value of smallest normal is undefined for double double',
493: (16)             UserWarning, stacklevel=2)
494: (8)     return self._machar.smallest_normal.flat[0]
495: (4) @property
496: (4) def tiny(self):
497: (8)     """Return the value for tiny, alias of smallest_normal.
498: (8)     Returns
499: (8)     -----
500: (8)     tiny : float
501: (12)         Value for the smallest normal, alias of smallest_normal.
502: (8)     Warns
503: (8)     -----
504: (8)     UserWarning
505: (12)         If the calculated value for the smallest normal is requested for
506: (12)         double-double.
507: (8)     """
508: (8)     return self.smallest_normal
509: (0) @set_module('numpy')
510: (0) class iinfo:
511: (4)     """
512: (4)     iinfo(type)
513: (4)         Machine limits for integer types.
514: (4)     Attributes
515: (4)     -----
516: (4)     bits : int
517: (8)         The number of bits occupied by the type.
518: (4)     dtype : dtype
519: (8)         Returns the dtype for which `iinfo` returns information.
520: (4)     min : int
521: (8)         The smallest integer expressible by the type.
522: (4)     max : int
523: (8)         The largest integer expressible by the type.
524: (4)     Parameters
525: (4)     -----
526: (4)     int_type : integer type, dtype, or instance
527: (8)         The kind of integer data type to get information about.
528: (4)     See Also
529: (4)     -----
530: (4)     finfo : The equivalent for floating point data types.
531: (4)     Examples
532: (4)     -----
533: (4)     With types:
534: (4)     >>> ii16 = np.iinfo(np.int16)
535: (4)     >>> ii16.min
536: (4)     -32768
537: (4)     >>> ii16.max
538: (4)     32767
539: (4)     >>> ii32 = np.iinfo(np.int32)

```

```

540: (4)          >>> ii32.min
541: (4)          -2147483648
542: (4)          >>> ii32.max
543: (4)          2147483647
544: (4)          With instances:
545: (4)          >>> ii32 = np.iinfo(np.int32(10))
546: (4)          >>> ii32.min
547: (4)          -2147483648
548: (4)          >>> ii32.max
549: (4)          2147483647
550: (4)          """
551: (4)          _min_vals = {}
552: (4)          _max_vals = {}
553: (4)          def __init__(self, int_type):
554: (8)            try:
555: (12)              self.dtype = numeric.dtype(int_type)
556: (8)            except TypeError:
557: (12)              self.dtype = numeric.dtype(type(int_type))
558: (8)            self.kind = self.dtype.kind
559: (8)            self.bits = self.dtype.itemsize * 8
560: (8)            self.key = "%s%d" % (self.kind, self.bits)
561: (8)            if self.kind not in 'iu':
562: (12)              raise ValueError("Invalid integer data type %r." % (self.kind,))
563: (4)          @property
564: (4)          def min(self):
565: (8)            """Minimum value of given dtype."""
566: (8)            if self.kind == 'u':
567: (12)              return 0
568: (8)            else:
569: (12)              try:
570: (16)                val = iinfo._min_vals[self.key]
571: (12)              except KeyError:
572: (16)                val = int(-(1 << (self.bits-1)))
573: (16)                iinfo._min_vals[self.key] = val
574: (12)              return val
575: (4)          @property
576: (4)          def max(self):
577: (8)            """Maximum value of given dtype."""
578: (8)            try:
579: (12)              val = iinfo._max_vals[self.key]
580: (8)            except KeyError:
581: (12)              if self.kind == 'u':
582: (16)                val = int((1 << self.bits) - 1)
583: (12)              else:
584: (16)                val = int((1 << (self.bits-1)) - 1)
585: (12)                iinfo._max_vals[self.key] = val
586: (8)              return val
587: (4)          def __str__(self):
588: (8)            """String representation."""
589: (8)            fmt = (
590: (12)              'Machine parameters for %(dtype)s\n'
591: (12)              '-----\n'
592: (12)              'min = %(min)s\n'
593: (12)              'max = %(max)s\n'
594: (12)              '-----\n'
595: (12)              )
596: (8)              return fmt % {'dtype': self.dtype, 'min': self.min, 'max': self.max}
597: (4)          def __repr__(self):
598: (8)            return "%s(min=%s, max=%s, dtype=%s)" % (self.__class__.__name__,
599: (36)                                self.min, self.max, self.dtype)
-----
```

File 51 - memmap.py:

```

1: (0)          from contextlib import nullcontext
2: (0)          import numpy as np
```

```

3: (0) from .._utils import set_module
4: (0) from .numeric import uint8, ndarray, dtype
5: (0) from numpy.compat import os_fspath, is_pathlib_path
6: (0) __all__ = ['memmap']
7: (0) dtypedescr = dtype
8: (0) valid_filemodes = ["r", "c", "r+", "w+"]
9: (0) writeable_filemodes = ["r+", "w+"]
10: (0) mode_equivalents = {
11: (4)     "readonly": "r",
12: (4)     "copyonwrite": "c",
13: (4)     "readwrite": "r+",
14: (4)     "write": "w"
15: (4) }
16: (0) @set_module('numpy')
17: (0) class memmap(ndarray):
18: (4)     """Create a memory-map to an array stored in a *binary* file on disk.
19: (4)     Memory-mapped files are used for accessing small segments of large files
20: (4)     on disk, without reading the entire file into memory. NumPy's
21: (4)     memmap's are array-like objects. This differs from Python's ``mmap``
22: (4)     module, which uses file-like objects.
23: (4)     This subclass of ndarray has some unpleasant interactions with
24: (4)     some operations, because it doesn't quite fit properly as a subclass.
25: (4)     An alternative to using this subclass is to create the ``mmap``
26: (4)     object yourself, then create an ndarray with ndarray.__new__ directly,
27: (4)     passing the object created in its 'buffer=' parameter.
28: (4)     This class may at some point be turned into a factory function
29: (4)     which returns a view into an mmap buffer.
30: (4)     Flush the memmap instance to write the changes to the file. Currently
there
31: (4)     is no API to close the underlying ``mmap``. It is tricky to ensure the
32: (4)     resource is actually closed, since it may be shared between different
33: (4)     memmap instances.
34: (4)     Parameters
35: (4)     -----
36: (4)     filename : str, file-like object, or pathlib.Path instance
37: (8)         The file name or file object to be used as the array data buffer.
38: (4)     dtype : data-type, optional
39: (8)         The data-type used to interpret the file contents.
40: (8)         Default is `uint8`.
41: (4)     mode : {'r+', 'r', 'w+', 'c'}, optional
42: (8)         The file is opened in this mode:
43: (8)         +-----+
44: (8)         | 'r' | Open existing file for reading only.
45: (8)         +-----+
46: (8)         | 'r+' | Open existing file for reading and writing.
47: (8)         +-----+
48: (8)         | 'w+' | Create or overwrite existing file for reading and writing.
49: (8)             | If ``mode == 'w+'`` then `shape` must also be specified.
50: (8)         +-----+
51: (8)         | 'c' | Copy-on-write: assignments affect data in memory, but
52: (8)             | changes are not saved to disk. The file on disk is
53: (8)             | read-only.
54: (8)         +-----+
55: (8)         Default is 'r+'.
56: (4)     offset : int, optional
57: (8)         In the file, array data starts at this offset. Since `offset` is
58: (8)         measured in bytes, it should normally be a multiple of the byte-size
59: (8)         of `dtype`. When ``mode != 'r'``, even positive offsets beyond end of
60: (8)         file are valid; The file will be extended to accommodate the
61: (8)         additional data. By default, ``memmap`` will start at the beginning of
62: (8)         the file, even if ``filename`` is a file pointer ``fp`` and
63: (8)             ``fp.tell() != 0``.
64: (4)     shape : tuple, optional
65: (8)         The desired shape of the array. If ``mode == 'r'`` and the number
66: (8)         of remaining bytes after `offset` is not a multiple of the byte-size
67: (8)         of `dtype`, you must specify `shape`. By default, the returned array
68: (8)         will be 1-D with the number of elements determined by file size
69: (8)         and data-type.
70: (4)     order : {'C', 'F'}, optional

```

```
71: (8)          Specify the order of the ndarray memory layout:
72: (8)          :term:`row-major`, C-style or :term:`column-major`,
73: (8)          Fortran-style. This only has an effect if the shape is
74: (8)          greater than 1-D. The default order is 'C'.
75: (4)          Attributes
76: (4)          -----
77: (4)          filename : str or pathlib.Path instance
78: (8)          Path to the mapped file.
79: (4)          offset : int
80: (8)          Offset position in the file.
81: (4)          mode : str
82: (8)          File mode.
83: (4)          Methods
84: (4)          -----
85: (4)          flush
86: (8)          Flush any changes in memory to file on disk.
87: (8)          When you delete a memmap object, flush is called first to write
88: (8)          changes to disk.
89: (4)          See also
90: (4)          -----
91: (4)          lib.format.open_memmap : Create or load a memory-mapped ``.npy`` file.
92: (4)          Notes
93: (4)          -----
94: (4)          The memmap object can be used anywhere an ndarray is accepted.
95: (4)          Given a memmap ``fp``, ``isinstance(fp, numpy.ndarray)`` returns
96: (4)          ``True``.
97: (4)          Memory-mapped files cannot be larger than 2GB on 32-bit systems.
98: (4)          When a memmap causes a file to be created or extended beyond its
99: (4)          current size in the filesystem, the contents of the new part are
100: (4)          unspecified. On systems with POSIX filesystem semantics, the extended
101: (4)          part will be filled with zero bytes.
102: (4)          Examples
103: (4)          -----
104: (4)          >>> data = np.arange(12, dtype='float32')
105: (4)          >>> data.resize((3,4))
106: (4)          This example uses a temporary file so that doctest doesn't write
107: (4)          files to your directory. You would use a 'normal' filename.
108: (4)          >>> from tempfile import mkdtemp
109: (4)          >>> import os.path as path
110: (4)          >>> filename = path.join(mkdtemp(), 'newfile.dat')
111: (4)          Create a memmap with dtype and shape that matches our data:
112: (4)          >>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))
113: (4)          >>> fp
114: (4)          memmap([[0., 0., 0., 0.],
115: (12)            [0., 0., 0., 0.],
116: (12)            [0., 0., 0., 0.]], dtype=float32)
117: (4)          Write data to memmap array:
118: (4)          >>> fp[:] = data[:]
119: (4)          >>> fp
120: (4)          memmap([[ 0.,   1.,   2.,   3.],
121: (12)            [ 4.,   5.,   6.,   7.],
122: (12)            [ 8.,   9.,  10.,  11.]], dtype=float32)
123: (4)          >>> fp.filename == path.abspath(filename)
124: (4)          True
125: (4)          Flushes memory changes to disk in order to read them back
126: (4)          >>> fp.flush()
127: (4)          Load the memmap and verify data was stored:
128: (4)          >>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
129: (4)          >>> newfp
130: (4)          memmap([[ 0.,   1.,   2.,   3.],
131: (12)            [ 4.,   5.,   6.,   7.],
132: (12)            [ 8.,   9.,  10.,  11.]], dtype=float32)
133: (4)          Read-only memmap:
134: (4)          >>> fpr = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
135: (4)          >>> fpr.flags.writeable
136: (4)          False
137: (4)          Copy-on-write memmap:
138: (4)          >>> fpc = np.memmap(filename, dtype='float32', mode='c', shape=(3,4))
139: (4)          >>> fpc.flags.writeable
```

```

140: (4)           True
141: (4)           It's possible to assign to copy-on-write array, but values are only
142: (4)           written into the memory copy of the array, and not written to disk:
143: (4)
144: (4)           >>> fpc
145: (12)          memmap([[ 0.,  1.,  2.,  3.],
146: (12)          [ 4.,  5.,  6.,  7.],
147: (12)          [ 8.,  9., 10., 11.]], dtype=float32)
148: (4)           >>> fpc[0,:] = 0
149: (4)           >>> fpc
150: (12)          memmap([[ 0.,  0.,  0.,  0.],
151: (12)          [ 4.,  5.,  6.,  7.],
152: (12)          [ 8.,  9., 10., 11.]], dtype=float32)
153: (4)           File on disk is unchanged:
154: (4)           >>> fpr
155: (12)          memmap([[ 0.,  1.,  2.,  3.],
156: (12)          [ 4.,  5.,  6.,  7.],
157: (12)          [ 8.,  9., 10., 11.]], dtype=float32)
158: (4)           Offset into a memmap:
159: (4)           >>> fpo = np.memmap(filename, dtype='float32', mode='r', offset=16)
160: (4)           >>> fpo
161: (4)           memmap([ 4.,  5.,  6.,  7.,  8.,  9., 10., 11.], dtype=float32)
162: (4)           """
163: (4)           __array_priority__ = -100.0
164: (16)          def __new__(subtype, filename, dtype:uint8, mode='r+', offset=0,
165: (8)             shape=None, order='C'):
166: (8)             import mmap
167: (8)             import os.path
168: (12)            try:
169: (8)               mode = mode_equivalents[mode]
170: (12)            except KeyError as e:
171: (16)              if mode not in valid_filemodes:
172: (20)                raise ValueError(
173: (20)                  "mode must be one of {!r} (got {!r})"
174: (16)                  .format(valid_filemodes + list(mode_equivalents.keys())),
175: (8)                  ) from None
176: (12)            if mode == 'w+' and shape is None:
177: (8)              raise ValueError("shape must be given if mode == 'w+'")
178: (12)            if hasattr(filename, 'read'):
179: (8)              f_ctx = nullcontext(filename)
180: (12)            else:
181: (8)              f_ctx = open(os_fspath(filename), ('r' if mode == 'c' else
182: (12)                mode)+'b')
183: (12)
184: (12)            with f_ctx as fid:
185: (12)              fid.seek(0, 2)
186: (12)             flen = fid.tell()
187: (16)              descr = dtypedescr(dtype)
188: (16)              _dbytes = descr.itemsize
189: (20)              if shape is None:
190: (28)                bytes =flen - offset
191: (16)                if bytes % _dbytes:
192: (16)                  raise ValueError("Size of available data is not a "
193: (12)                      "multiple of the data-type size.")
194: (16)                  size = bytes // _dbytes
195: (20)                  shape = (size,)
196: (16)                  size = np.intp(1) # avoid default choice of np.int_, which
197: (16)                  might overflow
198: (20)
199: (12)                  for k in shape:
200: (12)                    size *= k
201: (16)                    bytes = int(offset + size*_dbytes)
202: (16)                    if mode in ('w+', 'r+') and flen < bytes:
203: (16)                      fid.seek(bytes - 1, 0)
204: (12)                      fid.write(b'\0')
205: (16)                      fid.flush()
206: (12)                      if mode == 'c':
207: (16)                        acc = mmap.ACCESS_COPY

```

```

206: (12)           elif mode == 'r':
207: (16)             acc = mmap.ACCESS_READ
208: (12)
209: (16)             else:
210: (12)               acc = mmap.ACCESS_WRITE
211: (12)               start = offset - offset % mmap.ALLOCATIONGRANULARITY
212: (12)               bytes -= start
213: (12)               array_offset = offset - start
214: (12)               mm = mmap.mmap(fid.fileno(), bytes, access=acc, offset=start)
215: (35)               self = ndarray.__new__(subtype, shape, dtype=descr, buffer=mm,
216: (12)                             offset=array_offset, order=order)
217: (12)               self._mmap = mm
218: (12)               self.offset = offset
219: (12)               self.mode = mode
220: (16)               if is_pathlib_path(filename):
221: (12)                 self.filename = filename.resolve()
222: (16)               elif hasattr(fid, "name") and isinstance(fid.name, str):
223: (12)                 self.filename = os.path.abspath(fid.name)
224: (12)
225: (16)               else:
226: (8)                 self.filename = None
227: (4)             return self
228: (8)
229: (12)             def __array_finalize__(self, obj):
230: (12)               if hasattr(obj, '_mmap') and np.may_share_memory(self, obj):
231: (12)                 self._mmap = obj._mmap
232: (12)                 self.filename = obj.filename
233: (12)                 self.offset = obj.offset
234: (12)                 self.mode = obj.mode
235: (12)
236: (12)
237: (4)             def flush(self):
238: (8)               """
239: (8)               Write any changes in the array to the file on disk.
240: (8)               For further information, see `memmap`.
241: (8)               Parameters
242: (8)               -----
243: (8)               None
244: (8)               See Also
245: (8)               -----
246: (8)               memmap
247: (8)               """
248: (8)               if self.base is not None and hasattr(self.base, 'flush'):
249: (12)                 self.base.flush()
250: (4)             def __array_wrap__(self, arr, context=None):
251: (8)               arr = super().__array_wrap__(arr, context)
252: (8)               if self is arr or type(self) is not memmap:
253: (12)                 return arr
254: (8)               if arr.shape == ():
255: (12)                 return arr[()]
256: (8)               return arr.view(np.ndarray)
257: (4)             def __getitem__(self, index):
258: (8)               res = super().__getitem__(index)
259: (8)               if type(res) is memmap and res._mmap is None:
260: (12)                 return res.view(type=ndarray)
261: (8)               return res

```

-----

## File 52 - multiarray.py:

```

1: (0)           """
2: (0)             Create the numpy.core.multiarray namespace for backward compatibility. In
v1.16
3: (0)             the multiarray and umath c-extension modules were merged into a single
4: (0)             _multiarray_umath extension module. So we replicate the old namespace
5: (0)             by importing from the extension module.
6: (0)             """
7: (0)             import functools

```

```

8: (0) from . import overrides
9: (0) from . import _multiarray_umath
10: (0) from ._multiarray_umath import * # noqa: F403
11: (0) from ._multiarray_umath import (
12: (4)     fastCopyAndTranspose, _flagdict, from_dlpark, _place, _reconstruct,
13: (4)     _vec_string, _ARRAY_API, _monotonicity, _get_ndarray_c_version,
14: (4)     _get_madvise_hugepage, _set_madvise_hugepage,
15: (4)     _get_promotion_state, _set_promotion_state, _using_numpy2_behavior
16: (4) )
17: (0) __all__ = [
18: (4)     '_ARRAY_API', 'ALLOW_THREADS', 'BUFSIZE', 'CLIP', 'DATETIMEUNITS',
19: (4)     'ITEM_HASOBJECT', 'ITEM_IS_POINTER', 'LIST_PICKLE', 'MAXDIMS',
20: (4)     'MAY_SHARE_BOUNDS', 'MAY_SHARE_EXACT', 'NEEDS_INIT', 'NEEDS_PYAPI',
21: (4)     'RAISE', 'USE_GETITEM', 'USE_SETITEM', 'WRAP',
22: (4)     '_flagdict', 'from_dlpark', '_place', '_reconstruct', '_vec_string',
23: (4)     '_monotonicity', 'add_docstring', 'arange', 'array', 'asarray',
24: (4)     'asanyarray', 'ascontiguousarray', 'asfortranarray', 'bincount',
25: (4)     'broadcast', 'busday_count', 'busday_offset', 'busdaycalendar',
26: (4)     'can_cast',
27: (4)     'format_longfloat',
28: (4)
29: (4)     'set_datetimeparse_function',
30: (4)     'frombuffer', 'fromfile', 'fromiter', 'fromstring',
31: (4)     'get_handler_name', 'get_handler_version', 'inner', 'interp',
32: (4)     'interp_complex', 'is_busday', 'lexsort', 'matmul', 'may_share_memory',
33: (4)     'min_scalar_type', 'ndarray', 'nditer', 'nested_iters',
34: (4)     'normalize_axis_index', 'packbits', 'promote_types', 'putmask',
35: (4)     'ravel_multi_index', 'result_type', 'scalar',
36: (4)     'set_legacy_print_mode', 'set_numeric_ops', 'set_string_function',
37: (4)     'set_typeDict', 'shares_memory', 'tracemalloc_domain', 'typeinfo',
38: (4)     'unpackbits', 'unravel_index', 'vdot', 'where', 'zeros',
39: (4)     '_get_promotion_state', '_set_promotion_state', '_using_numpy2_behavior']
40: (0)     _reconstruct.__module__ = 'numpy.core.multiarray'
41: (0)     scalar.__module__ = 'numpy.core.multiarray'
42: (0)     from_dlpark.__module__ = 'numpy'
43: (0)     arange.__module__ = 'numpy'
44: (0)     array.__module__ = 'numpy'
45: (0)     asarray.__module__ = 'numpy'
46: (0)     asanyarray.__module__ = 'numpy'
47: (0)     ascontiguousarray.__module__ = 'numpy'
48: (0)     asfortranarray.__module__ = 'numpy'
49: (0)     datetime_data.__module__ = 'numpy'
50: (0)     empty.__module__ = 'numpy'
51: (0)     frombuffer.__module__ = 'numpy'
52: (0)     fromfile.__module__ = 'numpy'
53: (0)     fromiter.__module__ = 'numpy'
54: (0)     frompyfunc.__module__ = 'numpy'
55: (0)     fromstring.__module__ = 'numpy'
56: (0)     geterrobj.__module__ = 'numpy'
57: (0)     may_share_memory.__module__ = 'numpy'
58: (0)     nested_iters.__module__ = 'numpy'
59: (0)     promote_types.__module__ = 'numpy'
60: (0)     set_numeric_ops.__module__ = 'numpy'
61: (0)     seterrobj.__module__ = 'numpy'
62: (0)     zeros.__module__ = 'numpy'
63: (0)     _get_promotion_state.__module__ = 'numpy'
64: (0)     _set_promotion_state.__module__ = 'numpy'
65: (0)     _using_numpy2_behavior.__module__ = 'numpy'
66: (0)     array_function_from_c_func_and_dispatcher = functools.partial(
67: (4)         overrides.array_function_from_dispatcher,
68: (4)         module='numpy', docs_from_dispatcher=True, verify=False)
69: (0)     @array_function_from_c_func_and_dispatcher(_multiarray_umath.empty_like)
70: (0)     def empty_like(prototype, dtype=None, order=None, subok=None, shape=None):
71: (4)         """
72: (4)             empty_like(prototype, dtype=None, order='K', subok=True, shape=None)
73: (4)             Return a new array with the same shape and type as a given array.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

74: (4)   Parameters
75: (4)   -----
76: (4)   prototype : array_like
77: (8)   The shape and data-type of `prototype` define these same attributes
78: (8)   of the returned array.
79: (4)   dtype : data-type, optional
80: (8)   Overrides the data type of the result.
81: (8)   .. versionadded:: 1.6.0
82: (4)   order : {'C', 'F', 'A', or 'K'}, optional
83: (8)   Overrides the memory layout of the result. 'C' means C-order,
84: (8)   'F' means F-order, 'A' means 'F' if `prototype` is Fortran
85: (8)   contiguous, 'C' otherwise. 'K' means match the layout of `prototype`
86: (8)   as closely as possible.
87: (8)   .. versionadded:: 1.6.0
88: (4)   subok : bool, optional.
89: (8)   If True, then the newly created array will use the sub-class
90: (8)   type of `prototype`, otherwise it will be a base-class array. Defaults
91: (8)   to True.
92: (4)   shape : int or sequence of ints, optional.
93: (8)   Overrides the shape of the result. If order='K' and the number of
94: (8)   dimensions is unchanged, will try to keep order, otherwise,
95: (8)   order='C' is implied.
96: (8)   .. versionadded:: 1.17.0
97: (4)   Returns
98: (4)   -----
99: (4)   out : ndarray
100: (8)   Array of uninitialized (arbitrary) data with the same
101: (8)   shape and type as `prototype`.
102: (4)   See Also
103: (4)   -----
104: (4)   ones_like : Return an array of ones with shape and type of input.
105: (4)   zeros_like : Return an array of zeros with shape and type of input.
106: (4)   full_like : Return a new array with shape of input filled with value.
107: (4)   empty : Return a new uninitialized array.
108: (4)   Notes
109: (4)   -----
110: (4)   This function does *not* initialize the returned array; to do that use
111: (4)   `zeros_like` or `ones_like` instead. It may be marginally faster than
112: (4)   the functions that do set the array values.
113: (4)   Examples
114: (4)   -----
115: (4)   >>> a = ([1,2,3], [4,5,6])                                     # a is array-like
116: (4)
117: (4)   >>> np.empty_like(a)
118: (11)   array([[-1073741821, -1073741821, 3],      # uninitialized
119: (4)   [ 0, 0, -1073741821]])
120: (4)
121: (4)   >>> a = np.array([[1., 2., 3.],[4., 5., 6.]])
122: (11)   >>> np.empty_like(a)
123: (4)   array([[ -2.00000715e+000,  1.48219694e-323, -2.00000572e+000], #
124: (4)   [ 4.38791518e-305, -2.00000715e+000,  4.17269252e-309]])
125: (0)   """
126: (0)   return (prototype,)
127: (4)   @array_function_from_c_func_and_dispatcher(_multiarray_umath.concatenate)
128: (4)   def concatenate(arrays, axis=None, out=None, *, dtype=None, casting=None):
129: (4)   """
130: (4)   concatenate((a1, a2, ...), axis=0, out=None, dtype=None,
131: (4)   Join a sequence of arrays along an existing axis.
132: (4)   Parameters
133: (8)   -----
134: (8)   a1, a2, ... : sequence of array_like
135: (8)   The arrays must have the same shape, except in the dimension
136: (8)   corresponding to `axis` (the first, by default).
137: (8)   axis : int, optional
138: (8)   The axis along which the arrays will be joined. If axis is None,
139: (8)   arrays are flattened before use. Default is 0.
140: (8)   out : ndarray, optional
141: (8)   If provided, the destination to place the result. The shape must be
142: (8)   correct, matching that of what concatenate would have returned if no

```

```

141: (8)          out argument were specified.
142: (4)          dtype : str or dtype
143: (8)            If provided, the destination array will have this dtype. Cannot be
144: (8)            provided together with `out`.
145: (8)            .. versionadded:: 1.20.0
146: (4)          casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
147: (8)            Controls what kind of data casting may occur. Defaults to 'same_kind'.
148: (8)            .. versionadded:: 1.20.0
149: (4)          Returns
150: (4)          -----
151: (4)          res : ndarray
152: (8)            The concatenated array.
153: (4)          See Also
154: (4)          -----
155: (4)          ma.concatenate : Concatenate function that preserves input masks.
156: (4)          array_split : Split an array into multiple sub-arrays of equal or
157: (18)            near-equal size.
158: (4)          split : Split array into a list of multiple sub-arrays of equal size.
159: (4)          hsplit : Split array into multiple sub-arrays horizontally (column wise).
160: (4)          vsplit : Split array into multiple sub-arrays vertically (row wise).
161: (4)          dsplit : Split array into multiple sub-arrays along the 3rd axis (depth).
162: (4)          stack : Stack a sequence of arrays along a new axis.
163: (4)          block : Assemble arrays from blocks.
164: (4)          hstack : Stack arrays in sequence horizontally (column wise).
165: (4)          vstack : Stack arrays in sequence vertically (row wise).
166: (4)          dstack : Stack arrays in sequence depth wise (along third dimension).
167: (4)          column_stack : Stack 1-D arrays as columns into a 2-D array.
168: (4)          Notes
169: (4)          -----
170: (4)          When one or more of the arrays to be concatenated is a MaskedArray,
171: (4)          this function will return a MaskedArray object instead of an ndarray,
172: (4)          but the input masks are *not* preserved. In cases where a MaskedArray
173: (4)          is expected as input, use the ma.concatenate function from the masked
174: (4)          array module instead.
175: (4)          Examples
176: (4)          -----
177: (4)          >>> a = np.array([[1, 2], [3, 4]])
178: (4)          >>> b = np.array([[5, 6]])
179: (4)          >>> np.concatenate((a, b), axis=0)
180: (4)          array([[1, 2],
181: (11)            [3, 4],
182: (11)            [5, 6]])
183: (4)          >>> np.concatenate((a, b.T), axis=1)
184: (4)          array([[1, 2, 5],
185: (11)            [3, 4, 6]])
186: (4)          >>> np.concatenate((a, b), axis=None)
187: (4)          array([1, 2, 3, 4, 5, 6])
188: (4)          This function will not preserve masking of MaskedArray inputs.
189: (4)          >>> a = np.ma.arange(3)
190: (4)          >>> a[1] = np.ma.masked
191: (4)          >>> b = np.arange(2, 5)
192: (4)          >>> a
193: (4)          masked_array(data=[0, --, 2],
194: (17)            mask=[False, True, False],
195: (11)            fill_value=999999)
196: (4)          >>> b
197: (4)          array([2, 3, 4])
198: (4)          >>> np.concatenate([a, b])
199: (4)          masked_array(data=[0, 1, 2, 2, 3, 4],
200: (17)            mask=False,
201: (11)            fill_value=999999)
202: (4)          >>> np.ma.concatenate([a, b])
203: (4)          masked_array(data=[0, --, 2, 2, 3, 4],
204: (17)            mask=[False, True, False, False, False, False],
205: (11)            fill_value=999999)
206: (4)          """
207: (4)          if out is not None:
208: (8)              arrays = list(arrays)
209: (8)              arrays.append(out)

```

```

210: (4)
211: (0)
212: (0)
213: (4)
214: (4)
215: (4)
216: (4)
217: (4)
218: (4)
219: (4)
220: (4)
221: (8)
222: (4)
223: (4)
224: (4)
225: (8)
226: (8)
227: (8)
228: (8)
229: (4)
230: (4)
231: (4)
232: (8)
233: (8)
234: (4)
235: (4)
236: (4)
237: (4)
238: (4)
239: (4)
240: (4)
241: (4)
242: (8)
243: (4)
244: (8)
245: (4)
246: (8)
247: (13)
248: (4)
249: (7)
250: (4)
251: (4)
252: (4)
253: (4)
254: (4)
255: (4)
256: (4)
257: (4)
258: (4)
259: (4)
260: (4)
261: (4)
262: (4)
263: (4)
264: (4)
265: (11)
266: (4)
267: (4)
268: (4)
269: (4)
270: (4)
271: (4)
272: (4)
273: (4)
274: (4)
275: (4)
276: (11)
277: (4)
278: (4)

        return arrays
@array_function_from_c_func_and_dispatcher(_multiarray_umath.inner)
def inner(a, b):
    """
    inner(a, b, /)
    Inner product of two arrays.
    Ordinary inner product of vectors for 1-D arrays (without complex
    conjugation), in higher dimensions a sum product over the last axes.
    Parameters
    -----
    a, b : array_like
        If `a` and `b` are nonscalar, their last dimensions must match.
    Returns
    -----
    out : ndarray
        If `a` and `b` are both
        scalars or both 1-D arrays then a scalar is returned; otherwise
        an array is returned.
        ``out.shape = (*a.shape[:-1], *b.shape[:-1])``
    Raises
    -----
    ValueError
        If both `a` and `b` are nonscalar and their last dimensions have
        different sizes.
    See Also
    -----
    tensordot : Sum products over arbitrary axes.
    dot : Generalised matrix product, using second last dimension of `b` .
    einsum : Einstein summation convention.
    Notes
    -----
    For vectors (1-D arrays) it computes the ordinary inner-product::
        np.inner(a, b) = sum(a[:]*b[:])
    More generally, if ``ndim(a) = r > 0`` and ``ndim(b) = s > 0``::
        np.inner(a, b) = np.tensordot(a, b, axes=(-1,-1))
    or explicitly::
        np.inner(a, b)[i0,...,ir-2,j0,...,js-2]
        = sum(a[i0,...,ir-2,:]*b[j0,...,js-2,:])
    In addition `a` or `b` may be scalars, in which case::
        np.inner(a,b) = a*b
    Examples
    -----
    Ordinary inner product for vectors:
    >>> a = np.array([1,2,3])
    >>> b = np.array([0,1,0])
    >>> np.inner(a, b)
    2
    Some multidimensional examples:
    >>> a = np.arange(24).reshape((2,3,4))
    >>> b = np.arange(4)
    >>> c = np.inner(a, b)
    >>> c.shape
    (2, 3)
    >>> c
    array([[ 14,  38,  62],
           [ 86, 110, 134]])
    >>> a = np.arange(2).reshape((1,1,2))
    >>> b = np.arange(6).reshape((3,2))
    >>> c = np.inner(a, b)
    >>> c.shape
    (1, 1, 3)
    >>> c
    array([[[1, 3, 5]]])
    An example where `b` is a scalar:
    >>> np.inner(np.eye(2), 7)
    array([[7., 0.],
           [0., 7.]])
    """
    return (a, b)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

279: (0) @array_function_from_c_func_and_dispatcher(_multiarray_umath.where)
280: (0) def where(condition, x=None, y=None):
281: (4)     """
282: (4)         where(condition, [x, y], /)
283: (4)         Return elements chosen from `x` or `y` depending on `condition`.
284: (4)         .. note::
285: (8)             When only `condition` is provided, this function is a shorthand for
286: (8)             ``np.asarray(condition).nonzero()``. Using `nonzero` directly should
be
287: (8)             preferred, as it behaves correctly for subclasses. The rest of this
288: (8)             documentation covers only the case where all three arguments are
289: (8)             provided.
290: (4)         Parameters
291: (4)         -----
292: (4)             condition : array_like, bool
293: (8)                 Where True, yield `x`, otherwise yield `y`.
294: (4)             x, y : array_like
295: (8)                 Values from which to choose. `x`, `y` and `condition` need to be
296: (8)                 broadcastable to some shape.
297: (4)         Returns
298: (4)         -----
299: (4)             out : ndarray
300: (8)                 An array with elements from `x` where `condition` is True, and
elements
301: (8)                 from `y` elsewhere.
302: (4)         See Also
303: (4)         -----
304: (4)             choose
305: (4)             nonzero : The function that is called when x and y are omitted
306: (4)         Notes
307: (4)         -----
308: (4)             If all the arrays are 1-D, `where` is equivalent to::
309: (8)                 [xv if c else yv
310: (9)                     for c, xv, yv in zip(condition, x, y)]
311: (4)         Examples
312: (4)         -----
313: (4)             >>> a = np.arange(10)
314: (4)             >>> a
315: (4)             array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
316: (4)             >>> np.where(a < 5, a, 10*a)
317: (4)             array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
318: (4)             This can be used on multidimensional arrays too:
319: (4)             >>> np.where([[True, False], [True, True]],
320: (4)                         [[1, 2], [3, 4]],
321: (4)                         [[9, 8], [7, 6]])
322: (4)             array([[1, 8],
323: (11)                 [3, 4]])
324: (4)             The shapes of x, y, and the condition are broadcast together:
325: (4)             >>> x, y = np.ogrid[:3, :4]
326: (4)             >>> np.where(x < y, x, 10 + y) # both x and 10+y are broadcast
327: (4)             array([[10,  0,  0,  0],
328: (11)                 [10, 11,  1,  1],
329: (11)                 [10, 11, 12,  2]])
330: (4)             >>> a = np.array([[0, 1, 2],
331: (4)                         [0, 2, 4],
332: (4)                         [0, 3, 6]])
333: (4)             >>> np.where(a < 4, a, -1) # -1 is broadcast
334: (4)             array([[ 0,  1,  2],
335: (11)                 [ 0,  2, -1],
336: (11)                 [ 0,  3, -1]])
337: (4)             """
338: (4)             return (condition, x, y)
339: (0) @array_function_from_c_func_and_dispatcher(_multiarray_umath.lexsort)
340: (0) def lexsort(keys, axis=None):
341: (4)     """
342: (4)         lexsort(keys, axis=-1)
343: (4)         Perform an indirect stable sort using a sequence of keys.
344: (4)         Given multiple sorting keys, which can be interpreted as columns in a
spreadsheet, lexsort returns an array of integer indices that describes

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

346: (4)          the sort order by multiple columns. The last key in the sequence is used
347: (4)          for the primary sort order, the second-to-last key for the secondary sort
348: (4)          order, and so on. The keys argument must be a sequence of objects that
349: (4)          can be converted to arrays of the same shape. If a 2D array is provided
350: (4)          for the keys argument, its rows are interpreted as the sorting keys and
351: (4)          sorting is according to the last row, second last row etc.
352: (4)          Parameters
353: (4)
354: (4)          keys : (k, N) array or tuple containing k (N,) -shaped sequences
355: (8)          The `k` different "columns" to be sorted. The last column (or row if
356: (8)          `keys` is a 2D array) is the primary sort key.
357: (4)          axis : int, optional
358: (8)          Axis to be indirectly sorted. By default, sort over the last axis.
359: (4)          Returns
360: (4)
361: (4)          indices : (N,) ndarray of ints
362: (8)          Array of indices that sort the keys along the specified axis.
363: (4)          See Also
364: (4)
365: (4)          argsort : Indirect sort.
366: (4)          ndarray.sort : In-place sort.
367: (4)          sort : Return a sorted copy of an array.
368: (4)          Examples
369: (4)
370: (4)          Sort names: first by surname, then by name.
371: (4)          >>> surnames = ('Hertz', 'Galilei', 'Hertz')
372: (4)          >>> first_names = ('Heinrich', 'Galileo', 'Gustav')
373: (4)          >>> ind = np.lexsort((first_names, surnames))
374: (4)          >>> ind
375: (4)          array([1, 2, 0])
376: (4)          >>> [surnames[i] + ", " + first_names[i] for i in ind]
377: (4)          ['Galilei, Galileo', 'Hertz, Gustav', 'Hertz, Heinrich']
378: (4)          Sort two columns of numbers:
379: (4)          >>> a = [1, 5, 1, 4, 3, 4, 4] # First column
380: (4)          >>> b = [9, 4, 0, 4, 0, 2, 1] # Second column
381: (4)          >>> ind = np.lexsort((b, a)) # Sort by a, then by b
382: (4)          >>> ind
383: (4)          array([2, 0, 4, 6, 5, 3, 1])
384: (4)          >>> [(a[i], b[i]) for i in ind]
385: (4)          [(1, 0), (1, 9), (3, 0), (4, 1), (4, 2), (4, 4), (5, 4)]
386: (4)          Note that sorting is first according to the elements of ``a``.
387: (4)          Secondary sorting is according to the elements of ``b``.
388: (4)          A normal ``argsort`` would have yielded:
389: (4)          >>> [(a[i], b[i]) for i in np.argsort(a)]
390: (4)          [(1, 9), (1, 0), (3, 0), (4, 4), (4, 2), (4, 1), (5, 4)]
391: (4)          Structured arrays are sorted lexically by ``argsort``:
392: (4)          >>> x = np.array([(1, 9), (5, 4), (1, 0), (4, 4), (3, 0), (4, 2), (4, 1)], ...
393: (4)                      dtype=np.dtype([('x', int), ('y', int)]))
394: (4)          >>> np.argsort(x) # or np.argsort(x, order=('x', 'y'))
395: (4)          array([2, 0, 4, 6, 5, 3, 1])
396: (4)          """
397: (4)          if isinstance(keys, tuple):
398: (8)              return keys
399: (4)
400: (8)          else:
401: (0)              return (keys,)
402: (0)          @array_function_from_c_func_and_dispatcher(_multiarray_umath.can_cast)
403: (4)          def can_cast(from_, to, casting=None):
404: (4)              """
405: (4)                  can_cast(from_, to, casting='safe')
406: (4)                  Returns True if cast between data types can occur according to the
407: (4)                  casting rule. If from is a scalar or array scalar, also returns
408: (4)                  True if the scalar value can be cast without overflow or truncation
409: (4)                  to an integer.
410: (4)                  Parameters
411: (4)
412: (8)                  from_ : dtype, dtype specifier, scalar, or array
413: (4)                      Data type, scalar, or array to cast from.
414: (8)                  to : dtype or dtype specifier
415: (4)                      Data type to cast to.

```

```

415: (4)           casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
416: (8)           Controls what kind of data casting may occur.
417: (10)          * 'no' means the data types should not be cast at all.
418: (10)          * 'equiv' means only byte-order changes are allowed.
419: (10)          * 'safe' means only casts which can preserve values are allowed.
420: (10)          * 'same_kind' means only safe casts or casts within a kind,
421: (12)            like float64 to float32, are allowed.
422: (10)          * 'unsafe' means any data conversions may be done.
423: (4)           Returns
424: (4)           -----
425: (4)           out : bool
426: (8)             True if cast can occur according to the casting rule.
427: (4)           Notes
428: (4)           -----
429: (4)             .. versionchanged:: 1.17.0
430: (7)               Casting between a simple data type and a structured one is possible
only
431: (7)               for "unsafe" casting. Casting to multiple fields is allowed, but
432: (7)               casting from multiple fields is not.
433: (4)             .. versionchanged:: 1.9.0
434: (7)               Casting from numeric to string types in 'safe' casting mode requires
435: (7)               that the string dtype length is long enough to store the maximum
436: (7)               integer/float value converted.
437: (4)           See also
438: (4)           -----
439: (4)           dtype, result_type
440: (4)           Examples
441: (4)           -----
442: (4)             Basic examples
443: (4)             >>> np.can_cast(np.int32, np.int64)
444: (4)             True
445: (4)             >>> np.can_cast(np.float64, complex)
446: (4)             True
447: (4)             >>> np.can_cast(complex, float)
448: (4)             False
449: (4)             >>> np.can_cast('i8', 'f8')
450: (4)             True
451: (4)             >>> np.can_cast('i8', 'f4')
452: (4)             False
453: (4)             >>> np.can_cast('i4', 'S4')
454: (4)             False
455: (4)             Casting scalars
456: (4)             >>> np.can_cast(100, 'i1')
457: (4)             True
458: (4)             >>> np.can_cast(150, 'i1')
459: (4)             False
460: (4)             >>> np.can_cast(150, 'u1')
461: (4)             True
462: (4)             >>> np.can_cast(3.5e100, np.float32)
463: (4)             False
464: (4)             >>> np.can_cast(1000.0, np.float32)
465: (4)             True
466: (4)             Array scalar checks the value, array does not
467: (4)             >>> np.can_cast(np.array(1000.0), np.float32)
468: (4)             True
469: (4)             >>> np.can_cast(np.array([1000.0]), np.float32)
470: (4)             False
471: (4)             Using the casting rules
472: (4)             >>> np.can_cast('i8', 'i8', 'no')
473: (4)             True
474: (4)             >>> np.can_cast('<i8', '>i8', 'no')
475: (4)             False
476: (4)             >>> np.can_cast('<i8', '>i8', 'equiv')
477: (4)             True
478: (4)             >>> np.can_cast('<i4', '>i8', 'equiv')
479: (4)             False
480: (4)             >>> np.can_cast('<i4', '>i8', 'safe')
481: (4)             True
482: (4)             >>> np.can_cast('<i8', '>i4', 'safe')

```

```

483: (4)             False
484: (4)             >>> np.can_cast('<i8', '>i4', 'same_kind')
485: (4)             True
486: (4)             >>> np.can_cast('<i8', '>u4', 'same_kind')
487: (4)             False
488: (4)             >>> np.can_cast('<i8', '>u4', 'unsafe')
489: (4)             True
490: (4)             """
491: (4)             return (from_,)
492: (0) @array_function_from_c_func_and_dispatcher(_multiarray_umath.min_scalar_type)
493: (0) def min_scalar_type(a):
494: (4) """
495: (4)     min_scalar_type(a, /)
496: (4)     For scalar ``a``, returns the data type with the smallest size
497: (4)     and smallest scalar kind which can hold its value. For non-scalar
498: (4)     array ``a``, returns the vector's dtype unmodified.
499: (4)     Floating point values are not demoted to integers,
500: (4)     and complex values are not demoted to floats.
501: (4)     Parameters
502: (4)     -----
503: (4)     a : scalar or array_like
504: (8)         The value whose minimal data type is to be found.
505: (4)     Returns
506: (4)     -----
507: (4)     out : dtype
508: (8)         The minimal data type.
509: (4)     Notes
510: (4)     -----
511: (4)     .. versionadded:: 1.6.0
512: (4)     See Also
513: (4)     -----
514: (4)     result_type, promote_types, dtype, can_cast
515: (4)     Examples
516: (4)     -----
517: (4)     >>> np.min_scalar_type(10)
518: (4)     dtype('uint8')
519: (4)     >>> np.min_scalar_type(-260)
520: (4)     dtype('int16')
521: (4)     >>> np.min_scalar_type(3.1)
522: (4)     dtype('float16')
523: (4)     >>> np.min_scalar_type(1e50)
524: (4)     dtype('float64')
525: (4)     >>> np.min_scalar_type(np.arange(4,dtype='f8'))
526: (4)     dtype('float64')
527: (4)     """
528: (4)             return (a,)
529: (0) @array_function_from_c_func_and_dispatcher(_multiarray_umath.result_type)
530: (0) def result_type(*arrays_and_dtypes):
531: (4) """
532: (4)     result_type(*arrays_and_dtypes)
533: (4)     Returns the type that results from applying the NumPy
534: (4)     type promotion rules to the arguments.
535: (4)     Type promotion in NumPy works similarly to the rules in languages
536: (4)     like C++, with some slight differences. When both scalars and
537: (4)     arrays are used, the array's type takes precedence and the actual value
538: (4)     of the scalar is taken into account.
539: (4)     For example, calculating 3*a, where a is an array of 32-bit floats,
540: (4)     intuitively should result in a 32-bit float output. If the 3 is a
541: (4)     32-bit integer, the NumPy rules indicate it can't convert losslessly
542: (4)     into a 32-bit float, so a 64-bit float should be the result type.
543: (4)     By examining the value of the constant, '3', we see that it fits in
544: (4)     an 8-bit integer, which can be cast losslessly into the 32-bit float.
545: (4)     Parameters
546: (4)     -----
547: (4)     arrays_and_dtypes : list of arrays and dtypes
548: (8)         The operands of some operation whose result type is needed.
549: (4)     Returns
549: (4)     -----
550: (4)     out : dtype
551: (4)

```

```

552: (8)           The result type.
553: (4)           See also
554: (4)
555: (4)           -----
556: (4)           dtype, promote_types, min_scalar_type, can_cast
557: (4)
558: (4)           Notes
559: (4)           -----
560: (4)           .. versionadded:: 1.6.0
561: (4)           The specific algorithm used is as follows.
562: (4)           Categories are determined by first checking which of boolean,
563: (4)           integer (int/uint), or floating point (float/complex) the maximum
564: (4)           kind of all the arrays and the scalars are.
565: (4)           If there are only scalars or the maximum category of the scalars
566: (4)           is higher than the maximum category of the arrays,
567: (4)           the data types are combined with :func:`promote_types`  

568: (4)           to produce the return value.
569: (4)           Otherwise, `min_scalar_type` is called on each scalar, and
570: (4)           the resulting data types are all combined with :func:`promote_types`  

571: (4)           to produce the return value.
572: (4)           The set of int values is not a subset of the uint values for types
573: (4)           with the same number of bits, something not reflected in
574: (4)           :func:`min_scalar_type`, but handled as a special case in `result_type`.
575: (4)           Examples
576: (4)           -----
577: (4)           >>> np.result_type(3, np.arange(7, dtype='i1'))
578: (4)           dtype('int8')
579: (4)           >>> np.result_type('i4', 'c8')
580: (4)           dtype('complex128')
581: (4)           >>> np.result_type(3.0, -2)
582: (4)           dtype('float64')
583: (0)           """
584: (0)           return arrays_and_dtypes
585: (4)           @array_function_from_c_func_and_dispatcher(_multiarray_umath.dot)
586: (4)           def dot(a, b, out=None):
587: (4)               """
588: (4)               dot(a, b, out=None)
589: (6)               Dot product of two arrays. Specifically,
590: (4)               - If both `a` and `b` are 1-D arrays, it is inner product of vectors
591: (6)                   (without complex conjugation).
592: (4)               - If both `a` and `b` are 2-D arrays, it is matrix multiplication,
593: (6)                   but using :func:`matmul` or ``a @ b`` is preferred.
594: (4)               - If either `a` or `b` is 0-D (scalar), it is equivalent to
595: (6)                   :func:`multiply` and using ``numpy.multiply(a, b)`` or ``a * b`` is
596: (6)                   preferred.
597: (4)               - If `a` is an N-D array and `b` is a 1-D array, it is a sum product over
598: (6)                   the last axis of `a` and `b`.
599: (6)               - If `a` is an N-D array and `b` is an M-D array (where ``M>=2``), it is a
600: (8)                   sum product over the last axis of `a` and the second-to-last axis of
601: (4)                   `b`::
602: (4)                   dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,:m])
603: (4)               It uses an optimized BLAS library when possible (see `numpy.linalg`).
604: (4)               Parameters
605: (8)               a : array_like
606: (4)                   First argument.
607: (8)               b : array_like
608: (4)                   Second argument.
609: (8)               out : ndarray, optional
610: (8)                   Output argument. This must have the exact kind that would be returned
611: (8)                   if it was not used. In particular, it must have the right type, must
612: (8)                   be C-contiguous, and its dtype must be the dtype that would be returned
613: (8)                   for `dot(a,b)`. This is a performance feature. Therefore, if these
614: (8)                   conditions are not met, an exception is raised, instead of attempting
615: (4)                   to be flexible.
616: (4)               Returns
617: (4)               -----
618: (8)               output : ndarray
619: (8)                   Returns the dot product of `a` and `b`. If `a` and `b` are both

```

```

620: (8)                                an array is returned.
621: (8)                                If `out` is given, then it is returned.
622: (4)                                Raises
623: (4)                                -----
624: (4)                                ValueError
625: (8)                                If the last dimension of `a` is not the same size as
626: (8)                                the second-to-last dimension of `b`.
627: (4)                                See Also
628: (4)                                -----
629: (4)                                vdot : Complex-conjugating dot product.
630: (4)                                tensordot : Sum products over arbitrary axes.
631: (4)                                einsum : Einstein summation convention.
632: (4)                                matmul : '@' operator as method with out parameter.
633: (4)                                linalg.multi_dot : Chained dot product.
634: (4)                                Examples
635: (4)                                -----
636: (4)                                >>> np.dot(3, 4)
637: (4)                                12
638: (4)                                Neither argument is complex-conjugated:
639: (4)                                >>> np.dot([2j, 3j], [2j, 3j])
640: (4)                                (-13+0j)
641: (4)                                For 2-D arrays it is the matrix product:
642: (4)                                >>> a = [[1, 0], [0, 1]]
643: (4)                                >>> b = [[4, 1], [2, 2]]
644: (4)                                >>> np.dot(a, b)
645: (4)                                array([[4, 1],
646: (11)                               [2, 2]])
647: (4)                                >>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
648: (4)                                >>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
649: (4)                                >>> np.dot(a, b)[2,3,2,1,2,2]
650: (4)                                499128
651: (4)                                >>> sum(a[2,3,2,:]*b[1,2,:,:2])
652: (4)                                499128
653: (4)                                """
654: (4)                                return (a, b, out)
655: (0) @array_function_from_c_func_and_dispatcher(_multiarray_umath.vdot)
656: (0) def vdot(a, b):
657: (4)     """
658: (4)     vdot(a, b, /)
659: (4)     Return the dot product of two vectors.
660: (4)     The vdot(`a`, `b`) function handles complex numbers differently than
661: (4)     dot(`a`, `b`). If the first argument is complex the complex conjugate
662: (4)     of the first argument is used for the calculation of the dot product.
663: (4)     Note that `vdot` handles multidimensional arrays differently than `dot`:
664: (4)     it does *not* perform a matrix product, but flattens input arguments
665: (4)     to 1-D vectors first. Consequently, it should only be used for vectors.
666: (4)     Parameters
667: (4)     -----
668: (4)     a : array_like
669: (8)     If `a` is complex the complex conjugate is taken before calculation
670: (8)     of the dot product.
671: (4)     b : array_like
672: (8)     Second argument to the dot product.
673: (4)     Returns
674: (4)     -----
675: (4)     output : ndarray
676: (8)     Dot product of `a` and `b`. Can be an int, float, or
677: (8)     complex depending on the types of `a` and `b`.
678: (4)     See Also
679: (4)     -----
680: (4)     dot : Return the dot product without using the complex conjugate of the
681: (10)    first argument.
682: (4)     Examples
683: (4)     -----
684: (4)     >>> a = np.array([1+2j, 3+4j])
685: (4)     >>> b = np.array([5+6j, 7+8j])
686: (4)     >>> np.vdot(a, b)
687: (4)     (70-8j)
688: (4)     >>> np.vdot(b, a)

```

```

689: (4) (70+8j)
690: (4) Note that higher-dimensional arrays are flattened!
691: (4) >>> a = np.array([[1, 4], [5, 6]])
692: (4) >>> b = np.array([[4, 1], [2, 2]])
693: (4) >>> np.vdot(a, b)
694: (4) 30
695: (4) >>> np.vdot(b, a)
696: (4) 30
697: (4) >>> 1*4 + 4*1 + 5*2 + 6*2
698: (4) 30
699: (4) """
700: (4)     return (a, b)
701: (0) @array_function_from_c_func_and_dispatcher(_multiarray_umath.bincount)
702: (0) def bincount(x, weights=None, minlength=None):
703: (4) """
704: (4)     bincount(x, /, weights=None, minlength=0)
705: (4)     Count number of occurrences of each value in array of non-negative ints.
706: (4)     The number of bins (of size 1) is one larger than the largest value in
707: (4)     `x`. If `minlength` is specified, there will be at least this number
708: (4)     of bins in the output array (though it will be longer if necessary,
709: (4)     depending on the contents of `x`).
710: (4)     Each bin gives the number of occurrences of its index value in `x`.
711: (4)     If `weights` is specified the input array is weighted by it, i.e. if a
712: (4)     value ``n`` is found at position ``i``, ``out[n] += weight[i]`` instead
713: (4)     of ``out[n] += 1``.
714: (4)     Parameters
715: (4)     -----
716: (4)     x : array_like, 1 dimension, nonnegative ints
717: (8)         Input array.
718: (4)     weights : array_like, optional
719: (8)         Weights, array of the same shape as `x`.
720: (4)     minlength : int, optional
721: (8)         A minimum number of bins for the output array.
722: (8)         .. versionadded:: 1.6.0
723: (4)     Returns
724: (4)     -----
725: (4)     out : ndarray of ints
726: (8)         The result of binning the input array.
727: (8)         The length of `out` is equal to ``np.amax(x)+1``.
728: (4)     Raises
729: (4)     -----
730: (4)     ValueError
731: (8)         If the input is not 1-dimensional, or contains elements with negative
732: (8)         values, or if `minlength` is negative.
733: (4)     TypeError
734: (8)         If the type of the input is float or complex.
735: (4)     See Also
736: (4)     -----
737: (4)     histogram, digitize, unique
738: (4)     Examples
739: (4)     -----
740: (4)     >>> np.bincount(np.arange(5))
741: (4)     array([1, 1, 1, 1, 1])
742: (4)     >>> np.bincount(np.array([0, 1, 1, 3, 2, 1, 7]))
743: (4)     array([1, 3, 1, 1, 0, 0, 0, 1])
744: (4)     >>> x = np.array([0, 1, 1, 3, 2, 1, 7, 23])
745: (4)     >>> np.bincount(x).size == np.amax(x)+1
746: (4)     True
747: (4)     The input array needs to be of integer dtype, otherwise a
748: (4)     TypeError is raised:
749: (4)     >>> np.bincount(np.arange(5, dtype=float))
750: (4)     Traceback (most recent call last):
751: (6)     ...
752: (4)     TypeError: Cannot cast array data from dtype('float64') to dtype('int64')
753: (4)     according to the rule 'safe'
754: (4)     A possible use of ``bincount`` is to perform sums over
755: (4)     variable-size chunks of an array, using the ``weights`` keyword.
756: (4)     >>> w = np.array([0.3, 0.5, 0.2, 0.7, 1., -0.6]) # weights
757: (4)     >>> x = np.array([0, 1, 1, 2, 2, 2])

```

```

758: (4)             >>> np.bincount(x, weights=w)
759: (4)             array([ 0.3,  0.7,  1.1])
760: (4)             """
761: (4)             return (x, weights)
762: (0)

@array_function_from_c_func_and_dispatcher(_multiarray_umath.ravel_multi_index)
763: (0)     def ravel_multi_index(multi_index, dims, mode=None, order=None):
764: (4)         """
765: (4)             ravel_multi_index(multi_index, dims, mode='raise', order='C')
766: (4)             Converts a tuple of index arrays into an array of flat
767: (4)             indices, applying boundary modes to the multi-index.
768: (4)             Parameters
769: (4)             -----
770: (4)             multi_index : tuple of array_like
771: (8)                 A tuple of integer arrays, one array for each dimension.
772: (4)             dims : tuple of ints
773: (8)                 The shape of array into which the indices from ``multi_index`` apply.
774: (4)             mode : {'raise', 'wrap', 'clip'}, optional
775: (8)                 Specifies how out-of-bounds indices are handled. Can specify
776: (8)                 either one mode or a tuple of modes, one mode per index.
777: (8)                 * 'raise' -- raise an error (default)
778: (8)                 * 'wrap' -- wrap around
779: (8)                 * 'clip' -- clip to the range
780: (8)                 In 'clip' mode, a negative index which would normally
781: (8)                 wrap will clip to 0 instead.
782: (4)             order : {'C', 'F'}, optional
783: (8)                 Determines whether the multi-index should be viewed as
784: (8)                 indexing in row-major (C-style) or column-major
785: (8)                 (Fortran-style) order.
786: (4)             Returns
787: (4)             -----
788: (4)             raveled_indices : ndarray
789: (8)                 An array of indices into the flattened version of an array
790: (8)                 of dimensions ``dims``.
791: (4)             See Also
792: (4)             -----
793: (4)             unravel_index
794: (4)             Notes
795: (4)             -----
796: (4)             .. versionadded:: 1.6.0
797: (4)             Examples
798: (4)             -----
799: (4)             >>> arr = np.array([[3,6,6],[4,5,1]])
800: (4)             >>> np.ravel_multi_index(arr, (7,6))
801: (4)             array([22, 41, 37])
802: (4)             >>> np.ravel_multi_index(arr, (7,6), order='F')
803: (4)             array([31, 41, 13])
804: (4)             >>> np.ravel_multi_index(arr, (4,6), mode='clip')
805: (4)             array([22, 23, 19])
806: (4)             >>> np.ravel_multi_index(arr, (4,4), mode=('clip','wrap'))
807: (4)             array([12, 13, 13])
808: (4)             >>> np.ravel_multi_index((3,1,4,1), (6,7,8,9))
809: (4)             1621
810: (4)             """
811: (4)             return multi_index
812: (0) @array_function_from_c_func_and_dispatcher(_multiarray_umath.unravel_index)
813: (0)     def unravel_index(indices, shape=None, order=None):
814: (4)         """
815: (4)             unravel_index(indices, shape, order='C')
816: (4)             Converts a flat index or array of flat indices into a tuple
817: (4)             of coordinate arrays.
818: (4)             Parameters
819: (4)             -----
820: (4)             indices : array_like
821: (8)                 An integer array whose elements are indices into the flattened
822: (8)                 version of an array of dimensions ``shape``. Before version 1.6.0,
823: (8)                 this function accepted just one index value.
824: (4)             shape : tuple of ints
825: (8)                 The shape of the array to use for unraveling ``indices``.

```

```

826: (8)          .. versionchanged:: 1.16.0
827: (12)         Renamed from ``dims`` to ``shape``.
828: (4)          order : {'C', 'F'}, optional
829: (8)         Determines whether the indices should be viewed as indexing in
830: (8)         row-major (C-style) or column-major (Fortran-style) order.
831: (8)         .. versionadded:: 1.6.0
832: (4)          Returns
833: (4)          -----
834: (4)          unraveled_coords : tuple of ndarray
835: (8)         Each array in the tuple has the same shape as the ``indices``
836: (8)         array.
837: (4)          See Also
838: (4)          -----
839: (4)          ravel_multi_index
840: (4)          Examples
841: (4)          -----
842: (4)          >>> np.unravel_index([22, 41, 37], (7,6))
843: (4)          (array([3, 6, 6]), array([4, 5, 1]))
844: (4)          >>> np.unravel_index([31, 41, 13], (7,6), order='F')
845: (4)          (array([3, 6, 6]), array([4, 5, 1]))
846: (4)          >>> np.unravel_index(1621, (6,7,8,9))
847: (4)          (3, 1, 4, 1)
848: (4)          """
849: (4)          return (indices,)
850: (0)          @array_function_from_c_func_and_dispatcher(_multiarray_umath.copyto)
851: (0)          def copyto(dst, src, casting=None, where=None):
852: (4)          """
853: (4)          copyto(dst, src, casting='same_kind', where=True)
854: (4)          Copies values from one array to another, broadcasting as necessary.
855: (4)          Raises a TypeError if the `casting` rule is violated, and if
856: (4)          `where` is provided, it selects which elements to copy.
857: (4)          .. versionadded:: 1.7.0
858: (4)          Parameters
859: (4)          -----
860: (4)          dst : ndarray
861: (8)          The array into which values are copied.
862: (4)          src : array_like
863: (8)          The array from which values are copied.
864: (4)          casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
865: (8)          Controls what kind of data casting may occur when copying.
866: (10)         * 'no' means the data types should not be cast at all.
867: (10)         * 'equiv' means only byte-order changes are allowed.
868: (10)         * 'safe' means only casts which can preserve values are allowed.
869: (10)         * 'same_kind' means only safe casts or casts within a kind,
870: (12)          like float64 to float32, are allowed.
871: (10)         * 'unsafe' means any data conversions may be done.
872: (4)          where : array_like of bool, optional
873: (8)          A boolean array which is broadcasted to match the dimensions
874: (8)          of `dst`, and selects elements to copy from `src` to `dst`
875: (8)          wherever it contains the value True.
876: (4)          Examples
877: (4)          -----
878: (4)          >>> A = np.array([4, 5, 6])
879: (4)          >>> B = [1, 2, 3]
880: (4)          >>> np.copyto(A, B)
881: (4)          >>> A
882: (4)          array([1, 2, 3])
883: (4)          >>> A = np.array([[1, 2, 3], [4, 5, 6]])
884: (4)          >>> B = [[4, 5, 6], [7, 8, 9]]
885: (4)          >>> np.copyto(A, B)
886: (4)          >>> A
887: (4)          array([[4, 5, 6],
888: (11)            [7, 8, 9]])
889: (4)          """
890: (4)          return (dst, src, where)
891: (0)          @array_function_from_c_func_and_dispatcher(_multiarray_umath.putmask)
892: (0)          def putmask(a, /, mask, values):
893: (4)          """
894: (4)          putmask(a, mask, values)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

895: (4)          Changes elements of an array based on conditional and input values.
896: (4)          Sets ``a.flat[n] = values[n]`` for each n where ``mask.flat[n]==True``.
897: (4)          If `values` is not the same size as `a` and `mask` then it will repeat.
898: (4)          This gives behavior different from ``a[mask] = values``.
899: (4)          Parameters
900: (4)          -----
901: (4)          a : ndarray
902: (8)          Target array.
903: (4)          mask : array_like
904: (8)          Boolean mask array. It has to be the same shape as `a`.
905: (4)          values : array_like
906: (8)          Values to put into `a` where `mask` is True. If `values` is smaller
907: (8)          than `a` it will be repeated.
908: (4)          See Also
909: (4)          -----
910: (4)          place, put, take, copyto
911: (4)          Examples
912: (4)          -----
913: (4)          >>> x = np.arange(6).reshape(2, 3)
914: (4)          >>> np.putmask(x, x>2, x**2)
915: (4)          >>> x
916: (4)          array([[ 0,  1,  2],
917: (11)             [ 9, 16, 25]])
918: (4)          If `values` is smaller than `a` it is repeated:
919: (4)          >>> x = np.arange(5)
920: (4)          >>> np.putmask(x, x>1, [-33, -44])
921: (4)          >>> x
922: (4)          array([ 0,  1, -33, -44, -33])
923: (4)          """
924: (4)          return (a, mask, values)
925: (0)          @array_function_from_c_func_and_dispatcher(_multiarray_umath.packbits)
926: (0)          def packbits(a, axis=None, bitorder='big'):
927: (4)          """
928: (4)          packbits(a, /, axis=None, bitorder='big')
929: (4)          Packs the elements of a binary-valued array into bits in a uint8 array.
930: (4)          The result is padded to full bytes by inserting zero bits at the end.
931: (4)          Parameters
932: (4)          -----
933: (4)          a : array_like
934: (8)          An array of integers or booleans whose elements should be packed to
935: (8)          bits.
936: (4)          axis : int, optional
937: (8)          The dimension over which bit-packing is done.
938: (8)          ``None`` implies packing the flattened array.
939: (4)          bitorder : {'big', 'little'}, optional
940: (8)          The order of the input bits. 'big' will mimic bin(val),
941: (8)          ``[0, 0, 0, 0, 0, 1, 1] => 3 = 0b00000011``, 'little' will
942: (8)          reverse the order so ``[1, 1, 0, 0, 0, 0, 0] => 3``.
943: (8)          Defaults to 'big'.
944: (8)          .. versionadded:: 1.17.0
945: (4)          Returns
946: (4)          -----
947: (4)          packed : ndarray
948: (8)          Array of type uint8 whose elements represent bits corresponding to the
949: (8)          logical (0 or nonzero) value of the input elements. The shape of
950: (8)          `packed` has the same number of dimensions as the input (unless `axis`
951: (8)          is None, in which case the output is 1-D).
952: (4)          See Also
953: (4)          -----
954: (4)          unpackbits: Unpacks elements of a uint8 array into a binary-valued output
955: (16)          array.
956: (4)          Examples
957: (4)          -----
958: (4)          >>> a = np.array([[[1,0,1],
959: (4)                      ...                  [0,1,0]],
960: (4)                      ...                  [[1,1,0],
961: (4)                      ...                  [0,0,1]]])
962: (4)          >>> b = np.packbits(a, axis=-1)
963: (4)          >>> b

```

```

964: (4)
965: (12)
966: (11)
967: (12)
968: (4)
969: (4)
970: (4)
971: (4)
972: (0)
973: (0)
974: (4)
975: (4)
976: (4)
977: (4)
978: (4)
979: (4)
980: (4)
981: (4)
982: (4)
983: (4)
984: (7)
985: (4)
986: (8)
987: (8)
988: (4)
989: (8)
990: (8)
991: (8)
992: (8)
993: (8)
994: (8)
995: (8)
996: (8)
997: (8)
998: (4)
999: (8)
1000: (8)
1001: (8)
1002: (8)
1003: (8)
1004: (4)
1005: (4)
1006: (4)
1007: (7)
1008: (4)
1009: (4)
1010: (4)
1011: (15)
1012: (4)
1013: (4)
1014: (4)
1015: (4)
1016: (4)
1017: (11)
1018: (11)
1019: (4)
1020: (4)
1021: (4)
1022: (11)
1023: (11)
1024: (4)
1025: (4)
1026: (4)
1027: (11)
1028: (11)
1029: (4)
1030: (4)
1031: (4)
1032: (11)

        array([[[160],
                   [ 64],
                   [[192],
                    [ 32]]], dtype=uint8)
    Note that in binary 160 = 1010 0000, 64 = 0100 0000, 192 = 1100 0000,
    and 32 = 0010 0000.
    """
    return (a,)
@array_function_from_c_func_and_dispatcher(_multiarray_umath.unpackbits)
def unpackbits(a, axis=None, count=None, bitorder='big'):
    """
    unpackbits(a, /, axis=None, count=None, bitorder='big')
    Unpacks elements of a uint8 array into a binary-valued output array.
    Each element of `a` represents a bit-field that should be unpacked
    into a binary-valued output array. The shape of the output array is
    either 1-D (if `axis` is ``None``) or the same shape as the input
    array with unpacking done along the axis specified.

    Parameters
    -----
    a : ndarray, uint8 type
        Input array.
    axis : int, optional
        The dimension over which bit-unpacking is done.
        ``None`` implies unpacking the flattened array.
    count : int or None, optional
        The number of elements to unpack along `axis`, provided as a way
        of undoing the effect of packing a size that is not a multiple
        of eight. A non-negative number means to only unpack `count`
        bits. A negative number means to trim off that many bits from
        the end. ``None`` means to unpack the entire array (the
        default). Counts larger than the available number of bits will
        add zero padding to the output. Negative counts must not
        exceed the available number of bits.
        .. versionadded:: 1.17.0
    bitorder : {'big', 'little'}, optional
        The order of the returned bits. 'big' will mimic bin(val),
        ``3 = 0b00000011 => [0, 0, 0, 0, 0, 1, 1]``, 'little' will reverse
        the order to ``[1, 1, 0, 0, 0, 0, 0]``.
        Defaults to 'big'.
        .. versionadded:: 1.17.0
    Returns
    -----
    unpacked : ndarray, uint8 type
        The elements are binary-valued (0 or 1).
    See Also
    -----
    packbits : Packs the elements of a binary-valued array into bits in
               a uint8 array.
    Examples
    -----
    >>> a = np.array([[2], [7], [23]], dtype=np.uint8)
    >>> a
    array([[ 2],
           [ 7],
           [23]], dtype=uint8)
    >>> b = np.unpackbits(a, axis=1)
    >>> b
    array([[0, 0, 0, 0, 0, 0, 1, 0],
           [0, 0, 0, 0, 0, 1, 1, 1],
           [0, 0, 0, 1, 0, 1, 1, 1]], dtype=uint8)
    >>> c = np.unpackbits(a, axis=1, count=-3)
    >>> c
    array([[0, 0, 0, 0, 0],
           [0, 0, 0, 0, 0],
           [0, 0, 0, 1, 0]], dtype=uint8)
    >>> p = np.packbits(b, axis=0)
    >>> np.unpackbits(p, axis=0)
    array([[0, 0, 0, 0, 0, 0, 1, 0],
           [0, 0, 0, 0, 0, 1, 1, 1],
           [0, 0, 0, 1, 0, 1, 1, 1]], dtype=uint8)

```

```

1033: (11) [0, 0, 0, 1, 0, 1, 1, 1, 1],
1034: (11) [0, 0, 0, 0, 0, 0, 0, 0, 0],
1035: (11) [0, 0, 0, 0, 0, 0, 0, 0, 0],
1036: (11) [0, 0, 0, 0, 0, 0, 0, 0, 0],
1037: (11) [0, 0, 0, 0, 0, 0, 0, 0, 0],
1038: (11) [0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=uint8)
1039: (4)     >>> np.array_equal(b, np.unpackbits(p, axis=0, count=b.shape[0]))
1040: (4)     True
1041: (4)     """
1042: (4)         return (a,)
1043: (0) @array_function_from_c_func_and_dispatcher(_multiarray_umath.shares_memory)
1044: (0) def shares_memory(a, b, max_work=None):
1045: (4)     """
1046: (4)         shares_memory(a, b, /, max_work=None)
1047: (4)         Determine if two arrays share memory.
1048: (4)         .. warning::
1049: (7)             This function can be exponentially slow for some inputs, unless
1050: (7)             `max_work` is set to a finite number or ``MAY_SHARE_BOUNDS``.
1051: (7)             If in doubt, use `numpy.may_share_memory` instead.
1052: (4)     Parameters
1053: (4)     -----
1054: (4)         a, b : ndarray
1055: (8)             Input arrays
1056: (4)         max_work : int, optional
1057: (8)             Effort to spend on solving the overlap problem (maximum number
1058: (8)             of candidate solutions to consider). The following special
1059: (8)             values are recognized:
1060: (8)             max_work=MAY_SHARE_EXACT (default)
1061: (12)                 The problem is solved exactly. In this case, the function returns
1062: (12)                 True only if there is an element shared between the arrays.
Finding
1063: (12)                 the exact solution may take extremely long in some cases.
1064: (8)             max_work=MAY_SHARE_BOUNDS
1065: (12)                 Only the memory bounds of a and b are checked.
1066: (4)     Raises
1067: (4)     -----
1068: (4)         numpy.exceptions.TooHardError
1069: (8)             Exceeded max_work.
1070: (4)     Returns
1071: (4)     -----
1072: (4)         out : bool
1073: (4)     See Also
1074: (4)     -----
1075: (4)         may_share_memory
1076: (4)     Examples
1077: (4)     -----
1078: (4)     >>> x = np.array([1, 2, 3, 4])
1079: (4)     >>> np.shares_memory(x, np.array([5, 6, 7]))
1080: (4)     False
1081: (4)     >>> np.shares_memory(x[:2], x)
1082: (4)     True
1083: (4)     >>> np.shares_memory(x[:2], x[1::2])
1084: (4)     False
1085: (4)     Checking whether two arrays share memory is NP-complete, and
1086: (4)     runtime may increase exponentially in the number of
1087: (4)     dimensions. Hence, `max_work` should generally be set to a finite
1088: (4)     number, as it is possible to construct examples that take
1089: (4)     extremely long to run:
1090: (4)     >>> from numpy.lib.stride_tricks import as_strided
1091: (4)     >>> x = np.zeros([192163377], dtype=np.int8)
1092: (4)     >>> x1 = as_strided(x, strides=(36674, 61119, 85569), shape=(1049, 1049,
1049))
1093: (4)     >>> x2 = as_strided(x[64023025:], strides=(12223, 12224, 1), shape=(1049,
1049, 1))
1094: (4)     >>> np.shares_memory(x1, x2, max_work=1000)
Traceback (most recent call last):
...
numpy.exceptions.TooHardError: Exceeded max_work
Running ``np.shares_memory(x1, x2)`` without `max_work` set takes

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1099: (4)           around 1 minute for this case. It is possible to find problems
1100: (4)           that take still significantly longer.
1101: (4)
1102: (4)           """
1103: (0)           return (a, b)
1104: (0)           @array_function_from_c_func_and_dispatcher(_multiarray_umath.may_share_memory)
1105: (4)           def may_share_memory(a, b, max_work=None):
1106: (4)               """
1107: (4)               may_share_memory(a, b, /, max_work=None)
1108: (4)               Determine if two arrays might share memory
1109: (4)               A return of True does not necessarily mean that the two arrays
1110: (4)               share any element. It just means that they *might*.
1111: (4)               Only the memory bounds of a and b are checked by default.
1112: (4)           Parameters
1113: (4)               -----
1114: (8)               a, b : ndarray
1115: (4)                   Input arrays
1116: (8)               max_work : int, optional
1117: (8)                   Effort to spend on solving the overlap problem. See
1118: (8)                   `shares_memory` for details. Default for ``may_share_memory``
1119: (4)                   is to do a bounds check.
1120: (4)           Returns
1121: (4)               -----
1122: (4)               out : bool
1123: (4)           See Also
1124: (4)               -----
1125: (4)               shares_memory
1126: (4)           Examples
1127: (4)               -----
1128: (4)               >>> np.may_share_memory(np.array([1,2]), np.array([5,8,9]))
1129: (4)               False
1130: (4)               >>> x = np.zeros([3, 4])
1131: (4)               >>> np.may_share_memory(x[:,0], x[:,1])
1132: (4)               True
1133: (4)               """
1134: (0)           return (a, b)
1135: (0)           @array_function_from_c_func_and_dispatcher(_multiarray_umath.is_busday)
1136: (4)           def is_busday(dates, weekmask=None, holidays=None, out=None):
1137: (4)               """
1138: (4)               is_busday(dates, weekmask='1111100', holidays=None, busdaycal=None,
1139: (4)               out=None)
1140: (4)           Calculates which of the given dates are valid days, and which are not.
1141: (4)           .. versionadded:: 1.7.0
1142: (4)           Parameters
1143: (8)               -----
1144: (4)               dates : array_like of datetime64[D]
1145: (8)                   The array of dates to process.
1146: (8)               weekmask : str or array_like of bool, optional
1147: (8)                   A seven-element array indicating which of Monday through Sunday are
1148: (8)                   valid days. May be specified as a length-seven list or array, like
1149: (8)                   [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string
1150: (8)                   like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for
1151: (8)                   weekdays, optionally separated by white space. Valid abbreviations
1152: (8)                   are: Mon Tue Wed Thu Fri Sat Sun
1153: (8)               holidays : array_like of datetime64[D], optional
1154: (8)                   An array of dates to consider as invalid dates. They may be
1155: (8)                   specified in any order, and NaT (not-a-time) dates are ignored.
1156: (4)                   This list is saved in a normalized form that is suited for
1157: (8)                   fast calculations of valid days.
1158: (8)               busdaycal : busdaycalendar, optional
1159: (8)                   A `busdaycalendar` object which specifies the valid days. If this
1160: (8)                   parameter is provided, neither weekmask nor holidays may be
1161: (8)                   provided.
1162: (4)               out : array of bool, optional
1163: (4)                   If provided, this array is filled with the result.
1164: (4)           Returns
1165: (8)               -----
1166: (8)               out : array of bool
1167: (8)                   An array with the same shape as ``dates``, containing True for
1168: (8)                   each valid day, and False for each invalid day.

```

```

1167: (4)           See Also
1168: (4)
1169: (4)           -----
1170: (4)           busdaycalendar : An object that specifies a custom set of valid days.
1171: (4)           busday_offset : Applies an offset counted in valid days.
1172: (4)           busday_count : Counts how many valid days are in a half-open date range.
1173: (4)
1174: (4)           Examples
1175: (4)           -----
1176: (4)           >>> # The weekdays are Friday, Saturday, and Monday
1177: (4)           ... np.is_busday(['2011-07-01', '2011-07-02', '2011-07-18'],
1178: (4)           ...                      holidays=['2011-07-01', '2011-07-04', '2011-07-17'])
1179: (4)           array([False, False,  True])
1180: (0)           """
1181: (0)           return (dates, weekmask, holidays, out)
1182: (18)          @array_function_from_c_func_and_dispatcher(_multiarray_umath.busday_offset)
1183: (4)          def busday_offset(dates, offsets, roll=None, weekmask=None, holidays=None,
1184: (4)                      busdaycal=None, out=None):
1185: (4)                      """
1186: (4)                      busday_offset(dates, offsets, roll='raise', weekmask='1111100',
1187: (4)                      holidays=None, busdaycal=None, out=None)
1188: (4)                      First adjusts the date to fall on a valid day according to
1189: (4)                      the ``roll`` rule, then applies offsets to the given dates
1190: (4)                      counted in valid days.
1191: (4)                      .. versionadded:: 1.7.0
1192: (8)          Parameters
1193: (4)          -----
1194: (8)          dates : array_like of datetime64[D]
1195: (4)          The array of dates to process.
1196: (8)          offsets : array_like of int
1197: (4)          The array of offsets, which is broadcast with ``dates``.
1198: (8)          roll : {'raise', 'nat', 'forward', 'following', 'backward', 'preceding',
1199: (10)          'modifiedfollowing', 'modifiedpreceding'}, optional
1200: (10)          How to treat dates that do not fall on a valid day. The default
1201: (12)          is 'raise'.
1202: (10)          * 'raise' means to raise an exception for an invalid day.
1203: (12)          * 'nat' means to return a NaT (not-a-time) for an invalid day.
1204: (10)          * 'forward' and 'following' mean to take the first valid day
1205: (12)          later in time.
1206: (12)          * 'backward' and 'preceding' mean to take the first valid day
1207: (10)          earlier in time.
1208: (12)          * 'modifiedfollowing' means to take the first valid day
1209: (12)          later in time unless it is across a Month boundary, in which
1210: (4)          case to take the first valid day earlier in time.
1211: (8)          weekmask : str or array_like of bool, optional
1212: (8)          A seven-element array indicating which of Monday through Sunday are
1213: (8)          valid days. May be specified as a length-seven list or array, like
1214: (8)          [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string
1215: (8)          like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for
1216: (8)          weekdays, optionally separated by white space. Valid abbreviations
1217: (4)          are: Mon Tue Wed Thu Fri Sat Sun
1218: (8)          holidays : array_like of datetime64[D], optional
1219: (8)          An array of dates to consider as invalid dates. They may be
1220: (8)          specified in any order, and NaT (not-a-time) dates are ignored.
1221: (8)          This list is saved in a normalized form that is suited for
1222: (4)          fast calculations of valid days.
1223: (8)          busdaycal : busdaycalendar, optional
1224: (8)          A `busdaycalendar` object which specifies the valid days. If this
1225: (8)          parameter is provided, neither weekmask nor holidays may be
1226: (4)          provided.
1227: (8)          out : array of datetime64[D], optional
1228: (4)          If provided, this array is filled with the result.
1229: (4)
1230: (4)          Returns
1231: (8)          -----
1232: (8)          out : array of datetime64[D]
1233: (4)          An array with a shape from broadcasting ``dates`` and ``offsets``
1234: (4)          together, containing the dates with offsets applied.

```

See Also

```

1234: (4)
1235: (4)
1236: (4)
1237: (4)
1238: (4)
1239: (4)
1240: (4)
1241: (4)
1242: (4)
1243: (4)
1244: (4)
1245: (4)
1246: (4)
1247: (4)
1248: (4)
1249: (4)
1250: (4)
1251: (4)
1252: (4)
1253: (4)
1254: (4)
1255: (4)
1256: (4)
1257: (4)
1258: (4)
1259: (4)
1260: (4)
1261: (4)
1262: (4)
1263: (4)
1264: (0)
1265: (0)
1266: (17)
1267: (4)
1268: (4)
busdaycal=None, out=None)
1269: (4)
1270: (4)
1271: (4)
1272: (4)
1273: (4)
1274: (4)
1275: (4)
1276: (4)
1277: (8)
1278: (4)
1279: (8)
1280: (8)
1281: (4)
1282: (8)
1283: (8)
1284: (8)
1285: (8)
1286: (8)
1287: (8)
1288: (4)
1289: (8)
1290: (8)
1291: (8)
1292: (8)
1293: (4)
1294: (8)
1295: (8)
1296: (8)
1297: (4)
1298: (8)
1299: (4)
1300: (4)
1301: (4)

-----  

busdaycalendar : An object that specifies a custom set of valid days.  

is_busday : Returns a boolean array indicating valid days.  

busday_count : Counts how many valid days are in a half-open date range.  

Examples  

-----  

>>> # First business day in October 2011 (not accounting for holidays)  

... np.busday_offset('2011-10', 0, roll='forward')  

numpy.datetime64('2011-10-03')  

>>> # Last business day in February 2012 (not accounting for holidays)  

... np.busday_offset('2012-03', -1, roll='forward')  

numpy.datetime64('2012-02-29')  

>>> # Third Wednesday in January 2011  

... np.busday_offset('2011-01', 2, roll='forward', weekmask='Wed')  

numpy.datetime64('2011-01-19')  

>>> # 2012 Mother's Day in Canada and the U.S.  

... np.busday_offset('2012-05', 1, roll='forward', weekmask='Sun')  

numpy.datetime64('2012-05-13')  

>>> # First business day on or after a date  

... np.busday_offset('2011-03-20', 0, roll='forward')  

numpy.datetime64('2011-03-21')  

>>> np.busday_offset('2011-03-22', 0, roll='forward')  

numpy.datetime64('2011-03-22')  

>>> # First business day after a date  

... np.busday_offset('2011-03-20', 1, roll='backward')  

numpy.datetime64('2011-03-21')  

>>> np.busday_offset('2011-03-22', 1, roll='backward')  

numpy.datetime64('2011-03-23')  

"""  

    return (dates, offsets, weekmask, holidays, out)
@array_function_from_c_func_and_dispatcher(_multiarray_umath.busday_count)
def busday_count(begindates, enddates, weekmask=None, holidays=None,
                 busdaycal=None, out=None):
    """  

    busday_count(begindates, enddates, weekmask='1111100', holidays=[],  

    busdaycal=None, out=None)  

Counts the number of valid days between `begindates` and  

`enddates`, not including the day of `enddates`.  

If ``enddates`` specifies a date value that is earlier than the  

corresponding ``begindates`` date value, the count will be negative.  

.. versionadded:: 1.7.0  

Parameters  

-----  

begindates : array_like of datetime64[D]  

    The array of the first dates for counting.  

enddates : array_like of datetime64[D]  

    The array of the end dates for counting, which are excluded  

    from the count themselves.  

weekmask : str or array_like of bool, optional  

    A seven-element array indicating which of Monday through Sunday are  

    valid days. May be specified as a length-seven list or array, like  

    [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string  

    like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for  

    weekdays, optionally separated by white space. Valid abbreviations  

    are: Mon Tue Wed Thu Fri Sat Sun  

holidays : array_like of datetime64[D], optional  

    An array of dates to consider as invalid dates. They may be  

    specified in any order, and NaT (not-a-time) dates are ignored.  

    This list is saved in a normalized form that is suited for  

    fast calculations of valid days.  

busdaycal : busdaycalendar, optional  

    A `busdaycalendar` object which specifies the valid days. If this  

    parameter is provided, neither weekmask nor holidays may be  

    provided.  

out : array of int, optional  

    If provided, this array is filled with the result.  

Returns  

-----  

out : array of int

```

```

1302: (8)
``enddates``
1303: (8)          An array with a shape from broadcasting ``begindates`` and
1304: (8)          together, containing the number of valid days between
1305: (4)          the begin and end dates.
See Also
-----
1307: (4)          busdaycalendar : An object that specifies a custom set of valid days.
1308: (4)          is_busday : Returns a boolean array indicating valid days.
1309: (4)          busday_offset : Applies an offset counted in valid days.
1310: (4)          Examples
-----
1312: (4)          >>> # Number of weekdays in January 2011
1313: (4)          ... np.busday_count('2011-01', '2011-02')
1314: (4)          21
1315: (4)          >>> # Number of weekdays in 2011
1316: (4)          >>> np.busday_count('2011', '2012')
1317: (4)          260
1318: (4)          >>> # Number of Saturdays in 2011
1319: (4)          ... np.busday_count('2011', '2012', weekmask='Sat')
1320: (4)          53
"""
1322: (4)          return (begindates, enddates, weekmask, holidays, out)
1323: (0)          @array_function_from_c_func_and_dispatcher(
1324: (4)              _multiarray_umath.datetime_as_string)
1325: (0)          def datetime_as_string(arr, unit=None, timezone=None, casting=None):
1326: (4)              """
1327: (4)                  datetime_as_string(arr, unit=None, timezone='naive', casting='same_kind')
1328: (4)                  Convert an array of datetimes into an array of strings.
1329: (4)                  Parameters
-----
1330: (4)
1331: (4)                  arr : array_like of datetime64
1332: (8)                  The array of UTC timestamps to format.
1333: (4)                  unit : str
1334: (8)                  One of None, 'auto', or a :ref:`datetime unit
<arrays.dtypes.dateunits>`.
1335: (4)                  timezone : {'naive', 'UTC', 'local'} or tzinfo
1336: (8)                  Timezone information to use when displaying the datetime. If 'UTC',
end
1337: (8)                  with a Z to indicate UTC time. If 'local', convert to the local
timezone
1338: (8)                  first, and suffix with a +-#### timezone offset. If a tzinfo object,
1339: (8)                  then do as with 'local', but use the specified timezone.
1340: (4)                  casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}
1341: (8)                  Casting to allow when changing between datetime units.
1342: (4)                  Returns
-----
1343: (4)
1344: (4)                  str_arr : ndarray
1345: (8)                  An array of strings the same shape as `arr` .
1346: (4)                  Examples
-----
1348: (4)                  >>> import pytz
1349: (4)                  >>> d = np.arange('2002-10-27T04:30', 4*60, 60, dtype='M8[m]')
1350: (4)                  >>> d
1351: (4)                  array(['2002-10-27T04:30', '2002-10-27T05:30', '2002-10-27T06:30',
1352: (11)                      '2002-10-27T07:30'], dtype='datetime64[m]')
1353: (4)                  Setting the timezone to UTC shows the same information, but with a Z
suffix
1354: (4)                  >>> np.datetime_as_string(d, timezone='UTC')
1355: (4)                  array(['2002-10-27T04:30Z', '2002-10-27T05:30Z', '2002-10-27T06:30Z',
1356: (11)                      '2002-10-27T07:30Z'], dtype='|<U35')
1357: (4)                  Note that we picked datetimes that cross a DST boundary. Passing in a
`pytz` timezone object will print the appropriate offset
1358: (4)                  >>> np.datetime_as_string(d, timezone=pytz.timezone('US/Eastern'))
1359: (4)                  array(['2002-10-27T00:30-0400', '2002-10-27T01:30-0400',
1360: (4)                      '2002-10-27T01:30-0500', '2002-10-27T02:30-0500'], dtype='|<U39')
1361: (11)                  Passing in a unit will change the precision
1362: (4)                  >>> np.datetime_as_string(d, unit='h')
1363: (4)                  array(['2002-10-27T04', '2002-10-27T05', '2002-10-27T06', '2002-10-
27T07'], )

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1365: (10)           dtype='<U32')
1366: (4)            >>> np.datetime_as_string(d, unit='s')
1367: (4)            array(['2002-10-27T04:30:00', '2002-10-27T05:30:00', '2002-10-
27T06:30:00',
1368: (11)           '2002-10-27T07:30:00'], dtype='<U38')
1369: (4)            'casting' can be used to specify whether precision can be changed
1370: (4)            >>> np.datetime_as_string(d, unit='h', casting='safe')
1371: (4)            Traceback (most recent call last):
1372: (8)             ...
1373: (4)             TypeError: Cannot create a datetime string as units 'h' from a NumPy
1374: (4)             datetime with units 'm' according to the rule 'safe'
1375: (4)             """
1376: (4)             return (arr,)

-----

```

## File 53 - numeric.py:

```

1: (0)          import functools
2: (0)          import itertools
3: (0)          import operator
4: (0)          import sys
5: (0)          import warnings
6: (0)          import numbers
7: (0)          import builtins
8: (0)          import numpy as np
9: (0)          from . import multiarray
10: (0)         from .multiarray import (
11: (4)           fastCopyAndTranspose, ALLOW_THREADS,
12: (4)           BUFSIZE, CLIP, MAXDIMS, MAY_SHARE_BOUNDS, MAY_SHARE_EXACT, RAISE,
13: (4)           WRAP, arange, array, asarray, asanyarray, ascontiguousarray,
14: (4)           asfortranarray, broadcast, can_cast, compare_chararrays,
15: (4)           concatenate, copyto, dot, dtype, empty,
16: (4)           empty_like, flatiter, frombuffer, from_dlpck, fromfile, fromiter,
17: (4)           fromstring, inner, lexsort, matmul, may_share_memory,
18: (4)           min_scalar_type, ndarray, nditer, nested_iters, promote_types,
19: (4)           putmask, result_type, set_numeric_ops, shares_memory, vdot, where,
20: (4)           zeros, normalize_axis_index, _get_promotion_state, _set_promotion_state,
21: (4)           _using_numpy2_behavior)
22: (0)         from . import overrides
23: (0)         from . import umath
24: (0)         from . import shape_base
25: (0)         from .overrides import set_array_function_like_doc, set_module
26: (0)         from .umath import (multiply, invert, sin, PINF, NAN)
27: (0)         from . import numeric_types
28: (0)         from .numeric_types import longlong, intc, int_, float_, complex_, bool_
29: (0)         from ..exceptions import ComplexWarning, TooHardError, AxisError
30: (0)         from .ufunc_config import errstate, _no_nep50_warning
31: (0)         bitwise_not = invert
32: (0)         ufunc = type(sin)
33: (0)         newaxis = None
34: (0)         array_function_dispatch = functools.partial(
35: (4)           overrides.array_function_dispatch, module='numpy')
36: (0)         __all__ = [
37: (4)           'newaxis', 'ndarray', 'flatiter', 'nditer', 'nested_iters', 'ufunc',
38: (4)           'arange', 'array', 'asarray', 'asanyarray', 'ascontiguousarray',
39: (4)           'asfortranarray', 'zeros', 'count_nonzero', 'empty', 'broadcast', 'dtype',
40: (4)           'fromstring', 'fromfile', 'frombuffer', 'from_dlpck', 'where',
41: (4)           'argwhere', 'copyto', 'concatenate', 'fastCopyAndTranspose', 'lexsort',
42: (4)           'set_numeric_ops', 'can_cast', 'promote_types', 'min_scalar_type',
43: (4)           'result_type', 'isfortran', 'empty_like', 'zeros_like', 'ones_like',
44: (4)           'correlate', 'convolve', 'inner', 'dot', 'outer', 'vdot', 'roll',
45: (4)           'rollaxis', 'moveaxis', 'cross', 'tensordot', 'little_endian',
46: (4)           'fromiter', 'array_equal', 'array_equiv', 'indices', 'fromfunction',
47: (4)           'isclose', 'isscalar', 'binary_repr', 'base_repr', 'ones',
48: (4)           'identity', 'allclose', 'compare_chararrays', 'putmask',
49: (4)           'flatnonzero', 'Inf', 'inf', 'infty', 'Infinity', 'nan', 'NaN',
50: (4)           'False_', 'True_', 'bitwise_not', 'CLIP', 'RAISE', 'WRAP', 'MAXDIMS',
51: (4)           'BUFSIZE', 'ALLOW_THREADS', 'full', 'full_like',

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

52: (4)      'matmul', 'shares_memory', 'may_share_memory', 'MAY_SHARE_BOUNDS',
53: (4)      'MAY_SHARE_EXACT', '_get_promotion_state', '_set_promotion_state',
54: (4)      '_using_numpy2_behavior']
55: (0)      def _zeros_like_dispatcher(a, dtype=None, order=None, subok=None, shape=None):
56: (4)          return (a,)
57: (0)      @array_function_dispatch(_zeros_like_dispatcher)
58: (0)      def zeros_like(a, dtype=None, order='K', subok=True, shape=None):
59: (4)          """
60: (4)              Return an array of zeros with the same shape and type as a given array.
61: (4)          Parameters
62: (4)          -----
63: (4)          a : array_like
64: (8)              The shape and data-type of `a` define these same attributes of
65: (8)              the returned array.
66: (4)          dtype : data-type, optional
67: (8)              Overrides the data type of the result.
68: (8)              .. versionadded:: 1.6.0
69: (4)          order : {'C', 'F', 'A', or 'K'}, optional
70: (8)              Overrides the memory layout of the result. 'C' means C-order,
71: (8)              'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
72: (8)              'C' otherwise. 'K' means match the layout of `a` as closely
73: (8)              as possible.
74: (8)              .. versionadded:: 1.6.0
75: (4)          subok : bool, optional.
76: (8)              If True, then the newly created array will use the sub-class
77: (8)              type of `a`, otherwise it will be a base-class array. Defaults
78: (8)              to True.
79: (4)          shape : int or sequence of ints, optional.
80: (8)              Overrides the shape of the result. If order='K' and the number of
81: (8)              dimensions is unchanged, will try to keep order, otherwise,
82: (8)              order='C' is implied.
83: (8)              .. versionadded:: 1.17.0
84: (4)          Returns
85: (4)          -----
86: (4)          out : ndarray
87: (8)              Array of zeros with the same shape and type as `a`.
88: (4)          See Also
89: (4)          -----
90: (4)          empty_like : Return an empty array with shape and type of input.
91: (4)          ones_like : Return an array of ones with shape and type of input.
92: (4)          full_like : Return a new array with shape of input filled with value.
93: (4)          zeros : Return a new array setting values to zero.
94: (4)          Examples
95: (4)          -----
96: (4)          >>> x = np.arange(6)
97: (4)          >>> x = x.reshape((2, 3))
98: (4)          >>> x
99: (4)          array([[0, 1, 2],
100: (11)             [3, 4, 5]])
101: (4)          >>> np.zeros_like(x)
102: (4)          array([[0, 0, 0],
103: (11)             [0, 0, 0]])
104: (4)          >>> y = np.arange(3, dtype=float)
105: (4)          >>> y
106: (4)          array([0., 1., 2.])
107: (4)          >>> np.zeros_like(y)
108: (4)          array([0., 0., 0.])
109: (4)          """
110: (4)          res = empty_like(a, dtype=dtype, order=order, subok=subok, shape=shape)
111: (4)          z = zeros(1, dtype=res.dtype)
112: (4)          multiarray.copyto(res, z, casting='unsafe')
113: (4)          return res
114: (0)          @set_array_function_like_doc
115: (0)          @set_module('numpy')
116: (0)          def ones(shape, dtype=None, order='C', *, like=None):
117: (4)              """
118: (4)                  Return a new array of given shape and type, filled with ones.
119: (4)                  Parameters
119: (4)                  -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

121: (4) shape : int or sequence of ints
122: (8) Shape of the new array, e.g., ``(2, 3)`` or ``2``.
123: (4) dtype : data-type, optional
124: (8) The desired data-type for the array, e.g., `numpy.int8`. Default is
125: (8) `numpy.float64`.
126: (4) order : {'C', 'F'}, optional, default: C
127: (8) Whether to store multi-dimensional data in row-major
128: (8) (C-style) or column-major (Fortran-style) order in
129: (8) memory.
130: (4) ${ARRAY_FUNCTION_LIKE}
131: (8) .. versionadded:: 1.20.0
132: (4) Returns
133: (4) -----
134: (4) out : ndarray
135: (8) Array of ones with the given shape, dtype, and order.
136: (4) See Also
137: (4) -----
138: (4) ones_like : Return an array of ones with shape and type of input.
139: (4) empty : Return a new uninitialized array.
140: (4) zeros : Return a new array setting values to zero.
141: (4) full : Return a new array of given shape filled with value.
142: (4) Examples
143: (4) -----
144: (4) >>> np.ones(5)
145: (4) array([1., 1., 1., 1., 1.])
146: (4) >>> np.ones((5,), dtype=int)
147: (4) array([1, 1, 1, 1, 1])
148: (4) >>> np.ones((2, 1))
149: (4) array([[1.],
150: (11)     [1.]])
151: (4) >>> s = (2,2)
152: (4) >>> np.ones(s)
153: (4) array([[1., 1.],
154: (11)     [1., 1.]])
155: (4) """
156: (4) if like is not None:
157: (8)     return _ones_with_like(like, shape, dtype=dtype, order=order)
158: (4) a = empty(shape, dtype, order)
159: (4) multiarray.copyto(a, 1, casting='unsafe')
160: (4) return a
161: (0) _ones_with_like = array_function_dispatch()(ones)
162: (0) def _ones_like_dispatcher(a, dtype=None, order=None, subok=None, shape=None):
163: (4)     return (a,)
164: (0) @array_function_dispatch(_ones_like_dispatcher)
165: (0) def ones_like(a, dtype=None, order='K', subok=True, shape=None):
166: (4) """
167: (4)     Return an array of ones with the same shape and type as a given array.
168: (4) Parameters
169: (4) -----
170: (4) a : array_like
171: (8)     The shape and data-type of `a` define these same attributes of
172: (8)     the returned array.
173: (4) dtype : data-type, optional
174: (8)     Overrides the data type of the result.
175: (8)     .. versionadded:: 1.6.0
176: (4) order : {'C', 'F', 'A', or 'K'}, optional
177: (8)     Overrides the memory layout of the result. 'C' means C-order,
178: (8)     'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
179: (8)     'C' otherwise. 'K' means match the layout of `a` as closely
180: (8)     as possible.
181: (8)     .. versionadded:: 1.6.0
182: (4) subok : bool, optional.
183: (8)     If True, then the newly created array will use the sub-class
184: (8)     type of `a`, otherwise it will be a base-class array. Defaults
185: (8)     to True.
186: (4) shape : int or sequence of ints, optional.
187: (8)     Overrides the shape of the result. If order='K' and the number of
188: (8)     dimensions is unchanged, will try to keep order, otherwise,
189: (8)     order='C' is implied.

```

```

190: (8)          .. versionadded:: 1.17.0
191: (4)          Returns
192: (4)
193: (4)          out : ndarray
194: (8)          Array of ones with the same shape and type as `a`.
195: (4)          See Also
196: (4)
197: (4)          -----
198: (4)          empty_like : Return an empty array with shape and type of input.
199: (4)          zeros_like : Return an array of zeros with shape and type of input.
200: (4)          full_like : Return a new array with shape of input filled with value.
201: (4)          ones : Return a new array setting values to one.
202: (4)          Examples
203: (4)          -----
204: (4)          >>> x = np.arange(6)
205: (4)          >>> x = x.reshape((2, 3))
206: (4)          >>> x
207: (11)         array([[0, 1, 2],
208: (4)                  [3, 4, 5]])
209: (4)          >>> np.ones_like(x)
210: (11)         array([[1, 1, 1],
211: (4)                  [1, 1, 1]])
212: (4)          >>> y = np.arange(3, dtype=float)
213: (4)          >>> y
214: (4)          array([0., 1., 2.])
215: (4)          >>> np.ones_like(y)
216: (4)          array([1., 1., 1.])
217: (4)          """
218: (4)          res = empty_like(a, dtype=dtype, order=order, subok=subok, shape=shape)
219: (4)          multiarray.copyto(res, 1, casting='unsafe')
220: (0)          return res
221: (4)
222: (0)          def _full_dispatcher(shape, fill_value, dtype=None, order=None, *, like=None):
223: (0)              return(like,)
224: (0)          @set_array_function_like_doc
225: (0)          @set_module('numpy')
226: (0)          def full(shape, fill_value, dtype=None, order='C', *, like=None):
227: (4)              """
228: (4)              Return a new array of given shape and type, filled with `fill_value`.
229: (4)              Parameters
230: (4)              -----
231: (4)              shape : int or sequence of ints
232: (8)                  Shape of the new array, e.g., ``(2, 3)`` or ``2``.
233: (4)              fill_value : scalar or array_like
234: (8)                  Fill value.
235: (9)              dtype : data-type, optional
236: (4)                  The desired data-type for the array. The default, None, means
237: (4)                  ``np.array(fill_value).dtype``.
238: (4)              order : {'C', 'F'}, optional
239: (4)                  Whether to store multidimensional data in C- or Fortran-contiguous
240: (8)                  (row- or column-wise) order in memory.
241: (4)                  ${ARRAY_FUNCTION_LIKE}
242: (4)                      .. versionadded:: 1.20.0
243: (4)          Returns
244: (4)          -----
245: (4)          out : ndarray
246: (4)          Array of `fill_value` with the given shape, dtype, and order.
247: (4)          See Also
248: (4)
249: (4)          full_like : Return a new array with shape of input filled with value.
250: (4)          empty : Return a new uninitialized array.
251: (4)          ones : Return a new array setting values to one.
252: (4)          zeros : Return a new array setting values to zero.
253: (4)          Examples
254: (4)          -----
255: (4)          >>> np.full((2, 2), np.inf)
256: (4)          array([[inf, inf],
257: (11)                  [inf, inf]])
258: (4)          >>> np.full((2, 2), 10)
259: (4)          array([[10, 10],
260: (11)                  [10, 10]])

```

```

259: (4)          >>> np.full((2, 2), [1, 2])
260: (4)          array([[1, 2],
261: (11)           [1, 2]])
262: (4)
263: (4)          if like is not None:
264: (8)          return _full_with_like(
265: (16)              like, shape, fill_value, dtype=dtype, order=order)
266: (4)          if dtype is None:
267: (8)              fill_value = asarray(fill_value)
268: (8)              dtype = fill_value.dtype
269: (4)          a = empty(shape, dtype, order)
270: (4)          multiarray.copyto(a, fill_value, casting='unsafe')
271: (4)          return a
272: (0)          _full_with_like = array_function_dispatch()(full)
273: (0)          def _full_like_dispatcher(a, fill_value, dtype=None, order=None,
274: (4)              subok=None):
275: (0)              return (a,)
276: (0)          @array_function_dispatch(_full_like_dispatcher)
277: (4)          def full_like(a, fill_value, dtype=None, order='K', subok=True, shape=None):
278: (4)              """
279: (4)                  Return a full array with the same shape and type as a given array.
280: (4)          Parameters
281: (4)          -----
282: (4)          a : array_like
283: (8)              The shape and data-type of `a` define these same attributes of
284: (8)              the returned array.
285: (4)          fill_value : array_like
286: (8)              Fill value.
287: (4)          dtype : data-type, optional
288: (8)              Overrides the data type of the result.
289: (4)          order : {'C', 'F', 'A', or 'K'}, optional
290: (8)              Overrides the memory layout of the result. 'C' means C-order,
291: (8)              'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
292: (8)              'C' otherwise. 'K' means match the layout of `a` as closely
293: (4)              as possible.
294: (8)          subok : bool, optional.
295: (8)              If True, then the newly created array will use the sub-class
296: (8)              type of `a`, otherwise it will be a base-class array. Defaults
297: (4)              to True.
298: (8)          shape : int or sequence of ints, optional.
299: (8)              Overrides the shape of the result. If order='K' and the number of
300: (8)              dimensions is unchanged, will try to keep order, otherwise,
301: (8)              order='C' is implied.
302: (4)              .. versionadded:: 1.17.0
303: (4)          Returns
304: (4)          -----
305: (4)          out : ndarray
306: (8)              Array of `fill_value` with the same shape and type as `a`.
307: (4)          See Also
308: (4)          -----
309: (4)          empty_like : Return an empty array with shape and type of input.
310: (4)          ones_like : Return an array of ones with shape and type of input.
311: (4)          zeros_like : Return an array of zeros with shape and type of input.
312: (4)          full : Return a new array of given shape filled with value.
313: (4)          Examples
314: (4)          -----
315: (4)          >>> x = np.arange(6, dtype=int)
316: (4)          >>> np.full_like(x, 1)
317: (4)          array([1, 1, 1, 1, 1, 1])
318: (4)          >>> np.full_like(x, 0.1)
319: (4)          array([0, 0, 0, 0, 0, 0])
320: (4)          >>> np.full_like(x, 0.1, dtype=np.double)
321: (4)          array([0.1, 0.1, 0.1, 0.1, 0.1, 0.1])
322: (4)          >>> np.full_like(x, np.nan, dtype=np.double)
323: (4)          array([nan, nan, nan, nan, nan, nan])
324: (4)          >>> y = np.arange(6, dtype=np.double)
325: (4)          >>> np.full_like(y, 0.1)
326: (4)          array([0.1, 0.1, 0.1, 0.1, 0.1, 0.1])

```

```

327: (4)
328: (4)
329: (12)
330: (11)
331: (12)
332: (4)
333: (4)
334: (4)
335: (4)
336: (0)
337: (4)
338: (0)
339: (0)
340: (4)
341: (4)
342: (4)
343: (4)
344: (4)
345: (4)
346: (4)
347: (4)
348: (4)
349: (4)
350: (4)
351: (4)
352: (4)
353: (4)
354: (8)
355: (4)
356: (8)
357: (8)
358: (8)
359: (8)
360: (4)
361: (8)
362: (8)
363: (8)
364: (8)
365: (4)
366: (4)
367: (4)
368: (8)
369: (8)
370: (8)
371: (4)
372: (4)
373: (4)
374: (4)
375: (4)
376: (4)
377: (4)
378: (4)
379: (4)
380: (4)
381: (4)
382: (4)
383: (4)
384: (4)
385: (4)
386: (4)
387: (4)
388: (11)
389: (4)
390: (4)
391: (8)
392: (4)
393: (4)
394: (8)
395: (4)

    >>> np.full_like(y, [0, 0, 255])
    array([[[ 0,  0, 255],
            [ 0,  0, 255]],
           [[ 0,  0, 255],
            [ 0,  0, 255]]])
    """
    res = empty_like(a, dtype=dtype, order=order, subok=subok, shape=shape)
    multiarray.copyto(res, fill_value, casting='unsafe')
    return res

def _count_nonzero_dispatcher(a, axis=None, *, keepdims=None):
    return (a,)

@array_function_dispatch(_count_nonzero_dispatcher)
def count_nonzero(a, axis=None, *, keepdims=False):
    """
    Counts the number of non-zero values in the array ``a``.
    The word "non-zero" is in reference to the Python 2.x
    built-in method ``__nonzero__()`` (renamed ``__bool__()``)
    in Python 3.x) of Python objects that tests an object's
    "truthfulness". For example, any number is considered
    truthful if it is nonzero, whereas any string is considered
    truthful if it is not the empty string. Thus, this function
    (recursively) counts how many elements in ``a`` (and in
    sub-arrays thereof) have their ``__nonzero__()`` or ``__bool__()``
    method evaluated to ``True``.

    Parameters
    -----
    a : array_like
        The array for which to count non-zeros.
    axis : int or tuple, optional
        Axis or tuple of axes along which to count non-zeros.
        Default is None, meaning that non-zeros will be counted
        along a flattened version of ``a``.
        .. versionadded:: 1.12.0
    keepdims : bool, optional
        If this is set to True, the axes that are counted are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.
        .. versionadded:: 1.19.0

    Returns
    -----
    count : int or array of int
        Number of non-zero values in the array along a given axis.
        Otherwise, the total number of non-zero values in the array
        is returned.

    See Also
    -----
    nonzero : Return the coordinates of all the non-zero values.

    Examples
    -----
    >>> np.count_nonzero(np.eye(4))
    4
    >>> a = np.array([[0, 1, 7, 0],
    ...                  [3, 0, 2, 19]])
    >>> np.count_nonzero(a)
    5
    >>> np.count_nonzero(a, axis=0)
    array([1, 1, 2, 1])
    >>> np.count_nonzero(a, axis=1)
    array([2, 3])
    >>> np.count_nonzero(a, axis=1, keepdims=True)
    array([[2],
           [3]])
    """
    if axis is None and not keepdims:
        return multiarray.count_nonzero(a)
    a = asanyarray(a)
    if np.issubdtype(a.dtype, np.character):
        a_bool = a != a.dtype.type()
    else:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

396: (8)             a_bool = a.astype(np.bool_, copy=False)
397: (4)             return a_bool.sum(axis=axis, dtype=np.intp, keepdims=keepdims)
398: (0) @set_module('numpy')
399: (0) def isfortran(a):
400: (4)     """
401: (4)     Check if the array is Fortran contiguous but *not* C contiguous.
402: (4)     This function is obsolete and, because of changes due to relaxed stride
403: (4)     checking, its return value for the same array may differ for versions
404: (4)     of NumPy >= 1.10.0 and previous versions. If you only want to check if an
405: (4)     array is Fortran contiguous use ``a.flags.f_contiguous`` instead.
406: (4)     Parameters
407: (4)     -----
408: (4)     a : ndarray
409: (8)         Input array.
410: (4)     Returns
411: (4)     -----
412: (4)     isfortran : bool
413: (8)         Returns True if the array is Fortran contiguous but *not* C
contiguous.
414: (4)     Examples
415: (4)     -----
416: (4)     np.array allows to specify whether the array is written in C-contiguous
417: (4)     order (last index varies the fastest), or FORTRAN-contiguous order in
418: (4)     memory (first index varies the fastest).
419: (4)     >>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
420: (4)     >>> a
421: (4)     array([[1, 2, 3],
422: (11)           [4, 5, 6]])
423: (4)     >>> np.isfortran(a)
424: (4)     False
425: (4)     >>> b = np.array([[1, 2, 3], [4, 5, 6]], order='F')
426: (4)     >>> b
427: (4)     array([[1, 2, 3],
428: (11)           [4, 5, 6]])
429: (4)     >>> np.isfortran(b)
430: (4)     True
431: (4)     The transpose of a C-ordered array is a FORTRAN-ordered array.
432: (4)     >>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
433: (4)     >>> a
434: (4)     array([[1, 2, 3],
435: (11)           [4, 5, 6]])
436: (4)     >>> np.isfortran(a)
437: (4)     False
438: (4)     >>> b = a.T
439: (4)     >>> b
440: (4)     array([[1, 4],
441: (11)           [2, 5],
442: (11)           [3, 6]])
443: (4)     >>> np.isfortran(b)
444: (4)     True
445: (4)     C-ordered arrays evaluate as False even if they are also FORTRAN-ordered.
446: (4)     >>> np.isfortran(np.array([1, 2], order='F'))
447: (4)     False
448: (4)     """
449: (4)     return a.flags.fnc
450: (0) def _argwhere_dispatcher(a):
451: (4)     return (a,)
452: (0) @array_function_dispatch(_argwhere_dispatcher)
453: (0) def argwhere(a):
454: (4)     """
455: (4)     Find the indices of array elements that are non-zero, grouped by element.
456: (4)     Parameters
457: (4)     -----
458: (4)     a : array_like
459: (8)         Input data.
460: (4)     Returns
461: (4)     -----
462: (4)     index_array : (N, a.ndim) ndarray
463: (8)         Indices of elements that are non-zero. Indices are grouped by element.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

464: (8)          This array will have shape ``(N, a.ndim)`` where ``N`` is the number
of
465: (8)          non-zero items.
466: (4)          See Also
467: (4)          -----
468: (4)          where, nonzero
469: (4)          Notes
470: (4)          -----
471: (4)          ``np.argwhere(a)`` is almost the same as ``np.transpose(np.nonzero(a))``, but produces a result of the correct shape for a 0D array.
472: (4)          The output of ``argwhere`` is not suitable for indexing arrays.
473: (4)          For this purpose use ``nonzero(a)`` instead.
474: (4)          Examples
475: (4)          -----
476: (4)          >>> x = np.arange(6).reshape(2,3)
477: (4)          >>> x
478: (4)          array([[0, 1, 2],
479: (4)                      [3, 4, 5]])
480: (11)
481: (4)          >>> np.argwhere(x>1)
482: (4)          array([[0, 2],
483: (11)                      [1, 0],
484: (11)                      [1, 1],
485: (11)                      [1, 2]])
486: (4)          """
487: (4)          if np.ndim(a) == 0:
488: (8)              a = shape_base.atleast_1d(a)
489: (8)              return argwhere(a)[:, :0]
490: (4)              return transpose(nonzero(a))
491: (0)          def _flatnonzero_dispatcher(a):
492: (4)              return (a,)
493: (0)          @array_function_dispatch(_flatnonzero_dispatcher)
494: (0)          def flatnonzero(a):
495: (4)              """
496: (4)              Return indices that are non-zero in the flattened version of a.
497: (4)              This is equivalent to ``np.nonzero(np.ravel(a))[0]``.
498: (4)              Parameters
499: (4)              -----
500: (4)              a : array_like
501: (8)                  Input data.
502: (4)              Returns
503: (4)              -----
504: (4)              res : ndarray
505: (8)                  Output array, containing the indices of the elements of ``a.ravel()`` that are non-zero.
506: (8)
507: (4)              See Also
508: (4)              -----
509: (4)              nonzero : Return the indices of the non-zero elements of the input array.
510: (4)              ravel : Return a 1-D array containing the elements of the input array.
511: (4)              Examples
512: (4)              -----
513: (4)              >>> x = np.arange(-2, 3)
514: (4)              >>> x
515: (4)              array([-2, -1, 0, 1, 2])
516: (4)              >>> np.flatnonzero(x)
517: (4)              array([0, 1, 3, 4])
518: (4)              Use the indices of the non-zero elements as an index array to extract these elements:
519: (4)              >>> x.ravel()[np.flatnonzero(x)]
520: (4)              array([-2, -1, 1, 2])
521: (4)              """
522: (4)
523: (4)              return np.nonzero(np.ravel(a))[0]
524: (0)          def _correlate_dispatcher(a, v, mode=None):
525: (4)              return (a, v)
526: (0)          @array_function_dispatch(_correlate_dispatcher)
527: (0)          def correlate(a, v, mode='valid'):
528: (4)              """
529: (4)              Cross-correlation of two 1-dimensional sequences.
530: (4)              This function computes the correlation as generally defined in signal processing texts:
531: (4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

532: (4) .. math:: c_k = \sum_n a_{n+k} \cdot \overline{v}_n
533: (4) with a and v sequences being zero-padded where necessary and
534: (4) :math:`\overline{x}` denoting complex conjugation.
535: (4) Parameters
536: (4) -----
537: (4) a, v : array_like
538: (8) Input sequences.
539: (4) mode : {'valid', 'same', 'full'}, optional
540: (8) Refer to the `convolve` docstring. Note that the default
541: (8) is 'valid', unlike `convolve`, which uses 'full'.
542: (4) old_behavior : bool
543: (8) `old_behavior` was removed in NumPy 1.10. If you need the old
544: (8) behavior, use `multiarray.correlate`.
545: (4) Returns
546: (4) -----
547: (4) out : ndarray
548: (8) Discrete cross-correlation of `a` and `v`.
549: (4) See Also
550: (4) -----
551: (4) convolve : Discrete, linear convolution of two one-dimensional sequences.
552: (4) multiarray.correlate : Old, no conjugate, version of correlate.
553: (4) scipy.signal.correlate : uses FFT which has superior performance on large
arrays.
554: (4) Notes
555: (4) -----
556: (4) The definition of correlation above is not unique and sometimes
correlation
557: (4) may be defined differently. Another common definition is:
558: (4) .. math:: c'_k = \sum_n a_{\{n\}} \cdot \overline{v}_{\{n+k\}}
559: (4) which is related to :math:`c_k` by :math:`c'_k = c_{-k}`.
560: (4) `numpy.correlate` may perform slowly in large arrays (i.e. n = 1e5)
because it does
561: (4) not use the FFT to compute the convolution; in that case,
`scipy.signal.correlate` might
562: (4) be preferable.
563: (4) Examples
564: (4) -----
565: (4) >>> np.correlate([1, 2, 3], [0, 1, 0.5])
566: (4) array([3.5])
567: (4) >>> np.correlate([1, 2, 3], [0, 1, 0.5], "same")
568: (4) array([2., 3.5, 3.])
569: (4) >>> np.correlate([1, 2, 3], [0, 1, 0.5], "full")
570: (4) array([0.5, 2., 3.5, 3., 0.])
571: (4) Using complex sequences:
572: (4) >>> np.correlate([1+1j, 2, 3-1j], [0, 1, 0.5j], 'full')
573: (4) array([ 0.5-0.5j, 1.0+0.j, 1.5-1.5j, 3.0-1.j, 0.0+0.j])
574: (4) Note that you get the time reversed, complex conjugated result
575: (4) (:math:`\overline{c}_{-k}`) when the two input sequences a and v change
places:
576: (4) >>> np.correlate([0, 1, 0.5j], [1+1j, 2, 3-1j], 'full')
577: (4) array([ 0.0+0.j, 3.0+1.j, 1.5+1.5j, 1.0+0.j, 0.5+0.5j])
578: (4) """
579: (4) return multiarray.correlate2(a, v, mode)
580: (4) def _convolve_dispatcher(a, v, mode=None):
581: (0)     return (a, v)
582: (4) @array_function_dispatch(_convolve_dispatcher)
583: (0) def convolve(a, v, mode='full'):
584: (0)     """
585: (4) Returns the discrete, linear convolution of two one-dimensional sequences.
586: (4) The convolution operator is often seen in signal processing, where it
587: (4) models the effect of a linear time-invariant system on a signal [1]_. In
588: (4) probability theory, the sum of two independent random variables is
589: (4) distributed according to the convolution of their individual
590: (4) distributions.
591: (4) If `v` is longer than `a`, the arrays are swapped before computation.
592: (4) Parameters
593: (4) -----
594: (4) a : (N,) array_like
595: (4) First one-dimensional input array.
596: (8)
```

```

597: (4)           v : (M,) array_like
598: (8)             Second one-dimensional input array.
599: (4)           mode : {'full', 'valid', 'same'}, optional
600: (8)             'full':
601: (10)            By default, mode is 'full'. This returns the convolution
602: (10)            at each point of overlap, with an output shape of (N+M-1,). At
603: (10)            the end-points of the convolution, the signals do not overlap
604: (10)            completely, and boundary effects may be seen.
605: (8)           'same':
606: (10)            Mode 'same' returns output of length ``max(M, N)` `. Boundary
607: (10)            effects are still visible.
608: (8)           'valid':
609: (10)            Mode 'valid' returns output of length
610: (10)            ``max(M, N) - min(M, N) + 1`` . The convolution product is only
given
611: (10)            for points where the signals overlap completely. Values outside
612: (10)            the signal boundary have no effect.
613: (4)           Returns
614: (4)           -----
615: (4)           out : ndarray
616: (8)             Discrete, linear convolution of `a` and `v` .
617: (4)           See Also
618: (4)           -----
619: (4)           scipy.signal.fftconvolve : Convolve two arrays using the Fast Fourier
620: (31)             Transform.
621: (4)           scipy.linalg.toeplitz : Used to construct the convolution operator.
622: (4)           polymul : Polynomial multiplication. Same output as convolve, but also
623: (14)             accepts poly1d objects as input.
624: (4)           Notes
625: (4)           -----
626: (4)             The discrete convolution operation is defined as
627: (4)             .. math:: (a * v)_n = \sum_{m = -\infty}^{\infty} a_m v_{n - m}
628: (4)             It can be shown that a convolution :math:`x(t) * y(t)` in time/space
629: (4)             is equivalent to the multiplication :math:`X(f) Y(f)` in the Fourier
630: (4)             domain, after appropriate padding (padding is necessary to prevent
631: (4)             circular convolution). Since multiplication is more efficient (faster)
632: (4)             than convolution, the function `scipy.signal.fftconvolve` exploits the
633: (4)             FFT to calculate the convolution of large data-sets.
634: (4)           References
635: (4)           -----
636: (4)             .. [1] Wikipedia, "Convolution",
637: (8)               https://en.wikipedia.org/wiki/Convolution
638: (4)           Examples
639: (4)           -----
640: (4)             Note how the convolution operator flips the second array
641: (4)             before "sliding" the two across one another:
642: (4)             >>> np.convolve([1, 2, 3], [0, 1, 0.5])
643: (4)             array([0. , 1. , 2.5, 4. , 1.5])
644: (4)             Only return the middle values of the convolution.
645: (4)             Contains boundary effects, where zeros are taken
646: (4)             into account:
647: (4)             >>> np.convolve([1,2,3],[0,1,0.5], 'same')
648: (4)             array([1. , 2.5, 4. ])
649: (4)             The two arrays are of the same length, so there
650: (4)             is only one position where they completely overlap:
651: (4)             >>> np.convolve([1,2,3],[0,1,0.5], 'valid')
652: (4)             array([2.5])
653: (4)             """
654: (4)             a, v = array(a, copy=False, ndmin=1), array(v, copy=False, ndmin=1)
655: (4)             if (len(v) > len(a)):
656: (8)                 a, v = v, a
657: (4)             if len(a) == 0:
658: (8)                 raise ValueError('a cannot be empty')
659: (4)             if len(v) == 0:
660: (8)                 raise ValueError('v cannot be empty')
661: (4)             return multiarray.correlate(a, v[::-1], mode)
662: (0)           def _outer_dispatcher(a, b, out=None):
663: (4)               return (a, b, out)
664: (0)           @array_function_dispatch(_outer_dispatcher)

```

```

665: (0)
666: (4)
667: (4)
668: (4)
669: (4)
670: (6)
671: (7)
672: (7)
673: (7)
674: (4)
675: (4)
676: (4)
677: (8)
678: (8)
679: (4)
680: (8)
681: (8)
682: (4)
683: (8)
684: (8)
685: (4)
686: (4)
687: (4)
688: (8)
689: (4)
690: (4)
691: (4)
692: (4)
693: (4)
694: (18)
695: (18)
696: (4)
697: (16)
698: (4)
699: (4)
700: (4)
701: (11)
702: (11)
703: (4)
704: (4)
705: (4)
706: (4)
707: (4)
708: (4)
709: (11)
710: (11)
711: (11)
712: (11)
713: (4)
714: (4)
715: (4)
716: (11)
717: (11)
718: (11)
719: (11)
720: (4)
721: (4)
722: (4)
723: (11)
724: (11)
725: (11)
726: (11)
727: (4)
728: (4)
729: (4)
730: (4)
731: (11)
732: (11)
733: (4)

def outer(a, b, out=None):
    """
    Compute the outer product of two vectors.
    Given two vectors `a` and `b` of length ``M`` and ``N``, respectively,
    the outer product [1]_ is::
        [[a_0*b_0  a_0*b_1 ... a_0*b_{N-1} ]
         [a_1*b_0      .
         [ ...
         [a_{M-1}*b_0          a_{M-1}*b_{N-1} ]]
    Parameters
    -----
    a : (M,) array_like
        First input vector. Input is flattened if
        not already 1-dimensional.
    b : (N,) array_like
        Second input vector. Input is flattened if
        not already 1-dimensional.
    out : (M, N) ndarray, optional
        A location where the result is stored
        .. versionadded:: 1.9.0
    Returns
    -----
    out : (M, N) ndarray
        ``out[i, j] = a[i] * b[j]``
    See also
    -----
    inner
    einsum : ``einsum('i,j->ij', a.ravel(), b.ravel())`` is the equivalent.
    ufunc.outer : A generalization to dimensions other than 1D and other
                  operations. ``np.multiply.outer(a.ravel(), b.ravel())`` is the equivalent.
    tensordot : ``np.tensordot(a.ravel(), b.ravel(), axes=(((), ()))`` is the equivalent.
    References
    -----
    .. [1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd
           ed., Baltimore, MD, Johns Hopkins University Press, 1996,
           pg. 8.
    Examples
    -----
    Make a (*very* coarse) grid for computing a Mandelbrot set:
    >>> rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
    >>> rl
    array([[-2., -1.,  0.,  1.,  2.],
           [-2., -1.,  0.,  1.,  2.],
           [-2., -1.,  0.,  1.,  2.],
           [-2., -1.,  0.,  1.,  2.],
           [-2., -1.,  0.,  1.,  2.]])
    >>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
    >>> im
    array([[0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
           [0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
           [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
           [0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
           [0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])
    >>> grid = rl + im
    >>> grid
    array([[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
           [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
           [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
           [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
           [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]])
    An example using a "vector" of letters:
    >>> x = np.array(['a', 'b', 'c'], dtype=object)
    >>> np.outer(x, [1, 2, 3])
    array([['a', 'aa', 'aaa'],
           ['b', 'bb', 'bbb'],
           ['c', 'cc', 'ccc']], dtype=object)
    """

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

734: (4)             a = asarray(a)
735: (4)             b = asarray(b)
736: (4)             return multiply(a.ravel()[:, newaxis], b.ravel()[newaxis, :], out)
737: (0)             def _tensordot_dispatcher(a, b, axes=None):
738: (4)                 return (a, b)
739: (0)             @array_function_dispatch(_tensordot_dispatcher)
740: (0)             def tensordot(a, b, axes=2):
741: (4)                 """
742: (4)                 Compute tensor dot product along specified axes.
743: (4)                 Given two tensors, `a` and `b`, and an array_like object containing
744: (4)                 two array_like objects, ``(a_axes, b_axes)``, sum the products of
745: (4)                 `a`'s and `b`'s elements (components) over the axes specified by
746: (4)                 ``a_axes`` and ``b_axes``. The third argument can be a single non-negative
747: (4)                 integer_like scalar, ``N``; if it is such, then the last ``N`` dimensions
748: (4)                 of `a` and the first ``N`` dimensions of `b` are summed over.
749: (4)                 Parameters
750: (4)                 -----
751: (4)                 a, b : array_like
752: (8)                     Tensors to "dot".
753: (4)                 axes : int or (2,) array_like
754: (8)                     * integer_like
755: (10)                    If an int N, sum over the last N axes of `a` and the first N axes
756: (10)                    of `b` in order. The sizes of the corresponding axes must match.
757: (8)                     * (2,) array_like
758: (10)                    Or, a list of axes to be summed over, first sequence applying to
`a`,
759: (10)                    second to `b`. Both elements array_like must be of the same length.
760: (4)             Returns
761: (4)             -----
762: (4)             output : ndarray
763: (8)                 The tensor dot product of the input.
764: (4)             See Also
765: (4)             -----
766: (4)             dot, einsum
767: (4)             Notes
768: (4)             -----
769: (4)             Three common use cases are:
770: (8)                 * ``axes = 0`` : tensor product :math:`a \otimes b`
771: (8)                 * ``axes = 1`` : tensor dot product :math:`a \cdot b`
772: (8)                 * ``axes = 2`` : (default) tensor double contraction :math:`a:b`
773: (4)             When `axes` is integer_like, the sequence for evaluation will be: first
774: (4)             the -Nth axis in `a` and 0th axis in `b`, and the -1th axis in `a` and
775: (4)             Nth axis in `b` last.
776: (4)             When there is more than one axis to sum over - and they are not the last
777: (4)             (first) axes of `a` (`b`) - the argument `axes` should consist of
778: (4)             two sequences of the same length, with the first axis to sum over given
779: (4)             first in both sequences, the second axis second, and so forth.
780: (4)             The shape of the result consists of the non-contracted axes of the
781: (4)             first tensor, followed by the non-contracted axes of the second.
782: (4)             Examples
783: (4)             -----
784: (4)             A "traditional" example:
785: (4)             >>> a = np.arange(60.).reshape(3,4,5)
786: (4)             >>> b = np.arange(24.).reshape(4,3,2)
787: (4)             >>> c = np.tensordot(a,b, axes=[1,0],[0,1]))
788: (4)             >>> c.shape
789: (4)             (5, 2)
790: (4)             >>> c
791: (4)             array([[4400., 4730.],
792: (11)                 [4532., 4874.],
793: (11)                 [4664., 5018.],
794: (11)                 [4796., 5162.],
795: (11)                 [4928., 5306.]])
796: (4)             >>> # A slower but equivalent way of computing the same...
797: (4)             >>> d = np.zeros((5,2))
798: (4)             >>> for i in range(5):
799: (4)                 ...     for j in range(2):
800: (4)                     ...         for k in range(3):
801: (4)                         ...             for n in range(4):

```

```

802: (4)          ...
803: (4)          ...      d[i,j] += a[k,n,i] * b[n,k,j]
804: (4)          >>> c == d
805: (11)         array([[ True,  True],
806: (11)             [ True,  True],
807: (11)             [ True,  True],
808: (11)             [ True,  True]])
809: (4)          An extended example taking advantage of the overloading of + and \\*:
810: (4)          >>> a = np.array(range(1, 9))
811: (4)          >>> a.shape = (2, 2, 2)
812: (4)          >>> A = np.array(['a', 'b', 'c', 'd'), dtype=object)
813: (4)          >>> A.shape = (2, 2)
814: (4)          >>> a; A
815: (4)          array([[[1, 2],
816: (12)            [3, 4]],
817: (11)            [[5, 6],
818: (12)              [7, 8]]])
819: (4)          array([[['a', 'b'],
820: (11)            ['c', 'd']], dtype=object)
821: (4)          >>> np.tensordot(a, A) # third argument default is 2 for double-
contraction
822: (4)          array(['abbcccddd', 'aaaaabbbbbcccccddd'], dtype=object)
823: (4)          >>> np.tensordot(a, A, 1)
824: (4)          array([[[['acc', 'bdd'],
825: (12)            ['aaacccc', 'bbbdddd'],
826: (11)              ['aaaaaccccc', 'bbbbbddddd'],
827: (12)                ['aaaaaaaaaccccccc', 'bbbbbbbddd'], dtype=object)
828: (4)          >>> np.tensordot(a, A, 0) # tensor product (result too long to incl.)
829: (4)          array([[[[['a', 'b'],
830: (14)            ['c', 'd']],
831: (14)              ...
832: (4)          >>> np.tensordot(a, A, (0, 1))
833: (4)          array([[[['abbbb', 'cdddd'],
834: (12)            ['aabbbbbbb', 'ccddddd'],
835: (11)              ['aaabbbbbbb', 'cccdffff'],
836: (12)                ['aaaabbbbbbbb', 'ccccddddd'], dtype=object)
837: (4)          >>> np.tensordot(a, A, (2, 1))
838: (4)          array([[[['abb', 'cdd'],
839: (12)            ['aaabbbb', 'ccccddd'],
840: (11)              ['aaaaabbbbbbb', 'cccccdffff'],
841: (12)                ['aaaaaaaaabbbbbbb', 'ccccccccddddd'], dtype=object)
842: (4)          >>> np.tensordot(a, A, ((0, 1), (0, 1)))
843: (4)          array(['abbbcccccddd', 'aabbcccccddd'], dtype=object)
844: (4)          >>> np.tensordot(a, A, ((2, 1), (1, 0)))
845: (4)          array(['accbbddd', 'aaaaacccccccbbbbbbddd'], dtype=object)
846: (4)          """
847: (4)          try:
848: (8)          iter(axes)
849: (4)          except Exception:
850: (8)          axes_a = list(range(-axes, 0))
851: (8)          axes_b = list(range(0, axes))
852: (4)          else:
853: (8)          axes_a, axes_b = axes
854: (4)          try:
855: (8)          na = len(axes_a)
856: (8)          axes_a = list(axes_a)
857: (4)          except TypeError:
858: (8)          axes_a = [axes_a]
859: (8)          na = 1
860: (4)          try:
861: (8)          nb = len(axes_b)
862: (8)          axes_b = list(axes_b)
863: (4)          except TypeError:
864: (8)          axes_b = [axes_b]
865: (8)          nb = 1
866: (4)          a, b = asarray(a), asarray(b)
867: (4)          as_ = a.shape
868: (4)          nda = a.ndim
869: (4)          bs = b.shape

```

```

870: (4)             ndb = b.ndim
871: (4)             equal = True
872: (4)             if na != nb:
873: (8)                 equal = False
874: (4)             else:
875: (8)                 for k in range(na):
876: (12)                     if as_[axes_a[k]] != bs[axes_b[k]]:
877: (16)                         equal = False
878: (16)                         break
879: (12)                     if axes_a[k] < 0:
880: (16)                         axes_a[k] += nda
881: (12)                     if axes_b[k] < 0:
882: (16)                         axes_b[k] += ndb
883: (4)             if not equal:
884: (8)                 raise ValueError("shape-mismatch for sum")
885: (4)             notin = [k for k in range(ndb) if k not in axes_a]
886: (4)             newaxes_a = notin + axes_a
887: (4)             N2 = 1
888: (4)             for axis in axes_a:
889: (8)                 N2 *= as_[axis]
890: (4)             newshape_a = (int(multiply.reduce([as_[ax] for ax in notin])), N2)
891: (4)             olda = [as_[axis] for axis in notin]
892: (4)             notin = [k for k in range(ndb) if k not in axes_b]
893: (4)             newaxes_b = axes_b + notin
894: (4)             N2 = 1
895: (4)             for axis in axes_b:
896: (8)                 N2 *= bs[axis]
897: (4)             newshape_b = (N2, int(multiply.reduce([bs[ax] for ax in notin])))
898: (4)             oldb = [bs[axis] for axis in notin]
899: (4)             at = a.transpose(newaxes_a).reshape(newshape_a)
900: (4)             bt = b.transpose(newaxes_b).reshape(newshape_b)
901: (4)             res = dot(at, bt)
902: (4)             return res.reshape(olda + oldb)
903: (0)             def _roll_dispatcher(a, shift, axis=None):
904: (4)                 return (a,)
905: (0)             @array_function_dispatch(_roll_dispatcher)
906: (0)             def roll(a, shift, axis=None):
907: (4)                 """
908: (4)                 Roll array elements along a given axis.
909: (4)                 Elements that roll beyond the last position are re-introduced at
910: (4)                 the first.
911: (4)                 Parameters
912: (4)                 -----
913: (4)                 a : array_like
914: (8)                     Input array.
915: (4)                 shift : int or tuple of ints
916: (8)                     The number of places by which elements are shifted. If a tuple,
917: (8)                     then `axis` must be a tuple of the same size, and each of the
918: (8)                     given axes is shifted by the corresponding number. If an int
919: (8)                     while `axis` is a tuple of ints, then the same value is used for
920: (8)                     all given axes.
921: (4)                 axis : int or tuple of ints, optional
922: (8)                     Axis or axes along which elements are shifted. By default, the
923: (8)                     array is flattened before shifting, after which the original
924: (8)                     shape is restored.
925: (4)                 Returns
926: (4)                 -----
927: (4)                 res : ndarray
928: (8)                     Output array, with the same shape as `a`.
929: (4)                 See Also
930: (4)                 -----
931: (4)                 rollaxis : Roll the specified axis backwards, until it lies in a
932: (15)                     given position.
933: (4)                 Notes
934: (4)                 -----
935: (4)                 .. versionadded:: 1.12.0
936: (4)                 Supports rolling over multiple dimensions simultaneously.
937: (4)                 Examples
938: (4)                 -----

```

```

939: (4)          >>> x = np.arange(10)
940: (4)          >>> np.roll(x, 2)
941: (4)          array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
942: (4)          >>> np.roll(x, -2)
943: (4)          array([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])
944: (4)          >>> x2 = np.reshape(x, (2, 5))
945: (4)          >>> x2
946: (4)          array([[0, 1, 2, 3, 4],
947: (11)           [5, 6, 7, 8, 9]])
948: (4)          >>> np.roll(x2, 1)
949: (4)          array([[9, 0, 1, 2, 3],
950: (11)           [4, 5, 6, 7, 8]])
951: (4)          >>> np.roll(x2, -1)
952: (4)          array([[1, 2, 3, 4, 5],
953: (11)           [6, 7, 8, 9, 0]])
954: (4)          >>> np.roll(x2, 1, axis=0)
955: (4)          array([[5, 6, 7, 8, 9],
956: (11)           [0, 1, 2, 3, 4]])
957: (4)          >>> np.roll(x2, -1, axis=0)
958: (4)          array([[5, 6, 7, 8, 9],
959: (11)           [0, 1, 2, 3, 4]])
960: (4)          >>> np.roll(x2, 1, axis=1)
961: (4)          array([[4, 0, 1, 2, 3],
962: (11)           [9, 5, 6, 7, 8]])
963: (4)          >>> np.roll(x2, -1, axis=1)
964: (4)          array([[1, 2, 3, 4, 0],
965: (11)           [6, 7, 8, 9, 5]])
966: (4)          >>> np.roll(x2, (1, 1), axis=(1, 0))
967: (4)          array([[9, 5, 6, 7, 8],
968: (11)           [4, 0, 1, 2, 3]])
969: (4)          >>> np.roll(x2, (2, 1), axis=(1, 0))
970: (4)          array([[8, 9, 5, 6, 7],
971: (11)           [3, 4, 0, 1, 2]])
972: (4)          """
973: (4)          a = asanyarray(a)
974: (4)          if axis is None:
975: (8)              return roll(a.ravel(), shift, 0).reshape(a.shape)
976: (4)          else:
977: (8)              axis = normalize_axis_tuple(axis, a.ndim, allow_duplicate=True)
978: (8)              broadcasted = broadcast(shift, axis)
979: (8)              if broadcasted.ndim > 1:
980: (12)                  raise ValueError(
981: (16)                      "'shift' and 'axis' should be scalars or 1D sequences")
982: (8)              shifts = {ax: 0 for ax in range(a.ndim)}
983: (8)              for sh, ax in broadcasted:
984: (12)                  shifts[ax] += sh
985: (8)              rolls = [((slice(None), slice(None)),)] * a.ndim
986: (8)              for ax, offset in shifts.items():
987: (12)                  offset %= a.shape[ax] or 1 # If `a` is empty, nothing matters.
988: (12)                  if offset:
989: (16)                      rolls[ax] = ((slice(None, -offset), slice(offset, None)),
990: (29)                          (slice(-offset, None), slice(None, offset)))
991: (8)              result = empty_like(a)
992: (8)              for indices in itertools.product(*rolls):
993: (12)                  arr_index, res_index = zip(*indices)
994: (12)                  result[res_index] = a[arr_index]
995: (8)              return result
996: (0)          def _rollaxis_dispatcher(a, axis, start=None):
997: (4)              return (a,)
998: (0)          @array_function_dispatch(_rollaxis_dispatcher)
999: (0)          def rollaxis(a, axis, start=0):
999: (0)          """
1000: (4)          Roll the specified axis backwards, until it lies in a given position.
1001: (4)          This function continues to be supported for backward compatibility, but
1002: (4)          you
1003: (4)          should prefer `moveaxis`. The `moveaxis` function was added in NumPy
1004: (4)          1.11.
1005: (4)          Parameters
1006: (4)          -----

```

```

1007: (4)
1008: (8)
1009: (4)
1010: (8)
1011: (8)
1012: (4)
1013: (8)
1014: (8)
1015: (8)
1016: (8)
1017: (8)
1018: (8)
1019: (11)
1020: (11)
1021: (11)
1022: (11)
1023: (11)
1024: (11)
1025: (11)
1026: (11)
1027: (11)
1028: (11)
1029: (11)
1030: (11)
1031: (11)
1032: (11)
1033: (11)
1034: (11)
1035: (11)
1036: (11)
1037: (11)
1038: (11)
1039: (8)
1040: (4)
1041: (4)
1042: (4)
1043: (8)
1044: (8)
1045: (8)
1046: (4)
1047: (4)
1048: (4)
1049: (4)
1050: (8)
1051: (4)
1052: (4)
1053: (4)
1054: (4)
1055: (4)
1056: (4)
1057: (4)
1058: (4)
1059: (4)
1060: (4)
1061: (4)
1062: (4)
1063: (4)
1064: (8)
1065: (4)
1066: (4)
1067: (8)
1068: (4)
1069: (8)
1070: (4)
1071: (8)
1072: (4)
1073: (4)
1074: (4)
1075: (4)

    a : ndarray
        Input array.
    axis : int
        The axis to be rolled. The positions of the other axes do not
        change relative to one another.
    start : int, optional
        When ``start <= axis``, the axis is rolled back until it lies in
        this position. When ``start > axis``, the axis is rolled until it
        lies before this position. The default, 0, results in a "complete"
        roll. The following table describes how negative values of ``start``
        are interpreted:
        .. table::
            :align: left
            +-----+-----+
            | ``start`` | Normalized ``start`` |
            +=====+=====+
            | ``-(arr.ndim+1)`` | raise ``AxisError`` |
            +-----+-----+
            | ``-arr.ndim`` | 0 |
            +-----+-----+
            | |vdots| | |vdots| |
            +-----+-----+
            | ``-1`` | ``arr.ndim-1`` |
            +-----+-----+
            | ``0`` | ``0`` |
            +-----+-----+
            | |vdots| | |vdots| |
            +-----+-----+
            | ``arr.ndim`` | ``arr.ndim`` |
            +-----+-----+
            | ``arr.ndim + 1`` | raise ``AxisError`` |
            +-----+-----+
        .. |vdots| unicode:: U+22EE .. Vertical Ellipsis
    Returns
    -----
    res : ndarray
        For NumPy >= 1.10.0 a view of `a` is always returned. For earlier
        NumPy versions a view of `a` is returned only if the order of the
        axes is changed, otherwise the input array is returned.
    See Also
    -----
    moveaxis : Move array axes to new positions.
    roll : Roll the elements of an array by a number of positions along a
           given axis.
    Examples
    -----
    >>> a = np.ones((3,4,5,6))
    >>> np.rollaxis(a, 3, 1).shape
    (3, 6, 4, 5)
    >>> np.rollaxis(a, 2).shape
    (5, 3, 4, 6)
    >>> np.rollaxis(a, 1, 4).shape
    (3, 5, 6, 4)
    """
    n = a.ndim
    axis = normalize_axis_index(axis, n)
    if start < 0:
        start += n
    msg = "'%s' arg requires %d <= %s < %d, but %d was passed in"
    if not (0 <= start < n + 1):
        raise AxisError(msg % ('start', -n, 'start', n + 1, start))
    if axis < start:
        start -= 1
    if axis == start:
        return a[...]
    axes = list(range(0, n))
    axes.remove(axis)
    axes.insert(start, axis)
    return a.transpose(axes)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1076: (0)
1077: (4)
1078: (4)
1079: (4)
1080: (4)
1081: (4)
1082: (4)
1083: (4)
1084: (4)
1085: (4)
1086: (4)
1087: (4)
1088: (8)
1089: (4)
1090: (8)
1091: (8)
1092: (4)
1093: (8)
1094: (8)
1095: (4)
1096: (8)
1097: (4)
1098: (4)
1099: (4)
1100: (8)
1101: (4)
1102: (4)
1103: (4)
1104: (8)
1105: (4)
1106: (8)
1107: (4)
1108: (4)
1109: (4)
1110: (4)
1111: (4)
1112: (8)
1113: (12)
1114: (8)
1115: (12)
1116: (4)
1117: (4)
1118: (8)
1119: (12)
1120: (8)
1121: (12)
1122: (4)
1123: (0)
1124: (4)
1125: (0)
1126: (0)
1127: (4)
1128: (4)
1129: (4)
1130: (4)
1131: (4)
1132: (4)
1133: (4)
1134: (8)
1135: (4)
1136: (8)
1137: (4)
1138: (8)
be
1139: (8)
1140: (4)
1141: (4)
1142: (4)
1143: (8)

def normalize_axis_tuple(axis, ndim, argname=None, allow_duplicate=False):
    """
    Normalizes an axis argument into a tuple of non-negative integer axes.
    This handles shorthands such as ``1`` and converts them to ``(1,)``,
    as well as performing the handling of negative indices covered by
    `normalize_axis_index`.
    By default, this forbids axes from being specified multiple times.
    Used internally by multi-axis-checking logic.
    .. versionadded:: 1.13.0
    Parameters
    -----
    axis : int, iterable of int
        The un-normalized index or indices of the axis.
    ndim : int
        The number of dimensions of the array that `axis` should be normalized
        against.
    argname : str, optional
        A prefix to put before the error message, typically the name of the
        argument.
    allow_duplicate : bool, optional
        If False, the default, disallow an axis from being specified twice.
    Returns
    -----
    normalized_axes : tuple of int
        The normalized axis index, such that `0 <= normalized_axis < ndim`
    Raises
    -----
    AxisError
        If any axis provided is out of range
    ValueError
        If an axis is repeated
    See also
    -----
    normalize_axis_index : normalizing a single scalar axis
    """
    if type(axis) not in (tuple, list):
        try:
            axis = [operator.index(ax)]
        except TypeError:
            pass
    axis = tuple([normalize_axis_index(ax, ndim, argname) for ax in axis])
    if not allow_duplicate and len(set(axis)) != len(axis):
        if argname:
            raise ValueError('repeated axis in `{}` argument'.format(argname))
        else:
            raise ValueError('repeated axis')
    return axis

def _moveaxis_dispatcher(a, source, destination):
    return (a,)

@array_function_dispatch(_moveaxis_dispatcher)
def moveaxis(a, source, destination):
    """
    Move axes of an array to new positions.
    Other axes remain in their original order.
    .. versionadded:: 1.11.0
    Parameters
    -----
    a : np.ndarray
        The array whose axes should be reordered.
    source : int or sequence of int
        Original positions of the axes to move. These must be unique.
    destination : int or sequence of int
        Destination positions for each of the original axes. These must also
        be
        unique.
    Returns
    -----
    result : np.ndarray
        Array with moved axes. This array is a view of the input array.
    """

```

```

1144: (4)           See Also
1145: (4)           -----
1146: (4)           transpose : Permute the dimensions of an array.
1147: (4)           swapaxes : Interchange two axes of an array.
1148: (4)           Examples
1149: (4)           -----
1150: (4)           >>> x = np.zeros((3, 4, 5))
1151: (4)           >>> np.moveaxis(x, 0, -1).shape
1152: (4)           (4, 5, 3)
1153: (4)           >>> np.moveaxis(x, -1, 0).shape
1154: (4)           (5, 3, 4)
1155: (4)           These all achieve the same result:
1156: (4)           >>> np.transpose(x).shape
1157: (4)           (5, 4, 3)
1158: (4)           >>> np.swapaxes(x, 0, -1).shape
1159: (4)           (5, 4, 3)
1160: (4)           >>> np.moveaxis(x, [0, 1], [-1, -2]).shape
1161: (4)           (5, 4, 3)
1162: (4)           >>> np.moveaxis(x, [0, 1, 2], [-1, -2, -3]).shape
1163: (4)           (5, 4, 3)
1164: (4)           """
1165: (4)           try:
1166: (8)             transpose = a.transpose
1167: (4)           except AttributeError:
1168: (8)             a = asarray(a)
1169: (8)             transpose = a.transpose
1170: (4)           source = normalize_axis_tuple(source, a.ndim, 'source')
1171: (4)           destination = normalize_axis_tuple(destination, a.ndim, 'destination')
1172: (4)           if len(source) != len(destination):
1173: (8)             raise ValueError(`source` and `destination` arguments must have `
1174: (25)               'the same number of elements')
1175: (4)           order = [n for n in range(a.ndim) if n not in source]
1176: (4)           for dest, src in sorted(zip(destination, source)):
1177: (8)             order.insert(dest, src)
1178: (4)           result = transpose(order)
1179: (4)           return result
1180: (0)           def _cross_dispatcher(a, b, axisa=None, axisb=None, axisc=None, axis=None):
1181: (4)             return (a, b)
1182: (0)           @array_function_dispatch(_cross_dispatcher)
1183: (0)           def cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None):
1184: (4)             """
1185: (4)             Return the cross product of two (arrays of) vectors.
1186: (4)             The cross product of `a` and `b` in :math:`\mathbb{R}^3` is a vector perpendicular
1187: (4)             to both `a` and `b`. If `a` and `b` are arrays of vectors, the vectors
1188: (4)             are defined by the last axis of `a` and `b` by default, and these axes
1189: (4)             can have dimensions 2 or 3. Where the dimension of either `a` or `b` is
1190: (4)             2, the third component of the input vector is assumed to be zero and the
1191: (4)             cross product calculated accordingly. In cases where both input vectors
1192: (4)             have dimension 2, the z-component of the cross product is returned.
1193: (4)             Parameters
1194: (4)             -----
1195: (4)             a : array_like
1196: (8)               Components of the first vector(s).
1197: (4)             b : array_like
1198: (8)               Components of the second vector(s).
1199: (4)             axisa : int, optional
1200: (8)               Axis of `a` that defines the vector(s). By default, the last axis.
1201: (4)             axisb : int, optional
1202: (8)               Axis of `b` that defines the vector(s). By default, the last axis.
1203: (4)             axisc : int, optional
1204: (8)               Axis of `c` containing the cross product vector(s). Ignored if
1205: (8)               both input vectors have dimension 2, as the return is scalar.
1206: (8)               By default, the last axis.
1207: (4)             axis : int, optional
1208: (8)               If defined, the axis of `a`, `b` and `c` that defines the vector(s)
1209: (8)               and cross product(s). Overrides `axisa`, `axisb` and `axisc`.
1210: (4)             Returns
1211: (4)             -----
1212: (4)             c : ndarray

```

```

1213: (8)           Vector cross product(s).
1214: (4)           Raises
1215: (4)           -----
1216: (4)           ValueError
1217: (8)             When the dimension of the vector(s) in `a` and/or `b` does not
1218: (8)             equal 2 or 3.
1219: (4)           See Also
1220: (4)           -----
1221: (4)             inner : Inner product
1222: (4)             outer : Outer product.
1223: (4)             ix_ : Construct index arrays.
1224: (4)           Notes
1225: (4)           -----
1226: (4)             .. versionadded:: 1.9.0
1227: (4)             Supports full broadcasting of the inputs.
1228: (4)           Examples
1229: (4)           -----
1230: (4)             Vector cross-product.
1231: (4)             >>> x = [1, 2, 3]
1232: (4)             >>> y = [4, 5, 6]
1233: (4)             >>> np.cross(x, y)
1234: (4)             array([-3, 6, -3])
1235: (4)             One vector with dimension 2.
1236: (4)             >>> x = [1, 2]
1237: (4)             >>> y = [4, 5, 6]
1238: (4)             >>> np.cross(x, y)
1239: (4)             array([12, -6, -3])
1240: (4)             Equivalently:
1241: (4)             >>> x = [1, 2, 0]
1242: (4)             >>> y = [4, 5, 6]
1243: (4)             >>> np.cross(x, y)
1244: (4)             array([12, -6, -3])
1245: (4)             Both vectors with dimension 2.
1246: (4)             >>> x = [1, 2]
1247: (4)             >>> y = [4, 5]
1248: (4)             >>> np.cross(x, y)
1249: (4)             array(-3)
1250: (4)             Multiple vector cross-products. Note that the direction of the cross
1251: (4)             product vector is defined by the *right-hand rule*.
1252: (4)             >>> x = np.array([[1,2,3], [4,5,6]])
1253: (4)             >>> y = np.array([[4,5,6], [1,2,3]])
1254: (4)             >>> np.cross(x, y)
1255: (4)             array([[-3, 6, -3],
1256: (11)               [ 3, -6, 3]])
1257: (4)             The orientation of `c` can be changed using the `axisc` keyword.
1258: (4)             >>> np.cross(x, y, axisc=0)
1259: (4)             array([[-3, 3],
1260: (11)               [ 6, -6],
1261: (11)               [-3, 3]])
1262: (4)             Change the vector definition of `x` and `y` using `axisa` and `axisb`.
1263: (4)             >>> x = np.array([[1,2,3], [4,5,6], [7, 8, 9]])
1264: (4)             >>> y = np.array([[7, 8, 9], [4,5,6], [1,2,3]])
1265: (4)             >>> np.cross(x, y)
1266: (4)             array([[ -6, 12, -6],
1267: (11)               [ 0, 0, 0],
1268: (11)               [ 6, -12, 6]])
1269: (4)             >>> np.cross(x, y, axisa=0, axisb=0)
1270: (4)             array([[-24, 48, -24],
1271: (11)               [-30, 60, -30],
1272: (11)               [-36, 72, -36]])
1273: (4)             """
1274: (4)             if axis is not None:
1275: (8)                 axisa, axisb, axisc = (axis,) * 3
1276: (4)             a = asarray(a)
1277: (4)             b = asarray(b)
1278: (4)             axisa = normalize_axis_index(axisa, a.ndim, msg_prefix='axisa')
1279: (4)             axisb = normalize_axis_index(axisb, b.ndim, msg_prefix='axisb')
1280: (4)             a = moveaxis(a, axisa, -1)
1281: (4)             b = moveaxis(b, axisb, -1)

```

```

1282: (4)             msg = ("incompatible dimensions for cross product\n"
1283: (11)             "dimension must be 2 or 3)")
1284: (4)             if a.shape[-1] not in (2, 3) or b.shape[-1] not in (2, 3):
1285: (8)                 raise ValueError(msg)
1286: (4)             shape = broadcast(a[..., 0], b[..., 0]).shape
1287: (4)             if a.shape[-1] == 3 or b.shape[-1] == 3:
1288: (8)                 shape += (3,)
1289: (8)                 axisc = normalize_axis_index(axisc, len(shape), msg_prefix='axisc')
1290: (4)             dtype = promote_types(a.dtype, b.dtype)
1291: (4)             cp = empty(shape, dtype)
1292: (4)             a = a.astype(dtype)
1293: (4)             b = b.astype(dtype)
1294: (4)             a0 = a[..., 0]
1295: (4)             a1 = a[..., 1]
1296: (4)             if a.shape[-1] == 3:
1297: (8)                 a2 = a[..., 2]
1298: (4)             b0 = b[..., 0]
1299: (4)             b1 = b[..., 1]
1300: (4)             if b.shape[-1] == 3:
1301: (8)                 b2 = b[..., 2]
1302: (4)             if cp.ndim != 0 and cp.shape[-1] == 3:
1303: (8)                 cp0 = cp[..., 0]
1304: (8)                 cp1 = cp[..., 1]
1305: (8)                 cp2 = cp[..., 2]
1306: (4)             if a.shape[-1] == 2:
1307: (8)                 if b.shape[-1] == 2:
1308: (12)                   multiply(a0, b1, out=cp)
1309: (12)                   cp -= a1 * b0
1310: (12)                   return cp
1311: (8)             else:
1312: (12)                 assert b.shape[-1] == 3
1313: (12)                 multiply(a1, b2, out=cp0)
1314: (12)                 multiply(a0, b2, out=cp1)
1315: (12)                 negative(cp1, out=cp1)
1316: (12)                 multiply(a0, b1, out=cp2)
1317: (12)                 cp2 -= a1 * b0
1318: (4)             else:
1319: (8)                 assert a.shape[-1] == 3
1320: (8)                 if b.shape[-1] == 3:
1321: (12)                   multiply(a1, b2, out=cp0)
1322: (12)                   tmp = array(a2 * b1)
1323: (12)                   cp0 -= tmp
1324: (12)                   multiply(a2, b0, out=cp1)
1325: (12)                   multiply(a0, b2, out=tmp)
1326: (12)                   cp1 -= tmp
1327: (12)                   multiply(a0, b1, out=cp2)
1328: (12)                   multiply(a1, b0, out=tmp)
1329: (12)                   cp2 -= tmp
1330: (8)             else:
1331: (12)                 assert b.shape[-1] == 2
1332: (12)                 multiply(a2, b1, out=cp0)
1333: (12)                 negative(cp0, out=cp0)
1334: (12)                 multiply(a2, b0, out=cp1)
1335: (12)                 multiply(a0, b1, out=cp2)
1336: (12)                 cp2 -= a1 * b0
1337: (4)             return moveaxis(cp, -1, axisc)
1338: (0)             little_endian = (sys.byteorder == 'little')
1339: (0)             @set_module('numpy')
1340: (0)             def indices(dimensions, dtype=int, sparse=False):
1341: (4)                 """
1342: (4)                     Return an array representing the indices of a grid.
1343: (4)                     Compute an array where the subarrays contain index values 0, 1, ...
1344: (4)                     varying only along the corresponding axis.
1345: (4)                     Parameters
1346: (4)                         -----
1347: (4)                         dimensions : sequence of ints
1348: (8)                             The shape of the grid.
1349: (4)                         dtype : dtype, optional
1350: (8)                             Data type of the result.

```

```

1351: (4) sparse : boolean, optional
1352: (8)     Return a sparse representation of the grid instead of a dense
1353: (8)     representation. Default is False.
1354: (8)     .. versionadded:: 1.17
1355: (4) Returns -----
1356: (4) grid : one ndarray or tuple of ndarrays
1357: (4)     If sparse is False:
1358: (8)         Returns one array of grid indices,
1359: (12)         ``grid.shape = (len(dimensions),) + tuple(dimensions)``.
1360: (12)
1361: (8)     If sparse is True:
1362: (12)         Returns a tuple of arrays, with
1363: (12)         ``grid[i].shape = (1, ..., 1, dimensions[i], 1, ..., 1)`` with
1364: (12)         dimensions[i] in the ith place
1365: (4) See Also -----
1366: (4) mgrid, ogrid, meshgrid
1368: (4) Notes -----
1369: (4) The output shape in the dense case is obtained by prepending the number
1370: (4) of dimensions in front of the tuple of dimensions, i.e. if `dimensions`
1371: (4) is a tuple ``r0, ..., rN-1`` of length ``N``, the output shape is
1372: (4) ``N, r0, ..., rN-1``.
1373: (4) The subarrays ``grid[k]`` contains the N-D array of indices along the
1374: (4) ``k-th`` axis. Explicitly::
1375: (4)     grid[k, i0, i1, ..., iN-1] = ik
1376: (8)
1377: (4) Examples -----
1378: (4)
1379: (4) >>> grid = np.indices((2, 3))
1380: (4) >>> grid.shape
1381: (4) (2, 2, 3)
1382: (4) >>> grid[0]      # row indices
1383: (4) array([[0, 0, 0],
1384: (11)           [1, 1, 1]])
1385: (4) >>> grid[1]      # column indices
1386: (4) array([[0, 1, 2],
1387: (11)           [0, 1, 2]])
1388: (4) The indices can be used as an index into an array.
1389: (4) >>> x = np.arange(20).reshape(5, 4)
1390: (4) >>> row, col = np.indices((2, 3))
1391: (4) >>> x[row, col]
1392: (4) array([[0, 1, 2],
1393: (11)           [4, 5, 6]])
1394: (4) Note that it would be more straightforward in the above example to
1395: (4) extract the required elements directly with ``x[:2, :3]``.
1396: (4) If sparse is set to true, the grid will be returned in a sparse
1397: (4) representation.
1398: (4) >>> i, j = np.indices((2, 3), sparse=True)
1399: (4) >>> i.shape
1400: (4) (2, 1)
1401: (4) >>> j.shape
1402: (4) (1, 3)
1403: (4) >>> i      # row indices
1404: (4) array([[0],
1405: (11)           [1]])
1406: (4) >>> j      # column indices
1407: (4) array([[0, 1, 2]])
1408: (4) """
1409: (4) dimensions = tuple(dimensions)
1410: (4) N = len(dimensions)
1411: (4) shape = (1,)*N
1412: (4) if sparse:
1413: (8)     res = tuple()
1414: (4) else:
1415: (8)     res = empty((N,) + dimensions, dtype=dtype)
1416: (4) for i, dim in enumerate(dimensions):
1417: (8)     idx = arange(dim, dtype=dtype).reshape(
1418: (12)         shape[:i] + (dim,) + shape[i+1:])
1419: (8) 
```

```

1420: (8)
1421: (12)
1422: (8)
1423: (12)
1424: (4)
1425: (0)
1426: (0)
1427: (0)
1428: (4)
1429: (4)
1430: (4)
1431: (4)
1432: (4)
1433: (4)
1434: (4)
1435: (8)
1436: (8)
1437: (8)
1438: (8)
1439: (8)
1440: (4)
1441: (8)
1442: (8)
1443: (4)
1444: (8)
1445: (8)
1446: (4)
1447: (8)
1448: (4)
1449: (4)
1450: (4)
1451: (8)
1452: (8)
1453: (8)
1454: (8)
1455: (4)
1456: (4)
1457: (4)
1458: (4)
1459: (4)
1460: (4)
1461: (4)
1462: (4)
1463: (4)
1464: (4)
1465: (11)
1466: (4)
1467: (4)
1468: (11)
1469: (4)
1470: (4)
1471: (11)
1472: (11)
1473: (4)
1474: (4)
1475: (11)
1476: (11)
1477: (4)
1478: (4)
1479: (8)
1480: (16)
1481: (4)
1482: (4)
1483: (0)
1484: (0)
1485: (4)
1486: (0)
1487: (0)
1488: (4)

        if sparse:
            res = res + (idx,)
        else:
            res[i] = idx
    return res
@set_array_function_like_doc
@set_module('numpy')
def fromfunction(function, shape, *, dtype=float, like=None, **kwargs):
    """
    Construct an array by executing a function over each coordinate.
    The resulting array therefore has a value ``fn(x, y, z)`` at
    coordinate ``(x, y, z)``.

    Parameters
    -----
    function : callable
        The function is called with N parameters, where N is the rank of
        `shape`. Each parameter represents the coordinates of the array
        varying along a specific axis. For example, if `shape`
        were ``(2, 2)``, then the parameters would be
        ``array([[0, 0], [1, 1]])`` and ``array([[0, 1], [0, 1]])``.
    shape : (N,) tuple of ints
        Shape of the output array, which also determines the shape of
        the coordinate arrays passed to `function`.
    dtype : data-type, optional
        Data-type of the coordinate arrays passed to `function`.
        By default, `dtype` is float.
    ${ARRAY_FUNCTION_LIKE}
        .. versionadded:: 1.20.0

    Returns
    -----
    fromfunction : any
        The result of the call to `function` is passed back directly.
        Therefore the shape of `fromfunction` is completely determined by
        `function`. If `function` returns a scalar value, the shape of
        `fromfunction` would not match the `shape` parameter.

    See Also
    -----
    indices, meshgrid
    Notes
    -----
    Keywords other than `dtype` and `like` are passed to `function`.

    Examples
    -----
    >>> np.fromfunction(lambda i, j: i, (2, 2), dtype=float)
    array([[0., 0.],
           [1., 1.]])
    >>> np.fromfunction(lambda i, j: j, (2, 2), dtype=float)
    array([[0., 1.],
           [0., 1.]])
    >>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
    array([[ True, False, False],
           [False,  True, False],
           [False, False,  True]])
    >>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
    array([[0, 1, 2],
           [1, 2, 3],
           [2, 3, 4]])
    """
    if like is not None:
        return _fromfunction_with_like(
            like, function, shape, dtype=dtype, **kwargs)
    args = indices(shape, dtype=dtype)
    return function(*args, **kwargs)
_fromfunction_with_like = array_function_dispatch()(fromfunction)
def _frombuffer(buf, dtype, shape, order):
    return frombuffer(buf, dtype=dtype).reshape(shape, order=order)
@set_module('numpy')
def isscalar(element):
    """

```

```

1489: (4)          Returns True if the type of `element` is a scalar type.
1490: (4)          Parameters
1491: (4)          -----
1492: (4)          element : any
1493: (8)          Input argument, can be of any type and shape.
1494: (4)          Returns
1495: (4)          -----
1496: (4)          val : bool
1497: (8)          True if `element` is a scalar type, False if it is not.
1498: (4)          See Also
1499: (4)          -----
1500: (4)          ndim : Get the number of dimensions of an array
1501: (4)          Notes
1502: (4)          -----
1503: (4)          If you need a stricter way to identify a *numerical* scalar, use
1504: (4)          ```isinstance(x, numbers.Number)```, as that returns ``False`` for most
1505: (4)          non-numerical elements such as strings.
1506: (4)          In most cases ``np.ndim(x) == 0`` should be used instead of this function,
1507: (4)          as that will also return true for 0d arrays. This is how numpy overloads
1508: (4)          functions in the style of the ``dx`` arguments to `gradient` and the
1509: ``bins``
1510: (4)          argument to `histogram`. Some key differences:
+-----+-----+
1511: (4)          | x           | ``isscalar(x)`` | ``np.ndim(x) == 0`` |
1512: (4)
+=====+=====+=====+=====+
1513: (4)          | PEP 3141 numeric objects (including builtins) | ``True`` | ``True`` |
|                                +-----+
1514: (4)          | builtins) | | |
|                                +-----+
1515: (4)          | | |
+-----+-----+
1516: (4)          | builtin string and buffer objects | ``True`` | ``True`` |
|                                +-----+
1517: (4)          | | |
+-----+-----+
1518: (4)          | other builtin objects, like | ``False`` | ``True`` |
|                                +-----+
1519: (4)          | `pathlib.Path`, `Exception`, | | |
|                                +-----+
1520: (4)          | the result of `re.compile` | | |
|                                +-----+
1521: (4)          | | |
+-----+-----+
1522: (4)          | third-party objects like | ``False`` | ``True`` |
|                                +-----+
1523: (4)          | `matplotlib.figure.Figure` | | |
|                                +-----+
1524: (4)          | | |
+-----+-----+
1525: (4)          | zero-dimensional numpy arrays | ``False`` | ``True`` |
|                                +-----+
1526: (4)          | | |
+-----+-----+
1527: (4)          | other numpy arrays | ``False`` | ``False`` |
|                                +-----+
1528: (4)          | | |
+-----+-----+
1529: (4)          | `list`, `tuple`, and other sequence | ``False`` | ``False`` |
|                                +-----+
1530: (4)          | objects | | |
|                                +-----+
1531: (4)          | | |
+-----+-----+
1532: (4)          Examples
1533: (4)          -----
1534: (4)          >>> np.isscalar(3.1)

```

```

1535: (4)          True
1536: (4)          >>> np.isscalar(np.array(3.1))
1537: (4)          False
1538: (4)          >>> np.isscalar([3.1])
1539: (4)          False
1540: (4)          >>> np.isscalar(False)
1541: (4)          True
1542: (4)          >>> np.isscalar('numpy')
1543: (4)          True
1544: (4)          NumPy supports PEP 3141 numbers:
1545: (4)          >>> from fractions import Fraction
1546: (4)          >>> np.isscalar(Fraction(5, 17))
1547: (4)          True
1548: (4)          >>> from numbers import Number
1549: (4)          >>> np.isscalar(Number())
1550: (4)          True
1551: (4)          """
1552: (4)          return (isinstance(element, generic)
1553: (12)          or type(element) in ScalarType
1554: (12)          or isinstance(element, numbers.Number))
1555: (0)          @set_module('numpy')
1556: (0)          def binary_repr(num, width=None):
1557: (4)          """
1558: (4)          Return the binary representation of the input number as a string.
1559: (4)          For negative numbers, if width is not given, a minus sign is added to the
1560: (4)          front. If width is given, the two's complement of the number is
1561: (4)          returned, with respect to that width.
1562: (4)          In a two's-complement system negative numbers are represented by the two's
1563: (4)          complement of the absolute value. This is the most common method of
1564: (4)          representing signed integers on computers [1]_. A N-bit two's-complement
1565: (4)          system can represent every integer in the range
1566: (4)          :math:`-2^{N-1}` to :math:`+2^{N-1}-1`.
1567: (4)          Parameters
1568: (4)          -----
1569: (4)          num : int
1570: (8)          Only an integer decimal number can be used.
1571: (4)          width : int, optional
1572: (8)          The length of the returned string if `num` is positive, or the length
1573: (8)          of the two's complement if `num` is negative, provided that `width` is
1574: (8)          at least a sufficient number of bits for `num` to be represented in
the
1575: (8)          designated form.
1576: (8)          If the `width` value is insufficient, it will be ignored, and `num`
will
1577: (8)          be returned in binary (`num` > 0) or two's complement (`num` < 0) form
1578: (8)          with its width equal to the minimum number of bits needed to represent
1579: (8)          the number in the designated form. This behavior is deprecated and
will
1580: (8)          later raise an error.
1581: (8)          .. deprecate:: 1.12.0
1582: (4)          Returns
1583: (4)          -----
1584: (4)          bin : str
1585: (8)          Binary representation of `num` or two's complement of `num`.
1586: (4)          See Also
1587: (4)          -----
1588: (4)          base_repr: Return a string representation of a number in the given base
1589: (15)          system.
1590: (4)          bin: Python's built-in binary representation generator of an integer.
1591: (4)          Notes
1592: (4)          -----
1593: (4)          `binary_repr` is equivalent to using `base_repr` with base 2, but about
25x
1594: (4)          faster.
1595: (4)          References
1596: (4)          -----
1597: (4)          .. [1] Wikipedia, "Two's complement",
1598: (8)          https://en.wikipedia.org/wiki/Two's_complement
1599: (4)          Examples

```

```

1600: (4)
1601: (4)
1602: (4)
1603: (4)
1604: (4)
1605: (4)
1606: (4)
1607: (4)
1608: (4)
1609: (4)
1610: (4)
1611: (4)
1612: (4)
1613: (4)
1614: (4)
1615: (8)
1616: (12)
1617: (16)
1618: (16)
1619: (16)
1620: (4)
1621: (4)
1622: (8)
1623: (4)
1624: (8)
1625: (8)
1626: (8)
1627: (20)
1628: (8)
1629: (8)
1630: (4)
1631: (8)
1632: (12)
1633: (8)
1634: (12)
1635: (12)
1636: (16)
1637: (12)
1638: (12)
1639: (12)
1640: (12)
1641: (12)
1642: (12)
1643: (0)
1644: (0)
1645: (4)
1646: (4)
1647: (4)
1648: (4)
1649: (4)
1650: (8)
1651: (4)
1652: (8)
1653: (8)
1654: (4)
1655: (8)
1656: (4)
1657: (4)
1658: (4)
1659: (8)
1660: (4)
1661: (4)
1662: (4)
1663: (4)
1664: (4)
1665: (4)
1666: (4)
1667: (4)
1668: (4)

-----  

>>> np.binary_repr(3)  

'11'  

>>> np.binary_repr(-3)  

'-11'  

>>> np.binary_repr(3, width=4)  

'0011'  

The two's complement is returned when the input number is negative and  

width is specified:  

>>> np.binary_repr(-3, width=3)  

'101'  

>>> np.binary_repr(-3, width=5)  

'11101'  

"""  

def warn_if_insufficient(width, binwidth):  

    if width is not None and width < binwidth:  

        warnings.warn(  

            "Insufficient bit width provided. This behavior "  

            "will raise an error in the future.", DeprecationWarning,  

            stacklevel=3)  

    num = operator.index(num)  

if num == 0:  

    return '0' * (width or 1)  

elif num > 0:  

    binary = bin(num)[2:]  

    binwidth = len(binary)  

    outwidth = (binwidth if width is None  

                else builtins.max(binwidth, width))  

    warn_if_insufficient(width, binwidth)  

    return binary.zfill(outwidth)  

else:  

    if width is None:  

        return '-' + bin(-num)[2:]  

    else:  

        poswidth = len(bin(-num)[2:])  

        if 2**poswidth - 1 == -num:  

            poswidth -= 1  

        twocomp = 2**poswidth + num  

        binary = bin(twocomp)[2:]  

        binwidth = len(binary)  

        outwidth = builtins.max(binwidth, width)  

        warn_if_insufficient(width, binwidth)  

        return '1' * (outwidth - binwidth) + binary  

@set_module('numpy')
def base_repr(number, base=2, padding=0):
    """  

        Return a string representation of a number in the given base system.  

    Parameters  

    -----  

    number : int  

        The value to convert. Positive and negative values are handled.  

    base : int, optional  

        Convert `number` to the `base` number system. The valid range is 2-36,  

        the default value is 2.  

    padding : int, optional  

        Number of zeros padded on the left. Default is 0 (no padding).  

    Returns  

    -----  

    out : str  

        String representation of `number` in `base` system.  

    See Also  

    -----  

    binary_repr : Faster version of `base_repr` for base 2.  

    Examples  

    -----  

    >>> np.base_repr(5)  

'101'  

    >>> np.base_repr(6, 5)  

'11'

```

```

1669: (4)             >>> np.base_repr(7, base=5, padding=3)
1670: (4)             '00012'
1671: (4)             >>> np.base_repr(10, base=16)
1672: (4)             'A'
1673: (4)             >>> np.base_repr(32, base=16)
1674: (4)             '20'
1675: (4)             """
1676: (4)             digits = '0123456789ABCDEFGHIJKLMNPQRSTUVWXYZ'
1677: (4)             if base > len(digits):
1678: (8)                 raise ValueError("Bases greater than 36 not handled in base_repr.")
1679: (4)             elif base < 2:
1680: (8)                 raise ValueError("Bases less than 2 not handled in base_repr.")
1681: (4)             num = abs(number)
1682: (4)             res = []
1683: (4)             while num:
1684: (8)                 res.append(digits[num % base])
1685: (8)                 num //= base
1686: (4)             if padding:
1687: (8)                 res.append('0' * padding)
1688: (4)             if number < 0:
1689: (8)                 res.append('-')
1690: (4)             return ''.join(reversed(res or '0'))
1691: (0)             def _maketup(descr, val):
1692: (4)                 dt = dtype(descr)
1693: (4)                 fields = dt.fields
1694: (4)                 if fields is None:
1695: (8)                     return val
1696: (4)                 else:
1697: (8)                     res = [_maketup(fields[name][0], val) for name in dt.names]
1698: (8)                     return tuple(res)
1699: (0)             @set_array_function_like_doc
1700: (0)             @set_module('numpy')
1701: (0)             def identity(n, dtype=None, *, like=None):
1702: (4)                 """
1703: (4)                 Return the identity array.
1704: (4)                 The identity array is a square array with ones on
1705: (4)                 the main diagonal.
1706: (4)                 Parameters
1707: (4)                 -----
1708: (4)                 n : int
1709: (8)                     Number of rows (and columns) in `n` x `n` output.
1710: (4)                 dtype : data-type, optional
1711: (8)                     Data-type of the output. Defaults to ``float``.
1712: (4)                     ${ARRAY_FUNCTION_LIKE}
1713: (8)                     .. versionadded:: 1.20.0
1714: (4)                 Returns
1715: (4)                 -----
1716: (4)                 out : ndarray
1717: (8)                     `n` x `n` array with its main diagonal set to one,
1718: (8)                     and all other elements 0.
1719: (4)                 Examples
1720: (4)                 -----
1721: (4)                 >>> np.identity(3)
1722: (4)                 array([[1.,  0.,  0.],
1723: (11)                   [0.,  1.,  0.],
1724: (11)                   [0.,  0.,  1.]])
1725: (4)                 """
1726: (4)                 if like is not None:
1727: (8)                     return _identity_with_like(like, n, dtype=dtype)
1728: (4)                     from numpy import eye
1729: (4)                     return eye(n, dtype=dtype, like=like)
1730: (0)                     _identity_with_like = array_function_dispatch()(identity)
1731: (0)             def _allclose_dispatcher(a, b, rtol=None, atol=None, equal_nan=None):
1732: (4)                 return (a, b)
1733: (0)             @array_function_dispatch(_allclose_dispatcher)
1734: (0)             def allclose(a, b, rtol=1.e-5, atol=1.e-8, equal_nan=False):
1735: (4)                 """
1736: (4)                 Returns True if two arrays are element-wise equal within a tolerance.
1737: (4)                 The tolerance values are positive, typically very small numbers. The

```

```

1738: (4) relative difference (`rtol` * abs(`b`)) and the absolute difference
1739: (4) `atol` are added together to compare against the absolute difference
1740: (4) between `a` and `b`.
1741: (4) NaNs are treated as equal if they are in the same place and if
1742: (4) ``equal_nan=True``. Infs are treated as equal if they are in the same
1743: (4) place and of the same sign in both arrays.
1744: (4) Parameters
1745: (4) -----
1746: (4) a, b : array_like
1747: (8) Input arrays to compare.
1748: (4) rtol : float
1749: (8) The relative tolerance parameter (see Notes).
1750: (4) atol : float
1751: (8) The absolute tolerance parameter (see Notes).
1752: (4) equal_nan : bool
1753: (8) Whether to compare NaN's as equal. If True, NaN's in `a` will be
1754: (8) considered equal to NaN's in `b` in the output array.
1755: (8) .. versionadded:: 1.10.0
1756: (4) Returns
1757: (4) -----
1758: (4) allclose : bool
1759: (8) Returns True if the two arrays are equal within the given
1760: (8) tolerance; False otherwise.
1761: (4) See Also
1762: (4) -----
1763: (4) isclose, all, any, equal
1764: (4) Notes
1765: (4) -----
1766: (4) If the following equation is element-wise True, then allclose returns
1767: (4) True.
1768: (5) absolute(`a` - `b`) <= (`atol` + `rtol` * absolute(`b`))
1769: (4) The above equation is not symmetric in `a` and `b`, so that
1770: (4) ``allclose(a, b)`` might be different from ``allclose(b, a)`` in
1771: (4) some rare cases.
1772: (4) The comparison of `a` and `b` uses standard broadcasting, which
1773: (4) means that `a` and `b` need not have the same shape in order for
1774: (4) ``allclose(a, b)`` to evaluate to True. The same is true for
1775: (4) `equal` but not `array_equal`.
1776: (4) `allclose` is not defined for non-numeric data types.
1777: (4) `bool` is considered a numeric data-type for this purpose.
1778: (4) Examples
1779: (4) -----
1780: (4) >>> np.allclose([1e10,1e-7], [1.00001e10,1e-8])
1781: (4) False
1782: (4) >>> np.allclose([1e10,1e-8], [1.00001e10,1e-9])
1783: (4) True
1784: (4) >>> np.allclose([1e10,1e-8], [1.0001e10,1e-9])
1785: (4) False
1786: (4) >>> np.allclose([1.0, np.nan], [1.0, np.nan])
1787: (4) False
1788: (4) >>> np.allclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
1789: (4) True
1790: (4) """
1791: (4)     res = all(isclose(a, b, rtol=rtol, atol=atol, equal_nan=equal_nan))
1792: (4)     return bool(res)
1793: (0) def _isclose_dispatcher(a, b, rtol=None, atol=None, equal_nan=None):
1794: (4)     return (a, b)
1795: (0) @array_function_dispatch(_isclose_dispatcher)
1796: (0) def isclose(a, b, rtol=1.e-5, atol=1.e-8, equal_nan=False):
1797: (4) """
1798: (4)     Returns a boolean array where two arrays are element-wise equal within a
1799: (4) tolerance.
1800: (4)     The tolerance values are positive, typically very small numbers. The
1801: (4) relative difference (`rtol` * abs(`b`)) and the absolute difference
1802: (4) `atol` are added together to compare against the absolute difference
1803: (4) between `a` and `b`.
1804: (4)     .. warning:: The default `atol` is not appropriate for comparing numbers
1805: (17)             that are much smaller than one (see Notes).
1806: (4) Parameters

```

```

1807: (4)
1808: (4)
1809: (8)
1810: (4)
1811: (8)
1812: (4)
1813: (8)
1814: (4)
1815: (8)
1816: (8)
1817: (4)
1818: (4)
1819: (4)
1820: (8)
1821: (8)
1822: (8)
1823: (4)
1824: (4)
1825: (4)
1826: (4)
1827: (4)
1828: (4)
1829: (4)
1830: (4)
1831: (4)
1832: (5)
1833: (4)
1834: (4)
1835: (4)
1836: (4)
1837: (4)
1838: (4)
1839: (4)
1840: (4)
1841: (4)
1842: (4)
1843: (4)
1844: (4)
1845: (4)
1846: (4)
1847: (4)
1848: (4)
1849: (4)
1850: (4)
1851: (4)
1852: (4)
1853: (4)
1854: (4)
1855: (4)
1856: (4)
1857: (4)
1858: (4)
1859: (4)
1860: (4)
1861: (4)
1862: (4)
1863: (4)
1864: (4)
1865: (4)
1866: (8)
1867: (12)
1868: (4)
1869: (4)
1870: (4)
1871: (8)
1872: (8)
1873: (4)
1874: (4)
1875: (4)

-----  

1808: (4)  

1809: (8)  

1810: (4)  

1811: (8)  

1812: (4)  

1813: (8)  

1814: (4)  

1815: (8)  

1816: (8)  

1817: (4)  

1818: (4)  

1819: (4)  

1820: (8)  

1821: (8)  

1822: (8)  

1823: (4)  

1824: (4)  

1825: (4)  

1826: (4)  

1827: (4)  

1828: (4)  

1829: (4)  

1830: (4)  

1831: (4)  

1832: (5)  

1833: (4)  

1834: (4)  

1835: (4)  

1836: (4)  

1837: (4)  

1838: (4)  

1839: (4)  

1840: (4)  

1841: (4)  

1842: (4)  

1843: (4)  

1844: (4)  

1845: (4)  

1846: (4)  

1847: (4)  

1848: (4)  

1849: (4)  

1850: (4)  

1851: (4)  

1852: (4)  

1853: (4)  

1854: (4)  

1855: (4)  

1856: (4)  

1857: (4)  

1858: (4)  

1859: (4)  

1860: (4)  

1861: (4)  

1862: (4)  

1863: (4)  

1864: (4)  

1865: (4)  

1866: (8)  

1867: (12)  

1868: (4)  

1869: (4)  

1870: (4)  

1871: (8)  

1872: (8)  

1873: (4)  

1874: (4)  

1875: (4)

-----  

a, b : array_like  

      Input arrays to compare.  

rtol : float  

      The relative tolerance parameter (see Notes).  

atol : float  

      The absolute tolerance parameter (see Notes).  

equal_nan : bool  

      Whether to compare NaN's as equal. If True, NaN's in `a` will be  

      considered equal to NaN's in `b` in the output array.  

>Returns  

-----  

y : array_like  

      Returns a boolean array of where `a` and `b` are equal within the  

      given tolerance. If both `a` and `b` are scalars, returns a single  

      boolean value.  

See Also  

-----  

allclose  

math.isclose  

Notes  

-----  

.. versionadded:: 1.7.0  

For finite values, isclose uses the following equation to test whether  

two floating point values are equivalent.  


$$\text{absolute}(a - b) \leq (\text{atol} + \text{rtol} * \text{absolute}(b))$$
  

Unlike the built-in `math.isclose`, the above equation is not symmetric  

in `a` and `b` -- it assumes `b` is the reference value -- so that  

`isclose(a, b)` might be different from `isclose(b, a)`. Furthermore,  

the default value of atol is not zero, and is used to determine what  

small values should be considered close to zero. The default value is  

appropriate for expected values of order unity: if the expected values  

are significantly smaller than one, it can result in false positives.  

`atol` should be carefully selected for the use case at hand. A zero value  

for `atol` will result in `False` if either `a` or `b` is zero.  

`isclose` is not defined for non-numeric data types.  

`bool` is considered a numeric data-type for this purpose.  

Examples  

-----  

>>> np.isclose([1e10,1e-7], [1.00001e10,1e-8])  

array([ True, False])  

>>> np.isclose([1e10,1e-8], [1.00001e10,1e-9])  

array([ True, True])  

>>> np.isclose([1e10,1e-8], [1.0001e10,1e-9])  

array([False, True])  

>>> np.isclose([1.0, np.nan], [1.0, np.nan])  

array([ True, False])  

>>> np.isclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)  

array([ True, True])  

>>> np.isclose([1e-8, 1e-7], [0.0, 0.0])  

array([ True, False])  

>>> np.isclose([1e-100, 1e-7], [0.0, 0.0], atol=0.0)  

array([False, False])  

>>> np.isclose([1e-10, 1e-10], [1e-20, 0.0])  

array([ True, True])  

>>> np.isclose([1e-10, 1e-10], [1e-20, 0.999999e-10], atol=0.0)  

array([False, True])  

"""
def within_tol(x, y, atol, rtol):
    with errstate(invalid='ignore'), _no_nep50_warning():
        return less_equal(abs(x-y), atol + rtol * abs(y))
x = asanyarray(a)
y = asanyarray(b)
if y.dtype.kind != "m":  

    dt = multiarray.result_type(y, 1.)
    y = asanyarray(y, dtype=dt)
xfin = isfinite(x)
yfin = isfinite(y)
if all(xfin) and all(yfin):  


```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1876: (8)           return within_tol(x, y, atol, rtol)
1877: (4)           else:
1878: (8)             finite = xfin & yfin
1879: (8)             cond = zeros_like(finite, subok=True)
1880: (8)             x = x * ones_like(cond)
1881: (8)             y = y * ones_like(cond)
1882: (8)             cond[finite] = within_tol(x[finite], y[finite], atol, rtol)
1883: (8)             cond[~finite] = (x[~finite] == y[~finite])
1884: (8)             if equal_nan:
1885: (12)               both_nan = isnan(x) & isnan(y)
1886: (12)               cond[both_nan] = both_nan[both_nan]
1887: (8)             return cond[()] # Flatten 0d arrays to scalars
1888: (0)           def _array_equal_dispatcher(a1, a2, equal_nan=None):
1889: (4)             return (a1, a2)
1890: (0)           @array_function_dispatch(_array_equal_dispatcher)
1891: (0)           def array_equal(a1, a2, equal_nan=False):
1892: (4)             """
1893: (4)             True if two arrays have the same shape and elements, False otherwise.
1894: (4)             Parameters
1895: (4)             -----
1896: (4)             a1, a2 : array_like
1897: (8)               Input arrays.
1898: (4)             equal_nan : bool
1899: (8)               Whether to compare NaN's as equal. If the dtype of a1 and a2 is
1900: (8)               complex, values will be considered equal if either the real or the
1901: (8)               imaginary component of a given value is ``nan``.
1902: (8)               .. versionadded:: 1.19.0
1903: (4)             Returns
1904: (4)             -----
1905: (4)             b : bool
1906: (8)               Returns True if the arrays are equal.
1907: (4)             See Also
1908: (4)             -----
1909: (4)             allclose: Returns True if two arrays are element-wise equal within a
1910: (14)             tolerance.
1911: (4)             array_equiv: Returns True if input arrays are shape consistent and all
1912: (17)             elements equal.
1913: (4)             Examples
1914: (4)             -----
1915: (4)             >>> np.array_equal([1, 2], [1, 2])
1916: (4)             True
1917: (4)             >>> np.array_equal(np.array([1, 2]), np.array([1, 2]))
1918: (4)             True
1919: (4)             >>> np.array_equal([1, 2], [1, 2, 3])
1920: (4)             False
1921: (4)             >>> np.array_equal([1, 2], [1, 4])
1922: (4)             False
1923: (4)             >>> a = np.array([1, np.nan])
1924: (4)             >>> np.array_equal(a, a)
1925: (4)             False
1926: (4)             >>> np.array_equal(a, a, equal_nan=True)
1927: (4)             True
1928: (4)             When ``equal_nan`` is True, complex values with nan components are
1929: (4)             considered equal if either the real *or* the imaginary components are nan.
1930: (4)             >>> a = np.array([1 + 1j])
1931: (4)             >>> b = a.copy()
1932: (4)             >>> a.real = np.nan
1933: (4)             >>> b.imag = np.nan
1934: (4)             >>> np.array_equal(a, b, equal_nan=True)
1935: (4)             True
1936: (4)             """
1937: (4)             try:
1938: (8)               a1, a2 = asarray(a1), asarray(a2)
1939: (4)             except Exception:
1940: (8)               return False
1941: (4)             if a1.shape != a2.shape:
1942: (8)               return False
1943: (4)             if not equal_nan:
1944: (8)               return bool(asarray(a1 == a2).all())

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1945: (4)         a1nan, a2nan = isnan(a1), isnan(a2)
1946: (4)         if not (a1nan == a2nan).all():
1947: (8)             return False
1948: (4)         return bool(asarray(a1[~a1nan] == a2[~a1nan]).all())
1949: (0)     def _array_equiv_dispatcher(a1, a2):
1950: (4)         return (a1, a2)
1951: (0)     @array_function_dispatch(_array_equiv_dispatcher)
1952: (0)     def array_equiv(a1, a2):
1953: (4)         """
1954: (4)             Returns True if input arrays are shape consistent and all elements equal.
1955: (4)             Shape consistent means they are either the same shape, or one input array
1956: (4)             can be broadcasted to create the same shape as the other one.
1957: (4)             Parameters
1958: (4)             -----
1959: (4)             a1, a2 : array_like
1960: (8)                 Input arrays.
1961: (4)             Returns
1962: (4)             -----
1963: (4)             out : bool
1964: (8)                 True if equivalent, False otherwise.
1965: (4)             Examples
1966: (4)             -----
1967: (4)             >>> np.array_equiv([1, 2], [1, 2])
1968: (4)             True
1969: (4)             >>> np.array_equiv([1, 2], [1, 3])
1970: (4)             False
1971: (4)             Showing the shape equivalence:
1972: (4)             >>> np.array_equiv([1, 2], [[1, 2], [1, 2]])
1973: (4)             True
1974: (4)             >>> np.array_equiv([1, 2], [[1, 2, 1, 2], [1, 2, 1, 2]])
1975: (4)             False
1976: (4)             >>> np.array_equiv([1, 2], [[1, 2], [1, 3]])
1977: (4)             False
1978: (4)             """
1979: (4)             try:
1980: (8)                 a1, a2 = asarray(a1), asarray(a2)
1981: (4)             except Exception:
1982: (8)                 return False
1983: (4)             try:
1984: (8)                 multiarray.broadcast(a1, a2)
1985: (4)             except Exception:
1986: (8)                 return False
1987: (4)             return bool(asarray(a1 == a2).all())
1988: (0)     Inf = inf = infty = Infinity = PINF
1989: (0)     nan = NaN = NAN
1990: (0)     False_ = bool_(False)
1991: (0)     True_ = bool_(True)
1992: (0)     def extend_all(module):
1993: (4)         existing = set(__all__)
1994: (4)         mall = getattr(module, '__all__')
1995: (4)         for a in mall:
1996: (8)             if a not in existing:
1997: (12)                 __all__.append(a)
1998: (0)     from .umath import *
1999: (0)     from .numerictypes import *
2000: (0)     from . import fromnumeric
2001: (0)     from .fromnumeric import *
2002: (0)     from . import arrayprint
2003: (0)     from .arrayprint import *
2004: (0)     from . import _asarray
2005: (0)     from ._asarray import *
2006: (0)     from . import _ufunc_config
2007: (0)     from ._ufunc_config import *
2008: (0)     extend_all(fromnumeric)
2009: (0)     extend_all(umath)
2010: (0)     extend_all(numerictypes)
2011: (0)     extend_all(arrayprint)
2012: (0)     extend_all(_asarray)
2013: (0)     extend_all(_ufunc_config)

```

-----  
File 54 - numericatypes.py:

```

1: (0)      """
2: (0)      numericatypes: Define the numeric type objects
3: (0)      This module is designed so "from numericatypes import \\"*\" is safe.
4: (0)      Exported symbols include:
5: (2)          Dictionary with all registered number types (including aliases):
6: (4)              sctypeDict
7: (2)          Type objects (not all will be available, depends on platform):
8: (6)              see variable scotypes for which ones you have
9: (4)          Bit-width names
10: (4)              int8 int16 int32 int64 int128
11: (4)              uint8 uint16 uint32 uint64 uint128
12: (4)              float16 float32 float64 float96 float128 float256
13: (4)              complex32 complex64 complex128 complex192 complex256 complex512
14: (4)              datetime64 timedelta64
15: (4)          c-based names
16: (4)              bool_
17: (4)              object_
18: (4)              void, str_, unicode_
19: (4)              byte, ubyte,
20: (4)              short, ushort
21: (4)              intc, uintc,
22: (4)              intp, uintp,
23: (4)              int_, uint,
24: (4)              longlong, ulonglong,
25: (4)              single, csingle,
26: (4)              float_, complex_,
27: (4)              longfloat, clongfloat,
28: (3)          As part of the type-hierarchy:    xx -- is bit-width
29: (3)          generic
30: (5)              +-> bool_                                (kind=b)
31: (5)              +-> number
32: (5)                  +-> integer
33: (5)                      +-> signedinteger   (intxx)      (kind=i)
34: (5)                          | byte
35: (5)                          | short
36: (5)                          | intc
37: (5)                          | intp
38: (5)                          | int_
39: (5)                          | longlong
40: (5)                      \\-> unsignedinteger (uintxx)     (kind=u)
41: (5)                          | ubyte
42: (5)                          | ushort
43: (5)                          | uintc
44: (5)                          | uintp
45: (5)                          | uint_
46: (5)                          | ulonglong
47: (5)              +-> inexact
48: (5)                  +-> floating       (floatxx)     (kind=f)
49: (5)                      | half
50: (5)                      | single
51: (5)                      | float_          (double)
52: (5)                      | longfloat
53: (5)                  \\-> complexfloating (complexxx) (kind=c)
54: (5)                      | csingle        (singlecomplex)
55: (5)                      | complex_       (cfloat, cdouble)
56: (5)                      | clongfloat    (longcomplex)
57: (5)              +-> flexible
58: (5)                  +-> character
59: (5)                      | str_           (string_, bytes_)   (kind=S)      [Python 2]
60: (5)                      | unicode_       (unicode_)      (kind=U)      [Python 2]
61: (5)
62: (5)                      | bytes_         (string_)      (kind=S)      [Python 3]
63: (5)                      | str_           (unicode_)     (kind=U)      [Python 3]
64: (5)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

65: (5)           |   \\-> void                               (kind=V)
66: (5)           |   \\-> object_ (not used much)          (kind=O)
67: (0)           """
68: (0)           import numbers
69: (0)           import warnings
70: (0)           from .multiarray import (
71: (8)             ndarray, array, dtype, datetime_data, datetime_as_string,
72: (8)             busday_offset, busday_count, is_busday, busdaycalendar
73: (8)             )
74: (0)           from .._utils import set_module
75: (0)           __all__ = ['sctypeDict', 'sctypes',
76: (11)             'ScalarType', 'obj2sctype', 'cast', ' nbytes', 'sctype2char',
77: (11)             'maximum_sctype', 'issctype', 'typecodes', 'find_common_type',
78: (11)             'issubdtype', 'datetime_data', 'datetime_as_string',
79: (11)             'busday_offset', 'busday_count', 'is_busday', 'busdaycalendar',
80: (11)             ]
81: (0)           from ._string_helpers import (
82: (4)             english_lower, english_upper, english_capitalize, LOWER_TABLE, UPPER_TABLE
83: (0)
84: (0)           from ._type_aliases import (
85: (4)             sctypeDict,
86: (4)             allTypes,
87: (4)             bitname,
88: (4)             sctypes,
89: (4)             _concrete_types,
90: (4)             _concrete_typeinfo,
91: (4)             _bits_of,
92: (0)
93: (0)           from ._dtype import _kind_name
94: (0)           from builtins import bool, int, float, complex, object, str, bytes
95: (0)           from numpy.compat import long, unicode
96: (0)           generic = allTypes['generic']
97: (0)           genericTypeRank = ['bool', 'int8', 'uint8', 'int16', 'uint16',
98: (19)             'int32', 'uint32', 'int64', 'uint64', 'int128',
99: (19)             'uint128', 'float16',
100: (19)             'float32', 'float64', 'float80', 'float96', 'float128',
101: (19)             'float256',
102: (19)             'complex32', 'complex64', 'complex128', 'complex160',
103: (19)             'complex192', 'complex256', 'complex512', 'object']
104: (0)           @set_module('numpy')
105: (0)           def maximum_sctype(t):
106: (4)             """
107: (4)               Return the scalar type of highest precision of the same kind as the input.
108: (4)               Parameters
109: (4)               -----
110: (4)               t : dtype or dtype specifier
111: (8)                 The input data type. This can be a `dtype` object or an object that
112: (8)                   is convertible to a `dtype`.
113: (4)               Returns
114: (4)               -----
115: (4)               out : dtype
116: (8)                 The highest precision data type of the same kind (`dtype.kind`) as
`t`.
117: (4)               See Also
118: (4)               -----
119: (4)               obj2sctype, mintypecode, sctype2char
120: (4)               dtype
121: (4)               Examples
122: (4)               -----
123: (4)               >>> np.maximum_sctype(int)
124: (4)               <class 'numpy.int64'>
125: (4)               >>> np.maximum_sctype(np.uint8)
126: (4)               <class 'numpy.uint64'>
127: (4)               >>> np.maximum_sctype(complex)
128: (4)               <class 'numpy.complex256'> # may vary
129: (4)               >>> np.maximum_sctype(str)
130: (4)               <class 'numpy.str_'>
131: (4)               >>> np.maximum_sctype('i2')
132: (4)               <class 'numpy.int64'>

```

```

133: (4)          >>> np.maximum_sctype('f4')
134: (4)          <class 'numpy.float128'> # may vary
135: (4)          """
136: (4)          g = obj2sctype(t)
137: (4)          if g is None:
138: (8)              return t
139: (4)          t = g
140: (4)          base = _kind_name(dtype(t))
141: (4)          if base in sctypes:
142: (8)              return sctypes[base][-1]
143: (4)          else:
144: (8)              return t
145: (0)          @set_module('numpy')
146: (0)          def issctype(rep):
147: (4)          """
148: (4)          Determines whether the given object represents a scalar data-type.
149: (4)          Parameters
150: (4)          -----
151: (4)          rep : any
152: (8)          If `rep` is an instance of a scalar dtype, True is returned. If not,
153: (8)          False is returned.
154: (4)          Returns
155: (4)          -----
156: (4)          out : bool
157: (8)          Boolean result of check whether `rep` is a scalar dtype.
158: (4)          See Also
159: (4)          -----
160: (4)          issubscotype, issubdtype, obj2sctype, sctype2char
161: (4)          Examples
162: (4)          -----
163: (4)          >>> np.issctype(np.int32)
164: (4)          True
165: (4)          >>> np.issctype(list)
166: (4)          False
167: (4)          >>> np.issctype(1.1)
168: (4)          False
169: (4)          Strings are also a scalar type:
170: (4)          >>> np.issctype(np.dtype('str'))
171: (4)          True
172: (4)          """
173: (4)          if not isinstance(rep, (type, dtype)):
174: (8)              return False
175: (4)          try:
176: (8)              res = obj2sctype(rep)
177: (8)              if res and res != object_:
178: (12)                  return True
179: (8)              return False
180: (4)          except Exception:
181: (8)              return False
182: (0)          @set_module('numpy')
183: (0)          def obj2sctype(rep, default=None):
184: (4)          """
185: (4)          Return the scalar dtype or NumPy equivalent of Python type of an object.
186: (4)          Parameters
187: (4)          -----
188: (4)          rep : any
189: (8)          The object of which the type is returned.
190: (4)          default : any, optional
191: (8)          If given, this is returned for objects whose types can not be
192: (8)          determined. If not given, None is returned for those objects.
193: (4)          Returns
194: (4)          -----
195: (4)          dtype : dtype or Python type
196: (8)          The data type of `rep`.
197: (4)          See Also
198: (4)          -----
199: (4)          sctype2char, issctype, issubscotype, issubdtype, maximum_sctype
200: (4)          Examples
201: (4)          -----

```

```

202: (4)                                     >>> np.obj2sctype(np.int32)
203: (4)                                     <class 'numpy.int32'>
204: (4)                                     >>> np.obj2sctype(np.array([1., 2.]))
205: (4)                                     <class 'numpy.float64'>
206: (4)                                     >>> np.obj2sctype(np.array([1.j]))
207: (4)                                     <class 'numpy.complex128'>
208: (4)                                     >>> np.obj2sctype(dict)
209: (4)                                     <class 'numpy.object_'>
210: (4)                                     >>> np.obj2sctype('string')
211: (4)                                     >>> np.obj2sctype(1, default=list)
212: (4)                                     <class 'list'>
213: (4)                                     """
214: (4)                                     if isinstance(rep, type) and issubclass(rep, generic):
215: (8)   return rep
216: (4)                                     if isinstance(rep, ndarray):
217: (8)   return rep.dtype.type
218: (4)                                     try:
219: (8)   res = dtype(rep)
220: (4)                                     except Exception:
221: (8)   return default
222: (4)                                     else:
223: (8)   return res.type
224: (0) @set_module('numpy')
225: (0) def issubclass_(arg1, arg2):
226: (4) """
227: (4)                                     Determine if a class is a subclass of a second class.
228: (4)                                     `issubclass_` is equivalent to the Python built-in ``issubclass``,
229: (4)                                     except that it returns False instead of raising a TypeError if one
230: (4)                                     of the arguments is not a class.
231: (4)                                     Parameters
232: (4)                                     -----
233: (4)                                     arg1 : class
234: (8)   Input class. True is returned if `arg1` is a subclass of `arg2`.
235: (4)                                     arg2 : class or tuple of classes.
236: (8)   Input class. If a tuple of classes, True is returned if `arg1` is a
237: (8)   subclass of any of the tuple elements.
238: (4)                                     Returns
239: (4)                                     -----
240: (4)                                     out : bool
241: (8)   Whether `arg1` is a subclass of `arg2` or not.
242: (4)                                     See Also
243: (4)                                     -----
244: (4)                                     issubsctype, issubdtype, issctype
245: (4)                                     Examples
246: (4)                                     -----
247: (4)                                     >>> np.issubclass_(np.int32, int)
248: (4)   False
249: (4)                                     >>> np.issubclass_(np.int32, float)
250: (4)   False
251: (4)                                     >>> np.issubclass_(np.float64, float)
252: (4)   True
253: (4)                                     """
254: (4)                                     try:
255: (8)   return issubclass(arg1, arg2)
256: (4)                                     except TypeError:
257: (8)   return False
258: (0) @set_module('numpy')
259: (0) def issubsctype(arg1, arg2):
260: (4) """
261: (4)                                     Determine if the first argument is a subclass of the second argument.
262: (4)                                     Parameters
263: (4)                                     -----
264: (4)                                     arg1, arg2 : dtype or dtype specifier
265: (8)   Data-types.
266: (4)                                     Returns
267: (4)                                     -----
268: (4)                                     out : bool
269: (8)   The result.
270: (4)                                     See Also

```

```

271: (4)      -----
272: (4)      issctype, issubdtype, obj2sctype
273: (4)      Examples
274: (4)      -----
275: (4)      >>> np.issubdtype('S8', str)
276: (4)      False
277: (4)      >>> np.issubdtype(np.array([1]), int)
278: (4)      True
279: (4)      >>> np.issubdtype(np.array([1]), float)
280: (4)      False
281: (4)      """
282: (4)      return issubclass(obj2sctype(arg1), obj2sctype(arg2))
283: (0)      @set_module('numpy')
284: (0)      def issubdtype(arg1, arg2):
285: (4)          """
286: (4)          Returns True if first argument is a typecode lower/equal in type
hierarchy.
287: (4)          This is like the builtin :func:`issubclass`, but for `dtype`\ s.
288: (4)          Parameters
289: (4)          -----
290: (4)          arg1, arg2 : dtype_like
291: (8)              `dtype` or object coercible to one
292: (4)          Returns
293: (4)          -----
294: (4)          out : bool
295: (4)          See Also
296: (4)          -----
297: (4)          :ref:`arrays.scalars` : Overview of the numpy type hierarchy.
298: (4)          isssubtype, issubclass_
299: (4)          Examples
300: (4)          -----
301: (4)          `issubdtype` can be used to check the type of arrays:
302: (4)          >>> ints = np.array([1, 2, 3], dtype=np.int32)
303: (4)          >>> np.issubdtype(ints.dtype, np.integer)
304: (4)          True
305: (4)          >>> np.issubdtype(ints.dtype, np.floating)
306: (4)          False
307: (4)          >>> floats = np.array([1, 2, 3], dtype=np.float32)
308: (4)          >>> np.issubdtype(floats.dtype, np.integer)
309: (4)          False
310: (4)          >>> np.issubdtype(floats.dtype, np.floating)
311: (4)          True
312: (4)          Similar types of different sizes are not subtypes of each other:
313: (4)          >>> np.issubdtype(np.float64, np.float32)
314: (4)          False
315: (4)          >>> np.issubdtype(np.float32, np.float64)
316: (4)          False
317: (4)          but both are subtypes of `floating`:
318: (4)          >>> np.issubdtype(np.float64, np.floating)
319: (4)          True
320: (4)          >>> np.issubdtype(np.float32, np.floating)
321: (4)          True
322: (4)          For convenience, dtype-like objects are allowed too:
323: (4)          >>> np.issubdtype('S1', np.string_)
324: (4)          True
325: (4)          >>> np.issubdtype('i4', np.signedinteger)
326: (4)          True
327: (4)          """
328: (4)          if not issubclass_(arg1, generic):
329: (8)              arg1 = dtype(arg1).type
330: (4)          if not issubclass_(arg2, generic):
331: (8)              arg2 = dtype(arg2).type
332: (4)          return issubclass(arg1, arg2)
333: (0)      class _typedict(dict):
334: (4)          """
335: (4)          Base object for a dictionary for look-up with any alias for an array
dtype.
336: (4)          Instances of `_typedict` can not be used as dictionaries directly,
337: (4)          first they have to be populated.

```

```

338: (4)
339: (4)
340: (8)
341: (0)
342: (0)
343: (0)
344: (0)
345: (0)
346: (4)
347: (8)
348: (8)
349: (8)
350: (8)
351: (12)
352: (12)
353: (8)
354: (12)
355: (12)
356: (0)
357: (0)
358: (0)
359: (4)
360: (4)
361: (4)
362: (4)
363: (4)
364: (8)
365: (8)
366: (8)
367: (4)
368: (4)
369: (4)
370: (8)
371: (4)
372: (4)
373: (4)
374: (8)
375: (4)
376: (4)
377: (4)
378: (4)
379: (4)
380: (4)
np.ndarray]:
381: (4)
382: (4)
383: (4)
384: (4)
385: (4)
386: (4)
387: (4)
388: (4)
389: (4)
390: (4)
391: (4)
392: (4)
393: (4)
394: (4)
395: (8)
396: (4)
397: (8)
398: (4)
399: (0)
400: (0)
401: (4)
402: (0)
403: (4)
404: (4)
405: (4)

        """
        def __getitem__(self, obj):
            return dict.__getitem__(self, obj2sctype(obj))
        nbytes = _typedict()
        _alignment = _typedict()
        _maxvals = _typedict()
        _minvals = _typedict()
        def __construct_lookups():
            for name, info in _concrete_typeinfo.items():
                obj = info.type
                nbytes[obj] = info.bits // 8
                _alignment[obj] = info.alignment
                if len(info) > 5:
                    _maxvals[obj] = info.max
                    _minvals[obj] = info.min
                else:
                    _maxvals[obj] = None
                    _minvals[obj] = None
        __construct_lookups()
    @set_module('numpy')
    def sctype2char(sctype):
        """
        Return the string representation of a scalar dtype.

        Parameters
        -----
        sctype : scalar dtype or object
            If a scalar dtype, the corresponding string character is
            returned. If an object, `sctype2char` tries to infer its scalar type
            and then return the corresponding string character.

        Returns
        -----
        typechar : str
            The string character corresponding to the scalar type.

        Raises
        -----
        ValueError
            If `sctype` is an object for which the type can not be inferred.

        See Also
        -----
        obj2sctype, issctype, issubsctype, mintypecode

        Examples
        -----
        >>> for sctype in [np.int32, np.double, np.complex_, np.string_,
np.ndarray]:
            ...     print(np.sctype2char(sctype))
1 # may vary
d
D
S
O
>>> x = np.array([1., 2-1.j])
>>> np.sctype2char(x)
'D'
>>> np.sctype2char(list)
'O'
"""

        sctype = obj2sctype(sctype)
        if sctype is None:
            raise ValueError("unrecognized type")
        if sctype not in _concrete_types:
            raise KeyError(sctype)
        return dtype(sctype).char
    cast = _typedict()
    for key in _concrete_types:
        cast[key] = lambda x, k=key: array(x, copy=False).astype(k)
    def __scalar_type_key(typ):
        """A ``key`` function for `sorted`."""
        dt = dtype(typ)
        return (dt.kind.lower(), dt.itemsize)

```

```

406: (0) ScalarType = [int, float, complex, bool, bytes, str, memoryview]
407: (0) ScalarType += sorted(_concrete_types, key=_scalar_type_key)
408: (0) ScalarType = tuple(ScalarType)
409: (0) for key in allTypes:
410: (4)     globals()[key] = allTypes[key]
411: (4)     __all__.append(key)
412: (0) del key
413: (0) typecodes = {'Character':'c',
414: (13)         'Integer':'bhilqp',
415: (13)         'UnsignedInteger':'BHILQP',
416: (13)         'Float':'efdg',
417: (13)         'Complex':'FDG',
418: (13)         'AllInteger':'bBhHiIlLqQpP',
419: (13)         'AllFloat':'efdgFDG',
420: (13)         'Datetime': 'Mm',
421: (13)         'All':'?bhilqpBHILQPefdgFDGSUVOMm'}
422: (0) typeDict = sctypeDict
423: (0) _kind_list = ['b', 'u', 'i', 'f', 'c', 'S', 'U', 'V', 'O', 'M', 'm']
424: (0) __test_types = '?'+typecodes['AllInteger'][:-2]+typecodes['AllFloat']+ '0'
425: (0) __len_test_types = len(__test_types)
426: (0) def _find_common_coerce(a, b):
427: (4)     if a > b:
428: (8)         return a
429: (4)     try:
430: (8)         thisind = __test_types.index(a.char)
431: (4)     except ValueError:
432: (8)         return None
433: (4)     return _can_coerce_all([a, b], start=thisind)
434: (0) def _can_coerce_all(dtypelist, start=0):
435: (4)     N = len(dtypelist)
436: (4)     if N == 0:
437: (8)         return None
438: (4)     if N == 1:
439: (8)         return dtypelist[0]
440: (4)     thisind = start
441: (4)     while thisind < __len_test_types:
442: (8)         newdtype = dtype(__test_types[thisind])
443: (8)         numcoerce = len([x for x in dtypelist if newdtype >= x])
444: (8)         if numcoerce == N:
445: (12)             return newdtype
446: (8)         thisind += 1
447: (4)     return None
448: (0) def _register_types():
449: (4)     numbers.Integral.register(integer)
450: (4)     numbers.Complex.register(inexact)
451: (4)     numbers.Real.register(floating)
452: (4)     numbers.Number.register(number)
453: (0) _register_types()
454: (0) @set_module('numpy')
455: (0) def find_common_type(array_types, scalar_types):
456: (4)     """
457: (4)     Determine common type following standard coercion rules.
458: (4)     .. deprecated:: NumPy 1.25
459: (8)         This function is deprecated, use `numpy.promote_types` or
460: (8)         `numpy.result_type` instead. To achieve semantics for the
461: (8)         `scalar_types` argument, use `numpy.result_type` and pass the Python
462: (8)         values `0`, `0.0`, or `0j`.
463: (8)         This will give the same results in almost all cases.
464: (8)         More information and rare exception can be found in the
465: (8)         `NumPy 1.25 release notes <https://numpy.org/devdocs/release/1.25.0-notes.html>`_.
466: (4)     Parameters
467: (4)     -----
468: (4)     array_types : sequence
469: (8)         A list of dtypes or dtype convertible objects representing arrays.
470: (4)     scalar_types : sequence
471: (8)         A list of dtypes or dtype convertible objects representing scalars.
472: (4)     Returns
473: (4)     -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

474: (4)           datatype : dtype
475: (8)             The common data type, which is the maximum of `array_types` ignoring
476: (8)               `scalar_types`, unless the maximum of `scalar_types` is of a
477: (8)                 different kind (`dtype.kind`). If the kind is not understood, then
478: (8)                   None is returned.
479: (4)           See Also
480: (4)             -----
481: (4)               dtype, common_type, can_cast, mintypecode
482: (4)           Examples
483: (4)             -----
484: (4)               >>> np.find_common_type([], [np.int64, np.float32, complex])
485: (4)               dtype('complex128')
486: (4)               >>> np.find_common_type([np.int64, np.float32], [])
487: (4)               dtype('float64')
488: (4)               The standard casting rules ensure that a scalar cannot up-cast an
489: (4)                 array unless the scalar is of a fundamentally different kind of data
490: (4)                   (i.e. under a different hierarchy in the data type hierarchy) then
491: (4)                     the array:
492: (4)               >>> np.find_common_type([np.float32], [np.int64, np.float64])
493: (4)               dtype('float32')
494: (4)               Complex is of a different type, so it up-casts the float in the
495: (4)                 `array_types` argument:
496: (4)               >>> np.find_common_type([np.float32], [complex])
497: (4)               dtype('complex128')
498: (4)               Type specifier strings are convertible to dtypes and can therefore
499: (4)                 be used instead of dtypes:
500: (4)               >>> np.find_common_type(['f4', 'f4', 'i4'], ['c8'])
501: (4)               dtype('complex128')
502: (4)               """
503: (4)               warnings.warn(
504: (12)                 "np.find_common_type is deprecated. Please use `np.result_type` "
505: (12)                 "or `np.promote_types`.\\n"
506: (12)                 "See https://numpy.org/devdocs/release/1.25.0-notes.html and the "
507: (12)                 "docs for more information. (Deprecated NumPy 1.25)",
508: (12)                 DeprecationWarning, stacklevel=2)
509: (4)               array_types = [dtype(x) for x in array_types]
510: (4)               scalar_types = [dtype(x) for x in scalar_types]
511: (4)               maxa = _can_coerce_all(array_types)
512: (4)               maxsc = _can_coerce_all(scalar_types)
513: (4)               if maxa is None:
514: (8)                 return maxsc
515: (4)               if maxsc is None:
516: (8)                 return maxa
517: (4)               try:
518: (8)                 index_a = _kind_list.index(maxa.kind)
519: (8)                 index_sc = _kind_list.index(maxsc.kind)
520: (4)               except ValueError:
521: (8)                 return None
522: (4)               if index_sc > index_a:
523: (8)                 return _find_common_coerce(maxsc, maxa)
524: (4)               else:
525: (8)                 return maxa

```

-----

## File 55 - overrides.py:

```

1: (0)           """Implementation of __array_function__ overrides from NEP-18."""
2: (0)           import collections
3: (0)           import functools
4: (0)           import os
5: (0)           from .._utils import set_module
6: (0)           from .._utils._inspect import getargspec
7: (0)           from numpy.core._multiarray_umath import (
8: (4)             add_docstring, _get_implementing_args, _ArrayFunctionDispatcher)
9: (0)           ARRAY_FUNCTIONS = set()
10: (0)          array_function_like_doc = (
11: (4)            """like : array_like, optional
12: (8)              Reference object to allow the creation of arrays which are not

```

```

13: (8)             NumPy arrays. If an array-like passed in as ``like`` supports
14: (8)             the ``__array_function__`` protocol, the result will be defined
15: (8)             by it. In this case, it ensures the creation of an array object
16: (8)             compatible with that passed in via this argument."""
17: (0)
18: (0)         )
19: (4)     def set_array_function_like_doc(public_api):
20: (8)         if public_api.__doc__ is not None:
21: (12)             public_api.__doc__ = public_api.__doc__.replace(
22: (12)                 "${ARRAY_FUNCTION_LIKE}",
23: (8)                 array_function_like_doc,
24: (4)             )
25: (0)         return public_api
26: (4)     add_docstring(
27: (4)         "``ArrayFunctionDispatcher``,
28: (4)         """
29: (4)             Class to wrap functions with checks for __array_function__ overrides.
30: (4)             All arguments are required, and can only be passed by position.
31: (4)             Parameters
32: (4)             -----
33: (4)             dispatcher : function or None
34: (8)                 The dispatcher function that returns a single sequence-like object
35: (8)                 of all arguments relevant. It must have the same signature (except
36: (8)                 the default values) as the actual implementation.
37: (8)                 If ``None``, this is a ``like`` dispatcher and the
38: (8)                 ``_ArrayFunctionDispatcher`` must be called with ``like`` as the
39: (8)                 first (additional and positional) argument.
40: (4)             implementation : function
41: (8)                 Function that implements the operation on NumPy arrays without
42: (8)                 overrides. Arguments passed calling the ``_ArrayFunctionDispatcher``
43: (8)                 will be forwarded to this (and the ``dispatcher``) as if using
44: (8)                 ``*args, **kwargs``.
45: (4)             Attributes
46: (4)             -----
47: (4)             _implementation : function
48: (8)                 The original implementation passed in.
49: (0)         add_docstring(
50: (4)             "``get_implementing_args``,
51: (4)             """
52: (4)             Collect arguments on which to call __array_function__.
53: (4)             Parameters
54: (4)             -----
55: (4)             relevant_args : iterable of array-like
56: (8)                 Iterable of possibly array-like arguments to check for
57: (8)                 __array_function__ methods.
58: (4)             Returns
59: (4)             -----
60: (4)             Sequence of arguments with __array_function__ methods, in the order in
61: (4)             which they should be called.
62: (4)             """
63: (0)         ArgSpec = collections.namedtuple('ArgSpec', 'args varargs keywords defaults')
64: (0)     def verify_matching_signatures(implementation, dispatcher):
65: (4)         """Verify that a dispatcher function has the right signature."""
66: (4)         implementation_spec = ArgSpec(*getargspec(implementation))
67: (4)         dispatcher_spec = ArgSpec(*getargspec(dispatcher))
68: (4)         if (implementation_spec.args != dispatcher_spec.args or
69: (12)             implementation_spec.varargs != dispatcher_spec.varargs or
70: (12)             implementation_spec.keywords != dispatcher_spec.keywords or
71: (12)             (bool(implementation_spec.defaults) !=
72: (13)                 bool(dispatcher_spec.defaults)) or
73: (12)             (implementation_spec.defaults is not None and
74: (13)                 len(implementation_spec.defaults) !=
75: (13)                 len(dispatcher_spec.defaults))):
76: (8)             raise RuntimeError('implementation and dispatcher for %s have '
77: (27)                 'different function signatures' % implementation)
78: (4)         if implementation_spec.defaults is not None:
79: (8)             if dispatcher_spec.defaults != (None,) *
len(dispatcher_spec.defaults):
80: (12)             raise RuntimeError('dispatcher functions can only use None for '

```

```

81: (31)                                'default argument values')
82: (0)                                 def array_function_dispatch(dispatcher=None, module=None, verify=True,
83: (28)   docs_from_dispatcher=False):
84: (4)   """Decorator for adding dispatch with the __array_function__ protocol.
85: (4)   See NEP-18 for example usage.
86: (4)   Parameters
87: (4)   -----
88: (4)   dispatcher : callable or None
89: (8)   Function that when called like ``dispatcher(*args, **kwargs)`` with
90: (8)   arguments from the NumPy function call returns an iterable of
91: (8)   array-like arguments to check for ``__array_function__``.
92: (8)   If `None`, the first argument is used as the single `like=` argument
93: (8)   and not passed on. A function implementing `like=` must call its
94: (8)   dispatcher with `like` as the first non-keyword argument.
95: (4)   module : str, optional
96: (8)   __module__ attribute to set on new function, e.g., ``module='numpy'``.
97: (8)   By default, module is copied from the decorated function.
98: (4)   verify : bool, optional
99: (8)   If True, verify the that the signature of the dispatcher and decorated
100: (8)  function signatures match exactly: all required and optional arguments
101: (8)  should appear in order with the same names, but the default values for
102: (8)  all optional arguments should be ``None``. Only disable verification
103: (8)  if the dispatcher's signature needs to deviate for some particular
104: (8)  reason, e.g., because the function has a signature like
105: (8)   ``func(*args, **kwargs)``.
106: (4)   docs_from_dispatcher : bool, optional
107: (8)   If True, copy docs from the dispatcher function onto the dispatched
108: (8)  function, rather than from the implementation. This is useful for
109: (8)  functions defined in C, which otherwise don't have docstrings.
110: (4)   Returns
111: (4)   -----
112: (4)   Function suitable for decorating the implementation of a NumPy function.
113: (4)   """
114: (4)   def decorator(implementation):
115: (8)   if verify:
116: (12)   if dispatcher is not None:
117: (16)   verify_matching_signatures(implementation, dispatcher)
118: (12)   else:
119: (16)   co = implementation.__code__
120: (16)   last_arg = co.co_argcount + co.co_kwonlyargcount - 1
121: (16)   last_arg = co.co_varnames[last_arg]
122: (16)   if last_arg != "like" or co.co_kwonlyargcount == 0:
123: (20)   raise RuntimeError(
124: (24)   "__array_function__ expects `like=` to be the last "
125: (24)   "argument and a keyword-only argument. "
126: (24)   f"{implementation} does not seem to comply.")
127: (8)   if docs_from_dispatcher:
128: (12)   add_docstring(implementation, dispatcher.__doc__)
129: (8)   public_api = _ArrayFunctionDispatcher(dispatcher, implementation)
130: (8)   public_api = functools.wraps(implementation)(public_api)
131: (8)   if module is not None:
132: (12)   public_api.__module__ = module
133: (8)   ARRAY_FUNCTIONS.add(public_api)
134: (8)   return public_api
135: (4)   return decorator
136: (0)   def array_function_from_dispatcher(
137: (8)   implementation, module=None, verify=True, docs_from_dispatcher=True):
138: (4)   """Like array_function_dispatch, but with function arguments flipped."""
139: (4)   def decorator(dispatcher):
140: (8)   return array_function_dispatch(
141: (12)   dispatcher, module, verify=verify,
142: (12)   docs_from_dispatcher=docs_from_dispatcher)(implementation)
143: (4)   return decorator

```

-----  
File 56 - records.py:

```
1: (0)      """

```

```

2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (0)
8: (2)
  np.float64)])
9: (2)
10: (2)
11: (0)
12: (0)
13: (0)
14: (0)
15: (2)
16: (2)
17: (2)
18: (2)
19: (0)
20: (2)
21: (2)
22: (2)
23: (2)
24: (2)
25: (0)
26: (0)
27: (0)
28: (0)
29: (0)
30: (0)
31: (0)
32: (0)
33: (0)
34: (0)
35: (4)
36: (4)
37: (0)
38: (0)
39: (0)
40: (18)
41: (18)
42: (18)
43: (18)
44: (18)
45: (18)
46: (18)
47: (18)
48: (18)
49: (18)
50: (18)
51: (18)
52: (18)
53: (0)
54: (0)
55: (4)
56: (4)
57: (8)
58: (8)
59: (8)
60: (4)
61: (0)
62: (0)
63: (4)
64: (4)
65: (4)
66: (4)
67: (4)
68: (4)
69: (4)

Record Arrays
=====
Record arrays expose the fields of structured arrays as properties.
Most commonly, ndarrays contain elements of a single type, e.g. floats,
integers, bools etc. However, it is possible for elements to be combinations
of these using structured types, such as::
    >>> a = np.array([(1, 2.0), (1, 2.0)], dtype=[('x', np.int64), ('y',
np.float64)])
    >>> a
    array([(1, 2.), (1, 2.)], dtype=[('x', '<i8'), ('y', '<f8')])
Here, each element consists of two fields: x (and int), and y (a float).
This is known as a structured array. The different fields are analogous
to columns in a spread-sheet. The different fields can be accessed as
one would a dictionary::
    >>> a['x']
    array([1, 1])
    >>> a['y']
    array([2., 2.])
Record arrays allow us to access fields as properties::
    >>> ar = np.rec.array(a)
    >>> ar.x
    array([1, 1])
    >>> ar.y
    array([2., 2.])
"""

import warnings
from collections import Counter
from contextlib import nullcontext
from .._utils import set_module
from . import numeric as sb
from . import numerictypes as nt
from numpy.compat import os_fspath
from .arrayprint import _get_legacy_print_mode
__all__ = [
    'record', 'recarray', 'format_parser',
    'fromarrays', 'fromrecords', 'fromstring', 'fromfile', 'array',
]
ndarray = sb.ndarray
_byteorderconv = {'b': '>',
                  'l': '<',
                  'n': '=',
                  'B': '>',
                  'L': '<',
                  'N': '=',
                  'S': 's',
                  's': 's',
                  '>': '>',
                  '<': '<',
                  '=': '=',
                  '|': '|',
                  'I': '|',
                  'i': '|'}
numfmt = nt.sctypeDict
def find_duplicate(list):
    """Find duplication in a list, return a list of duplicated elements"""
    return [
        item
        for item, counts in Counter(list).items()
        if counts > 1
    ]
@set_module('numpy')
class format_parser:
    """
    Class to convert formats, names, titles description to a dtype.
    After constructing the format_parser object, the dtype attribute is
    the converted data-type:
    ``dtype = format_parser(formats, names, titles).dtype``
    Attributes
    -----
    """

    Class to convert formats, names, titles description to a dtype.
    After constructing the format_parser object, the dtype attribute is
    the converted data-type:
    ``dtype = format_parser(formats, names, titles).dtype``
    Attributes
    -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

70: (4)          dtype : dtype
71: (8)          The converted data-type.
72: (4)          Parameters
73: (4)
74: (4)          formats : str or list of str
75: (8)          The format description, either specified as a string with
76: (8)          comma-separated format descriptions in the form ``'f8, i4, a5'``, or
77: (8)          a list of format description strings in the form
78: (8)          ``['f8', 'i4', 'a5']``.
79: (4)          names : str or list/tuple of str
80: (8)          The field names, either specified as a comma-separated string in the
81: (8)          form ``'col1, col2, col3'``, or as a list or tuple of strings in the
82: (8)          form ``['col1', 'col2', 'col3']``.
83: (8)          An empty list can be used, in that case default field names
84: (8)          ('f0', 'f1', ...) are used.
85: (4)          titles : sequence
86: (8)          Sequence of title strings. An empty list can be used to leave titles
87: (8)          out.
88: (4)          aligned : bool, optional
89: (8)          If True, align the fields by padding as the C-compiler would.
90: (8)          Default is False.
91: (4)          byteorder : str, optional
92: (8)          If specified, all the fields will be changed to the
93: (8)          provided byte-order. Otherwise, the default byte-order is
94: (8)          used. For all available string specifiers, see `dtype.newbyteorder`.
95: (4)          See Also
96: (4)
97: (4)          dtype, typename, sctype2char
98: (4)          Examples
99: (4)
100: (4)          >>> np.format_parser(['<f8', '<i4', '<a5'], ['col1', 'col2', 'col3'],
101: (4)                      ...                           ['T1', 'T2', 'T3']).dtype
102: (4)          dtype([('T1', 'col1'), ('<f8'), ('T2', 'col2'), ('<i4'),
103: (4)                      ('T3', 'col3'), ('S5')])]
104: (4)          `names` and/or `titles` can be empty lists. If `titles` is an empty list,
105: (4)          titles will simply not appear. If `names` is empty, default field names
106: (4)          will be used.
107: (4)          >>> np.format_parser(['f8', 'i4', 'a5'], ['col1', 'col2', 'col3'],
108: (4)                      ...                           []]).dtype
109: (4)          dtype([('col1', '<f8'), ('col2', '<i4'), ('col3', '<S5')])
110: (4)          >>> np.format_parser(['<f8', '<i4', '<a5'], [], []).dtype
111: (4)          dtype([('f0', '<f8'), ('f1', '<i4'), ('f2', 'S5')])
112: (4)          """
113: (8)          def __init__(self, formats, names, titles, aligned=False, byteorder=None):
114: (8)              self._parseFormats(formats, aligned)
115: (8)              self._setfieldnames(names, titles)
116: (8)              self._createdtype(byteorder)
117: (8)          def _parseFormats(self, formats, aligned=False):
118: (8)              """ Parse the field formats """
119: (12)              if formats is None:
120: (8)                  raise ValueError("Need formats argument")
121: (12)              if isinstance(formats, list):
122: (16)                  dtype = sb.dtype(
123: (16)                      [('{f}'.format(i), format_) for i, format_ in
124: (12)                          enumerate(formats)],
125: (8)                          aligned,
126: (12)                          )
127: (8)              else:
128: (8)                  dtype = sb.dtype(formats, aligned)
129: (12)                  fields = dtype.fields
130: (12)                  if fields is None:
131: (8)                      dtype = sb.dtype([('f1', dtype)], aligned)
132: (8)                      fields = dtype.fields
133: (8)                  keys = dtype.names
134: (8)                  self._formats = [fields[key][0] for key in keys]
135: (4)                  self._offsets = [fields[key][1] for key in keys]
136: (8)                  self._nfields = len(keys)
137: (4)          def _setfieldnames(self, names, titles):
138: (8)              """Convert input field names into a list and assign to the _names

```

```

137: (8)             attribute """
138: (8)             if names:
139: (12)                 if type(names) in [list, tuple]:
140: (16)                     pass
141: (12)                 elif isinstance(names, str):
142: (16)                     names = names.split(',')
143: (12)                 else:
144: (16)                     raise ValueError("illegal input names %s" % repr(names))
145: (12)                 self._names = [n.strip() for n in names[:self._nfields]]
146: (8)             else:
147: (12)                 self._names = []
148: (8)             self._names += ['f%dd' % i for i in range(len(self._names),
149: (49)                                     self._nfields)]
150: (8)             _dup = find_duplicate(self._names)
151: (8)             if _dup:
152: (12)                 raise ValueError("Duplicate field names: %s" % _dup)
153: (8)             if titles:
154: (12)                 self._titles = [n.strip() for n in titles[:self._nfields]]
155: (8)             else:
156: (12)                 self._titles = []
157: (12)                 titles = []
158: (8)             if self._nfields > len(titles):
159: (12)                 self._titles += [None] * (self._nfields - len(titles))
160: (4)             def __createdtype(self, byteorder):
161: (8)                 dtype = np.dtype({
162: (12)                     'names': self._names,
163: (12)                     'formats': self._f_formats,
164: (12)                     'offsets': self._offsets,
165: (12)                     'titles': self._titles,
166: (8)                 })
167: (8)                 if byteorder is not None:
168: (12)                     byteorder = _byteorderconv[byteorder[0]]
169: (12)                     dtype = dtype.newbyteorder(byteorder)
170: (8)                 self.dtype = dtype
171: (0)             class record(nt.void):
172: (4)                 """A data-type scalar that allows field access as attribute lookup.
173: (4)                 """
174: (4)                 __name__ = 'record'
175: (4)                 __module__ = 'numpy'
176: (4)                 def __repr__(self):
177: (8)                     if __get_legacy_print_mode() <= 113:
178: (12)                         return self.__str__()
179: (8)                     return super().__repr__()
180: (4)                 def __str__(self):
181: (8)                     if __get_legacy_print_mode() <= 113:
182: (12)                         return str(self.item())
183: (8)                     return super().__str__()
184: (4)                 def __getattribute__(self, attr):
185: (8)                     if attr in ('setfield', 'getfield', 'dtype'):
186: (12)                         return nt.void.__getattribute__(self, attr)
187: (8)                     try:
188: (12)                         return nt.void.__getattribute__(self, attr)
189: (8)                     except AttributeError:
190: (12)                         pass
191: (8)                     fielddict = nt.void.__getattribute__(self, 'dtype').fields
192: (8)                     res = fielddict.get(attr, None)
193: (8)                     if res:
194: (12)                         obj = self.getfield(*res[:2])
195: (12)                         try:
196: (16)                             dt = obj.dtype
197: (12)                         except AttributeError:
198: (16)                             return obj
199: (12)                         if dt.names is not None:
200: (16)                             return obj.view((self.__class__, obj.dtype))
201: (12)                         return obj
202: (8)                     else:
203: (12)                         raise AttributeError("'record' object has no "
204: (20)                               "attribute '%s'" % attr)
205: (4)             def __setattr__(self, attr, val):

```

```

206: (8)             if attr in ('setfield', 'getfield', 'dtype'):
207: (12)             raise AttributeError("Cannot set '%s' attribute" % attr)
208: (8)             fielddict = nt.void.__getattribute__(self, 'dtype').fields
209: (8)             res = fielddict.get(attr, None)
210: (8)             if res:
211: (12)                 return self.setfield(val, *res[:2])
212: (8)             else:
213: (12)                 if getattr(self, attr, None):
214: (16)                     return nt.void.__setattr__(self, attr, val)
215: (12)                 else:
216: (16)                     raise AttributeError("'record' object has no "
217: (24)                         "attribute '%s'" % attr)
218: (4)             def __getitem__(self, indx):
219: (8)                 obj = nt.void.__getitem__(self, indx)
220: (8)                 if isinstance(obj, nt.void) and obj.dtype.names is not None:
221: (12)                     return obj.view((self.__class__, obj.dtype))
222: (8)                 else:
223: (12)                     return obj
224: (4)             def pprint(self):
225: (8)                 """Pretty-print all fields."""
226: (8)                 names = self.dtype.names
227: (8)                 maxlen = max(len(name) for name in names)
228: (8)                 fmt = '%# %ds: %s' % maxlen
229: (8)                 rows = [fmt % (name, getattr(self, name)) for name in names]
230: (8)                 return "\n".join(rows)
231: (0)             class recarray(ndarray):
232: (4)                 """Construct an ndarray that allows field access using attributes.
233: (4)                 Arrays may have a data-types containing fields, analogous
234: (4)                 to columns in a spread sheet. An example is ``[(x, int), (y, float)]``,
235: (4)                 where each entry in the array is a pair of ``(int, float)``. Normally,
236: (4)                 these attributes are accessed using dictionary lookups such as
237: (4)                 ``arr['x']``.
238: (4)             members
239: (4)             Parameters
240: (4)             -----
241: (4)             shape : tuple
242: (8)                 Shape of output array.
243: (4)             dtype : data-type, optional
244: (8)                 The desired data-type. By default, the data-type is determined
245: (8)                 from `formats`, `names`, `titles`, `aligned` and `byteorder`.
246: (4)             formats : list of data-types, optional
247: (8)                 A list containing the data-types for the different columns, e.g.
248: (8)                 ``['i4', 'f8', 'i4']``. `formats` does *not* support the new
249: (8)                 convention of using types directly, i.e. ``(int, float, int)``.
250: (8)                 Note that `formats` must be a list, not a tuple.
251: (8)                 Given that `formats` is somewhat limited, we recommend specifying
252: (8)                 `dtype` instead.
253: (4)             names : tuple of str, optional
254: (8)                 The name of each column, e.g. ``('x', 'y', 'z')``.
255: (4)             buf : buffer, optional
256: (8)                 By default, a new array is created of the given shape and data-type.
257: (8)                 If `buf` is specified and is an object exposing the buffer interface,
258: (8)                 the array will use the memory from the existing buffer. In this case,
259: (8)                 the `offset` and `strides` keywords are available.
260: (4)             Other Parameters
261: (4)             -----
262: (4)             titles : tuple of str, optional
263: (8)                 Aliases for column names. For example, if `names` were
264: (8)                 ``('x', 'y', 'z')`` and `titles` is
265: (8)                 ``('x_coordinate', 'y_coordinate', 'z_coordinate')``, then
266: (8)                 ``arr['x']`` is equivalent to both ``arr.x`` and ``arr.x_coordinate``.
267: (4)             byteorder : {'<', '>', '='}, optional
268: (8)                 Byte-order for all fields.
269: (4)             aligned : bool, optional
270: (8)                 Align the fields in memory as the C-compiler would.
271: (4)             strides : tuple of ints, optional
272: (8)                 Buffer (`buf`) is interpreted according to these strides (strides

```

```

273: (8)           define how many bytes each array element, row, column, etc.
274: (8)           occupy in memory).
275: (4)           offset : int, optional
276: (8)           Start reading buffer (`buf`) from this offset onwards.
277: (4)           order : {'C', 'F'}, optional
278: (8)           Row-major (C-style) or column-major (Fortran-style) order.
279: (4)           Returns
280: (4)           -----
281: (4)           rec : recarray
282: (8)           Empty array of the given shape and type.
283: (4)           See Also
284: (4)           -----
285: (4)           core.records.fromrecords : Construct a record array from data.
286: (4)           record : fundamental data-type for `recarray`.
287: (4)           format_parser : determine a data-type from formats, names, titles.
288: (4)           Notes
289: (4)           -----
290: (4)           This constructor can be compared to ``empty``: it creates a new record
291: (4)           array but does not fill it with data. To create a record array from data,
292: (4)           use one of the following methods:
293: (4)           1. Create a standard ndarray and convert it to a record array,
294: (7)             using ``arr.view(np.recarray)``
295: (4)           2. Use the `buf` keyword.
296: (4)           3. Use `np.rec.fromrecords`.
297: (4)           Examples
298: (4)           -----
299: (4)           Create an array with two fields, ``x`` and ``y``:
300: (4)           >>> x = np.array([(1.0, 2), (3.0, 4)], dtype=[('x', '<f8'), ('y', '<i8')])
301: (4)           >>> x
302: (4)           array([(1., 2), (3., 4)], dtype=[('x', '<f8'), ('y', '<i8')])
303: (4)           >>> x['x']
304: (4)           array([1., 3.])
305: (4)           View the array as a record array:
306: (4)           >>> x = x.view(np.recarray)
307: (4)           >>> x.x
308: (4)           array([1., 3.])
309: (4)           >>> x.y
310: (4)           array([2, 4])
311: (4)           Create a new, empty record array:
312: (4)           >>> np.recarray((2,),
313: (4)             ... dtype=[('x', int), ('y', float), ('z', int)]) #doctest: +SKIP
314: (4)           rec.array([-1073741821, 1.2249118382103472e-301, 24547520),
315: (11)             (3471280, 1.2134086255804012e-316, 0)],
316: (10)             dtype=[('x', '<i4'), ('y', '<f8'), ('z', '<i4')])
317: (4)             """
318: (4)             __name__ = 'recarray'
319: (4)             __module__ = 'numpy'
320: (4)             def __new__(subtype, shape, dtype=None, buf=None, offset=0, strides=None,
321: (16)               formats=None, names=None, titles=None,
322: (16)               byteorder=None, aligned=False, order='C'):
323: (8)               if dtype is not None:
324: (12)                 descr = sb.dtype(dtype)
325: (8)               else:
326: (12)                 descr = format_parser(formats, names, titles, aligned,
327: (8)                   if buf is None:
328: (12)                     self = ndarray.__new__(subtype, shape, (record, descr),
329: (8)                   else:
330: (12)                     self = ndarray.__new__(subtype, shape, (record, descr),
331: (38)                         buffer=buf, offset=offset,
332: (38)                         strides=strides, order=order)
333: (8)                     return self
334: (4)             def __array_finalize__(self, obj):
335: (8)               if self.dtype.type is not record and self.dtype.names is not None:
336: (12)                 self.dtype = self.dtype
337: (4)             def __getattribute__(self, attr):
338: (8)               try:
339: (12)                 return object.__getattribute__(self, attr)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

340: (8)
341: (12)
342: (8)
343: (8)
344: (12)
345: (8)
346: (12)
347: (8)
348: (8)
349: (12)
350: (16)
351: (12)
352: (8)
353: (12)
354: (4)
355: (8)
not None:
356: (12)
357: (8)
358: (8)
359: (12)
360: (8)
361: (12)
362: (12)
363: (16)
364: (8)
365: (12)
366: (12)
367: (16)
368: (12)
369: (16)
370: (20)
371: (16)
372: (20)
373: (8)
374: (12)
375: (8)
376: (12)
377: (16)
378: (12)
379: (8)
380: (4)
381: (8)
382: (8)
383: (12)
384: (16)
385: (16)
386: (20)
387: (16)
388: (12)
389: (16)
390: (8)
391: (12)
392: (4)
393: (8)
394: (8)
nt_void):
395: (12)
396: (16)
397: (12)
398: (12)
399: (8)
400: (12)
401: (12)
402: (8)
403: (12)
404: (16)
405: (8)
406: (12)

except AttributeError: # attr must be a fieldname
    pass
fielddict = ndarray.__getattribute__(self, 'dtype').fields
try:
    res = fielddict[attr][:2]
except (TypeError, KeyError) as e:
    raise AttributeError("recarray has no attribute %s" % attr) from e
obj = self.getfield(*res)
if obj.dtype.names is not None:
    if issubclass(obj.dtype.type, nt_void):
        return obj.view(dtype=(self.dtype.type, obj.dtype))
    return obj
else:
    return obj.view(ndarray)
def __setattr__(self, attr, val):
    if attr == 'dtype' and issubclass(val.type, nt_void) and val.names is
        val = sb.dtype((record, val))
newattr = attr not in self.__dict__
try:
    ret = object.__setattr__(self, attr, val)
except Exception:
    fielddict = ndarray.__getattribute__(self, 'dtype').fields or {}
    if attr not in fielddict:
        raise
else:
    fielddict = ndarray.__getattribute__(self, 'dtype').fields or {}
    if attr not in fielddict:
        return ret
    if newattr:
        try:
            object.__delattr__(self, attr)
        except Exception:
            return ret
    try:
        res = fielddict[attr][:2]
    except (TypeError, KeyError) as e:
        raise AttributeError(
            "record array has no attribute %s" % attr
        ) from e
    return self.setfield(val, *res)
def __getitem__(self, indx):
    obj = super().__getitem__(indx)
    if isinstance(obj, ndarray):
        if obj.dtype.names is not None:
            obj = obj.view(type(self))
            if issubclass(obj.dtype.type, nt_void):
                return obj.view(dtype=(self.dtype.type, obj.dtype))
            return obj
        else:
            return obj.view(type=ndarray)
    else:
        return obj
def __repr__(self):
    repr_dtype = self.dtype
    if self.dtype.type is record or not issubclass(self.dtype.type,
  nt_void):
        if repr_dtype.type is record:
            repr_dtype = sb.dtype((nt_void, repr_dtype))
    prefix = "rec.array("
    fmt = 'rec.array(%s,%sdtype=%s)'
    else:
        prefix = "array("
        fmt = 'array(%s,%sdtype=%s).view(numpy.recarray)'
    if self.size > 0 or self.shape == (0,):
        lst = sb.array2string(
            self, separator=', ', prefix=prefix, suffix=',' )
    else:
        lst = "[]", shape=%s" % (repr(self.shape),)

```

```

407: (8)           lf = '\n' + '*len(prefix)
408: (8)           if _get_legacy_print_mode() <= 113:
409: (12)             lf = ' ' + lf # trailing space
410: (8)           return fmt % (lst, lf, repr_dtype)
411: (4)           def field(self, attr, val=None):
412: (8)             if isinstance(attr, int):
413: (12)               names = ndarray.__getattribute__(self, 'dtype').names
414: (12)               attr = names[attr]
415: (8)             fielddict = ndarray.__getattribute__(self, 'dtype').fields
416: (8)             res = fielddict[attr][:2]
417: (8)             if val is None:
418: (12)               obj = self.getfield(*res)
419: (12)               if obj.dtype.names is not None:
420: (16)                 return obj
421: (12)               return obj.view(ndarray)
422: (8)             else:
423: (12)               return self.setfield(val, *res)
424: (0)           def _deprecate_shape_0_as_None(shape):
425: (4)             if shape == 0:
426: (8)               warnings.warn(
427: (12)                 "Passing `shape=0` to have the shape be inferred is deprecated, "
428: (12)                 "and in future will be equivalent to `shape=(0,)`. To infer "
429: (12)                 "the shape and suppress this warning, pass `shape=None` instead.", 
430: (12)                 FutureWarning, stacklevel=3)
431: (8)               return None
432: (4)             else:
433: (8)               return shape
434: (0)           @set_module("numpy.rec")
435: (0)           def fromarrays(arrayList, dtype=None, shape=None, formats=None,
436: (15)             names=None, titles=None, aligned=False, byteorder=None):
437: (4)             """Create a record array from a (flat) list of arrays
438: (4)             Parameters
439: (4)             -----
440: (4)             arrayList : list or tuple
441: (8)               List of array-like objects (such as lists, tuples,
442: (8)                 and ndarrays).
443: (4)             dtype : data-type, optional
444: (8)               valid dtype for all arrays
445: (4)             shape : int or tuple of ints, optional
446: (8)               Shape of the resulting array. If not provided, inferred from
447: (8)                 ``arrayList[0]``.
448: (4)             formats, names, titles, aligned, byteorder :
449: (8)               If `dtype` is ``None``, these arguments are passed to
450: (8)                 `numpy.format_parser` to construct a dtype. See that function for
451: (8)                 detailed documentation.
452: (4)             Returns
453: (4)             -----
454: (4)             np.recarray
455: (8)               Record array consisting of given arrayList columns.
456: (4)             Examples
457: (4)             -----
458: (4)             >>> x1=np.array([1,2,3,4])
459: (4)             >>> x2=np.array(['a','dd','xyz','12'])
460: (4)             >>> x3=np.array([1.1,2,3,4])
461: (4)             >>> r = np.core.records.fromarrays([x1,x2,x3],names='a,b,c')
462: (4)             >>> print(r[1])
463: (4)               (2, 'dd', 2.0) # may vary
464: (4)             >>> x1[1]=34
465: (4)             >>> r.a
466: (4)               array([1, 2, 3, 4])
467: (4)             >>> x1 = np.array([1, 2, 3, 4])
468: (4)             >>> x2 = np.array(['a', 'dd', 'xyz', '12'])
469: (4)             >>> x3 = np.array([1.1, 2, 3, 4])
470: (4)             >>> r = np.core.records.fromarrays(
471: (4)               ...      [x1, x2, x3],
472: (4)               ...      dtype=np.dtype([('a', np.int32), ('b', 'S3'), ('c', np.float32)]))
473: (4)             >>> r
474: (4)               rec.array([(1, b'a', 1.1), (2, b'dd', 2. ), (3, b'xyz', 3. ),
475: (15)                 (4, b'12', 4. )]),

```

```

476: (14)                               dtype=[('a', '<i4'), ('b', 'S3'), ('c', '<f4')])"
477: (4)
478: (4)                                 """
479: (4)                                 arrayList = [sb.asarray(x) for x in arrayList]
480: (4)                                 shape = _deprecate_shape_0_as_None(shape)
481: (8)                                 if shape is None:
482: (4)                                     shape = arrayList[0].shape
483: (8)                                 elif isinstance(shape, int):
484: (4)                                     shape = (shape,)
485: (8)                                 if formats is None and dtype is None:
486: (4)                                     formats = [obj.dtype for obj in arrayList]
487: (8)                                 if dtype is not None:
488: (4)                                     descr = sb.dtype(dtype)
489: (8)                                 else:
490: (4)                                     descr = format_parser(formats, names, titles, aligned,
491: (4)   _names = descr.names
492: (8)   if len(descr) != len(arrayList):
493: (16)   raise ValueError("mismatch between the number of fields "
494: (4)   "and the number of arrays")
495: (4)   d0 = descr[0].shape
496: (4)   nn = len(d0)
497: (4)   if nn > 0:
498: (8)   shape = shape[:-nn]
499: (4)   _array = recarray(shape, descr)
500: (8)   for k, obj in enumerate(arrayList):
501: (8)   nn = descr[k].ndim
502: (8)   testshape = obj.shape[:obj.ndim - nn]
503: (8)   name = _names[k]
504: (12)   if testshape != shape:
505: (8)   raise ValueError(f'array-shape mismatch in array {k} ({name})')
506: (4)   _array[name] = obj
507: (0)   return _array
@set_module("numpy.rec")
508: (0) def fromrecords(recList, dtype=None, shape=None, formats=None, names=None,
509: (16)   titles=None, aligned=False, byteorder=None):
510: (4)     """Create a recarray from a list of records in text form.
511: (4)     Parameters
512: (4)     -----
513: (4)     recList : sequence
514: (8)         data in the same field may be heterogeneous - they will be promoted
515: (8)         to the highest data type.
516: (4)     dtype : data-type, optional
517: (8)         valid dtype for all arrays
518: (4)     shape : int or tuple of ints, optional
519: (8)         shape of each array.
520: (4)     formats, names, titles, aligned, byteorder :
521: (8)         If `dtype` is ``None``, these arguments are passed to
522: (8)         `numpy.format_parser` to construct a dtype. See that function for
523: (8)         detailed documentation.
524: (8)         If both `formats` and `dtype` are None, then this will auto-detect
525: (8)         formats. Use list of tuples rather than list of lists for faster
526: (8)         processing.
527: (4)     Returns
528: (4)     -----
529: (4)     np.recarray
530: (8)         record array consisting of given recList rows.
531: (4)     Examples
532: (4)     -----
533: (4)     >>> r=np.core.records.fromrecords([(456,'dbe',1.2),(2,'de',1.3)],
534: (4)         ... names='col1,col2,col3')
535: (4)     >>> print(r[0])
536: (4)     (456, 'dbe', 1.2)
537: (4)     >>> r.col1
538: (4)     array([456, 2])
539: (4)     >>> r.col2
540: (4)     array(['dbe', 'de'], dtype='<U3')
541: (4)     >>> import pickle
542: (4)     >>> pickle.loads(pickle.dumps(r))
543: (4)     rec.array([(456, 'dbe', 1.2), (2, 'de', 1.3)],
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

544: (14)           dtype=[('col1', '<i8'), ('col2', '<U3'), ('col3', '<f8')])"
545: (4)
546: (4)
547: (8)
548: (8)
range(obj.shape[-1])]"
549: (8)
550: (26)
551: (4)
552: (8)
553: (4)
554: (8)
byteorder).dtype
555: (4)
556: (8)
557: (4)
558: (8)
559: (8)
560: (12)
561: (8)
562: (12)
563: (8)
564: (12)
565: (8)
566: (8)
567: (12)
568: (8)
569: (12)
570: (12)
571: (12)
572: (8)
573: (4)
574: (8)
575: (12)
576: (4)
577: (4)
578: (0)
579: (0)
580: (15)
581: (4)
582: (4)
583: (4)
584: (4)
585: (4)
586: (4)
587: (8)
588: (4)
589: (8)
590: (4)
591: (8)
592: (4)
593: (8)
594: (4)
595: (8)
596: (8)
597: (8)
598: (4)
599: (4)
600: (4)
601: (8)
602: (8)
603: (4)
604: (4)
605: (4)
606: (4)
607: (4)
608: (4)
609: (4)
610: (4)

        """
        if formats is None and dtype is None: # slower
            obj = sb.array(recList, dtype=object)
            arrlist = [sb.array(obj[..., i].tolist()) for i in
                       range(obj.shape[-1])]

        return frommarrays(arrlist, formats=formats, shape=shape, names=names,
                            titles=titles, aligned=aligned, byteorder=byteorder)
    if dtype is not None:
        descr = sb.dtype((record, dtype))
    else:
        descr = format_parser(formats, names, titles, aligned,
                               byteorder).dtype

    try:
        retval = sb.array(recList, dtype=descr)
    except (TypeError, ValueError):
        shape = _deprecate_shape_0_as_None(shape)
        if shape is None:
            shape = len(recList)
        if isinstance(shape, int):
            shape = (shape,)
        if len(shape) > 1:
            raise ValueError("Can only deal with 1-d array.")
        _array = recarray(shape, descr)
        for k in range(_array.size):
            _array[k] = tuple(recList[k])
        warnings.warn(
            "fromrecords expected a list of tuples, may have received a list "
            "of lists instead. In the future that will raise an error",
            FutureWarning, stacklevel=2)
        return _array
    else:
        if shape is not None and retval.shape != shape:
            retval.shape = shape
    res = retval.view(recarray)
    return res

@set_module("numpy.rec")
def fromstring(datastring, dtype=None, shape=None, offset=0, formats=None,
               names=None, titles=None, aligned=False, byteorder=None):
    r"""Create a record array from binary data
    Note that despite the name of this function it does not accept `str` instances.

    Parameters
    -----
    datastring : bytes-like
        Buffer of binary data
    dtype : data-type, optional
        Valid dtype for all arrays
    shape : int or tuple of ints, optional
        Shape of each array.
    offset : int, optional
        Position in the buffer to start reading from.
    formats, names, titles, aligned, byteorder :
        If `dtype` is ``None``, these arguments are passed to
        `numpy.format_parser` to construct a dtype. See that function for
        detailed documentation.

    Returns
    -----
    np.recarray
        Record array view into the data in datastring. This will be readonly
        if `datastring` is readonly.

    See Also
    -----
    numpy.frombuffer

    Examples
    -----
    >>> a = b'\x01\x02\x03abc'
    >>> np.core.records.fromstring(a, dtype='u1,u1,u1,S3')
    rec.array([(1, 2, 3, b'abc')],
```

```

611: (12)          dtype=[('f0', 'u1'), ('f1', 'u1'), ('f2', 'u1'), ('f3', 'S3')])  

612: (4)          >>> grades_dtype = [('Name', (np.str_, 10)), ('Marks', np.float64),  

613: (4)          ...           ('GradeLevel', np.int32)]  

614: (4)          >>> grades_array = np.array([('Sam', 33.3, 3), ('Mike', 44.4, 5),  

615: (4)          ...           ('Aadi', 66.6, 6)], dtype=grades_dtype)  

616: (4)          >>> np.core.records.fromstring(grades_array.tobytes(), dtype=grades_dtype)  

617: (4)          rec.array([('Sam', 33.3, 3), ('Mike', 44.4, 5), ('Aadi', 66.6, 6)],  

618: (12)          dtype=[('Name', '<U10'), ('Marks', 'f8'), ('GradeLevel', 'i4')])  

619: (4)          >>> s = '\x01\x02\x03abc'  

620: (4)          >>> np.core.records.fromstring(s, dtype='u1,u1,u1,S3')  

621: (4)          Traceback (most recent call last)  

622: (7)          ...  

623: (4)          TypeError: a bytes-like object is required, not 'str'  

624: (4)  

625: (4)          if dtype is None and formats is None:  

626: (8)          raise TypeError("fromstring() needs a 'dtype' or 'formats' argument")  

627: (4)          if dtype is not None:  

628: (8)          descr = sb.dtype(dtype)  

629: (4)  

630: (8)          descr = format_parser(formats, names, titles, aligned,  

byteorder).dtype  

631: (4)          itemsize = descr.itemsize  

632: (4)          shape = _deprecate_shape_0_as_None(shape)  

633: (4)          if shape in (None, -1):  

634: (8)          shape = (len(datastring) - offset) // itemsize  

635: (4)          _array = recarray(shape, descr, buf=datastring, offset=offset)  

636: (4)          return _array  

637: (0)          def get_remaining_size(fd):  

638: (4)          pos = fd.tell()  

639: (4)          try:  

640: (8)          fd.seek(0, 2)  

641: (8)          return fd.tell() - pos  

642: (4)          finally:  

643: (8)          fd.seek(pos, 0)  

644: (0)          @set_module("numpy.rec")  

645: (0)          def fromfile(fd, dtype=None, shape=None, offset=0, formats=None,  

646: (13)          names=None, titles=None, aligned=False, byteorder=None):  

647: (4)          """Create an array from binary file data  

648: (4)          Parameters  

649: (4)  

650: (4)          fd : str or file type  

651: (8)          If file is a string or a path-like object then that file is opened,  

652: (8)          else it is assumed to be a file object. The file object must  

653: (8)          support random access (i.e. it must have tell and seek methods).  

654: (4)          dtype : data-type, optional  

655: (8)          valid dtype for all arrays  

656: (4)          shape : int or tuple of ints, optional  

657: (8)          shape of each array.  

658: (4)          offset : int, optional  

659: (8)          Position in the file to start reading from.  

660: (4)          formats, names, titles, aligned, byteorder :  

661: (8)          If `dtype` is ``None``, these arguments are passed to  

662: (8)          `numpy.format_parser` to construct a dtype. See that function for  

663: (8)          detailed documentation  

664: (4)          Returns  

665: (4)  

666: (4)          np.recarray  

667: (8)          record array consisting of data enclosed in file.  

668: (4)          Examples  

669: (4)  

670: (4)          >>> from tempfile import TemporaryFile  

671: (4)          >>> a = np.empty(10,dtype='f8,i4,a5')  

672: (4)          >>> a[5] = (0.5,10,'abcde')  

673: (4)  

674: (4)          >>> fd=TemporaryFile()  

675: (4)          >>> a = a.newbyteorder('<')  

676: (4)          >>> a.tofile(fd)  

677: (4)  

678: (4)          >>> _ = fd.seek(0)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

679: (4)             >>> r=np.core.records.fromfile(fd, formats='f8,i4,a5', shape=10,
680: (4)             ... byteorder='<')
681: (4)             >>> print(r[5])
682: (4)             (0.5, 10, 'abcde')
683: (4)             >>> r.shape
684: (4)             (10,)
685: (4)             """
686: (4)             if dtype is None and formats is None:
687: (8)                 raise TypeError("fromfile() needs a 'dtype' or 'formats' argument")
688: (4)             shape = _deprecate_shape_0_as_None(shape)
689: (4)             if shape is None:
690: (8)                 shape = (-1,)
691: (4)             elif isinstance(shape, int):
692: (8)                 shape = (shape,)
693: (4)             if hasattr(fd, 'readinto'):
694: (8)                 ctx = nullcontext(fd)
695: (4)             else:
696: (8)                 ctx = open(os_fspath(fd), 'rb')
697: (4)             with ctx as fd:
698: (8)                 if offset > 0:
699: (12)                     fd.seek(offset, 1)
700: (8)                 size = get_remaining_size(fd)
701: (8)                 if dtype is not None:
702: (12)                     descr = sb.dtype(dtype)
703: (8)                 else:
704: (12)                     descr = format_parser(formats, names, titles, aligned,
byteorder).dtype
705: (8)                     itemsize = descr.itemsize
706: (8)                     shapeprod = sb.array(shape).prod(dtype=nt.intp)
707: (8)                     shapesize = shapeprod * itemsize
708: (8)                     if shapesize < 0:
709: (12)                         shape = list(shape)
710: (12)                         shape[shape.index(-1)] = size // -shapesize
711: (12)                         shape = tuple(shape)
712: (12)                         shapeprod = sb.array(shape).prod(dtype=nt.intp)
713: (8)                     nbytes = shapeprod * itemsize
714: (8)                     if nbytes > size:
715: (12)                         raise ValueError(
716: (20)                             "Not enough bytes left in file for specified shape and
type")
717: (8)                         _array = recarray(shape, descr)
718: (8)                         nbytesread = fd.readinto(_array.data)
719: (8)                         if nbytesread != nbytes:
720: (12)                             raise OSError("Didn't read as many bytes as expected")
721: (4)                         return _array
722: (0) @set_module("numpy.rec")
723: (0) def array(obj, dtype=None, shape=None, offset=0, strides=None, formats=None,
724: (10)             names=None, titles=None, aligned=False, byteorder=None, copy=True):
725: (4)             """
726: (4)             Construct a record array from a wide-variety of objects.
727: (4)             A general-purpose record array constructor that dispatches to the
728: (4)             appropriate `recarray` creation function based on the inputs (see Notes).
729: (4)             Parameters
730: (4)             -----
731: (4)             obj : any
732: (8)                 Input object. See Notes for details on how various input types are
733: (8)                 treated.
734: (4)             dtype : data-type, optional
735: (8)                 Valid dtype for array.
736: (4)             shape : int or tuple of ints, optional
737: (8)                 Shape of each array.
738: (4)             offset : int, optional
739: (8)                 Position in the file or buffer to start reading from.
740: (4)             strides : tuple of ints, optional
741: (8)                 Buffer (`buf`) is interpreted according to these strides (strides
742: (8)                 define how many bytes each array element, row, column, etc.
743: (8)                 occupy in memory).
744: (4)             formats, names, titles, aligned, byteorder :
745: (8)                 If `dtype` is ``None``, these arguments are passed to

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

746: (8) `numpy.format_parser` to construct a dtype. See that function for
747: (8) detailed documentation.
748: (4) copy : bool, optional
749: (8)     Whether to copy the input object (True), or to use a reference
instead.
750: (8)     This option only applies when the input is an ndarray or recarray.
751: (8)     Defaults to True.
752: (4) Returns -----
753: (4)
754: (4) np.recarray
755: (8)     Record array created from the specified object.
756: (4) Notes -----
757: (4)
758: (4) If `obj` is ``None``, then call the `~numpy.recarray` constructor. If
759: (4) `obj` is a string, then call the `fromstring` constructor. If `obj` is a
760: (4) list or a tuple, then if the first object is an `~numpy.ndarray`, call
761: (4) `fromarrays`, otherwise call `fromrecords`. If `obj` is a
762: (4) `~numpy.recarray`, then make a copy of the data in the recarray
763: (4) (if ``copy=True``) and use the new formats, names, and titles. If `obj`
764: (4) is a file, then call `fromfile`. Finally, if obj is an `ndarray`, then
765: (4) return ``obj.view(recarray)``, making a copy of the data if ``copy=True``.
766: (4) Examples -----
767: (4)
768: (4) >>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
769: (4) array([[1, 2, 3],
770: (11)         [4, 5, 6],
771: (11)         [7, 8, 9]])
772: (4) >>> np.core.records.array(a)
773: (4) rec.array([[1, 2, 3],
774: (15)         [4, 5, 6],
775: (15)         [7, 8, 9]],
776: (8)         dtype=int32)
777: (4) >>> b = [(1, 1), (2, 4), (3, 9)]
778: (4) >>> c = np.core.records.array(b, formats = ['i2', 'f2'], names = ('x',
'y'))
779: (4)
780: (4)
781: (14) >>> c
782: (4) rec.array([(1, 1.0), (2, 4.0), (3, 9.0)],
783: (4)           dtype=[('x', '<i2'), ('y', '<f2')])
784: (4) >>> c.x
785: (4) rec.array([1, 2, 3], dtype=int16)
786: (4) >>> r = np.rec.array(['abc','def'], names=['col1','col2'])
787: (4) >>> print(r.col1)
788: (4) abc
789: (4) >>> r.col1
790: (4) array('abc', dtype='<U3')
791: (4) >>> r.col2
792: (4) array('def', dtype='<U3')
793: (4) """
794: (4) if ((isinstance(obj, (type(None), str)) or hasattr(obj, 'readinto')) and
795: (11)         formats is None and dtype is None):
796: (8)     raise ValueError("Must define formats (or dtype) if object is "
797: (25)             "None, string, or an open file")
798: (4) kwds = {}
799: (4) if dtype is not None:
800: (8)     dtype = sb.dtype(dtype)
801: (4) elif formats is not None:
802: (8)     dtype = format_parser(formats, names, titles,
803: (30)                     aligned, byteorder).dtype
804: (4) else:
805: (8)     kwds = {'formats': formats,
806: (16)         'names': names,
807: (16)         'titles': titles,
808: (16)         'aligned': aligned,
809: (16)         'byteorder': byteorder
810: (16)     }
811: (4) if obj is None:
812: (8)     if shape is None:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

813: (12)           raise ValueError("Must define a shape if obj is None")
814: (8)            return recarray(shape, dtype, buf=obj, offset=offset, strides=strides)
815: (4)            elif isinstance(obj, bytes):
816: (8)              return fromstring(obj, dtype, shape=shape, offset=offset, **kwds)
817: (4)            elif isinstance(obj, (list, tuple)):
818: (8)              if isinstance(obj[0], (tuple, list)):
819: (12)                return fromrecords(obj, dtype=dtype, shape=shape, **kwds)
820: (8)              else:
821: (12)                return fromarrays(obj, dtype=dtype, shape=shape, **kwds)
822: (4)            elif isinstance(obj, recarray):
823: (8)              if dtype is not None and (obj.dtype != dtype):
824: (12)                new = obj.view(dtype)
825: (8)              else:
826: (12)                new = obj
827: (8)              if copy:
828: (12)                new = new.copy()
829: (8)              return new
830: (4)            elif hasattr(obj, 'readinto'):
831: (8)              return fromfile(obj, dtype=dtype, shape=shape, offset=offset)
832: (4)            elif isinstance(obj, ndarray):
833: (8)              if dtype is not None and (obj.dtype != dtype):
834: (12)                new = obj.view(dtype)
835: (8)              else:
836: (12)                new = obj
837: (8)              if copy:
838: (12)                new = new.copy()
839: (8)              return new.view(recarray)
840: (4)            else:
841: (8)              interface = getattr(obj, "__array_interface__", None)
842: (8)              if interface is None or not isinstance(interface, dict):
843: (12)                raise ValueError("Unknown input type")
844: (8)              obj = sb.array(obj)
845: (8)              if dtype is not None and (obj.dtype != dtype):
846: (12)                obj = obj.view(dtype)
847: (8)              return obj.view(recarray)

```

---

## File 57 - shape\_base.py:

```

1: (0)          __all__ = ['atleast_1d', 'atleast_2d', 'atleast_3d', 'block', 'hstack',
2: (11)            'stack', 'vstack']
3: (0)          import functools
4: (0)          import itertools
5: (0)          import operator
6: (0)          import warnings
7: (0)          from . import numeric as _nx
8: (0)          from . import overrides
9: (0)          from .multiarray import array, asanyarray, normalize_axis_index
10: (0)         from . import fromnumeric as _from_nx
11: (0)         array_function_dispatch = functools.partial(
12: (4)           overrides.array_function_dispatch, module='numpy')
13: (0)         def _atleast_1d_dispatcher(*arys):
14: (4)           return arys
15: (0)         @array_function_dispatch(_atleast_1d_dispatcher)
16: (0)         def atleast_1d(*arys):
17: (4)           """
18: (4)             Convert inputs to arrays with at least one dimension.
19: (4)             Scalar inputs are converted to 1-dimensional arrays, whilst
20: (4)             higher-dimensional inputs are preserved.
21: (4)             Parameters
22: (4)             -----
23: (4)             arys1, arys2, ... : array_like
24: (8)               One or more input arrays.
25: (4)             Returns
26: (4)             -----
27: (4)             ret : ndarray
28: (8)               An array, or list of arrays, each with ``a.ndim >= 1``.
29: (8)               Copies are made only if necessary.

```

```

30: (4)           See Also
31: (4)           -----
32: (4)           atleast_2d, atleast_3d
33: (4)           Examples
34: (4)           -----
35: (4)           >>> np.atleast_1d(1.0)
36: (4)           array([1.])
37: (4)           >>> x = np.arange(9.0).reshape(3,3)
38: (4)           >>> np.atleast_1d(x)
39: (4)           array([[0., 1., 2.],
40: (11)          [3., 4., 5.],
41: (11)          [6., 7., 8.]])
42: (4)           >>> np.atleast_1d(x) is x
43: (4)           True
44: (4)           >>> np.atleast_1d(1, [3, 4])
45: (4)           [array([1]), array([3, 4])]
46: (4)           """
47: (4)           res = []
48: (4)           for ary in arys:
49: (8)             ary = asanyarray(ary)
50: (8)             if ary.ndim == 0:
51: (12)               result = ary.reshape(1)
52: (8)             else:
53: (12)               result = ary
54: (8)             res.append(result)
55: (4)             if len(res) == 1:
56: (8)               return res[0]
57: (4)             else:
58: (8)               return res
59: (0)           def _atleast_2d_dispatcher(*arys):
60: (4)             return arys
61: (0)           @array_function_dispatch(_atleast_2d_dispatcher)
62: (0)           def atleast_2d(*arys):
63: (4)             """
64: (4)               View inputs as arrays with at least two dimensions.
65: (4)               Parameters
66: (4)               -----
67: (4)               arys1, arys2, ... : array_like
68: (8)                 One or more array-like sequences. Non-array inputs are converted
69: (8)                 to arrays. Arrays that already have two or more dimensions are
70: (8)                 preserved.
71: (4)               Returns
72: (4)               -----
73: (4)               res, res2, ... : ndarray
74: (8)                 An array, or list of arrays, each with ``a.ndim >= 2``.
75: (8)                 Copies are avoided where possible, and views with two or more
76: (8)                 dimensions are returned.
77: (4)               See Also
78: (4)               -----
79: (4)               atleast_1d, atleast_3d
80: (4)               Examples
81: (4)               -----
82: (4)               >>> np.atleast_2d(3.0)
83: (4)               array([[3.]])
84: (4)               >>> x = np.arange(3.0)
85: (4)               >>> np.atleast_2d(x)
86: (4)               array([[0., 1., 2.]])
87: (4)               >>> np.atleast_2d(x).base is x
88: (4)               True
89: (4)               >>> np.atleast_2d(1, [1, 2], [[1, 2]])
90: (4)               [array([[1]]), array([[1, 2]]), array([[1, 2]])]
91: (4)               """
92: (4)               res = []
93: (4)               for ary in arys:
94: (8)                 ary = asanyarray(ary)
95: (8)                 if ary.ndim == 0:
96: (12)                   result = ary.reshape(1, 1)
97: (8)                 elif ary.ndim == 1:
98: (12)                   result = ary[_nx.newaxis, :]

```

```

99: (8)
100: (12)
101: (8)
102: (4)
103: (8)
104: (4)
105: (8)
106: (0)
107: (4)
108: (0)
109: (0)
110: (4)
111: (4)
112: (4)
113: (4)
114: (4)
115: (8)
116: (8)
117: (8)
118: (4)
119: (4)
120: (4)
121: (8)
122: (8)
123: (8)
124: (8)
125: (8)
126: (4)
127: (4)
128: (4)
129: (4)
130: (4)
131: (4)
132: (4)
133: (4)
134: (4)
135: (4)
136: (4)
137: (4)
138: (4)
139: (4)
140: (4)
141: (4)
142: (4)
143: (4)
144: (4)
145: (6)
146: (4)
147: (6)
148: (4)
149: (4)
150: (4)
151: (4)
152: (8)
153: (8)
154: (12)
155: (8)
156: (12)
157: (8)
158: (12)
159: (8)
160: (12)
161: (8)
162: (4)
163: (8)
164: (4)
165: (8)
166: (0)
167: (4)

        else:
            result = ary
            res.append(result)
        if len(res) == 1:
            return res[0]
        else:
            return res
    def _atleast_3d_dispatcher(*arys):
        return arys
    @array_function_dispatch(_atleast_3d_dispatcher)
    def atleast_3d(*arys):
        """
        View inputs as arrays with at least three dimensions.
        Parameters
        -----
        arys1, arys2, ... : array_like
            One or more array-like sequences. Non-array inputs are converted to
            arrays. Arrays that already have three or more dimensions are
            preserved.
        Returns
        -----
        res1, res2, ... : ndarray
            An array, or list of arrays, each with ``a.ndim >= 3``. Copies are
            avoided where possible, and views with three or more dimensions are
            returned. For example, a 1-D array of shape ``(N,)`` becomes a view
            of shape ``(1, N, 1)``, and a 2-D array of shape ``(M, N)`` becomes a
            view of shape ``(M, N, 1)``.
        See Also
        -----
        atleast_1d, atleast_2d
        Examples
        -----
        >>> np.atleast_3d(3.0)
        array([[[3.]]])
        >>> x = np.arange(3.0)
        >>> np.atleast_3d(x).shape
        (1, 3, 1)
        >>> x = np.arange(12.0).reshape(4,3)
        >>> np.atleast_3d(x).shape
        (4, 3, 1)
        >>> np.atleast_3d(x).base is x.base # x is a reshape, so not base itself
        True
        >>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
        ...     print(arr, arr.shape) # doctest: +SKIP
        ...
        [[[1
        [2]]]] (1, 2, 1)
        [[[1
        [2]]]] (1, 2, 1)
        [[[1 2]]]] (1, 1, 2)
        """
        res = []
        for ary in arys:
            ary = asanyarray(ary)
            if ary.ndim == 0:
                result = ary.reshape(1, 1, 1)
            elif ary.ndim == 1:
                result = ary[_nx.newaxis, :, _nx.newaxis]
            elif ary.ndim == 2:
                result = ary[:, :, _nx.newaxis]
            else:
                result = ary
            res.append(result)
        if len(res) == 1:
            return res[0]
        else:
            return res
    def _arrays_for_stack_dispatcher(arrays):
        if not hasattr(arrays, "__getitem__"):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

168: (8)           raise TypeError('arrays to stack must be passed as a "sequence" type '
169: (24)             'such as list or tuple.')
170: (4)           return tuple(arrays)
171: (0)           def _vhstack_dispatcher(tup, *, dtype=None, casting=None):
172: (4)             return _arrays_for_stack_dispatcher(tup)
173: (0)           @array_function_dispatch(_vhstack_dispatcher)
174: (0)           def vstack(tup, *, dtype=None, casting="same_kind"):
175: (4)             """
176: (4)               Stack arrays in sequence vertically (row wise).
177: (4)               This is equivalent to concatenation along the first axis after 1-D arrays
178: (4)                 of shape `(N,)` have been reshaped to `(1,N)`. Rebuilds arrays divided by
179: (4)                 `vsplit`.
180: (4)               This function makes most sense for arrays with up to 3 dimensions. For
181: (4)                 instance, for pixel-data with a height (first axis), width (second axis),
182: (4)                 and r/g/b channels (third axis). The functions `concatenate`, `stack` and
183: (4)                 `block` provide more general stacking and concatenation operations.
184: (4)               ``np.row_stack`` is an alias for `vstack`. They are the same function.
185: (4)           Parameters
186: (4)             -----
187: (4)             tup : sequence of ndarrays
188: (8)               The arrays must have the same shape along all but the first axis.
189: (8)               1-D arrays must have the same length.
190: (4)             dtype : str or dtype
191: (8)               If provided, the destination array will have this dtype. Cannot be
192: (8)                 provided together with `out`.
193: (4)             .. versionadded:: 1.24
194: (4)             casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
195: (8)               Controls what kind of data casting may occur. Defaults to 'same_kind'.
196: (4)             .. versionadded:: 1.24
197: (4)           Returns
198: (4)             -----
199: (4)             stacked : ndarray
200: (8)               The array formed by stacking the given arrays, will be at least 2-D.
201: (4)           See Also
202: (4)             -----
203: (4)             concatenate : Join a sequence of arrays along an existing axis.
204: (4)             stack : Join a sequence of arrays along a new axis.
205: (4)             block : Assemble an nd-array from nested lists of blocks.
206: (4)             hstack : Stack arrays in sequence horizontally (column wise).
207: (4)             dstack : Stack arrays in sequence depth wise (along third axis).
208: (4)             column_stack : Stack 1-D arrays as columns into a 2-D array.
209: (4)             vsplit : Split an array into multiple sub-arrays vertically (row-wise).
210: (4)           Examples
211: (4)             -----
212: (4)             >>> a = np.array([1, 2, 3])
213: (4)             >>> b = np.array([4, 5, 6])
214: (4)             >>> np.vstack((a,b))
215: (4)               array([[1, 2, 3],
216: (11)                 [4, 5, 6]])
217: (4)             >>> a = np.array([[1], [2], [3]])
218: (4)             >>> b = np.array([[4], [5], [6]])
219: (4)             >>> np.vstack((a,b))
220: (4)               array([[1],
221: (11)                 [2],
222: (11)                 [3],
223: (11)                 [4],
224: (11)                 [5],
225: (11)                 [6]])
226: (4)               """
227: (4)             arrs = atleast_2d(*tup)
228: (4)             if not isinstance(arrs, list):
229: (8)               arrs = [arrs]
230: (4)             return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)
231: (0)           @array_function_dispatch(_vhstack_dispatcher)
232: (0)           def hstack(tup, *, dtype=None, casting="same_kind"):
233: (4)             """
234: (4)               Stack arrays in sequence horizontally (column wise).
235: (4)               This is equivalent to concatenation along the second axis, except for 1-D
236: (4)                 arrays where it concatenates along the first axis. Rebuilds arrays divided

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

237: (4)                                by `hsplit`.
238: (4)                                This function makes most sense for arrays with up to 3 dimensions. For
239: (4)                                instance, for pixel-data with a height (first axis), width (second axis),
240: (4)                                and r/g/b channels (third axis). The functions `concatenate`, `stack` and
241: (4)                                `block` provide more general stacking and concatenation operations.
242: (4)                                Parameters
243: (4)                                -----
244: (4)                                tup : sequence of ndarrays
245: (8)                                The arrays must have the same shape along all but the second axis,
246: (8)                                except 1-D arrays which can be any length.
247: (4)                                dtype : str or dtype
248: (8)                                If provided, the destination array will have this dtype. Cannot be
249: (8)                                provided together with `out`.
250: (4)                                .. versionadded:: 1.24
251: (4)                                casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
252: (8)                                Controls what kind of data casting may occur. Defaults to 'same_kind'.
253: (4)                                .. versionadded:: 1.24
254: (4)                                Returns
255: (4)                                -----
256: (4)                                stacked : ndarray
257: (8)                                The array formed by stacking the given arrays.
258: (4)                                See Also
259: (4)                                -----
260: (4)                                concatenate : Join a sequence of arrays along an existing axis.
261: (4)                                stack : Join a sequence of arrays along a new axis.
262: (4)                                block : Assemble an nd-array from nested lists of blocks.
263: (4)                                vstack : Stack arrays in sequence vertically (row wise).
264: (4)                                dstack : Stack arrays in sequence depth wise (along third axis).
265: (4)                                column_stack : Stack 1-D arrays as columns into a 2-D array.
266: (4)                                hsplit : Split an array into multiple sub-arrays horizontally (column-
wise).
267: (4)                                Examples
268: (4)                                -----
269: (4)                                >>> a = np.array((1,2,3))
270: (4)                                >>> b = np.array((4,5,6))
271: (4)                                >>> np.hstack((a,b))
272: (4)                                array([1, 2, 3, 4, 5, 6])
273: (4)                                >>> a = np.array([[1],[2],[3]])
274: (4)                                >>> b = np.array([[4],[5],[6]])
275: (4)                                >>> np.hstack((a,b))
276: (4)                                array([[1, 4],
277: (11)                               [2, 5],
278: (11)                               [3, 6]])
279: (4)                                """
280: (4)                                arrs = atleast_1d(*tup)
281: (4)                                if not isinstance(arrs, list):
282: (8)                                    arrs = [arrs]
283: (4)                                if arrs and arrs[0].ndim == 1:
284: (8)                                    return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)
285: (4)                                else:
286: (8)                                    return _nx.concatenate(arrs, 1, dtype=dtype, casting=casting)
287: (0)                                def _stack_dispatcher(arrays, axis=None, out=None, *,
288: (22)                               dtype=None, casting=None):
289: (4)                                arrays = _arrays_for_stack_dispatcher(arrays)
290: (4)                                if out is not None:
291: (8)                                    arrays = list(arrays)
292: (8)                                    arrays.append(out)
293: (4)                                return arrays
294: (0)                                @array_function_dispatch(_stack_dispatcher)
295: (0)                                def stack(arrays, axis=0, out=None, *, dtype=None, casting="same_kind"):
296: (4)                                """
297: (4)                                Join a sequence of arrays along a new axis.
298: (4)                                The ``axis`` parameter specifies the index of the new axis in the
299: (4)                                dimensions of the result. For example, if ``axis=0`` it will be the first
300: (4)                                dimension and if ``axis=-1`` it will be the last dimension.
301: (4)                                .. versionadded:: 1.10.0
302: (4)                                Parameters
303: (4)                                -----
304: (4)                                arrays : sequence of array_like

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

305: (8)           Each array must have the same shape.
306: (4)           axis : int, optional
307: (8)           The axis in the result array along which the input arrays are stacked.
308: (4)           out : ndarray, optional
309: (8)           If provided, the destination to place the result. The shape must be
310: (8)           correct, matching that of what stack would have returned if no
311: (8)           out argument were specified.
312: (4)           dtype : str or dtype
313: (8)           If provided, the destination array will have this dtype. Cannot be
314: (8)           provided together with `out`.
315: (8)           .. versionadded:: 1.24
316: (4)           casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
317: (8)           Controls what kind of data casting may occur. Defaults to 'same_kind'.
318: (8)           .. versionadded:: 1.24
319: (4)           Returns
320: (4)           -----
321: (4)           stacked : ndarray
322: (8)           The stacked array has one more dimension than the input arrays.
323: (4)           See Also
324: (4)           -----
325: (4)           concatenate : Join a sequence of arrays along an existing axis.
326: (4)           block : Assemble an nd-array from nested lists of blocks.
327: (4)           split : Split array into a list of multiple sub-arrays of equal size.
328: (4)           Examples
329: (4)           -----
330: (4)           >>> arrays = [np.random.randn(3, 4) for _ in range(10)]
331: (4)           >>> np.stack(arrays, axis=0).shape
332: (4)           (10, 3, 4)
333: (4)           >>> np.stack(arrays, axis=1).shape
334: (4)           (3, 10, 4)
335: (4)           >>> np.stack(arrays, axis=2).shape
336: (4)           (3, 4, 10)
337: (4)           >>> a = np.array([1, 2, 3])
338: (4)           >>> b = np.array([4, 5, 6])
339: (4)           >>> np.stack((a, b))
340: (4)           array([[1, 2, 3],
341: (11)             [4, 5, 6]])
342: (4)           >>> np.stack((a, b), axis=-1)
343: (4)           array([[1, 4],
344: (11)             [2, 5],
345: (11)             [3, 6]])
346: (4)           """
347: (4)           arrays = [asanyarray(arr) for arr in arrays]
348: (4)           if not arrays:
349: (8)               raise ValueError('need at least one array to stack')
350: (4)           shapes = {arr.shape for arr in arrays}
351: (4)           if len(shapes) != 1:
352: (8)               raise ValueError('all input arrays must have the same shape')
353: (4)           result_ndim = arrays[0].ndim + 1
354: (4)           axis = normalize_axis_index(axis, result_ndim)
355: (4)           s1 = (slice(None),) * axis + (_nx.newaxis,)
356: (4)           expanded_arrays = [arr[s1] for arr in arrays]
357: (4)           return _nx.concatenate(expanded_arrays, axis=axis, out=out,
358: (27)                 dtype=dtype, casting=casting)
359: (0)           _size = getattr(_from_nx.size, '__wrapped__', _from_nx.size)
360: (0)           _ndim = getattr(_from_nx.ndim, '__wrapped__', _from_nx.ndim)
361: (0)           _concatenate = getattr(_from_nx.concatenate,
362: (23)                 '__wrapped__', _from_nx.concatenate)
363: (0)           def _block_format_index(index):
364: (4)               """
365: (4)               Convert a list of indices ``[0, 1, 2]`` into ``"arrays[0][1][2]"``.
366: (4)               """
367: (4)               idx_str = ''.join('[{}]'.format(i) for i in index if i is not None)
368: (4)               return 'arrays' + idx_str
369: (0)           def _block_check_depths_match(arrays, parent_index=[]):
370: (4)               """
371: (4)               Recursive function checking that the depths of nested lists in `arrays`
372: (4)               all match. Mismatch raises a ValueError as described in the block
373: (4)               docstring below.

```

```

374: (4)             The entire index (rather than just the depth) needs to be calculated
375: (4)             for each innermost list, in case an error needs to be raised, so that
376: (4)             the index of the offending list can be printed as part of the error.
377: (4)             Parameters
378: (4)             -----
379: (4)             arrays : nested list of arrays
380: (8)               The arrays to check
381: (4)             parent_index : list of int
382: (8)               The full index of `arrays` within the nested lists passed to
383: (8)               `_block_check_depths_match` at the top of the recursion.
384: (4)             Returns
385: (4)             -----
386: (4)             first_index : list of int
387: (8)               The full index of an element from the bottom of the nesting in
388: (8)               `arrays`. If any element at the bottom is an empty list, this will
389: (8)               refer to it, and the last index along the empty axis will be None.
390: (4)             max_arr_ndim : int
391: (8)               The maximum of the ndims of the arrays nested in `arrays`.
392: (4)             final_size: int
393: (8)               The number of elements in the final array. This is used the motivate
394: (8)               the choice of algorithm used using benchmarking wisdom.
395: (4)             """
396: (4)             if type(arrays) is tuple:
397: (8)               raise TypeError(
398: (12)                 '{} is a tuple. '
399: (12)                 'Only lists can be used to arrange blocks, and np.block does '
400: (12)                 'not allow implicit conversion from tuple to ndarray.'.format(
401: (16)                   _block_format_index(parent_index)
402: (12)                 )
403: (8)               )
404: (4)             elif type(arrays) is list and len(arrays) > 0:
405: (8)               idxs_ndims = (_block_check_depths_match(arr, parent_index + [i])
406: (22)                 for i, arr in enumerate(arrays))
407: (8)               first_index, max_arr_ndim, final_size = next(idxs_ndims)
408: (8)               for index, ndim, size in idxs_ndims:
409: (12)                 final_size += size
410: (12)                 if ndim > max_arr_ndim:
411: (16)                   max_arr_ndim = ndim
412: (12)                 if len(index) != len(first_index):
413: (16)                   raise ValueError(
414: (20)                     "List depths are mismatched. First element was at depth "
415: (20)                     "{}, but there is an element at depth {} ({}).".format(
416: (24)                       len(first_index),
417: (24)                       len(index),
418: (24)                       _block_format_index(index)
419: (20)                     )
420: (16)               )
421: (12)               if index[-1] is None:
422: (16)                 first_index = index
423: (8)               return first_index, max_arr_ndim, final_size
424: (4)             elif type(arrays) is list and len(arrays) == 0:
425: (8)               return parent_index + [None], 0, 0
426: (4)             else:
427: (8)               size = _size(arrays)
428: (8)               return parent_index, _ndim(arrays), size
429: (0)             def _atleast_nd(a, ndim):
430: (4)               return array(a, ndmin=ndim, copy=False, subok=True)
431: (0)             def _accumulate(values):
432: (4)               return list(itertools.accumulate(values))
433: (0)             def _concatenate_shapes(shapes, axis):
434: (4)               """Given array shapes, return the resulting shape and slices prefixes.
435: (4)               These help in nested concatenation.
436: (4)               Returns
437: (4)               -----
438: (4)               shape: tuple of int
439: (8)                 This tuple satisfies::
440: (12)                   shape, _ = _concatenate_shapes([arr.shape for shape in arrs],
axis)
441: (12)                   shape == concatenate(arrs, axis).shape

```

```

442: (4) slice_prefixes: tuple of (slice(start, end), )
443: (8)     For a list of arrays being concatenated, this returns the slice
444: (8)     in the larger array at axis that needs to be sliced into.
445: (8)     For example, the following holds::
446: (12)         ret = concatenate([a, b, c], axis)
447: (12)         _, (sl_a, sl_b, sl_c) = concatenate_slices([a, b, c], axis)
448: (12)         ret[(slice(None),) * axis + sl_a] == a
449: (12)         ret[(slice(None),) * axis + sl_b] == b
450: (12)         ret[(slice(None),) * axis + sl_c] == c
451: (8)     These are called slice prefixes since they are used in the recursive
452: (8)     blocking algorithm to compute the left-most slices during the
453: (8)     recursion. Therefore, they must be prepended to rest of the slice
454: (8)     that was computed deeper in the recursion.
455: (8)     These are returned as tuples to ensure that they can quickly be added
456: (8)     to existing slice tuple without creating a new tuple every time.
457: (4) """
458: (4)     shape_at_axis = [shape[axis] for shape in shapes]
459: (4)     first_shape = shapes[0]
460: (4)     first_shape_pre = first_shape[:axis]
461: (4)     first_shape_post = first_shape[axis+1:]
462: (4)     if any(shape[:axis] != first_shape_pre or
463: (11)         shape[axis+1:] != first_shape_post for shape in shapes):
464: (8)         raise ValueError(
465: (12)             'Mismatched array shapes in block along axis {}'.format(axis))
466: (4)     shape = (first_shape_pre + (sum(shape_at_axis),) + first_shape[axis+1:])
467: (4)     offsets_at_axis = _accumulate(shape_at_axis)
468: (4)     slice_prefixes = [(slice(start, end),)
469: (22)         for start, end in zip([0] + offsets_at_axis,
470: (44)                         offsets_at_axis)]
471: (4)     return shape, slice_prefixes
472: (0) def _block_info_recursion(arrays, max_depth, result_ndim, depth=0):
473: (4) """
474: (4)     Returns the shape of the final array, along with a list
475: (4)     of slices and a list of arrays that can be used for assignment inside the
476: (4)     new array
477: (4)     Parameters
478: (4)     -----
479: (4)     arrays : nested list of arrays
480: (8)         The arrays to check
481: (4)     max_depth : list of int
482: (8)         The number of nested lists
483: (4)     result_ndim : int
484: (8)         The number of dimensions in thefinal array.
485: (4)     Returns
486: (4)     -----
487: (4)     shape : tuple of int
488: (8)         The shape that the final array will take on.
489: (4)     slices: list of tuple of slices
490: (8)         The slices into the full array required for assignment. These are
491: (8)         required to be prepended with ``Ellipsis, )`` to obtain to correct
492: (8)         final index.
493: (4)     arrays: list of ndarray
494: (8)         The data to assign to each slice of the full array
495: (4) """
496: (4)     if depth < max_depth:
497: (8)         shapes, slices, arrays = zip(
498: (12)             *[_block_info_recursion(arr, max_depth, result_ndim, depth+1)
499: (14)                 for arr in arrays])
500: (8)         axis = result_ndim - max_depth + depth
501: (8)         shape, slice_prefixes = _concatenate_shapes(shapes, axis)
502: (8)         slices = [slice_prefix + the_slice
503: (18)             for slice_prefix, inner_slices in zip(slice_prefixes,
slices)
504: (18)                 for the_slice in inner_slices]
505: (8)         arrays = functools.reduce(operator.add, arrays)
506: (8)         return shape, slices, arrays
507: (4)     else:
508: (8)         arr = _atleast_nd(arrays, result_ndim)
509: (8)         return arr.shape, [()], [arr]

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

510: (0)
511: (4)
512: (4)
513: (4)
514: (4)
515: (4)
516: (4)
517: (4)
518: (4)
519: (4)
520: (8)
521: (16)
522: (8)
523: (4)
524: (8)
525: (0)
526: (4)
527: (8)
528: (12)
529: (4)
530: (8)
531: (0)
532: (0)
533: (4)
534: (4)
535: (4)
536: (4)
537: (4)
538: (4)
normal
539: (4)
``block.ndim``
540: (4)
scalars,
541: (4)
542: (4)
543: (4)
544: (4)
545: (4)
546: (4)
547: (4)
548: (4)
549: (8)
550: (8)
551: (8)
552: (8)
553: (8)
554: (4)
555: (4)
556: (4)
557: (8)
558: (8)
559: (8)
560: (8)
561: (4)
562: (4)
563: (4)
564: (8)
565: (10)
566: (8)
567: (4)
568: (4)
569: (4)
570: (4)
571: (4)
572: (4)
573: (4)
574: (4)
575: (4)

def _block(arrays, max_depth, result_ndim, depth=0):
    """
    Internal implementation of block based on repeated concatenation.
    `arrays` is the argument passed to
    block. `max_depth` is the depth of nested lists within `arrays` and
    `result_ndim` is the greatest of the dimensions of the arrays in
    `arrays` and the depth of the lists in `arrays` (see block docstring
    for details).
    """
    if depth < max_depth:
        arrs = [_block(arr, max_depth, result_ndim, depth+1)
                for arr in arrays]
        return _concatenate(arrs, axis=-(max_depth-depth))
    else:
        return _atleast_nd(arrays, result_ndim)

def _block_dispatcher(arrays):
    if type(arrays) is list:
        for subarrays in arrays:
            yield from _block_dispatcher(subarrays)
    else:
        yield arrays

@array_function_dispatch(_block_dispatcher)
def block(arrays):
    """
    Assemble an nd-array from nested lists of blocks.
    Blocks in the innermost lists are concatenated (see `concatenate`) along
    the last dimension (-1), then these are concatenated along the
    second-last dimension (-2), and so on until the outermost list is reached.
    Blocks can be of any dimension, but will not be broadcasted using the
    rules. Instead, leading axes of size 1 are inserted, to make
    the same for all blocks. This is primarily useful for working with
    and means that code like ``np.block([v, 1])`` is valid, where
    ``v.ndim == 1``.
    When the nested list is two levels deep, this allows block matrices to be
    constructed from their components.
    .. versionadded:: 1.13.0
    Parameters
    -----
    arrays : nested list of array_like or scalars (but not tuples)
        If passed a single ndarray or scalar (a nested list of depth 0), this
        is returned unmodified (and not copied).
        Elements shapes must match along the appropriate axes (without
        broadcasting), but leading 1s will be prepended to the shape as
        necessary to make the dimensions match.
    Returns
    -----
    block_array : ndarray
        The array assembled from the given blocks.
        The dimensionality of the output is equal to the greatest of:
        * the dimensionality of all the inputs
        * the depth to which the input list is nested
    Raises
    -----
    ValueError
        * If list depths are mismatched - for instance, ``[[a, b], c]`` is
          illegal, and should be spelt ``[[a, b], [c]]``
        * If lists are empty - for instance, ``[[a, b], []]``
    See Also
    -----
    concatenate : Join a sequence of arrays along an existing axis.
    stack : Join a sequence of arrays along a new axis.
    vstack : Stack arrays in sequence vertically (row wise).
    hstack : Stack arrays in sequence horizontally (column wise).
    dstack : Stack arrays in sequence depth wise (along third axis).
    column_stack : Stack 1-D arrays as columns into a 2-D array.
    vsplit : Split an array into multiple sub-arrays vertically (row-wise).

```

```

576: (4)          Notes
577: (4)          -----
578: (4)          When called with only scalars, ``np.block`` is equivalent to an ndarray
579: (4)          call. So ``np.block([[1, 2], [3, 4]])`` is equivalent to
580: (4)          ``np.array([[1, 2], [3, 4]])``.
581: (4)          This function does not enforce that the blocks lie on a fixed grid.
582: (4)          ``np.block([[a, b], [c, d]])`` is not restricted to arrays of the form::
583: (8)          AAAabb
584: (8)          AAAabb
585: (8)          cccDD
586: (4)          But is also allowed to produce, for some ``a, b, c, d``:::
587: (8)          AAAabb
588: (8)          AAAabb
589: (8)          cDDDD
590: (4)          Since concatenation happens along the last axis first, `block` is _not_
591: (4)          capable of producing the following directly:::
592: (8)          AAAabb
593: (8)          cccbb
594: (8)          cccDD
595: (4)          Matlab's "square bracket stacking", ``[A, B, ...; p, q, ...]``, is
596: (4)          equivalent to ``np.block([[A, B, ...], [p, q, ...]])``.
597: (4)          Examples
598: (4)          -----
599: (4)          The most common use of this function is to build a block matrix
600: (4)          >>> A = np.eye(2) * 2
601: (4)          >>> B = np.eye(3) * 3
602: (4)          >>> np.block([
603: (4)              ...      [A,                      np.zeros((2, 3))],
604: (4)              ...      [np.ones((3, 2)), B        ],
605: (4)              ...  ])
606: (4)          array([[2., 0., 0., 0., 0.],
607: (11)             [0., 2., 0., 0., 0.],
608: (11)             [1., 1., 3., 0., 0.],
609: (11)             [1., 1., 0., 3., 0.],
610: (11)             [1., 1., 0., 0., 3.]])
611: (4)          With a list of depth 1, `block` can be used as `hstack`
612: (4)          >>> np.block([1, 2, 3])           # hstack([1, 2, 3])
613: (4)          array([1, 2, 3])
614: (4)          >>> a = np.array([1, 2, 3])
615: (4)          >>> b = np.array([4, 5, 6])
616: (4)          >>> np.block([a, b, 10])       # hstack([a, b, 10])
617: (4)          array([ 1, 2, 3, 4, 5, 6, 10])
618: (4)          >>> A = np.ones((2, 2), int)
619: (4)          >>> B = 2 * A
620: (4)          >>> np.block([A, B])           # hstack([A, B])
621: (4)          array([[1, 1, 2, 2],
622: (11)             [1, 1, 2, 2]])
623: (4)          With a list of depth 2, `block` can be used in place of `vstack`:
624: (4)          >>> a = np.array([1, 2, 3])
625: (4)          >>> b = np.array([4, 5, 6])
626: (4)          >>> np.block([[a], [b]])       # vstack([a, b])
627: (4)          array([[1, 2, 3],
628: (11)             [4, 5, 6]])
629: (4)          >>> A = np.ones((2, 2), int)
630: (4)          >>> B = 2 * A
631: (4)          >>> np.block([[A], [B]])       # vstack([A, B])
632: (4)          array([[1, 1],
633: (11)             [1, 1],
634: (11)             [2, 2],
635: (11)             [2, 2]])
636: (4)          It can also be used in places of `atleast_1d` and `atleast_2d`
637: (4)          >>> a = np.array(0)
638: (4)          >>> b = np.array([1])
639: (4)          >>> np.block([a])           # atleast_1d(a)
640: (4)          array([0])
641: (4)          >>> np.block([b])           # atleast_1d(b)
642: (4)          array([1])
643: (4)          >>> np.block([[a]])        # atleast_2d(a)
644: (4)          array([[0]])

```

```

645: (4)             >>> np.block([[b]])                      # atleast_2d(b)
646: (4)             array([[1]])
647: (4)
648: (4)             """
649: (4)             arrays, list_ndim, result_ndim, final_size = _block_setup(arrays)
650: (8)             if list_ndim * final_size > (2 * 512 * 512):
651: (4)                 return _block_slicing(arrays, list_ndim, result_ndim)
652: (8)             else:
653: (0)                 return _block_concatenate(arrays, list_ndim, result_ndim)
654: (4)             def _block_setup(arrays):
655: (4)                 """
656: (4)                 Returns
657: (4)                 (`arrays`, list_ndim, result_ndim, final_size)
658: (4)                 """
659: (4)                 bottom_index, arr_ndim, final_size = _block_check_depths_match(arrays)
660: (4)                 list_ndim = len(bottom_index)
661: (4)                 if bottom_index and bottom_index[-1] is None:
662: (8)                     raise ValueError(
663: (12)                         'List at {} cannot be empty'.format(
664: (16)                             _block_format_index(bottom_index)
665: (12)                         )
666: (8)                     )
667: (4)                 result_ndim = max(arr_ndim, list_ndim)
668: (0)                 return arrays, list_ndim, result_ndim, final_size
669: (4)             def _block_slicing(arrays, list_ndim, result_ndim):
670: (8)                 shape, slices, arrays = _block_info_recursion(
671: (4)                     arrays, list_ndim, result_ndim)
672: (4)                 dtype = _nx.result_type(*[arr.dtype for arr in arrays])
673: (4)                 F_order = all(arr.flags['F_CONTIGUOUS'] for arr in arrays)
674: (4)                 C_order = all(arr.flags['C_CONTIGUOUS'] for arr in arrays)
675: (4)                 order = 'F' if F_order and not C_order else 'C'
676: (4)                 result = _nx.empty(shape=shape, dtype=dtype, order=order)
677: (8)                 for the_slice, arr in zip(slices, arrays):
678: (4)                     result[(Ellipsis,) + the_slice] = arr
679: (0)             return result
680: (4)             def _block_concatenate(arrays, list_ndim, result_ndim):
681: (4)                 result = _block(arrays, list_ndim, result_ndim)
682: (8)                 if list_ndim == 0:
683: (4)                     result = result.copy()

```

---

## File 58 - umath.py:

```

1: (0)             """
2: (0)             Create the numpy.core.umath namespace for backward compatibility. In v1.16
3: (0)             the multiarray and umath c-extension modules were merged into a single
4: (0)             _multiarray_umath extension module. So we replicate the old namespace
5: (0)             by importing from the extension module.
6: (0)             """
7: (0)             from . import _multiarray_umath
8: (0)             from ._multiarray_umath import * # noqa: F403
9: (0)             from ._multiarray_umath import _UFUNC_API, _add_newdoc_ufunc, _ones_like
10: (0)             __all__ = [
11: (4)                 '_UFUNC_API', 'ERR_CALL', 'ERR_DEFAULT', 'ERR_IGNORE', 'ERR_LOG',
12: (4)                 'ERR_PRINT', 'ERR_RAISE', 'ERR_WARN', 'FLOATING_POINT_SUPPORT',
13: (4)                 'FPE_DIVIDEBYZERO', 'FPE_INVALID', 'FPE_OVERFLOW', 'FPE_UNDERFLOW', 'NAN',
14: (4)                 'NINF', 'NZERO', 'PINF', 'PZERO', 'SHIFT_DIVIDEBYZERO', 'SHIFT_INVALID',
15: (4)                 'SHIFT_OVERFLOW', 'SHIFT_UNDERFLOW', 'UFUNC_BUFSIZE_DEFAULT',
16: (4)                 'UFUNC_PYVALS_NAME', '_add_newdoc_ufunc', 'absolute', 'add',
17: (4)                 'arccos', 'arccosh', 'arcsin', 'arcsinh', 'arctan', 'arctan2', 'arctanh',
18: (4)                 'bitwise_and', 'bitwise_or', 'bitwise_xor', 'cbrt', 'ceil', 'conj',
19: (4)                 'conjugate', 'copysign', 'cos', 'cosh', 'deg2rad', 'degrees', 'divide',
20: (4)                 'divmod', 'e', 'equal', 'euler_gamma', 'exp', 'exp2', 'expm1', 'fabs',
21: (4)                 'floor', 'floor_divide', 'float_power', 'fmax', 'fmin', 'fmod', 'frexp',
22: (4)                 'frompyfunc', 'gcd', 'geterrobj', 'greater', 'greater_equal', 'heaviside',
23: (4)                 'hypot', 'invert', 'isfinite', 'isinf', 'isnan', 'isnat', 'lcm', 'ldexp',
24: (4)                 'left_shift', 'less', 'less_equal', 'log', 'log10', 'log1p', 'log2',
25: (4)                 'logaddexp', 'logaddexp2', 'logical_and', 'logical_not', 'logical_or',

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

26: (4)      'logical_xor', 'maximum', 'minimum', 'mod', 'modf', 'multiply',
'negative',
27: (4)      'nextafter', 'not_equal', 'pi', 'positive', 'power', 'rad2deg', 'radians',
28: (4)      'reciprocal', 'remainder', 'right_shift', 'rint', 'seterrobj', 'sign',
29: (4)      'signbit', 'sin', 'sinh', 'spacing', 'sqrt', 'square', 'subtract', 'tan',
30: (4)      'tanh', 'true_divide', 'trunc']
-----
```

## File 59 - umath\_tests.py:

```

1: (0)      """
2: (0)      Shim for _umath_tests to allow a deprecation period for the new name.
3: (0)      """
4: (0)      import warnings
5: (0)      warnings.warn(("numpy.core.umath_tests is an internal NumPy "
6: (15)          "module and should not be imported. It will "
7: (15)          "be removed in a future NumPy release."),
8: (14)          category=DeprecationWarning, stacklevel=2)
9: (0)      from ._umath_tests import *
-----
```

## File 60 - \_add\_newdocs.py:

```

1: (0)      """
2: (0)      This is only meant to add docs to objects defined in C-extension modules.
3: (0)      The purpose is to allow easier editing of the docstrings without
4: (0)      requiring a re-compile.
5: (0)      NOTE: Many of the methods of ndarray have corresponding functions.
6: (6)          If you update these docstrings, please keep also the ones in
7: (6)          core/fromnumeric.py, core/defmatrix.py up-to-date.
8: (0)      """
9: (0)      from numpy.core.function_base import add_newdoc
10: (0)      from numpy.core.overrides import array_function_like_doc
11: (0)      add_newdoc('numpy.core', 'flatiter',
12: (4)          """
13: (4)          Flat iterator object to iterate over arrays.
14: (4)          A `flatiter` iterator is returned by ``x.flat`` for any array `x`.
15: (4)          It allows iterating over the array as if it were a 1-D array,
16: (4)          either in a for-loop or by calling its `next` method.
17: (4)          Iteration is done in row-major, C-style order (the last
18: (4)          index varying the fastest). The iterator can also be indexed using
19: (4)          basic slicing or advanced indexing.
20: (4)          See Also
21: (4)          -----
22: (4)          ndarray.flat : Return a flat iterator over an array.
23: (4)          ndarray.flatten : Returns a flattened copy of an array.
24: (4)          Notes
25: (4)          -----
26: (4)          A `flatiter` iterator can not be constructed directly from Python code
27: (4)          by calling the `flatiter` constructor.
28: (4)          Examples
29: (4)          -----
30: (4)          >>> x = np.arange(6).reshape(2, 3)
31: (4)          >>> f1 = x.flat
32: (4)          >>> type(f1)
33: (4)          <class 'numpy.flatiter'>
34: (4)          >>> for item in f1:
35: (4)              ...      print(item)
36: (4)              ...
37: (4)              0
38: (4)              1
39: (4)              2
40: (4)              3
41: (4)              4
42: (4)              5
43: (4)              >>> f1[2:4]
44: (4)              array([2, 3])
-----
```

```

45: (4)      """
46: (0)      add_newdoc('numpy.core', 'flatiter', ('base',
47: (4)          """
48: (4)          A reference to the array that is iterated over.
49: (4)          Examples
50: (4)          -----
51: (4)          >>> x = np.arange(5)
52: (4)          >>> f1 = x.flat
53: (4)          >>> f1.base is x
54: (4)          True
55: (4)          """)
56: (0)      add_newdoc('numpy.core', 'flatiter', ('coords',
57: (4)          """
58: (4)          An N-dimensional tuple of current coordinates.
59: (4)          Examples
60: (4)          -----
61: (4)          >>> x = np.arange(6).reshape(2, 3)
62: (4)          >>> f1 = x.flat
63: (4)          >>> f1.coords
64: (4)          (0, 0)
65: (4)          >>> next(f1)
66: (4)          0
67: (4)          >>> f1.coords
68: (4)          (0, 1)
69: (4)          ""))
70: (0)      add_newdoc('numpy.core', 'flatiter', ('index',
71: (4)          """
72: (4)          Current flat index into the array.
73: (4)          Examples
74: (4)          -----
75: (4)          >>> x = np.arange(6).reshape(2, 3)
76: (4)          >>> f1 = x.flat
77: (4)          >>> f1.index
78: (4)          0
79: (4)          >>> next(f1)
80: (4)          0
81: (4)          >>> f1.index
82: (4)          1
83: (4)          ""))
84: (0)      add_newdoc('numpy.core', 'flatiter', ('__array__',
85: (4)          """
86: (4)          __array__(type=None) Get array from iterator
87: (4)          """))
88: (4)      add_newdoc('numpy.core', 'flatiter', ('copy',
89: (4)          """
90: (4)          copy()
91: (4)          Get a copy of the iterator as a 1-D array.
92: (4)          Examples
93: (4)          -----
94: (4)          >>> x = np.arange(6).reshape(2, 3)
95: (4)          >>> x
96: (11)          array([[0, 1, 2],
97: (4)              [3, 4, 5]])
98: (4)          >>> f1 = x.flat
99: (4)          >>> f1.copy()
100: (4)          array([0, 1, 2, 3, 4, 5])
101: (4)          ""))
102: (0)      add_newdoc('numpy.core', 'nditer',
103: (4)          """
104: (4)          nditer(op, flags=None, op_flags=None, op_dtypes=None, order='K',
casting='safe', op_axes=None, itershape=None, buffersize=0)
105: (4)          Efficient multi-dimensional iterator object to iterate over arrays.
106: (4)          To get started using this object, see the
107: (4)          :ref:`introductory guide to array iteration <arrays.nditer>`.
108: (4)          Parameters
109: (4)          -----
110: (8)          op : ndarray or sequence of array_like
111: (4)          The array(s) to iterate over.
112: (10)         flags : sequence of str, optional
113: (4)          Flags to control the behavior of the iterator.

```

```

113: (10) * ``buffered`` enables buffering when required.
114: (10) * ``c_index`` causes a C-order index to be tracked.
115: (10) * ``f_index`` causes a Fortran-order index to be tracked.
116: (10) * ``multi_index`` causes a multi-index, or a tuple of indices
117: (12) with one per iteration dimension, to be tracked.
118: (10) * ``common_dtype`` causes all the operands to be converted to
119: (12) a common data type, with copying or buffering as necessary.
120: (10) * ``copy_if_overlap`` causes the iterator to determine if read
121: (12) operands have overlap with write operands, and make temporary
122: (12) copies as necessary to avoid overlap. False positives (needless
123: (12) copying) are possible in some cases.
124: (10) * ``delay_bufalloc`` delays allocation of the buffers until
125: (12) a reset() call is made. Allows ``allocate`` operands to
126: (12) be initialized before their values are copied into the buffers.
127: (10) * ``external_loop`` causes the ``values`` given to be
128: (12) one-dimensional arrays with multiple values instead of
129: (12) zero-dimensional arrays.
130: (10) * ``grow_inner`` allows the ``value`` array sizes to be made
131: (12) larger than the buffer size when both ``buffered`` and
132: (12) ``external_loop`` is used.
133: (10) * ``ranged`` allows the iterator to be restricted to a sub-range
134: (12) of the iterindex values.
135: (10) * ``refs_ok`` enables iteration of reference types, such as
136: (12) object arrays.
137: (10) * ``reduce_ok`` enables iteration of ``readwrite`` operands
138: (12) which are broadcasted, also known as reduction operands.
139: (10) * ``zerosize_ok`` allows `itersize` to be zero.

op_flags : list of list of str, optional
    This is a list of flags for each operand. At minimum, one of
    ``readonly``, ``readwrite``, or ``writeonly`` must be specified.
    * ``readonly`` indicates the operand will only be read from.
    * ``readwrite`` indicates the operand will be read from and written
    to.
    * ``writeonly`` indicates the operand will only be written to.
    * ``no_broadcast`` prevents the operand from being broadcasted.
    * ``contig`` forces the operand data to be contiguous.
    * ``aligned`` forces the operand data to be aligned.
    * ``nbo`` forces the operand data to be in native byte order.
    * ``copy`` allows a temporary read-only copy if required.
    * ``updateifcopy`` allows a temporary read-write copy if required.
    * ``allocate`` causes the array to be allocated if it is None
    in the ``op`` parameter.
    * ``no_subtype`` prevents an ``allocate`` operand from using a
    subtype.
    * ``arraymask`` indicates that this operand is the mask to use
    for selecting elements when writing to operands with the
    'writemasked' flag set. The iterator does not enforce this,
    but when writing from a buffer back to the array, it only
    copies those elements indicated by this mask.
    * ``writemasked`` indicates that only elements where the chosen
    ``arraymask`` operand is True will be written to.
    * ``overlap_assume_elementwise`` can be used to mark operands that
    accessed only in the iterator order, to allow less conservative
    copying when ``copy_if_overlap`` is present.

op_dtotypes : dtype or tuple of dtype(s), optional
    The required data type(s) of the operands. If copying or buffering
    is enabled, the data will be converted to/from their original types.

order : {'C', 'F', 'A', 'K'}, optional
    Controls the iteration order. 'C' means C order, 'F' means
    Fortran order, 'A' means 'F' order if all the arrays are Fortran
    contiguous, 'C' order otherwise, and 'K' means as close to the
    order the array elements appear in memory as possible. This also
    affects the element memory order of ``allocate`` operands, as they
    are allocated to be compatible with iteration order.
    Default is 'K'.

casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
    Controls what kind of data casting may occur when making a copy
    or buffering. Setting this to 'unsafe' is not recommended,

```

```

179: (8)           as it can adversely affect accumulations.
180: (8)           * 'no' means the data types should not be cast at all.
181: (8)           * 'equiv' means only byte-order changes are allowed.
182: (8)           * 'safe' means only casts which can preserve values are allowed.
183: (8)           * 'same_kind' means only safe casts or casts within a kind,
184: (10)          like float64 to float32, are allowed.
185: (8)           * 'unsafe' means any data conversions may be done.
186: (4)            op_axes : list of list of ints, optional
187: (8)              If provided, is a list of ints or None for each operands.
188: (8)              The list of axes for an operand is a mapping from the dimensions
189: (8)              of the iterator to the dimensions of the operand. A value of
190: (8)              -1 can be placed for entries, causing that dimension to be
191: (8)              treated as `newaxis`.
192: (4)            iitershape : tuple of ints, optional
193: (8)              The desired shape of the iterator. This allows ``allocate`` operands
194: (8)              with a dimension mapped by op_axes not corresponding to a dimension
195: (8)              of a different operand to get a value not equal to 1 for that
196: (8)              dimension.
197: (4)            buffersize : int, optional
198: (8)              When buffering is enabled, controls the size of the temporary
199: (8)              buffers. Set to 0 for the default value.
200: (4)            Attributes
201: (4)            -----
202: (4)            dtypes : tuple of dtype(s)
203: (8)              The data types of the values provided in `value`. This may be
204: (8)              different from the operand data types if buffering is enabled.
205: (8)              Valid only before the iterator is closed.
206: (4)            finished : bool
207: (8)              Whether the iteration over the operands is finished or not.
208: (4)            has_delayed_bufalloc : bool
209: (8)              If True, the iterator was created with the ``delay_bufalloc`` flag,
210: (8)              and no reset() function was called on it yet.
211: (4)            has_index : bool
212: (8)              If True, the iterator was created with either the ``c_index`` or
213: (8)              the ``f_index`` flag, and the property `index` can be used to
214: (8)              retrieve it.
215: (4)            has_multi_index : bool
216: (8)              If True, the iterator was created with the ``multi_index`` flag,
217: (8)              and the property `multi_index` can be used to retrieve it.
218: (4)            index
219: (8)              When the ``c_index`` or ``f_index`` flag was used, this property
220: (8)              provides access to the index. Raises a ValueError if accessed
221: (8)              and ``has_index`` is False.
222: (4)            iterationneedsapi : bool
223: (8)              Whether iteration requires access to the Python API, for example
224: (8)              if one of the operands is an object array.
225: (4)            iterindex : int
226: (8)              An index which matches the order of iteration.
227: (4)            itersize : int
228: (8)              Size of the iterator.
229: (4)            itviews
230: (8)              Structured view(s) of `operands` in memory, matching the reordered
231: (8)              and optimized iterator access pattern. Valid only before the iterator
232: (8)              is closed.
233: (4)            multi_index
234: (8)              When the ``multi_index`` flag was used, this property
235: (8)              provides access to the index. Raises a ValueError if accessed
236: (8)              accessed and ``has_multi_index`` is False.
237: (4)            ndim : int
238: (8)              The dimensions of the iterator.
239: (4)            nops : int
240: (8)              The number of iterator operands.
241: (4)            operands : tuple of operand(s)
242: (8)              The array(s) to be iterated over. Valid only before the iterator is
243: (8)              closed.
244: (4)            shape : tuple of ints
245: (8)              Shape tuple, the shape of the iterator.
246: (4)            value
247: (8)              Value of ``operands`` at current iteration. Normally, this is a

```

```

248: (8)          tuple of array scalars, but if the flag ``external_loop`` is used,
249: (8)          it is a tuple of one dimensional arrays.
250: (4)          Notes
251: (4)          -----
252: (4)          `nditer` supersedes `flatiter`. The iterator implementation behind
253: (4)          `nditer` is also exposed by the NumPy C API.
254: (4)          The Python exposure supplies two iteration interfaces, one which follows
255: (4)          the Python iterator protocol, and another which mirrors the C-style
256: (4)          do-while pattern. The native Python approach is better in most cases, but
257: (4)          if you need the coordinates or index of an iterator, use the C-style
pattern.
258: (4)          Examples
259: (4)          -----
260: (4)          Here is how we might write an ``iter_add`` function, using the
261: (4)          Python iterator protocol:
262: (4)          >>> def iter_add_py(x, y, out=None):
263: (4)              ...      addop = np.add
264: (4)              ...      it = np.nditer([x, y, out], [],
265: (4)                          ...                  [['readonly']], ['readonly'],
['writeonly','allocate']])
266: (4)              ...      with it:
267: (4)                  ...          for (a, b, c) in it:
268: (4)                      ...              addop(a, b, out=c)
269: (4)                      ...          return it.operands[2]
270: (4)          Here is the same function, but following the C-style pattern:
271: (4)          >>> def iter_add(x, y, out=None):
272: (4)              ...      addop = np.add
273: (4)              ...      it = np.nditer([x, y, out], [],
274: (4)                          ...                  [['readonly']], ['readonly'],
['writeonly','allocate']))
275: (4)              ...      with it:
276: (4)                  ...          while not it.finished:
277: (4)                      ...              addop(it[0], it[1], out=it[2])
278: (4)                      ...              it.iternext()
279: (4)                  ...          return it.operands[2]
280: (4)          Here is an example outer product function:
281: (4)          >>> def outer_it(x, y, out=None):
282: (4)              ...      mulop = np.multiply
283: (4)              ...      it = np.nditer([x, y, out], ['external_loop'],
284: (4)                          ...                  [['readonly']], ['readonly'],
['writeonly', 'allocate']],
285: (4)                          ...                  op_axes=[list(range(x.ndim)) + [-1] * y.ndim,
286: (4)  ...                               [-1] * x.ndim + list(range(y.ndim)),
287: (4)  ...                               None])
288: (4)              ...      with it:
289: (4)                  ...          for (a, b, c) in it:
290: (4)                      ...              mulop(a, b, out=c)
291: (4)                      ...          return it.operands[2]
292: (4)          >>> a = np.arange(2)+1
293: (4)          >>> b = np.arange(3)+1
294: (4)          >>> outer_it(a,b)
295: (4)          array([[1, 2, 3],
296: (11)             [2, 4, 6]])
297: (4)          Here is an example function which operates like a "lambda" ufunc:
298: (4)          >>> def luf(lamdaexpr, *args, **kwargs):
299: (4)              ...      '''luf(lambdaexpr, op1, ..., opn, out=None, order='K',
casting='safe', buffersize=0)'''
300: (4)                  ...      nargs = len(args)
301: (4)                  ...      op = (kwargs.get('out',None),) + args
302: (4)                  ...      it = np.nditer(op, ['buffered','external_loop'],
303: (4)                          ...                  [['writeonly','allocate','no_broadcast']] +
304: (4)   ...                  [['readonly','nbo','aligned']]*nargs,
305: (4)   ...                  order=kwargs.get('order','K'),
306: (4)   ...                  casting=kwargs.get('casting','safe'),
307: (4)   ...                  buffersize=kwargs.get('buffersize',0))
308: (4)                  ...      while not it.finished:
309: (4)                      ...          it[0] = lamdaexpr(*it[1:])
310: (4)                      ...          it.iternext()
311: (4)                  ...          return it.operands[0]
312: (4)          >>> a = np.arange(5)
313: (4)          >>> b = np.ones(5)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

314: (4)          >>> luf(lambda i,j:i*i + j/2, a, b)
315: (4)          array([ 0.5,  1.5,  4.5,  9.5, 16.5])
316: (4)          If operand flags ``"writeonly"`` or ``"readwrite"`` are used the
317: (4)          operands may be views into the original data with the
318: (4)          ``WRITEREADBACKIFCOPY`` flag. In this case `nditer` must be used as a
319: (4)          context manager or the `nditer.close` method must be called before
320: (4)          using the result. The temporary data will be written back to the
321: (4)          original data when the `__exit__` function is called but not before:
322: (4)          >>> a = np.arange(6, dtype='i4')[::-2]
323: (4)          >>> with np.nditer(a, []),
324: (4)              ...      [['writeonly', 'updateifcopy']],
325: (4)              ...      casting='unsafe',
326: (4)              ...      op_dtypes=[np.dtype('f4')]) as i:
327: (4)              ...      x = i.operands[0]
328: (4)              ...      x[:] = [-1, -2, -3]
329: (4)              ...      # a still unchanged here
330: (4)          >>> a, x
331: (4)          (array([-1, -2, -3], dtype=int32), array([-1., -2., -3.], dtype=float32))
332: (4)          It is important to note that once the iterator is exited, dangling
333: (4)          references (like `x` in the example) may or may not share data with
334: (4)          the original data `a`. If writeback semantics were active, i.e. if
335: (4)          `x.base.flags.writebackifcopy` is `True`, then exiting the iterator
336: (4)          will sever the connection between `x` and `a`, writing to `x` will
337: (4)          no longer write to `a`. If writeback semantics are not active, then
338: (4)          `x.data` will still point at some part of `a.data`, and writing to
339: (4)          one will affect the other.
340: (4)          Context management and the `close` method appeared in version 1.15.0.
341: (4)          """
342: (0)          add_newdoc('numpy.core', 'nditer', ('copy',
343: (4)              """
344: (4)              copy()
345: (4)              Get a copy of the iterator in its current state.
346: (4)              Examples
347: (4)              -----
348: (4)              >>> x = np.arange(10)
349: (4)              >>> y = x + 1
350: (4)              >>> it = np.nditer([x, y])
351: (4)              >>> next(it)
352: (4)              (array(0), array(1))
353: (4)              >>> it2 = it.copy()
354: (4)              >>> next(it2)
355: (4)              (array(1), array(2))
356: (4)              """
357: (0)          add_newdoc('numpy.core', 'nditer', ('operands',
358: (4)              """
359: (4)              operands['Slice']
360: (4)              The array(s) to be iterated over. Valid only before the iterator is
closed.
361: (4)              """
362: (0)          add_newdoc('numpy.core', 'nditer', ('debug_print',
363: (4)              """
364: (4)              debug_print()
365: (4)              Print the current state of the `nditer` instance and debug info to stdout.
366: (4)              """
367: (0)          add_newdoc('numpy.core', 'nditer', ('enable_external_loop',
368: (4)              """
369: (4)              enable_external_loop()
370: (4)              When the "external_loop" was not used during construction, but
371: (4)              is desired, this modifies the iterator to behave as if the flag
372: (4)              was specified.
373: (4)              """
374: (0)          add_newdoc('numpy.core', 'nditer', ('iternext',
375: (4)              """
376: (4)              iternext()
377: (4)              Check whether iterations are left, and perform a single internal iteration
378: (4)              without returning the result. Used in the C-style pattern do-while
379: (4)              pattern. For an example, see `nditer`.
380: (4)              Returns
381: (4)              -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

382: (4)         iternext : bool
383: (8)           Whether or not there are iterations left.
384: (4)
385: (0)         add_newdoc('numpy.core', 'nditer', ('remove_axis',
386: (4)           """
387: (4)             remove_axis(i, /)
388: (4)             Removes axis `i` from the iterator. Requires that the flag "multi_index"
389: (4)             be enabled.
390: (4)             """
391: (0)         add_newdoc('numpy.core', 'nditer', ('remove_multi_index',
392: (4)           """
393: (4)             remove_multi_index()
394: (4)             When the "multi_index" flag was specified, this removes it, allowing
395: (4)             the internal iteration structure to be optimized further.
396: (4)             """
397: (0)         add_newdoc('numpy.core', 'nditer', ('reset',
398: (4)           """
399: (4)             reset()
400: (4)             Reset the iterator to its initial state.
401: (4)             """
402: (0)         add_newdoc('numpy.core', 'nested_iters',
403: (4)           """
404: (4)             nested_iters(op, axes, flags=None, op_flags=None, op_dtypes=None, \
405: (4)               order="K", casting="safe", buffersize=0)
406: (4)             Create nditers for use in nested loops
407: (4)             Create a tuple of `nditer` objects which iterate in nested loops over
408: (4)             different axes of the op argument. The first iterator is used in the
409: (4)             outermost loop, the last in the innermost loop. Advancing one will change
410: (4)             the subsequent iterators to point at its new element.
411: (4)             Parameters
412: (4)             -----
413: (4)             op : ndarray or sequence of array_like
414: (8)               The array(s) to iterate over.
415: (4)             axes : list of list of int
416: (8)               Each item is used as an "op_axes" argument to an nditer
417: (4)               flags, op_flags, op_dtypes, order, casting, buffersize (optional)
418: (8)                 See `nditer` parameters of the same name
419: (4)             Returns
420: (4)             -----
421: (4)             iter : tuple of nditer
422: (8)               An nditer for each item in `axes`, outermost first
423: (4)             See Also
424: (4)             -----
425: (4)             nditer
426: (4)             Examples
427: (4)             -----
428: (4)             Basic usage. Note how y is the "flattened" version of
429: (4)             [a[:, 0, :], a[:, 1, 0], a[:, 2, :]] since we specified
430: (4)             the first iter's axes as [1]
431: (4)             >>> a = np.arange(12).reshape(2, 3, 2)
432: (4)             >>> i, j = np.nested_iters(a, [[1], [0, 2]], flags=["multi_index"])
433: (4)             >>> for x in i:
434: (4)               ...     print(i.multi_index)
435: (4)               ...     for y in j:
436: (4)                 ...         print('', j.multi_index, y)
437: (4)             (0,)
438: (5)               (0, 0) 0
439: (5)               (0, 1) 1
440: (5)               (1, 0) 6
441: (5)               (1, 1) 7
442: (4)               (1,)
443: (5)               (0, 0) 2
444: (5)               (0, 1) 3
445: (5)               (1, 0) 8
446: (5)               (1, 1) 9
447: (4)               (2,)
448: (5)               (0, 0) 4
449: (5)               (0, 1) 5
450: (5)               (1, 0) 10

```

```

451: (5)          (1, 1) 11
452: (4)          """
453: (0)          add_newdoc('numpy.core', 'nditer', ('close',
454: (4)          """"
455: (4)          close()
456: (4)          Resolve all writeback semantics in writeable operands.
457: (4)          .. versionadded:: 1.15.0
458: (4)          See Also
459: (4)          -----
460: (4)          :ref:`nditer-context-manager`"
461: (4)          """")
462: (0)          add_newdoc('numpy.core', 'broadcast',
463: (4)          """
464: (4)          Produce an object that mimics broadcasting.
465: (4)          Parameters
466: (4)          -----
467: (4)          in1, in2, ... : array_like
468: (8)          Input parameters.
469: (4)          Returns
470: (4)          -----
471: (4)          b : broadcast object
472: (8)          Broadcast the input parameters against one another, and
473: (8)          return an object that encapsulates the result.
474: (8)          Amongst others, it has ``shape`` and ``nd`` properties, and
475: (8)          may be used as an iterator.
476: (4)          See Also
477: (4)          -----
478: (4)          broadcast_arrays
479: (4)          broadcast_to
480: (4)          broadcast_shapes
481: (4)          Examples
482: (4)          -----
483: (4)          Manually adding two vectors, using broadcasting:
484: (4)          >>> x = np.array([[1], [2], [3]])
485: (4)          >>> y = np.array([4, 5, 6])
486: (4)          >>> b = np.broadcast(x, y)
487: (4)          >>> out = np.empty(b.shape)
488: (4)          >>> out.flat = [u+v for (u,v) in b]
489: (4)          >>> out
490: (4)          array([[5.,  6.,  7.],
491: (11)           [6.,  7.,  8.],
492: (11)           [7.,  8.,  9.]])
493: (4)          Compare against built-in broadcasting:
494: (4)          >>> x + y
495: (4)          array([[5, 6, 7],
496: (11)           [6, 7, 8],
497: (11)           [7, 8, 9]])
498: (4)          """
499: (0)          add_newdoc('numpy.core', 'broadcast', ('index',
500: (4)          """
501: (4)          current index in broadcasted result
502: (4)          Examples
503: (4)          -----
504: (4)          >>> x = np.array([[1], [2], [3]])
505: (4)          >>> y = np.array([4, 5, 6])
506: (4)          >>> b = np.broadcast(x, y)
507: (4)          >>> b.index
508: (4)          0
509: (4)          >>> next(b), next(b), next(b)
510: (4)          ((1, 4), (1, 5), (1, 6))
511: (4)          >>> b.index
512: (4)          3
513: (4)          """))
514: (0)          add_newdoc('numpy.core', 'broadcast', ('iters',
515: (4)          """
516: (4)          tuple of iterators along ``self``'s "components."
517: (4)          Returns a tuple of `numpy.flatiter` objects, one for each "component"
518: (4)          of ``self``.
519: (4)          See Also

```

```
520: (4)      -----
521: (4)      numpy.flatiter
522: (4)      Examples
523: (4)      -----
524: (4)      >>> x = np.array([1, 2, 3])
525: (4)      >>> y = np.array([[4], [5], [6]])
526: (4)      >>> b = np.broadcast(x, y)
527: (4)      >>> row, col = b.iters
528: (4)      >>> next(row), next(col)
529: (4)      (1, 4)
530: (4)      ""))
531: (0)      add_newdoc('numpy.core', 'broadcast', ('ndim',
532: (4)      """
533: (4)          Number of dimensions of broadcasted result. Alias for `nd`.
534: (4)          .. versionadded:: 1.12.0
535: (4)          Examples
536: (4)          -----
537: (4)          >>> x = np.array([1, 2, 3])
538: (4)          >>> y = np.array([[4], [5], [6]])
539: (4)          >>> b = np.broadcast(x, y)
540: (4)          >>> b.ndim
541: (4)          2
542: (4)          ""))
543: (0)      add_newdoc('numpy.core', 'broadcast', ('nd',
544: (4)      """
545: (4)          Number of dimensions of broadcasted result. For code intended for NumPy
546: (4)          1.12.0 and later the more consistent `ndim` is preferred.
547: (4)          Examples
548: (4)          -----
549: (4)          >>> x = np.array([1, 2, 3])
550: (4)          >>> y = np.array([[4], [5], [6]])
551: (4)          >>> b = np.broadcast(x, y)
552: (4)          >>> b.nd
553: (4)          2
554: (4)          ""))
555: (0)      add_newdoc('numpy.core', 'broadcast', ('numiter',
556: (4)      """
557: (4)          Number of iterators possessed by the broadcasted result.
558: (4)          Examples
559: (4)          -----
560: (4)          >>> x = np.array([1, 2, 3])
561: (4)          >>> y = np.array([[4], [5], [6]])
562: (4)          >>> b = np.broadcast(x, y)
563: (4)          >>> b.numiter
564: (4)          2
565: (4)          ""))
566: (0)      add_newdoc('numpy.core', 'broadcast', ('shape',
567: (4)      """
568: (4)          Shape of broadcasted result.
569: (4)          Examples
570: (4)          -----
571: (4)          >>> x = np.array([1, 2, 3])
572: (4)          >>> y = np.array([[4], [5], [6]])
573: (4)          >>> b = np.broadcast(x, y)
574: (4)          >>> b.shape
575: (4)          (3, 3)
576: (4)          ""))
577: (0)      add_newdoc('numpy.core', 'broadcast', ('size',
578: (4)      """
579: (4)          Total size of broadcasted result.
580: (4)          Examples
581: (4)          -----
582: (4)          >>> x = np.array([1, 2, 3])
583: (4)          >>> y = np.array([[4], [5], [6]])
584: (4)          >>> b = np.broadcast(x, y)
585: (4)          >>> b.size
586: (4)          9
587: (4)          ""))
588: (0)      add_newdoc('numpy.core', 'broadcast', ('reset',
```

```

589: (4)
590: (4)
591: (4)
592: (4)
593: (4)
594: (4)
595: (4)
596: (4)
597: (4)
598: (4)
599: (4)
600: (4)
601: (4)
602: (4)
603: (4)
604: (4)
605: (4)
606: (4)
607: (4)
608: (4)
609: (4)
610: (4)
611: (4)
612: (4)
613: (0)
614: (4)
615: (4)
616: (10)
617: (4)
618: (4)
619: (4)
620: (4)
621: (8)
622: (8)
623: (8)
624: (8)
625: (4)
626: (8)
use
627: (8)
promotion
628: (8)
629: (4)
copy : bool, optional
630: (8)
631: (8)
632: (8)
633: (8)
634: (4)
order : {'K', 'A', 'C', 'F'}, optional
635: (8)
636: (8)
637: (8)
638: (8)
639: (8)
640: (8)
641: (8)
642: (8)
643: (8)
644: (8)
645: (8)
646: (8)
647: (8)
is
648: (8)
649: (8)
650: (4)
651: (8)
652: (8)
653: (4)
654: (8)

    """
    reset()
    Reset the broadcasted result's iterator(s).
    Parameters
    -----
    None
    Returns
    -----
    None
    Examples
    -----
    >>> x = np.array([1, 2, 3])
    >>> y = np.array([[4], [5], [6]])
    >>> b = np.broadcast(x, y)
    >>> b.index
    0
    >>> next(b), next(b), next(b)
    ((1, 4), (2, 4), (3, 4))
    >>> b.index
    3
    >>> b.reset()
    >>> b.index
    0
    """))

add_newdoc('numpy.core.multiarray', 'array',
"""
array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
      like=None)
Create an array.
Parameters
-----
object : array_like
    An array, any object exposing the array interface, an object whose
    ``__array__`` method returns an array, or any (nested) sequence.
    If object is a scalar, a 0-dimensional array containing object is
    returned.
dtype : data-type, optional
    The desired data-type for the array. If not given, NumPy will try to
    a default ``dtype`` that can represent the values (by applying
    rules when necessary.)
copy : bool, optional
    If true (default), then the object is copied. Otherwise, a copy will
    only be made if ``__array__`` returns a copy, if obj is a nested
    sequence, or if a copy is needed to satisfy any of the other
    requirements (``dtype``, ``order``, etc.).
order : {'K', 'A', 'C', 'F'}, optional
    Specify the memory layout of the array. If object is not an array, the
    newly created array will be in C order (row major) unless 'F' is
    specified, in which case it will be in Fortran order (column major).
    If object is an array the following holds.
=====
order no copy                  copy=True
=====
'K'   unchanged F & C order preserved, otherwise most similar order
'A'   unchanged F order if input is F and not C, otherwise C order
'C'   C order   C order
'F'   F order   F order
=====
When ``copy=False`` and a copy is made for other reasons, the result
is
    the same as if ``copy=True``, with some exceptions for 'A', see the
    Notes section. The default order is 'K'.
subok : bool, optional
    If True, then sub-classes will be passed-through, otherwise
    the returned array will be forced to be a base-class array (default).
ndmin : int, optional
    Specifies the minimum number of dimensions that the resulting

```

```

655: (8)                                array should have. Ones will be prepended to the shape as
656: (8)                                needed to meet this requirement.
657: (4)                                ${ARRAY_FUNCTION_LIKE}
658: (8)                                .. versionadded:: 1.20.0
659: (4)                                Returns
660: (4)                                -----
661: (4)                                out : ndarray
662: (8)                                An array object satisfying the specified requirements.
663: (4)                                See Also
664: (4)                                -----
665: (4)                                empty_like : Return an empty array with shape and type of input.
666: (4)                                ones_like : Return an array of ones with shape and type of input.
667: (4)                                zeros_like : Return an array of zeros with shape and type of input.
668: (4)                                full_like : Return a new array with shape of input filled with value.
669: (4)                                empty : Return a new uninitialized array.
670: (4)                                ones : Return a new array setting values to one.
671: (4)                                zeros : Return a new array setting values to zero.
672: (4)                                full : Return a new array of given shape filled with value.
673: (4)                                Notes
674: (4)                                -----
675: (4)                                When order is 'A' and ``object`` is an array in neither 'C' nor 'F' order,
676: (4)                                and a copy is forced by a change in dtype, then the order of the result is
677: (4)                                not necessarily 'C' as expected. This is likely a bug.
678: (4)                                Examples
679: (4)                                -----
680: (4)                                >>> np.array([1, 2, 3])
681: (4)                                array([1, 2, 3])
682: (4)                                Upcasting:
683: (4)                                >>> np.array([1, 2, 3.0])
684: (4)                                array([ 1.,  2.,  3.])
685: (4)                                More than one dimension:
686: (4)                                >>> np.array([[1, 2], [3, 4]])
687: (4)                                array([[1, 2],
688: (11)                               [3, 4]])
689: (4)                                Minimum dimensions 2:
690: (4)                                >>> np.array([1, 2, 3], ndmin=2)
691: (4)                                array([[1, 2, 3]])
692: (4)                                Type provided:
693: (4)                                >>> np.array([1, 2, 3], dtype=complex)
694: (4)                                array([ 1.+0.j,  2.+0.j,  3.+0.j])
695: (4)                                Data-type consisting of more than one element:
696: (4)                                >>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
697: (4)                                >>> x['a']
698: (4)                                array([1, 3])
699: (4)                                Creating an array from sub-classes:
700: (4)                                >>> np.array(np.mat('1 2; 3 4'))
701: (4)                                array([[1, 2],
702: (11)                               [3, 4]])
703: (4)                                >>> np.array(np.mat('1 2; 3 4'), subok=True)
704: (4)                                matrix([[1, 2],
705: (12)                               [3, 4]])
706: (4)                                """".replace(
707: (8)                                  "${ARRAY_FUNCTION_LIKE}",
708: (8)                                  array_function_like_doc,
709: (4)                                ))
710: (0)                                add_newdoc('numpy.core.multiarray', 'asarray',
711: (4)                                """
712: (4)                                asarray(a, dtype=None, order=None, *, like=None)
713: (4)                                Convert the input to an array.
714: (4)                                Parameters
715: (4)                                -----
716: (4)                                a : array_like
717: (8)                                Input data, in any form that can be converted to an array. This
718: (8)                                includes lists, lists of tuples, tuples, tuples of tuples, tuples
719: (8)                                of lists and ndarrays.
720: (4)                                dtype : data-type, optional
721: (8)                                By default, the data-type is inferred from the input data.
722: (4)                                order : {'C', 'F', 'A', 'K'}, optional
723: (8)                                Memory layout. 'A' and 'K' depend on the order of input array a.

```

```

724: (8)           'C' row-major (C-style),
725: (8)           'F' column-major (Fortran-style) memory representation.
726: (8)           'A' (any) means 'F' if `a` is Fortran contiguous, 'C' otherwise
727: (8)           'K' (keep) preserve input order
728: (8)           Defaults to 'K'.
729: (4)           ${ARRAY_FUNCTION_LIKE}
730: (8)           .. versionadded:: 1.20.0
731: (4)           Returns
732: (4)
733: (4)           out : ndarray
734: (8)           Array interpretation of `a`. No copy is performed if the input
735: (8)           is already an ndarray with matching dtype and order. If `a` is a
736: (8)           subclass of ndarray, a base class ndarray is returned.
737: (4)           See Also
738: (4)
739: (4)           asanyarray : Similar function which passes through subclasses.
740: (4)           ascontiguousarray : Convert input to a contiguous array.
741: (4)           asfarray : Convert input to a floating point ndarray.
742: (4)           asfortranarray : Convert input to an ndarray with column-major
743: (21)             memory order.
744: (4)           asarray_chkfinite : Similar function which checks input for NaNs and Infs.
745: (4)           fromiter : Create an array from an iterator.
746: (4)           fromfunction : Construct an array by executing a function on grid
747: (19)             positions.
748: (4)           Examples
749: (4)
750: (4)           Convert a list into an array:
751: (4)           >>> a = [1, 2]
752: (4)           >>> np.asarray(a)
753: (4)           array([1, 2])
754: (4)           Existing arrays are not copied:
755: (4)           >>> a = np.array([1, 2])
756: (4)           >>> np.asarray(a) is a
757: (4)           True
758: (4)           If `dtype` is set, array is copied only if dtype does not match:
759: (4)           >>> a = np.array([1, 2], dtype=np.float32)
760: (4)           >>> np.asarray(a, dtype=np.float32) is a
761: (4)           True
762: (4)           >>> np.asarray(a, dtype=np.float64) is a
763: (4)           False
764: (4)           Contrary to `asanyarray`, ndarray subclasses are not passed through:
765: (4)           >>> issubclass(np.recarray, np.ndarray)
766: (4)           True
767: (4)           >>> a = np.array([(1.0, 2), (3.0, 4)], dtype='f4,i4').view(np.recarray)
768: (4)           >>> np.asarray(a) is a
769: (4)           False
770: (4)           >>> np.asanyarray(a) is a
771: (4)           True
772: (4)           """".replace(
773: (8)             "${ARRAY_FUNCTION_LIKE}",
774: (8)             array_function_like_doc,
775: ())
776: (0)           add_newdoc('numpy.core.multiarray', 'asanyarray',
777: (4)             """
778: (4)               asanyarray(a, dtype=None, order=None, *, like=None)
779: (4)               Convert the input to an ndarray, but pass ndarray subclasses through.
780: (4)               Parameters
781: (4)
782: (4)               a : array_like
783: (8)                 Input data, in any form that can be converted to an array. This
784: (8)                 includes scalars, lists, lists of tuples, tuples, tuples of tuples,
785: (8)                 tuples of lists, and ndarrays.
786: (4)               dtype : data-type, optional
787: (8)                 By default, the data-type is inferred from the input data.
788: (4)               order : {'C', 'F', 'A', 'K'}, optional
789: (8)                 Memory layout. 'A' and 'K' depend on the order of input array a.
790: (8)                 'C' row-major (C-style),
791: (8)                 'F' column-major (Fortran-style) memory representation.
792: (8)                 'A' (any) means 'F' if `a` is Fortran contiguous, 'C' otherwise

```

```

793: (8)          'K' (keep) preserve input order
794: (8)          Defaults to 'C'.
795: (4)          ${ARRAY_FUNCTION_LIKE}
796: (8)          .. versionadded:: 1.20.0
797: (4)          Returns
798: (4)          -----
799: (4)          out : ndarray or an ndarray subclass
800: (8)          Array interpretation of `a`. If `a` is an ndarray or a subclass
801: (8)          of ndarray, it is returned as-is and no copy is performed.
802: (4)          See Also
803: (4)          -----
804: (4)          asarray : Similar function which always returns ndarrays.
805: (4)          ascontiguousarray : Convert input to a contiguous array.
806: (4)          asfarray : Convert input to a floating point ndarray.
807: (4)          asfortranarray : Convert input to an ndarray with column-major
808: (21)         memory order.
809: (4)          asarray_chkfinite : Similar function which checks input for NaNs and
810: (24)         Infs.
811: (4)          fromiter : Create an array from an iterator.
812: (4)          fromfunction : Construct an array by executing a function on grid
813: (19)         positions.

814: (4)          Examples
815: (4)          -----
816: (4)          Convert a list into an array:
817: (4)          >>> a = [1, 2]
818: (4)          >>> np.asanyarray(a)
819: (4)          array([1, 2])
820: (4)          Instances of `ndarray` subclasses are passed through as-is:
821: (4)          >>> a = np.array([(1.0, 2), (3.0, 4)], dtype='f4,i4').view(np.recarray)
822: (4)          >>> np.asanyarray(a) is a
823: (4)          True
824: (4)          """".replace(
825: (8)            "${ARRAY_FUNCTION_LIKE}",
826: (8)            array_function_like_doc,
827: (4)          ))
828: (0)          add_newdoc('numpy.core.multiarray', 'ascontiguousarray',
829: (4)          """
830: (4)          ascontiguousarray(a, dtype=None, *, like=None)
831: (4)          Return a contiguous array (ndim >= 1) in memory (C order).
832: (4)          Parameters
833: (4)          -----
834: (4)          a : array_like
835: (8)          Input array.
836: (4)          dtype : str or dtype object, optional
837: (8)          Data-type of returned array.
838: (4)          ${ARRAY_FUNCTION_LIKE}
839: (8)          .. versionadded:: 1.20.0
840: (4)          Returns
841: (4)          -----
842: (4)          out : ndarray
843: (8)          Contiguous array of same shape and content as `a`, with type `dtype`
844: (8)          if specified.
845: (4)          See Also
846: (4)          -----
847: (4)          asfortranarray : Convert input to an ndarray with column-major
848: (21)         memory order.
849: (4)          require : Return an ndarray that satisfies requirements.
850: (4)          ndarray.flags : Information about the memory layout of the array.
851: (4)          Examples
852: (4)          -----
853: (4)          Starting with a Fortran-contiguous array:
854: (4)          >>> x = np.ones((2, 3), order='F')
855: (4)          >>> x.flags['F_CONTIGUOUS']
856: (4)          True
857: (4)          Calling ``ascontiguousarray`` makes a C-contiguous copy:
858: (4)          >>> y = np.ascontiguousarray(x)
859: (4)          >>> y.flags['C_CONTIGUOUS']
860: (4)          True
861: (4)          >>> np.may_share_memory(x, y)

```

```

862: (4)           False
863: (4)           Now, starting with a C-contiguous array:
864: (4)           >>> x = np.ones((2, 3), order='C')
865: (4)           >>> x.flags['C_CONTIGUOUS']
866: (4)           True
867: (4)           Then, calling ``ascontiguousarray`` returns the same object:
868: (4)           >>> y = np.ascontiguousarray(x)
869: (4)           >>> x is y
870: (4)           True
871: (4)           Note: This function returns an array with at least one-dimension (1-d)
872: (4)           so it will not preserve 0-d arrays.
873: (4)           """".replace(
874: (8)             "${ARRAY_FUNCTION_LIKE}",
875: (8)             array_function_like_doc,
876: (4)         ))
877: (0)         add_newdoc('numpy.core.multiarray', 'asfortranarray',
878: (4)         """
879: (4)             asfortranarray(a, dtype=None, *, like=None)
880: (4)             Return an array (ndim >= 1) laid out in Fortran order in memory.
881: (4)             Parameters
882: (4)             -----
883: (4)             a : array_like
884: (8)               Input array.
885: (4)             dtype : str or dtype object, optional
886: (8)               By default, the data-type is inferred from the input data.
887: (4)             ${ARRAY_FUNCTION_LIKE}
888: (8)               .. versionadded:: 1.20.0
889: (4)             Returns
890: (4)             -----
891: (4)             out : ndarray
892: (8)               The input `a` in Fortran, or column-major, order.
893: (4)             See Also
894: (4)             -----
895: (4)             ascontiguousarray : Convert input to a contiguous (C order) array.
896: (4)             asanyarray : Convert input to an ndarray with either row or
897: (8)               column-major memory order.
898: (4)             require : Return an ndarray that satisfies requirements.
899: (4)             ndarray.flags : Information about the memory layout of the array.
900: (4)             Examples
901: (4)             -----
902: (4)             Starting with a C-contiguous array:
903: (4)             >>> x = np.ones((2, 3), order='C')
904: (4)             >>> x.flags['C_CONTIGUOUS']
905: (4)             True
906: (4)             Calling ``asfortranarray`` makes a Fortran-contiguous copy:
907: (4)             >>> y = np.asfortranarray(x)
908: (4)             >>> y.flags['F_CONTIGUOUS']
909: (4)             True
910: (4)             >>> np.may_share_memory(x, y)
911: (4)             False
912: (4)             Now, starting with a Fortran-contiguous array:
913: (4)             >>> x = np.ones((2, 3), order='F')
914: (4)             >>> x.flags['F_CONTIGUOUS']
915: (4)             True
916: (4)             Then, calling ``asfortranarray`` returns the same object:
917: (4)             >>> y = np.asfortranarray(x)
918: (4)             >>> x is y
919: (4)             True
920: (4)             Note: This function returns an array with at least one-dimension (1-d)
921: (4)             so it will not preserve 0-d arrays.
922: (4)             """".replace(
923: (8)               "${ARRAY_FUNCTION_LIKE}",
924: (8)               array_function_like_doc,
925: (4)         ))
926: (0)         add_newdoc('numpy.core.multiarray', 'empty',
927: (4)         """
928: (4)             empty(shape, dtype=float, order='C', *, like=None)
929: (4)             Return a new array of given shape and type, without initializing entries.
930: (4)             Parameters

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

931: (4)
932: (4)
933: (8)
934: (4)
935: (8)
936: (8)
937: (4)
938: (8)
939: (8)
940: (8)
941: (4)
942: (8)
943: (4)
944: (4)
945: (4)
946: (8)
947: (8)
948: (4)
949: (4)
950: (4)
951: (4)
952: (4)
953: (4)
954: (4)
955: (4)
956: (4)
957: (4)
958: (4)
959: (4)
960: (4)
961: (4)
962: (4)
963: (4)
964: (11)
965: (4)
966: (4)
967: (11)
968: (4)
969: (8)
970: (8)
971: (4)
972: (0)
973: (4)
974: (4)
975: (4)
976: (4)
977: (4)
978: (4)
979: (4)
980: (4)
981: (4)
982: (0)
983: (4)
984: (4)
985: (4)
986: (4)
987: (4)
988: (4)
989: (8)
990: (4)
991: (8)
992: (8)
993: (4)
994: (8)
995: (8)
996: (8)
997: (4)
998: (8)
999: (4)

-----  

shape : int or tuple of int  

    Shape of the empty array, e.g., ``(2, 3)`` or ``2``.  

dtype : data-type, optional  

    Desired output data-type for the array, e.g., `numpy.int8`. Default is  

`numpy.float64`.  

order : {'C', 'F'}, optional, default: 'C'  

    Whether to store multi-dimensional data in row-major  

(C-style) or column-major (Fortran-style) order in  

memory.  

${ARRAY_FUNCTION_LIKE}  

.. versionadded:: 1.20.0  

>Returns  

-----  

out : ndarray  

    Array of uninitialized (arbitrary) data of the given shape, dtype, and  

order. Object arrays will be initialized to None.  

See Also  

-----  

empty_like : Return an empty array with shape and type of input.  

ones : Return a new array setting values to one.  

zeros : Return a new array setting values to zero.  

full : Return a new array of given shape filled with value.  

Notes  

-----  

`empty`, unlike `zeros`, does not set the array values to zero,  

and may therefore be marginally faster. On the other hand, it requires  

the user to manually set all the values in the array, and should be  

used with caution.  

Examples  

-----  

>>> np.empty([2, 2])  

array([[ -9.74499359e+001,   6.69583040e-309],  

       [  2.13182611e-314,   3.06959433e-309]]) #uninitialized  

>>> np.empty([2, 2], dtype=int)  

array([[ -1073741821, -1067949133],  

       [  496041986,   19249760]]) #uninitialized  

""".replace(  

    "${ARRAY_FUNCTION_LIKE}",  

    array_function_like_doc,  

))  

add_newdoc('numpy.core.multiarray', 'scalar',  

"""  

scalar(dtype, obj)  

Return a new scalar array of the given type initialized with obj.  

This function is meant mainly for pickle support. `dtype` must be a  

valid data-type descriptor. If `dtype` corresponds to an object  

descriptor, then `obj` can be any object, otherwise `obj` must be a  

string. If `obj` is not given, it will be interpreted as None for object  

type and as zeros for all other types.  

""")  

add_newdoc('numpy.core.multiarray', 'zeros',  

"""  

zeros(shape, dtype=float, order='C', *, like=None)  

Return a new array of given shape and type, filled with zeros.  

Parameters  

-----  

shape : int or tuple of ints  

    Shape of the new array, e.g., ``(2, 3)`` or ``2``.  

dtype : data-type, optional  

    The desired data-type for the array, e.g., `numpy.int8`. Default is  

`numpy.float64`.  

order : {'C', 'F'}, optional, default: 'C'  

    Whether to store multi-dimensional data in row-major  

(C-style) or column-major (Fortran-style) order in  

memory.  

${ARRAY_FUNCTION_LIKE}  

.. versionadded:: 1.20.0  

>Returns

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1000: (4)
1001: (4)
1002: (8)
1003: (4)
1004: (4)
1005: (4)
1006: (4)
1007: (4)
1008: (4)
1009: (4)
1010: (4)
1011: (4)
1012: (4)
1013: (4)
1014: (4)
1015: (4)
1016: (4)
1017: (11)
1018: (4)
1019: (4)
1020: (4)
1021: (11)
1022: (4)
1023: (4)
1024: (10)
1025: (4)
1026: (8)
1027: (8)
1028: (4)
1029: (0)
1030: (4)
1031: (4)
1032: (4)
1033: (4)
1034: (0)
1035: (4)
1036: (4)
1037: (4)
1038: (4)
1039: (4)
1040: (4)
1041: (8)
1042: (4)
1043: (8)
1044: (8)
are
1045: (8)
1046: (8)
1047: (12)
1048: (4)
1049: (8)
1050: (8)
1051: (8)
1052: (4)
1053: (8)
1054: (8)
1055: (8)
1056: (12)
1057: (12)
1058: (12)
1059: (12)
1060: (12)
1061: (12)
1062: (12)
1063: (4)
1064: (8)
1065: (4)
1066: (4)
1067: (4)

      -----
      out : ndarray
          Array of zeros with the given shape, dtype, and order.
      See Also
      -----
      zeros_like : Return an array of zeros with shape and type of input.
      empty : Return a new uninitialized array.
      ones : Return a new array setting values to one.
      full : Return a new array of given shape filled with value.
      Examples
      -----
      >>> np.zeros(5)
      array([ 0.,  0.,  0.,  0.,  0.])
      >>> np.zeros((5,), dtype=int)
      array([0, 0, 0, 0, 0])
      >>> np.zeros((2, 1))
      array([[ 0.],
             [ 0.]])
      >>> s = (2,2)
      >>> np.zeros(s)
      array([[ 0.,  0.],
             [ 0.,  0.]])
      >>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
      array([(0, 0), (0, 0)],
            dtype=[('x', '<i4'), ('y', '<i4')])
      """.
      replace(
          "${ARRAY_FUNCTION_LIKE}",
          array_function_like_doc,
      ))
      add_newdoc('numpy.core.multiarray', 'set_typeDict',
      """.
      set_typeDict(dict)
      Set the internal dictionary that can look up an array type using a
      registered code.
      """")
      add_newdoc('numpy.core.multiarray', 'fromstring',
      """
      fromstring(string, dtype=float, count=-1, *, sep, like=None)
      A new 1-D array initialized from text data in a string.
      Parameters
      -----
      string : str
          A string containing the data.
      dtype : data-type, optional
          The data type of the array; default: float. For binary input data,
          the data must be in exactly this format. Most builtin numeric types
          supported and extension types may be supported.
      .. versionadded:: 1.18.0
          Complex dtypes.
      count : int, optional
          Read this number of `dtype` elements from the data. If this is
          negative (the default), the count will be determined from the
          length of the data.
      sep : str, optional
          The string separating numbers in the data; extra whitespace between
          elements is also ignored.
      .. deprecated:: 1.14
          Passing ``sep='''', the default, is deprecated since it will
          trigger the deprecated binary mode of this function. This mode
          interprets `string` as binary bytes, rather than ASCII text with
          decimal numbers, an operation which is better spelt
          ``frombuffer(string, dtype, count)``. If `string` contains unicode
          text, the binary mode of `fromstring` will first encode it into
          bytes using utf-8, which will not produce sane results.
      ${ARRAY_FUNCTION_LIKE}
      .. versionadded:: 1.20.0
      Returns
      -----
      arr : ndarray
  
```

```

1068: (8)           The constructed array.
1069: (4)           Raises
1070: (4)
1071: (4)
1072: (8)           -----
1073: (8)           ValueError
1074: (4)           If the string is not the correct size to satisfy the requested
1075: (4)           `dtype` and `count`.
1076: (4)           See Also
1077: (4)
1078: (4)
1079: (4)           -----
1080: (4)           frombuffer, fromfile, fromiter
1081: (4)           Examples
1082: (4)
1083: (4)
1084: (8)           -----
1085: (8)           ${ARRAY_FUNCTION_LIKE},
1086: (4)           array_function_like_doc,
1087: (0)           ))
1088: (4)           add_newdoc('numpy.core.multiarray', 'compare_chararrays',
1089: (4)           """
1090: (4)           compare_chararrays(a1, a2, cmp, rstrip)
1091: (4)           Performs element-wise comparison of two string arrays using the
1092: (4)           comparison operator specified by `cmp_op`.
1093: (4)           Parameters
1094: (4)           -----
1095: (8)           a1, a2 : array_like
1096: (4)           Arrays to be compared.
1097: (8)           cmp : {"<", "<=", "==", ">=", ">", "!="}
1098: (4)           Type of comparison.
1099: (8)           rstrip : Boolean
1100: (4)           If True, the spaces at the end of Strings are removed before the
comparison.
1101: (4)
1102: (4)
1103: (8)           Returns
1104: (4)           -----
1105: (4)           out : ndarray
1106: (4)           The output array of type Boolean with the same shape as a and b.
1107: (4)           Raises
1108: (4)
1109: (4)           -----
1110: (4)           ValueError
1111: (4)           If `cmp_op` is not valid.
1112: (4)           TypeError
1113: (4)           If at least one of `a` or `b` is a non-string array
1114: (4)
1115: (4)
1116: (4)
1117: (0)           Examples
1118: (4)
1119: (4)           -----
1120: (4)           >>> a = np.array(["a", "b", "cde"])
1121: (4)           >>> b = np.array(["a", "a", "dec"])
1122: (4)           >>> np.compare_chararrays(a, b, ">", True)
1123: (4)           array([False, True, False])
1124: (4)           """
1125: (4)           add_newdoc('numpy.core.multiarray', 'fromiter',
1126: (4)           """
1127: (4)           fromiter(iter, dtype, count=-1, *, like=None)
1128: (4)           Create a new 1-dimensional array from an iterable object.
1129: (4)           Parameters
1130: (4)
1131: (4)           iter : iterable object
1132: (4)           An iterable object providing data for the array.
1133: (4)           dtype : data-type
1134: (4)           The data-type of the returned array.
1135: (4)           .. versionchanged:: 1.23
1136: (4)           Object and subarray dtypes are now supported (note that the final
1137: (4)           result is not 1-D for a subarray dtype).
1138: (4)           count : int, optional
1139: (4)           The number of items to read from *iterable*. The default is -1,
1140: (4)           which means all data is read.
1141: (4)           ${ARRAY_FUNCTION_LIKE}
1142: (4)           .. versionadded:: 1.20.0
1143: (4)
1144: (4)           Returns

```

```

1136: (4)
1137: (4)
1138: (8)
1139: (4)
1140: (4)
1141: (4)
1142: (4)
1143: (4)
1144: (4)
1145: (4)
1146: (4)
1147: (4)
1148: (4)
1149: (4)
1150: (4)
1151: (4)
1152: (4)
1153: (11)
1154: (11)
1155: (11)
1156: (11)
1157: (4)
1158: (8)
1159: (8)
1160: (4)
1161: (0)
1162: (4)
1163: (4)
1164: (4)
1165: (4)
1166: (4)
1167: (4)
1168: (4)
1169: (4)
1170: (4)
1171: (8)
1172: (8)
1173: (12)
1174: (4)
1175: (8)
1176: (8)
1177: (8)
1178: (8)
supported.
1179: (8)
1180: (12)
1181: (4)
1182: (8)
1183: (8)
1184: (4)
1185: (8)
1186: (8)
1187: (8)
characters.
1188: (8)
1189: (8)
1190: (4)
1191: (8)
1192: (8)
1193: (8)
1194: (4)
1195: (8)
1196: (4)
1197: (4)
1198: (4)
1199: (4)
1200: (4)
1201: (4)
1202: (4)

        -----
        out : ndarray
              The output array.
        Notes
        -----
Specify `count` to improve performance. It allows ``fromiter`` to
pre-allocate the output array, instead of resizing it on demand.
        Examples
        -----
        >>> iterable = (x*x for x in range(5))
        >>> np.fromiter(iterable, float)
        array([ 0.,  1.,  4.,  9., 16.])
A carefully constructed subarray dtype will lead to higher dimensional
results:
        >>> iterable = ((x+1, x+2) for x in range(5))
        >>> np.fromiter(iterable, dtype=np.dtype((int, 2)))
        array([[1, 2],
               [2, 3],
               [3, 4],
               [4, 5],
               [5, 6]])
        """.
replace(
        "${ARRAY_FUNCTION_LIKE}",
        array_function_like_doc,
    ))
add_newdoc('numpy.core.multiarray', 'fromfile',
"""
        fromfile(file, dtype=float, count=-1, sep='', offset=0, *, like=None)
Construct an array from data in a text or binary file.
A highly efficient way of reading binary data with a known data-type,
as well as parsing simply formatted text files. Data written using the
`tofile` method can be read using this function.
        Parameters
        -----
        file : file or str or Path
            Open file object or filename.
            .. versionchanged:: 1.17.0
                `pathlib.Path` objects are now accepted.
        dtype : data-type
            Data type of the returned array.
            For binary files, it is used to determine the size and byte-order
            of the items in the file.
            Most builtin numeric types are supported and extension types may be
            ..
            .. versionadded:: 1.18.0
                Complex dtypes.
        count : int
            Number of items to read. ``-1`` means all items (i.e., the complete
            file).
        sep : str
            Separator between items if file is a text file.
            Empty ("") separator means the file should be treated as binary.
            Spaces (" ") in the separator match zero or more whitespace
            A separator consisting only of spaces must match at least one
            whitespace.
        offset : int
            The offset (in bytes) from the file's current position. Defaults to 0.
            Only permitted for binary files.
            ..
            .. versionadded:: 1.17.0
${ARRAY_FUNCTION_LIKE}
            ..
            .. versionadded:: 1.20.0
        See also
        -----
        load, save
        ndarray.tofile
        loadtxt : More flexible way of loading data from a text file.
        Notes
        -----

```

```

1203: (4)          Do not rely on the combination of `tofile` and `fromfile` for
1204: (4)          data storage, as the binary files generated are not platform
1205: (4)          independent. In particular, no byte-order or data-type information is
1206: (4)          saved. Data can be stored in the platform independent ``.npy`` format
1207: (4)          using `save` and `load` instead.
1208: (4)          Examples
1209: (4)
1210: (4)          Construct an ndarray:
1211: (4)          >>> dt = np.dtype([('time', [(('min', np.int64), ('sec', np.int64))]),
1212: (4)                      ...                               ('temp', float)])
1213: (4)          >>> x = np.zeros((1,), dtype=dt)
1214: (4)          >>> x['time']['min'] = 10; x['temp'] = 98.25
1215: (4)          >>> x
1216: (4)          array([(10, 0), 98.25]),
1217: (10)          dtype=[('time', [('min', '<i8'), ('sec', '<i8')]), ('temp', '<f8')])
1218: (4)          Save the raw data to disk:
1219: (4)          >>> import tempfile
1220: (4)          >>> fname = tempfile.mkstemp()[1]
1221: (4)          >>> x.tofile(fname)
1222: (4)          Read the raw data from disk:
1223: (4)          >>> np.fromfile(fname, dtype=dt)
1224: (4)          array([(10, 0), 98.25]),
1225: (10)          dtype=[('time', [('min', '<i8'), ('sec', '<i8')]), ('temp', '<f8')])
1226: (4)          The recommended way to store and load data:
1227: (4)          >>> np.save(fname, x)
1228: (4)          >>> np.load(fname + '.npy')
1229: (4)          array([(10, 0), 98.25]),
1230: (10)          dtype=[('time', [('min', '<i8'), ('sec', '<i8')]), ('temp', '<f8')])
1231: (4)          """.replace(
1232: (8)              "${ARRAY_FUNCTION_LIKE}",
1233: (8)              array_function_like_doc,
1234: (4)          ))
1235: (0)          add_newdoc('numpy.core.multiarray', 'frombuffer',
1236: (4)          """
1237: (4)              frombuffer(buffer, dtype=float, count=-1, offset=0, *, like=None)
1238: (4)              Interpret a buffer as a 1-dimensional array.
1239: (4)              Parameters
1240: (4)
1241: (4)                  buffer : buffer_like
1242: (8)                      An object that exposes the buffer interface.
1243: (4)                  dtype : data-type, optional
1244: (8)                      Data-type of the returned array; default: float.
1245: (4)                  count : int, optional
1246: (8)                      Number of items to read. ``-1`` means all data in the buffer.
1247: (4)                  offset : int, optional
1248: (8)                      Start reading the buffer from this offset (in bytes); default: 0.
1249: (4)                  ${ARRAY_FUNCTION_LIKE}
1250: (8)                      .. versionadded:: 1.20.0
1251: (4)          Returns
1252: (4)
1253: (4)          out : ndarray
1254: (4)          See also
1255: (4)
1256: (4)          ndarray.tobytes
1257: (8)              Inverse of this operation, construct Python bytes from the raw data
1258: (8)              bytes in the array.
1259: (4)          Notes
1260: (4)
1261: (4)          If the buffer has data that is not in machine byte-order, this should
1262: (4)          be specified as part of the data-type, e.g.::
1263: (6)              >>> dt = np.dtype(int)
1264: (6)              >>> dt = dt.newbyteorder('>')
1265: (6)              >>> np.frombuffer(buf, dtype=dt) # doctest: +SKIP
1266: (4)          The data of the resulting array will not be byteswapped, but will be
1267: (4)          interpreted correctly.
1268: (4)          This function creates a view into the original object. This should be
safe
1269: (4)          in general, but it may make sense to copy the result when the original
1270: (4)          object is mutable or untrusted.

```

```

1271: (4)          Examples
1272: (4)          -----
1273: (4)          >>> s = b'hello world'
1274: (4)          >>> np.frombuffer(s, dtype='S1', count=5, offset=6)
1275: (4)          array([b'w', b'o', b'r', b'l', b'd'], dtype='|S1')
1276: (4)          >>> np.frombuffer(b'\x01\x02', dtype=np.uint8)
1277: (4)          array([1, 2], dtype=uint8)
1278: (4)          >>> np.frombuffer(b'\x01\x02\x03\x04\x05', dtype=np.uint8, count=3)
1279: (4)          array([1, 2, 3], dtype=uint8)
1280: (4)          """".replace(
1281: (8)            "${ARRAY_FUNCTION_LIKE}",
1282: (8)            array_function_like_doc,
1283: (4)        ))
1284: (0)        add_newdoc('numpy.core.multiarray', 'from_dlpark',
1285: (4)          """
1286: (4)          from_dlpark(x, /)
1287: (4)          Create a NumPy array from an object implementing the ``__dlpack__``
1288: (4)          protocol. Generally, the returned NumPy array is a read-only view
1289: (4)          of the input object. See [1]_ and [2]_ for more details.
1290: (4)          Parameters
1291: (4)          -----
1292: (4)          x : object
1293: (8)            A Python object that implements the ``__dlpack__`` and
1294: (8)            ``__dlpack_device__`` methods.
1295: (4)          Returns
1296: (4)          -----
1297: (4)          out : ndarray
1298: (4)          References
1299: (4)          -----
1300: (4)            .. [1] Array API documentation,
1301: (7)              https://data-apis.org/array-
api/latest/design_topics/data_interchange.html#syntax-for-data-interchange-with-dlpark
1302: (4)            .. [2] Python specification for DLPack,
1303: (7)              https://dmlc.github.io/dlpark/latest/python_spec.html
1304: (4)          Examples
1305: (4)          -----
1306: (4)          >>> import torch
1307: (4)          >>> x = torch.arange(10)
1308: (4)          >>> # create a view of the torch tensor "x" in NumPy
1309: (4)          >>> y = np.from_dlpark(x)
1310: (4)          """")
1311: (0)        add_newdoc('numpy.core', 'fastCopyAndTranspose',
1312: (4)          """
1313: (4)          fastCopyAndTranspose(a)
1314: (4)          .. deprecated:: 1.24
1315: (7)            fastCopyAndTranspose is deprecated and will be removed. Use the copy
and
1316: (7)            transpose methods instead, e.g. ``arr.T.copy()``
1317: (4)          """")
1318: (0)        add_newdoc('numpy.core.multiarray', 'correlate',
1319: (4)          """cross_correlate(a,v, mode=0)""")
1320: (0)        add_newdoc('numpy.core.multiarray', 'arange',
1321: (4)          """
1322: (4)            arange([start,] stop[, step,], dtype=None, *, like=None)
1323: (4)            Return evenly spaced values within a given interval.
1324: (4)            ``arange`` can be called with a varying number of positional arguments:
1325: (4)            * ``arange(stop)``: Values are generated within the half-open interval
1326: (6)              ``[0, stop)`` (in other words, the interval including `start` but
1327: (6)              excluding `stop`).
1328: (4)            * ``arange(start, stop)``: Values are generated within the half-open
1329: (6)              interval ``[start, stop)``.
1330: (4)            * ``arange(start, stop, step)``: Values are generated within the half-open
1331: (6)              interval ``[start, stop)``, with spacing between values given by
1332: (6)              ``step``.
1333: (4)            For integer arguments the function is roughly equivalent to the Python
1334: (4)            built-in :py:class:`range`, but returns an ndarray rather than a ``range``
1335: (4)            instance.
1336: (4)            When using a non-integer step, such as 0.1, it is often better to use
1337: (4)              ``numpy.linspace``.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1338: (4) See the Warning sections below for more information.
1339: (4) Parameters
1340: (4)
1341: (4) start : integer or real, optional
1342: (8) Start of interval. The interval includes this value. The default
1343: (8) start value is 0.
1344: (4) stop : integer or real
1345: (8) End of interval. The interval does not include this value, except
1346: (8) in some cases where `step` is not an integer and floating point
1347: (8) round-off affects the length of `out`.
1348: (4) step : integer or real, optional
1349: (8) Spacing between values. For any output `out`, this is the distance
1350: (8) between two adjacent values, ``out[i+1] - out[i]``. The default
1351: (8) step size is 1. If `step` is specified as a position argument,
1352: (8) `start` must also be given.
1353: (4) dtype : dtype, optional
1354: (8) The type of the output array. If `dtype` is not given, infer the data
1355: (8) type from the other input arguments.
1356: (4) ${ARRAY_FUNCTION_LIKE}
1357: (8) .. versionadded:: 1.20.0
1358: (4) Returns
1359: (4)
1360: (4) arange : ndarray
1361: (8) Array of evenly spaced values.
1362: (8) For floating point arguments, the length of the result is
1363: (8) ``ceil((stop - start)/step)``. Because of floating point overflow,
1364: (8) this rule may result in the last element of `out` being greater
1365: (8) than `stop`.
1366: (4) Warnings
1367: (4)
1368: (4) The length of the output might not be numerically stable.
1369: (4) Another stability issue is due to the internal implementation of
1370: (4) `numpy.arange`.
1371: (4) The actual step value used to populate the array is
1372: (4) ``dtype(start + step) - dtype(start)`` and not `step`. Precision loss
1373: (4) can occur here, due to casting or due to using floating points when
1374: (4) `start` is much larger than `step`. This can lead to unexpected
1375: (4) behaviour. For example::
1376: (6) >>> np.arange(0, 5, 0.5, dtype=int)
1377: (6) array([0, 0, 0, 0, 0, 0, 0, 0])
1378: (6) >>> np.arange(-3, 3, 0.5, dtype=int)
1379: (6) array([-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8])
1380: (4) In such cases, the use of `numpy.linspace` should be preferred.
1381: (4) The built-in :py:class:`range` generates :std:doc:`Python built-in
integers
1382: (4) integers
1383: (4) that have arbitrary size <python:c-api/long>, while `numpy.arange`
1384: (4) produces `numpy.int32` or `numpy.int64` numbers. This may result in
incorrect results for large integer values::
1385: (6) >>> power = 40
1386: (6) >>> modulo = 10000
1387: (6) >>> x1 = [(n ** power) % modulo for n in range(8)]
1388: (6) >>> x2 = [(n ** power) % modulo for n in np.arange(8)]
1389: (6) >>> print(x1)
1390: (6) [0, 1, 7776, 8801, 6176, 625, 6576, 4001] # correct
1391: (6) >>> print(x2)
1392: (6) [0, 1, 7776, 7185, 0, 5969, 4816, 3361] # incorrect
1393: (4) See Also
1394: (4)
1395: (4) numpy.linspace : Evenly spaced numbers with careful handling of endpoints.
1396: (4) numpy.ogrid: Arrays of evenly spaced numbers in N-dimensions.
1397: (4) numpy.mgrid: Grid-shaped arrays of evenly spaced numbers in N-dimensions.
1398: (4) :ref:`how-to-partition`
1399: (4) Examples
1400: (4)
1401: (4) >>> np.arange(3)
1402: (4) array([0, 1, 2])
1403: (4) >>> np.arange(3.0)
1404: (4) array([ 0.,  1.,  2.])
1405: (4) >>> np.arange(3,7)

```

```

1406: (4)           array([3, 4, 5, 6])
1407: (4)           >>> np.arange(3,7,2)
1408: (4)           array([3, 5])
1409: (4)           """.replace(
1410: (8)             "${ARRAY_FUNCTION_LIKE}",
1411: (8)             array_function_like_doc,
1412: (4)         )
1413: (0)           add_newdoc('numpy.core.multiarray', '_get_ndarray_c_version',
1414: (4)             """_get_ndarray_c_version()
1415: (4)             Return the compile time NPY_VERSION (formerly called NDARRAY_VERSION)
number.
1416: (4)           """
1417: (0)           add_newdoc('numpy.core.multiarray', '_reconstruct',
1418: (4)             """_reconstruct(subtype, shape, dtype)
1419: (4)             Construct an empty array. Used by Pickles.
1420: (4)             """
1421: (0)           add_newdoc('numpy.core.multiarray', 'set_string_function',
1422: (4)             """
1423: (4)               set_string_function(f, repr=1)
1424: (4)               Internal method to set a function to be used when pretty printing arrays.
1425: (4)             """
1426: (0)           add_newdoc('numpy.core.multiarray', 'set_numeric_ops',
1427: (4)             """
1428: (4)               set_numeric_ops(op1=func1, op2=func2, ...)
1429: (4)               Set numerical operators for array objects.
1430: (4)               .. deprecated:: 1.16
1431: (8)                 For the general case, use :c:func:`PyUFunc.ReplaceLoopBySignature`.
1432: (8)                 For ndarray subclasses, define the ``__array_ufunc__`` method and
1433: (8)                 override the relevant ufunc.
1434: (4)             Parameters
1435: (4)             -----
1436: (4)               op1, op2, ... : callable
1437: (8)                 Each ``op = func`` pair describes an operator to be replaced.
1438: (8)                 For example, ``add = lambda x, y: np.add(x, y) % 5`` would replace
1439: (8)                 addition by modulus 5 addition.
1440: (4)             Returns
1441: (4)             -----
1442: (4)               saved_ops : list of callables
1443: (8)                 A list of all operators, stored before making replacements.
1444: (4)             Notes
1445: (4)             -----
1446: (4)               .. warning::
1447: (7)                 Use with care! Incorrect usage may lead to memory errors.
1448: (4)             A function replacing an operator cannot make use of that operator.
1449: (4)             For example, when replacing add, you may not use ``+``. Instead,
1450: (4)             directly call ufuncs.
1451: (4)             Examples
1452: (4)             -----
1453: (4)               >>> def add_mod5(x, y):
1454: (4)                 ...     return np.add(x, y) % 5
1455: (4)                 ...
1456: (4)               >>> old_funcs = np.set_numeric_ops(add=add_mod5)
1457: (4)               >>> x = np.arange(12).reshape((3, 4))
1458: (4)               >>> x + x
1459: (4)                 array([[0, 2, 4, 1],
1460: (11)                   [3, 0, 2, 4],
1461: (11)                   [1, 3, 0, 2]])
1462: (4)               >>> ignore = np.set_numeric_ops(**old_funcs) # restore operators
1463: (4)             """
1464: (0)           add_newdoc('numpy.core.multiarray', 'promote_types',
1465: (4)             """
1466: (4)               promote_types(type1, type2)
1467: (4)               Returns the data type with the smallest size and smallest scalar
1468: (4)               kind to which both ``type1`` and ``type2`` may be safely cast.
1469: (4)               The returned data type is always considered "canonical", this mainly
1470: (4)               means that the promoted dtype will always be in native byte order.
1471: (4)               This function is symmetric, but rarely associative.
1472: (4)             Parameters
1473: (4)             -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1474: (4) type1 : dtype or dtype specifier
1475: (8)     First data type.
1476: (4) type2 : dtype or dtype specifier
1477: (8)     Second data type.
1478: (4) Returns
1479: (4) -----
1480: (4) out : dtype
1481: (8)     The promoted data type.
1482: (4) Notes
1483: (4) -----
1484: (4) Please see `numpy.result_type` for additional information about promotion.
1485: (4) .. versionadded:: 1.6.0
1486: (4) Starting in NumPy 1.9, promote_types function now returns a valid string
1487: (4) length when given an integer or float dtype as one argument and a string
1488: (4) dtype as another argument. Previously it always returned the input string
1489: (4) dtype, even if it wasn't long enough to store the max integer/float value
1490: (4) converted to a string.
1491: (4) .. versionchanged:: 1.23.0
1492: (4) NumPy now supports promotion for more structured dtypes. It will now
1493: (4) remove unnecessary padding from a structure dtype and promote included
1494: (4) fields individually.
1495: (4) See Also
1496: (4) -----
1497: (4) result_type, dtype, can_cast
1498: (4) Examples
1499: (4) -----
1500: (4) >>> np.promote_types('f4', 'f8')
1501: (4) dtype('float64')
1502: (4) >>> np.promote_types('i8', 'f4')
1503: (4) dtype('float64')
1504: (4) >>> np.promote_types('>i8', '<c8')
1505: (4) dtype('complex128')
1506: (4) >>> np.promote_types('i4', 'S8')
1507: (4) dtype('S11')
1508: (4) An example of a non-associative case:
1509: (4) >>> p = np.promote_types
1510: (4) >>> p('S', p('i1', 'u1'))
1511: (4) dtype('S6')
1512: (4) >>> p(p('S', 'i1'), 'u1')
1513: (4) dtype('S4')
1514: (4) """
1515: (0) add_newdoc('numpy.core.multiarray', 'c_einsum',
1516: (4) """
1517: (4)     c_einsum(subscripts, *operands, out=None, dtype=None, order='K',
1518: (11)         casting='safe')
1519: (4)     *This documentation shadows that of the native python implementation of
the `einsum` function,
1520: (4)     except all references and examples related to the `optimize` argument (v
0.12.0) have been removed.*
1521: (4)     Evaluates the Einstein summation convention on the operands.
1522: (4)     Using the Einstein summation convention, many common multi-dimensional,
1523: (4)     linear algebraic array operations can be represented in a simple fashion.
1524: (4)     In *implicit* mode `einsum` computes these values.
1525: (4)     In *explicit* mode, `einsum` provides further flexibility to compute
1526: (4)     other array operations that might not be considered classical Einstein
1527: (4)     summation operations, by disabling, or forcing summation over specified
1528: (4)     subscript labels.
1529: (4)     See the notes and examples for clarification.
1530: (4) Parameters
1531: (4) -----
1532: (4) subscripts : str
1533: (8)     Specifies the subscripts for summation as comma separated list of
1534: (8)     subscript labels. An implicit (classical Einstein summation)
1535: (8)     calculation is performed unless the explicit indicator '->' is
1536: (8)     included as well as subscript labels of the precise output form.
1537: (4) operands : list of array_like
1538: (8)     These are the arrays for the operation.
1539: (4) out : ndarray, optional
1540: (8)     If provided, the calculation is done into this array.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1541: (4)          dtype : {data-type, None}, optional
1542: (8)          If provided, forces the calculation to use the data type specified.
1543: (8)          Note that you may have to also give a more liberal `casting` parameter to allow the conversions. Default is None.
1544: (8)
1545: (4)          order : {'C', 'F', 'A', 'K'}, optional
1546: (8)          Controls the memory layout of the output. 'C' means it should be C contiguous. 'F' means it should be Fortran contiguous, 'A' means it should be 'F' if the inputs are all 'F', 'C' otherwise. 'K' means it should be as close to the layout of the inputs as is possible, including arbitrarily permuted axes.
1547: (8)          Default is 'K'.
1548: (8)
1549: (8)
1550: (8)
1551: (8)
1552: (4)          casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
1553: (8)          Controls what kind of data casting may occur. Setting this to 'unsafe' is not recommended, as it can adversely affect accumulations.
1554: (8)          * 'no' means the data types should not be cast at all.
1555: (10)         * 'equiv' means only byte-order changes are allowed.
1556: (10)         * 'safe' means only casts which can preserve values are allowed.
1557: (10)         * 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
1558: (10)
1559: (12)
1560: (10)
1561: (8)          * 'unsafe' means any data conversions may be done.
1562: (4)          Default is 'safe'.
1563: (8)          optimize : {False, True, 'greedy', 'optimal'}, optional
1564: (8)          Controls if intermediate optimization should occur. No optimization will occur if False and True will default to the 'greedy' algorithm.
1565: (8)          Also accepts an explicit contraction list from the ``np.einsum_path`` function. See ``np.einsum_path`` for more details. Defaults to False.
1566: (8)
1567: (4)          Returns
1568: (4)          -----
1569: (4)          output : ndarray
1570: (8)          The calculation based on the Einstein summation convention.
1571: (4)          See Also
1572: (4)          -----
1573: (4)          einsum_path, dot, inner, outer, tensordot, linalg.multi_dot
1574: (4)
1575: (4)
1576: (4)          .. versionadded:: 1.6.0
1577: (4)          The Einstein summation convention can be used to compute many multi-dimensional, linear algebraic array operations. `einsum` provides a succinct way of representing these.
1578: (4)          A non-exhaustive list of these operations,
1579: (4)          which can be computed by `einsum`, is shown below along with examples:
1580: (4)          * Trace of an array, :py:func:`numpy.trace`.
1581: (4)          * Return a diagonal, :py:func:`numpy.diag`.
1582: (4)          * Array axis summations, :py:func:`numpy.sum`.
1583: (4)          * Transpositions and permutations, :py:func:`numpy.transpose`.
1584: (4)          * Matrix multiplication and dot product, :py:func:`numpy.matmul`.
1585: (4)          * Vector inner and outer products, :py:func:`numpy.inner`.
1586: (4)          * Broadcasting, element-wise and scalar multiplication, :py:func:`numpy.multiply`.
1587: (4)          * Tensor contractions, :py:func:`numpy.tensordot`.
1588: (4)          * Chained array operations, in efficient calculation order, :py:func:`numpy.einsum_path`.
1589: (4)
1590: (4)
1591: (4)          The subscripts string is a comma-separated list of subscript labels, where each label refers to a dimension of the corresponding operand.
1592: (4)          Whenever a label is repeated it is summed, so ``np.einsum('i,i', a, b)`` is equivalent to :py:func:`np.inner(a,b) <numpy.inner>`. If a label appears only once, it is not summed, so ``np.einsum('i', a)`` produces a view of ``a`` with no changes. A further example ``np.einsum('ij,jk', a, b)``
1593: (4)
1594: (4)
1595: (4)
1596: (4)
1597: (4)
1598: (4)
1599: (4)
equivalent
1600: (4)          describes traditional matrix multiplication and is equivalent to :py:func:`np.matmul(a,b) <numpy.matmul>`. Repeated subscript labels in one operand take the diagonal. For example, ``np.einsum('ii', a)`` is
1601: (4)          to :py:func:`np.trace(a) <numpy.trace>`.
1602: (4)          In *implicit mode*, the chosen subscripts are important since the axes of the output are reordered alphabetically. This means that ``np.einsum('ij', a)`` doesn't affect a 2D array, while
1603: (4)

```

```

1604: (4) ``np.einsum('ji', a)`` takes its transpose. Additionally,
1605: (4) ``np.einsum('ij,jk', a, b)`` returns a matrix multiplication, while,
1606: (4) ``np.einsum('ij,jh', a, b)`` returns the transpose of the
1607: (4) multiplication since subscript 'h' precedes subscript 'i'.
1608: (4) In *explicit mode* the output can be directly controlled by
1609: (4) specifying output subscript labels. This requires the
1610: (4) identifier ' $\rightarrow$ ' as well as the list of output subscript labels.
1611: (4) This feature increases the flexibility of the function since
1612: (4) summing can be disabled or forced when required. The call
1613: (4) ``np.einsum('i->', a)`` is like :py:func:`np.sum(a, axis=-1)` <numpy.sum>,
1614: (4) and ``np.einsum('ii->i', a)`` is like :py:func:`np.diag(a)` <numpy.diag>.
1615: (4) The difference is that `einsum` does not allow broadcasting by default.
1616: (4) Additionally ``np.einsum('ij,jh->ih', a, b)`` directly specifies the
1617: (4) order of the output subscript labels and therefore returns matrix
1618: (4) multiplication, unlike the example above in implicit mode.
1619: (4) To enable and control broadcasting, use an ellipsis. Default
1620: (4) NumPy-style broadcasting is done by adding an ellipsis
1621: (4) to the left of each term, like ``np.einsum('...ii->...i', a)``.
1622: (4) To take the trace along the first and last axes,
1623: (4) you can do ``np.einsum('i...i', a)``, or to do a matrix-matrix
1624: (4) product with the left-most indices instead of rightmost, one can do
1625: (4) ``np.einsum('ij...,jk....>ik...', a, b)``.
1626: (4) When there is only one operand, no axes are summed, and no output
1627: (4) parameter is provided, a view into the operand is returned instead
1628: (4) of a new array. Thus, taking the diagonal as ``np.einsum('ii->i', a)``
1629: (4) produces a view (changed in version 1.10.0).
1630: (4) `einsum` also provides an alternative way to provide the subscripts
1631: (4) and operands as ``einsum(op0, sublist0, op1, sublist1, ...,
[sublistout])``.

1632: (4) If the output shape is not provided in this format `einsum` will be
1633: (4) calculated in implicit mode, otherwise it will be performed explicitly.
1634: (4) The examples below have corresponding `einsum` calls with the two
1635: (4) parameter methods.
1636: (4) .. versionadded:: 1.10.0
1637: (4) Views returned from einsum are now writeable whenever the input array
1638: (4) is writeable. For example, ``np.einsum('ijk...->kji...', a)`` will now
1639: (4) have the same effect as :py:func:`np.swapaxes(a, 0, 2)` <numpy.swapaxes>
1640: (4) and ``np.einsum('ii->i', a)`` will return a writeable view of the diagonal
1641: (4) of a 2D array.
1642: (4) Examples
1643: (4) -----
1644: (4) >>> a = np.arange(25).reshape(5,5)
1645: (4) >>> b = np.arange(5)
1646: (4) >>> c = np.arange(6).reshape(2,3)
1647: (4) Trace of a matrix:
1648: (4) >>> np.einsum('ii', a)
1649: (4) 60
1650: (4) >>> np.einsum(a, [0,0])
1651: (4) 60
1652: (4) >>> np.trace(a)
1653: (4) 60
1654: (4) Extract the diagonal (requires explicit form):
1655: (4) >>> np.einsum('ii->i', a)
1656: (4) array([ 0,  6, 12, 18, 24])
1657: (4) >>> np.einsum(a, [0,0], [0])
1658: (4) array([ 0,  6, 12, 18, 24])
1659: (4) >>> np.diag(a)
1660: (4) array([ 0,  6, 12, 18, 24])
1661: (4) Sum over an axis (requires explicit form):
1662: (4) >>> np.einsum('ij->i', a)
1663: (4) array([ 10,  35,  60,  85, 110])
1664: (4) >>> np.einsum(a, [0,1], [0])
1665: (4) array([ 10,  35,  60,  85, 110])
1666: (4) >>> np.sum(a, axis=1)
1667: (4) array([ 10,  35,  60,  85, 110])
1668: (4) For higher dimensional arrays summing a single axis can be done with
ellipsis:
1669: (4) >>> np.einsum('...j->...', a)
1670: (4) array([ 10,  35,  60,  85, 110])

```

```

1671: (4)          >>> np.einsum(a, [Ellipsis,1], [Ellipsis])
1672: (4)          array([ 10,  35,  60,  85, 110])
1673: (4)          Compute a matrix transpose, or reorder any number of axes:
1674: (4)          >>> np.einsum('ji', c)
1675: (4)          array([[0, 3],
1676: (11)             [1, 4],
1677: (11)             [2, 5]])
1678: (4)          >>> np.einsum('ij->ji', c)
1679: (4)          array([[0, 3],
1680: (11)             [1, 4],
1681: (11)             [2, 5]])
1682: (4)          >>> np.einsum(c, [1,0])
1683: (4)          array([[0, 3],
1684: (11)             [1, 4],
1685: (11)             [2, 5]])
1686: (4)          >>> np.transpose(c)
1687: (4)          array([[0, 3],
1688: (11)             [1, 4],
1689: (11)             [2, 5]])
1690: (4)          Vector inner products:
1691: (4)          >>> np.einsum('i,i', b, b)
1692: (4)          30
1693: (4)          >>> np.einsum(b, [0], b, [0])
1694: (4)          30
1695: (4)          >>> np.inner(b,b)
1696: (4)          30
1697: (4)          Matrix vector multiplication:
1698: (4)          >>> np.einsum('ij,j', a, b)
1699: (4)          array([ 30,  80, 130, 180, 230])
1700: (4)          >>> np.einsum(a, [0,1], b, [1])
1701: (4)          array([ 30,  80, 130, 180, 230])
1702: (4)          >>> np.dot(a, b)
1703: (4)          array([ 30,  80, 130, 180, 230])
1704: (4)          >>> np.einsum('...j,j', a, b)
1705: (4)          array([ 30,  80, 130, 180, 230])
1706: (4)          Broadcasting and scalar multiplication:
1707: (4)          >>> np.einsum('..., ...', 3, c)
1708: (4)          array([[ 0,  3,  6],
1709: (11)             [ 9, 12, 15]])
1710: (4)          >>> np.einsum(',ij', 3, c)
1711: (4)          array([[ 0,  3,  6],
1712: (11)             [ 9, 12, 15]])
1713: (4)          >>> np.einsum(3, [Ellipsis], c, [Ellipsis])
1714: (4)          array([[ 0,  3,  6],
1715: (11)             [ 9, 12, 15]])
1716: (4)          >>> np.multiply(3, c)
1717: (4)          array([[ 0,  3,  6],
1718: (11)             [ 9, 12, 15]])
1719: (4)          Vector outer product:
1720: (4)          >>> np.einsum('i,j', np.arange(2)+1, b)
1721: (4)          array([[ 0,  1,  2,  3,  4],
1722: (11)             [ 0,  2,  4,  6,  8]])
1723: (4)          >>> np.einsum(np.arange(2)+1, [0], b, [1])
1724: (4)          array([[ 0,  1,  2,  3,  4],
1725: (11)             [ 0,  2,  4,  6,  8]])
1726: (4)          >>> np.outer(np.arange(2)+1, b)
1727: (4)          array([[ 0,  1,  2,  3,  4],
1728: (11)             [ 0,  2,  4,  6,  8]])
1729: (4)          Tensor contraction:
1730: (4)          >>> a = np.arange(60.).reshape(3,4,5)
1731: (4)          >>> b = np.arange(24.).reshape(4,3,2)
1732: (4)          >>> np.einsum('ijk,jil->kl', a, b)
1733: (4)          array([[ 4400.,   4730.],
1734: (11)             [ 4532.,   4874.],
1735: (11)             [ 4664.,   5018.],
1736: (11)             [ 4796.,   5162.],
1737: (11)             [ 4928.,   5306.]])
1738: (4)          >>> np.einsum(a, [0,1,2], b, [1,0,3], [2,3])
1739: (4)          array([[ 4400.,   4730.],

```

```

1740: (11) [ 4532., 4874.],
1741: (11) [ 4664., 5018.],
1742: (11) [ 4796., 5162.],
1743: (11) [ 4928., 5306.]])
1744: (4) >>> np.tensordot(a,b, axes=([1,0],[0,1]))
1745: (4) array([[ 4400., 4730.],
1746: (11) [ 4532., 4874.],
1747: (11) [ 4664., 5018.],
1748: (11) [ 4796., 5162.],
1749: (11) [ 4928., 5306.]])
1750: (4) Writeable returned arrays (since version 1.10.0):
1751: (4) >>> a = np.zeros((3, 3))
1752: (4) >>> np.einsum('ii->i', a)[:] = 1
1753: (4) >>> a
1754: (4) array([[ 1., 0., 0.],
1755: (11) [ 0., 1., 0.],
1756: (11) [ 0., 0., 1.]])
1757: (4) Example of ellipsis use:
1758: (4) >>> a = np.arange(6).reshape((3,2))
1759: (4) >>> b = np.arange(12).reshape((4,3))
1760: (4) >>> np.einsum('ki,jk->ij', a, b)
1761: (4) array([[10, 28, 46, 64],
1762: (11) [13, 40, 67, 94]])
1763: (4) >>> np.einsum('ki,...k->i...', a, b)
1764: (4) array([[10, 28, 46, 64],
1765: (11) [13, 40, 67, 94]])
1766: (4) >>> np.einsum('k...,jk', a, b)
1767: (4) array([[10, 28, 46, 64],
1768: (11) [13, 40, 67, 94]])
1769: (4) """
1770: (0) add_newdoc('numpy.core.multiarray', 'ndarray',
1771: (4) """
1772: (4) ndarray(shape, dtype=float, buffer=None, offset=0,
1773: (12) strides=None, order=None)
1774: (4) An array object represents a multidimensional, homogeneous array
1775: (4) of fixed-size items. An associated data-type object describes the
1776: (4) format of each element in the array (its byte-order, how many bytes it
1777: (4) occupies in memory, whether it is an integer, a floating point number,
1778: (4) or something else, etc.)
1779: (4) Arrays should be constructed using `array`, `zeros` or `empty` (refer
1780: (4) to the See Also section below). The parameters given here refer to
1781: (4) a low-level method (`ndarray(...)`) for instantiating an array.
1782: (4) For more information, refer to the `numpy` module and examine the
1783: (4) methods and attributes of an array.
1784: (4) Parameters
1785: (4) -----
1786: (4) (for the __new__ method; see Notes below)
1787: (4) shape : tuple of ints
1788: (8) Shape of created array.
1789: (4) dtype : data-type, optional
1790: (8) Any object that can be interpreted as a numpy data type.
1791: (4) buffer : object exposing buffer interface, optional
1792: (8) Used to fill the array with data.
1793: (4) offset : int, optional
1794: (8) Offset of array data in buffer.
1795: (4) strides : tuple of ints, optional
1796: (8) Strides of data in memory.
1797: (4) order : {'C', 'F'}, optional
1798: (8) Row-major (C-style) or column-major (Fortran-style) order.
1799: (4) Attributes
1800: (4) -----
1801: (4) T : ndarray
1802: (8) Transpose of the array.
1803: (4) data : buffer
1804: (8) The array's elements, in memory.
1805: (4) dtype : dtype object
1806: (8) Describes the format of the elements in the array.
1807: (4) flags : dict
1808: (8) Dictionary containing information related to memory use, e.g.,

```

```

1809: (8)           'C_CONTIGUOUS', 'OWNDATA', 'WRITEABLE', etc.
1810: (4)           flat : numpy.flatiter object
1811: (8)           Flattened version of the array as an iterator. The iterator
1812: (8)           allows assignments, e.g., ``x.flat = 3`` (See `ndarray.flat` for
1813: (8)           assignment examples; TODO).
1814: (4)           imag : ndarray
1815: (8)           Imaginary part of the array.
1816: (4)           real : ndarray
1817: (8)           Real part of the array.
1818: (4)           size : int
1819: (8)           Number of elements in the array.
1820: (4)           itemsize : int
1821: (8)           The memory use of each array element in bytes.
1822: (4)           nbytes : int
1823: (8)           The total number of bytes required to store the array data,
1824: (8)           i.e., ``itemsize * size``.
1825: (4)           ndim : int
1826: (8)           The array's number of dimensions.
1827: (4)           shape : tuple of ints
1828: (8)           Shape of the array.
1829: (4)           strides : tuple of ints
1830: (8)           The step-size required to move from one element to the next in
1831: (8)           memory. For example, a contiguous ``(3, 4)`` array of type
1832: (8)           ``int16`` in C-order has strides ``(8, 2)``. This implies that
1833: (8)           to move from element to element in memory requires jumps of 2 bytes.
1834: (8)           To move from row-to-row, one needs to jump 8 bytes at a time
1835: (8)           (``2 * 4``).
1836: (4)           ctypes : ctypes object
1837: (8)           Class containing properties of the array needed for interaction
1838: (8)           with ctypes.
1839: (4)           base : ndarray
1840: (8)           If the array is a view into another array, that array is its `base`
1841: (8)           (unless that array is also a view). The `base` array is where the
1842: (8)           array data is actually stored.
1843: (4)           See Also
1844: (4)           -----
1845: (4)           array : Construct an array.
1846: (4)           zeros : Create an array, each element of which is zero.
1847: (4)           empty : Create an array, but leave its allocated memory unchanged (i.e.,
1848: (12)             it contains "garbage").
1849: (4)           dtype : Create a data-type.
1850: (4)           numpy.typing.NDArray : An ndarray alias :term:`generic <generic type>`  

1851: (27)             w.r.t. its `dtype.type <numpy.dtype.type>`.
1852: (4)           Notes
1853: (4)           -----
1854: (4)           There are two modes of creating an array using ``__new__``:
1855: (4)           1. If `buffer` is None, then only `shape`, `dtype`, and `order`
1856: (7)             are used.
1857: (4)           2. If `buffer` is an object exposing the buffer interface, then
1858: (7)             all keywords are interpreted.
1859: (4)           No ``__init__`` method is needed because the array is fully initialized
1860: (4)             after the ``__new__`` method.
1861: (4)           Examples
1862: (4)           -----
1863: (4)           These examples illustrate the low-level `ndarray` constructor. Refer
1864: (4)             to the `See Also` section above for easier ways of constructing an
1865: (4)             ndarray.
1866: (4)           First mode, `buffer` is None:
1867: (4)           >>> np.ndarray(shape=(2,2), dtype=float, order='F')
1868: (4)           array([[0.0e+000, 0.0e+000], # random
1869: (11)             [      nan, 2.5e-323]])
1870: (4)           Second mode:
1871: (4)           >>> np.ndarray((2,), buffer=np.array([1,2,3]),
1872: (4)             ...          offset=np.int_().itemsize,
1873: (4)             ...          dtype=int) # offset = 1*itemsize, i.e. skip first element
1874: (4)             array([2, 3])
1875: (4)             """)
1876: (0)             add_newdoc('numpy.core.multiarray', 'ndarray', ('__array_interface__',
1877: (4)               """Array protocol: Python side."""))
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1878: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('__array_priority__',
1879: (4)     """Array priority."""))
1880: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('__array_struct__',
1881: (4)     """Array protocol: C-struct side.""))
1882: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('__dlpack__',
1883: (4)     """a.__dlpack__(*, stream=None)
1884: (4)         DLPack Protocol: Part of the Array API.""))
1885: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('__dlpack_device__',
1886: (4)     """a.__dlpack_device__()
1887: (4)         DLPack Protocol: Part of the Array API.""))
1888: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('base',
1889: (4)     """
1890: (4)         Base object if memory is from some other object.
1891: (4)     Examples
1892: (4)     -----
1893: (4)     The base of an array that owns its memory is None:
1894: (4)     >>> x = np.array([1,2,3,4])
1895: (4)     >>> x.base is None
1896: (4)     True
1897: (4)     Slicing creates a view, whose memory is shared with x:
1898: (4)     >>> y = x[2:]
1899: (4)     >>> y.base is x
1900: (4)     True
1901: (4)     """
1902: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('ctypes',
1903: (4)     """
1904: (4)         An object to simplify the interaction of the array with the ctypes
1905: (4)         module.
1906: (4)         This attribute creates an object that makes it easier to use arrays
1907: (4)         when calling shared libraries with the ctypes module. The returned
1908: (4)         object has, among others, data, shape, and strides attributes (see
1909: (4)         Notes below) which themselves return ctypes objects that can be used
1910: (4)         as arguments to a shared library.
1911: (4)     Parameters
1912: (4)     -----
1913: (4)     None
1914: (4)     Returns
1915: (4)     -----
1916: (4)     c : Python object
1917: (8)     Possessing attributes data, shape, strides, etc.
1918: (4)     See Also
1919: (4)     -----
1920: (4)     numpy.ctypeslib
1921: (4)     Notes
1922: (4)     -----
1923: (4)     Below are the public attributes of this object which were documented
1924: (4)     in "Guide to NumPy" (we have omitted undocumented public attributes,
1925: (4)     as well as documented private attributes):
1926: (4)     .. autoattribute:: numpy.core._internal._ctypes.data
1927: (8)     :noindex:
1928: (4)     .. autoattribute:: numpy.core._internal._ctypes.shape
1929: (8)     :noindex:
1930: (4)     .. autoattribute:: numpy.core._internal._ctypes.strides
1931: (8)     :noindex:
1932: (4)     .. automethod:: numpy.core._internal._ctypes.data_as
1933: (8)     :noindex:
1934: (4)     .. automethod:: numpy.core._internal._ctypes.shape_as
1935: (8)     :noindex:
1936: (4)     .. automethod:: numpy.core._internal._ctypes.strides_as
1937: (8)     :noindex:
1938: (4)     If the ctypes module is not available, then the ctypes attribute
1939: (4)     of array objects still returns something useful, but ctypes objects
1940: (4)     are not returned and errors may be raised instead. In particular,
1941: (4)     the object will still have the ``as_parameter`` attribute which will
1942: (4)     return an integer equal to the data attribute.
1943: (4)     Examples
1944: (4)     -----
1945: (4)     >>> import ctypes
1946: (4)     >>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)

```

```

1947: (4)
1948: (4)
1949: (11)
1950: (4)
1951: (4)
1952: (4)
1953: (4)
1954: (4)
1955: (4)
1956: (4)
1957: (4)
1958: (4)
1959: (4)
1960: (4)
1961: (4)
1962: (4)
1963: (0)
1964: (4)
1965: (0)
1966: (4)
1967: (4)
1968: (4)
1969: (8)
1970: (8)
1971: (8)
1972: (4)
1973: (4)
1974: (4)
1975: (4)
1976: (4)
1977: (4)
1978: (4)
1979: (4)
1980: (4)
type.
1981: (4)
1982: (4)
1983: (4)
1984: (4)
1985: (4)
1986: (4)
1987: (11)
1988: (4)
1989: (4)
1990: (4)
1991: (4)
1992: (4)
1993: (0)
1994: (4)
1995: (4)
1996: (4)
1997: (4)
1998: (4)
1999: (4)
2000: (4)
2001: (4)
2002: (4)
2003: (4)
2004: (0)
2005: (4)
2006: (4)
2007: (4)
2008: (4)
2009: (4)
2010: (4)
2011: (4)
2012: (4)
2013: (4)
2014: (4)

        >>> x
        array([[0, 1],
               [2, 3]], dtype=int32)
        >>> x.ctypes.data
        31962608 # may vary
        >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
        <__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
        >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
        c_uint(0)
        >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
        c_ulong(4294967296)
        >>> x.ctypes.shape
        <numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
        >>> x.ctypes.strides
        <numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
        """))

add_newdoc('numpy.core.multiarray', 'ndarray', ('data',
    """Python buffer object pointing to the start of the array's data."""))
add_newdoc('numpy.core.multiarray', 'ndarray', ('dtype',
    """
        Data-type of the array's elements.
        .. warning::
            Setting ``arr.dtype`` is discouraged and may be deprecated in the
            future. Setting will replace the ``dtype`` without modifying the
            memory (see also `ndarray.view` and `ndarray.astype`).
    Parameters
    -----
    None
    Returns
    -----
    d : numpy dtype object
    See Also
    -----
    ndarray.astype : Cast the values contained in the array to a new data-
    type.
    ndarray.view : Create a view of the same data but a different data-type.
    numpy.dtype
    Examples
    -----
    >>> x
    array([[0, 1],
           [2, 3]])
    >>> x.dtype
    dtype('int32')
    >>> type(x.dtype)
    <type 'numpy.dtype'>
    """))

add_newdoc('numpy.core.multiarray', 'ndarray', ('imag',
    """
        The imaginary part of the array.
    Examples
    -----
    >>> x = np.sqrt([1+0j, 0+1j])
    >>> x.imag
    array([ 0.          ,  0.70710678])
    >>> x.imag.dtype
    dtype('float64')
    """))

add_newdoc('numpy.core.multiarray', 'ndarray', ('itemsize',
    """
        Length of one array element in bytes.
    Examples
    -----
    >>> x = np.array([1,2,3], dtype=np.float64)
    >>> x.itemsize
    8
    >>> x = np.array([1,2,3], dtype=np.complex128)
    >>> x.itemsize
    16

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2015: (4)      """"))
2016: (0)      add_newdoc('numpy.core.multiarray', 'ndarray', ('flags',
2017: (4)          """
2018: (4)              Information about the memory layout of the array.
2019: (4)              Attributes
2020: (4)              -----
2021: (4)              C_CONTIGUOUS (C)
2022: (8)                  The data is in a single, C-style contiguous segment.
2023: (4)              F_CONTIGUOUS (F)
2024: (8)                  The data is in a single, Fortran-style contiguous segment.
2025: (4)              OWNDATA (O)
2026: (8)                  The array owns the memory it uses or borrows it from another object.
2027: (4)              WRITEABLE (W)
2028: (8)                  The data area can be written to. Setting this to False locks
2029: (8)                      the data, making it read-only. A view (slice, etc.) inherits
2030: (8)                          from its base array at creation time, but a view of a writeable
2031: (8)                          array may be subsequently locked while the base array remains
2032: (8)      (The opposite is not true, in that a view of a locked array may not
2033: (8)      be made writeable. However, currently, locking a base object does not
2034: (8)      lock any views that already reference it, so under that circumstance
2035: (8)          is possible to alter the contents of a locked array via a previously
2036: (8)          created writeable view onto it.) Attempting to change a non-writeable
2037: (8)          array raises a RuntimeError exception.
2038: (4)      ALIGNED (A)
2039: (8)          The data and all elements are aligned appropriately for the hardware.
2040: (4)      WRITEBACKIFCOPY (X)
2041: (8)          This array is a copy of some other array. The C-API function
2042: (8)          PyArray_ResolveWritebackIfCopy must be called before deallocated
2043: (8)          to the base array will be updated with the contents of this array.
2044: (4)      FNC
2045: (8)          F_CONTIGUOUS and not C_CONTIGUOUS.
2046: (4)      FORC
2047: (8)          F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
2048: (4)      BEHAVED (B)
2049: (8)          ALIGNED and WRITEABLE.
2050: (4)      CARRAY (CA)
2051: (8)          BEHAVED and C_CONTIGUOUS.
2052: (4)      FARRAY (FA)
2053: (8)          BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.
2054: (4)      Notes
2055: (4)      -----
2056: (4)          The `flags` object can be accessed dictionary-like (as in
``a.flags['WRITEABLE']`),
2057: (4)          or by using lowercased attribute names (as in ``a.flags.writeable``).
2058: (4)          names are only supported in dictionary access.
2059: (4)          Only the WRITEBACKIFCOPY, WRITEABLE, and ALIGNED flags can be
2060: (4)          changed by the user, via direct assignment to the attribute or dictionary
2061: (4)          entry, or by calling `ndarray.setflags`.
2062: (4)          The array flags cannot be set arbitrarily:
2063: (4)          - WRITEBACKIFCOPY can only be set ``False``.
2064: (4)          - ALIGNED can only be set ``True`` if the data is truly aligned.
2065: (4)          - WRITEABLE can only be set ``True`` if the array owns its own memory
2066: (6)          or the ultimate owner of the memory exposes a writeable buffer
2067: (6)          interface or is a string.
2068: (4)          Arrays can be both C-style and Fortran-style contiguous simultaneously.
2069: (4)          This is clear for 1-dimensional arrays, but can also be true for higher
2070: (4)          dimensional arrays.
2071: (4)          Even for contiguous arrays a stride for a given dimension
2072: (4)          ``arr.strides[dim]`` may be *arbitrary* if ``arr.shape[dim] == 1``
2073: (4)          or the array has no elements.
2074: (4)          It does *not* generally hold that ``self.strides[-1] == self.itemsize``
2075: (4)          for C-style contiguous arrays or ``self.strides[0] == self.itemsize`` for
2076: (4)          Fortran-style contiguous arrays is true.
2077: (4)          """
2078: (0)      add_newdoc('numpy.core.multiarray', 'ndarray', ('flat',

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2079: (4)
2080: (4)
2081: (4)
2082: (4)
2083: (4)
2084: (4)
2085: (4)
2086: (4)
2087: (4)
2088: (4)
2089: (4)
2090: (4)
2091: (4)
2092: (11)
2093: (4)
2094: (4)
2095: (4)
2096: (4)
2097: (11)
2098: (11)
2099: (4)
2100: (4)
2101: (4)
2102: (4)
2103: (4)
2104: (4)
2105: (4)
2106: (11)
2107: (4)
2108: (4)
2109: (11)
2110: (4)
2111: (0)
2112: (4)
2113: (4)
2114: (4)
2115: (4)
2116: (4)
2117: (4)
2118: (4)
2119: (4)
2120: (4)
2121: (8)
2122: (8)
2123: (4)
2124: (4)
2125: (4)
2126: (4)
2127: (4)
2128: (4)
2129: (4)
2130: (4)
2131: (0)
2132: (4)
2133: (4)
2134: (4)
2135: (4)
2136: (4)
2137: (4)
2138: (4)
2139: (4)
2140: (4)
2141: (4)
2142: (4)
2143: (0)
2144: (4)
2145: (4)
2146: (4)
2147: (4)

    """
    A 1-D iterator over the array.
    This is a `numpy.flatiter` instance, which acts similarly to, but is not
    a subclass of, Python's built-in iterator object.
    See Also
    -----
    flatten : Return a copy of the array collapsed into one dimension.
    flatiter
    Examples
    -----
    >>> x = np.arange(1, 7).reshape(2, 3)
    >>> x
    array([[1, 2, 3],
           [4, 5, 6]])
    >>> x.flat[3]
    4
    >>> x.T
    array([[1, 4],
           [2, 5],
           [3, 6]])
    >>> x.T.flat[3]
    5
    >>> type(x.flat)
    <class 'numpy.flatiter'>
    An assignment example:
    >>> x.flat = 3; x
    array([[3, 3, 3],
           [3, 3, 3]])
    >>> x.flat[[1,4]] = 1; x
    array([[3, 1, 3],
           [3, 1, 3]])
    """))

add_newdoc('numpy.core.multiarray', 'ndarray', ('nbytes',
"")
    """
    Total bytes consumed by the elements of the array.
    Notes
    -----
    Does not include memory consumed by non-element attributes of the
    array object.
    See Also
    -----
    sys.getsizeof
        Memory consumed by the object itself without parents in case view.
        This does include memory consumed by non-element attributes.
    Examples
    -----
    >>> x = np.zeros((3,5,2), dtype=np.complex128)
    >>> x.nbytes
    480
    >>> np.prod(x.shape) * x.itemsize
    480
    """))

add_newdoc('numpy.core.multiarray', 'ndarray', ('ndim',
"")
    """
    Number of array dimensions.
    Examples
    -----
    >>> x = np.array([1, 2, 3])
    >>> x.ndim
    1
    >>> y = np.zeros((2, 3, 4))
    >>> y.ndim
    3
    """))

add_newdoc('numpy.core.multiarray', 'ndarray', ('real',
"")
    """
    The real part of the array.
    Examples
    -----

```

```

2148: (4)
2149: (4)
2150: (4)
2151: (4)
2152: (4)
2153: (4)
2154: (4)
2155: (4)
2156: (4)
2157: (0)      add_newdoc('numpy.core.multiarray', 'ndarray', ('shape',
2158: (4)          """
2159: (4)          Tuple of array dimensions.
2160: (4)          The shape property is usually used to get the current shape of an array,
2161: (4)          but may also be used to reshape the array in-place by assigning a tuple of
2162: (4)          array dimensions to it. As with `numpy.reshape`, one of the new shape
2163: (4)          dimensions can be -1, in which case its value is inferred from the size of
2164: (4)          the array and the remaining dimensions. Reshaping an array in-place will
2165: (4)          fail if a copy is required.
2166: (4)          .. warning::
2167: (8)          Setting ``arr.shape`` is discouraged and may be deprecated in the
2168: (8)          future. Using `ndarray.reshape` is the preferred approach.
2169: (4)          Examples
2170: (4)          -----
2171: (4)          >>> x = np.array([1, 2, 3, 4])
2172: (4)          >>> x.shape
2173: (4)          (4,)
2174: (4)          >>> y = np.zeros((2, 3, 4))
2175: (4)          >>> y.shape
2176: (4)          (2, 3, 4)
2177: (4)          >>> y.shape = (3, 8)
2178: (4)          >>> y
2179: (4)          array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
2180: (11)             [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
2181: (11)             [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]])
2182: (4)          >>> y.shape = (3, 6)
2183: (4)          Traceback (most recent call last):
2184: (6)              File "<stdin>", line 1, in <module>
2185: (4)          ValueError: total size of new array must be unchanged
2186: (4)          >>> np.zeros((4,2))[:,2].shape = (-1,)
2187: (4)          Traceback (most recent call last):
2188: (6)              File "<stdin>", line 1, in <module>
2189: (4)          AttributeError: Incompatible shape for in-place modification. Use
2190: (4)          `reshape()` to make a copy with the desired shape.
2191: (4)          See Also
2192: (4)          -----
2193: (4)          numpy.shape : Equivalent getter function.
2194: (4)          numpy.reshape : Function similar to setting ``shape``.
2195: (4)          ndarray.reshape : Method similar to setting ``shape``.
2196: (4)          """
2197: (0)      add_newdoc('numpy.core.multiarray', 'ndarray', ('size',
2198: (4)          """
2199: (4)          Number of elements in the array.
2200: (4)          Equal to ``np.prod(a.shape)``, i.e., the product of the array's
2201: (4)          dimensions.
2202: (4)          Notes
2203: (4)          -----
2204: (4)          `a.size` returns a standard arbitrary precision Python integer. This
2205: (4)          may not be the case with other methods of obtaining the same value
2206: (4)          (like the suggested ``np.prod(a.shape)``, which returns an instance
2207: (4)          of ``np.int_``), and may be relevant if the value is used further in
2208: (4)          calculations that may overflow a fixed size integer type.
2209: (4)          Examples
2210: (4)          -----
2211: (4)          >>> x = np.zeros((3, 5, 2), dtype=np.complex128)
2212: (4)          >>> x.size
2213: (4)          30
2214: (4)          >>> np.prod(x.shape)
2215: (4)          30
2216: (4)          """

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2217: (0)
2218: (4)
2219: (4)
2220: (4)
2221: (4)
2222: (8)
2223: (4)
2224: (4)
2225: (4)
2226: (8)
2227: (8)
2228: (8)
2229: (4)
2230: (4)
2231: (4)
2232: (6)
2233: (20)
2234: (4)
2235: (4)
2236: (4)
2237: (4)
2238: (4)
2239: (4)
2240: (4)
2241: (4)
2242: (4)
2243: (4)
2244: (4)
2245: (4)
2246: (4)
2247: (4)
2248: (4)
2249: (12)
2250: (12)
2251: (11)
2252: (12)
2253: (12)
2254: (4)
2255: (4)
2256: (4)
2257: (4)
2258: (4)
2259: (4)
2260: (4)
2261: (4)
2262: (4)
2263: (4)
2264: (4)
2265: (4)
2266: (4)
2267: (4)
2268: (4)
2269: (4)
2270: (4)
2271: (0)
2272: (4)
2273: (4)
2274: (4)
2275: (4)
2276: (4)
2277: (4)
2278: (4)
2279: (4)
2280: (11)
2281: (4)
2282: (4)
2283: (11)
2284: (4)
2285: (4)

add_newdoc('numpy.core.multiarray', 'ndarray', ('strides',
        """
        Tuple of bytes to step in each dimension when traversing an array.
        The byte offset of element ``(i[0], i[1], ..., i[n])`` in an array `a` is::
            offset = sum(np.array(i) * a.strides)
        A more detailed explanation of strides can be found in the
        "ndarray.rst" file in the NumPy reference guide.
        .. warning::
            Setting ``arr.strides`` is discouraged and may be deprecated in the
            future. ``numpy.lib.stride_tricks.as_strided`` should be preferred
            to create a new view of the same data in a safer way.
    Notes
    -----
    Imagine an array of 32-bit integers (each 4 bytes)::

        x = np.array([[0, 1, 2, 3, 4],
                     [5, 6, 7, 8, 9]], dtype=np.int32)
    This array is stored in memory as 40 bytes, one after the other
    (known as a contiguous block of memory). The strides of an array tell
    us how many bytes we have to skip in memory to move to the next position
    along a certain axis. For example, we have to skip 4 bytes (1 value) to
    move to the next column, but 20 bytes (5 values) to get to the same
    position in the next row. As such, the strides for the array `x` will be
    ``(20, 4)``.
    See Also
    -----
    numpy.lib.stride_tricks.as_strided
    Examples
    -----
    >>> y = np.reshape(np.arange(2*3*4), (2,3,4))
    >>> y
    array([[[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]],
          [[12, 13, 14, 15],
           [16, 17, 18, 19],
           [20, 21, 22, 23]]])
    >>> y.strides
    (48, 16, 4)
    >>> y[1,1,1]
    17
    >>> offset=sum(y.strides * np.array((1,1,1)))
    >>> offset/y.itemsize
    17
    >>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
    >>> x.strides
    (32, 4, 224, 1344)
    >>> i = np.array([3,5,2,2])
    >>> offset = sum(i * x.strides)
    >>> x[3,5,2,2]
    813
    >>> offset / x.itemsize
    813
    """)

add_newdoc('numpy.core.multiarray', 'ndarray', ('T',
        """
        View of the transposed array.
        Same as ``self.transpose()``.
    Examples
    -----
    >>> a = np.array([[1, 2], [3, 4]])
    >>> a
    array([[1, 2],
           [3, 4]])
    >>> a.T
    array([[1, 3],
           [2, 4]])
    >>> a = np.array([1, 2, 3, 4])
    >>> a

```

```

2286: (4)           array([1, 2, 3, 4])
2287: (4)           >>> a.T
2288: (4)           array([1, 2, 3, 4])
2289: (4)           See Also
2290: (4)           -----
2291: (4)           transpose
2292: (4)           """))
2293: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__array__',
2294: (4)           """ a.__array__(dtype), /)
2295: (4)           Returns either a new reference to self if dtype is not given or a new
array
2296: (4)           of provided data type if dtype is different from the current dtype of the
2297: (4)           array.
2298: (4)           """))
2299: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__array_finalize__',
2300: (4)           """a.__array_finalize__(obj, /)
2301: (4)           Present so subclasses can call super. Does nothing.
2302: (4)           """))
2303: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__array_prepare__',
2304: (4)           """a.__array_prepare__(array[, context], /)
2305: (4)           Returns a view of `array` with the same type as self.
2306: (4)           """))
2307: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__array_wrap__',
2308: (4)           """a.__array_wrap__(array[, context], /)
2309: (4)           Returns a view of `array` with the same type as self.
2310: (4)           """))
2311: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__copy__',
2312: (4)           """a.__copy__()
2313: (4)           Used if :func:`copy.copy` is called on an array. Returns a copy of the
array.
2314: (4)           Equivalent to ``a.copy(order='K')``.
2315: (4)           """))
2316: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__class_getitem__',
2317: (4)           """a.__class_getitem__(item, /)
2318: (4)           Return a parametrized wrapper around the `~numpy.ndarray` type.
.. versionadded:: 1.22
2319: (4)           Returns
2320: (4)           -----
2321: (4)           alias : types.GenericAlias
2322: (4)           A parametrized `~numpy.ndarray` type.
2323: (8)           Examples
2324: (4)           -----
2325: (4)           >>> from typing import Any
2326: (4)           >>> import numpy as np
2327: (4)           >>> np.ndarray[Any, np.dtype[Any]]
2328: (4)           numpy.ndarray[typing.Any, numpy.dtype[typing.Any]]
2329: (4)           See Also
2330: (4)           -----
2331: (4)           :pep:`585` : Type hinting generics in standard collections.
2332: (4)           numpy.typing.NDArray : An ndarray alias :term:`generic <generic type>`
2333: (4)           w.r.t. its `dtype.type <numpy.dtype.type>` .
2334: (24)           """
2335: (4)           """)
2336: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__deepcopy__',
2337: (4)           """a.__deepcopy__(memo, /)
2338: (4)           Used if :func:`copy.deepcopy` is called on an array.
2339: (4)           """))
2340: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__reduce__',
2341: (4)           """a.__reduce__()
2342: (4)           For pickling.
2343: (4)           """))
2344: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('__setstate__',
2345: (4)           """a.__setstate__(state, /)
2346: (4)           For unpickling.
2347: (4)           The `state` argument must be a sequence that contains the following
elements:
2348: (4)           Parameters
2349: (4)           -----
2350: (4)           version : int
2351: (4)           optional pickle version. If omitted defaults to 0.
2352: (8)

```

```

2353: (4)           shape : tuple
2354: (4)           dtype : data-type
2355: (4)           isFortran : bool
2356: (4)           rawdata : string or list
2357: (8)           a binary string with the data (or a list if 'a' is an object array)
2358: (4)           """"))
2359: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('all',
2360: (4)           """
2361: (4)           a.all(axis=None, out=None, keepdims=False, *, where=True)
2362: (4)           Returns True if all elements evaluate to True.
2363: (4)           Refer to `numpy.all` for full documentation.
2364: (4)           See Also
2365: (4)           -----
2366: (4)           numpy.all : equivalent function
2367: (4)           """"))
2368: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('any',
2369: (4)           """
2370: (4)           a.any(axis=None, out=None, keepdims=False, *, where=True)
2371: (4)           Returns True if any of the elements of `a` evaluate to True.
2372: (4)           Refer to `numpy.any` for full documentation.
2373: (4)           See Also
2374: (4)           -----
2375: (4)           numpy.any : equivalent function
2376: (4)           """"))
2377: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('argmax',
2378: (4)           """
2379: (4)           a.argmax(axis=None, out=None, *, keepdims=False)
2380: (4)           Return indices of the maximum values along the given axis.
2381: (4)           Refer to `numpy.argmax` for full documentation.
2382: (4)           See Also
2383: (4)           -----
2384: (4)           numpy.argmax : equivalent function
2385: (4)           """"))
2386: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('argmin',
2387: (4)           """
2388: (4)           a.argmin(axis=None, out=None, *, keepdims=False)
2389: (4)           Return indices of the minimum values along the given axis.
2390: (4)           Refer to `numpy.argmin` for detailed documentation.
2391: (4)           See Also
2392: (4)           -----
2393: (4)           numpy.argmin : equivalent function
2394: (4)           """"))
2395: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('argsort',
2396: (4)           """
2397: (4)           a.argsort(axis=-1, kind=None, order=None)
2398: (4)           Returns the indices that would sort this array.
2399: (4)           Refer to `numpy.argsort` for full documentation.
2400: (4)           See Also
2401: (4)           -----
2402: (4)           numpy.argsort : equivalent function
2403: (4)           """"))
2404: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('argpartition',
2405: (4)           """
2406: (4)           a.argpartition(kth, axis=-1, kind='introselect', order=None)
2407: (4)           Returns the indices that would partition this array.
2408: (4)           Refer to `numpy.argpartition` for full documentation.
2409: (4)           .. versionadded:: 1.8.0
2410: (4)           See Also
2411: (4)           -----
2412: (4)           numpy.argpartition : equivalent function
2413: (4)           """"))
2414: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('astype',
2415: (4)           """
2416: (4)           a.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)
2417: (4)           Copy of the array, cast to a specified type.
2418: (4)           Parameters
2419: (4)           -----
2420: (4)           dtype : str or dtype
2421: (8)           Typecode or data-type to which the array is cast.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2422: (4)          order : {'C', 'F', 'A', 'K'}, optional
2423: (8)          Controls the memory layout order of the result.
2424: (8)          'C' means C order, 'F' means Fortran order, 'A'
2425: (8)          means 'F' order if all the arrays are Fortran contiguous,
2426: (8)          'C' order otherwise, and 'K' means as close to the
2427: (8)          order the array elements appear in memory as possible.
2428: (8)          Default is 'K'.
2429: (4)          casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
2430: (8)          Controls what kind of data casting may occur. Defaults to 'unsafe'
2431: (8)          for backwards compatibility.
2432: (10)         * 'no' means the data types should not be cast at all.
2433: (10)         * 'equiv' means only byte-order changes are allowed.
2434: (10)         * 'safe' means only casts which can preserve values are allowed.
2435: (10)         * 'same_kind' means only safe casts or casts within a kind,
2436: (12)          like float64 to float32, are allowed.
2437: (10)         * 'unsafe' means any data conversions may be done.
2438: (4)          subok : bool, optional
2439: (8)          If True, then sub-classes will be passed-through (default), otherwise
2440: (8)          the returned array will be forced to be a base-class array.
2441: (4)          copy : bool, optional
2442: (8)          By default, astype always returns a newly allocated array. If this
2443: (8)          is set to false, and the `dtype`, `order`, and `subok`
2444: (8)          requirements are satisfied, the input array is returned instead
2445: (8)          of a copy.
2446: (4)          Returns
2447: (4)          -----
2448: (4)          arr_t : ndarray
2449: (8)          Unless `copy` is False and the other conditions for returning the
input
2450: (8)          array are satisfied (see description for `copy` input parameter),
`arr_t`
2451: (8)          is a new array of the same shape as the input array, with dtype, order
2452: (8)          given by `dtype`, `order`.
2453: (4)          Notes
2454: (4)          -----
2455: (4)          .. versionchanged:: 1.17.0
2456: (7)          Casting between a simple data type and a structured one is possible
only
2457: (7)          for "unsafe" casting. Casting to multiple fields is allowed, but
2458: (7)          casting from multiple fields is not.
2459: (4)          .. versionchanged:: 1.9.0
2460: (7)          Casting from numeric to string types in 'safe' casting mode requires
2461: (7)          that the string dtype length is long enough to store the max
2462: (7)          integer/float value converted.
2463: (4)          Raises
2464: (4)          -----
2465: (4)          ComplexWarning
2466: (8)          When casting from complex to float or int. To avoid this,
2467: (8)          one should use ``a.real.astype(t)``.
2468: (4)          Examples
2469: (4)          -----
2470: (4)          >>> x = np.array([1, 2, 2.5])
2471: (4)          >>> x
2472: (4)          array([1., 2., 2.5])
2473: (4)          >>> x.astype(int)
2474: (4)          array([1, 2, 2])
2475: (4)          "''")
2476: (0)          add_newdoc('numpy.core.multiarray', 'ndarray', ('byteswap',
2477: (4)          """"
2478: (4)          a.byteswap(inplace=False)
2479: (4)          Swap the bytes of the array elements
2480: (4)          Toggle between low-endian and big-endian data representation by
2481: (4)          returning a byteswapped array, optionally swapped in-place.
2482: (4)          Arrays of byte-strings are not swapped. The real and imaginary
2483: (4)          parts of a complex number are swapped individually.
2484: (4)          Parameters
2485: (4)          -----
2486: (4)          inplace : bool, optional
2487: (8)          If ``True``, swap bytes in-place, default is ``False``.

```

```

2488: (4)          Returns
2489: (4)          -----
2490: (4)          out : ndarray
2491: (8)          The byteswapped array. If `inplace` is ``True``, this is
2492: (8)          a view to self.
2493: (4)          Examples
2494: (4)          -----
2495: (4)          >>> A = np.array([1, 256, 8755], dtype=np.int16)
2496: (4)          >>> list(map(hex, A))
2497: (4)          ['0x1', '0x100', '0x2233']
2498: (4)          >>> A.byteswap(inplace=True)
2499: (4)          array([ 256,      1, 13090], dtype=int16)
2500: (4)          >>> list(map(hex, A))
2501: (4)          ['0x100', '0x1', '0x3322']
2502: (4)          Arrays of byte-strings are not swapped
2503: (4)          >>> A = np.array([b'ceg', b'fac'])
2504: (4)          >>> A.byteswap()
2505: (4)          array([b'ceg', b'fac'], dtype='|S3')
2506: (4)          ``A.newbyteorder().byteswap()`` produces an array with the same values
2507: (6)          but different representation in memory
2508: (4)          >>> A = np.array([1, 2, 3])
2509: (4)          >>> A.view(np.uint8)
2510: (4)          array([1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
2511: (11)          0, 0], dtype=uint8)
2512: (4)          >>> A.newbyteorder().byteswap(inplace=True)
2513: (4)          array([1, 2, 3])
2514: (4)          >>> A.view(np.uint8)
2515: (4)          array([0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
2516: (11)          0, 3], dtype=uint8)
2517: (4)          """))
2518: (0)          add_newdoc('numpy.core.multiarray', 'ndarray', ('choose',
2519: (4)          """
2520: (4)          a.choose(choices, out=None, mode='raise')
2521: (4)          Use an index array to construct a new array from a set of choices.
2522: (4)          Refer to `numpy.choose` for full documentation.
2523: (4)          See Also
2524: (4)          -----
2525: (4)          numpy.choose : equivalent function
2526: (4)          """
2527: (0)          add_newdoc('numpy.core.multiarray', 'ndarray', ('clip',
2528: (4)          """
2529: (4)          a.clip(min=None, max=None, out=None, **kwargs)
2530: (4)          Return an array whose values are limited to ``[min, max]``.
2531: (4)          One of max or min must be given.
2532: (4)          Refer to `numpy.clip` for full documentation.
2533: (4)          See Also
2534: (4)          -----
2535: (4)          numpy.clip : equivalent function
2536: (4)          """
2537: (0)          add_newdoc('numpy.core.multiarray', 'ndarray', ('compress',
2538: (4)          """
2539: (4)          a.compress(condition, axis=None, out=None)
2540: (4)          Return selected slices of this array along given axis.
2541: (4)          Refer to `numpy.compress` for full documentation.
2542: (4)          See Also
2543: (4)          -----
2544: (4)          numpy.compress : equivalent function
2545: (4)          """
2546: (0)          add_newdoc('numpy.core.multiarray', 'ndarray', ('conj',
2547: (4)          """
2548: (4)          a.conj()
2549: (4)          Complex-conjugate all elements.
2550: (4)          Refer to `numpy.conjugate` for full documentation.
2551: (4)          See Also
2552: (4)          -----
2553: (4)          numpy.conjugate : equivalent function
2554: (4)          """
2555: (0)          add_newdoc('numpy.core.multiarray', 'ndarray', ('conjugate',
2556: (4)          """

```

```

2557: (4)           a.conjugate()
2558: (4)           Return the complex conjugate, element-wise.
2559: (4)           Refer to `numpy.conjugate` for full documentation.
2560: (4)           See Also
2561: (4)           -----
2562: (4)           numpy.conjugate : equivalent function
2563: (4)           """
2564: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('copy',
2565: (4)           """
2566: (4)           a.copy(order='C')
2567: (4)           Return a copy of the array.
2568: (4)           Parameters
2569: (4)           -----
2570: (4)           order : {'C', 'F', 'A', 'K'}, optional
2571: (8)           Controls the memory layout of the copy. 'C' means C-order,
2572: (8)           'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
2573: (8)           'C' otherwise. 'K' means match the layout of `a` as closely
2574: (8)           as possible. (Note that this function and :func:`numpy.copy` are very
2575: (8)           similar but have different default values for their order=
2576: (8)           arguments, and this function always passes sub-classes through.)
2577: (4)           See also
2578: (4)           -----
2579: (4)           numpy.copy : Similar function with different default behavior
2580: (4)           numpy.copyto
2581: (4)           Notes
2582: (4)           -----
2583: (4)           This function is the preferred method for creating an array copy. The
2584: (4)           function :func:`numpy.copy` is similar, but it defaults to using order
2585: (4)           'K',
2586: (4)           and will not pass sub-classes through by default.
2587: (4)           Examples
2588: (4)           >>> x = np.array([[1,2,3],[4,5,6]], order='F')
2589: (4)           >>> y = x.copy()
2590: (4)           >>> x.fill(0)
2591: (4)           >>> x
2592: (4)           array([[0, 0, 0],
2593: (11)             [0, 0, 0]])
2594: (4)           >>> y
2595: (4)           array([[1, 2, 3],
2596: (11)             [4, 5, 6]])
2597: (4)           >>> y.flags['C_CONTIGUOUS']
2598: (4)           True
2599: (4)           """
2600: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('cumprod',
2601: (4)           """
2602: (4)           a.cumprod(axis=None, dtype=None, out=None)
2603: (4)           Return the cumulative product of the elements along the given axis.
2604: (4)           Refer to `numpy.cumprod` for full documentation.
2605: (4)           See Also
2606: (4)           -----
2607: (4)           numpy.cumprod : equivalent function
2608: (4)           """
2609: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('cumsum',
2610: (4)           """
2611: (4)           a.cumsum(axis=None, dtype=None, out=None)
2612: (4)           Return the cumulative sum of the elements along the given axis.
2613: (4)           Refer to `numpy.cumsum` for full documentation.
2614: (4)           See Also
2615: (4)           -----
2616: (4)           numpy.cumsum : equivalent function
2617: (4)           """
2618: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('diagonal',
2619: (4)           """
2620: (4)           a.diagonal(offset=0, axis1=0, axis2=1)
2621: (4)           Return specified diagonals. In NumPy 1.9 the returned array is a
2622: (4)           read-only view instead of a copy as in previous NumPy versions. In
2623: (4)           a future version the read-only restriction will be removed.
2624: (4)           Refer to :func:`numpy.diagonal` for full documentation.

```

```

2625: (4)           See Also
2626: (4)           -----
2627: (4)           numpy.diagonal : equivalent function
2628: (4)
2629: (0)           """
2630: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('dot'))
2631: (4)           add_newdoc('numpy.core.multiarray', 'ndarray', ('dump',
2632: (4)             """a.dump(file)
2633: (4)             Dump a pickle of the array to the specified file.
2634: (4)             The array can be read back with pickle.load or numpy.load.
2635: (4)             Parameters
2636: (4)               -----
2637: (8)                 file : str or Path
2638: (8)                   A string naming the dump file.
2639: (12)                  .. versionchanged:: 1.17.0
2640: (4)                     `pathlib.Path` objects are now accepted.
2641: (0)           """
2642: (4)
2643: (4)           a.dumps()
2644: (4)           Returns the pickle of the array as a string.
2645: (4)           pickle.loads will convert the string back to an array.
2646: (4)           Parameters
2647: (4)             -----
2648: (4)               None
2649: (4)
2650: (0)           """
2651: (4)
2652: (4)           add_newdoc('numpy.core.multiarray', 'ndarray', ('fill',
2653: (4)             """
2654: (4)               a.fill(value)
2655: (4)               Fill the array with a scalar value.
2656: (4)               Parameters
2657: (4)                 -----
2658: (4)                   value : scalar
2659: (4)                     All elements of `a` will be assigned this value.
2660: (4)           Examples
2661: (4)             -----
2662: (4)               >>> a = np.array([1, 2])
2663: (4)               >>> a.fill(0)
2664: (4)               >>> a
2665: (4)               array([0, 0])
2666: (4)               >>> a = np.empty(2)
2667: (4)               >>> a.fill(1)
2668: (4)               >>> a
2669: (4)               array([1., 1.])
2670: (4)               Fill expects a scalar value and always behaves the same as assigning
2671: (4)               to a single array element. The following is a rare example where this
2672: (4)               distinction is important:
2673: (4)               >>> a = np.array([None, None], dtype=object)
2674: (4)               >>> a[0] = np.array(3)
2675: (4)               >>> a
2676: (4)               array([array(3), None], dtype=object)
2677: (4)               >>> a.fill(np.array(3))
2678: (4)               >>> a
2679: (4)               array([array(3), array(3)], dtype=object)
2680: (4)               Where other forms of assignments will unpack the array being assigned:
2681: (4)               >>> a[...] = np.array(3)
2682: (4)
2683: (0)           """
2684: (4)
2685: (4)           add_newdoc('numpy.core.multiarray', 'ndarray', ('flatten',
2686: (4)             """
2687: (4)               a.flatten(order='C')
2688: (4)               Return a copy of the array collapsed into one dimension.
2689: (4)               Parameters
2690: (8)                 -----
2691: (8)                   order : {'C', 'F', 'A', 'K'}, optional
2692: (8)                     'C' means to flatten in row-major (C-style) order.
2693: (8)                     'F' means to flatten in column-major (Fortran-
2694: (8)                     style) order. 'A' means to flatten in column-major
2695: (8)                     order if `a` is Fortran *contiguous* in memory,

```

```

2694: (8)           row-major order otherwise. 'K' means to flatten
2695: (8)           `a` in the order the elements occur in memory.
2696: (8)           The default is 'C'.
2697: (4)           Returns
2698: (4)           -----
2699: (4)           y : ndarray
2700: (8)           A copy of the input array, flattened to one dimension.
2701: (4)           See Also
2702: (4)           -----
2703: (4)           ravel : Return a flattened array.
2704: (4)           flat : A 1-D flat iterator over the array.
2705: (4)           Examples
2706: (4)           -----
2707: (4)           >>> a = np.array([[1,2], [3,4]])
2708: (4)           >>> a.flatten()
2709: (4)           array([1, 2, 3, 4])
2710: (4)           >>> a.flatten('F')
2711: (4)           array([1, 3, 2, 4])
2712: (4)           "''")
2713: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('getfield',
2714: (4)           """"
2715: (4)           a.getfield(dtype, offset=0)
2716: (4)           Returns a field of the given array as a certain type.
2717: (4)           A field is a view of the array data with a given data-type. The values in
2718: (4)           the view are determined by the given type and the offset into the current
2719: (4)           array in bytes. The offset needs to be such that the view dtype fits in
the
2720: (4)           array dtype; for example an array of dtype complex128 has 16-byte
elements.
2721: (4)           If taking a view with a 32-bit integer (4 bytes), the offset needs to be
2722: (4)           between 0 and 12 bytes.
2723: (4)           Parameters
2724: (4)           -----
2725: (4)           dtype : str or dtype
2726: (8)           The data type of the view. The dtype size of the view can not be
larger
2727: (8)           than that of the array itself.
2728: (4)           offset : int
2729: (8)           Number of bytes to skip before beginning the element view.
2730: (4)           Examples
2731: (4)           -----
2732: (4)           >>> x = np.diag([1.+1.j]*2)
2733: (4)           >>> x[1, 1] = 2 + 4.j
2734: (4)           >>> x
2735: (4)           array([[1.+1.j,  0.+0.j],
2736: (11)           [0.+0.j,  2.+4.j]])
2737: (4)           >>> x.getfield(np.float64)
2738: (4)           array([[1.,  0.],
2739: (11)           [0.,  2.]])
2740: (4)           By choosing an offset of 8 bytes we can select the complex part of the
2741: (4)           array for our view:
2742: (4)           >>> x.getfield(np.float64, offset=8)
2743: (4)           array([[1.,  0.],
2744: (11)           [0.,  4.]])
2745: (4)           """")
2746: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('item',
2747: (4)           """"
2748: (4)           a.item(*args)
2749: (4)           Copy an element of an array to a standard Python scalar and return it.
2750: (4)           Parameters
2751: (4)           -----
2752: (4)           \\\*args : Arguments (variable number and type)
2753: (8)           * none: in this case, the method only works for arrays
2754: (10)           with one element (`a.size == 1`), which element is
2755: (10)           copied into a standard Python scalar object and returned.
2756: (8)           * int_type: this argument is interpreted as a flat index into
2757: (10)           the array, specifying which element to copy and return.
2758: (8)           * tuple of int_types: functions as does a single int_type argument,
2759: (10)           except that the argument is interpreted as an nd-index into the

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2760: (10)           array.
2761: (4)           Returns
2762: (4)
2763: (4)           z : Standard Python scalar object
2764: (8)           A copy of the specified element of the array as a suitable
2765: (8)           Python scalar
2766: (4)           Notes
2767: (4)
2768: (4)           When the data type of `a` is longdouble or clongdouble, item() returns
2769: (4)           a scalar array object because there is no available Python scalar that
2770: (4)           would not lose information. Void arrays return a buffer object for item(),
2771: (4)           unless fields are defined, in which case a tuple is returned.
2772: (4)           `item` is very similar to a[args], except, instead of an array scalar,
2773: (4)           a standard Python scalar is returned. This can be useful for speeding up
2774: (4)           access to elements of the array and doing arithmetic on elements of the
2775: (4)           array using Python's optimized math.
2776: (4)           Examples
2777: (4)
2778: (4)           >>> np.random.seed(123)
2779: (4)           >>> x = np.random.randint(9, size=(3, 3))
2780: (4)           >>> x
2781: (4)           array([[2, 2, 6],
2782: (11)             [1, 3, 6],
2783: (11)             [1, 0, 1]])
2784: (4)           >>> x.item(3)
2785: (4)           1
2786: (4)           >>> x.item(7)
2787: (4)           0
2788: (4)           >>> x.item((0, 1))
2789: (4)           2
2790: (4)           >>> x.item((2, 2))
2791: (4)           1
2792: (4)           ""))
2793: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('itemset',
2794: (4)           """
2795: (4)           a.itemset(*args)
2796: (4)           Insert scalar into an array (scalar is cast to array's dtype, if possible)
2797: (4)           There must be at least 1 argument, and define the last argument
2798: (4)           as *item*. Then, ``a.itemset(*args)`` is equivalent to but faster
2799: (4)           than ``a[args] = item``. The item should be a scalar value and `args`
2800: (4)           must select a single item in the array `a`.
2801: (4)           Parameters
2802: (4)
2803: (4)           \\*args : Arguments
2804: (8)           If one argument: a scalar, only used in case `a` is of size 1.
2805: (8)           If two arguments: the last argument is the value to be set
2806: (8)           and must be a scalar, the first argument specifies a single array
2807: (8)           element location. It is either an int or a tuple.
2808: (4)           Notes
2809: (4)
2810: (4)           Compared to indexing syntax, `itemset` provides some speed increase
2811: (4)           for placing a scalar into a particular location in an `ndarray`,
2812: (4)           if you must do this. However, generally this is discouraged:
2813: (4)           among other problems, it complicates the appearance of the code.
2814: (4)           Also, when using `itemset` (and `item`) inside a loop, be sure
2815: (4)           to assign the methods to a local variable to avoid the attribute
2816: (4)           look-up at each loop iteration.
2817: (4)           Examples
2818: (4)
2819: (4)           >>> np.random.seed(123)
2820: (4)           >>> x = np.random.randint(9, size=(3, 3))
2821: (4)           >>> x
2822: (4)           array([[2, 2, 6],
2823: (11)             [1, 3, 6],
2824: (11)             [1, 0, 1]])
2825: (4)           >>> x.itemset(4, 0)
2826: (4)           >>> x.itemset((2, 2), 9)
2827: (4)           >>> x
2828: (4)           array([[2, 2, 6],

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```
2829: (11) [1, 0, 6],  
2830: (11) [1, 0, 9]])  
2831: (4) """))  
2832: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('max',  
2833: (4) """  
2834: (4) a.max(axis=None, out=None, keepdims=False, initial=<no value>, where=True)  
2835: (4) Returns the maximum along a given axis.  
2836: (4) Refer to `numpy.amax` for full documentation.  
2837: (4) See Also  
2838: (4) -----  
2839: (4) numpyamax : equivalent function  
2840: (4) """))  
2841: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('mean',  
2842: (4) """  
2843: (4) a.mean(axis=None, dtype=None, out=None, keepdims=False, *, where=True)  
2844: (4) Returns the average of the array elements along given axis.  
2845: (4) Refer to `numpy.mean` for full documentation.  
2846: (4) See Also  
2847: (4) -----  
2848: (4) numpy.mean : equivalent function  
2849: (4) """))  
2850: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('min',  
2851: (4) """  
2852: (4) a.min(axis=None, out=None, keepdims=False, initial=<no value>, where=True)  
2853: (4) Returns the minimum along a given axis.  
2854: (4) Refer to `numpy.amin` for full documentation.  
2855: (4) See Also  
2856: (4) -----  
2857: (4) numpy.amin : equivalent function  
2858: (4) """))  
2859: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('newbyteorder',  
2860: (4) """  
2861: (4) arr.newbyteorder(new_order='S', /)  
2862: (4) Returns the array with the same data viewed with a different byte order.  
2863: (4) Equivalent to::  
2864: (8) arr.view(arr.dtype.newbytorder(new_order))  
2865: (4) Changes are also made in all fields and sub-arrays of the array data  
2866: (4) type.  
2867: (4) Parameters  
2868: (4) -----  
2869: (4) new_order : string, optional  
2870: (8) Byte order to force; a value from the byte order specifications  
2871: (8) below. `new_order` codes can be any of:  
2872: (8) * 'S' - swap dtype from current to opposite endian  
2873: (8) * {'<', 'little'} - little endian  
2874: (8) * {'>', 'big'} - big endian  
2875: (8) * {'=', 'native'} - native order, equivalent to `sys.byteorder`  
2876: (8) * {'|', 'I'} - ignore (no change to byte order)  
2877: (8) The default value ('S') results in swapping the current  
2878: (8) byte order.  
2879: (4) Returns  
2880: (4) -----  
2881: (4) new_arr : array  
2882: (8) New array object with the dtype reflecting given change to the  
2883: (8) byte order.  
2884: (4) """))  
2885: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('nonzero',  
2886: (4) """  
2887: (4) a.nonzero()  
2888: (4) Returns the indices of the elements that are non-zero.  
2889: (4) Refer to `numpy.nonzero` for full documentation.  
2890: (4) See Also  
2891: (4) -----  
2892: (4) numpy.nonzero : equivalent function  
2893: (4) """))  
2894: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('prod',  
2895: (4) """  
2896: (4) a.prod(axis=None, dtype=None, out=None, keepdims=False, initial=1,  
where=True)
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

2897: (4)       Return the product of the array elements over the given axis  
2898: (4)       Refer to `numpy.prod` for full documentation.  
2899: (4)       See Also  
2900: (4)  
2901: (4)       -----  
2902: (4)        numpy.prod : equivalent function  
2903: (0)        """")  
2904: (4)        add\_newdoc('numpy.core.multiarray', 'ndarray', ('ptp',  
2905: (4)        """  
2906: (4)        a.ptp(axis=None, out=None, keepdims=False)  
2907: (4)        Peak to peak (maximum - minimum) value along a given axis.  
2908: (4)        Refer to `numpy.ptp` for full documentation.  
2909: (4)        See Also  
2910: (4)  
2911: (4)        -----  
2912: (0)        numpy.ptp : equivalent function  
2913: (4)        """")  
2914: (4)        add\_newdoc('numpy.core.multiarray', 'ndarray', ('put',  
2915: (4)        """  
2916: (4)        a.put(indices, values, mode='raise')  
2917: (4)        Set ``a.flat[n] = values[n]`` for all `n` in indices.  
2918: (4)        Refer to `numpy.put` for full documentation.  
2919: (4)        See Also  
2920: (4)  
2921: (0)        -----  
2922: (4)        numpy.put : equivalent function  
2923: (4)        """")  
2924: (4)        add\_newdoc('numpy.core.multiarray', 'ndarray', ('ravel',  
2925: (4)        """  
2926: (4)        a.ravel([order])  
2927: (4)        Return a flattened array.  
2928: (4)        Refer to `numpy.ravel` for full documentation.  
2929: (4)        See Also  
2930: (4)  
2931: (0)        -----  
2932: (4)        numpy.ravel : equivalent function  
2933: (4)        ndarray.flat : a flat iterator on the array.  
2934: (4)        """")  
2935: (4)        add\_newdoc('numpy.core.multiarray', 'ndarray', ('repeat',  
2936: (4)        """  
2937: (4)        a.repeat(repeats, axis=None)  
2938: (4)        Repeat elements of an array.  
2939: (4)        Refer to `numpy.repeat` for full documentation.  
2940: (0)        See Also  
2941: (4)  
2942: (4)  
2943: (4)        -----  
2944: (4)        numpy.repeat : equivalent function  
2945: (4)        """")  
2946: (4)        add\_newdoc('numpy.core.multiarray', 'ndarray', ('reshape',  
2947: (4)        """  
2948: (4)        a.reshape(shape, order='C')  
2949: (4)        Returns an array containing the same data with a new shape.  
2950: (4)        Refer to `numpy.reshape` for full documentation.  
2951: (4)        See Also  
2952: (4)  
2953: (4)        -----  
2954: (4)        numpy.reshape : equivalent function  
2955: (0)        Notes  
2956: (4)  
2957: (4)        -----  
2958: (4)        Unlike the free function `numpy.reshape`, this method on `ndarray` allows  
2959: (4)        the elements of the shape parameter to be passed in as separate arguments.  
2960: (4)        For example, ``a.reshape(10, 11)`` is equivalent to  
2961: (4)        ```a.reshape((10, 11))```.  
2962: (8)  
2963: (4)        """")  
2964: (8)        add\_newdoc('numpy.core.multiarray', 'ndarray', ('resize',  
2965: (4)        """  
2966: (4)        a.resize(new\_shape, refcheck=True)  
2967: (4)        Change shape and size of array in-place.  
2968: (4)        Parameters  
2969: (4)  
2970: (4)        -----  
2971: (4)        new\_shape : tuple of ints, or `n` ints  
2972: (4)        Shape of resized array.  
2973: (4)        refcheck : bool, optional  
2974: (4)        If False, reference count will not be checked. Default is True.  
2975: (4)        Returns

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2966: (4)
2967: (4)
2968: (4)
2969: (4)
2970: (4)
2971: (8)      -----
2972: (8)      None
2973: (8)      Raises
2974: (8)      -----
2975: (8)      ValueError
2976: (8)      If `a` does not own its own data or references or views to it exist,
2977: (8)      and the data memory must be changed.
2978: (8)      PyPy only: will always raise if the data memory must be changed, since
2979: (8)      there is no reliable way to determine if references or views to it
2980: (8)      exist.
2981: (8)      -----
2982: (8)      SystemError
2983: (8)      If the `order` keyword argument is specified. This behaviour is a
2984: (8)      bug in NumPy.
2985: (8)      See Also
2986: (8)      -----
2987: (8)      resize : Return a new array with the specified shape.
2988: (8)      Notes
2989: (8)      -----
2990: (8)      This reallocates space for the data area if necessary.
2991: (8)      Only contiguous arrays (data elements consecutive in memory) can be
2992: (8)      resized.
2993: (8)      The purpose of the reference count check is to make sure you
2994: (8)      do not use this array as a buffer for another Python object and then
2995: (8)      reallocate the memory. However, reference counts can increase in
2996: (8)      other ways so if you are sure that you have not shared the memory
2997: (8)      for this array with another Python object, then you may safely set
2998: (8)      `refcheck` to False.
2999: (8)      Examples
3000: (8)      -----
3001: (11)     Shrinking an array: array is flattened (in the order that the data are
3002: (4)      stored in memory), resized, and reshaped:
3003: (4)      >>> a = np.array([[0, 1], [2, 3]], order='C')
3004: (4)      >>> a.resize((2, 1))
3005: (4)      >>> a
3006: (11)     array([[0],
3007: (4)      [1]])
3008: (4)      >>> a = np.array([[0, 1], [2, 3]], order='F')
3009: (4)      >>> a.resize((2, 1))
3010: (4)      >>> a
3011: (4)      array([[0],
3012: (11)     [1]])
3013: (4)      Enlarging an array: as above, but missing entries are filled with zeros:
3014: (4)      >>> b = np.array([[0, 1], [2, 3]])
3015: (4)      >>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
3016: (4)      >>> b
3017: (4)      array([[0, 1, 2],
3018: (4)      [3, 0, 0]])
3019: (4)      Referencing an array prevents resizing...
3020: (4)      >>> c = a
3021: (4)      >>> a.resize((1, 1))
3022: (4)      Traceback (most recent call last):
3023: (4)      ...
3024: (4)      ValueError: cannot resize an array that references or is referenced ...
3025: (4)      Unless `refcheck` is False:
3026: (0)      >>> a.resize((1, 1), refcheck=False)
3027: (4)      >>> a
3028: (4)      array([[0]])
3029: (4)      >>> c
3030: (4)      array([[0]])
3031: (4)      """
3032: (4)      add_newdoc('numpy.core.multiarray', 'ndarray', ('round',
3033: (4)      """
3034: (4)      a.round(decimals=0, out=None)
3035: (4)      Return `a` with each element rounded to the given number of decimals.
3036: (4)      Refer to `numpy.around` for full documentation.
3037: (4)      See Also
3038: (4)      -----
3039: (4)      numpy.around : equivalent function
3040: (4)      """

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3035: (0)
3036: (4)
3037: (4)
3038: (4)
order.
3039: (4)
3040: (4)
3041: (4)
3042: (4)
3043: (4)
3044: (0)
3045: (4)
3046: (4)
3047: (4)
3048: (4)
3049: (4)
3050: (4)
3051: (4)
3052: (4)
3053: (8)
3054: (4)
3055: (8)
3056: (4)
3057: (8)
3058: (4)
3059: (4)
3060: (4)
3061: (4)
3062: (4)
3063: (4)
3064: (4)
3065: (4)
3066: (4)
3067: (4)
3068: (4)
3069: (11)
3070: (11)
3071: (4)
3072: (4)
3073: (4)
3074: (11)
3075: (11)
3076: (4)
3077: (4)
3078: (11)
3079: (11)
3080: (4)
3081: (4)
3082: (4)
3083: (11)
3084: (11)
3085: (4)
3086: (0)
3087: (4)
3088: (4)
3089: (4)
3090: (4)
3091: (4)
3092: (4)
3093: (4)
3094: (4)
3095: (4)
3096: (4)
3097: (4)
3098: (4)
3099: (4)
3100: (4)
3101: (4)
3102: (8)

add_newdoc('numpy.core.multiarray', 'ndarray', ('searchsorted',
        """
            a.searchsorted(v, side='left', sorter=None)
            Find indices where elements of v should be inserted in a to maintain
            order.
            For full documentation, see `numpy.searchsorted`"
            See Also
            -----
            numpy.searchsorted : equivalent function
            """))
add_newdoc('numpy.core.multiarray', 'ndarray', ('setfield',
        """
            a.setfield(val, dtype, offset=0)
            Put a value into a specified place in a field defined by a data-type.
            Place `val` into `a`'s field defined by `dtype` and beginning `offset`
            bytes into the field.
            Parameters
            -----
            val : object
                Value to be placed in field.
            dtype : dtype object
                Data-type of the field in which to place `val`.
            offset : int, optional
                The number of bytes into the field at which to place `val`.
            Returns
            -----
            None
            See Also
            -----
            getfield
            Examples
            -----
            >>> x = np.eye(3)
            >>> x.getfield(np.float64)
            array([[1.,  0.,  0.],
                   [0.,  1.,  0.],
                   [0.,  0.,  1.]])
            >>> x.setfield(3, np.int32)
            >>> x.getfield(np.int32)
            array([[3, 3, 3],
                   [3, 3, 3],
                   [3, 3, 3]], dtype=int32)
            >>> x
            array([[1.0e+000, 1.5e-323, 1.5e-323],
                   [1.5e-323, 1.0e+000, 1.5e-323],
                   [1.5e-323, 1.5e-323, 1.0e+000]])
            >>> x.setfield(np.eye(3), np.int32)
            >>> x
            array([[1.,  0.,  0.],
                   [0.,  1.,  0.],
                   [0.,  0.,  1.]])
        """))
add_newdoc('numpy.core.multiarray', 'ndarray', ('setflags',
        """
            a.setflags(write=None, align=None, uic=None)
            Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY,
            respectively.
            These Boolean-valued flags affect how numpy interprets the memory
            area used by `a` (see Notes below). The ALIGNED flag can only
            be set to True if the data is actually aligned according to the type.
            The WRITEBACKIFCOPY and flag can never be set
            to True. The flag WRITEABLE can only be set to True if the array owns its
            own memory, or the ultimate owner of the memory exposes a writeable buffer
            interface, or is a string. (The exception for string is made so that
            unpickling can be done without copying memory.)
            Parameters
            -----
            write : bool, optional
                Describes whether or not `a` can be written to.
        """))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3103: (4) align : bool, optional
3104: (8)     Describes whether or not `a` is aligned properly for its type.
3105: (4) uic : bool, optional
3106: (8)     Describes whether or not `a` is a copy of another "base" array.
3107: (4) Notes
3108: (4) -----
3109: (4) Array flags provide information about how the memory area used
3110: (4) for the array is to be interpreted. There are 7 Boolean flags
3111: (4) in use, only four of which can be changed by the user:
3112: (4) WRITEBACKIFCOPY, WRITEABLE, and ALIGNED.
3113: (4) WRITEABLE (W) the data area can be written to;
3114: (4) ALIGNED (A) the data and strides are aligned appropriately for the
hardware
3115: (4) (as determined by the compiler);
3116: (4) WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced
3117: (4) by .base). When the C-API function PyArray_ResolveWritebackIfCopy is
3118: (4) called, the base array will be updated with the contents of this array.
3119: (4) All flags can be accessed using the single (upper case) letter as well
3120: (4) as the full name.
3121: (4) Examples
3122: (4) -----
3123: (4) >>> y = np.array([[3, 1, 7],
3124: (4) ...           [2, 0, 0],
3125: (4) ...           [8, 5, 9]])
3126: (4) >>> y
3127: (4) array([[3, 1, 7],
3128: (11) ...           [2, 0, 0],
3129: (11) ...           [8, 5, 9]])
3130: (4) >>> y.flags
3131: (6)     C_CONTIGUOUS : True
3132: (6)     F_CONTIGUOUS : False
3133: (6)     OWNDATA : True
3134: (6)     WRITEABLE : True
3135: (6)     ALIGNED : True
3136: (6)     WRITEBACKIFCOPY : False
3137: (4) >>> y.setflags(write=0, align=0)
3138: (4) >>> y.flags
3139: (6)     C_CONTIGUOUS : True
3140: (6)     F_CONTIGUOUS : False
3141: (6)     OWNDATA : True
3142: (6)     WRITEABLE : False
3143: (6)     ALIGNED : False
3144: (6)     WRITEBACKIFCOPY : False
3145: (4) >>> y.setflags(uic=1)
3146: (4) Traceback (most recent call last):
3147: (6)   File "<stdin>", line 1, in <module>
3148: (4) ValueError: cannot set WRITEBACKIFCOPY flag to True
3149: (4) """
3150: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('sort',
3151: (4) """
3152: (4)     a.sort(axis=-1, kind=None, order=None)
3153: (4)     Sort an array in-place. Refer to `numpy.sort` for full documentation.
3154: (4) Parameters
3155: (4) -----
3156: (4)     axis : int, optional
3157: (8)         Axis along which to sort. Default is -1, which means sort along the
3158: (8)         last axis.
3159: (4)     kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
3160: (8)         Sorting algorithm. The default is 'quicksort'. Note that both 'stable'
3161: (8)         and 'mergesort' use timsort under the covers and, in general, the
3162: (8)         actual implementation will vary with datatype. The 'mergesort' option
3163: (8)         is retained for backwards compatibility.
3164: (8)         .. versionchanged:: 1.15.0
3165: (11)             The 'stable' option was added.
3166: (4)     order : str or list of str, optional
3167: (8)         When `a` is an array with fields defined, this argument specifies
3168: (8)         which fields to compare first, second, etc. A single field can
3169: (8)         be specified as a string, and not all fields need be specified,
3170: (8)         but unspecified fields will still be used, in the order in which

```

```

3171: (8)           they come up in the dtype, to break ties.
3172: (4)           See Also
3173: (4)
3174: (4)           -----
3175: (4)           numpy.sort : Return a sorted copy of an array.
3176: (4)           numpy.argsort : Indirect sort.
3177: (4)           numpy.lexsort : Indirect stable sort on multiple keys.
3178: (4)           numpy.searchsorted : Find elements in sorted array.
3179: (4)           numpy.partition: Partial sort.
3180: (4)           Notes
3181: (4)           -----
3182: (4)           See `numpy.sort` for notes on the different sorting algorithms.
3183: (4)           Examples
3184: (4)           -----
3185: (4)           >>> a = np.array([[1,4], [3,1]])
3186: (4)           >>> a.sort(axis=1)
3187: (4)           >>> a
3188: (11)          array([[1, 4],
3189: (4)           [1, 3]])
3190: (4)           >>> a.sort(axis=0)
3191: (4)           >>> a
3192: (11)          array([[1, 3],
3193: (4)           [1, 4]])
3194: (4)           Use the `order` keyword to specify a field to use when sorting a
3195: (4)           structured array:
3196: (4)           >>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
3197: (4)           >>> a.sort(order='y')
3198: (4)           >>> a
3199: (10)          array([(b'c', 1), (b'a', 2)],
3200: (4)           dtype=[('x', 'S1'), ('y', '<i8')])
3201: (0)           """
3202: (4)           add_newdoc('numpy.core.multiarray', 'ndarray', ('partition',
3203: (4)           """
3204: (4)           a.partition(kth, axis=-1, kind='introselect', order=None)
3205: (4)           Rearranges the elements in the array in such a way that the value of the
3206: (4)           element in kth position is in the position it would be in a sorted array.
3207: (4)           All elements smaller than the kth element are moved before this element
and
3208: (4)           all equal or greater are moved behind it. The ordering of the elements in
3209: (4)           the two partitions is undefined.
3210: (4)           .. versionadded:: 1.8.0
3211: (4)           Parameters
3212: (4)           -----
3213: (8)           kth : int or sequence of ints
3214: (8)           Element index to partition by. The kth element value will be in its
3215: (8)           final sorted position and all smaller elements will be moved before it
3216: (8)           and all equal or greater elements behind it.
3217: (8)           The order of all elements in the partitions is undefined.
3218: (8)           If provided with a sequence of kth it will partition all elements
3219: (8)           indexed by kth of them into their sorted position at once.
3220: (12)          .. deprecated:: 1.22.0
3221: (4)           Passing booleans as index is deprecated.
3222: (8)           axis : int, optional
3223: (8)           Axis along which to sort. Default is -1, which means sort along the
3224: (4)           last axis.
3225: (8)           kind : {'introselect'}, optional
3226: (4)           Selection algorithm. Default is 'introselect'.
3227: (8)           order : str or list of str, optional
3228: (8)           When `a` is an array with fields defined, this argument specifies
3229: (8)           which fields to compare first, second, etc. A single field can
3230: (8)           be specified as a string, and not all fields need to be specified,
3231: (8)           but unspecified fields will still be used, in the order in which
3232: (4)           they come up in the dtype, to break ties.
3233: (4)           See Also
3234: (4)           -----
3235: (4)           numpy.partition : Return a partitioned copy of an array.
3236: (4)           argpartition : Indirect partition.
3237: (4)           sort : Full sort.
3238: (4)           Notes

```

```

3239: (4)           See ``np.partition`` for notes on the different algorithms.
3240: (4)
3241: (4)
3242: (4)
3243: (4)
3244: (4)
3245: (4)
3246: (4)
3247: (4)
3248: (4)
3249: (4)
3250: (0)           add_newdoc('numpy.core.multiarray', 'ndarray', ('squeeze',
3251: (4)             """
3252: (4)               a.squeeze(axis=None)
3253: (4)               Remove axes of length one from `a`.
3254: (4)               Refer to `numpy.squeeze` for full documentation.
3255: (4)               See Also
3256: (4)
3257: (4)               -----
3258: (4)               numpy.squeeze : equivalent function
3259: (0)           """
3260: (4)
3261: (4)           where=True)
3262: (4)           Returns the standard deviation of the array elements along given axis.
3263: (4)           Refer to `numpy.std` for full documentation.
3264: (4)           See Also
3265: (4)
3266: (4)               -----
3267: (4)               numpy.std : equivalent function
3268: (0)           """
3269: (4)
3270: (4)           where=True)
3271: (4)           Return the sum of the array elements over the given axis.
3272: (4)           Refer to `numpy.sum` for full documentation.
3273: (4)           See Also
3274: (4)
3275: (4)               -----
3276: (4)               numpy.sum : equivalent function
3277: (0)           """
3278: (4)
3279: (4)           add_newdoc('numpy.core.multiarray', 'ndarray', ('swapaxes',
3280: (4)             """
3281: (4)               a.swapaxes(axis1, axis2)
3282: (4)               Return a view of the array with `axis1` and `axis2` interchanged.
3283: (4)               Refer to `numpy.swapaxes` for full documentation.
3284: (4)               See Also
3285: (4)
3286: (0)           """
3287: (4)
3288: (4)           add_newdoc('numpy.core.multiarray', 'ndarray', ('take',
3289: (4)             """
3290: (4)               a.take(indices, axis=None, out=None, mode='raise')
3291: (4)               Return an array formed from the elements of `a` at the given indices.
3292: (4)               Refer to `numpy.take` for full documentation.
3293: (4)               See Also
3294: (4)
3295: (0)           """
3296: (4)
3297: (4)           add_newdoc('numpy.core.multiarray', 'ndarray', ('tofile',
3298: (4)             """
3299: (4)               a.tofile(fid, sep="", format="%s")
3300: (4)               Write array to a file as text or binary (default).
3301: (4)               Data is always written in 'C' order, independent of the order of `a`.
3302: (4)               The data produced by this method can be recovered using the function
3303: (4)               fromfile().
3304: (4)               Parameters
3305: (8)                 fid : file or str or Path
3305: (8)                   An open file object, or a string containing a filename.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3306: (8)          .. versionchanged:: 1.17.0
3307: (12)         ``pathlib.Path`` objects are now accepted.
3308: (4)
3309: (8)
3310: (8)
3311: (8)
3312: (4)
3313: (8)
3314: (8)
3315: (8)
3316: (4)
3317: (4)
3318: (4) Notes
3319: (4)
3320: (4)
3321: (4)
3322: (4)
3323: (4)
3324: (4)
3325: (4)
3326: (4)
3327: (4)
3328: (4)
3329: (0) add_newdoc('numpy.core.multiarray', 'ndarray', ('tolist',
3330: (4)           """
3331: (4)           a.tolist()
3332: (4)           Return the array as an ``a.ndim``-levels deep nested list of Python
scalars.
3333: (4)           Return a copy of the array data as a (nested) Python list.
3334: (4)           Data items are converted to the nearest compatible builtin Python type,
via
3335: (4)           the ``~numpy.ndarray.item`` function.
3336: (4)           If ``a.ndim`` is 0, then since the depth of the nested list is 0, it will
3337: (4)           not be a list at all, but a simple Python scalar.
3338: (4)           Parameters
3339: (4)           -----
3340: (4)           none
3341: (4)           Returns
3342: (4)           -----
3343: (4)           y : object, or list of object, or list of list of object, or ...
3344: (8)           The possibly nested list of array elements.
3345: (4)           Notes
3346: (4)           -----
3347: (4)           The array may be recreated via ``a = np.array(a.tolist())``, although this
3348: (4)           may sometimes lose precision.
3349: (4)           Examples
3350: (4)           -----
3351: (4)           For a 1D array, ``a.tolist()`` is almost the same as ``list(a)``,
3352: (4)           except that ``tolist`` changes numpy scalars to Python scalars:
3353: (4)           >>> a = np.uint32([1, 2])
3354: (4)           >>> a_list = list(a)
3355: (4)           >>> a_list
3356: (4)           [1, 2]
3357: (4)           >>> type(a_list[0])
3358: (4)           <class 'numpy.uint32'>
3359: (4)           >>> a.tolist = a.tolist()
3360: (4)           >>> a.tolist
3361: (4)           [1, 2]
3362: (4)           >>> type(a.tolist[0])
3363: (4)           <class 'int'>
3364: (4)           Additionally, for a 2D array, ``tolist`` applies recursively:
3365: (4)           >>> a = np.array([[1, 2], [3, 4]])
3366: (4)           >>> list(a)
3367: (4)           [array([1, 2]), array([3, 4])]
3368: (4)
3369: (4)
3370: (4)
3371: (4)
3372: (4)

```

```

3373: (4)           Traceback (most recent call last):
3374: (6)             ...
3375: (4)             TypeError: iteration over a 0-d array
3376: (4)             >>> a.tolist()
3377: (4)             1
3378: (4)             "''")
3379: (0)             add_newdoc('numpy.core.multiarray', 'ndarray', ('tobytes', """
3380: (4)               a.tobytes(order='C')
3381: (4)               Construct Python bytes containing the raw data bytes in the array.
3382: (4)               Constructs Python bytes showing a copy of the raw contents of
3383: (4)               data memory. The bytes object is produced in C-order by default.
3384: (4)               This behavior is controlled by the ``order`` parameter.
3385: (4)               .. versionadded:: 1.9.0
3386: (4)             Parameters
3387: (4)             -----
3388: (4)             order : {'C', 'F', 'A'}, optional
3389: (8)               Controls the memory layout of the bytes object. 'C' means C-order,
3390: (8)               'F' means F-order, 'A' (short for *Any*) means 'F' if `a` is
3391: (8)               Fortran contiguous, 'C' otherwise. Default is 'C'.
3392: (4)             Returns
3393: (4)             -----
3394: (4)             s : bytes
3395: (8)               Python bytes exhibiting a copy of `a`'s raw data.
3396: (4)             See also
3397: (4)             -----
3398: (4)             frombuffer
3399: (8)               Inverse of this operation, construct a 1-dimensional array from Python
3400: (8)               bytes.
3401: (4)             Examples
3402: (4)             -----
3403: (4)             >>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
3404: (4)             >>> x.tobytes()
3405: (4)             b'\x00\x00\x01\x00\x02\x00\x03\x00'
3406: (4)             >>> x.tobytes('C') == x.tobytes()
3407: (4)             True
3408: (4)             >>> x.tobytes('F')
3409: (4)             b'\x00\x00\x02\x00\x01\x00\x03\x00'
3410: (4)             "''")
3411: (0)             add_newdoc('numpy.core.multiarray', 'ndarray', ('tostring', r"""
3412: (4)               a.tostring(order='C')
3413: (4)               A compatibility alias for `tobytes`, with exactly the same behavior.
3414: (4)               Despite its name, it returns `bytes` not `str`\ s.
3415: (4)               .. deprecated:: 1.19.0
3416: (4)               "''"))
3417: (0)             add_newdoc('numpy.core.multiarray', 'ndarray', ('trace',
3418: (4)               """
3419: (4)                 a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)
3420: (4)                 Return the sum along diagonals of the array.
3421: (4)                 Refer to `numpy.trace` for full documentation.
3422: (4)                 See Also
3423: (4)                 -----
3424: (4)                 numpy.trace : equivalent function
3425: (4)                 "''")
3426: (0)             add_newdoc('numpy.core.multiarray', 'ndarray', ('transpose',
3427: (4)               """
3428: (4)                 a.transpose(*axes)
3429: (4)                 Returns a view of the array with axes transposed.
3430: (4)                 Refer to `numpy.transpose` for full documentation.
3431: (4)                 Parameters
3432: (4)                 -----
3433: (4)                 axes : None, tuple of ints, or `n` ints
3434: (5)                   * None or no argument: reverses the order of the axes.
3435: (5)                   * tuple of ints: `i` in the `j`-th place in the tuple means that the
3436: (7)                     array's `i`-th axis becomes the transposed array's `j`-th axis.
3437: (5)                   * `n` ints: same as an n-tuple of the same ints (this form is
3438: (7)                     intended simply as a "convenience" alternative to the tuple form).
3439: (4)                 Returns
3440: (4)                 -----
3441: (4)                 p : ndarray

```

```

3442: (8)                                View of the array with its axes suitably permuted.
3443: (4)                                See Also
3444: (4)
3445: (4)                                transpose : Equivalent function.
3446: (4)                                ndarray.T : Array property returning the array transposed.
3447: (4)                                ndarray.reshape : Give a new shape to an array without changing its data.
3448: (4)                                Examples
3449: (4)
3450: (4)                                >>> a = np.array([[1, 2], [3, 4]])
3451: (4)                                >>> a
3452: (4)                                array([[1, 2],
3453: (11)                               [3, 4]])
3454: (4)                                >>> a.transpose()
3455: (4)                                array([[1, 3],
3456: (11)                               [2, 4]])
3457: (4)                                >>> a.transpose((1, 0))
3458: (4)                                array([[1, 3],
3459: (11)                               [2, 4]])
3460: (4)                                >>> a.transpose(1, 0)
3461: (4)                                array([[1, 3],
3462: (11)                               [2, 4]])
3463: (4)                                >>> a = np.array([1, 2, 3, 4])
3464: (4)                                >>> a
3465: (4)                                array([1, 2, 3, 4])
3466: (4)                                >>> a.transpose()
3467: (4)                                array([1, 2, 3, 4])
3468: (4)                                "''")
3469: (0)                                add_newdoc('numpy.core.multiarray', 'ndarray', ('var',
3470: (4)                                "''"
3471: (4)                                a.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *,
where=True)
3472: (4)                                Returns the variance of the array elements, along given axis.
3473: (4)                                Refer to `numpy.var` for full documentation.
3474: (4)                                See Also
3475: (4)
3476: (4)                                numpy.var : equivalent function
3477: (4)                                "'''")
3478: (0)                                add_newdoc('numpy.core.multiarray', 'ndarray', ('view',
3479: (4)                                "''"
3480: (4)                                a.view([dtype][, type]))
3481: (4)                                New view of array with the same data.
3482: (4)                                .. note::
3483: (8)                                Passing None for ``dtype`` is different from omitting the parameter,
3484: (8)                                since the former invokes ``dtype(None)`` which is an alias for
3485: (8)                                ``dtype('float_')``.
3486: (4)                                Parameters
3487: (4)
3488: (4)                                dtype : data-type or ndarray sub-class, optional
3489: (8)                                Data-type descriptor of the returned view, e.g., float32 or int16.
3490: (8)                                Omitting it results in the view having the same data-type as `a`.
3491: (8)                                This argument can also be specified as an ndarray sub-class, which
3492: (8)                                then specifies the type of the returned object (this is equivalent to
3493: (8)                                setting the ``type`` parameter).
3494: (4)                                type : Python type, optional
3495: (8)                                Type of the returned view, e.g., ndarray or matrix. Again, omission
3496: (8)                                of the parameter results in type preservation.
3497: (4)                                Notes
3498: (4)
3499: (4)                                ``a.view()`` is used two different ways:
3500: (4)                                ``a.view(some_dtype)`` or ``a.view(dtype=some_dtype)`` constructs a view
3501: (4)                                of the array's memory with a different data-type. This can cause a
3502: (4)                                reinterpretation of the bytes of memory.
3503: (4)                                ``a.view(ndarray_subclass)`` or ``a.view(type=ndarray_subclass)`` just
3504: (4)                                returns an instance of `ndarray_subclass` that looks at the same array
3505: (4)                                (same shape, dtype, etc.) This does not cause a reinterpretation of the
3506: (4)                                memory.
3507: (4)                                For ``a.view(some_dtype)``, if ``some_dtype`` has a different number of
3508: (4)                                bytes per entry than the previous dtype (for example, converting a regular
3509: (4)                                array to a structured array), then the last axis of ``a`` must be

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3510: (4) contiguous. This axis will be resized in the result.
3511: (4) .. versionchanged:: 1.23.0
3512: (7) Only the last axis needs to be contiguous. Previously, the entire array
3513: (7) had to be C-contiguous.
3514: (4) Examples
3515: (4) -----
3516: (4) >>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
3517: (4) Viewing array data using a different type and dtype:
3518: (4) >>> y = x.view(dtype=np.int16, type=np.matrix)
3519: (4) >>> y
3520: (4) matrix([[513]], dtype=int16)
3521: (4) >>> print(type(y))
3522: (4) <class 'numpy.matrix'>
3523: (4) Creating a view on a structured array so it can be used in calculations
3524: (4) >>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
3525: (4) >>> xv = x.view(dtype=np.int8).reshape(-1,2)
3526: (4) >>> xv
3527: (4) array([[1, 2],
3528: (11)     [3, 4]], dtype=int8)
3529: (4) >>> xv.mean(0)
3530: (4) array([2., 3.])
3531: (4) Making changes to the view changes the underlying array
3532: (4) >>> xv[0,1] = 20
3533: (4) >>> x
3534: (4) array([(1, 20), (3, 4)], dtype=[('a', 'i1'), ('b', 'i1')])
3535: (4) Using a view to convert an array to a recarray:
3536: (4) >>> z = x.view(np.recarray)
3537: (4) >>> z.a
3538: (4) array([1, 3], dtype=int8)
3539: (4) Views share data:
3540: (4) >>> x[0] = (9, 10)
3541: (4) >>> z[0]
3542: (4) (9, 10)
3543: (4) Views that change the dtype size (bytes per entry) should normally be
3544: (4) avoided on arrays defined by slices, transposes, fortran-ordering, etc.:
3545: (4) >>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
3546: (4) >>> y = x[:, ::2]
3547: (4) >>> y
3548: (4) array([[1, 3],
3549: (11)     [4, 6]], dtype=int16)
3550: (4) >>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
3551: (4) Traceback (most recent call last):
3552: (8) ...
3553: (4) ValueError: To change to a dtype of a different size, the last axis must
be contiguous
3554: (4) >>> z = y.copy()
3555: (4) >>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
3556: (4) array([[[(1, 3)],
3557: (11)     [(4, 6)]], dtype=[('width', '<i2'), ('length', '<i2')])
3558: (4) However, views that change dtype are totally fine for arrays with a
3559: (4) contiguous last axis, even if the rest of the axes are not C-contiguous:
3560: (4) >>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
3561: (4) >>> x.transpose(1, 0, 2).view(np.int16)
3562: (4) array([[[ 256,  770],
3563: (12)     [3340, 3854]],
3564: (4) <BLANKLINE>
3565: (11)     [[1284, 1798],
3566: (12)     [4368, 4882]],
3567: (4) <BLANKLINE>
3568: (11)     [[2312, 2826],
3569: (12)     [5396, 5910]]], dtype=int16)
3570: (4)     ""))
3571: (0) add_newdoc('numpy.core.umath', 'frompyfunc',
3572: (4)     """
3573: (4)         frompyfunc(func, /, nin, nout, *[, identity])
3574: (4)         Takes an arbitrary Python function and returns a NumPy ufunc.
3575: (4)         Can be used, for example, to add broadcasting to a built-in Python
3576: (4)         function (see Examples section).
3577: (4)         Parameters

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3578: (4)
3579: (4)
3580: (8)
3581: (4)
3582: (8)
3583: (4)
3584: (8)
3585: (4)
3586: (8)
resulting
3587: (8)
3588: (8)
3589: (8)
3590: (8)
the
3591: (8)
3592: (4)
3593: (4)
3594: (4)
3595: (8)
3596: (4)
3597: (4)
3598: (4)
numpy.
3599: (4)
3600: (4)
3601: (4)
3602: (4)
3603: (4)
3604: (4)
3605: (4)
3606: (4)
3607: (4)
3608: (4)
3609: (4)
3610: (4)
3611: (0)
3612: (4)
3613: (4)
3614: (4)
3615: (4)
3616: (4)
3617: (4)
3618: (4)
3619: (4)
3620: (4)
3621: (4)
3622: (8)
3623: (8)
3624: (8)
information
3625: (8)
3626: (8)
we
3627: (8)
3628: (8)
3629: (8)
3630: (8)
3631: (8)
3632: (8)
3633: (8)
3634: (8)
3635: (4)
3636: (4)
3637: (4)
3638: (4)
3639: (4)
3640: (4)
3641: (4)

-----  

func : Python function object  

An arbitrary Python function.  

nin : int  

The number of input arguments.  

nout : int  

The number of objects returned by `func`.  

identity : object, optional  

The value to use for the `~numpy.ufunc.identity` attribute of the  

object. If specified, this is equivalent to setting the underlying  

C ``identity`` field to ``PyUFunc_IdentityValue``.  

If omitted, the identity is set to ``PyUFunc_None``. Note that this is  

_not_ equivalent to setting the identity to ``None``, which implies  

operation is reorderable.  

Returns  

-----  

out : ufunc  

Returns a NumPy universal function (``ufunc``) object.  

See Also  

-----  

vectorize : Evaluates pyfunc over input arrays using broadcasting rules of  

numpy.  

Notes  

-----  

The returned ufunc always returns PyObject arrays.  

Examples  

-----  

Use frompyfunc to add broadcasting to the Python function ``oct``:  

>>> oct_array = np.frompyfunc(oct, 1, 1)  

>>> oct_array(np.array((10, 30, 100)))  

array(['0o12', '0o36', '0o144'], dtype=object)  

>>> np.array((oct(10), oct(30), oct(100))) # for comparison  

array(['0o12', '0o36', '0o144'], dtype='<U5')  

""")  

add_newdoc('numpy.core.umath', 'geterrobj',  

        """  

geterrobj()  

Return the current object that defines floating-point error handling.  

The error object contains all information that defines the error handling  

behavior in NumPy. `geterrobj` is used internally by the other  

functions that get and set error handling behavior (`geterr`, `seterr`,  

`geterrcall`, `seterrcall`).  

Returns  

-----  

errobj : list  

The error object, a list containing three elements:  

[internal numpy buffer size, error mask, error callback function].  

The error mask is a single integer that holds the treatment  

on all four floating point errors. The information for each error type  

is contained in three bits of the integer. If we print it in base 8,  

we  

can see what treatment is set for "invalid", "under", "over", and  

"divide" (in that order). The printed string can be interpreted with  

* 0 : 'ignore'  

* 1 : 'warn'  

* 2 : 'raise'  

* 3 : 'call'  

* 4 : 'print'  

* 5 : 'log'  

See Also  

-----  

seterrobj, seterr, geterr, seterrcall, geterrcall  

getbufsize, setbufsize  

Notes  

-----  

For complete documentation of the types of floating-point exceptions and

```

```

3642: (4) treatment options, see `seterr`.
3643: (4) Examples
3644: (4) -----
3645: (4) >>> np.geterrorobj() # first get the defaults
3646: (4) [8192, 521, None]
3647: (4) >>> def err_handler(type, flag):
3648: (4) ...     print("Floating point error (%s), with flag %s" % (type, flag))
3649: (4) ...
3650: (4) >>> old_bufsize = np.setbufsize(20000)
3651: (4) >>> old_err = np.seterr(divide='raise')
3652: (4) >>> old_handler = np.seterrcall(err_handler)
3653: (4) >>> np.geterrorobj()
3654: (4) [8192, 521, <function err_handler at 0x91dcaac>]
3655: (4) >>> old_err = np.seterr(all='ignore')
3656: (4) >>> np.base_repr(np.geterrorobj()[1], 8)
3657: (4) '0'
3658: (4) >>> old_err = np.seterr(divide='warn', over='log', under='call',
3659: (4) ...                           invalid='print')
3660: (4) >>> np.base_repr(np.geterrorobj()[1], 8)
3661: (4) '4351'
3662: (4) "''")
3663: (0) add_newdoc('numpy.core.umath', 'seterrorobj',
3664: (4) """
3665: (4) seterrorobj(errobj, /)
3666: (4) Set the object that defines floating-point error handling.
3667: (4) The error object contains all information that defines the error handling
3668: (4) behavior in NumPy. `seterrorobj` is used internally by the other
3669: (4) functions that set error handling behavior (`seterr`, `seterrcall`).
3670: (4) Parameters
3671: (4) -----
3672: (4) errobj : list
3673: (8) The error object, a list containing three elements:
3674: (8) [internal numpy buffer size, error mask, error callback function].
3675: (8) The error mask is a single integer that holds the treatment
information
3676: (8) on all four floating point errors. The information for each error type
3677: (8) is contained in three bits of the integer. If we print it in base 8,
we
3678: (8) can see what treatment is set for "invalid", "under", "over", and
3679: (8) "divide" (in that order). The printed string can be interpreted with
3680: (8) * 0 : 'ignore'
3681: (8) * 1 : 'warn'
3682: (8) * 2 : 'raise'
3683: (8) * 3 : 'call'
3684: (8) * 4 : 'print'
3685: (8) * 5 : 'log'
3686: (4) See Also
3687: (4) -----
3688: (4) geterrorobj, seterr, geterr, seterrcall, geterrcall
3689: (4) getbufsize, setbufsize
3690: (4) Notes
3691: (4) -----
3692: (4) For complete documentation of the types of floating-point exceptions and
3693: (4) treatment options, see `seterr`.
3694: (4) Examples
3695: (4) -----
3696: (4) >>> old_errorobj = np.geterrorobj() # first get the defaults
3697: (4) >>> old_errorobj
3698: (4) [8192, 521, None]
3699: (4) >>> def err_handler(type, flag):
3700: (4) ...     print("Floating point error (%s), with flag %s" % (type, flag))
3701: (4) ...
3702: (4) >>> new_errorobj = [20000, 12, err_handler]
3703: (4) >>> np.seterrorobj(new_errorobj)
3704: (4) >>> np.base_repr(12, 8) # int for divide=4 ('print') and over=1 ('warn')
3705: (4) '14'
3706: (4) >>> np.geterr()
3707: (4) {'over': 'warn', 'divide': 'print', 'invalid': 'ignore', 'under':
'ignore'}

```

```

3708: (4)             >>> np.geterrcall() is err_handler
3709: (4)             True
3710: (4)             """
3711: (0)             add_newdoc('numpy.core.multiarray', 'add_docstring',
3712: (4)             """
3713: (4)                 add_docstring(obj, docstring)
3714: (4)                 Add a docstring to a built-in obj if possible.
3715: (4)                 If the obj already has a docstring raise a RuntimeError
3716: (4)                 If this routine does not know how to add a docstring to the object
3717: (4)                 raise a TypeError
3718: (4)             """
3719: (0)             add_newdoc('numpy.core.umath', '_add_newdoc_ufunc',
3720: (4)             """
3721: (4)                 add_ufunc_docstring(ufunc, new_docstring)
3722: (4)                 Replace the docstring for a ufunc with new_docstring.
3723: (4)                 This method will only work if the current docstring for
3724: (4)                 the ufunc is NULL. (At the C level, i.e. when ufunc->doc is NULL.)
3725: (4)                 Parameters
3726: (4)                 -----
3727: (4)                 ufunc : numpy.ufunc
3728: (8)                   A ufunc whose current doc is NULL.
3729: (4)                 new_docstring : string
3730: (8)                   The new docstring for the ufunc.
3731: (4)                 Notes
3732: (4)                 -----
3733: (4)                   This method allocates memory for new_docstring on
3734: (4)                   the heap. Technically this creates a memory leak, since this
3735: (4)                   memory will not be reclaimed until the end of the program
3736: (4)                   even if the ufunc itself is removed. However this will only
3737: (4)                   be a problem if the user is repeatedly creating ufuncs with
3738: (4)                   no documentation, adding documentation via add_newdoc_ufunc,
3739: (4)                   and then throwing away the ufunc.
3740: (4)             """
3741: (0)             add_newdoc('numpy.core.multiarray', 'get_handler_name',
3742: (4)             """
3743: (4)                 get_handler_name(a: ndarray) -> str, None
3744: (4)                 Return the name of the memory handler used by `a`. If not provided, return
3745: (4)                 the name of the memory handler that will be used to allocate data for the
3746: (4)                 next `ndarray` in this context. May return None if `a` does not own its
3747: (4)                 memory, in which case you can traverse ``a.base`` for a memory handler.
3748: (4)             """
3749: (0)             add_newdoc('numpy.core.multiarray', 'get_handler_version',
3750: (4)             """
3751: (4)                 get_handler_version(a: ndarray) -> int, None
3752: (4)                 Return the version of the memory handler used by `a`. If not provided,
3753: (4)                 return the version of the memory handler that will be used to allocate
3754: (4)                 for the next `ndarray` in this context. May return None if `a` does not
3755: (4)                 own its memory, in which case you can traverse ``a.base`` for a memory
3756: (4)                 handler.
3757: (0)             add_newdoc('numpy.core.multiarray', '_get_madvise_hugepage',
3758: (4)             """
3759: (4)                 _get_madvise_hugepage() -> bool
3760: (4)                 Get use of ``madvise (2)`` MADV_HUGEPAGE support when
3761: (4)                 allocating the array data. Returns the currently set value.
3762: (4)                 See `global_state` for more information.
3763: (4)             """
3764: (0)             add_newdoc('numpy.core.multiarray', '_set_madvise_hugepage',
3765: (4)             """
3766: (4)                 _set_madvise_hugepage(enabled: bool) -> bool
3767: (4)                 Set or unset use of ``madvise (2)`` MADV_HUGEPAGE support when
3768: (4)                 allocating the array data. Returns the previously set value.
3769: (4)                 See `global_state` for more information.
3770: (4)             """
3771: (0)             add_newdoc('numpy.core._multiarray_tests', 'format_float_OSpprintf_g',
3772: (4)             """
3773: (4)                 format_float_OSpprintf_g(val, precision)

```

```

3774: (4)          Print a floating point scalar using the system's printf function,
3775: (4)          equivalent to:
3776: (8)            printf("%.*g", precision, val);
3777: (4)          for half/float/double, or replacing 'g' by 'Lg' for longdouble. This
3778: (4)          method is designed to help cross-validate the format_float_* methods.
3779: (4)          Parameters
3780: (4)            -----
3781: (4)              val : python float or numpy floating scalar
3782: (8)                Value to format.
3783: (4)              precision : non-negative integer, optional
3784: (8)                Precision given to printf.
3785: (4)          Returns
3786: (4)            -----
3787: (4)              rep : string
3788: (8)                The string representation of the floating point value
3789: (4)          See Also
3790: (4)            -----
3791: (4)              format_float_scientific
3792: (4)              format_float_positional
3793: (4)              """")
3794: (0)          add_newdoc('numpy.core', 'ufunc',
3795: (4)            """
3796: (4)          Functions that operate element by element on whole arrays.
3797: (4)          To see the documentation for a specific ufunc, use `info`. For
3798: (4)          example, ``np.info(np.sin)``. Because ufuncs are written in C
3799: (4)          (for speed) and linked into Python with NumPy's ufunc facility,
3800: (4)          Python's help() function finds this page whenever help() is called
3801: (4)          on a ufunc.
3802: (4)          A detailed explanation of ufuncs can be found in the docs for
:ref:`ufuncs`.
3803: (4)          **Calling ufuncs:** ``op(*x[, out], where=True, **kwargs)``
3804: (4)          Apply `op` to the arguments ``*x`` elementwise, broadcasting the arguments.
3805: (4)          The broadcasting rules are:
3806: (4)            * Dimensions of length 1 may be prepended to either array.
3807: (4)            * Arrays may be repeated along dimensions of length 1.
3808: (4)          Parameters
3809: (4)            -----
3810: (4)              *x : array_like
3811: (8)                Input arrays.
3812: (4)              out : ndarray, None, or tuple of ndarray and None, optional
3813: (8)                Alternate array object(s) in which to put the result; if provided, it
3814: (8)                must have a shape that the inputs broadcast to. A tuple of arrays
3815: (8)                (possibly only as a keyword argument) must have length equal to the
3816: (8)                number of outputs; use None for uninitialized outputs to be
3817: (8)                allocated by the ufunc.
3818: (4)              where : array_like, optional
3819: (8)                This condition is broadcast over the input. At locations where the
3820: (8)                condition is True, the `out` array will be set to the ufunc result.
3821: (8)                Elsewhere, the `out` array will retain its original value.
3822: (8)                Note that if an uninitialized `out` array is created via the default
3823: (8)                ``out=None``, locations within it where the condition is False will
3824: (8)                remain uninitialized.
3825: (4)              **kwargs
3826: (8)                For other keyword-only arguments, see the :ref:`ufunc docs
<ufuncs.kwargs>`.
```

3827: (4) Returns

3828: (4) -----

3829: (4) r : ndarray or tuple of ndarray

3830: (8) `r` will have the shape that the arrays in `x` broadcast to; if `out`

is

3831: (8) provided, it will be returned. If not, `r` will be allocated and

3832: (8) may contain uninitialized values. If the function has more than one

3833: (8) output, then the result will be a tuple of arrays.

3834: (4) """")

3835: (0) add\_newdoc('numpy.core', 'ufunc', ('identity',
3836: (4) """
3837: (4) The identity value.
3838: (4) Data attribute containing the identity element for the ufunc, if it has
one.

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

3839: (4) If it does not, the attribute value is None.  
3840: (4) Examples  
3841: (4) -----  
3842: (4) >>> np.add.identity  
3843: (4) 0  
3844: (4) >>> np.multiply.identity  
3845: (4) 1  
3846: (4) >>> np.power.identity  
3847: (4) 1  
3848: (4) >>> print(np.exp.identity)  
3849: (4) None  
3850: (4) """))  
3851: (0) add\_newdoc('numpy.core', 'ufunc', ('nargs',  
3852: (4) """  
3853: (4) The number of arguments.  
3854: (4) Data attribute containing the number of arguments the ufunc takes,  
including  
3855: (4) optional ones.  
3856: (4) Notes  
3857: (4) -----  
3858: (4) Typically this value will be one more than what you might expect because  
all  
3859: (4) ufuncs take the optional "out" argument.  
3860: (4) Examples  
3861: (4) -----  
3862: (4) >>> np.add.nargs  
3863: (4) 3  
3864: (4) >>> np.multiply.nargs  
3865: (4) 3  
3866: (4) >>> np.power.nargs  
3867: (4) 3  
3868: (4) >>> np.exp.nargs  
3869: (4) 2  
3870: (4) """))  
3871: (0) add\_newdoc('numpy.core', 'ufunc', ('nin',  
3872: (4) """  
3873: (4) The number of inputs.  
3874: (4) Data attribute containing the number of arguments the ufunc treats as  
input.  
3875: (4) Examples  
3876: (4) -----  
3877: (4) >>> np.add.nin  
3878: (4) 2  
3879: (4) >>> np.multiply.nin  
3880: (4) 2  
3881: (4) >>> np.power.nin  
3882: (4) 2  
3883: (4) >>> np.exp.nin  
3884: (4) 1  
3885: (4) """))  
3886: (0) add\_newdoc('numpy.core', 'ufunc', ('nout',  
3887: (4) """  
3888: (4) The number of outputs.  
3889: (4) Data attribute containing the number of arguments the ufunc treats as  
output.  
3890: (4) Notes  
3891: (4) -----  
3892: (4) Since all ufuncs can take output arguments, this will always be (at least)  
1.  
3893: (4) Examples  
3894: (4) -----  
3895: (4) >>> np.add.nout  
3896: (4) 1  
3897: (4) >>> np.multiply.nout  
3898: (4) 1  
3899: (4) >>> np.power.nout  
3900: (4) 1  
3901: (4) >>> np.exp.nout  
3902: (4) 1

```

3903: (4)      """"))
3904: (0)      add_newdoc('numpy.core', 'ufunc', ('ntypes',
3905: (4)          """
3906: (4)          The number of types.
3907: (4)          The number of numerical NumPy types - of which there are 18 total - on
which
3908: (4)          the ufunc can operate.
3909: (4)          See Also
3910: (4)          -----
3911: (4)          numpy.ufunc.ntypes
3912: (4)          Examples
3913: (4)          -----
3914: (4)          >>> np.add.ntypes
3915: (4)          18
3916: (4)          >>> np.multiply.ntypes
3917: (4)          18
3918: (4)          >>> np.power.ntypes
3919: (4)          17
3920: (4)          >>> np.exp.ntypes
3921: (4)          7
3922: (4)          >>> np.remainder.ntypes
3923: (4)          14
3924: (4)          """"))
3925: (0)      add_newdoc('numpy.core', 'ufunc', ('types',
3926: (4)          """
3927: (4)          Returns a list with types grouped input->output.
3928: (4)          Data attribute listing the data-type "Domain-Range" groupings the ufunc
can
3929: (4)          deliver. The data-types are given using the character codes.
3930: (4)          See Also
3931: (4)          -----
3932: (4)          numpy.ufunc.ntypes
3933: (4)          Examples
3934: (4)          -----
3935: (4)          >>> np.add.types
3936: (4)          ['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
3937: (4)          'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
3938: (4)          'GG->G', 'OO->O']
3939: (4)          >>> np.multiply.types
3940: (4)          ['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
3941: (4)          'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
3942: (4)          'GG->G', 'OO->O']
3943: (4)          >>> np.power.types
3944: (4)          ['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
3945: (4)          'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D', 'GG->G',
3946: (4)          'OO->O']
3947: (4)          >>> np.exp.types
3948: (4)          ['f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']
3949: (4)          >>> np.remainder.types
3950: (4)          ['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
3951: (4)          'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'OO->O']
3952: (4)          """"))
3953: (0)      add_newdoc('numpy.core', 'ufunc', ('signature',
3954: (4)          """
3955: (4)          Definition of the core elements a generalized ufunc operates on.
3956: (4)          The signature determines how the dimensions of each input/output array
are split into core and loop dimensions:
3957: (4)          1. Each dimension in the signature is matched to a dimension of the
corresponding passed-in array, starting from the end of the shape
tuple.
3958: (4)
3959: (7)          tuple.
3960: (4)          2. Core dimensions assigned to the same label in the signature must have
exactly matching sizes, no broadcasting is performed.
3961: (7)          3. The core dimensions are removed from all inputs and the remaining
dimensions are broadcast together, defining the loop dimensions.
3962: (4)          Notes
3963: (4)          -----
3964: (4)          Generalized ufuncs are used internally in many linalg functions, and in
the testing suite; the examples below are taken from these.
3965: (4)          For ufuncs that operate on scalars, the signature is None, which is

```

```

3969: (4) equivalent to '()' for every argument.
3970: (4) Examples
3971: (4)
3972: (4) -----
3973: (4) >>> np.core.umath_tests.matrix_multiply.signature
3974: (4) '(m,n),(n,p)->(m,p)'
3975: (4) >>> np.linalg._umath_linalg.det.signature
3976: (4) '(m,m)->()'
3977: (4) >>> np.add.signature is None
3978: (4) True # equivalent to '(),()->()'
3979: (4) """
3980: (4) add_newdoc('numpy.core', 'ufunc', ('reduce',
3981: (4) """"
3982: (4) reduce(array, axis=0, dtype=None, out=None, keepdims=False, initial=<no
3983: (4) value>, where=True)
3984: (4) =
3985: (4) the result of iterating `j` over :math:`range(N_i)`, cumulatively applying
3986: (4) ufunc to each :math:`array[k_0, \dots, k_{i-1}, j, k_{i+1}, \dots, k_{M-1}]` .
3987: (4) For a one-dimensional array, reduce produces results equivalent to:
3988: (4) :::
3989: (5) r = op.identity # op = ufunc
3990: (5) for i in range(len(A)):
3991: (7)     r = op(r, A[i])
3992: (5) return r
3993: (4) For example, add.reduce() is equivalent to sum().
3994: (4) Parameters
3995: (4) -----
3996: (4) array : array_like
3997: (8)     The array to act on.
3998: (4) axis : None or int or tuple of ints, optional
3999: (8)     Axis or axes along which a reduction is performed.
4000: (8)     The default (`axis` = 0) is perform a reduction over the first
4001: (8)     dimension of the input array. `axis` may be negative, in
4002: (8)     which case it counts from the last to the first axis.
4003: (8) .. versionadded:: 1.7.0
4004: (8) If this is None, a reduction is performed over all the axes.
4005: (8) If this is a tuple of ints, a reduction is performed on multiple
4006: (8) axes, instead of a single axis or all the axes as before.
4007: (8) For operations which are either not commutative or not associative,
4008: (8) doing a reduction over multiple axes is not well-defined. The
4009: (8) ufuncs do not currently raise an exception in this case, but will
4010: (8) likely do so in the future.
4011: (4) dtype : data-type code, optional
4012: (8)     The type used to represent the intermediate results. Defaults
4013: (8)     to the data-type of the output array if this is provided, or
4014: (8)     the data-type of the input array if no output array is provided.
4015: (4) out : ndarray, None, or tuple of ndarray and None, optional
4016: (8)     A location into which the result is stored. If not provided or None,
4017: (8)     a freshly-allocated array is returned. For consistency with
4018: (8)     ``ufunc.__call__``, if given as a keyword, this may be wrapped in a
4019: (8)     1-element tuple.
4020: (8) .. versionchanged:: 1.13.0
4021: (11)     Tuples are allowed for keyword argument.
4022: (4) keepdims : bool, optional
4023: (8)     If this is set to True, the axes which are reduced are left
4024: (8)     in the result as dimensions with size one. With this option,
4025: (8)     the result will broadcast correctly against the original `array` .
4026: (8) .. versionadded:: 1.7.0
4027: (4) initial : scalar, optional
4028: (8)     The value with which to start the reduction.
4029: (8)     If the ufunc has no identity or the dtype is object, this defaults
4030: (8)     to None - otherwise it defaults to ufunc.identity.
4031: (8)     If ``None`` is given, the first element of the reduction is used,
4032: (8)     and an error is thrown if the reduction is empty.
4033: (8) .. versionadded:: 1.15.0
4034: (4) where : array_like of bool, optional
4035: (8)     A boolean array which is broadcasted to match the dimensions

```

```

4036: (8)          of `array`, and selects elements to include in the reduction. Note
4037: (8)          that for ufuncs like ``minimum`` that do not have an identity
4038: (8)          defined, one has to pass in also ``initial``.
4039: (8)          .. versionadded:: 1.17.0
4040: (4)          Returns
4041: (4)          -----
4042: (4)          r : ndarray
4043: (8)          The reduced array. If `out` was supplied, `r` is a reference to it.
4044: (4)          Examples
4045: (4)          -----
4046: (4)          >>> np.multiply.reduce([2,3,5])
4047: (4)          30
4048: (4)          A multi-dimensional array example:
4049: (4)          >>> X = np.arange(8).reshape((2,2,2))
4050: (4)          >>> X
4051: (4)          array([[[0, 1],
4052: (12)             [2, 3]],
4053: (11)               [[4, 5],
4054: (12)                 [6, 7]]])
4055: (4)          >>> np.add.reduce(X, 0)
4056: (4)          array([[ 4,  6],
4057: (11)             [ 8, 10]])
4058: (4)          >>> np.add.reduce(X) # confirm: default axis value is 0
4059: (4)          array([[ 4,  6],
4060: (11)             [ 8, 10]])
4061: (4)          >>> np.add.reduce(X, 1)
4062: (4)          array([[ 2,  4],
4063: (11)             [10, 12]])
4064: (4)          >>> np.add.reduce(X, 2)
4065: (4)          array([[ 1,  5],
4066: (11)             [ 9, 13]])
4067: (4)          You can use the ``initial`` keyword argument to initialize the reduction
4068: (4)          with a different value, and ``where`` to select specific elements to
4069: (4)          include:
4070: (4)          >>> np.add.reduce([10], initial=5)
4071: (4)          15
4072: (4)          >>> np.add.reduce(np.ones((2, 2, 2)), axis=(0, 2), initial=10)
4073: (4)          array([14., 14.])
4074: (4)          >>> a = np.array([10., np.nan, 10])
4075: (4)          >>> np.add.reduce(a, where=~np.isnan(a))
4076: (4)          20.0
4077: (4)          Allows reductions of empty arrays where they would normally fail, i.e.
4078: (4)          for ufuncs without an identity.
4079: (4)          >>> np.minimum.reduce([], initial=np.inf)
4080: (4)          inf
4081: (4)          >>> np.minimum.reduce([[1., 2.], [3., 4.]], initial=10., where=[True,
4082: (4)             False])
4083: (4)          array([ 1., 10.])
4084: (8)          ...
4085: (4)          ValueError: zero-size array to reduction operation minimum which has no
4086: (4)          identity
4087: (0)          """
4088: (4)          add_newdoc('numpy.core', 'ufunc', ('accumulate',
4089: (4)             """
4090: (4)             accumulate(array, axis=0, dtype=None, out=None)
4091: (4)             Accumulate the result of applying the operator to all elements.
4092: (6)             For a one-dimensional array, accumulate produces results equivalent to::
4093: (6)             r = np.empty(len(A))
4094: (6)             t = op.identity      # op = the ufunc being applied to A's elements
4095: (10)            for i in range(len(A)):
4096: (10)                t = op(t, A[i])
4097: (6)                r[i] = t
4098: (4)            return r
4099: (4)            For example, add.accumulate() is equivalent to np.cumsum().
4100: (4)            For a multi-dimensional array, accumulate is applied along only one
4101: (4)            axis (axis zero by default; see Examples below) so repeated use is

```

```

4102: (4)          Parameters
4103: (4)          -----
4104: (4)          array : array_like
4105: (8)          The array to act on.
4106: (4)          axis : int, optional
4107: (8)          The axis along which to apply the accumulation; default is zero.
4108: (4)          dtype : data-type code, optional
4109: (8)          The data-type used to represent the intermediate results. Defaults
4110: (8)          to the data-type of the output array if such is provided, or the
4111: (8)          data-type of the input array if no output array is provided.
4112: (4)          out : ndarray, None, or tuple of ndarray and None, optional
4113: (8)          A location into which the result is stored. If not provided or None,
4114: (8)          a freshly-allocated array is returned. For consistency with
4115: (8)          ``ufunc.__call__``, if given as a keyword, this may be wrapped in a
4116: (8)          1-element tuple.
4117: (8)          .. versionchanged:: 1.13.0
4118: (11)          Tuples are allowed for keyword argument.
4119: (4)          Returns
4120: (4)          -----
4121: (4)          r : ndarray
4122: (8)          The accumulated values. If `out` was supplied, `r` is a reference to
4123: (8)          `out`.
4124: (4)          Examples
4125: (4)          -----
4126: (4)          1-D array examples:
4127: (4)          >>> np.add.accumulate([2, 3, 5])
4128: (4)          array([ 2,  5, 10])
4129: (4)          >>> np.multiply.accumulate([2, 3, 5])
4130: (4)          array([ 2,  6, 30])
4131: (4)          2-D array examples:
4132: (4)          >>> I = np.eye(2)
4133: (4)          >>> I
4134: (4)          array([[1.,  0.],
4135: (11)          [0.,  1.]])
4136: (4)          Accumulate along axis 0 (rows), down columns:
4137: (4)          >>> np.add.accumulate(I, 0)
4138: (4)          array([[1.,  0.],
4139: (11)          [1.,  1.]])
4140: (4)          >>> np.add.accumulate(I) # no axis specified = axis zero
4141: (4)          array([[1.,  0.],
4142: (11)          [1.,  1.]])
4143: (4)          Accumulate along axis 1 (columns), through rows:
4144: (4)          >>> np.add.accumulate(I, 1)
4145: (4)          array([[1.,  1.],
4146: (11)          [0.,  1.]])
4147: (4)          """))
4148: (0)          add_newdoc('numpy.core', 'ufunc', ('reduceat',
4149: (4)          """
4150: (4)          reduceat(array, indices, axis=0, dtype=None, out=None)
4151: (4)          Performs a (local) reduce with specified slices over a single axis.
4152: (4)          For i in ``range(len(indices))``, `reduceat` computes
4153: (4)          ``ufunc.reduce(array[indices[i]:indices[i+1]])``, which becomes the i-th
4154: (4)          generalized "row" parallel to `axis` in the final result (i.e., in a
4155: (4)          2-D array, for example, if `axis = 0`, it becomes the i-th row, but if
4156: (4)          `axis = 1`, it becomes the i-th column). There are three exceptions to
4157: (4)          this:
4158: (6)          * when ``i = len(indices) - 1`` (so for the last index),
4159: (4)          ``indices[i+1] = array.shape[axis]``.
4160: (6)          * if ``indices[i] >= indices[i + 1]``, the i-th generalized "row" is
4161: (4)          simply ``array[indices[i]]``.
4162: (4)          * if ``indices[i] >= len(array)`` or ``indices[i] < 0``, an error is
4163: (4)          raised.
4164: (4)          The shape of the output depends on the size of `indices`, and may be
4165: (4)          larger than `array` (this happens if ``len(indices) >
4166: (4)          array.shape[axis]``).
4167: (8)          Parameters
4168: (4)          -----
4169: (4)          array : array_like
4170: (4)          The array to act on.

```

```

4168: (4)           indices : array_like
4169: (8)             Paired indices, comma separated (not colon), specifying slices to
4170: (8)             reduce.
4171: (4)           axis : int, optional
4172: (8)             The axis along which to apply the reduceat.
4173: (4)           dtype : data-type code, optional
4174: (8)             The type used to represent the intermediate results. Defaults
4175: (8)             to the data type of the output array if this is provided, or
4176: (8)             the data type of the input array if no output array is provided.
4177: (4)           out : ndarray, None, or tuple of ndarray and None, optional
4178: (8)             A location into which the result is stored. If not provided or None,
4179: (8)             a freshly-allocated array is returned. For consistency with
4180: (8)             ``ufunc.__call__``, if given as a keyword, this may be wrapped in a
4181: (8)             1-element tuple.
4182: (8)             .. versionchanged:: 1.13.0
4183: (11)            Tuples are allowed for keyword argument.
4184: (4)           Returns
4185: (4)           -----
4186: (4)           r : ndarray
4187: (8)             The reduced values. If `out` was supplied, `r` is a reference to
4188: (8)             `out`.
4189: (4)           Notes
4190: (4)           -----
4191: (4)             A descriptive example:
4192: (4)             If `array` is 1-D, the function `ufunc.accumulate(array)` is the same as
4193: (4)             ``ufunc.reduceat(array, indices)[::2]`` where `indices` is
4194: (4)             ``range(len(array) - 1)`` with a zero placed
4195: (4)             in every other element:
4196: (4)             ``indices = zeros(2 * len(array) - 1)``,
4197: (4)             ``indices[1::2] = range(1, len(array))``.
4198: (4)             Don't be fooled by this attribute's name: `reduceat(array)` is not
4199: (4)             necessarily smaller than `array`.
4200: (4)           Examples
4201: (4)           -----
4202: (4)             To take the running sum of four successive values:
4203: (4)             >>> np.add.reduceat(np.arange(8), [0,4, 1,5, 2,6, 3,7])[::2]
4204: (4)             array([ 6, 10, 14, 18])
4205: (4)             A 2-D example:
4206: (4)             >>> x = np.linspace(0, 15, 16).reshape(4,4)
4207: (4)             >>> x
4208: (4)             array([[ 0.,   1.,   2.,   3.],
4209: (11)               [ 4.,   5.,   6.,   7.],
4210: (11)               [ 8.,   9.,  10.,  11.],
4211: (11)               [12.,  13.,  14.,  15.]])
4212: (4)             :::
4213: (4)             >>> np.add.reduceat(x, [0, 3, 1, 2, 0])
4214: (4)             array([[12.,  15.,  18.,  21.],
4215: (11)               [12.,  13.,  14.,  15.],
4216: (11)               [ 4.,   5.,   6.,   7.],
4217: (11)               [ 8.,   9.,  10.,  11.],
4218: (11)               [24.,  28.,  32.,  36.]])
4219: (4)             :::
4220: (4)             >>> np.multiply.reduceat(x, [0, 3], 1)
4221: (4)             array([[ 0.,    3.],
4222: (11)               [120.,    7.],
4223: (11)               [720.,   11.],
4224: (11)               [2184.,  15.]])
4225: (4)             """))
4226: (0)           add_newdoc('numpy.core', 'ufunc', ('outer',
4227: (4)             """
4228: (4)             outer(A, B, /, **kwargs)
4229: (4)             Apply the ufunc `op` to all pairs (a, b) with a in `A` and b in `B`.
4230: (4)             Let ``M = A.ndim``, ``N = B.ndim``. Then the result, `C`, of
4231: (4)             ``op.outer(A, B)`` is an array of dimension M + N such that:
4232: (4)             .. math:: C[i_0, \dots, i_{M-1}, j_0, \dots, j_{N-1}] =
4233: (7)               op(A[i_0, \dots, i_{M-1}], B[j_0, \dots, j_{N-1}])
4234: (4)             For `A` and `B` one-dimensional, this is equivalent to:
4235: (6)               r = empty(len(A),len(B))
4236: (6)               for i in range(len(A)):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4237: (10)
4238: (14)
4239: (4)
4240: (4)
4241: (4)
4242: (8)
4243: (4)
4244: (8)
4245: (4)
4246: (8)
4247: (8)
4248: (4)
4249: (4)
4250: (4)
4251: (8)
4252: (4)
4253: (4)
4254: (4)
4255: (18)
4256: (18)
4257: (4)
4258: (16)
4259: (16)
4260: (4)
4261: (4)
4262: (4)
4263: (4)
4264: (11)
4265: (11)
4266: (4)
4267: (4)
4268: (4)
4269: (4)
4270: (4)
4271: (4)
4272: (4)
4273: (4)
4274: (4)
4275: (4)
4276: (4)
4277: (12)
4278: (12)
4279: (11)
4280: (12)
4281: (12)
4282: (4)
4283: (0)
4284: (4)
4285: (4)
4286: (4)
4287: (4)
4288: (4)
4289: (4)
4290: (4)
4291: (4)
4292: (4)
4293: (4)
4294: (4)
4295: (4)
4296: (8)
4297: (4)
4298: (8)
4299: (8)
4300: (8)
4301: (4)
4302: (8)
4303: (8)
4304: (4)
4305: (4)

        for j in range(len(B)):
            r[i,j] = op(A[i], B[j]) # op = ufunc in question

    Parameters
    -----
    A : array_like
        First array
    B : array_like
        Second array
    kwargs : any
        Arguments to pass on to the ufunc. Typically `dtype` or `out`.
        See `ufunc` for a comprehensive overview of all available arguments.

    Returns
    -----
    r : ndarray
        Output array

    See Also
    -----
    numpy.outer : A less powerful version of ``np.multiply.outer`` that `ravel`\ s all inputs to 1D. This exists primarily for compatibility with old code.
    tensordot : ``np.tensordot(a, b, axes=((), ()))`` and ``np.multiply.outer(a, b)`` behave same for all dimensions of a and b.

    Examples
    -----
    >>> np.multiply.outer([1, 2, 3], [4, 5, 6])
    array([[ 4,  5,  6],
           [ 8, 10, 12],
           [12, 15, 18]])

    A multi-dimensional example:
    >>> A = np.array([[1, 2, 3], [4, 5, 6]])
    >>> A.shape
    (2, 3)
    >>> B = np.array([[1, 2, 3, 4]])
    >>> B.shape
    (1, 4)
    >>> C = np.multiply.outer(A, B)
    >>> C.shape; C
    (2, 3, 1, 4)
    array([[[[ 1,  2,  3,  4]],
           [[ 2,  4,  6,  8]],
           [[ 3,  6,  9, 12]]],
           [[[ 4,  8, 12, 16]],
           [[ 5, 10, 15, 20]],
           [[ 6, 12, 18, 24]]])

    add_newdoc('numpy.core', 'ufunc', ('at',
    """
        at(a, indices, b=None, /)
        Performs unbuffered in place operation on operand 'a' for elements specified by 'indices'. For addition ufunc, this method is equivalent to ``a[indices] += b``, except that results are accumulated for elements that are indexed more than once. For example, ``a[[0,0]] += 1`` will only increment the first element once because of buffering, whereas ``add.at(a, [0,0], 1)`` will increment the first element twice.
        .. versionadded:: 1.8.0
    Parameters
    -----
    a : array_like
        The array to perform in place operation on.
    indices : array_like or tuple
        Array like index object or slice object for indexing into first operand. If first operand has multiple dimensions, indices can be a tuple of array like index objects or slice objects.
    b : array_like
        Second operand for ufuncs requiring two operands. Operand must be broadcastable over first operand after indexing or slicing.

    Examples
    -----

```

```

4306: (4) Set items 0 and 1 to their negative values:
4307: (4) >>> a = np.array([1, 2, 3, 4])
4308: (4) >>> np.negative.at(a, [0, 1])
4309: (4) >>> a
4310: (4) array([-1, -2,  3,  4])
4311: (4) Increment items 0 and 1, and increment item 2 twice:
4312: (4) >>> a = np.array([1, 2, 3, 4])
4313: (4) >>> np.add.at(a, [0, 1, 2, 2], 1)
4314: (4) >>> a
4315: (4) array([2, 3, 5, 4])
4316: (4) Add items 0 and 1 in first array to second array,
4317: (4) and store results in first array:
4318: (4) >>> a = np.array([1, 2, 3, 4])
4319: (4) >>> b = np.array([1, 2])
4320: (4) >>> np.add.at(a, [0, 1], b)
4321: (4) >>> a
4322: (4) array([2, 4, 3, 4])
4323: (4) """
4324: (0) add_newdoc('numpy.core', 'ufunc', ('resolve_dtypes',
4325: (4) """
4326: (4) resolve_dtypes(dtypes, *, signature=None, casting=None, reduction=False)
4327: (4) Find the dtypes NumPy will use for the operation. Both input and
4328: (4) output dtypes are returned and may differ from those provided.
4329: (4) .. note::
4330: (8) This function always applies NEP 50 rules since it is not provided
4331: (8) any actual values. The Python types ``int``, ``float``, and
4332: (8) ``complex`` thus behave weak and should be passed for "untyped"
4333: (8) Python input.
4334: (4) Parameters
4335: (4) -----
4336: (4) dtypes : tuple of dtypes, None, or literal int, float, complex
4337: (8) The input dtypes for each operand. Output operands can be
4338: (8) None, indicating that the dtype must be found.
4339: (4) signature : tuple of DTypes or None, optional
4340: (8) If given, enforces exact Dtype (classes) of the specific operand.
4341: (8) The ufunc ``dtype`` argument is equivalent to passing a tuple with
4342: (8) only output dtypes set.
4343: (4) casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
4344: (8) The casting mode when casting is necessary. This is identical to
4345: (8) the ufunc call casting modes.
4346: (4) reduction : boolean
4347: (8) If given, the resolution assumes a reduce operation is happening
4348: (8) which slightly changes the promotion and type resolution rules.
4349: (8) `dtypes` is usually something like ``(``None, np.dtype("i2"), None)``
4350: (8) for reductions (first input is also the output).
4351: (8) .. note::
4352: (12) The default casting mode is "same_kind", however, as of
4353: (12) NumPy 1.24, NumPy uses "unsafe" for reductions.
4354: (4) Returns
4355: (4) -----
4356: (4) dtypes : tuple of dtypes
4357: (8) The dtypes which NumPy would use for the calculation. Note that
4358: (8) dtypes may not match the passed in ones (casting is necessary).
4359: (4) See Also
4360: (4) -----
4361: (4) numpy.ufunc._resolve_dtypes_and_context :
4362: (8) Similar function to this, but returns additional information which
4363: (8) give access to the core C functionality of NumPy.
4364: (4) Examples
4365: (4) -----
4366: (4) This API requires passing dtypes, define them for convenience:
4367: (4) >>> int32 = np.dtype("int32")
4368: (4) >>> float32 = np.dtype("float32")
4369: (4) The typical ufunc call does not pass an output dtype. `np.add` has two
4370: (4) inputs and one output, so leave the output as ``None`` (not provided):
4371: (4) >>> np.add.resolve_dtypes((int32, float32, None))
4372: (4) (dtype('float64'), dtype('float64'), dtype('float64'))
4373: (4) The loop found uses "float64" for all operands (including the output), the
4374: (4) first input would be cast.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4375: (4) ``resolve_dtypes`` supports "weak" handling for Python scalars by passing
4376: (4) ``int``, ``float``, or ``complex``:
4377: (4) >>> np.add.resolve_dtypes((float32, float, None))
4378: (4) (dtype('float32'), dtype('float32'), dtype('float32'))
4379: (4) Where the Python ``float`` behaves similar to a Python value ``0.0``
4380: (4) in a ufunc call. (See :ref:`NEP 50 <NEP50>` for details.)
4381: (4) """
4382: (0) add_newdoc('numpy.core', 'ufunc', ('_resolve_dtypes_and_context',
4383: (4) """
4384: (4) _resolve_dtypes_and_context(dtypes, *, signature=None, casting=None,
reduction=False)
4385: (4) See `numpy.ufunc.resolve_dtypes` for parameter information. This
4386: (4) function is considered *unstable*. You may use it, but the returned
4387: (4) information is NumPy version specific and expected to change.
4388: (4) Large API/ABI changes are not expected, but a new NumPy version is
4389: (4) expected to require updating code using this functionality.
4390: (4) This function is designed to be used in conjunction with
4391: (4) `numpy.ufunc._get_strided_loop`. The calls are split to mirror the C API
4392: (4) and allow future improvements.
4393: (4) Returns
4394: (4) -----
4395: (4) dtypes : tuple of dtypes
4396: (4) call_info :
4397: (8) PyCapsule with all necessary information to get access to low level
4398: (8) C calls. See `numpy.ufunc._get_strided_loop` for more information.
4399: (4) """
4400: (0) add_newdoc('numpy.core', 'ufunc', ('_get_strided_loop',
4401: (4) """
4402: (4) _get_strided_loop(call_info, /, *, fixed_strides=None)
4403: (4) This function fills in the ``call_info`` capsule to include all
4404: (4) information necessary to call the low-level strided loop from NumPy.
4405: (4) See notes for more information.
4406: (4) Parameters
4407: (4) -----
4408: (4) call_info : PyCapsule
4409: (8) The PyCapsule returned by `numpy.ufunc._resolve_dtypes_and_context`.
4410: (4) fixed_strides : tuple of int or None, optional
4411: (8) A tuple with fixed byte strides of all input arrays. NumPy may use
4412: (8) this information to find specialized loops, so any call must follow
4413: (8) the given stride. Use ``None`` to indicate that the stride is not
4414: (8) known (or not fixed) for all calls.
4415: (4) Notes
4416: (4) -----
4417: (4) Together with `numpy.ufunc._resolve_dtypes_and_context` this function
4418: (4) gives low-level access to the NumPy ufunc loops.
4419: (4) The first function does general preparation and returns the required
4420: (4) information. It returns this as a C capsule with the version specific
4421: (4) name ``numpy_1.24_ufunc_call_info``.
4422: (4) The NumPy 1.24 ufunc call info capsule has the following layout::
4423: (8) typedef struct {
4424: (12)     PyArrayMethod_StridedLoop *strided_loop;
4425: (12)     PyArrayMethod_Context *context;
4426: (12)     NpyAuxData *auxdata;
4427: (12)     /* Flag information (expected to change) */
4428: (12)     npy_bool requires_pyapi; /* GIL is required by loop */
4429: (12)     /* Loop doesn't set FPE flags; if not set check FPE flags */
4430: (12)     npy_bool no_floatingpoint_errors;
4431: (8) } ufunc_call_info;
4432: (4) Note that the first call only fills in the ``context``. The call to
4433: (4) ``_get_strided_loop`` fills in all other data.
4434: (4) Please see the ``numpy/experimental_dtype_api.h`` header for exact
4435: (4) call information; the main thing to note is that the new-style loops
4436: (4) return 0 on success, -1 on failure. They are passed context as new
4437: (4) first input and ``auxdata`` as (replaced) last.
4438: (4) Only the ``strided_loop`` signature is considered guaranteed stable
4439: (4) for NumPy bug-fix releases. All other API is tied to the experimental
4440: (4) API versioning.
4441: (4) The reason for the split call is that cast information is required to
4442: (4) decide what the fixed-strides will be.

```

```

4443: (4)           NumPy ties the lifetime of the ``auxdata`` information to the capsule.
4444: (4)           """
4445: (0)           add_newdoc('numpy.core.multiarray', 'dtype',
4446: (4)           """
4447: (4)           dtype(dtype, align=False, copy=False, [metadata])
4448: (4)           Create a data type object.
4449: (4)           A numpy array is homogeneous, and contains elements described by a
4450: (4)           dtype object. A dtype object can be constructed from different
4451: (4)           combinations of fundamental numeric types.
4452: (4)           Parameters
4453: (4)           -----
4454: (4)           dtype
4455: (8)           Object to be converted to a data type object.
4456: (4)           align : bool, optional
4457: (8)           Add padding to the fields to match what a C compiler would output
4458: (8)           for a similar C-struct. Can be ``True`` only if `obj` is a dictionary
4459: (8)           or a comma-separated string. If a struct dtype is being created,
4460: (8)           this also sets a sticky alignment flag ``isalignedstruct``.
4461: (4)           copy : bool, optional
4462: (8)           Make a new copy of the data-type object. If ``False``, the result
4463: (8)           may just be a reference to a built-in data-type object.
4464: (4)           metadata : dict, optional
4465: (8)           An optional dictionary with dtype metadata.
4466: (4)           See also
4467: (4)           -----
4468: (4)           result_type
4469: (4)           Examples
4470: (4)           -----
4471: (4)           Using array-scalar type:
4472: (4)           >>> np.dtype(np.int16)
4473: (4)           dtype('int16')
4474: (4)           Structured type, one field name 'f1', containing int16:
4475: (4)           >>> np.dtype([('f1', np.int16)])
4476: (4)           dtype([('f1', '<i2')])
4477: (4)           Structured type, one field named 'f1', in itself containing a structured
4478: (4)           type with one field:
4479: (4)           >>> np.dtype([('f1', [('f1', np.int16)]))]
4480: (4)           dtype([('f1', [('f1', '<i2')])])
4481: (4)           Structured type, two fields: the first field contains an unsigned int, the
4482: (4)           second an int32:
4483: (4)           >>> np.dtype([('f1', np.uint64), ('f2', np.int32)])
4484: (4)           dtype([('f1', '<u8'), ('f2', '<i4')])
4485: (4)           Using array-protocol type strings:
4486: (4)           >>> np.dtype([('a','f8'),('b','S10')])
4487: (4)           dtype([('a', '<f8'), ('b', 'S10')])
4488: (4)           Using comma-separated field formats. The shape is (2,3):
4489: (4)           >>> np.dtype("i4, (2,3)f8")
4490: (4)           dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
4491: (4)           Using tuples. ``int`` is a fixed type, 3 the field's shape. ``void``
4492: (4)           is a flexible type, here of size 10:
4493: (4)           >>> np.dtype([('hello',(np.int64,3)),('world',np void,10)])
4494: (4)           dtype([('hello', '<i8', (3,)), ('world', 'V10')])
4495: (4)           Subdivide ``int16`` into 2 ``int8``'s, called x and y. 0 and 1 are
4496: (4)           the offsets in bytes:
4497: (4)           >>> np.dtype((np.int16, {'x':(np.int8,0), 'y':(np.int8,1)}))
4498: (4)           dtype((numpy.int16, [('x', 'i1'), ('y', 'i1')]))
4499: (4)           Using dictionaries. Two fields named 'gender' and 'age':
4500: (4)           >>> np.dtype({'names':['gender','age'], 'formats':['S1',np.uint8]})
4501: (4)           dtype([('gender', 'S1'), ('age', 'u1')])
4502: (4)           Offsets in bytes, here 0 and 25:
4503: (4)           >>> np.dtype({'surname':('S25',0), 'age':(np.uint8,25)})
4504: (4)           dtype([('surname', 'S25'), ('age', 'u1')])
4505: (4)           """
4506: (0)           add_newdoc('numpy.core.multiarray', 'dtype', ('alignment',
4507: (4)           """
4508: (4)           The required alignment (bytes) of this data-type according to the
compiler.
4509: (4)           More information is available in the C-API section of the manual.
4510: (4)           Examples

```

```

4511: (4)      -----
4512: (4)      >>> x = np.dtype('i4')
4513: (4)      >>> x.alignment
4514: (4)      4
4515: (4)      >>> x = np.dtype(float)
4516: (4)      >>> x.alignment
4517: (4)      8
4518: (4)      "'''")
4519: (0)      add_newdoc('numpy.core.multiarray', 'dtype', ('byteorder',
4520: (4)      """
4521: (4)          A character indicating the byte-order of this data-type object.
4522: (4)          One of:
4523: (4)          === =====
4524: (4)          '=' native
4525: (4)          '<' little-endian
4526: (4)          '>' big-endian
4527: (4)          '|' not applicable
4528: (4)          === =====
4529: (4)          All built-in data-type objects have byteorder either '=' or '|'.
4530: (4)          Examples
4531: (4)          -----
4532: (4)          >>> dt = np.dtype('i2')
4533: (4)          >>> dt.byteorder
4534: (4)          '='
4535: (4)          >>> # endian is not relevant for 8 bit numbers
4536: (4)          >>> np.dtype('i1').byteorder
4537: (4)          '|'
4538: (4)          >>> # or ASCII strings
4539: (4)          >>> np.dtype('S2').byteorder
4540: (4)          '|'
4541: (4)          >>> # Even if specific code is given, and it is native
4542: (4)          >>> # '=' is the byteorder
4543: (4)          >>> import sys
4544: (4)          >>> sys_is_le = sys.byteorder == 'little'
4545: (4)          >>> native_code = '<' if sys_is_le else '>'
4546: (4)          >>> swapped_code = '>' if sys_is_le else '<'
4547: (4)          >>> dt = np.dtype(native_code + 'i2')
4548: (4)          >>> dt.byteorder
4549: (4)          '='
4550: (4)          >>> # Swapped code shows up as itself
4551: (4)          >>> dt = np.dtype(swapped_code + 'i2')
4552: (4)          >>> dt.byteorder == swapped_code
4553: (4)          True
4554: (4)          "'''")
4555: (0)      add_newdoc('numpy.core.multiarray', 'dtype', ('char',
4556: (4)          """A unique character code for each of the 21 different built-in types.
4557: (4)          Examples
4558: (4)          -----
4559: (4)          >>> x = np.dtype(float)
4560: (4)          >>> x.char
4561: (4)          'd'
4562: (4)          "'''")
4563: (0)      add_newdoc('numpy.core.multiarray', 'dtype', ('descr',
4564: (4)          """
4565: (4)          `__array_interface__` description of the data-type.
4566: (4)          The format is that required by the 'descr' key in the
4567: (4)          `__array_interface__` attribute.
4568: (4)          Warning: This attribute exists specifically for `__array_interface__`,
4569: (4)          and passing it directly to `np.dtype` will not accurately reconstruct
4570: (4)          some dtypes (e.g., scalar and subarray dtypes).
4571: (4)          Examples
4572: (4)          -----
4573: (4)          >>> x = np.dtype(float)
4574: (4)          >>> x.descr
4575: (4)          [('', '<f8')]
4576: (4)          >>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
4577: (4)          >>> dt.descr
4578: (4)          [('name', '<U16'), ('grades', '<f8', (2,))]
4579: (4)          "'''")

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4580: (0)
4581: (4)
4582: (4)
4583: (4)
4584: (4)
4585: (6)
4586: (4)
4587: (4)
4588: (4)
4589: (4)
4590: (4)
``ndarray.getfield``
4591: (4)
4592: (4)
4593: (4)
4594: (4)
4595: (4)
4596: (4)
4597: (4)
4598: (4)
4599: (4)
4600: (4)
4601: (0)
4602: (4)
add_newdoc('numpy.core.multiarray', 'dtype', ('fields',
"")
Dictionary of named fields defined for this data type, or ``None``.
The dictionary is indexed by keys that are the names of the fields.
Each entry in the dictionary is a tuple fully describing the field::
    (dtype, offset[, title])
Offset is limited to C int, which is signed and usually 32 bits.
If present, the optional title can be any object (if it is a string
or unicode then it will also be a key in the fields dictionary,
otherwise it's meta-data). Notice also that the first two elements
of the tuple can be passed directly as arguments to the
`ndarray.getfield` and `ndarray.setfield` methods.
See Also
-----
ndarray.getfield, ndarray.setfield
Examples
-----
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print(dt.fields)
{'grades': (dtype('float64',(2,))), 16, 'name': (dtype('|S16'), 0)}
""")

4603: (4)
4604: (4)
4605: (4)
4606: (4)
4607: (4)
4608: (4)
4609: (4)
4610: (4)
4611: (4)
4612: (4)
4613: (4)
4614: (4)
4615: (4)
4616: (4)
4617: (4)
4618: (4)
4619: (0)
4620: (4)
add_newdoc('numpy.core.multiarray', 'dtype', ('flags',
"")
Bit-flags describing how this data type is to be interpreted.
Bit-masks are in `numpy.core.multiarray` as the constants
`ITEM_HASOBJECT`, `LIST_PICKLE`, `ITEM_IS_POINTER`, `NEEDS_INIT`,
`NEEDS_PYAPI`, `USE_GETITEM`, `USE_SETITEM`. A full explanation
of these flags is in C-API documentation; they are largely useful
for user-defined data-types.
The following example demonstrates that operations on this particular
dtype requires Python C-API.
Examples
-----
>>> x = np.dtype([('a', np.int32, 8), ('b', np.float64, 6)])
>>> x.flags
16
>>> np.core.multiarray.NEEDS_PYAPI
16
"""))

4621: (4)
4622: (4)
4623: (4)
4624: (4)
4625: (4)
4626: (4)
4627: (4)
4628: (4)
4629: (0)
4630: (4)
4631: (4)
4632: (4)
4633: (4)
add_newdoc('numpy.core.multiarray', 'dtype', ('hasobject',
"")
Boolean indicating whether this dtype contains any reference-counted
objects in any fields or sub-dtypes.
Recall that what is actually in the ndarray memory representing
the Python object is the memory address of that object (a pointer).
Special handling may be required, and this attribute is useful for
distinguishing data types that may contain arbitrary Python objects
and data-types that won't.
"""))

4634: (4)
4635: (4)
4636: (4)
4637: (7)
4638: (7)
4639: (7)
4640: (4)
=====
0 if this is a structured array type, with fields
1 if this is a dtype compiled into numpy (such as ints, floats etc)
2 if the dtype is for a user-defined numpy type
A user-defined type uses the numpy C-API machinery to extend
numpy to handle a new array type. See
:ref:`user.user-defined-data-types` in the NumPy manual.
=
```

=====

```

4641: (4)
4642: (4)
4643: (4)
4644: (4)
4645: (4)
        Examples
        -----
        >>> dt = np.dtype('i2')
        >>> dt.isbuiltin
        1

```

```

4646: (4)          >>> dt = np.dtype('f8')
4647: (4)          >>> dt.isbuiltin
4648: (4)          1
4649: (4)          >>> dt = np.dtype([('field1', 'f8')])
4650: (4)          >>> dt.isbuiltin
4651: (4)          0
4652: (4)          """
4653: (0)          add_newdoc('numpy.core.multiarray', 'dtype', ('isnative',
4654: (4)          """
4655: (4)          Boolean indicating whether the byte order of this dtype is native
4656: (4)          to the platform.
4657: (4)          """
4658: (0)          add_newdoc('numpy.core.multiarray', 'dtype', ('isalignedstruct',
4659: (4)          """
4660: (4)          Boolean indicating whether the dtype is a struct which maintains
4661: (4)          field alignment. This flag is sticky, so when combining multiple
4662: (4)          structs together, it is preserved and produces new dtypes which
4663: (4)          are also aligned.
4664: (4)          """
4665: (0)          add_newdoc('numpy.core.multiarray', 'dtype', ('itemsize',
4666: (4)          """
4667: (4)          The element size of this data-type object.
4668: (4)          For 18 of the 21 types this number is fixed by the data-type.
4669: (4)          For the flexible data-types, this number can be anything.
4670: (4)          Examples
4671: (4)          -----
4672: (4)          >>> arr = np.array([[1, 2], [3, 4]])
4673: (4)          >>> arr.dtype
4674: (4)          dtype('int64')
4675: (4)          >>> arr.itemsize
4676: (4)          8
4677: (4)          >>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
4678: (4)          >>> dt.itemsize
4679: (4)          80
4680: (4)          """
4681: (0)          add_newdoc('numpy.core.multiarray', 'dtype', ('kind',
4682: (4)          """
4683: (4)          A character code (one of 'biufcmMOSUV') identifying the general kind of
4684: (4)          data.
4685: (4)          =
4686: (4)          b  boolean
4687: (4)          i  signed integer
4688: (4)          u  unsigned integer
4689: (4)          f  floating-point
4690: (4)          c  complex floating-point
4691: (4)          m  timedelta
4692: (4)          M  datetime
4693: (4)          O  object
4694: (4)          S  (byte-)string
4695: (4)          U  Unicode
4696: (4)          V  void
4697: (4)          =
4698: (4)          Examples
4699: (4)          -----
4700: (4)          >>> dt = np.dtype('i4')
4701: (4)          >>> dt.kind
4702: (4)          'i'
4703: (4)          >>> dt = np.dtype('f8')
4704: (4)          >>> dt.kind
4705: (4)          'f'
4706: (4)          >>> dt = np.dtype([('field1', 'f8')])
4707: (4)          >>> dt.kind
4708: (4)          'V'
4709: (0)          """
4710: (4)          add_newdoc('numpy.core.multiarray', 'dtype', ('metadata',
4711: (4)          """
4712: (4)          Either ``None`` or a readonly dictionary of metadata (mappingproxy).
4713: (4)          The metadata field can be set using any dictionary at data-type
4714: (4)          creation. NumPy currently has no uniform approach to propagating

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4714: (4)           metadata; although some array operations preserve it, there is no
4715: (4)           guarantee that others will.
4716: (4)
4717: (8)           .. warning::
4718: (8)             Although used in certain projects, this feature was long undocumented
4719: (8)             and is not well supported. Some aspects of metadata propagation
4720: (4)             are expected to change in the future.
4721: (4)
4722: (4)           Examples
4723: (4)           -----
4724: (4)             >>> dt = np.dtype(float, metadata={"key": "value"})
4725: (4)             >>> dt.metadata["key"]
4726: (4)               'value'
4727: (4)             >>> arr = np.array([1, 2, 3], dtype=dt)
4728: (4)             >>> arr.dtype.metadata
4729: (4)               mappingproxy({'key': 'value'})
4730: (4)             Adding arrays with identical datatypes currently preserves the metadata:
4731: (4)             >>> (arr + arr).dtype.metadata
4732: (4)               mappingproxy({'key': 'value'})
4733: (4)             But if the arrays have different dtype metadata, the metadata may be
4734: (4)             dropped:
4735: (4)             >>> dt2 = np.dtype(float, metadata={"key2": "value2"})
4736: (4)             >>> arr2 = np.array([3, 2, 1], dtype=dt2)
4737: (4)             >>> (arr + arr2).dtype.metadata is None
4738: (0)               True # The metadata field is cleared so None is returned
4739: (4)
4740: (4)             add_newdoc('numpy.core.multiarray', 'dtype', ('name',
4741: (4)               """
4742: (4)                 A bit-width name for this data-type.
4743: (4)                 Un-sized flexible data-type objects do not have this attribute.
4744: (4)               Examples
4745: (4)               -----
4746: (4)                 >>> x = np.dtype(float)
4747: (4)                 >>> x.name
4748: (4)                   'float64'
4749: (4)                 >>> x = np.dtype([('a', np.int32, 8), ('b', np.float64, 6)])
4750: (4)                 >>> x.name
4751: (0)                   'void640'
4752: (4)
4753: (4)             add_newdoc('numpy.core.multiarray', 'dtype', ('names',
4754: (4)               """
4755: (4)                 Ordered list of field names, or ``None`` if there are no fields.
4756: (4)                 The names are ordered according to increasing byte offset. This can be
4757: (4)                 used, for example, to walk through all of the named fields in offset
4758: (4)               Examples
4759: (4)               -----
4760: (4)                 >>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
4761: (4)                 >>> dt.names
4762: (0)                   ('name', 'grades')
4763: (4)
4764: (4)             add_newdoc('numpy.core.multiarray', 'dtype', ('num',
4765: (4)               """
4766: (4)                 A unique number for each of the 21 different built-in types.
4767: (4)                 These are roughly ordered from least-to-most precision.
4768: (4)               Examples
4769: (4)               -----
4770: (4)                 >>> dt = np.dtype(str)
4771: (4)                 >>> dt.num
4772: (4)                   19
4773: (4)                 >>> dt = np.dtype(float)
4774: (4)                 >>> dt.num
4775: (0)                   12
4776: (4)
4777: (4)             add_newdoc('numpy.core.multiarray', 'dtype', ('shape',
4778: (4)               """
4779: (4)                 Shape tuple of the sub-array if this data type describes a sub-array,
4780: (4)                 and `()` otherwise.
4781: (4)               Examples
4782: (4)               -----
4783: (4)                 >>> dt = np.dtype(('i4', 4))

```

```

4782: (4)          >>> dt.shape
4783: (4)          (4,)
4784: (4)          >>> dt = np.dtype(('i4', (2, 3)))
4785: (4)          >>> dt.shape
4786: (4)          (2, 3)
4787: (4)          ""))
4788: (0)          add_newdoc('numpy.core.multiarray', 'dtype', ('ndim',
4789: (4)          """
4790: (4)          Number of dimensions of the sub-array if this data type describes a
4791: (4)          sub-array, and ``0`` otherwise.
4792: (4)          .. versionadded:: 1.13.0
4793: (4)          Examples
4794: (4)          -----
4795: (4)          >>> x = np.dtype(float)
4796: (4)          >>> x.ndim
4797: (4)          0
4798: (4)          >>> x = np.dtype((float, 8))
4799: (4)          >>> x.ndim
4800: (4)          1
4801: (4)          >>> x = np.dtype(('i4', (3, 4)))
4802: (4)          >>> x.ndim
4803: (4)          2
4804: (4)          ""))
4805: (0)          add_newdoc('numpy.core.multiarray', 'dtype', ('str',
4806: (4)          """
4807: (0)          "The array-protocol typestring of this data-type object.""))
4808: (4)          add_newdoc('numpy.core.multiarray', 'dtype', ('subdtype',
4809: (4)          """
4810: (4)          Tuple ``(item_dtype, shape)`` if this `dtype` describes a sub-array, and
4811: (4)          None otherwise.
4812: (4)          The *shape* is the fixed shape of the sub-array described by this
4813: (4)          data type, and *item_dtype* the data type of the array.
4814: (4)          If a field whose dtype object has this attribute is retrieved,
4815: (4)          then the extra dimensions implied by *shape* are tacked on to
4816: (4)          the end of the retrieved array.
4817: (4)          See Also
4818: (4)          -----
4819: (4)          dtype.base
4820: (4)          Examples
4821: (4)          -----
4822: (4)          >>> x = numpy.dtype('8f')
4823: (4)          >>> x.subdtype
4824: (4)          (dtype('float32'), (8,))
4825: (4)          >>> x = numpy.dtype('i2')
4826: (4)          >>>
4827: (4)          ""))
4828: (0)          add_newdoc('numpy.core.multiarray', 'dtype', ('base',
4829: (4)          """
4830: (4)          Returns dtype for the base element of the subarrays,
4831: (4)          regardless of their dimension or shape.
4832: (4)          See Also
4833: (4)          -----
4834: (4)          dtype.subdtype
4835: (4)          Examples
4836: (4)          -----
4837: (4)          >>> x = numpy.dtype('8f')
4838: (4)          >>> x.base
4839: (4)          dtype('float32')
4840: (4)          >>> x = numpy.dtype('i2')
4841: (4)          >>> x.base
4842: (4)          dtype('int16')
4843: (4)          ""))
4844: (0)          add_newdoc('numpy.core.multiarray', 'dtype', ('type',
4845: (4)          """
4846: (0)          "The type object used to instantiate a scalar of this data-type.""))
4847: (4)          add_newdoc('numpy.core.multiarray', 'dtype', ('newbyteorder',
4848: (4)          """
4849: (4)          newbyteorder(new_order='S', /)
4850: (4)          Return a new dtype with a different byte order.
4850: (4)          Changes are also made in all fields and sub-arrays of the data type.

```

```

4851: (4)             Parameters
4852: (4)
4853: (4)             new_order : string, optional
4854: (8)               Byte order to force; a value from the byte order specifications
4855: (8)               below. The default value ('S') results in swapping the current
4856: (8)               byte order. `new_order` codes can be any of:
4857: (8)                 * 'S' - swap dtype from current to opposite endian
4858: (8)                 * {'<', 'little'} - little endian
4859: (8)                 * {'>', 'big'} - big endian
4860: (8)                 * {'=', 'native'} - native order
4861: (8)                 * {'|', 'I'} - ignore (no change to byte order)
4862: (4)             Returns
4863: (4)
4864: (4)             new_dtype : dtype
4865: (8)               New dtype object with the given change to the byte order.
4866: (4)             Notes
4867: (4)
4868: (4)               Changes are also made in all fields and sub-arrays of the data type.
4869: (4)             Examples
4870: (4)
4871: (4)               >>> import sys
4872: (4)               >>> sys_is_le = sys.byteorder == 'little'
4873: (4)               >>> native_code = '<' if sys_is_le else '>'
4874: (4)               >>> swapped_code = '>' if sys_is_le else '<'
4875: (4)               >>> native_dt = np.dtype(native_code+'i2')
4876: (4)               >>> swapped_dt = np.dtype(swapped_code+'i2')
4877: (4)               >>> native_dt.newbyteorder('S') == swapped_dt
4878: (4)               True
4879: (4)               >>> native_dt.newbyteorder() == swapped_dt
4880: (4)               True
4881: (4)               >>> native_dt == swapped_dt.newbyteorder('S')
4882: (4)               True
4883: (4)               >>> native_dt == swapped_dt.newbyteorder('=')
4884: (4)               True
4885: (4)               >>> native_dt == swapped_dt.newbyteorder('N')
4886: (4)               True
4887: (4)               >>> native_dt == native_dt.newbyteorder('|')
4888: (4)               True
4889: (4)               >>> np.dtype('<i2') == native_dt.newbyteorder('<')
4890: (4)               True
4891: (4)               >>> np.dtype('<i2') == native_dt.newbyteorder('L')
4892: (4)               True
4893: (4)               >>> np.dtype('>i2') == native_dt.newbyteorder('>')
4894: (4)               True
4895: (4)               >>> np.dtype('>i2') == native_dt.newbyteorder('B')
4896: (4)               True
4897: (4)               """))
4898: (0)             add_newdoc('numpy.core.multiarray', 'dtype', ('__class_getitem__',
4899: (4)               """
4900: (4)                 __class_getitem__(item, /)
4901: (4)               Return a parametrized wrapper around the `~numpy.dtype` type.
4902: (4)               .. versionadded:: 1.22
4903: (4)             Returns
4904: (4)
4905: (4)             alias : types.GenericAlias
4906: (8)               A parametrized `~numpy.dtype` type.
4907: (4)             Examples
4908: (4)
4909: (4)               >>> import numpy as np
4910: (4)               >>> np.dtype[np.int64]
4911: (4)               numpy.dtype[numpy.int64]
4912: (4)             See Also
4913: (4)
4914: (4)               :pep:`585` : Type hinting generics in standard collections.
4915: (4)               """
4916: (0)             add_newdoc('numpy.core.multiarray', 'dtype', ('__ge__',
4917: (4)               """
4918: (4)                 __ge__(value, /)
4919: (4)               Return ``self >= value``.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4920: (4) Equivalent to ``np.can_cast(value, self, casting="safe")``.
4921: (4) See Also -----
4922: (4)
4923: (4) can_cast : Returns True if cast between data types can occur according to
4924: (15) the casting rule.
4925: (4) """
4926: (0) add_newdoc('numpy.core.multiarray', 'dtype', ('__le__', ,
4927: (4) """
4928: (4) __le__(value, /)
4929: (4) Return ``self <= value``.
4930: (4) Equivalent to ``np.can_cast(self, value, casting="safe")``.
4931: (4) See Also -----
4932: (4)
4933: (4) can_cast : Returns True if cast between data types can occur according to
4934: (15) the casting rule.
4935: (4) """
4936: (0) add_newdoc('numpy.core.multiarray', 'dtype', ('__gt__', ,
4937: (4) """
4938: (4) __ge__(value, /)
4939: (4) Return ``self > value``.
4940: (4) Equivalent to
4941: (4) ``self != value and np.can_cast(value, self, casting="safe")``.
4942: (4) See Also -----
4943: (4)
4944: (4) can_cast : Returns True if cast between data types can occur according to
4945: (15) the casting rule.
4946: (4) """
4947: (0) add_newdoc('numpy.core.multiarray', 'dtype', ('__lt__', ,
4948: (4) """
4949: (4) __lt__(value, /)
4950: (4) Return ``self < value``.
4951: (4) Equivalent to
4952: (4) ``self != value and np.can_cast(self, value, casting="safe")``.
4953: (4) See Also -----
4954: (4)
4955: (4) can_cast : Returns True if cast between data types can occur according to
4956: (15) the casting rule.
4957: (4) """
4958: (0) add_newdoc('numpy.core.multiarray', 'busdaycalendar',
4959: (4) """
4960: (4) busdaycalendar(weekmask='1111100', holidays=None)
4961: (4) A business day calendar object that efficiently stores information
4962: (4) defining valid days for the busday family of functions.
4963: (4) The default valid days are Monday through Friday ("business days").
4964: (4) A busdaycalendar object can be specified with any set of weekly
4965: (4) valid days, plus an optional "holiday" dates that always will be invalid.
4966: (4) Once a busdaycalendar object is created, the weekmask and holidays
4967: (4) cannot be modified.
4968: (4) .. versionadded:: 1.7.0
4969: (4) Parameters
4970: (4) -----
4971: (4) weekmask : str or array_like of bool, optional
4972: (8) A seven-element array indicating which of Monday through Sunday are
4973: (8) valid days. May be specified as a length-seven list or array, like
4974: (8) [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string
4975: (8) like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for
4976: (8) weekdays, optionally separated by white space. Valid abbreviations
4977: (8) are: Mon Tue Wed Thu Fri Sat Sun
4978: (4) holidays : array_like of datetime64[D], optional
4979: (8) An array of dates to consider as invalid dates, no matter which
4980: (8) weekday they fall upon. Holiday dates may be specified in any
4981: (8) order, and NaT (not-a-time) dates are ignored. This list is
4982: (8) saved in a normalized form that is suited for fast calculations
4983: (8) of valid days.
4984: (4) Returns
4985: (4) -----
4986: (4) out : busdaycalendar
4987: (8) A business day calendar object containing the specified
4988: (8) weekmask and holidays values.

```

```

4989: (4) See Also
4990: (4) -----
4991: (4) is_busday : Returns a boolean array indicating valid days.
4992: (4) busday_offset : Applies an offset counted in valid days.
4993: (4) busday_count : Counts how many valid days are in a half-open date range.
4994: (4) Attributes
4995: (4) -----
4996: (4) Note: once a busdaycalendar object is created, you cannot modify the
4997: (4) weekmask or holidays. The attributes return copies of internal data.
4998: (4) weekmask : (copy) seven-element array of bool
4999: (4) holidays : (copy) sorted array of datetime64[D]
5000: (4) Examples
5001: (4) -----
5002: (4) >>> # Some important days in July
5003: (4) ... bdd = np.busdaycalendar(
5004: (4) ...           holidays=['2011-07-01', '2011-07-04', '2011-07-17'])
5005: (4) >>> # Default is Monday to Friday weekdays
5006: (4) ... bdd.weekmask
5007: (4) array([ True,  True,  True,  True, False, False])
5008: (4) >>> # Any holidays already on the weekend are removed
5009: (4) ... bdd.holidays
5010: (4) array(['2011-07-01', '2011-07-04'], dtype='datetime64[D]')
5011: (4) """
5012: (0) add_newdoc('numpy.core.multiarray', 'busdaycalendar', ('weekmask',
5013: (4) """A copy of the seven-element boolean mask indicating valid days.""))
5014: (0) add_newdoc('numpy.core.multiarray', 'busdaycalendar', ('holidays',
5015: (4) """A copy of the holiday array indicating additional invalid days.""))
5016: (0) add_newdoc('numpy.core.multiarray', 'normalize_axis_index',
5017: (4) """
5018: (4)     normalize_axis_index(axis, ndim, msg_prefix=None)
5019: (4)     Normalizes an axis index, `axis`, such that is a valid positive index into
5020: (4)     the shape of array with `ndim` dimensions. Raises an AxisError with an
5021: (4)     appropriate message if this is not possible.
5022: (4)     Used internally by all axis-checking logic.
5023: (4) .. versionadded:: 1.13.0
5024: (4) Parameters
5025: (4) -----
5026: (4) axis : int
5027: (8)     The un-normalized index of the axis. Can be negative
5028: (4) ndim : int
5029: (8)     The number of dimensions of the array that `axis` should be normalized
5030: (8)     against
5031: (4) msg_prefix : str
5032: (8)     A prefix to put before the message, typically the name of the argument
5033: (4) Returns
5034: (4) -----
5035: (4) normalized_axis : int
5036: (8)     The normalized axis index, such that `0 <= normalized_axis < ndim`
5037: (4) Raises
5038: (4) -----
5039: (4) AxisError
5040: (8)     If the axis index is invalid, when `~ndim <= axis < ndim` is false.
5041: (4) Examples
5042: (4) -----
5043: (4) >>> normalize_axis_index(0, ndim=3)
5044: (4) 0
5045: (4) >>> normalize_axis_index(1, ndim=3)
5046: (4) 1
5047: (4) >>> normalize_axis_index(-1, ndim=3)
5048: (4) 2
5049: (4) >>> normalize_axis_index(3, ndim=3)
5050: (4) Traceback (most recent call last):
5051: (4) ...
5052: (4) AxisError: axis 3 is out of bounds for array of dimension 3
5053: (4) >>> normalize_axis_index(-4, ndim=3, msg_prefix='axes_arg')
5054: (4) Traceback (most recent call last):
5055: (4) ...
5056: (4) AxisError: axes_arg: axis -4 is out of bounds for array of dimension 3
5057: (4) """

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

5058: (0)
5059: (4)
5060: (4)
5061: (4)
5062: (4)
`numpy.datetime64` and
5063: (4)
5064: (4)
5065: (4)
5066: (4)
5067: (8)
5068: (4)
5069: (4)
5070: (4)
5071: (8)
5072: (8)
5073: (4)
5074: (8)
5075: (4)
5076: (4)
5077: (4)
5078: (4)
5079: (4)
5080: (4)
5081: (4)
5082: (4)
5083: (4)
5084: (4)
5085: (4)
5086: (4)
5087: (0)
5088: (4)
5089: (4)
5090: (4)
5091: (4)
5092: (4)
5093: (4)
5094: (4)
5095: (4)
5096: (0)
5097: (4)
5098: (4)
5099: (4)
5100: (4)
5101: (4)
5102: (0)
5103: (11)
5104: (0)
5105: (11)
5106: (0)
5107: (4)
5108: (0)
5109: (4)
5110: (0)
5111: (4)
5112: (0)
5113: (4)
5114: (0)
5115: (4)
5116: (0)
5117: (4)
5118: (0)
5119: (4)
5120: (0)
5121: (4)
5122: (0)
5123: (4)
5124: (0)
5125: (4)

    add_newdoc('numpy.core.multiarray', 'datetime_data',
               """
               datetime_data(dtype, /)
               Get information about the step size of a date or time type.
               The returned tuple can be passed as the second argument of
               `numpy.datetime64` and
               `numpy.timedelta64` .
               Parameters
               -----
               dtype : dtype
                   The dtype object, which must be a `datetime64` or `timedelta64` type.
               Returns
               -----
               unit : str
                   The :ref:`datetime unit <arrays.dtypes.dateunits>` on which this dtype
                   is based.
               count : int
                   The number of base units in a step.
               Examples
               -----
               >>> dt_25s = np.dtype('timedelta64[25s]')
               >>> np.datetime_data(dt_25s)
               ('s', 25)
               >>> np.array(10, dt_25s).astype('timedelta64[s]')
               array(250, dtype='timedelta64[s]')
               The result can be used to construct a datetime that uses the same units
               as a timedelta
               >>> np.datetime64('2010', np.datetime_data(dt_25s))
               numpy.datetime64('2010-01-01T00:00:00', '25s')
               """)

    add_newdoc('numpy.core.numericatypes', 'generic',
               """
               Base class for numpy scalar types.
               Class from which most (all?) numpy scalar types are derived. For
               consistency, exposes the same API as `ndarray`, despite many
               consequent attributes being either "get-only," or completely irrelevant.
               This is the class from which it is strongly suggested users should derive
               custom scalar types.
               """)

    def refer_to_array_attribute(attr, method=True):
        docstring = """
        Scalar {} identical to the corresponding array attribute.
        Please see `ndarray.{}`.
        """
        return attr, docstring.format("method" if method else "attribute", attr)

    add_newdoc('numpy.core.numericatypes', 'generic',
               refer_to_array_attribute('T', method=False))
    add_newdoc('numpy.core.numericatypes', 'generic',
               refer_to_array_attribute('base', method=False))
    add_newdoc('numpy.core.numericatypes', 'generic', ('data',
   """Pointer to start of data."""))
    add_newdoc('numpy.core.numericatypes', 'generic', ('dtype',
   """Get array data-descriptor."""))
    add_newdoc('numpy.core.numericatypes', 'generic', ('flags',
   """The integer value of flags."""))
    add_newdoc('numpy.core.numericatypes', 'generic', ('flat',
   """A 1-D view of the scalar."""))
    add_newdoc('numpy.core.numericatypes', 'generic', ('imag',
   """The imaginary part of the scalar."""))
    add_newdoc('numpy.core.numericatypes', 'generic', ('itemsize',
   """The length of one element in bytes."""))
    add_newdoc('numpy.core.numericatypes', 'generic', (' nbytes',
   """The length of the scalar in bytes."""))
    add_newdoc('numpy.core.numericatypes', 'generic', ('ndim',
   """The number of array dimensions."""))
    add_newdoc('numpy.core.numericatypes', 'generic', ('real',
   """The real part of the scalar."""))
    add_newdoc('numpy.core.numericatypes', 'generic', ('shape',
   """Tuple of array dimensions.""""))


```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

5126: (0)
5127: (4)
5128: (0)
5129: (4)
5130: (0)
5131: (11)
5132: (0)
5133: (11)
5134: (0)
5135: (11)
5136: (0)
5137: (11)
5138: (0)
5139: (11)
5140: (0)
5141: (11)
5142: (0)
5143: (11)
5144: (0)
5145: (11)
5146: (0)
5147: (11)
5148: (0)
5149: (11)
5150: (0)
5151: (11)
5152: (0)
5153: (11)
5154: (0)
5155: (11)
5156: (0)
5157: (11)
5158: (0)
5159: (11)
5160: (0)
5161: (11)
5162: (0)
5163: (11)
5164: (0)
5165: (11)
5166: (0)
5167: (11)
5168: (0)
5169: (11)
5170: (0)
5171: (11)
5172: (0)
5173: (11)
5174: (0)
5175: (11)
5176: (0)
5177: (11)
5178: (0)
5179: (11)
5180: (0)
5181: (4)
5182: (4)
5183: (4)
5184: (4)
5185: (4)
5186: (4)
5187: (4)
5188: (4)
5189: (4)
5190: (4)
5191: (4)
5192: (4)
5193: (4)
5194: (8)

add_newdoc('numpy.core.numericatypes', 'generic', ('size',
    """The number of elements in the gentype.""))
add_newdoc('numpy.core.numericatypes', 'generic', ('strides',
    """Tuple of bytes steps in each dimension.""))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('all'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('any'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('argmax'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('argmin'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('argsort'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('astype'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('byteswap'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('choose'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('clip'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('compress'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('conjugate'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('copy'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('cumprod'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('cumsum'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('diagonal'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('dump'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('dumps'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('fill'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('flatten'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('getfield'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('item'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('itemset'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('max'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('mean'))
add_newdoc('numpy.core.numericatypes', 'generic',
    refer_to_array_attribute('min'))
add_newdoc('numpy.core.numericatypes', 'generic', ('newbyteorder',
    """
        newbyteorder(new_order='S', /)
        Return a new `dtype` with a different byte order.
        Changes are also made in all fields and sub-arrays of the data type.
        The `new_order` code can be any from the following:
        * 'S' - swap dtype from current to opposite endian
        * {'<', 'little'} - little endian
        * {'>', 'big'} - big endian
        * {'=', 'native'} - native order
        * {'|', 'I'} - ignore (no change to byte order)
        Parameters
        -----
        new_order : str, optional
            Byte order to force; a value from the byte order specifications

```

```

5195: (8)           above. The default value ('S') results in swapping the current
5196: (8)           byte order.
5197: (4)           Returns
5198: (4)
5199: (4)
5200: (8)           -----
5201: (4)           new_dtype : dtype
5202: (0)           New `dtype` object with the given change to the byte order.
5203: (11)          """
5204: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5205: (11)             refer_to_array_attribute('nonzero'))
5206: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5207: (11)             refer_to_array_attribute('prod'))
5208: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5209: (11)             refer_to_array_attribute('ptp'))
5210: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5211: (11)             refer_to_array_attribute('put'))
5212: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5213: (11)             refer_to_array_attribute('ravel'))
5214: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5215: (11)             refer_to_array_attribute('repeat'))
5216: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5217: (11)             refer_to_array_attribute('reshape'))
5218: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5219: (11)             refer_to_array_attribute('resize'))
5220: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5221: (11)             refer_to_array_attribute('searchsorted'))
5222: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5223: (11)             refer_to_array_attribute('setfield'))
5224: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5225: (11)             refer_to_array_attribute('setflags'))
5226: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5227: (11)             refer_to_array_attribute('sort'))
5228: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5229: (11)             refer_to_array_attribute('squeeze'))
5230: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5231: (11)             refer_to_array_attribute('std'))
5232: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5233: (11)             refer_to_array_attribute('sum'))
5234: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5235: (11)             refer_to_array_attribute('swapaxes'))
5236: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5237: (11)             refer_to_array_attribute('take'))
5238: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5239: (11)             refer_to_array_attribute('tofile'))
5240: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5241: (11)             refer_to_array_attribute('tolist'))
5242: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5243: (11)             refer_to_array_attribute('tostring'))
5244: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5245: (11)             refer_to_array_attribute('trace'))
5246: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5247: (11)             refer_to_array_attribute('transpose'))
5248: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5249: (11)             refer_to_array_attribute('var'))
5250: (0)           add_newdoc('numpy.core.numericatypes', 'generic',
5251: (11)             refer_to_array_attribute('view'))
5252: (0)           add_newdoc('numpy.core.numericatypes', 'number', ('__class_getitem__',
5253: (4)             """
5254: (4)               __class_getitem__(item, /)
5255: (4)               Return a parametrized wrapper around the `~numpy.number` type.
5256: (4)               .. versionadded:: 1.22
5257: (4)               Returns
5258: (4)
5259: (4)               alias : types.GenericAlias
5260: (8)                 A parametrized `~numpy.number` type.
5261: (4)               Examples
5262: (4)
5263: (4)                 >>> from typing import Any

```

```

5264: (4)      >>> import numpy as np
5265: (4)      >>> np.signedinteger[Any]
5266: (4)      numpy.signedinteger[typing.Any]
5267: (4)      See Also
5268: (4)      -----
5269: (4)      :pep:`585` : Type hinting generics in standard collections.
5270: (4)      """")
5271: (0)      add_newdoc('numpy.core.numericatypes', 'number',
5272: (4)      """
5273: (4)      Abstract base class of all numeric scalar types.
5274: (4)      """")
5275: (0)      add_newdoc('numpy.core.numericatypes', 'integer',
5276: (4)      """
5277: (4)      Abstract base class of all integer scalar types.
5278: (4)      """")
5279: (0)      add_newdoc('numpy.core.numericatypes', 'signedinteger',
5280: (4)      """
5281: (4)      Abstract base class of all signed integer scalar types.
5282: (4)      """")
5283: (0)      add_newdoc('numpy.core.numericatypes', 'unsignedinteger',
5284: (4)      """
5285: (4)      Abstract base class of all unsigned integer scalar types.
5286: (4)      """")
5287: (0)      add_newdoc('numpy.core.numericatypes', 'inexact',
5288: (4)      """
5289: (4)      Abstract base class of all numeric scalar types with a (potentially)
5290: (4)      inexact representation of the values in its range, such as
5291: (4)      floating-point numbers.
5292: (4)      """")
5293: (0)      add_newdoc('numpy.core.numericatypes', 'floating',
5294: (4)      """
5295: (4)      Abstract base class of all floating-point scalar types.
5296: (4)      """")
5297: (0)      add_newdoc('numpy.core.numericatypes', 'complexfloating',
5298: (4)      """
5299: (4)      Abstract base class of all complex number scalar types that are made up of
5300: (4)      floating-point numbers.
5301: (4)      """")
5302: (0)      add_newdoc('numpy.core.numericatypes', 'flexible',
5303: (4)      """
5304: (4)      Abstract base class of all scalar types without predefined length.
5305: (4)      The actual size of these types depends on the specific `np.dtype`
5306: (4)      instantiation.
5307: (4)      """")
5308: (0)      add_newdoc('numpy.core.numericatypes', 'character',
5309: (4)      """
5310: (4)      Abstract base class of all character string scalar types.
5311: (4)      """")

```

---

## File 61 - \_add\_newdocs\_scalars.py:

```

1: (0)      """
2: (0)      This file is separate from ``_add_newdocs.py`` so that it can be mocked out by
3: (0)      our sphinx ``conf.py`` during doc builds, where we want to avoid showing
4: (0)      platform-dependent information.
5: (0)      """
6: (0)      import sys
7: (0)      import os
8: (0)      from numpy.core import dtype
9: (0)      from numpy.core import numericatypes as _numericatypes
10: (0)      from numpy.core.function_base import add_newdoc
11: (0)      def numeric_type_aliases(aliases):
12: (4)          def type_aliases_gen():
13: (8)              for alias, doc in aliases:
14: (12)                  try:
15: (16)                      alias_type = getattr(_numericatypes, alias)
16: (12)                  except AttributeError:

```

```

17: (16)           pass
18: (12)         else:
19: (16)             yield (alias_type, alias, doc)
20: (4)         return list(type_aliases_gen())
21: (0)     possible_aliases = numeric_type_aliases([
22: (4)         ('int8', '8-bit signed integer (``-128`` to ``127``)'),
23: (4)         ('int16', '16-bit signed integer (``-32_768`` to ``32_767``)'),
24: (4)         ('int32', '32-bit signed integer (``-2_147_483_648`` to
``2_147_483_647``)'),
25: (4)         ('int64', '64-bit signed integer (``-9_223_372_036_854_775_808`` to
``9_223_372_036_854_775_807``)'),
26: (4)         ('intptr', 'Signed integer large enough to fit pointer, compatible with C
``intptr_t``'),
27: (4)         ('uint8', '8-bit unsigned integer (``0`` to ``255``)'),
28: (4)         ('uint16', '16-bit unsigned integer (``0`` to ``65_535``)'),
29: (4)         ('uint32', '32-bit unsigned integer (``0`` to ``4_294_967_295``)'),
30: (4)         ('uint64', '64-bit unsigned integer (``0`` to
``18_446_744_073_709_551_615``)'),
31: (4)         ('uintptr', 'Unsigned integer large enough to fit pointer, compatible with C
``uintptr_t``'),
32: (4)         ('float16', '16-bit-precision floating-point number type: sign bit, 5 bits
exponent, 10 bits mantissa'),
33: (4)         ('float32', '32-bit-precision floating-point number type: sign bit, 8 bits
exponent, 23 bits mantissa'),
34: (4)         ('float64', '64-bit precision floating-point number type: sign bit, 11
bits exponent, 52 bits mantissa'),
35: (4)         ('float96', '96-bit extended-precision floating-point number type'),
36: (4)         ('float128', '128-bit extended-precision floating-point number type'),
37: (4)         ('complex64', 'Complex number type composed of 2 32-bit-precision
floating-point numbers'),
38: (4)         ('complex128', 'Complex number type composed of 2 64-bit-precision
floating-point numbers'),
39: (4)         ('complex192', 'Complex number type composed of 2 96-bit extended-
precision floating-point numbers'),
40: (4)         ('complex256', 'Complex number type composed of 2 128-bit extended-
precision floating-point numbers'),
41: (4)     ])
42: (0)     def _get_platform_and_machine():
43: (4)         try:
44: (8)             system, _, _, _, machine = os.uname()
45: (4)         except AttributeError:
46: (8)             system = sys.platform
47: (8)             if system == 'win32':
48: (12)                 machine = os.environ.get('PROCESSOR_ARCHITEW6432', '') \
49: (20)                     or os.environ.get('PROCESSOR_ARCHITECTURE', '')
50: (8)             else:
51: (12)                 machine = 'unknown'
52: (4)         return system, machine
53: (0)     _system, _machine = _get_platform_and_machine()
54: (0)     _doc_alias_string = f":Alias on this platform {_system} {_machine}:""
55: (0)     def add_newdoc_for_scalar_type(obj, fixed_aliases, doc):
56: (4)         o = getattr(_numerictypes, obj)
57: (4)         character_code = dtype(o).char
58: (4)         canonical_name_doc = "" if obj == o.__name__ else \
59: (24)                         f":Canonical name: `numpy.{obj}`\n      "
60: (4)         if fixed_aliases:
61: (8)             alias_doc = ''.join(f":Alias: `numpy.{alias}`\n      "
62: (28)                             for alias in fixed_aliases)
63: (4)         else:
64: (8)             alias_doc = ''
65: (4)             alias_doc += ''.join(f"{_doc_alias_string} `numpy.{alias}`: {doc}.\n      "
66: (25)                             for (alias_type, alias, doc) in possible_aliases if
alias_type is o)
67: (4)             docstring = f"""
68: (4)             {doc.strip()}
69: (4)             :Character code: ``'{character_code}'``
70: (4)             {canonical_name_doc}{alias_doc}
71: (4)             """
72: (4)             add_newdoc('numpy.core.numerictypes', obj, docstring)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

73: (0)
74: (4)
75: (4)
76: (4)
77: (7)
78: (7)
79: (7)
80: (7)
81: (4)
82: (0)
83: (4)
84: (4)
85: (4)
86: (0)
87: (4)
88: (4)
89: (4)
90: (0)
91: (4)
92: (4)
93: (4)
94: (0)
95: (4)
96: (4)
97: (4)
98: (0)
99: (4)
100: (4)
101: (4)
102: (0)
103: (4)
104: (4)
105: (4)
106: (0)
107: (4)
108: (4)
109: (4)
110: (0)
111: (4)
112: (4)
113: (4)
114: (0)
115: (4)
116: (4)
117: (4)
118: (0)
119: (4)
120: (4)
121: (4)
122: (0)
123: (4)
124: (4)
125: (4)
126: (0)
127: (4)
128: (4)
129: (4)
130: (0)
131: (4)
132: (4)
`float'
133: (4)
134: (4)
135: (0)
136: (4)
137: (4)
138: (4)
139: (4)
140: (0)

    add_newdoc_for_scalar_type('bool_', [], [
        """
        Boolean type (True or False), stored as a byte.
        .. warning::
            The :class:`bool` type is not a subclass of the :class:`int` type
            (the :class:`bool` is not even a number type). This is different
            than Python's default implementation of :class:`bool` as a
            sub-class of :class:`int`.
        """
    ])
    add_newdoc_for_scalar_type('byte', [], [
        """
        Signed integer type, compatible with C ``char``.
        """
    ])
    add_newdoc_for_scalar_type('short', [], [
        """
        Signed integer type, compatible with C ``short``.
        """
    ])
    add_newdoc_for_scalar_type('intc', [], [
        """
        Signed integer type, compatible with C ``int``.
        """
    ])
    add_newdoc_for_scalar_type('int_', [], [
        """
        Signed integer type, compatible with Python `int` and C ``long``.
        """
    ])
    add_newdoc_for_scalar_type('longlong', [], [
        """
        Signed integer type, compatible with C ``long long``.
        """
    ])
    add_newdoc_for_scalar_type('ubyte', [], [
        """
        Unsigned integer type, compatible with C ``unsigned char``.
        """
    ])
    add_newdoc_for_scalar_type('ushort', [], [
        """
        Unsigned integer type, compatible with C ``unsigned short``.
        """
    ])
    add_newdoc_for_scalar_type('uintc', [], [
        """
        Unsigned integer type, compatible with C ``unsigned int``.
        """
    ])
    add_newdoc_for_scalar_type('uint', [], [
        """
        Unsigned integer type, compatible with C ``unsigned long``.
        """
    ])
    add_newdoc_for_scalar_type('ulonglong', [], [
        """
        Signed integer type, compatible with C ``unsigned long long``.
        """
    ])
    add_newdoc_for_scalar_type('half', [], [
        """
        Half-precision floating-point number type.
        """
    ])
    add_newdoc_for_scalar_type('single', [], [
        """
        Single-precision floating-point number type, compatible with C ``float``.
        """
    ])
    add_newdoc_for_scalar_type('double', ['float_'], [
        """
        Double-precision floating-point number type, compatible with Python
        `float`
        and C ``double``.
        """
    ])
    add_newdoc_for_scalar_type('longdouble', ['longfloat'], [
        """
        Extended-precision floating-point number type, compatible with C
        ``long double`` but not necessarily with IEEE 754 quadruple-precision.
        """
    ])
    add_newdoc_for_scalar_type('csingle', ['singlecomplex'], [
        """
        """
    ])

```

```

141: (4)
142: (4)
143: (4)
144: (4)
145: (0)
146: (4)
147: (4)
148: (4)
149: (4)
150: (0)
151: (4)
152: (4)
153: (4)
154: (4)
155: (0)
156: (4)
157: (4)
158: (4)
159: (0)
160: (4)
161: (4)
162: (4)
163: (4)
164: (4)
165: (4)
166: (4)
exposing its
167: (4)
168: (4)
169: (4)
170: (4)
171: (4)
172: (4)
173: (4)
174: (0)
175: (4)
176: (4)
177: (4)
178: (4)
179: (0)
180: (4)
181: (4)
182: (4)
183: (4)
184: (4)
185: (4)
186: (7)
187: (7)
188: (7)
189: (7)
190: (7)
191: (7)
192: (4)
193: (8)
194: (8)
195: (8)
196: (7)
197: (4)
198: (4)
199: (4)
200: (4)
201: (4)
202: (4)
203: (7)
204: (4)
205: (7)
206: (4)
207: (7)
208: (4)

        """
        Complex number type composed of two single-precision floating-point
        numbers.
        """
add_newdoc_for_scalar_type('cdouble', ['cfloat', 'complex_'],
                           """
        Complex number type composed of two double-precision floating-point
        numbers, compatible with Python `complex`.
        """
add_newdoc_for_scalar_type('clongdouble', ['clongfloat', 'longcomplex'],
                           """
        Complex number type composed of two extended-precision floating-point
        numbers.
        """
add_newdoc_for_scalar_type('object_', [],
                           """
        Any Python object.
        """
add_newdoc_for_scalar_type('str_', ['unicode_'],
                           r"""
        A unicode string.
        This type strips trailing null codepoints.
        >>> s = np.str_("abc\x00")
        >>> s
        'abc'
        Unlike the builtin `str`, this supports the :ref:`python:bufferobjects`,
        contents as UCS4:
        >>> m = memoryview(np.str_("abc"))
        >>> m.format
        '3w'
        >>> m.tobytes()
        b'a\x00\x00\x00b\x00\x00\x00c\x00\x00\x00\x00'
        """
add_newdoc_for_scalar_type('bytes_', ['string_'],
                           r"""
        A byte string.
        When used in arrays, this type strips trailing null bytes.
        """
add_newdoc_for_scalar_type('void', [],
                           r"""
        np.void(length_or_data, /, dtype=None)
        Create a new structured or unstructured void scalar.
        Parameters
        -----
        length_or_data : int, array-like, bytes-like, object
            One of multiple meanings (see notes). The length or
            bytes data of an unstructured void. Or alternatively,
            the data to be stored in the new scalar when `dtype`
            is provided.
        This can be an array-like, in which case an array may
        be returned.
        dtype : dtype, optional
            If provided the dtype of the new scalar. This dtype must
            be "void" dtype (i.e. a structured or unstructured void,
            see also :ref:`defining-structured-types`).
        ..versionadded:: 1.24
        Notes
        -----
        For historical reasons and because void scalars can represent both
        arbitrary byte data and structured dtypes, the void constructor
        has three calling conventions:
        1. ``np.void(5)`` creates a ``dtype="V5"`` scalar filled with five
           ``\0`` bytes. The 5 can be a Python or NumPy integer.
        2. ``np.void(b"bytes-like")`` creates a void scalar from the byte string.
           The dtype itemsize will match the byte string length, here ``"V10"``.
        3. When a ``dtype=`` is passed the call is roughly the same as an
           array creation. However, a void scalar rather than array is returned.
           Please see the examples which show all three different conventions.

```

```

209: (4)          Examples
210: (4)          -----
211: (4)          >>> np.void(5)
212: (4)          void(b'\x00\x00\x00\x00\x00\x00')
213: (4)          >>> np.void(b'abcd')
214: (4)          void(b'\x61\x62\x63\x64')
215: (4)          >>> np.void((5, 3.2, "eggs"), dtype="i,d,S5")
216: (4)          (5, 3.2, b'eggs') # looks like a tuple, but is `np.void`
217: (4)          >>> np.void(3, dtype=[('x', np.int8), ('y', np.int8)])
218: (4)          (3, 3) # looks like a tuple, but is `np.void`
219: (4)          """
220: (0)          add_newdoc_for_scalar_type('datetime64', [], [
221: (4)          """
222: (4)          If created from a 64-bit integer, it represents an offset from
223: (4)          ``1970-01-01T00:00:00``.
224: (4)          If created from string, the string can be in ISO 8601 date
225: (4)          or datetime format.
226: (4)          >>> np.datetime64(10, 'Y')
227: (4)          numpy.datetime64('1980')
228: (4)          >>> np.datetime64('1980', 'Y')
229: (4)          numpy.datetime64('1980')
230: (4)          >>> np.datetime64(10, 'D')
231: (4)          numpy.datetime64('1970-01-11')
232: (4)          See :ref:`arrays.datetime` for more information.
233: (4)          """
234: (0)          add_newdoc_for_scalar_type('timedelta64', [], [
235: (4)          """
236: (4)          A timedelta stored as a 64-bit integer.
237: (4)          See :ref:`arrays.datetime` for more information.
238: (4)          """
239: (0)          add_newdoc('numpy.core.numericatypes', "integer", ('is_integer',
240: (4)          """
241: (4)              integer.is_integer() -> bool
242: (4)              Return ``True`` if the number is finite with integral value.
243: (4)              .. versionadded:: 1.22
244: (4)              Examples
245: (4)              -----
246: (4)              >>> np.int64(-2).is_integer()
247: (4)              True
248: (4)              >>> np.uint32(5).is_integer()
249: (4)              True
250: (4)              """
251: (0)          for float_name in ('half', 'single', 'double', 'longdouble'):
252: (4)              add_newdoc('numpy.core.numericatypes', float_name, ('as_integer_ratio',
253: (8)              """
254: (8)                  {ftype}.as_integer_ratio() -> (int, int)
255: (8)                  Return a pair of integers, whose ratio is exactly equal to the
256: (8)          original
257: (8)          floating point number, and with a positive denominator.
258: (8)          Raise `OverflowError` on infinities and a `ValueError` on NaNs.
259: (8)          >>> np.{ftype}(10.0).as_integer_ratio()
260: (8)          (10, 1)
261: (8)          >>> np.{ftype}(0.0).as_integer_ratio()
262: (8)          (0, 1)
263: (8)          >>> np.{ftype}(-.25).as_integer_ratio()
264: (8)          (-1, 4)
265: (4)          """ .format(ftype=ftype)))
266: (8)          add_newdoc('numpy.core.numericatypes', float_name, ('is_integer',
267: (8)          """
268: (8)              {float_name}.is_integer() -> bool
269: (8)              Return ``True`` if the floating point number is finite with integral
270: (8)              value, and ``False`` otherwise.
271: (8)              .. versionadded:: 1.22
272: (8)              Examples
273: (8)              -----
274: (8)              >>> np.{float_name}(-2.0).is_integer()
275: (8)              True
276: (8)              >>> np.{float_name}(3.2).is_integer()
276: (8)              False

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

277: (8)          """")
278: (0)          for int_name in ('int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32',
279: (8)                  'int64', 'uint64', 'int64', 'uint64', 'int64', 'uint64'):
280: (4)                  add_newdoc('numpy.core.numericatypes', int_name, ('bit_count',
281: (8)                      f""""
282: (8)                          {int_name}.bit_count() -> int
283: (8)                          Computes the number of 1-bits in the absolute value of the input.
284: (8)                          Analogous to the builtin `int.bit_count` or ``popcount`` in C++.
285: (8)                          Examples
286: (8)                          -----
287: (8)                          >>> np.{int_name}(127).bit_count()
288: (8)                          7"""
289: (8)                          (f"""
290: (8)                          >>> np.{int_name}(-127).bit_count()
291: (8)                          7
292: (8)                          """ if dtype(int_name).char.islower() else ""))

```

---

## File 62 - \_asarray.py:

```

1: (0)          """
2: (0)          Functions in the ``as*array`` family that promote array-likes into arrays.
3: (0)          ``require`` fits this category despite its name not matching this pattern.
4: (0)          """
5: (0)          from .overrides import (
6: (4)                  array_function_dispatch,
7: (4)                  set_array_function_like_doc,
8: (4)                  set_module,
9: (0)          )
10: (0)         from .multiarray import array, asanyarray
11: (0)         __all__ = ["require"]
12: (0)         POSSIBLE_FLAGS = {
13: (4)             'C': 'C', 'C_CONTIGUOUS': 'C', 'CONTIGUOUS': 'C',
14: (4)             'F': 'F', 'F_CONTIGUOUS': 'F', 'FORTRAN': 'F',
15: (4)             'A': 'A', 'ALIGNED': 'A',
16: (4)             'W': 'W', 'WRITEABLE': 'W',
17: (4)             'O': 'O', 'OWNDATA': 'O',
18: (4)             'E': 'E', 'ENSUREARRAY': 'E'
19: (0)         }
20: (0)         @set_array_function_like_doc
21: (0)         @set_module('numpy')
22: (0)         def require(a, dtype=None, requirements=None, *, like=None):
23: (4)             """
24: (4)                 Return an ndarray of the provided type that satisfies requirements.
25: (4)                 This function is useful to be sure that an array with the correct flags
26: (4)                 is returned for passing to compiled code (perhaps through ctypes).
27: (4)                 Parameters
28: (4)                 -----
29: (4)                 a : array_like
30: (7)                     The object to be converted to a type-and-requirement-satisfying array.
31: (4)                 dtype : data-type
32: (7)                     The required data-type. If None preserve the current dtype. If your
33: (7)                     application requires the data to be in native byteorder, include
34: (7)                     a byteorder specification as a part of the dtype specification.
35: (4)                     requirements : str or sequence of str
36: (7)                         The requirements list can be any of the following
37: (7)                         * 'F_CONTIGUOUS' ('F') - ensure a Fortran-contiguous array
38: (7)                         * 'C_CONTIGUOUS' ('C') - ensure a C-contiguous array
39: (7)                         * 'ALIGNED' ('A') - ensure a data-type aligned array
40: (7)                         * 'WRITEABLE' ('W') - ensure a writable array
41: (7)                         * 'OWNDATA' ('O') - ensure an array that owns its own data
42: (7)                         * 'ENSUREARRAY', ('E') - ensure a base array, instead of a subclass
43: (4)                         ${ARRAY_FUNCTION_LIKE}
44: (8)                             .. versionadded:: 1.20.0
45: (4)                         Returns
46: (4)                         -----
47: (4)                         out : ndarray
48: (8)                             Array with specified requirements and type if given.

```

```

49: (4)          See Also
50: (4)          -----
51: (4)          asarray : Convert input to an ndarray.
52: (4)          asanyarray : Convert to an ndarray, but pass through ndarray subclasses.
53: (4)          ascontiguousarray : Convert input to a contiguous array.
54: (4)          asfortranarray : Convert input to an ndarray with column-major
55: (21)          memory order.
56: (4)          ndarray.flags : Information about the memory layout of the array.
57: (4)          Notes
58: (4)          -----
59: (4)          The returned array will be guaranteed to have the listed requirements
60: (4)          by making a copy if needed.
61: (4)          Examples
62: (4)          -----
63: (4)          >>> x = np.arange(6).reshape(2,3)
64: (4)          >>> x.flags
65: (6)          C_CONTIGUOUS : True
66: (6)          F_CONTIGUOUS : False
67: (6)          OWNDATA : False
68: (6)          WRITEABLE : True
69: (6)          ALIGNED : True
70: (6)          WRITEBACKIFCOPY : False
71: (4)          >>> y = np.require(x, dtype=np.float32, requirements=['A', 'O', 'W', 'F'])
72: (4)          >>> y.flags
73: (6)          C_CONTIGUOUS : False
74: (6)          F_CONTIGUOUS : True
75: (6)          OWNDATA : True
76: (6)          WRITEABLE : True
77: (6)          ALIGNED : True
78: (6)          WRITEBACKIFCOPY : False
79: (4)          """
80: (4)          if like is not None:
81: (8)              return _require_with_like(
82: (12)                  like,
83: (12)                  a,
84: (12)                  dtype=dtype,
85: (12)                  requirements=requirements,
86: (8)              )
87: (4)          if not requirements:
88: (8)              return asanyarray(a, dtype=dtype)
89: (4)          requirements = {POSSIBLE_FLAGS[x.upper()] for x in requirements}
90: (4)          if 'E' in requirements:
91: (8)              requirements.remove('E')
92: (8)              subok = False
93: (4)          else:
94: (8)              subok = True
95: (4)          order = 'A'
96: (4)          if requirements >= {'C', 'F'}:
97: (8)              raise ValueError('Cannot specify both "C" and "F" order')
98: (4)          elif 'F' in requirements:
99: (8)              order = 'F'
100: (8)              requirements.remove('F')
101: (4)          elif 'C' in requirements:
102: (8)              order = 'C'
103: (8)              requirements.remove('C')
104: (4)          arr = array(a, dtype=dtype, order=order, copy=False, subok=subok)
105: (4)          for prop in requirements:
106: (8)              if not arr.flags[prop]:
107: (12)                  return arr.copy(order)
108: (4)          return arr
109: (0)          _require_with_like = array_function_dispatch()(require)

-----

```

File 63 - \_dtype.py:

```

1: (0)          """
2: (0)          A place for code to be called from the implementation of np.dtype
3: (0)          String handling is much easier to do correctly in python.

```

```

4: (0)
5: (0)
6: (0)
7: (4)
8: (4)
9: (4)
10: (4)
11: (4)
12: (4)
13: (4)
14: (4)
15: (4)
16: (4)
17: (4)
18: (0)
19: (0)
20: (4)
21: (8)
22: (4)
23: (8)
24: (12)
25: (12)
26: (8)
27: (0)
28: (4)
29: (8)
30: (4)
31: (8)
32: (4)
33: (8)
34: (4)
35: (8)
36: (0)
37: (4)
38: (4)
39: (8)
40: (4)
41: (0)
42: (4)
43: (4)
44: (4)
45: (4)
46: (4)
47: (4)
48: (0)
49: (4)
50: (0)
51: (4)
52: (4)
53: (4)
54: (4)
55: (4)
56: (4)
57: (4)
58: (4)
59: (4)
60: (4)
61: (8)
instead
62: (8)
63: (4)
64: (8)
65: (8)
66: (8)
67: (8)
68: (8)
69: (8)
70: (8)
71: (8)

    """
    import numpy as np
    _kind_to_stem = {
        'u': 'uint',
        'i': 'int',
        'c': 'complex',
        'f': 'float',
        'b': 'bool',
        'V': 'void',
        'O': 'object',
        'M': 'datetime',
        'm': 'timedelta',
        'S': 'bytes',
        'U': 'str',
    }
def _kind_name(dtype):
    try:
        return _kind_to_stem[dtype.kind]
    except KeyError as e:
        raise RuntimeError(
            "internal dtype error, unknown kind {!r}"
            .format(dtype.kind)
        ) from None
def __str__(dtype):
    if dtype.fields is not None:
        return _struct_str(dtype, include_align=True)
    elif dtype.subdtype:
        return _subarray_str(dtype)
    elif issubclass(dtype.type, np.flexible) or not dtype.isnative:
        return dtype.str
    else:
        return dtype.name
def __repr__(dtype):
    arg_str = _construction_repr(dtype, include_align=False)
    if dtype.isalignedstruct:
        arg_str = arg_str + ", align=True"
    return "dtype({})".format(arg_str)
def _unpack_field(dtype, offset, title=None):
    """
        Helper function to normalize the items in dtype.fields.
        Call as:
        dtype, offset, title = _unpack_field(*dtype.fields[name])
    """
    return dtype, offset, title
def _unsized(dtype):
    return dtype.itemsize == 0
def _construction_repr(dtype, include_align=False, short=False):
    """
        Creates a string repr of the dtype, excluding the 'dtype()' part
        surrounding the object. This object may be a string, a list, or
        a dict depending on the nature of the dtype. This
        is the object passed as the first parameter to the dtype
        constructor, and if no additional constructor parameters are
        given, will reproduce the exact memory layout.
        Parameters
        -----
        short : bool
            If true, this creates a shorter repr using 'kind' and 'itemsize',
            instead
            of the longer type name.
        include_align : bool
            If true, this includes the 'align=True' parameter
            inside the struct dtype construction dict when needed. Use this flag
            if you want a proper repr string without the 'dtype()' part around it.
            If false, this does not preserve the
            'align=True' parameter or sticky NPY_ALIGNED_STRUCT flag for
            struct arrays like the regular repr does, because the 'align'
            flag is not part of first dtype constructor parameter. This
            mode is intended for a full 'repr', where the 'align=True' is
    """

```

```

72: (8)           provided as the second parameter.
73: (4)
74: (4)
75: (8)           if dtype.fields is not None:
76: (4)             return _struct_str(dtype, include_align=include_align)
77: (4)           elif dtype.subdtype:
78: (4)             return _subarray_str(dtype)
79: (8)           else:
80: (0)             return _scalar_str(dtype, short=short)
81: (4)
82: (4)
83: (8)
84: (12)
85: (8)
86: (12)
87: (4)
88: (8)
89: (4)
90: (8)
91: (12)
92: (8)
93: (12)
94: (4)
95: (8)
96: (12)
97: (8)
98: (12)
99: (4)
100: (8)
101: (12)
102: (8)
103: (12)
104: (4)
105: (8)
106: (4)
107: (8)
108: (4)
109: (8)
110: (12)
111: (8)
112: (12)
113: (4)
114: (8)
115: (4)
116: (8)
117: (12)
118: (0)
119: (4)
120: (4)
121: (4)
122: (4)
123: (4)
124: (8)
125: (4)
126: (8)
127: (4)
128: (8)
129: (4)
130: (8)
131: (0)
132: (4)
133: (4)
134: (8)
135: (4)
136: (8)
137: (4)
138: (8)
139: (0)
140: (4)

    """
    if dtype.fields is not None:
        return _struct_str(dtype, include_align=include_align)
    elif dtype.subdtype:
        return _subarray_str(dtype)
    else:
        return _scalar_str(dtype, short=short)

def _scalar_str(dtype, short):
    byteorder = _byte_order_str(dtype)
    if dtype.type == np.bool_:
        if short:
            return "?"
        else:
            return "bool"
    elif dtype.type == np.object_:
        return "O"
    elif dtype.type == np.bytes_:
        if _isunsized(dtype):
            return "S"
        else:
            return "S%d" % dtype.itemsize
    elif dtype.type == np.str_:
        if _isunsized(dtype):
            return "%sU" % byteorder
        else:
            return "%sU%d" % (byteorder, dtype.itemsize / 4)
    elif issubclass(dtype.type, np.void):
        if _isunsized(dtype):
            return "V"
        else:
            return "V%d" % dtype.itemsize
    elif dtype.type == np.datetime64:
        return "%sM8%" % (byteorder, _datetime_metadata_str(dtype))
    elif dtype.type == np.timedelta64:
        return "%sm8%" % (byteorder, _datetime_metadata_str(dtype))
    elif np.issubdtype(dtype, np.number):
        if short or dtype.byteorder not in ('=', '|'):
            return "%c%d" % (byteorder, dtype.kind, dtype.itemsize)
        else:
            return "%s%d" % (_kind_name(dtype), 8 * dtype.itemsize)
    elif dtype.isbuiltin == 2:
        return dtype.type.__name__
    else:
        raise RuntimeError(
            "Internal error: NumPy dtype unrecognized type number")

def _byte_order_str(dtype):
    """
    Normalize byteorder to '<' or '>'

    swapped = np.dtype(int).newbyteorder('S')
    native = swapped.newbyteorder('S')
    byteorder = dtype.byteorder
    if byteorder == '=':
        return native.byteorder
    if byteorder == 'S':
        return swapped.byteorder
    elif byteorder == '|':
        return ''
    else:
        return byteorder

    """
    unit, count = np.datetime_data(dtype)
    if unit == 'generic':
        return ''
    elif count == 1:
        return '[{}].format(unit)"
    else:
        return '[{}{}].format(count, unit)"

def _struct_dict_str(dtype, includealignedflag):
    names = dtype.names

```

```

141: (4) fld_dtypes = []
142: (4) offsets = []
143: (4) titles = []
144: (4) for name in names:
145: (8)     fld_dtype, offset, title = _unpack_field(*dtype.fields[name])
146: (8)     fld_dtypes.append(fld_dtype)
147: (8)     offsets.append(offset)
148: (8)     titles.append(title)
149: (4) if np.core.arrayprint._get_legacy_print_mode() <= 121:
150: (8)     colon = ":"
151: (8)     fieldsep = ","
152: (4) else:
153: (8)     colon = " : "
154: (8)     fieldsep = ", "
155: (4) ret = "{'names':%s[" % colon
156: (4) ret += fieldsep.join(repr(name) for name in names)
157: (4) ret += "], 'formats':%s[" % colon
158: (4) ret += fieldsep.join(
159: (8)     _construction_repr(fld_dtype, short=True) for fld_dtype in fld_dtypes)
160: (4) ret += "], 'offsets':%s[" % colon
161: (4) ret += fieldsep.join("%d" % offset for offset in offsets)
162: (4) if any(title is not None for title in titles):
163: (8)     ret += "], 'titles':%s[" % colon
164: (8)     ret += fieldsep.join(repr(title) for title in titles)
165: (4)     ret += "], 'itemsize':%s%d" % (colon, dtype.itemsize)
166: (4)     if (includealignedflag and dtype.isalignedstruct):
167: (8)         ret += ", 'aligned':%sTrue}" % colon
168: (4)     else:
169: (8)         ret += "}"
170: (4) return ret
171: (0) def _aligned_offset(offset, alignment):
172: (4)     return -(-offset // alignment) * alignment
173: (0) def _is_packed(dtype):
174: (4) """
175: (4)     Checks whether the structured data type in 'dtype'
176: (4)     has a simple layout, where all the fields are in order,
177: (4)     and follow each other with no alignment padding.
178: (4)     When this returns true, the dtype can be reconstructed
179: (4)     from a list of the field names and dtypes with no additional
180: (4)     dtype parameters.
181: (4)    Duplicates the C `is_dtype_struct_simple_unaligned_layout` function.
182: (4) """
183: (4)     align = dtype.isalignedstruct
184: (4)     max_alignment = 1
185: (4)     total_offset = 0
186: (4)     for name in dtype.names:
187: (8)         fld_dtype, fld_offset, title = _unpack_field(*dtype.fields[name])
188: (8)         if align:
189: (12)             total_offset = _aligned_offset(total_offset, fld_dtype.alignment)
190: (12)             max_alignment = max(max_alignment, fld_dtype.alignment)
191: (8)             if fld_offset != total_offset:
192: (12)                 return False
193: (8)             total_offset += fld_dtype.itemsize
194: (4)     if align:
195: (8)         total_offset = _aligned_offset(total_offset, max_alignment)
196: (4)     if total_offset != dtype.itemsize:
197: (8)         return False
198: (4)     return True
199: (0) def _struct_list_str(dtype):
200: (4)     items = []
201: (4)     for name in dtype.names:
202: (8)         fld_dtype, fld_offset, title = _unpack_field(*dtype.fields[name])
203: (8)         item = "("
204: (8)         if title is not None:
205: (12)             item += "({!r}, {!r}), ".format(title, name)
206: (8)         else:
207: (12)             item += "{!r}, ".format(name)
208: (8)         if fld_dtype.subdtype is not None:
209: (12)             base, shape = fld_dtype.subdtype

```

```

210: (12)             item += "{}, {}".format(
211: (16)                 _construction_repr(base, short=True),
212: (16)                 shape
213: (12)             )
214: (8)             else:
215: (12)                 item += _construction_repr(fld_dtype, short=True)
216: (8)             item += ")"
217: (8)             items.append(item)
218: (4)             return "[" + ", ".join(items) + "]"
219: (0)             def _struct_str(dtype, include_align):
220: (4)                 if not (include_align and dtype.isalignedstruct) and _is_packed(dtype):
221: (8)                     sub = _struct_list_str(dtype)
222: (4)                 else:
223: (8)                     sub = _struct_dict_str(dtype, include_align)
224: (4)                 if dtype.type != np.void:
225: (8)                     return "{t.__module__}.{t.__name__}, {f})".format(t=dtype.type,
f=sub)
226: (4)                 else:
227: (8)                     return sub
228: (0)             def _subarray_str(dtype):
229: (4)                 base, shape = dtype.subdtype
230: (4)                 return "({}, {})".format(
231: (8)                     _construction_repr(base, short=True),
232: (8)                     shape
233: (4)                 )
234: (0)             def _name_includes_bit_suffix(dtype):
235: (4)                 if dtype.type == np.object_:
236: (8)                     return False
237: (4)                 elif dtype.type == np.bool_:
238: (8)                     return False
239: (4)                 elif dtype.type is None:
240: (8)                     return True
241: (4)                 elif np.issubdtype(dtype, np.flexible) and _isunsized(dtype):
242: (8)                     return False
243: (4)                 else:
244: (8)                     return True
245: (0)             def _name_get(dtype):
246: (4)                 if dtype.isbuiltin == 2:
247: (8)                     return dtype.type.__name__
248: (4)                 if dtype.kind == '\x00':
249: (8)                     name = type(dtype).__name__
250: (4)                 elif issubclass(dtype.type, np.void):
251: (8)                     name = dtype.type.__name__
252: (4)                 else:
253: (8)                     name = _kind_name(dtype)
254: (4)                 if _name_includes_bit_suffix(dtype):
255: (8)                     name += "{}".format(dtype.itemsize * 8)
256: (4)                 if dtype.type in (np.datetime64, np.timedelta64):
257: (8)                     name += _datetime_metadata_str(dtype)
258: (4)             return name

```

---

## File 64 - \_dtype\_ctypes.py:

```

1: (0)             """
2: (0)             Conversion from ctypes to dtype.
3: (0)             In an ideal world, we could achieve this through the PEP3118 buffer protocol,
4: (0)             something like::
5: (4)                 def dtype_from_ctypes_type(t):
6: (8)                     class DummyStruct(ctypes.Structure):
7: (12)                         __fields__ = [('a', t)]
8: (8)                         ctype_0 = (DummyStruct * 0)()
9: (8)                         mv = memoryview(ctype_0)
10: (8)                         return _dtype_from_pep3118(mv.format)[ 'a' ]
11: (0)             Unfortunately, this fails because:
12: (0)                 * ctypes cannot handle length-0 arrays with PEP3118 (bpo-32782)
13: (0)                 * PEP3118 cannot represent unions, but both numpy and ctypes can
14: (0)                 * ctypes cannot handle big-endian structs with PEP3118 (bpo-32780)

```

```

15: (0)
16: (0)
17: (0)
18: (4)
19: (0)
20: (4)
21: (8)
22: (12)
23: (16)
24: (4)
25: (8)
26: (8)
27: (8)
28: (8)
29: (8)
30: (8)
31: (12)
32: (12)
33: (12)
34: (12)
effective_pack) * effective_pack
35: (12)
36: (12)
37: (8)
38: (12)
39: (12)
40: (12)
41: (12)
42: (4)
43: (8)
44: (8)
45: (12)
46: (8)
47: (0)
48: (4)
49: (4)
50: (4)
51: (4)
52: (8)
53: (4)
54: (8)
55: (4)
56: (8)
57: (0)
58: (4)
59: (4)
60: (4)
61: (4)
62: (4)
63: (8)
64: (8)
65: (8)
66: (4)
67: (8)
68: (8)
69: (8)
70: (8)
71: (0)
72: (4)
73: (4)
74: (4)
75: (4)
76: (4)
77: (8)
78: (4)
79: (8)
80: (4)
81: (8)
82: (4)

      """
      import numpy as np
      def _from_ctypes_array(t):
          return np.dtype(dtype_from_ctypes_type(t._type_), (t._length_,))
      def _from_ctypes_structure(t):
          for item in t._fields_:
              if len(item) > 2:
                  raise TypeError(
                      "ctypes bitfields have no dtype equivalent")
          if hasattr(t, "_pack_"):
              import ctypes
              formats = []
              offsets = []
              names = []
              current_offset = 0
              for fname, ftyp in t._fields_:
                  names.append(fname)
                  formats.append(dtype_from_ctypes_type(ftyp))
                  effective_pack = min(t._pack_, ctypes.alignment(ftyp))
                  current_offset = ((current_offset + effective_pack - 1) //
effective_pack) * effective_pack
                  offsets.append(current_offset)
                  current_offset += ctypes.sizeof(ftyp)
              return np.dtype(dict(
                  formats=formats,
                  offsets=offsets,
                  names=names,
                  itemsize=ctypes.sizeof(t)))
          else:
              fields = []
              for fname, ftyp in t._fields_:
                  fields.append((fname, dtype_from_ctypes_type(ftyp)))
              return np.dtype(fields, align=True)
      def _from_ctypes_scalar(t):
          """
          Return the dtype type with endianness included if it's the case
          """
          if getattr(t, '__ctype_be__', None) is t:
              return np.dtype('>' + t._type_)
          elif getattr(t, '__ctype_le__', None) is t:
              return np.dtype('<' + t._type_)
          else:
              return np.dtype(t._type_)
      def _from_ctypes_union(t):
          import ctypes
          formats = []
          offsets = []
          names = []
          for fname, ftyp in t._fields_:
              names.append(fname)
              formats.append(dtype_from_ctypes_type(ftyp))
              offsets.append(0) # Union fields are offset to 0
          return np.dtype(dict(
              formats=formats,
              offsets=offsets,
              names=names,
              itemsize=ctypes.sizeof(t)))
      def dtype_from_ctypes_type(t):
          """
          Construct a dtype object from a ctypes type
          """
          import _ctypes
          if issubclass(t, _ctypes.Array):
              return _from_ctypes_array(t)
          elif issubclass(t, _ctypes._Pointer):
              raise TypeError("ctypes pointers have no dtype equivalent")
          elif issubclass(t, _ctypes.Structure):
              return _from_ctypes_structure(t)
          elif issubclass(t, _ctypes.Union):

```

```

83: (8)             return _from_ctypes_union(t)
84: (4)         elif isinstance(getattr(t, '_type_', None), str):
85: (8)             return _from_ctypes_scalar(t)
86: (4)         else:
87: (8)             raise NotImplementedError(
88: (12)                 "Unknown ctypes type {}".format(t.__name__))
-----
```

## File 65 - \_exceptions.py:

```

1: (0) """
2: (0)     Various richly-typed exceptions, that also help us deal with string formatting
3: (0)     in python where it's easier.
4: (0)     By putting the formatting in `__str__`, we also avoid paying the cost for
5: (0)     users who silence the exceptions.
6: (0) """
7: (0)     from .._utils import set_module
8: (0)     def _unpack_tuple(tup):
9: (4)         if len(tup) == 1:
10: (8)             return tup[0]
11: (4)         else:
12: (8)             return tup
13: (0)     def _display_as_base(cls):
14: (4) """
15: (4)         A decorator that makes an exception class look like its base.
16: (4)         We use this to hide subclasses that are implementation details - the user
17: (4)         should catch the base type, which is what the traceback will show them.
18: (4)         Classes decorated with this decorator are subject to removal without a
19: (4)         deprecation warning.
20: (4) """
21: (4)         assert issubclass(cls, Exception)
22: (4)         cls.__name__ = cls.__base__.__name__
23: (4)         return cls
24: (0)     class UFuncTypeError(TypeError):
25: (4)         """ Base class for all ufunc exceptions """
26: (4)         def __init__(self, ufunc):
27: (8)             self.ufunc = ufunc
28: (0)         @_display_as_base
29: (0)     class _UFuncNoLoopError(UFuncTypeError):
30: (4)         """ Thrown when a ufunc loop cannot be found """
31: (4)         def __init__(self, ufunc, dtypes):
32: (8)             super().__init__(ufunc)
33: (8)             self.dtypes = tuple(dtypes)
34: (4)         def __str__(self):
35: (8)             return (
36: (12)                 "ufunc {!r} did not contain a loop with signature matching types "
37: (12)                 "{!r} -> {!r}"
38: (8)             ).format(
39: (12)                 self.ufunc.__name__,
40: (12)                 _unpack_tuple(self.dtypes[:self.ufunc.nin]),
41: (12)                 _unpack_tuple(self.dtypes[self.ufunc.nin:]))
42: (8)         )
43: (0)         @_display_as_base
44: (0)     class _UFuncBinaryResolutionError(_UFuncNoLoopError):
45: (4)         """ Thrown when a binary resolution fails """
46: (4)         def __init__(self, ufunc, dtypes):
47: (8)             super().__init__(ufunc, dtypes)
48: (8)             assert len(self.dtypes) == 2
49: (4)         def __str__(self):
50: (8)             return (
51: (12)                 "ufunc {!r} cannot use operands with types {!r} and {!r}"
52: (8)             ).format(
53: (12)                 self.ufunc.__name__, *self.dtypes
54: (8)             )
55: (0)         @_display_as_base
56: (0)     class _UFuncCastingError(UFuncTypeError):
57: (4)         def __init__(self, ufunc, casting, from_, to):
58: (8)             super().__init__(ufunc)
```

```

59: (8)                     self.casting = casting
60: (8)                     self.from_ = from_
61: (8)                     self.to = to
62: (0) @display_as_base
63: (0) class _UFuncInputCastingError(_UFuncCastingError):
64: (4)     """Thrown when a ufunc input cannot be casted"""
65: (4)     def __init__(self, ufunc, casting, from_, to, i):
66: (8)         super().__init__(ufunc, casting, from_, to)
67: (8)         self.in_i = i
68: (4)     def __str__(self):
69: (8)         i_str = "{}".format(self.in_i) if self.ufunc.nin != 1 else ""
70: (8)         return (
71: (12)             "Cannot cast ufunc {!r} input {}from {!r} to {!r} with casting "
72: (12)             "rule {!r}"
73: (8)         ).format(
74: (12)             self.ufunc.__name__, i_str, self.from_, self.to, self.casting
75: (8)         )
76: (0) @display_as_base
77: (0) class _UFuncOutputCastingError(_UFuncCastingError):
78: (4)     """Thrown when a ufunc output cannot be casted"""
79: (4)     def __init__(self, ufunc, casting, from_, to, i):
80: (8)         super().__init__(ufunc, casting, from_, to)
81: (8)         self.out_i = i
82: (4)     def __str__(self):
83: (8)         i_str = "{}".format(self.out_i) if self.ufunc.nout != 1 else ""
84: (8)         return (
85: (12)             "Cannot cast ufunc {!r} output {}from {!r} to {!r} with casting "
86: (12)             "rule {!r}"
87: (8)         ).format(
88: (12)             self.ufunc.__name__, i_str, self.from_, self.to, self.casting
89: (8)         )
90: (0) @display_as_base
91: (0) class _ArrayMemoryError(MemoryError):
92: (4)     """Thrown when an array cannot be allocated"""
93: (4)     def __init__(self, shape, dtype):
94: (8)         self.shape = shape
95: (8)         self.dtype = dtype
96: (4)     @property
97: (4)     def _total_size(self):
98: (8)         num_bytes = self.dtype.itemsize
99: (8)         for dim in self.shape:
100: (12)             num_bytes *= dim
101: (8)         return num_bytes
102: (4)     @staticmethod
103: (4)     def _size_to_string(num_bytes):
104: (8)         """Convert a number of bytes into a binary size string"""
105: (8)         LOG2_STEP = 10
106: (8)         STEP = 1024
107: (8)         units = ['bytes', 'KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB']
108: (8)         unit_i = max(num_bytes.bit_length() - 1, 1) // LOG2_STEP
109: (8)         unit_val = 1 << (unit_i * LOG2_STEP)
110: (8)         n_units = num_bytes / unit_val
111: (8)         del unit_val
112: (8)         if round(n_units) == STEP:
113: (12)             unit_i += 1
114: (12)             n_units /= STEP
115: (8)         if unit_i >= len(units):
116: (12)             new_unit_i = len(units) - 1
117: (12)             n_units *= 1 << ((unit_i - new_unit_i) * LOG2_STEP)
118: (12)             unit_i = new_unit_i
119: (8)         unit_name = units[unit_i]
120: (8)         if unit_i == 0:
121: (12)             return '{:.0f} {}'.format(n_units, unit_name)
122: (8)         elif round(n_units) < 1000:
123: (12)             return '{:#.3g} {}'.format(n_units, unit_name)
124: (8)         else:
125: (12)             return '{:#.0f} {}'.format(n_units, unit_name)
126: (4)     def __str__(self):
127: (8)         size_str = self._size_to_string(self._total_size)

```

```

128: (8)             return (
129: (12)           "Unable to allocate {} for an array with shape {} and data type
{}"
130: (12)           .format(size_str, self.shape, self.dtype)
131: (8)         )
-----
```

## File 66 - \_internal.py:

```

1: (0)      """
2: (0)      A place for internal code
3: (0)      Some things are more easily handled Python.
4: (0)      """
5: (0)      import ast
6: (0)      import re
7: (0)      import sys
8: (0)      import warnings
9: (0)      from ..exceptions import DtypePromotionError
10: (0)     from .multiarray import dtype, array, ndarray, promote_types
11: (0)     try:
12: (4)       import ctypes
13: (0)     except ImportError:
14: (4)       ctypes = None
15: (0)     IS_PYPY = sys.implementation.name == 'pypy'
16: (0)     if sys.byteorder == 'little':
17: (4)       _nbo = '<'
18: (0)     else:
19: (4)       _nbo = '>'
20: (0)     def _makenames_list(adict, align):
21: (4)       allfields = []
22: (4)       for fname, obj in adict.items():
23: (8)         n = len(obj)
24: (8)         if not isinstance(obj, tuple) or n not in (2, 3):
25: (12)           raise ValueError("entry not a 2- or 3- tuple")
26: (8)         if n > 2 and obj[2] == fname:
27: (12)           continue
28: (8)         num = int(obj[1])
29: (8)         if num < 0:
30: (12)           raise ValueError("invalid offset.")
31: (8)         format = dtype(obj[0], align=align)
32: (8)         if n > 2:
33: (12)           title = obj[2]
34: (8)         else:
35: (12)           title = None
36: (8)         allfields.append((fname, format, num, title))
37: (4)       allfields.sort(key=lambda x: x[2])
38: (4)       names = [x[0] for x in allfields]
39: (4)       formats = [x[1] for x in allfields]
40: (4)       offsets = [x[2] for x in allfields]
41: (4)       titles = [x[3] for x in allfields]
42: (4)       return names, formats, offsets, titles
43: (0)     def _usefields(adict, align):
44: (4)       try:
45: (8)         names = adict[-1]
46: (4)       except KeyError:
47: (8)         names = None
48: (4)       if names is None:
49: (8)         names, formats, offsets, titles = _makenames_list(adict, align)
50: (4)       else:
51: (8)         formats = []
52: (8)         offsets = []
53: (8)         titles = []
54: (8)         for name in names:
55: (12)           res = adict[name]
56: (12)           formats.append(res[0])
57: (12)           offsets.append(res[1])
58: (12)           if len(res) > 2:
59: (16)             titles.append(res[2])
```

```

60: (12)           else:
61: (16)             titles.append(None)
62: (4)           return dtype({"names": names,
63: (18)               "formats": formats,
64: (18)               "offsets": offsets,
65: (18)               "titles": titles}, align)
66: (0)           def _array_descr(descriptor):
67: (4)             fields = descriptor.fields
68: (4)             if fields is None:
69: (8)               subtype = descriptor.subdtype
70: (8)               if subtype is None:
71: (12)                 if descriptor.metadata is None:
72: (16)                   return descriptor.str
73: (12)               else:
74: (16)                 new = descriptor.metadata.copy()
75: (16)                 if new:
76: (20)                   return (descriptor.str, new)
77: (16)                 else:
78: (20)                   return descriptor.str
79: (8)             else:
80: (12)               return (_array_descr(subdtype[0]), subtype[1])
81: (4)           names = descriptor.names
82: (4)           ordered_fields = [fields[x] + (x,) for x in names]
83: (4)           result = []
84: (4)           offset = 0
85: (4)           for field in ordered_fields:
86: (8)             if field[1] > offset:
87: (12)               num = field[1] - offset
88: (12)               result.append(('', f'|V{num}'))
89: (12)               offset += num
90: (8)             elif field[1] < offset:
91: (12)               raise ValueError(
92: (16)                 "dtype.descr is not defined for types with overlapping or "
93: (16)                 "out-of-order fields")
94: (8)           if len(field) > 3:
95: (12)             name = (field[2], field[3])
96: (8)           else:
97: (12)             name = field[2]
98: (8)           if field[0].subdtype:
99: (12)             tup = (name, _array_descr(field[0].subdtype[0]),
100: (19)                           field[0].subdtype[1])
101: (8)           else:
102: (12)             tup = (name, _array_descr(field[0]))
103: (8)             offset += field[0].itemsize
104: (8)             result.append(tup)
105: (4)           if descriptor.itemsize > offset:
106: (8)             num = descriptor.itemsize - offset
107: (8)             result.append(('', f'|V{num}'))
108: (4)           return result
109: (0)           def _reconstruct(subtype, shape, dtype):
110: (4)             return ndarray.__new__(subtype, shape, dtype)
111: (0)           format_re = re.compile(r'(?P<order1>[<>|=]?')
112: (23)             r'(?P<repeats> *[()?[ ,0-9]*[]]? *)'
113: (23)             r'(?P<order2>[<>|=]?')
114: (23)             r'(?P<dtype>[A-Za-z0-9.?]*(?:\|[a-zA-Z0-9,.]+\])?)')
115: (0)           sep_re = re.compile(r'\s*,\s*')
116: (0)           space_re = re.compile(r'\s+$')
117: (0)           _convorder = {'=': _nbo}
118: (0)           def _commastring(astr):
119: (4)             startindex = 0
120: (4)             result = []
121: (4)             while startindex < len(astr):
122: (8)               mo = format_re.match(astr, pos=startindex)
123: (8)               try:
124: (12)                 (order1, repeats, order2, dtype) = mo.groups()
125: (8)               except (TypeError, AttributeError):
126: (12)                 raise ValueError(
127: (16)                   f'format number {len(result)+1} of "{astr}" is not recognized'
128: (16)                   ) from None

```

```

129: (8)             startindex = mo.end()
130: (8)             if startindex < len(astr):
131: (12)             if space_re.match(astr, pos=startindex):
132: (16)                 startindex = len(astr)
133: (12)             else:
134: (16)                 mo = sep_re.match(astr, pos=startindex)
135: (16)                 if not mo:
136: (20)                     raise ValueError(
137: (24)                         'format number %d of "%s" is not recognized' %
138: (24)                         (len(result)+1, astr))
139: (16)                     startindex = mo.end()
140: (8)             if order2 == '':
141: (12)                 order = order1
142: (8)             elif order1 == '':
143: (12)                 order = order2
144: (8)             else:
145: (12)                 order1 = _convorder.get(order1, order1)
146: (12)                 order2 = _convorder.get(order2, order2)
147: (12)                 if (order1 != order2):
148: (16)                     raise ValueError(
149: (20)                         'inconsistent byte-order specification %s and %s' %
150: (20)                         (order1, order2))
151: (12)                 order = order1
152: (8)             if order in ('|', '=', _nbo):
153: (12)                 order = ''
154: (8)             dtype = order + dtype
155: (8)             if (repeats == ''):
156: (12)                 newitem = dtype
157: (8)             else:
158: (12)                 newitem = (dtype, ast.literal_eval(repeats))
159: (8)             result.append(newitem)
160: (4)             return result
161: (0)             class dummy_ctype:
162: (4)                 def __init__(self, cls):
163: (8)                     self._cls = cls
164: (4)                 def __mul__(self, other):
165: (8)                     return self
166: (4)                 def __call__(self, *other):
167: (8)                     return self._cls(other)
168: (4)                 def __eq__(self, other):
169: (8)                     return self._cls == other._cls
170: (4)                 def __ne__(self, other):
171: (8)                     return self._cls != other._cls
172: (0)             def _getintp_ctype():
173: (4)                 val = _getintp_ctype.cache
174: (4)                 if val is not None:
175: (8)                     return val
176: (4)                 if ctypes is None:
177: (8)                     import numpy as np
178: (8)                     val = dummy_ctype(np.intp)
179: (4)                 else:
180: (8)                     char = dtype('p').char
181: (8)                     if char == 'i':
182: (12)                         val = ctypes.c_int
183: (8)                     elif char == 'l':
184: (12)                         val = ctypes.c_long
185: (8)                     elif char == 'q':
186: (12)                         val = ctypes.c_longlong
187: (8)                     else:
188: (12)                         val = ctypes.c_long
189: (4)                     _getintp_ctype.cache = val
190: (4)             return val
191: (0)             _getintp_ctype.cache = None
192: (0)             class _missing_ctypes:
193: (4)                 def cast(self, num, obj):
194: (8)                     return num.value
195: (4)                 class c_void_p:
196: (8)                     def __init__(self, ptr):
197: (12)                         self.value = ptr

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

198: (0)
199: (4)
200: (8)
201: (8)
202: (12)
203: (12)
204: (8)
205: (12)
206: (12)
207: (12)
208: (8)
209: (12)
210: (8)
211: (12)
212: (4)
213: (8)
214: (8)
215: (8)
216: (8)
a
217: (8)
218: (8)
219: (8)
220: (8)
221: (8)
222: (8)
223: (8)
224: (4)
225: (8)
226: (8)
227: (8)
228: (8)
229: (8)
230: (12)
231: (8)
232: (4)
233: (8)
234: (8)
235: (8)
236: (8)
237: (8)
238: (12)
239: (8)
240: (4)
241: (4)
242: (8)
243: (8)
244: (8)
correct
245: (8)
246: (8)
this
247: (8)
248: (8)
249: (8)
250: (8)
array:
251: (8)
252: (8)
253: (8)
254: (8)
255: (8)
256: (4)
257: (4)
258: (8)
259: (8)
260: (8)
261: (8)
262: (8)

class _ctypes:
    def __init__(self, array, ptr=None):
        self._arr = array
        if ctypes:
            self._ctypes = ctypes
            self._data = self._ctypes.c_void_p(ptr)
        else:
            self._ctypes = _missing_ctypes()
            self._data = self._ctypes.c_void_p(ptr)
            self._data._objects = array
        if self._arr.ndim == 0:
            self._zerod = True
        else:
            self._zerod = False
    def data_as(self, obj):
        """
        Return the data pointer cast to a particular c-types object.
        For example, calling ``self._as_parameter_`` is equivalent to
        ``self.data_as(ctypes.c_void_p)``. Perhaps you want to use the data as
        a
        pointer to a ctypes array of floating-point data:
        ``self.data_as(ctypes.POINTER(ctypes.c_double))``.
        The returned pointer will keep a reference to the array.
        """
        ptr = self._ctypes.cast(self._data, obj)
        ptr._arr = self._arr
        return ptr
    def shape_as(self, obj):
        """
        Return the shape tuple as an array of some other c-types
        type. For example: ``self.shape_as(ctypes.c_short)``.
        """
        if self._zerod:
            return None
        return (obj*self._arr.ndim)(*self._arr.shape)
    def strides_as(self, obj):
        """
        Return the strides tuple as an array of some other
        c-types type. For example: ``self.strides_as(ctypes.c_longlong)``.
        """
        if self._zerod:
            return None
        return (obj*self._arr.ndim)(*self._arr.strides)
@property
def data(self):
    """
    A pointer to the memory area of the array as a Python integer.
    This memory area may contain data that is not aligned, or not in
    byte-order. The memory area may not even be writeable. The array
    flags and data-type of this array should be respected when passing
    attribute to arbitrary C-code to avoid trouble that can include Python
    crashing. User Beware! The value of this attribute is exactly the same
    as ``self._array_interface_['data'][0]``.
    Note that unlike ``data_as``, a reference will not be kept to the
    code like ``ctypes.c_void_p((a + b).ctypes.data)`` will result in a
    pointer to a deallocated array, and should be spelt
    ``(a + b).ctypes.data_as(ctypes.c_void_p)``
    """
    return self._data.value
@property
def shape(self):
    """
    (c_intp*self.ndim): A ctypes array of length self.ndim where
    the basetype is the C-integer corresponding to ``dtype('p')`` on this
    platform (see `~numpy.ctypeslib.c_intp`). This base-type could be
    `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

263: (8)           the platform. The ctypes array contains the shape of
264: (8)           the underlying array.
265: (8)
266: (8)           return self.shape_as(_getintp_ctype())
267: (4) @property
268: (4) def strides(self):
269: (8) """
270: (8)   (c_intp*self.ndim): A ctypes array of length self.ndim where
271: (8)   the basetype is the same as for the shape attribute. This ctypes array
272: (8)   contains the strides information from the underlying array. This
strides
273: (8)   information is important for showing how many bytes must be jumped to
274: (8)   get to the next element in the array.
275: (8) """
276: (8)           return self.strides_as(_getintp_ctype())
277: (4) @property
278: (4) def _as_parameter_(self):
279: (8) """
280: (8)   Overrides the ctypes semi-magic method
281: (8)   Enables `c_func(some_array.ctypes)`
282: (8) """
283: (8)           return self.data_as(ctypes.c_void_p)
def get_data(self):
285: (8) """
286: (8)   Deprecated getter for the `_ctypes.data` property.
287: (8)   .. deprecated:: 1.21
288: (8) """
289: (22)   warnings.warn("get_data" is deprecated. Use "data" instead',
290: (8)                               DeprecationWarning, stacklevel=2)
291: (4)           return self.data
def get_shape(self):
292: (8) """
293: (8)   Deprecated getter for the `_ctypes.shape` property.
294: (8)   .. deprecated:: 1.21
295: (8) """
296: (22)   warnings.warn("get_shape" is deprecated. Use "shape" instead',
297: (8)                               DeprecationWarning, stacklevel=2)
298: (4)           return self.shape
def get_strides(self):
299: (8) """
300: (8)   Deprecated getter for the `_ctypes.strides` property.
301: (8)   .. deprecated:: 1.21
302: (8) """
303: (22)   warnings.warn("get_strides" is deprecated. Use "strides" instead',
304: (8)                               DeprecationWarning, stacklevel=2)
305: (4)           return self.strides
def get_as_parameter(self):
306: (8) """
307: (8)   Deprecated getter for the `_ctypes._as_parameter_` property.
308: (8)   .. deprecated:: 1.21
309: (8) """
310: (12)   warnings.warn(
311: (12)     "'get_as_parameter' is deprecated. Use '_as_parameter_' instead',
312: (8)     DeprecationWarning, stacklevel=2,
313: (8)   )
314: (0)           return self._as_parameter_
def _newnames(datatype, order):
315: (4) """
316: (4)   Given a datatype and an order object, return a new names tuple, with the
317: (4)   order indicated
318: (4) """
319: (4)   oldnames = datatype.names
320: (4)   nameslist = list(oldnames)
321: (4)   if isinstance(order, str):
322: (8)     order = [order]
323: (4)   seen = set()
324: (4)   if isinstance(order, (list, tuple)):
325: (8)     for name in order:
326: (12)       try:
327: (16)         nameslist.remove(name)
328: (12)     except ValueError:
329: (16)       if name in seen:
330: (20)         raise ValueError(f"duplicate field name: {name}") from

```

```

None

331: (16)           else:
332: (20)             raise ValueError(f"unknown field name: {name}") from None
333: (12)             seen.add(name)
334: (8)             return tuple(list(order) + nameslist)
335: (4)             raise ValueError(f"unsupported order value: {order}")
336: (0) def _copy_fields(ary):
337: (4)     """Return copy of structured array with padding between fields removed.
338: (4)     Parameters
339: (4)     -----
340: (4)     ary : ndarray
341: (7)         Structured array from which to remove padding bytes
342: (4)     Returns
343: (4)     -----
344: (4)     ary_copy : ndarray
345: (7)         Copy of ary with padding bytes removed
346: (4)     """
347: (4)     dt = ary.dtype
348: (4)     copy_dtype = {'names': dt.names,
349: (18)                 'formats': [dt.fields[name][0] for name in dt.names]}
350: (4)     return array(ary, dtype=copy_dtype, copy=True)
351: (0) def _promote_fields(dt1, dt2):
352: (4)     """ Perform type promotion for two structured dtypes.
353: (4)     Parameters
354: (4)     -----
355: (4)     dt1 : structured dtype
356: (8)         First dtype.
357: (4)     dt2 : structured dtype
358: (8)         Second dtype.
359: (4)     Returns
360: (4)     -----
361: (4)     out : dtype
362: (8)         The promoted dtype
363: (4)     Notes
364: (4)     -----
365: (4)     If one of the inputs is aligned, the result will be. The titles of
366: (4)     both descriptors must match (point to the same field).
367: (4)     """
368: (4)     if (dt1.names is None or dt2.names is None) or dt1.names != dt2.names:
369: (8)         raise DtypePromotionError(
370: (16)             f"field names `'{dt1.names}` and `'{dt2.names}` mismatch.")
371: (4)     identical = dt1 is dt2
372: (4)     new_fields = []
373: (4)     for name in dt1.names:
374: (8)         field1 = dt1.fields[name]
375: (8)         field2 = dt2.fields[name]
376: (8)         new_descr = promote_types(field1[0], field2[0])
377: (8)         identical = identical and new_descr is field1[0]
378: (8)         if field1[2:] != field2[2:]:
379: (12)             raise DtypePromotionError(
380: (20)                 f"field titles of field '{name}' mismatch")
381: (8)         if len(field1) == 2:
382: (12)             new_fields.append((name, new_descr))
383: (8)         else:
384: (12)             new_fields.append(((field1[2], name), new_descr))
385: (4)     res = dtype(new_fields, align=dt1.isalignedstruct or dt2.isalignedstruct)
386: (4)     if identical and res.itemsize == dt1.itemsize:
387: (8)         for name in dt1.names:
388: (12)             if dt1.fields[name][1] != res.fields[name][1]:
389: (16)                 return res # the dtype changed.
390: (8)             return dt1
391: (4)     return res
392: (0) def _getfield_is_safe(oldtype, newtype, offset):
393: (4)     """ Checks safety of getfield for object arrays.
394: (4)     As in _view_is_safe, we need to check that memory containing objects is
not
395: (4)     reinterpreted as a non-object datatype and vice versa.
396: (4)     Parameters
397: (4)     -----

```

```

398: (4)          oldtype : data-type
399: (8)          Data type of the original ndarray.
400: (4)          newtype : data-type
401: (8)          Data type of the field being accessed by ndarray.getfield
402: (4)          offset : int
403: (8)          Offset of the field being accessed by ndarray.getfield
404: (4)          Raises
405: (4)          -----
406: (4)          TypeError
407: (8)          If the field access is invalid
408: (4)
409: (4)          if newtype.hasobject or oldtype.hasobject:
410: (8)              if offset == 0 and newtype == oldtype:
411: (12)                  return
412: (8)              if oldtype.names is not None:
413: (12)                  for name in oldtype.names:
414: (16)                      if (oldtype.fields[name][1] == offset and
415: (24)                          oldtype.fields[name][0] == newtype):
416: (20)                          return
417: (8)                  raise TypeError("Cannot get/set field of an object array")
418: (4)          return
419: (0)          def _view_is_safe(oldtype, newtype):
420: (4)              """ Checks safety of a view involving object arrays, for example when
421: (4)              doing::
422: (8)                  np.zeros(10, dtype=oldtype).view(newtype)
423: (4)          Parameters
424: (4)          -----
425: (4)          oldtype : data-type
426: (8)          Data type of original ndarray
427: (4)          newtype : data-type
428: (8)          Data type of the view
429: (4)          Raises
430: (4)          -----
431: (4)          TypeError
432: (8)          If the new type is incompatible with the old type.
433: (4)
434: (4)          if oldtype == newtype:
435: (8)              return
436: (4)          if newtype.hasobject or oldtype.hasobject:
437: (8)              raise TypeError("Cannot change data-type for object array.")
438: (4)          return
439: (0)          _pep3118_native_map = {
440: (4)              '?': '?',
441: (4)              'c': 'S1',
442: (4)              'b': 'b',
443: (4)              'B': 'B',
444: (4)              'h': 'h',
445: (4)              'H': 'H',
446: (4)              'i': 'i',
447: (4)              'I': 'I',
448: (4)              'l': 'l',
449: (4)              'L': 'L',
450: (4)              'q': 'q',
451: (4)              'Q': 'Q',
452: (4)              'e': 'e',
453: (4)              'f': 'f',
454: (4)              'd': 'd',
455: (4)              'g': 'g',
456: (4)              'Zf': 'F',
457: (4)              'Zd': 'D',
458: (4)              'Zg': 'G',
459: (4)              's': 'S',
460: (4)              'w': 'U',
461: (4)              'o': 'O',
462: (4)              'x': 'V',  # padding
463: (0)          }
464: (0)          _pep3118_native_typechars = ''.join(_pep3118_native_map.keys())
465: (0)          _pep3118_standard_map = {
466: (4)              '?': '?',

```

```

467: (4)             'c': 'S1',
468: (4)             'b': 'b',
469: (4)             'B': 'B',
470: (4)             'h': 'i2',
471: (4)             'H': 'u2',
472: (4)             'i': 'i4',
473: (4)             'I': 'u4',
474: (4)             'l': 'i4',
475: (4)             'L': 'u4',
476: (4)             'q': 'i8',
477: (4)             'Q': 'u8',
478: (4)             'e': 'f2',
479: (4)             'f': 'f',
480: (4)             'd': 'd',
481: (4)             'Zf': 'F',
482: (4)             'Zd': 'D',
483: (4)             's': 'S',
484: (4)             'w': 'U',
485: (4)             'O': 'O',
486: (4)             'x': 'V', # padding
487: (0)
488: (0) _pep3118_standard_typechars = ''.join(_pep3118_standard_map.keys())
489: (0) _pep3118_unsupported_map = {
490: (4)     'u': 'UCS-2 strings',
491: (4)     '&': 'pointers',
492: (4)     't': 'bitfields',
493: (4)     'X': 'function pointers',
494: (0)
495: (0) class _Stream:
496: (4)     def __init__(self, s):
497: (8)         self.s = s
498: (8)         self.byteorder = '@'
499: (4)     def advance(self, n):
500: (8)         res = self.s[:n]
501: (8)         self.s = self.s[n:]
502: (8)         return res
503: (4)     def consume(self, c):
504: (8)         if self.s[:len(c)] == c:
505: (12)             self.advance(len(c))
506: (12)             return True
507: (8)         return False
508: (4)     def consume_until(self, c):
509: (8)         if callable(c):
510: (12)             i = 0
511: (12)             while i < len(self.s) and not c(self.s[i]):
512: (16)                 i = i + 1
513: (12)             return self.advance(i)
514: (8)         else:
515: (12)             i = self.s.index(c)
516: (12)             res = self.advance(i)
517: (12)             self.advance(len(c))
518: (12)             return res
519: (4)     @property
520: (4)     def next(self):
521: (8)         return self.s[0]
522: (4)     def __bool__(self):
523: (8)         return bool(self.s)
524: (0)     def _dtype_from_pep3118(spec):
525: (4)         stream = _Stream(spec)
526: (4)         dtype, align = _dtype_from_pep3118(stream, is_subdtype=False)
527: (4)         return dtype
528: (0)     def _dtype_from_pep3118(stream, is_subdtype):
529: (4)         field_spec = dict(
530: (8)             names=[],
531: (8)             formats=[],
532: (8)             offsets=[],
533: (8)             itemsize=0
534: (4)         )
535: (4)         offset = 0

```

```

536: (4)           common_alignment = 1
537: (4)           is_padding = False
538: (4)           while stream:
539: (8)             value = None
540: (8)             if stream.consume('}'):
541: (12)               break
542: (8)             shape = None
543: (8)             if stream.consume('('):
544: (12)               shape = stream.consume_until(')')
545: (12)               shape = tuple(map(int, shape.split(',')))
546: (8)             if stream.next in ('@', '=', '<', '>', '^', '!'):
547: (12)               byteorder = stream.advance(1)
548: (12)               if byteorder == '!':
549: (16)                 byteorder = '>'
550: (12)                 stream.byteorder = byteorder
551: (8)             if stream.byteorder in ('@', '^'):
552: (12)               type_map = _pep3118_native_map
553: (12)               type_map_chars = _pep3118_native_typechars
554: (8)
555: (12)             else:
556: (12)               type_map = _pep3118_standard_map
557: (8)               type_map_chars = _pep3118_standard_typechars
558: (8)
559: (12)             itemsize_str = stream.consume_until(lambda c: not c.isdigit())
560: (8)
561: (12)             if itemsize_str:
562: (16)               itemsize = int(itemsize_str)
563: (8)
564: (12)             else:
565: (16)               itemsize = 1
566: (8)
567: (12)             is_padding = False
568: (8)
569: (12)             if stream.consume('T{'):
570: (16)               value, align = __dtype_from_pep3118(
571: (16)                 stream, is_subtype=True)
572: (8)
573: (12)             elif stream.next in type_map_chars:
574: (12)               if stream.next == 'Z':
575: (16)                 typechar = stream.advance(2)
576: (8)
577: (16)               else:
578: (12)                 typechar = stream.advance(1)
579: (12)               is_padding = (typechar == 'x')
580: (8)
581: (12)               dtypechar = type_map[typechar]
582: (12)               if dtypechar in 'USV':
583: (16)                 dtypechar += '%d' % itemsize
584: (16)                 itemsize = 1
585: (8)
586: (12)               numpy_byteorder = {'@': '=', '^': '='}.get(
587: (8)                 stream.byteorder, stream.byteorder)
588: (8)               value = dtype(numpy_byteorder + dtypechar)
589: (12)               align = value.alignment
590: (12)
591: (12)             elif stream.next in _pep3118_unsupported_map:
592: (12)               desc = _pep3118_unsupported_map[stream.next]
593: (16)               raise NotImplementedError(
594: (20)                 "Unrepresentable PEP 3118 data type {!r} ({}))"
595: (16)                 .format(stream.next, desc))
596: (20)
597: (12)
598: (8)
599: (12)
600: (8)
601: (12)
602: (8)
603: (12)             else:
604: (16)               raise ValueError("Unknown PEP 3118 data type specifier %r" %
605: (8)                 extra_offset = 0
606: (8)               if stream.byteorder == '@':
607: (12)                 start_padding = (-offset) % align
608: (12)                 intra_padding = (-value.itemsize) % align
609: (12)                 offset += start_padding
610: (12)                 if intra_padding != 0:
611: (16)                   if itemsize > 1 or (shape is not None and _prod(shape) > 1):
612: (20)                     value = _add_trailing_padding(value, intra_padding)
613: (16)                   else:
614: (20)                     extra_offset += intra_padding
615: (12)                   common_alignment = _lcm(alignment, common_alignment)
616: (8)
617: (12)               if itemsize != 1:
618: (16)                 value = dtype((value, (itemsize,)))
619: (8)
620: (12)               if shape is not None:
621: (16)                 value = dtype((value, shape))
622: (8)
623: (12)               if stream.consume(':'):
624: (16)                 name = stream.consume_until(':')

```

```

604: (8)           else:
605: (12)          name = None
606: (8)          if not (is_padding and name is None):
607: (12)            if name is not None and name in field_spec['names']:
608: (16)              raise RuntimeError(f"Duplicate field name '{name}' in PEP3118
format")
609: (12)            field_spec['names'].append(name)
610: (12)            field_spec['formats'].append(value)
611: (12)            field_spec['offsets'].append(offset)
612: (8)              offset += value.itemsize
613: (8)              offset += extra_offset
614: (8)              field_spec['itemsize'] = offset
615: (4)            if stream.byteorder == '@':
616: (8)              field_spec['itemsize'] += (-offset) % common_alignment
617: (4)            if (field_spec['names'] == [None]
618: (12)              and field_spec['offsets'][0] == 0
619: (12)              and field_spec['itemsize'] == field_spec['formats'][0].itemsize
620: (12)              and not is_subtype):
621: (8)                ret = field_spec['formats'][0]
622: (4)            else:
623: (8)              _fix_names(field_spec)
624: (8)              ret = dtype(field_spec)
625: (4)            return ret, common_alignment
626: (0)          def _fix_names(field_spec):
627: (4)            """ Replace names which are None with the next unused f%d name """
628: (4)            names = field_spec['names']
629: (4)            for i, name in enumerate(names):
630: (8)              if name is not None:
631: (12)                continue
632: (8)              j = 0
633: (8)              while True:
634: (12)                name = f'f{j}'
635: (12)                if name not in names:
636: (16)                  break
637: (12)                j = j + 1
638: (8)                names[i] = name
639: (0)          def _add_trailing_padding(value, padding):
640: (4)            """Inject the specified number of padding bytes at the end of a dtype"""
641: (4)            if value.fields is None:
642: (8)              field_spec = dict(
643: (12)                names=['f0'],
644: (12)                formats=[value],
645: (12)                offsets=[0],
646: (12)                itemsize=value.itemsize
647: (8)              )
648: (4)            else:
649: (8)              fields = value.fields
650: (8)              names = value.names
651: (8)              field_spec = dict(
652: (12)                names=names,
653: (12)                formats=[fields[name][0] for name in names],
654: (12)                offsets=[fields[name][1] for name in names],
655: (12)                itemsize=value.itemsize
656: (8)              )
657: (4)              field_spec['itemsize'] += padding
658: (4)            return dtype(field_spec)
659: (0)          def _prod(a):
660: (4)            p = 1
661: (4)            for x in a:
662: (8)              p *= x
663: (4)            return p
664: (0)          def _gcd(a, b):
665: (4)            """Calculate the greatest common divisor of a and b"""
666: (4)            while b:
667: (8)              a, b = b, a % b
668: (4)            return a
669: (0)          def _lcm(a, b):
670: (4)            return a // _gcd(a, b) * b
671: (0)          def array_ufunc_errmsg_formatter(dummy, ufunc, method, *inputs, **kwargs):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

672: (4)         """ Format the error message for when __array_ufunc__ gives up. """
673: (4)         args_string = ', '.join(['{!r}'.format(arg) for arg in inputs] +
674: (28)           ['{}={!r}'.format(k, v)
675: (29)             for k, v in kwargs.items()])
676: (4)         args = inputs + kwargs.get('out', ())
677: (4)         types_string = ', '.join(repr(type(arg).__name__) for arg in args)
678: (4)         return ('operand type(s) all returned NotImplemented from '
679: (12)           '__array_ufunc__( {!r}, {!r}, {}): {}'
680: (12)             .format(ufunc, method, args_string, types_string))
681: (0)     def array_function_errmsg_formatter(public_api, types):
682: (4)         """ Format the error message for when __array_ufunc__ gives up. """
683: (4)         func_name = '{}.{}'.format(public_api.__module__, public_api.__name__)
684: (4)         return ("no implementation found for '{}' on types that implement "
685: (12)           '__array_function__: {}'.format(func_name, list(types)))
686: (0)     def __ufunc_doc_signature_formatter(ufunc):
687: (4)         """
688: (4)             Builds a signature string which resembles PEP 457
689: (4)             This is used to construct the first line of the docstring
690: (4)             """
691: (4)             if ufunc.nin == 1:
692: (8)                 in_args = 'x'
693: (4)             else:
694: (8)                 in_args = ', '.join(f'x{i+1}' for i in range(ufunc.nin))
695: (4)             if ufunc.nout == 0:
696: (8)                 out_args = ', /, out=()'
697: (4)             elif ufunc.nout == 1:
698: (8)                 out_args = ', /, out=None'
699: (4)             else:
700: (8)                 out_args = '[, {positional}], / [, out={default}].format(
701: (12)                   positional=', '.join(
702: (16)                     'out{}'.format(i+1) for i in range(ufunc.nout)),
703: (12)                     default=repr((None,) * ufunc.nout)
704: (8)               )
705: (4)             kwargs = (
706: (8)                 ", casting='same_kind'"
707: (8)                 ", order='K'"
708: (8)                 ", dtype=None"
709: (8)                 ", subok=True"
710: (4)             )
711: (4)             if ufunc.signature is None:
712: (8)                 kwargs = f", where=True{kwargs}[, signature, extobj]"
713: (4)             else:
714: (8)                 kwargs += "[, signature, extobj, axes, axis]"
715: (4)             return '{name}({in_args}{out_args}, *{kwargs})'.format(
716: (8)               name=ufunc.__name__,
717: (8)               in_args=in_args,
718: (8)               out_args=out_args,
719: (8)               kwargs=kwargs
720: (4)             )
721: (0)     def npy_ctypes_check(cls):
722: (4)         try:
723: (8)             if IS_PYPY:
724: (12)                 ctype_base = cls.__mro__[-3]
725: (8)             else:
726: (12)                 ctype_base = cls.__mro__[-2]
727: (8)             return '_ctypes' in ctype_base.__module__
728: (4)         except Exception:
729: (8)             return False

```

-----  
File 67 - \_machar.py:

```

1: (0)         """
2: (0)             Machine arithmetic - determine the parameters of the
3: (0)             floating-point arithmetic system
4: (0)             Author: Pearu Peterson, September 2003
5: (0)             """
6: (0)             __all__ = ['MachAr']

```

```

7: (0)          from .fromnumeric import any
8: (0)          from ._ufunc_config import errstate
9: (0)          from .._utils import set_module
10: (0)         class MachAr:
11: (4)             """
12: (4)                 Diagnosing machine parameters.
13: (4)                 Attributes
14: (4)                 -----
15: (4)                 ibeta : int
16: (8)                     Radix in which numbers are represented.
17: (4)                 it : int
18: (8)                     Number of base-`ibeta` digits in the floating point mantissa M.
19: (4)                 machep : int
20: (8)                     Exponent of the smallest (most negative) power of `ibeta` that,
21: (8)                         added to 1.0, gives something different from 1.0
22: (4)                 eps : float
23: (8)                     Floating-point number ``beta**machep`` (floating point precision)
24: (4)                 negep : int
25: (8)                     Exponent of the smallest power of `ibeta` that, subtracted
26: (8)                         from 1.0, gives something different from 1.0.
27: (4)                 epsneg : float
28: (8)                     Floating-point number ``beta**negep``.
29: (4)                 iexp : int
30: (8)                     Number of bits in the exponent (including its sign and bias).
31: (4)                 minexp : int
32: (8)                     Smallest (most negative) power of `ibeta` consistent with there
33: (8)                         being no leading zeros in the mantissa.
34: (4)                 xmin : float
35: (8)                     Floating-point number ``beta**minexp`` (the smallest [in
36: (8)                         magnitude] positive floating point number with full precision).
37: (4)                 maxexp : int
38: (8)                     Smallest (positive) power of `ibeta` that causes overflow.
39: (4)                 xmax : float
40: (8)                     ``-(1-epsneg) * beta**maxexp`` (the largest [in magnitude]
41: (8)                         usable floating value).
42: (4)                 irnd : int
43: (8)                     In ``range(6)``, information on what kind of rounding is done
44: (8)                         in addition, and on how underflow is handled.
45: (4)                 ngrd : int
46: (8)                     Number of 'guard digits' used when truncating the product
47: (8)                         of two mantissas to fit the representation.
48: (4)                 epsilon : float
49: (8)                     Same as `eps`.
50: (4)                 tiny : float
51: (8)                     An alias for `smallest_normal`, kept for backwards compatibility.
52: (4)                 huge : float
53: (8)                     Same as `xmax`.
54: (4)                 precision : float
55: (8)                     ``- int(-log10(eps))``
56: (4)                 resolution : float
57: (8)                     ``- 10**(-precision)``
58: (4)                 smallest_normal : float
59: (8)                     The smallest positive floating point number with 1 as leading bit in
60: (8)                         the mantissa following IEEE-754. Same as `xmin`.
61: (4)                 smallest_subnormal : float
62: (8)                     The smallest positive floating point number with 0 as leading bit in
63: (8)                         the mantissa following IEEE-754.
64: (4)                 Parameters
65: (4)                 -----
66: (4)                 float_conv : function, optional
67: (8)                     Function that converts an integer or integer array to a float
68: (8)                         or float array. Default is `float`.
69: (4)                 int_conv : function, optional
70: (8)                     Function that converts a float or float array to an integer or
71: (8)                         integer array. Default is `int`.
72: (4)                 float_to_float : function, optional
73: (8)                     Function that converts a float array to float. Default is `float`.
74: (8)                     Note that this does not seem to do anything useful in the current
75: (8)                         implementation.

```

```

76: (4)          float_to_str : function, optional
77: (8)            Function that converts a single float to a string. Default is
78: (8)              ``lambda v:'%24.16e' %v``.
79: (4)          title : str, optional
80: (8)            Title that is printed in the string representation of `MachAr`.
81: (4)          See Also
82: (4)            -----
83: (4)              finfo : Machine limits for floating point types.
84: (4)              iinfo : Machine limits for integer types.
85: (4)          References
86: (4)            -----
87: (4)              .. [1] Press, Teukolsky, Vetterling and Flannery,
88: (11)                "Numerical Recipes in C++," 2nd ed,
89: (11)                  Cambridge University Press, 2002, p. 31.
90: (4)          """
91: (4)          def __init__(self, float_conv=float,int_conv=int,
92: (17)                    float_to_float=float,
93: (17)                    float_to_str=lambda v:'%24.16e' % v,
94: (17)                    title='Python floating point number'):
95: (8)            """
96: (8)              float_conv - convert integer to float (array)
97: (8)              int_conv   - convert float (array) to integer
98: (8)              float_to_float - convert float array to float
99: (8)              float_to_str - convert array float to str
100: (8)              title       - description of used floating point numbers
101: (8)            """
102: (8)            with errstate(under='ignore'):
103: (12)              self._do_init(float_conv, int_conv, float_to_float, float_to_str,
title)
104: (4)          def _do_init(self, float_conv, int_conv, float_to_float, float_to_str,
title):
105: (8)            max_iterN = 10000
106: (8)            msg = "Did not converge after %d tries with %s"
107: (8)            one = float_conv(1)
108: (8)            two = one + one
109: (8)            zero = one - one
110: (8)            a = one
111: (8)            for _ in range(max_iterN):
112: (12)              a = a + a
113: (12)              temp = a + one
114: (12)              temp1 = temp - a
115: (12)              if any(temp1 - one != zero):
116: (16)                break
117: (8)            else:
118: (12)              raise RuntimeError(msg % (_, one.dtype))
119: (8)            b = one
120: (8)            for _ in range(max_iterN):
121: (12)              b = b + b
122: (12)              temp = a + b
123: (12)              itemp = int_conv(temp-a)
124: (12)              if any(itemp != 0):
125: (16)                break
126: (8)            else:
127: (12)              raise RuntimeError(msg % (_, one.dtype))
128: (8)            ibeta = itemp
129: (8)            beta = float_conv(ibeta)
130: (8)            it = -1
131: (8)            b = one
132: (8)            for _ in range(max_iterN):
133: (12)              it = it + 1
134: (12)              b = b * beta
135: (12)              temp = b + one
136: (12)              temp1 = temp - b
137: (12)              if any(temp1 - one != zero):
138: (16)                break
139: (8)            else:
140: (12)              raise RuntimeError(msg % (_, one.dtype))
141: (8)            betah = beta / two
142: (8)            a = one

```

```

143: (8)           for _ in range(max_iterN):
144: (12)             a = a + a
145: (12)             temp = a + one
146: (12)             temp1 = temp - a
147: (12)             if any(temp1 - one != zero):
148: (16)                 break
149: (8)             else:
150: (12)                 raise RuntimeError(msg % (_, one.dtype))
151: (8)             temp = a + betah
152: (8)             irnd = 0
153: (8)             if any(temp-a != zero):
154: (12)                 irnd = 1
155: (8)             tempa = a + beta
156: (8)             temp = tempa + betah
157: (8)             if irnd == 0 and any(temp-tempa != zero):
158: (12)                 irnd = 2
159: (8)             negep = it + 3
160: (8)             betain = one / beta
161: (8)             a = one
162: (8)             for i in range(negep):
163: (12)                 a = a * betain
164: (8)             b = a
165: (8)             for _ in range(max_iterN):
166: (12)                 temp = one - a
167: (12)                 if any(temp-one != zero):
168: (16)                     break
169: (12)                 a = a * beta
170: (12)                 negep = negep - 1
171: (12)                 if negep < 0:
172: (16)                     raise RuntimeError("could not determine machine tolerance "
173: (35)                                     "for 'negep', locals() -> %s" % (locals()))
174: (8)             else:
175: (12)                 raise RuntimeError(msg % (_, one.dtype))
176: (8)             negep = -negep
177: (8)             epsneg = a
178: (8)             machep = - it - 3
179: (8)             a = b
180: (8)             for _ in range(max_iterN):
181: (12)                 temp = one + a
182: (12)                 if any(temp-one != zero):
183: (16)                     break
184: (12)                 a = a * beta
185: (12)                 machep = machep + 1
186: (8)             else:
187: (12)                 raise RuntimeError(msg % (_, one.dtype))
188: (8)             eps = a
189: (8)             ngrd = 0
190: (8)             temp = one + eps
191: (8)             if irnd == 0 and any(temp*one - one != zero):
192: (12)                 ngrd = 1
193: (8)             i = 0
194: (8)             k = 1
195: (8)             z = betain
196: (8)             t = one + eps
197: (8)             nxres = 0
198: (8)             for _ in range(max_iterN):
199: (12)                 y = z
200: (12)                 z = y*y
201: (12)                 a = z*one # Check here for underflow
202: (12)                 temp = z*t
203: (12)                 if any(a+a == zero) or any(abs(z) >= y):
204: (16)                     break
205: (12)                 temp1 = temp * betain
206: (12)                 if any(temp1*beta == z):
207: (16)                     break
208: (12)                 i = i + 1
209: (12)                 k = k + k
210: (8)             else:
211: (12)                 raise RuntimeError(msg % (_, one.dtype))

```

```

212: (8)             if ibeta != 10:
213: (12)            iexp = i + 1
214: (12)            mx = k + k
215: (8)            else:
216: (12)              iexp = 2
217: (12)              iz = ibeta
218: (12)              while k >= iz:
219: (16)                iz = iz * ibeta
220: (16)                iexp = iexp + 1
221: (12)                mx = iz + iz - 1
222: (8)            for _ in range(max_iterN):
223: (12)              xmin = y
224: (12)              y = y * betain
225: (12)              a = y * one
226: (12)              temp = y * t
227: (12)              if any((a + a) != zero) and any(abs(y) < xmin):
228: (16)                k = k + 1
229: (16)                temp1 = temp * betain
230: (16)                if any(temp1*beta == y) and any(temp != y):
231: (20)                  nxres = 3
232: (20)                  xmin = y
233: (20)                  break
234: (12)                else:
235: (16)                  break
236: (8)            else:
237: (12)              raise RuntimeError(msg % (_, one.dtype))
238: (8)            minexp = -k
239: (8)            if mx <= k + k - 3 and ibeta != 10:
240: (12)              mx = mx + mx
241: (12)              iexp = iexp + 1
242: (8)            maxexp = mx + minexp
243: (8)            irnd = irnd + nxres
244: (8)            if irnd >= 2:
245: (12)              maxexp = maxexp - 2
246: (8)            i = maxexp + minexp
247: (8)            if ibeta == 2 and not i:
248: (12)              maxexp = maxexp - 1
249: (8)            if i > 20:
250: (12)              maxexp = maxexp - 1
251: (8)            if any(a != y):
252: (12)              maxexp = maxexp - 2
253: (8)            xmax = one - epsneg
254: (8)            if any(xmax*one != xmax):
255: (12)              xmax = one - beta*epsneg
256: (8)            xmax = xmax / (xmin*beta*beta*beta)
257: (8)            i = maxexp + minexp + 3
258: (8)            for j in range(i):
259: (12)              if ibeta == 2:
260: (16)                xmax = xmax + xmax
261: (12)              else:
262: (16)                xmax = xmax * beta
263: (8)            smallest_subnormal = abs(xmin / beta ** (it))
264: (8)            self.ibeta = ibeta
265: (8)            self.it = it
266: (8)            self.negep = negep
267: (8)            self.epsneg = float_to_float(epsneg)
268: (8)            self._str_epsneg = float_to_str(epsneg)
269: (8)            self.machep = machep
270: (8)            self.eps = float_to_float(eps)
271: (8)            self._str_eps = float_to_str(eps)
272: (8)            self.ngrd = ngrd
273: (8)            self.iexp = iexp
274: (8)            self.minexp = minexp
275: (8)            self.xmin = float_to_float(xmin)
276: (8)            self._str_xmin = float_to_str(xmin)
277: (8)            self.maxexp = maxexp
278: (8)            self.xmax = float_to_float(xmax)
279: (8)            self._str_xmax = float_to_str(xmax)
280: (8)            self.irnd = irnd

```

```

281: (8)             self.title = title
282: (8)             self.epsilon = self.eps
283: (8)             self.tiny = self.xmin
284: (8)             self.huge = self.xmax
285: (8)             self.smallest_normal = self.xmin
286: (8)             self._str_smallest_normal = float_to_str(self.xmin)
287: (8)             self.smallest_subnormal = float_to_float(smallest_subnormal)
288: (8)             self._str_smallest_subnormal = float_to_str(smallest_subnormal)
289: (8)             import math
290: (8)             self.precision = int(-math.log10(float_to_float(self.eps)))
291: (8)             ten = two + two + two + two + two
292: (8)             resolution = ten ** (-self.precision)
293: (8)             self.resolution = float_to_float(resolution)
294: (8)             self._str_resolution = float_to_str(resolution)
295: (4)             def __str__(self):
296: (8)                 fmt = (
297: (11)                     'Machine parameters for %(title)s\n'
298: (11)                     '-----\n'
299: (11)                     'ibeta=%(ibeta)s it=%(it)s iexp=%(iexp)s ngrd=%(ngrd)s irnd=%
(irnd)s\n'
300: (11)                     'machep=%(machep)s      eps=%(_str_eps)s (beta**machep ==
epsilon)\n'
301: (11)                     'negep=%(negep)s  epsneg=%(_str_epsneg)s (beta**epsneg)\n'
302: (11)                     'minexp=%(minexp)s  xmin=%(_str_xmin)s (beta**minexp == tiny)\n'
303: (11)                     'maxexp=%(maxexp)s  xmax=%(_str_xmax)s ((1-epsneg)*beta**maxexp
== huge)\n'
304: (11)                     'smallest_normal=%(smallest_normal)s      '
305: (11)                     'smallest_subnormal=%(smallest_subnormal)s\n'
306: (11)                     '-----\n'
307: (11)             )
308: (8)             return fmt % self.__dict__
309: (0)             if __name__ == '__main__':
310: (4)                 print(MachAr())
-----
```

## File 68 - \_methods.py:

```

1: (0)             """
2: (0)             Array methods which are called by both the C-code for the method
3: (0)             and the Python code for the NumPy-namespace function
4: (0)             """
5: (0)             import warnings
6: (0)             from contextlib import nullcontext
7: (0)             from numpy.core import multiarray as mu
8: (0)             from numpy.core import umath as um
9: (0)             from numpy.core.multiarray import asanyarray
10: (0)            from numpy.core import numeric_types as nt
11: (0)            from numpy.core import _exceptions
12: (0)            from numpy.core._ufunc_config import _no_nep50_warning
13: (0)            from numpy._globals import _NoValue
14: (0)            from numpy.compat import pickle, os_fspath
15: (0)            umr_maximum = um.maximum.reduce
16: (0)            umr_minimum = um.minimum.reduce
17: (0)            umr_sum = um.add.reduce
18: (0)            umr_prod = um.multiply.reduce
19: (0)            umr_any = um.logical_or.reduce
20: (0)            umr_all = um.logical_and.reduce
21: (0)            _complex_to_float = {
22: (4)                nt.dtype(nt.csingle) : nt.dtype(nt.single),
23: (4)                nt.dtype(nt.cdouble) : nt.dtype(nt.double),
24: (0)            }
25: (0)            if nt.dtype(nt.longdouble) != nt.dtype(nt.double):
26: (4)                _complex_to_float.update({
27: (8)                    nt.dtype(nt.clongdouble) : nt.dtype(nt.longdouble),
28: (4)                })
29: (0)            def _amax(a, axis=None, out=None, keepdims=False,
```

```

30: (10)           initial=_NoValue, where=True):
31: (4)            return umr_maximum(a, axis, None, out, keepdims, initial, where)
32: (0)            def _amin(a, axis=None, out=None, keepdims=False,
33: (10)              initial=_NoValue, where=True):
34: (4)            return umr_minimum(a, axis, None, out, keepdims, initial, where)
35: (0)            def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
36: (9)              initial=_NoValue, where=True):
37: (4)            return umr_sum(a, axis, dtype, out, keepdims, initial, where)
38: (0)            def _prod(a, axis=None, dtype=None, out=None, keepdims=False,
39: (10)              initial=_NoValue, where=True):
40: (4)            return umr_prod(a, axis, dtype, out, keepdims, initial, where)
41: (0)            def _any(a, axis=None, dtype=None, out=None, keepdims=False, *, where=True):
42: (4)              if where is True:
43: (8)                return umr_any(a, axis, dtype, out, keepdims)
44: (4)                return umr_any(a, axis, dtype, out, keepdims, where=where)
45: (0)            def _all(a, axis=None, dtype=None, out=None, keepdims=False, *, where=True):
46: (4)              if where is True:
47: (8)                return umr_all(a, axis, dtype, out, keepdims)
48: (4)                return umr_all(a, axis, dtype, out, keepdims, where=where)
49: (0)            def _count_reduce_items(arr, axis, keepdims=False, where=True):
50: (4)              if where is True:
51: (8)                if axis is None:
52: (12)                  axis = tuple(range(arr.ndim))
53: (8)                elif not isinstance(axis, tuple):
54: (12)                  axis = (axis,)
55: (8)                items = 1
56: (8)                for ax in axis:
57: (12)                  items *= arr.shape[mu.normalize_axis_index(ax, arr.ndim)]
58: (8)                items = nt.intp(items)
59: (4)              else:
60: (8)                from numpy.lib.stride_tricks import broadcast_to
61: (8)                items = umr_sum(broadcast_to(where, arr.shape), axis, nt.intp, None,
62: (24)                  keepdims)
63: (4)              return items
64: (0)            def _clip(a, min=None, max=None, out=None, **kwargs):
65: (4)              if min is None and max is None:
66: (8)                raise ValueError("One of max or min must be given")
67: (4)              if min is None:
68: (8)                return um.minimum(a, max, out=out, **kwargs)
69: (4)              elif max is None:
70: (8)                return um.maximum(a, min, out=out, **kwargs)
71: (4)              else:
72: (8)                return um.clip(a, min, max, out=out, **kwargs)
73: (0)            def _mean(a, axis=None, dtype=None, out=None, keepdims=False, *, where=True):
74: (4)              arr = asanyarray(a)
75: (4)              is_float16_result = False
76: (4)              rcount = _count_reduce_items(arr, axis, keepdims=keepdims, where=where)
77: (4)              if rcount == 0 if where is True else umr_any(rcount == 0, axis=None):
78: (8)                warnings.warn("Mean of empty slice.", RuntimeWarning, stacklevel=2)
79: (4)              if dtype is None:
80: (8)                if issubclass(arr.dtype.type, (nt.integer, nt.bool_)):
81: (12)                  dtype = mu.dtype('f8')
82: (8)                elif issubclass(arr.dtype.type, nt.float16):
83: (12)                  dtype = mu.dtype('f4')
84: (12)                  is_float16_result = True
85: (4)              ret = umr_sum(arr, axis, dtype, out, keepdims, where=where)
86: (4)              if isinstance(ret, mu.ndarray):
87: (8)                with _no_nep50_warning():
88: (12)                  ret = um.true_divide(
89: (20)                      ret, rcount, out=ret, casting='unsafe', subok=False)
90: (8)                  if is_float16_result and out is None:
91: (12)                      ret = arr.dtype.type(ret)
92: (4)                      elif hasattr(ret, 'dtype'):
93: (8)                        if is_float16_result:
94: (12)                          ret = arr.dtype.type(ret / rcount)
95: (8)                        else:
96: (12)                          ret = ret.dtype.type(ret / rcount)
97: (4)                        else:
98: (8)                          ret = ret / rcount

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

99: (4)
100: (0)
101: (9)
102: (4)
103: (4)
104: (4)
axis=None):
105: (8)
106: (22)
107: (4)
108: (8)
109: (4)
110: (4)
111: (8)
112: (4)
113: (8)
114: (4)
115: (8)
116: (12)
117: (37)
118: (4)
119: (8)
120: (4)
121: (8)
122: (4)
123: (4)
124: (8)
125: (4)
126: (8)
127: (8)
128: (8)
129: (4)
130: (8)
131: (4)
132: (4)
133: (4)
134: (8)
135: (12)
136: (20)
137: (4)
138: (8)
139: (4)
140: (8)
141: (4)
return ret
def _var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False, *,
        where=True):
    arr = asanyarray(a)
    rcount = _count_reduce_items(arr, axis, keepdims=keepdims, where=where)
    if ddof >= rcount if where is True else umr_any(ddof >= rcount,
   warnings.warn("Degrees of freedom <= 0 for slice", RuntimeWarning,
   stacklevel=2)
    if dtype is None and issubclass(arr.dtype.type, (nt.integer, nt.bool_)):
        dtype = mu.dtype('f8')
    arrmean = umr_sum(arr, axis, dtype, keepdims=True, where=where)
    if rcount.ndim == 0:
        div = rcount
    else:
        div = rcount.reshape(arrmean.shape)
    if isinstance(arrmean, mu.ndarray):
        with _no_nep50_warning():
            arrmean = um.true_divide(arrmean, div, out=arrmean,
                                      casting='unsafe', subok=False)
    elif hasattr(arrmean, "dtype"):
        arrmean = arrmean.dtype.type(arrmean / rcount)
    else:
        arrmean = arrmean / rcount
    x = asanyarray(arr - arrmean)
    if issubclass(arr.dtype.type, (nt.floating, nt.integer)):
        x = um.multiply(x, x, out=x)
    elif x.dtype in _complex_to_float:
        xv = x.view(dtype=_complex_to_float[x.dtype], (2,))
        um.multiply(xv, xv, out=xv)
        x = um.add(xv[..., 0], xv[..., 1], out=x.real).real
    else:
        x = um.multiply(x, um.conjugate(x), out=x).real
    ret = umr_sum(x, axis, dtype, out, keepdims=keepdims, where=where)
    rcount = um.maximum(rcount - ddof, 0)
    if isinstance(ret, mu.ndarray):
        with _no_nep50_warning():
            ret = um.true_divide(
                ret, rcount, out=ret, casting='unsafe', subok=False)
    elif hasattr(ret, 'dtype'):
        ret = ret.dtype.type(ret / rcount)
    else:
        ret = ret / rcount
    return ret
def _std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False, *,
        where=True):
    ret = _var(a, axis=axis, dtype=dtype, out=out, ddof=ddof,
               keepdims=keepdims, where=where)
    if isinstance(ret, mu.ndarray):
        ret = um.sqrt(ret, out=out)
    elif hasattr(ret, 'dtype'):
        ret = ret.dtype.type(um.sqrt(ret))
    else:
        ret = um.sqrt(ret)
    return ret
def _ptp(a, axis=None, keepdims=False):
    return um.subtract(
        umr_maximum(a, axis, None, out, keepdims),
        umr_minimum(a, axis, None, None, keepdims),
        out
    )
def _dump(self, file, protocol=2):
    if hasattr(file, 'write'):
        ctx = nullcontext(file)
    else:
        ctx = open(os_fspath(file), "wb")
    with ctx as f:
        pickle.dump(self, f, protocol=protocol)
def _dumps(self, protocol=2):

```

167: (4)

```
        return pickle.dumps(self, protocol=protocol)
```

-----  
File 69 - \_string\_helpers.py:

```
1: (0)      """
2: (0)      String-handling utilities to avoid locale-dependence.
3: (0)      Used primarily to generate type name aliases.
4: (0)      """
5: (0)      _all_chars = tuple(map(chr, range(256)))
6: (0)      _ascii_upper = _all_chars[65:65+26]
7: (0)      _ascii_lower = _all_chars[97:97+26]
8: (0)      LOWER_TABLE = "".join(_all_chars[:65] + _ascii_lower + _all_chars[65+26:])
9: (0)      UPPER_TABLE = "".join(_all_chars[:97] + _ascii_upper + _all_chars[97+26:])
10: (0)     def english_lower(s):
11: (4)         """ Apply English case rules to convert ASCII strings to all lower case.
12: (4)         This is an internal utility function to replace calls to str.lower() such
13: (4)         that we can avoid changing behavior with changing locales. In particular,
14: (4)         Turkish has distinct dotted and dotless variants of the Latin letter "I"
15: (4)     in
16: (4)         both lowercase and uppercase. Thus, "I".lower() != "i" in a "tr" locale.
17: (4)     Parameters
18: (4)     -----
19: (4)     s : str
20: (4)     Returns
21: (4)     -----
22: (4)     lowered : str
23: (4)     Examples
24: (4)     -----
25: (4)     >>> from numpy.core.numerictypes import english_lower
26: (4)     >>>
english_lower('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_')
27: (4)     'abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz0123456789_'
28: (4)     >>> english_lower('')
29: (4)     ''
30: (4)     """
31: (4)     lowered = s.translate(LOWER_TABLE)
32: (0)     return lowered
33: (0)     def english_upper(s):
34: (4)         """ Apply English case rules to convert ASCII strings to all upper case.
35: (4)         This is an internal utility function to replace calls to str.upper() such
36: (4)         that we can avoid changing behavior with changing locales. In particular,
37: (4)         Turkish has distinct dotted and dotless variants of the Latin letter "I"
38: (4)     in
39: (4)         both lowercase and uppercase. Thus, "i".upper() != "I" in a "tr" locale.
40: (4)     Parameters
41: (4)     -----
42: (4)     s : str
43: (4)     Returns
44: (4)     -----
45: (4)     uppered : str
46: (4)     Examples
47: (4)     -----
48: (4)     >>> from numpy.core.numerictypes import english_upper
49: (4)     >>>
english_upper('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_')
50: (4)     'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_'
51: (4)     >>> english_upper('')
52: (4)     ''
53: (4)     """
54: (0)     uppered = s.translate(UPPER_TABLE)
55: (4)     return uppered
56: (0)     def english_capitalize(s):
57: (4)         """ Apply English case rules to convert the first character of an ASCII
58: (4)         string to upper case.
59: (4)         This is an internal utility function to replace calls to str.capitalize()
59: (4)         such that we can avoid changing behavior with changing locales.
59: (4)     Parameters
```

```

60: (4)      -----
61: (4)      s : str
62: (4)      Returns
63: (4)      -----
64: (4)      capitalized : str
65: (4)      Examples
66: (4)      -----
67: (4)      >>> from numpy.core.numerictypes import english_capitalize
68: (4)      >>> english_capitalize('int8')
69: (4)      'Int8'
70: (4)      >>> english_capitalize('Int8')
71: (4)      'Int8'
72: (4)      >>> english_capitalize('')
73: (4)      ''
74: (4)      """
75: (4)      if s:
76: (8)          return english_upper(s[0]) + s[1:]
77: (4)      else:
78: (8)          return s
-----
```

## File 70 - \_type\_aliases.py:

```

1: (0)      """
2: (0)      Due to compatibility, numpy has a very large number of different naming
3: (0)      conventions for the scalar types (those subclassing from `numpy.generic`).
4: (0)      This file produces a convoluted set of dictionaries mapping names to types,
5: (0)      and sometimes other mappings too.
6: (0)      .. data:: allTypes
7: (4)          A dictionary of names to types that will be exposed as attributes through
8: (4)          ``np.core.numerictypes.*``
9: (0)      .. data:: sctypeDict
10: (4)         Similar to `allTypes`, but maps a broader set of aliases to their types.
11: (0)      .. data:: scatypes
12: (4)         A dictionary keyed by a "type group" string, providing a list of types
13: (4)         under that group.
14: (0)      """
15: (0)      from numpy.compat import unicode
16: (0)      from numpy.core._string_helpers import english_lower
17: (0)      from numpy.core.multiarray import typeinfo, dtype
18: (0)      from numpy.core._dtype import _kind_name
19: (0)      sctypeDict = {}      # Contains all leaf-node scalar types with aliases
20: (0)      allTypes = {}        # Collect the types we will add to the module
21: (0)      _abstract_types = {}
22: (0)      _concrete_typeinfo = {}
23: (0)      for k, v in typeinfo.items():
24: (4)          k = english_lower(k)
25: (4)          if isinstance(v, type):
26: (8)              _abstract_types[k] = v
27: (4)          else:
28: (8)              _concrete_typeinfo[k] = v
29: (0)      _concrete_types = {v.type for k, v in _concrete_typeinfo.items()}
30: (0)      def _bits_of(obj):
31: (4)          try:
32: (8)              info = next(v for v in _concrete_typeinfo.values() if v.type is obj)
33: (4)          except StopIteration:
34: (8)              if obj in _abstract_types.values():
35: (12)                  msg = "Cannot count the bits of an abstract type"
36: (12)                  raise ValueError(msg) from None
37: (8)              return dtype(obj).itemsize * 8
38: (4)          else:
39: (8)              return info.bits
40: (0)      def bitname(obj):
41: (4)          """Return a bit-width name for a given type object"""
42: (4)          bits = _bits_of(obj)
43: (4)          dt = dtype(obj)
44: (4)          char = dt.kind
45: (4)          base = _kind_name(dt)
```



```

115: (17)                     'complex', 'bool', 'string', 'datetime', 'timedelta',
116: (17)                     'bytes', 'str']
117: (4)          for t in to_remove:
118: (8)            try:
119: (12)              del allTypes[t]
120: (12)              del sctypeDict[t]
121: (8)            except KeyError:
122: (12)              pass
123: (4)          attrs_to_remove = ['ulong']
124: (4)          for t in attrs_to_remove:
125: (8)            try:
126: (12)              del allTypes[t]
127: (8)            except KeyError:
128: (12)              pass
129: (0)          _set_up_aliases()
130: (0)          sctypes = {'int': [],
131: (11)            'uint': [],
132: (11)            'float': [],
133: (11)            'complex': [],
134: (11)            'others':[bool, object, bytes, unicode, void]}
135: (0)          def _add_array_type(typename, bits):
136: (4)            try:
137: (8)              t = allTypes['%s%d' % (typename, bits)]
138: (4)            except KeyError:
139: (8)              pass
140: (4)            else:
141: (8)              sctypes[typename].append(t)
142: (0)          def _set_array_types():
143: (4)            ibytes = [1, 2, 4, 8, 16, 32, 64]
144: (4)            fbytes = [2, 4, 8, 10, 12, 16, 32, 64]
145: (4)            for bytes in ibytes:
146: (8)              bits = 8*bytes
147: (8)              _add_array_type('int', bits)
148: (8)              _add_array_type('uint', bits)
149: (4)            for bytes in fbytes:
150: (8)              bits = 8*bytes
151: (8)              _add_array_type('float', bits)
152: (8)              _add_array_type('complex', 2*bits)
153: (4)              _gi = dtype('p')
154: (4)              if _gi.type not in sctypes['int']:
155: (8)                indx = 0
156: (8)                sz = _gi.itemsize
157: (8)                _lst = sctypes['int']
158: (8)                while (indx < len(_lst) and sz >= _lst[indx](0).itemsize):
159: (12)                  indx += 1
160: (8)                sctypes['int'].insert(indx, _gi.type)
161: (8)                sctypes['uint'].insert(indx, dtype('P').type)
162: (0)          _set_array_types()
163: (0)          _toadd = ['int', 'float', 'complex', 'bool', 'object',
164: (10)            'str', 'bytes', ('a', 'bytes_'),
165: (10)            ('int0', 'intp'), ('uint0', 'uintp')]
166: (0)          for name in _toadd:
167: (4)            if isinstance(name, tuple):
168: (8)              sctypeDict[name[0]] = allTypes[name[1]]
169: (4)            else:
170: (8)              sctypeDict[name] = allTypes['%s_' % name]
171: (0)          del _toadd, name

```

---

## File 71 - \_ufunc\_config.py:

```

1: (0)      """
2: (0)      Functions for changing global ufunc configuration
3: (0)      This provides helpers which wrap `umath.geterrobj` and `umath.seterrobj`
4: (0)      """
5: (0)      import collections.abc
6: (0)      import contextlib
7: (0)      import contextvars

```

```

8: (0)
9: (0)
10: (4)
11: (4)
ERR_DEFAULT,
12: (4)
13: (0)
14: (0)
15: (0)
16: (4)
"seterrcall",
17: (4)
18: (0)
19: (0)
20: (12)
21: (12)
22: (12)
23: (12)
24: (12)
25: (0)
26: (0)
27: (0)
28: (4)
29: (4)
30: (4)
31: (4)
32: (4)
33: (4)
34: (4)
35: (8)
36: (8)
37: (8)
38: (8)
39: (8)
40: (8)
41: (8)
42: (8)
43: (4)
44: (8)
45: (4)
46: (8)
47: (4)
48: (8)
49: (4)
50: (8)
51: (4)
52: (4)
53: (4)
54: (8)
55: (4)
56: (4)
57: (4)
58: (4)
59: (4)
60: (4)
61: (4)
62: (4)
63: (4)
64: (4)
65: (6)
66: (4)
67: (6)
68: (4)
69: (4)
70: (4)
71: (4)
72: (4)
73: (4)
'ignore'}
from .._utils import set_module
from .umath import (
    UFUNC_BUFSIZE_DEFAULT,
    ERR_IGNORE, ERR_WARN, ERR_RAISE, ERR_CALL, ERR_PRINT, ERR_LOG,
    SHIFT_DIVIDEBYZERO, SHIFT_OVERFLOW, SHIFT_UNDERFLOW, SHIFT_INVALID,
)
from . import umath
__all__ = [
    "seterr", "geterr", "setbufsize", "getbufsize", "seterrcall",
    "errstate", '_no_nep50_warning'
]
_errdict = {"ignore": ERR_IGNORE,
            "warn": ERR_WARN,
            "raise": ERR_RAISE,
            "call": ERR_CALL,
            "print": ERR_PRINT,
            "log": ERR_LOG}
_errdict_rev = {value: key for key, value in _errdict.items()}
@set_module('numpy')
def seterr(all=None, divide=None, over=None, under=None, invalid=None):
    """
    Set how floating-point errors are handled.
    Note that operations on integer scalar types (such as `int16`) are
    handled like floating point, and are affected by these settings.
    Parameters
    -----
    all : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
        Set treatment for all types of floating-point errors at once:
        - ignore: Take no action when the exception occurs.
        - warn: Print a `RuntimeWarning` (via the Python `warnings` module).
        - raise: Raise a `FloatingPointError`.
        - call: Call a function specified using the `seterrcall` function.
        - print: Print a warning directly to ``stdout``.
        - log: Record error in a Log object specified by `seterrcall`.
    The default is not to change the current behavior.
    divide : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
        Treatment for division by zero.
    over : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
        Treatment for floating-point overflow.
    under : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
        Treatment for floating-point underflow.
    invalid : {'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional
        Treatment for invalid floating-point operation.
    Returns
    -----
    old_settings : dict
        Dictionary containing the old settings.
    See also
    -----
    seterrcall : Set a callback function for the 'call' mode.
    geterr, geterrcall, errstate
    Notes
    -----
    The floating-point exceptions are defined in the IEEE 754 standard [1]_:
    - Division by zero: infinite result obtained from finite numbers.
    - Overflow: result too large to be expressed.
    - Underflow: result so close to zero that some precision
        was lost.
    - Invalid operation: result is not an expressible number, typically
        indicates that a NaN was produced.
    .. [1] https://en.wikipedia.org/wiki/IEEE\_754
    Examples
    -----
    >>> old_settings = np.seterr(all='ignore') #seterr to known value
    >>> np.seterr(over='raise')
    {'divide': 'ignore', 'over': 'ignore', 'under': 'ignore', 'invalid':
    'ignore'}

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

74: (4)
75: (4)
'ignore'}
76: (4)
77: (4)
78: (4)
79: (4)
80: (4)
Traceback (most recent call last):
81: (6)
File "<stdin>", line 1, in <module>
82: (4)
FloatingPointError: overflow encountered in scalar multiply
83: (4)
84: (4)
85: (4)
{'divide': 'print', 'over': 'print', 'under': 'print', 'invalid': 'print'}
86: (4)
87: (4)
88: (4)
89: (4)
pyvals = umath.geterrobj()
90: (4)
old = geterr()
91: (4)
if divide is None:
92: (8)
    divide = all or old['divide']
93: (4)
if over is None:
94: (8)
    over = all or old['over']
95: (4)
if under is None:
96: (8)
    under = all or old['under']
97: (4)
if invalid is None:
98: (8)
    invalid = all or old['invalid']
99: (4)
maskvalue = (_errdict[divide] << SHIFT_DIVIDEZERO) +
100: (17)
        (_errdict[over] << SHIFT_OVERFLOW) +
101: (17)
        (_errdict[under] << SHIFT_UNDERFLOW) +
102: (17)
        (_errdict[invalid] << SHIFT_INVALID))
103: (4)
pyvals[1] = maskvalue
104: (4)
umath.seterrobj(pyvals)
105: (4)
return old
106: (0)
@set_module('numpy')
107: (0)
def geterr():
108: (4)
    """
109: (4)
Get the current way of handling floating-point errors.
110: (4)
Returns
111: (4)
-----
112: (4)
res : dict
113: (8)
    A dictionary with keys "divide", "over", "under", and "invalid",
114: (8)
    whose values are from the strings "ignore", "print", "log", "warn",
115: (8)
    "raise", and "call". The keys represent possible floating-point
116: (8)
    exceptions, and the values define how these exceptions are handled.
117: (4)
See Also
118: (4)
-----
119: (4)
geterrcall, seterr, seterrcall
120: (4)
Notes
121: (4)
-----
122: (4)
For complete documentation of the types of floating-point exceptions and
123: (4)
treatment options, see `seterr`.
124: (4)
Examples
125: (4)
-----
126: (4)
>>> np.geterr()
127: (4)
{'divide': 'warn', 'over': 'warn', 'under': 'ignore', 'invalid': 'warn'}
128: (4)
>>> np.arange(3.) / np.arange(3.)
129: (4)
array([nan, 1., 1.])
130: (4)
>>> oldsettings = np.seterr(all='warn', over='raise')
131: (4)
>>> np.geterr()
132: (4)
{'divide': 'warn', 'over': 'raise', 'under': 'warn', 'invalid': 'warn'}
133: (4)
>>> np.arange(3.) / np.arange(3.)
134: (4)
array([nan, 1., 1.])
135: (4)
"""
136: (4)
maskvalue = umath.geterrobj()[1]
137: (4)
mask = 7
138: (4)
res = {}
139: (4)
val = (maskvalue >> SHIFT_DIVIDEZERO) & mask
140: (4)
res['divide'] = _errdict_rev[val]
141: (4)
val = (maskvalue >> SHIFT_OVERFLOW) & mask

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

142: (4)             res['over'] = _errdict_rev[val]
143: (4)             val = (maskvalue >> SHIFT_UNDERFLOW) & mask
144: (4)             res['under'] = _errdict_rev[val]
145: (4)             val = (maskvalue >> SHIFT_INVALID) & mask
146: (4)             res['invalid'] = _errdict_rev[val]
147: (4)             return res
148: (0) @set_module('numpy')
149: (0) def setbufsize(size):
150: (4)     """
151: (4)         Set the size of the buffer used in ufuncs.
152: (4)         Parameters
153: (4)         -----
154: (4)         size : int
155: (8)             Size of buffer.
156: (4)
157: (4)         if size > 10e6:
158: (8)             raise ValueError("Buffer size, %s, is too big." % size)
159: (4)         if size < 5:
160: (8)             raise ValueError("Buffer size, %s, is too small." % size)
161: (4)         if size % 16 != 0:
162: (8)             raise ValueError("Buffer size, %s, is not a multiple of 16." % size)
163: (4)         pyvals = umath.geterrobj()
164: (4)         old = getbufsize()
165: (4)         pyvals[0] = size
166: (4)         umath.seterrobj(pyvals)
167: (4)         return old
168: (0) @set_module('numpy')
169: (0) def getbufsize():
170: (4)     """
171: (4)         Return the size of the buffer used in ufuncs.
172: (4)         Returns
173: (4)         -----
174: (4)         bufsize : int
175: (8)             Size of ufunc buffer in bytes.
176: (4)
177: (4)         return umath.geterrobj()[0]
178: (0) @set_module('numpy')
179: (0) def seterrcall(func):
180: (4)     """
181: (4)         Set the floating-point error callback function or log object.
182: (4)         There are two ways to capture floating-point error messages. The first
183: (4)         is to set the error-handler to 'call', using `seterr`. Then, set
184: (4)         the function to call using this function.
185: (4)         The second is to set the error-handler to 'log', using `seterr`.
186: (4)         Floating-point errors then trigger a call to the 'write' method of
187: (4)         the provided object.
188: (4)         Parameters
189: (4)         -----
190: (4)         func : callable f(err, flag) or object with write method
191: (8)             Function to call upon floating-point errors ('call'-mode) or
192: (8)             object whose 'write' method is used to log such message ('log'-mode).
193: (8)
describing
194: (8)             The call function takes two arguments. The first is a string
195: (8)
196: (8)             the type of error (such as "divide by zero", "overflow", "underflow",
one
197: (8)                 or "invalid value"), and the second is the status flag. The flag is a
198: (10)                 byte, whose four least-significant bits indicate the type of error,
of "divide", "over", "under", "invalid"::
199: (8)                 [0 0 0 0 divide over under invalid]
200: (8)                 In other words, ``flags = divide + 2*over + 4*under + 8*invalid``.
201: (8)                 If an object is provided, its write method should take one argument,
a string.
202: (4)             Returns
203: (4)             -----
204: (4)             h : callable, log instance or None
205: (8)                 The old error handler.
206: (4)             See Also
207: (4)             -----
208: (4)                 seterr, geterr, seterrcall

```

```

209: (4) Examples
210: (4) -----
211: (4) Callback upon error:
212: (4) >>> def err_handler(type, flag):
213: (4) ...     print("Floating point error (%s), with flag %s" % (type, flag))
214: (4) ...
215: (4) >>> saved_handler = np.seterrcall(err_handler)
216: (4) >>> save_err = np.seterr(all='call')
217: (4) >>> np.array([1, 2, 3]) / 0.0
218: (4) Floating point error (divide by zero), with flag 1
219: (4) array([inf, inf, inf])
220: (4) >>> np.seterrcall(saved_handler)
221: (4) <function err_handler at 0x...>
222: (4) >>> np.seterr(**save_err)
223: (4) {'divide': 'call', 'over': 'call', 'under': 'call', 'invalid': 'call'}
224: (4) Log error message:
225: (4) >>> class Log:
226: (4) ...     def write(self, msg):
227: (4) ...         print("LOG: %s" % msg)
228: (4) ...
229: (4) >>> log = Log()
230: (4) >>> saved_handler = np.seterrcall(log)
231: (4) >>> save_err = np.seterr(all='log')
232: (4) >>> np.array([1, 2, 3]) / 0.0
233: (4) LOG: Warning: divide by zero encountered in divide
234: (4) array([inf, inf, inf])
235: (4) >>> np.seterrcall(saved_handler)
236: (4) <numpy.core.numeric.Log object at 0x...>
237: (4) >>> np.seterr(**save_err)
238: (4) {'divide': 'log', 'over': 'log', 'under': 'log', 'invalid': 'log'}
239: (4) """
240: (4) if func is not None and not isinstance(func, collections.abc.Callable):
241: (8)     if (not hasattr(func, 'write') or
242: (16)             not isinstance(func.write, collections.abc.Callable)):
243: (12)     raise ValueError("Only callable can be used as callback")
244: (4) pyvals = umath.geterrobj()
245: (4) old = geterrcall()
246: (4) pyvals[2] = func
247: (4) umath.seterrobj(pyvals)
248: (4) return old
249: (0) @set_module('numpy')
250: (0) def geterrcall():
251: (4) """
252: (4)     Return the current callback function used on floating-point errors.
253: (4)     When the error handling for a floating-point error (one of "divide",
254: (4)     "over", "under", or "invalid") is set to 'call' or 'log', the function
255: (4)     that is called or the log instance that is written to is returned by
256: (4)     `geterrcall`. This function or log instance has been set with
257: (4)     `seterrcall`.
258: (4)     Returns
259: (4)     -----
260: (4)     errobj : callable, log instance or None
261: (8)     The current error handler. If no handler was set through `seterrcall`,
262: (8)     ``None`` is returned.
263: (4) See Also
264: (4) -----
265: (4) seterrcall, seterr, geterr
266: (4) Notes
267: (4) -----
268: (4) For complete documentation of the types of floating-point exceptions and
269: (4) treatment options, see `seterr`.
270: (4) Examples
271: (4) -----
272: (4) >>> np.geterrcall() # we did not yet set a handler, returns None
273: (4) >>> oldsettings = np.seterr(all='call')
274: (4) >>> def err_handler(type, flag):
275: (4) ...     print("Floating point error (%s), with flag %s" % (type, flag))
276: (4) >>> oldhandler = np.seterrcall(err_handler)
277: (4) >>> np.array([1, 2, 3]) / 0.0

```

```

278: (4)           Floating point error (divide by zero), with flag 1
279: (4)           array([inf, inf, inf])
280: (4)           >>> cur_handler = np.geterrcall()
281: (4)           >>> cur_handler is err_handler
282: (4)           True
283: (4)           """
284: (4)           return umath.geterrobj()[2]
285: (0)           class _unspecified:
286: (4)               pass
287: (0)           _Unspecified = _unspecified()
288: (0)           @set_module('numpy')
289: (0)           class errstate(contextlib.ContextDecorator):
290: (4)               """
291: (4)               errstate(**kwargs)
292: (4)               Context manager for floating-point error handling.
293: (4)               Using an instance of `errstate` as a context manager allows statements in
294: (4)               that context to execute with a known error handling behavior. Upon
entering
295: (4)               the context the error handling is set with `seterr` and `seterrcall`, and
296: (4)               upon exiting it is reset to what it was before.
297: (4)               .. versionchanged:: 1.17.0
298: (8)                   `errstate` is also usable as a function decorator, saving
299: (8)                   a level of indentation if an entire function is wrapped.
300: (8)                   See :py:class:`contextlib.ContextDecorator` for more information.
301: (4)           Parameters
302: (4)           -----
303: (4)           kwargs : {divide, over, under, invalid}
304: (8)               Keyword arguments. The valid keywords are the possible floating-point
305: (8)               exceptions. Each keyword should have a string value that defines the
306: (8)               treatment for the particular error. Possible values are
307: (8)               {'ignore', 'warn', 'raise', 'call', 'print', 'log'}.
308: (4)           See Also
309: (4)           -----
310: (4)           seterr, geterr, seterrcall, geterrcall
311: (4)           Notes
312: (4)           -----
313: (4)           For complete documentation of the types of floating-point exceptions and
314: (4)           treatment options, see `seterr`.
315: (4)           Examples
316: (4)           -----
317: (4)           >>> olderr = np.seterr(all='ignore') # Set error handling to known state.
318: (4)           >>> np.arange(3) / 0.
319: (4)           array([nan, inf, inf])
320: (4)           >>> with np.errstate(divide='warn'):
321: (4)               ...     np.arange(3) / 0.
322: (4)           array([nan, inf, inf])
323: (4)           >>> np.sqrt(-1)
324: (4)           nan
325: (4)           >>> with np.errstate(invalid='raise'):
326: (4)               ...     np.sqrt(-1)
327: (4)           Traceback (most recent call last):
328: (6)               File "<stdin>", line 2, in <module>
329: (4)               FloatingPointError: invalid value encountered in sqrt
330: (4)           Outside the context the error handling behavior has not changed:
331: (4)           >>> np.geterr()
332: (4)           {'divide': 'ignore', 'over': 'ignore', 'under': 'ignore', 'invalid':
333: ('ignore')}
334: (4)           """
335: (8)           def __init__(self, *, call=_Unspecified, **kwargs):
336: (8)               self.call = call
337: (8)               self.kwargs = kwargs
338: (8)           def __enter__(self):
339: (8)               self.oldstate = seterr(**self.kwargs)
340: (12)              if self.call is not _Unspecified:
341: (4)                  self.oldcall = seterrcall(self.call)
342: (8)           def __exit__(self, *exc_info):
343: (8)               seterr(**self.oldstate)
344: (12)              if self.call is not _Unspecified:
344: (12)                  seterrcall(self.oldcall)

```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

345: (0) def _setdef():
346: (4)     defval = [UFUNC_BUFSIZE_DEFAULT, ERR_DEFAULT, None]
347: (4)     umath.seterrobj(defval)
348: (0)     _setdef()
349: (0)     NO_NE50_WARNING = contextvars.ContextVar("_no_nep50_warning", default=False)
350: (0)     @set_module('numpy')
351: (0)     @contextlib.contextmanager
352: (0)     def _no_nep50_warning():
353: (4)         """
354: (4)             Context manager to disable NEP 50 warnings. This context manager is
355: (4)             only relevant if the NEP 50 warnings are enabled globally (which is not
356: (4)             thread/context safe).
357: (4)             This warning context manager itself is fully safe, however.
358: (4)             """
359: (4)             token = NO_NE50_WARNING.set(True)
360: (4)             try:
361: (8)                 yield
362: (4)             finally:
363: (8)                 NO_NE50_WARNING.reset(token)
```

---

## File 72 - \_\_init\_\_.py:

```
1: (0) """
2: (0) Contains the core of NumPy: ndarray, ufuncs, dtypes, etc.
3: (0) Please note that this module is private. All functions and objects
4: (0) are available in the main ``numpy`` namespace - use that instead.
5: (0) """
6: (0) import os
7: (0) import warnings
8: (0) from numpy.version import version as __version__
9: (0) env_added = []
10: (0) for envkey in ['OPENBLAS_MAIN_FREE', 'GOTOBLAS_MAIN_FREE']:
11: (4)     if envkey not in os.environ:
12: (8)         os.environ[envkey] = '1'
13: (8)         env_added.append(envkey)
14: (0) try:
15: (4)     from . import multiarray
16: (0) except ImportError as exc:
17: (4)     import sys
18: (4)     msg = """
19: (0) IMPORTANT: PLEASE READ THIS FOR ADVICE ON HOW TO SOLVE THIS ISSUE!
20: (0) Importing the numpy C-extensions failed. This error can happen for
21: (0) many reasons, often due to issues with your setup or how NumPy was
22: (0) installed.
23: (0) We have compiled some common reasons and troubleshooting tips at:
24: (4)     https://numpy.org/devdocs/user/troubleshooting-importerror.html
25: (0) Please note and check the following:
26: (2)     * The Python version is: Python%d.%d from "%s"
27: (2)     * The NumPy version is: "%s"
28: (0) and make sure that they are the versions you expect.
29: (0) Please carefully study the documentation linked above for further help.
30: (0) Original error was: %s
31: (0)     """ % (sys.version_info[0], sys.version_info[1], sys.executable,
32: (8)         __version__, exc)
33: (4)     raise ImportError(msg)
34: (0) finally:
35: (4)     for envkey in env_added:
36: (8)         del os.environ[envkey]
37: (0)     del envkey
38: (0)     del env_added
39: (0)     del os
40: (0)     from . import umath
41: (0)     if not (hasattr(multiarray, '_multiarray_umath') and
42: (8)         hasattr(umath, '_multiarray_umath')):
43: (4)         import sys
44: (4)         path = sys.modules['numpy'].__path__
45: (4)         msg = ("Something is wrong with the numpy installation. ")
```

```

46: (8)             "While importing we detected an older version of "
47: (8)             "numpy in {}. One method of fixing this is to repeatedly uninstall "
48: (8)             "numpy until none is found, then reinstall this version.")
49: (4)             raise ImportError(msg.format(path))
50: (0)             from . import numeric_types as nt
51: (0)             multiarray.set_typeDict(nt.sctypeDict)
52: (0)             from . import numeric
53: (0)             from .numeric import *
54: (0)             from . import fromnumeric
55: (0)             from .fromnumeric import *
56: (0)             from . import defchararray as char
57: (0)             from . import records
58: (0)             from . import records as rec
59: (0)             from .records import record, recarray, format_parser
60: (0)             from .memmap import *
61: (0)             from .defchararray import chararray
62: (0)             from . import function_base
63: (0)             from .function_base import *
64: (0)             from . import _machar
65: (0)             from . import getlimits
66: (0)             from .getlimits import *
67: (0)             from . import shape_base
68: (0)             from .shape_base import *
69: (0)             from . import einsumfunc
70: (0)             from .einsumfunc import *
71: (0)             del nt
72: (0)             from .numeric import absolute as abs
73: (0)             from . import _add_newdocs
74: (0)             from . import _add_newdocs_scalars
75: (0)             from . import _dtype_ctypes
76: (0)             from . import _internal
77: (0)             from . import _dtype
78: (0)             from . import _methods
79: (0)             __all__ = ['char', 'rec', 'memmap']
80: (0)             __all__ += numeric.__all__
81: (0)             __all__ += ['record', 'recarray', 'format_parser']
82: (0)             __all__ += ['chararray']
83: (0)             __all__ += function_base.__all__
84: (0)             __all__ += getlimits.__all__
85: (0)             __all__ += shape_base.__all__
86: (0)             __all__ += einsumfunc.__all__
87: (0)             def _ufunc_reconstruct(module, name):
88: (4)                 mod = __import__(module, fromlist=[name])
89: (4)                 return getattr(mod, name)
90: (0)             def _ufunc_reduce(func):
91: (4)                 return func.__name__
92: (0)             def _DType_reconstruct(scalar_type):
93: (4)                 return type(dtype(scalar_type))
94: (0)             def _DType_reduce(DType):
95: (4)                 if not DType._legacy or DType.__module__ == "numpy.dtypes":
96: (8)                     return DType.__name__
97: (4)                     scalar_type = DType.type
98: (4)                     return _DType_reconstruct, (scalar_type,)
99: (0)             def __getattr__(name):
100: (4)                 if name == "MachAr":
101: (8)                     warnings.warn(
102: (12)                         "The `np.core.MachAr` is considered private API (NumPy 1.24)",
103: (12)                         DeprecationWarning, stacklevel=2,
104: (8)                     )
105: (8)                     return _machar.MachAr
106: (4)                     raise AttributeError(f"Module {__name__!r} has no attribute {name!r}")
107: (0)             import copyreg
108: (0)             copyreg.pickle(ufunc, _ufunc_reduce)
109: (0)             copyreg.pickle(type(dtype), _DType_reduce, _DType_reconstruct)
110: (0)             del copyreg
111: (0)             del _ufunc_reduce
112: (0)             del _DType_reduce
113: (0)             from numpy._pytesttester import PytestTester
114: (0)             test = PytestTester(__name__)

```

115: (0)

del PytestTester

-----  
File 73 - test\_abc.py:

```

1: (0)          from numpy.testing import assert_
2: (0)          import numbers
3: (0)          import numpy as np
4: (0)          from numpy.core.numericatypes import sctypes
5: (0)          class TestABC:
6: (4)              def test_abstract(self):
7: (8)                  assert_(issubclass(np.number, numbers.Number))
8: (8)                  assert_(issubclass(np.inexact, numbers.Complex))
9: (8)                  assert_(issubclass(np.complexfloating, numbers.Complex))
10: (8)                 assert_(issubclass(np.floating, numbers.Real))
11: (8)                 assert_(issubclass(np.integer, numbers.Integral))
12: (8)                 assert_(issubclass(np.signedinteger, numbers.Integral))
13: (8)                 assert_(issubclass(np.unsignedinteger, numbers.Integral))
14: (4)             def test_floats(self):
15: (8)                 for t in sctypes['float']:
16: (12)                     assert_(isinstance(t(), numbers.Real),
17: (20)                         f"{t.__name__} is not instance of Real")
18: (12)                     assert_(issubclass(t, numbers.Real),
19: (20)                         f"{t.__name__} is not subclass of Real")
20: (12)                     assert_(not isinstance(t(), numbers.Rational),
21: (20)                         f"{t.__name__} is instance of Rational")
22: (12)                     assert_(not issubclass(t, numbers.Rational),
23: (20)                         f"{t.__name__} is subclass of Rational")
24: (4)             def test_complex(self):
25: (8)                 for t in sctypes['complex']:
26: (12)                     assert_(isinstance(t(), numbers.Complex),
27: (20)                         f"{t.__name__} is not instance of Complex")
28: (12)                     assert_(issubclass(t, numbers.Complex),
29: (20)                         f"{t.__name__} is not subclass of Complex")
30: (12)                     assert_(not isinstance(t(), numbers.Real),
31: (20)                         f"{t.__name__} is instance of Real")
32: (12)                     assert_(not issubclass(t, numbers.Real),
33: (20)                         f"{t.__name__} is subclass of Real")
34: (4)             def test_int(self):
35: (8)                 for t in sctypes['int']:
36: (12)                     assert_(isinstance(t(), numbers.Integral),
37: (20)                         f"{t.__name__} is not instance of Integral")
38: (12)                     assert_(issubclass(t, numbers.Integral),
39: (20)                         f"{t.__name__} is not subclass of Integral")
40: (4)             def test_uint(self):
41: (8)                 for t in sctypes['uint']:
42: (12)                     assert_(isinstance(t(), numbers.Integral),
43: (20)                         f"{t.__name__} is not instance of Integral")
44: (12)                     assert_(issubclass(t, numbers.Integral),
45: (20)                         f"{t.__name__} is not subclass of Integral")

```

-----  
File 74 - test\_api.py:

```

1: (0)          import sys
2: (0)          import numpy as np
3: (0)          from numpy.core._rational_tests import rational
4: (0)          import pytest
5: (0)          from numpy.testing import (
6: (5)              assert_, assert_equal, assert_array_equal, assert_raises, assert_warns,
7: (5)              HAS_REFCOUNT
8: (4)          )
9: (0)          def test_array_array():
10: (4)              tobj = type(object)
11: (4)              ones11 = np.ones((1, 1), np.float64)
12: (4)              tndarray = type(ones11)
13: (4)              assert_equal(np.array(ones11, dtype=np.float64), ones11)

```

```

14: (4)           if HAS_REFCOUNT:
15: (8)             old_refcount = sys.getrefcount(tndarray)
16: (8)             np.array(ones11)
17: (8)             assert_equal(old_refcount, sys.getrefcount(tndarray))
18: (4)             assert_equal(np.array(None, dtype=np.float64),
19: (17)                           np.array(np.nan, dtype=np.float64))
20: (4)           if HAS_REFCOUNT:
21: (8)             old_refcount = sys.getrefcount(tobj)
22: (8)             np.array(None, dtype=np.float64)
23: (8)             assert_equal(old_refcount, sys.getrefcount(tobj))
24: (4)             assert_equal(np.array(1.0, dtype=np.float64),
25: (17)                           np.ones(), dtype=np.float64))
26: (4)           if HAS_REFCOUNT:
27: (8)             old_refcount = sys.getrefcount(np.float64)
28: (8)             np.array(np.array(1.0, dtype=np.float64), dtype=np.float64)
29: (8)             assert_equal(old_refcount, sys.getrefcount(np.float64))
30: (4)             S2 = np.dtype((bytes, 2))
31: (4)             S3 = np.dtype((bytes, 3))
32: (4)             S5 = np.dtype((bytes, 5))
33: (4)             assert_equal(np.array(b"1.0", dtype=np.float64),
34: (17)                           np.ones(), dtype=np.float64))
35: (4)             assert_equal(np.array(b"1.0").dtype, S3)
36: (4)             assert_equal(np.array(b"1.0", dtype=bytes).dtype, S3)
37: (4)             assert_equal(np.array(b"1.0", dtype=S2), np.array(b"1."))
38: (4)             assert_equal(np.array(b"1", dtype=S5), np.ones(), dtype=S5))
39: (4)             U2 = np.dtype((str, 2))
40: (4)             U3 = np.dtype((str, 3))
41: (4)             U5 = np.dtype((str, 5))
42: (4)             assert_equal(np.array("1.0", dtype=np.float64),
43: (17)                           np.ones(), dtype=np.float64))
44: (4)             assert_equal(np.array("1.0").dtype, U3)
45: (4)             assert_equal(np.array("1.0", dtype=str).dtype, U3)
46: (4)             assert_equal(np.array("1.0", dtype=U2), np.array(str("1.")))
47: (4)             assert_equal(np.array("1", dtype=U5), np.ones(), dtype=U5))
48: (4)             builtins = getattr(__builtins__, '__dict__', __builtins__)
49: (4)             assert_(hasattr(builtins, 'get'))
50: (4)             dat = np.array(memoryview(b'1.0'), dtype=np.float64)
51: (4)             assert_equal(dat, [49.0, 46.0, 48.0])
52: (4)             assert_(dat.dtype.type is np.float64)
53: (4)             dat = np.array(memoryview(b'1.0'))
54: (4)             assert_equal(dat, [49, 46, 48])
55: (4)             assert_(dat.dtype.type is np.uint8)
56: (4)             a = np.array(100.0, dtype=np.float64)
57: (4)             o = type("o", (object,), {
58: (13)               dict(__array_interface__=a.__array_interface__)
59: (4)             })
60: (4)             assert_equal(np.array(o, dtype=np.float64), a)
61: (17)             a = np.array([(1, 4.0, 'Hello'), (2, 6.0, 'World')],
62: (4)                           dtype=[('f0', int), ('f1', float), ('f2', str)])
63: (13)             o = type("o", (object,), {
64: (4)               dict(__array_struct__=a.__array_struct__)
65: (4)             })
66: (13)             assert_equal(bytes(np.array(o).data), bytes(a.data))
67: (4)             o = type("o", (object,), {
68: (4)               dict(__array_=lambda *x: np.array(100.0, dtype=np.float64))()
69: (4)             })
69: (4)             assert_equal(np.array(o, dtype=np.float64), np.array(100.0, np.float64))
70: (8)             nested = 1.5
71: (4)             for i in range(np.MAXDIMS):
72: (8)               nested = [nested]
73: (4)               np.array(nested)
74: (4)               assert_raises(ValueError, np.array, [nested], dtype=np.float64)
75: (4)               assert_equal(np.array([None] * 10, dtype=np.float32),
76: (17)                             np.full((10,), np.nan, dtype=np.float32))
77: (4)               assert_equal(np.array([[None]] * 10, dtype=np.float32),
78: (17)                             np.full((10, 1), np.nan, dtype=np.float32))
79: (4)               assert_equal(np.array([[None] * 10], dtype=np.float32),
80: (17)                             np.full((1, 10), np.nan, dtype=np.float32))
81: (4)               assert_equal(np.array([[None] * 10] * 10, dtype=np.float32),
82: (17)                             np.full((10, 10), np.nan, dtype=np.float32))
81: (4)               assert_equal(np.array([None] * 10, dtype=np.float64),
82: (17)                             np.full((10,), np.nan, dtype=np.float64))

```

```

83: (4) assert_equal(np.array([[None]] * 10, dtype=np.float64),
84: (17)         np.full((10, 1), np.nan, dtype=np.float64))
85: (4) assert_equal(np.array([[None] * 10], dtype=np.float64),
86: (17)         np.full((1, 10), np.nan, dtype=np.float64))
87: (4) assert_equal(np.array([[None] * 10] * 10, dtype=np.float64),
88: (17)         np.full((10, 10), np.nan, dtype=np.float64))
89: (4) assert_equal(np.array([1.0] * 10, dtype=np.float64),
90: (17)         np.ones((10,), dtype=np.float64))
91: (4) assert_equal(np.array([[1.0]] * 10, dtype=np.float64),
92: (17)         np.ones((10, 1), dtype=np.float64))
93: (4) assert_equal(np.array([[1.0] * 10], dtype=np.float64),
94: (17)         np.ones((1, 10), dtype=np.float64))
95: (4) assert_equal(np.array([[1.0] * 10] * 10, dtype=np.float64),
96: (17)         np.ones((10, 10), dtype=np.float64))
97: (4) assert_equal(np.array([(None,) * 10, dtype=np.float64),
98: (17)         np.full((10,), np.nan, dtype=np.float64))
99: (4) assert_equal(np.array([(None,) * 10], dtype=np.float64),
100: (17)         np.full((10, 1), np.nan, dtype=np.float64))
101: (4) assert_equal(np.array([(None,) * 10], dtype=np.float64),
102: (17)         np.full((1, 10), np.nan, dtype=np.float64))
103: (4) assert_equal(np.array([(None,) * 10] * 10, dtype=np.float64),
104: (17)         np.full((10, 10), np.nan, dtype=np.float64))
105: (4) assert_equal(np.array((1.0,) * 10, dtype=np.float64),
106: (17)         np.ones((10,), dtype=np.float64))
107: (4) assert_equal(np.array([(1.0,) * 10, dtype=np.float64),
108: (17)         np.ones((10, 1), dtype=np.float64))
109: (4) assert_equal(np.array([(1.0,) * 10], dtype=np.float64),
110: (17)         np.ones((1, 10), dtype=np.float64))
111: (4) assert_equal(np.array([(1.0,) * 10] * 10, dtype=np.float64),
112: (17)         np.ones((10, 10), dtype=np.float64))
113: (0) @pytest.mark.parametrize("array", [True, False])
114: (0) def test_array_impossible_casts(array):
115: (4)     rt = rational(1, 2)
116: (4)     if array:
117: (8)         rt = np.array(rt)
118: (4)     with assert_raises(TypeError):
119: (8)         np.array(rt, dtype="M8")
120: (0) @pytest.mark.parametrize("a",
121: (4)     (
122: (8)         np.array(2), # 0D array
123: (8)         np.array([3, 2, 7, 0]), # 1D array
124: (8)         np.arange(6).reshape(2, 3) # 2D array
125: (4)     ),
126: (0)
127: (0) def test_fastCopyAndTranspose(a):
128: (4)     with pytest.deprecated_call():
129: (8)         b = np.fastCopyAndTranspose(a)
130: (8)         assert_equal(b, a.T)
131: (8)         assert b.flags.owndata
132: (0) def test_array_astype():
133: (4)     a = np.arange(6, dtype='f4').reshape(2, 3)
134: (4)     b = a.astype('i4')
135: (4)     assert_equal(a, b)
136: (4)     assert_equal(b.dtype, np.dtype('i4'))
137: (4)     assert_equal(a.strides, b.strides)
138: (4)     b = a.T.astype('i4')
139: (4)     assert_equal(a.T, b)
140: (4)     assert_equal(b.dtype, np.dtype('i4'))
141: (4)     assert_equal(a.T.strides, b.strides)
142: (4)     b = a.astype('f4')
143: (4)     assert_equal(a, b)
144: (4)     assert_(not (a is b))
145: (4)     b = a.astype('f4', copy=False)
146: (4)     assert_(a is b)
147: (4)     b = a.astype('f4', order='F', copy=False)
148: (4)     assert_equal(a, b)
149: (4)     assert_(not (a is b))
150: (4)     assert_(b.flags.f_contiguous)
151: (4)     b = a.astype('f4', order='C', copy=False)

```

```

152: (4)             assert_equal(a, b)
153: (4)             assert_(a is b)
154: (4)             assert_(b.flags.c_contiguous)
155: (4)             b = a.astype('c8', casting='safe')
156: (4)             assert_equal(a, b)
157: (4)             assert_equal(b.dtype, np.dtype('c8'))
158: (4)             assert_raises(TypeError, a.astype, 'i4', casting='safe')
159: (4)             b = a.astype('f4', subok=0, copy=False)
160: (4)             assert_(a is b)
161: (4)             class MyNDArray(np.ndarray):
162: (8)                 pass
163: (4)                 a = np.array([[0, 1, 2], [3, 4, 5]], dtype='f4').view(MyNDArray)
164: (4)                 b = a.astype('f4', subok=True, copy=False)
165: (4)                 assert_(a is b)
166: (4)                 b = a.astype('i4', copy=False)
167: (4)                 assert_equal(a, b)
168: (4)                 assert_equal(type(b), MyNDArray)
169: (4)                 b = a.astype('f4', subok=False, copy=False)
170: (4)                 assert_equal(a, b)
171: (4)                 assert_(not (a is b))
172: (4)                 assert_(type(b) is not MyNDArray)
173: (4)                 a = np.array([b'a'*100], dtype='O')
174: (4)                 b = a.astype('S')
175: (4)                 assert_equal(a, b)
176: (4)                 assert_equal(b.dtype, np.dtype('S100'))
177: (4)                 a = np.array(['a'*100], dtype='O')
178: (4)                 b = a.astype('U')
179: (4)                 assert_equal(a, b)
180: (4)                 assert_equal(b.dtype, np.dtype('U100'))
181: (4)                 a = np.array([b'a'*10], dtype='O')
182: (4)                 b = a.astype('S')
183: (4)                 assert_equal(a, b)
184: (4)                 assert_equal(b.dtype, np.dtype('S10'))
185: (4)                 a = np.array(['a'*10], dtype='O')
186: (4)                 b = a.astype('U')
187: (4)                 assert_equal(a, b)
188: (4)                 assert_equal(b.dtype, np.dtype('U10'))
189: (4)                 a = np.array(123456789012345678901234567890, dtype='O').astype('S')
190: (4)                 assert_array_equal(a, np.array(b'1234567890' * 3, dtype='S30'))
191: (4)                 a = np.array(123456789012345678901234567890, dtype='O').astype('U')
192: (4)                 assert_array_equal(a, np.array('1234567890' * 3, dtype='U30'))
193: (4)                 a = np.array([123456789012345678901234567890], dtype='O').astype('S')
194: (4)                 assert_array_equal(a, np.array(b'1234567890' * 3, dtype='S30'))
195: (4)                 a = np.array([123456789012345678901234567890], dtype='O').astype('U')
196: (4)                 assert_array_equal(a, np.array('1234567890' * 3, dtype='U30'))
197: (4)                 a = np.array(123456789012345678901234567890, dtype='S')
198: (4)                 assert_array_equal(a, np.array(b'1234567890' * 3, dtype='S30'))
199: (4)                 a = np.array(123456789012345678901234567890, dtype='U')
200: (4)                 assert_array_equal(a, np.array('1234567890' * 3, dtype='U30'))
201: (4)                 a = np.array('a\u0140', dtype='U')
202: (4)                 b = np.ndarray(buffer=a, dtype='uint32', shape=2)
203: (4)                 assert_(b.size == 2)
204: (4)                 a = np.array([1000], dtype='i4')
205: (4)                 assert_raises(TypeError, a.astype, 'S1', casting='safe')
206: (4)                 a = np.array(1000, dtype='i4')
207: (4)                 assert_raises(TypeError, a.astype, 'U1', casting='safe')
208: (4)                 assert_raises(TypeError, a.astype)
209: (0) @pytest.mark.parametrize("dt", ["S", "U"])
210: (0) def test_array_astype_to_string_discovery_empty(dt):
211: (4)     arr = np.array([""], dtype=object)
212: (4)     assert arr.astype(dt).dtype.itemsize == np.dtype(f"{dt}1").itemsize
213: (4)     assert np.can_cast(arr, dt, casting="unsafe")
214: (4)     assert not np.can_cast(arr, dt, casting="same_kind")
215: (4)     assert np.can_cast("0", dt, casting="unsafe")
216: (0) @pytest.mark.parametrize("dt", ["d", "f", "S13", "U32"])
217: (0) def test_array_astype_to_void(dt):
218: (4)     dt = np.dtype(dt)
219: (4)     arr = np.array([], dtype=dt)
220: (4)     assert arr.astype("V").dtype.itemsize == dt.itemsize

```

```

221: (0)
222: (4)
223: (4)
224: (0)
225: (4)
226: (0)
227: (0)
228: (4)
229: (4)
230: (0)
231: (8)
232: (9)
233: (9)
234: (0)
235: (4)
236: (4)
237: (4)
238: (4)
239: (4)
240: (4)
241: (4)
242: (4)
243: (0)
244: (8)
245: (9)
246: (9)
247: (0)
248: (4)
249: (4)
250: (4)
251: (4)
252: (4)
253: (8)
254: (8)
255: (12)
256: (0)
257: (0)
258: (8)
259: (0)
260: (4)
261: (4)
262: (4)
263: (4)
264: (4)
265: (4)
266: (4)
267: (0)
268: (0)
269: (4)
270: (4)
271: (4)
272: (4)
273: (4)
274: (4)
275: (0)
276: (4)
277: (4)
278: (4)
279: (4)
280: (4)
281: (4)
282: (4)
283: (4)
284: (4)
285: (4)
286: (4)
287: (0)
288: (4)
289: (4)

def test_object_array_astype_to_void():
    arr = np.array([], dtype="O").astype("V")
    assert arr.dtype == "V8"
@pytest.mark.parametrize("t",
    np.sctypes['uint'] + np.sctypes['int'] + np.sctypes['float']
)
def test_array_astype_warning(t):
    a = np.array(10, dtype=np.complex_)
    assert_warns(np.ComplexWarning, a.astype, t)
@pytest.mark.parametrize(["dtype", "out_dtype"],
    [(np.bytes_, np.bool_),
     (np.str_, np.bool_),
     (np.dtype("S10,S9"), np.dtype("?,?"))]
)
def test_string_to_boolean_cast(dtype, out_dtype):
    """
    Currently, for `astype` strings are cast to booleans effectively by
    calling `bool(int(string))`. This is not consistent (see gh-9875) and
    will eventually be deprecated.
    """
    arr = np.array(["10", "10\0\0\0", "0\0\0", "0"], dtype=dtype)
    expected = np.array([True, True, False, False], dtype=out_dtype)
    assert_array_equal(arr.astype(out_dtype), expected)
@pytest.mark.parametrize(["dtype", "out_dtype"],
    [(np.bytes_, np.bool_),
     (np.str_, np.bool_),
     (np.dtype("S10,S9"), np.dtype("?,?"))]
)
def test_string_to_boolean_cast_errors(dtype, out_dtype):
    """
    These currently error out, since cast to integers fails, but should not
    error out in the future.
    """
    for invalid in ["False", "True", "", "\0", "non-empty"]:
        arr = np.array([invalid], dtype=dtype)
        with assert_raises(ValueError):
            arr.astype(out_dtype)
@pytest.mark.parametrize("str_type", [str, bytes, np.str_, np_unicode_])
@pytest.mark.parametrize("scalar_type",
    [np.complex64, np.complex128, np.clongdouble]
)
def test_string_to_complex_cast(str_type, scalar_type):
    value = scalar_type(b"1+3j")
    assert scalar_type(value) == 1+3j
    assert np.array([value], dtype=object).astype(scalar_type)[()] == 1+3j
    assert np.array(value).astype(scalar_type)[()] == 1+3j
    arr = np.zeros(1, dtype=scalar_type)
    arr[0] = value
    assert arr[0] == 1+3j
@pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
def test_none_to_nan_cast(dtype):
    arr = np.zeros(1, dtype=dtype)
    arr[0] = None
    assert np.isnan(arr)[0]
    assert np.isnan(np.array(None, dtype=dtype))[()]
    assert np.isnan(np.array([None], dtype=dtype))[0]
    assert np.isnan(np.array(None).astype(dtype))[()]
def test_copyto_fromscalar():
    a = np.arange(6, dtype='f4').reshape(2, 3)
    np.copyto(a, 1.5)
    assert_equal(a, 1.5)
    np.copyto(a.T, 2.5)
    assert_equal(a, 2.5)
    mask = np.array([[0, 1, 0], [0, 0, 1]], dtype='?')
    np.copyto(a, 3.5, where=mask)
    assert_equal(a, [[2.5, 3.5, 2.5], [2.5, 2.5, 3.5]])
    mask = np.array([[0, 1], [1, 1], [1, 0]], dtype='?')
    np.copyto(a.T, 4.5, where=mask)
    assert_equal(a, [[2.5, 4.5, 4.5], [4.5, 4.5, 3.5]])
def test_copyto():
    a = np.arange(6, dtype='i4').reshape(2, 3)
    np.copyto(a, [[3, 1, 5], [6, 2, 1]])

```

```

290: (4) assert_equal(a, [[3, 1, 5], [6, 2, 1]])
291: (4) np.copyto(a[:, :2], a[::-1, 1::-1])
292: (4) assert_equal(a, [[2, 6, 5], [1, 3, 1]])
293: (4) assert_raises(TypeError, np.copyto, a, 1.5)
294: (4) np.copyto(a, 1.5, casting='unsafe')
295: (4) assert_equal(a, 1)
296: (4) np.copyto(a, 3, where=[True, False, True])
297: (4) assert_equal(a, [[3, 1, 3], [3, 1, 3]])
298: (4) assert_raises(TypeError, np.copyto, a, 3.5, where=[True, False, True])
299: (4) np.copyto(a, 4.0, casting='unsafe', where=[[0, 1, 1], [1, 0, 0]])
300: (4) assert_equal(a, [[3, 4, 4], [4, 1, 3]])
301: (4) np.copyto(a[:, :2], a[::-1, 1::-1], where=[[0, 1], [1, 1]])
302: (4) assert_equal(a, [[3, 4, 4], [4, 3, 3]])
303: (4) assert_raises(TypeError, np.copyto, [1, 2, 3], [2, 3, 4])
304: (0) def test_copyto_permut():
305: (4)     pad = 500
306: (4)     l = [True] * pad + [True, True, True, True]
307: (4)     r = np.zeros(len(l)-pad)
308: (4)     d = np.ones(len(l)-pad)
309: (4)     mask = np.array(l)[pad:]
310: (4)     np.copyto(r, d, where=mask[::-1])
311: (4)     power = 9
312: (4)     d = np.ones(power)
313: (4)     for i in range(2**power):
314: (8)         r = np.zeros(power)
315: (8)         l = [(i & x) != 0 for x in range(power)]
316: (8)         mask = np.array(l)
317: (8)         np.copyto(r, d, where=mask)
318: (8)         assert_array_equal(r == 1, 1)
319: (8)         assert_equal(r.sum(), sum(l))
320: (8)         r = np.zeros(power)
321: (8)         np.copyto(r, d, where=mask[::-1])
322: (8)         assert_array_equal(r == 1, l[::-1])
323: (8)         assert_equal(r.sum(), sum(l))
324: (8)         r = np.zeros(power)
325: (8)         np.copyto(r[::-2], d[::-2], where=mask[::-2])
326: (8)         assert_array_equal(r[::-2] == 1, l[::-2])
327: (8)         assert_equal(r[::-2].sum(), sum(l[::-2]))
328: (8)         r = np.zeros(power)
329: (8)         np.copyto(r[::-2], d[::-2], where=mask[::-2])
330: (8)         assert_array_equal(r[::-2] == 1, l[::-2])
331: (8)         assert_equal(r[::-2].sum(), sum(l[::-2]))
332: (8)         for c in [0xFF, 0x7F, 0x02, 0x10]:
333: (12)             r = np.zeros(power)
334: (12)             mask = np.array(l)
335: (12)             imask = np.array(l).view(np.uint8)
336: (12)             imask[mask != 0] = c
337: (12)             np.copyto(r, d, where=mask)
338: (12)             assert_array_equal(r == 1, 1)
339: (12)             assert_equal(r.sum(), sum(l))
340: (4)             r = np.zeros(power)
341: (4)             np.copyto(r, d, where=True)
342: (4)             assert_equal(r.sum(), r.size)
343: (4)             r = np.ones(power)
344: (4)             d = np.zeros(power)
345: (4)             np.copyto(r, d, where=False)
346: (4)             assert_equal(r.sum(), r.size)
347: (0) def test_copy_order():
348: (4)     a = np.arange(24).reshape(2, 1, 3, 4)
349: (4)     b = a.copy(order='F')
350: (4)     c = np.arange(24).reshape(2, 1, 4, 3).swapaxes(2, 3)
351: (4)     def check_copy_result(x, y, ccontig, fcontig, strides=False):
352: (8)         assert_(not (x is y))
353: (8)         assert_equal(x, y)
354: (8)         assert_equal(res.flags.c_contiguous, ccontig)
355: (8)         assert_equal(res.flags.f_contiguous, fcontig)
356: (4)         assert_(a.flags.c_contiguous)
357: (4)         assert_(not a.flags.f_contiguous)
358: (4)         assert_(not b.flags.c_contiguous)

```

```

359: (4) assert_(b.flags.f_contiguous)
360: (4) assert_(not c.flags.c_contiguous)
361: (4) assert_(not c.flags.f_contiguous)
362: (4) res = a.copy(order='C')
363: (4) check_copy_result(res, a, ccontig=True, fcontig=False, strides=True)
364: (4) res = b.copy(order='C')
365: (4) check_copy_result(res, b, ccontig=True, fcontig=False, strides=False)
366: (4) res = c.copy(order='C')
367: (4) check_copy_result(res, c, ccontig=True, fcontig=False, strides=False)
368: (4) res = np.copy(a, order='C')
369: (4) check_copy_result(res, a, ccontig=True, fcontig=False, strides=True)
370: (4) res = np.copy(b, order='C')
371: (4) check_copy_result(res, b, ccontig=True, fcontig=False, strides=False)
372: (4) res = np.copy(c, order='C')
373: (4) check_copy_result(res, c, ccontig=True, fcontig=False, strides=False)
374: (4) res = a.copy(order='F')
375: (4) check_copy_result(res, a, ccontig=False, fcontig=True, strides=False)
376: (4) res = b.copy(order='F')
377: (4) check_copy_result(res, b, ccontig=False, fcontig=True, strides=True)
378: (4) res = c.copy(order='F')
379: (4) check_copy_result(res, c, ccontig=False, fcontig=True, strides=False)
380: (4) res = np.copy(a, order='F')
381: (4) check_copy_result(res, a, ccontig=False, fcontig=True, strides=False)
382: (4) res = np.copy(b, order='F')
383: (4) check_copy_result(res, b, ccontig=False, fcontig=True, strides=True)
384: (4) res = np.copy(c, order='F')
385: (4) check_copy_result(res, c, ccontig=False, fcontig=True, strides=False)
386: (4) res = a.copy(order='K')
387: (4) check_copy_result(res, a, ccontig=True, fcontig=False, strides=True)
388: (4) res = b.copy(order='K')
389: (4) check_copy_result(res, b, ccontig=False, fcontig=True, strides=True)
390: (4) res = c.copy(order='K')
391: (4) check_copy_result(res, c, ccontig=False, fcontig=False, strides=True)
392: (4) res = np.copy(a, order='K')
393: (4) check_copy_result(res, a, ccontig=True, fcontig=False, strides=True)
394: (4) res = np.copy(b, order='K')
395: (4) check_copy_result(res, b, ccontig=False, fcontig=True, strides=True)
396: (4) res = np.copy(c, order='K')
397: (4) check_copy_result(res, c, ccontig=False, fcontig=False, strides=True)
398: (0) def test_contiguous_flags():
399: (4)     a = np.ones((4, 4, 1))[:, ::2, :, :]
400: (4)     a.strides = a.strides[:2] + (-123,)
401: (4)     b = np.ones((2, 2, 1, 2, 2)).swapaxes(3, 4)
402: (4)     def check_contig(a, ccontig, fcontig):
403: (8)         assert_(a.flags.c_contiguous == ccontig)
404: (8)         assert_(a.flags.f_contiguous == fcontig)
405: (4)     check_contig(a, False, False)
406: (4)     check_contig(b, False, False)
407: (4)     check_contig(np.empty((2, 2, 0, 2, 2)), True, True)
408: (4)     check_contig(np.array([[1], [2]]), order='F'), True, True)
409: (4)     check_contig(np.empty((2, 2)), True, False)
410: (4)     check_contig(np.empty((2, 2), order='F'), False, True)
411: (4)     check_contig(np.array(a, copy=False), False, False)
412: (4)     check_contig(np.array(a, copy=False, order='C'), True, False)
413: (4)     check_contig(np.array(a, ndmin=4, copy=False, order='F'), False, True)
414: (4)     check_contig(a[0], True, True)
415: (4)     check_contig(a[None, ::4, ..., None], True, True)
416: (4)     check_contig(b[0, 0, ...], False, True)
417: (4)     check_contig(b[:, :, 0:0, :, :], True, True)
418: (4)     check_contig(a.ravel(), True, True)
419: (4)     check_contig(np.ones((1, 3, 1)).squeeze(), True, True)
420: (0) def test_broadcast_arrays():
421: (4)     a = np.array([(1, 2, 3)], dtype='u4,u4,u4')
422: (4)     b = np.array([(1, 2, 3), (4, 5, 6), (7, 8, 9)], dtype='u4,u4,u4')
423: (4)     result = np.broadcast_arrays(a, b)
424: (4)     assert_equal(result[0], np.array([(1, 2, 3), (1, 2, 3), (1, 2, 3)],
425: (4)                               dtype='u4,u4,u4'))
426: (4)     assert_equal(result[1], np.array([(1, 2, 3), (4, 5, 6), (7, 8, 9)],
427: (4)                               dtype='u4,u4,u4'))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

426: (0)
427: (8)
428: (9)
2.0]])))
429: (0)
430: (4)
431: (4)
432: (0)
433: (4)
434: (4)
435: (4)
436: (4)
437: (4)
438: (4)
439: (4)
440: (4)
441: (4)
442: (4)
443: (4)
444: (4)
445: (4)
446: (4)
447: (31)
448: (4)
449: (4)
450: (18)

-----

```

## File 75 - test\_argparse.py:

```

1: (0)
2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (4)
8: (8)
9: (8)
10: (0)
11: (0)
12: (0)
13: (0)
14: (0)
def test_invalid_integers():
    def func(arg1, /, arg2, *, arg3):
        i = integer(arg1) # reproducing the 'i' parsing in Python.
        return None
    """
    Tests for the private NumPy argument parsing functionality.
    They mainly exists to ensure good test coverage without having to try the
    weirder cases on actual numpy functions but test them in one place.
    The test function is defined in C to be equivalent to (errors may not always
    match exactly, and could be adjusted):
    """
    import pytest
    import numpy as np
    from numpy.core._multiarray_tests import argparse_example_function as func
    def test_missing_arguments():
        with pytest.raises(TypeError,
                           match="missing required positional argument 0"):
            func()
        with pytest.raises(OverflowError):
            func(2**100)
    def test_too_many_positional():
        with pytest.raises(TypeError,
                           match="takes from 2 to 3 positional arguments but 4 were given"):
            func(1, 2, 3, 4)
    def test_multiple_values():
        with pytest.raises(TypeError,
                           match=r"given by name \('arg2'\) and position \((position 1\)\)"):
            func(1, 2, arg2=3)
    def test_string_fallbacks():

```

```
39: (4)                 arg2 = np.str_("arg2")
40: (4)                 missing_arg = np.str_("missing_arg")
41: (4)                 func(1, **{arg2: 3})
42: (4)                 with pytest.raises(TypeError,
43: (12)                     match="got an unexpected keyword argument 'missing_arg'"):
44: (8)                         func(2, **{missing_arg: 3})
```

## File 76 - test\_arraymethod.py:

```

1: (0)
2: (0)    """
3: (0)        This file tests the generic aspects of ArrayMethod. At the time of writing
4: (0)        this is private API, but when added, public API may be added here.
5: (0)    """
6: (0)        from __future__ import annotations
7: (0)        import sys
8: (0)        import types
9: (0)        from typing import Any
10: (0)       import pytest
11: (0)       import numpy as np
12: (0)       from numpy.core._multiarray_umath import _get_castingimpl as get_castingimpl
13: (4)       class TestResolveDescriptors:
14: (4)           method = get_castingimpl(type(np.dtype("d")), type(np.dtype("f")))
15: (8)           @pytest.mark.parametrize("args", [
16: (8)               (True,), # Not a tuple.
17: (8)               ((None,)), # Too few elements
18: (8)               ((None, None, None)), # Too many
19: (8)               ((None, None),), # Input dtype is None, which is invalid.
20: (8)               ((np.dtype("d"), True),), # Output dtype is not a dtype
21: (8)               ((np.dtype("f"), None),), # Input dtype does not match method
22: (4)           ])
23: (4)           def test_invalid_arguments(self, args):
24: (8)               with pytest.raises(TypeError):
25: (8)                   self.method._resolve_descriptors(*args)
26: (4)       class TestSimpleStridedCall:
27: (4)           method = get_castingimpl(type(np.dtype("d")), type(np.dtype("f")))
28: (4)           @pytest.mark.parametrize(["args", "error"], [
29: (8)               ((True,), TypeError), # Not a tuple
30: (8)               (((None,)), TypeError), # Too few elements
31: (8)               ((None, None), TypeError), # Inputs are not arrays.
32: (8)               (((None, None, None)), TypeError), # Too many
33: (8)               (((np.arange(3), np.arange(3)),), TypeError), # Incorrect dtypes
34: (9)               (((np.ones(3, dtype=">d"), np.ones(3, dtype=<f"))),
35: (8)                   TypeError), # Does not support byte-swapping
36: (9)               (((np.ones((2, 2), dtype="d"), np.ones((2, 2), dtype="f"))),
37: (8)                   ValueError), # not 1-D
38: (10)              (((np.ones(3, dtype="d"), np.ones(4, dtype="f"))),
39: (8)                   ValueError), # different length
40: (11)              (((np.frombuffer(b"\x00" * 3 * 2, dtype="d"),
41: (9)                  np.frombuffer(b"\x00" * 3, dtype="f"))),
42: (4)                   ValueError), # output not writeable
43: (4)           ])
44: (8)           def test_invalid_arguments(self, args, error):
45: (8)               with pytest.raises(error):
46: (8)                   self.method._simple_strided_call(*args)
47: (4)           @pytest.mark.parametrize(
48: (8)               "cls", [np.ndarray, np.recarray, np.chararray, np.matrix, np.memmap]
49: (0)           )
50: (4)       class TestClassGetItem:
51: (4)           def test_class_getitem(self, cls: type[np.ndarray]) -> None:
52: (8)               """Test `ndarray.__class_getitem__`."""
53: (8)               alias = cls[Any, Any]
54: (8)               assert isinstance(alias, types.GenericAlias)
55: (8)               assert alias.__origin__ is cls
56: (4)           @pytest.mark.parametrize("arg_len", range(4))
57: (4)           def test_subscript_tup(self, cls: type[np.ndarray], arg_len: int) -> None:
58: (8)               arg_tup = (Any,) * arg_len
59: (8)               if arg_len in (1, 2):

```

```

59: (12)             assert cls[arg_tup]
60: (8)         else:
61: (12)             match = f"Too {'few' if arg_len == 0 else 'many'} arguments"
62: (12)             with pytest.raises(TypeError, match=match):
63: (16)                 cls[arg_tup]

-----

```

## File 77 - test\_arrayprint.py:

```

1: (0)             import sys
2: (0)             import gc
3: (0)             from hypothesis import given
4: (0)             from hypothesis.extra import numpy as hynp
5: (0)             import pytest
6: (0)             import numpy as np
7: (0)             from numpy.testing import (
8: (4)                 assert_, assert_equal, assert_raises, assert_warns, HAS_REFCOUNT,
9: (4)                 assert_raises_regex,
10: (4)             )
11: (0)             from numpy.core.arrayprint import _typelessdata
12: (0)             import textwrap
13: (0)             class TestArrayRepr:
14: (4)                 def test_nan_inf(self):
15: (8)                     x = np.array([np.nan, np.inf])
16: (8)                     assert_equal(repr(x), 'array([nan, inf])')
17: (4)                 def test_subclass(self):
18: (8)                     class sub(np.ndarray): pass
19: (8)                     x1d = np.array([1, 2]).view(sub)
20: (8)                     assert_equal(repr(x1d), 'sub([1, 2])')
21: (8)                     x2d = np.array([[1, 2], [3, 4]]).view(sub)
22: (8)                     assert_equal(repr(x2d),
23: (12)                         'sub([[1, 2],\n'
24: (12)                           '[3, 4]])')
25: (8)                     xstruct = np.ones((2,2), dtype=[('a', '<i4')]).view(sub)
26: (8)                     assert_equal(repr(xstruct),
27: (12)                         "sub([(1,), (1)],\n"
28: (12)                           "[1,), (1)]], dtype=[('a', '<i4')])"
29: (8)                 )
30: (4)             @pytest.mark.xfail(reason="See gh-10544")
31: (4)             def test_object_subclass(self):
32: (8)                 class sub(np.ndarray):
33: (12)                     def __new__(cls, inp):
34: (16)                         obj = np.asarray(inp).view(cls)
35: (16)                         return obj
36: (12)                     def __getitem__(self, ind):
37: (16)                         ret = super().__getitem__(ind)
38: (16)                         return sub(ret)
39: (8)                     x = sub([None, None])
40: (8)                     assert_equal(repr(x), 'sub([None, None], dtype=object)')
41: (8)                     assert_equal(str(x), '[None None]')
42: (8)                     x = sub([None, sub([None, None])])
43: (8)                     assert_equal(repr(x),
44: (12)                         'sub([None, sub([None, None], dtype=object)], dtype=object)')
45: (8)                         assert_equal(str(x), '[None sub([None, None], dtype=object)]')
46: (4)             def test_0d_object_subclass(self):
47: (8)                 class sub(np.ndarray):
48: (12)                     def __new__(cls, inp):
49: (16)                         obj = np.asarray(inp).view(cls)
50: (16)                         return obj
51: (12)                     def __getitem__(self, ind):
52: (16)                         ret = super().__getitem__(ind)
53: (16)                         return sub(ret)
54: (8)                     x = sub(1)
55: (8)                     assert_equal(repr(x), 'sub(1)')
56: (8)                     assert_equal(str(x), '1')
57: (8)                     x = sub([1, 1])
58: (8)                     assert_equal(repr(x), 'sub([1, 1])')
59: (8)                     assert_equal(str(x), '[1 1]')

```

```

60: (8)           x = sub(None)
61: (8)           assert_equal(repr(x), 'sub(None, dtype=object)')
62: (8)           assert_equal(str(x), 'None')
63: (8)           y = sub(None)
64: (8)           x[()] = y
65: (8)           y[()] = x
66: (8)           assert_equal(repr(x),
67: (12)             'sub(sub(sub(..., dtype=object), dtype=object), dtype=object)')
68: (8)           assert_equal(str(x), '...')
69: (8)           x[()] = 0 # resolve circular references for garbage collector
70: (8)           x = sub(None)
71: (8)           x[()] = sub(None)
72: (8)           assert_equal(repr(x), 'sub(sub(None, dtype=object), dtype=object)')
73: (8)           assert_equal(str(x), 'None')
74: (8)           class DuckCounter(np.ndarray):
75: (12)             def __getitem__(self, item):
76: (16)               result = super().__getitem__(item)
77: (16)               if not isinstance(result, DuckCounter):
78: (20)                 result = result[...].view(DuckCounter)
79: (16)               return result
80: (12)             def to_string(self):
81: (16)               return {0: 'zero', 1: 'one', 2: 'two'}.get(self.item(),
82: ('many')
83: (12)
84: (16)
85: (20)
86: (16)
87: (20)
88: (8)           dc = np.arange(5).view(DuckCounter)
89: (8)           assert_equal(str(dc), "[zero one two many many]")
90: (8)           assert_equal(str(dc[0]), "zero")
91: (4)           def test_self_containing(self):
92: (8)             arr0d = np.array(None)
93: (8)             arr0d[()] = arr0d
94: (8)             assert_equal(repr(arr0d),
95: (12)               'array(array(..., dtype=object), dtype=object)')
96: (8)             arr0d[()] = 0 # resolve recursion for garbage collector
97: (8)             arr1d = np.array([None, None])
98: (8)             arr1d[1] = arr1d
99: (8)             assert_equal(repr(arr1d),
100: (12)               'array([None, array(..., dtype=object)], dtype=object)')
101: (8)             arr1d[1] = 0 # resolve recursion for garbage collector
102: (8)             first = np.array(None)
103: (8)             second = np.array(None)
104: (8)             first[()] = second
105: (8)             second[()] = first
106: (8)             assert_equal(repr(first),
107: (12)               'array(array(array(..., dtype=object), dtype=object),
108: ('many')
109: (8)             first[()] = 0 # resolve circular references for garbage collector
110: (4)           def testContainingList(self):
111: (8)             arr1d = np.array([None, None])
112: (8)             arr1d[0] = [1, 2]
113: (8)             arr1d[1] = [3]
114: (8)             assert_equal(repr(arr1d),
115: (12)               'array([list([1, 2]), list([3])], dtype=object)')
116: (4)           def test_void_scalar_recursion(self):
117: (8)             repr(np.void(b'test')) # RecursionError ?
118: (4)           def test_fieldless_structured(self):
119: (8)             no_fields = np.dtype([])
120: (8)             arr_no_fields = np.empty(4, dtype=no_fields)
121: (0)             assert_equal(repr(arr_no_fields), 'array([((), (), (), (), ()], dtype=[])')
122: (4)           class TestComplexArray:
123: (8)             def test__str__(self):
124: (8)               rvals = [0, 1, -1, np.inf, -np.inf, np.nan]
125: (8)               cvals = [complex(rp, ip) for rp in rvals for ip in rvals]
126: (8)               dtypes = [np.complex64, np.cdouble, np.clongdouble]

```

```

127: (8)           wanted = [
128: (12)          '[0.+0.j]', '[0.+0.j]', '[0.+0.j]',
129: (12)          '[0.+1.j]', '[0.+1.j]', '[0.+1.j]',
130: (12)          '[0.-1.j]', '[0.-1.j]', '[0.-1.j]',
131: (12)          '[0.+infj]', '[0.+infj]', '[0.+infj]',
132: (12)          '[0.-infj]', '[0.-infj]', '[0.-infj]',
133: (12)          '[0.+nanj]', '[0.+nanj]', '[0.+nanj]',
134: (12)          '[1.+0.j]', '[1.+0.j]', '[1.+0.j]',
135: (12)          '[1.+1.j]', '[1.+1.j]', '[1.+1.j]',
136: (12)          '[1.-1.j]', '[1.-1.j]', '[1.-1.j]',
137: (12)          '[1.+infj]', '[1.+infj]', '[1.+infj]',
138: (12)          '[1.-infj]', '[1.-infj]', '[1.-infj]',
139: (12)          '[1.+nanj]', '[1.+nanj]', '[1.+nanj]',
140: (12)          '[-1.+0.j]', '[-1.+0.j]', '[-1.+0.j]',
141: (12)          '[-1.+1.j]', '[-1.+1.j]', '[-1.+1.j]',
142: (12)          '[-1.-1.j]', '[-1.-1.j]', '[-1.-1.j]',
143: (12)          '[-1.+infj]', '[-1.+infj]', '[-1.+infj]',
144: (12)          '[-1.-infj]', '[-1.-infj]', '[-1.-infj]',
145: (12)          '[-1.+nanj]', '[-1.+nanj]', '[-1.+nanj]',
146: (12)          '[inf+0.j]', '[inf+0.j]', '[inf+0.j]',
147: (12)          '[inf+1.j]', '[inf+1.j]', '[inf+1.j]',
148: (12)          '[inf-1.j]', '[inf-1.j]', '[inf-1.j]',
149: (12)          '[inf+infj]', '[inf+infj]', '[inf+infj]',
150: (12)          '[inf-infj]', '[inf-infj]', '[inf-infj]',
151: (12)          '[inf+nanj]', '[inf+nanj]', '[inf+nanj]',
152: (12)          '[-inf+0.j]', '[-inf+0.j]', '[-inf+0.j]',
153: (12)          '[-inf+1.j]', '[-inf+1.j]', '[-inf+1.j]',
154: (12)          '[-inf-1.j]', '[-inf-1.j]', '[-inf-1.j]',
155: (12)          '[-inf+infj]', '[-inf+infj]', '[-inf+infj]',
156: (12)          '[-inf-infj]', '[-inf-infj]', '[-inf-infj]',
157: (12)          '[-inf+nanj]', '[-inf+nanj]', '[-inf+nanj]',
158: (12)          '[nan+0.j]', '[nan+0.j]', '[nan+0.j]',
159: (12)          '[nan+1.j]', '[nan+1.j]', '[nan+1.j]',
160: (12)          '[nan-1.j]', '[nan-1.j]', '[nan-1.j]',
161: (12)          '[nan+infj]', '[nan+infj]', '[nan+infj]',
162: (12)          '[nan-infj]', '[nan-infj]', '[nan-infj]',
163: (12)          '[nan+nanj]', '[nan+nanj]', '[nan+nanj]']

164: (8)           for res, val in zip(actual, wanted):
165: (12)             assert_equal(res, val)

166: (0)            class TestArray2String:
167: (4)              def test_basic(self):
168: (8)                """Basic test of array2string."""
169: (8)                a = np.arange(3)
170: (8)                assert_(np.array2string(a) == '[0 1 2]')
171: (8)                assert_(np.array2string(a, max_line_width=4, legacy='1.13') == '[0 1\n2]')
172: (8)                assert_(np.array2string(a, max_line_width=4) == '[0\n1\n2]')
173: (4)              def test_unexpected_kwarg(self):
174: (8)                with assert_raises_regex(TypeError, 'nonsense'):
175: (12)                  np.array2string(np.array([1, 2, 3]),
176: (28)                      nonsense=None)
177: (4)              def test_format_function(self):
178: (8)                """Test custom format function for each element in array."""
179: (8)                def _format_function(x):
180: (12)                  if np.abs(x) < 1:
181: (16)                      return '.'
182: (12)                  elif np.abs(x) < 2:
183: (16)                      return 'o'
184: (12)                  else:
185: (16)                      return 'O'
186: (8)                x = np.arange(3)
187: (8)                x_hex = "[0x0 0x1 0x2]"
188: (8)                x_oct = "[0o0 0o1 0o2]"
189: (8)                assert_(np.array2string(x, formatter={'all':_format_function}) ==
190: (16)                      "[. o O]")
191: (8)                assert_(np.array2string(x, formatter={'int_kind':_format_function}) ==
192: (16)                      "[. o O]")
193: (8)                assert_(np.array2string(x, formatter={'all':lambda x: "%4f" % x}) ==
194: (16)                      "[0.0000 1.0000 2.0000]")

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

195: (8) assert_equal(np.array2string(x, formatter={'int':lambda x: hex(x)}),
196: (16)         x_hex)
197: (8) assert_equal(np.array2string(x, formatter={'int':lambda x: oct(x)}),
198: (16)         x_oct)
199: (8) x = np.arange(3.)
200: (8) assert_(np.array2string(x, formatter={'float_kind':lambda x: "%.2f" % x}) ==
201: (16)         "[0.00 1.00 2.00]")
202: (8) assert_(np.array2string(x, formatter={'float':lambda x: "% .2f" % x}) ==
203: (16)         "[0.00 1.00 2.00]")
204: (8) s = np.array(['abc', 'def'])
205: (8) assert_(np.array2string(s, formatter={'numpystr':lambda s: s*2}) ==
206: (16)         '[abcabc defdef]')
207: (4) def test_structure_format_mixed(self):
208: (8) dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
209: (8) x = np.array([('Sarah', (8.0, 7.0)), ('John', (6.0, 7.0))], dtype=dt)
210: (8) assert_equal(np.array2string(x),
211: (16)         "[('Sarah', [8., 7.]) ('John', [6., 7.])]")
212: (8) np.set_printoptions(legacy='1.13')
213: (8) try:
214: (12)     A = np.zeros(shape=10, dtype=[("A", "M8[s]")])
215: (12)     A[5:]._fill(np.datetime64('NaT'))
216: (12)     assert_equal(
217: (16)         np.array2string(A),
218: (16)         textwrap.dedent("""\
219: ('1970-01-01T00:00:00',) ('1970-01-01T00:00:00',) ('1970-01-
01T00:00:00',)
220: (17) ('NaT',)
221: (17) ('NaT',) ('NaT',) ('NaT',)]""")
222: (12) )
223: (8) finally:
224: (12)     np.set_printoptions(legacy=False)
225: (8) assert_equal(
226: (12)     np.array2string(A),
227: (12)     textwrap.dedent("""\
228: ('1970-01-01T00:00:00',) ('1970-01-01T00:00:00',)
229: (13) ('1970-01-01T00:00:00',) ('1970-01-01T00:00:00',)
230: (13) ('1970-01-01T00:00:00',) (          'NaT',)
231: (13) (          'NaT',) (          'NaT',)
232: (13) (          'NaT',) (          'NaT',)]""")
233: (8) )
234: (8) A = np.full(10, 123456, dtype=[("A", "m8[s]")])
235: (8) A[5:]._fill(np.datetime64('NaT'))
236: (8) assert_equal(
237: (12)     np.array2string(A),
238: (12)     textwrap.dedent("""\
239: [(123456,) (123456,) (123456,) (123456,) (123456,) ( 'NaT',) (
'NaT',)
240: (13) ( 'NaT',) ( 'NaT',) ( 'NaT',)]"""))
241: (8) )
242: (4) def test_structure_format_int(self):
243: (8) struct_int = np.array([(1, -1), (123, 1)], dtype=[('B', 'i4',
2)]) )
244: (8) assert_equal(np.array2string(struct_int),
245: (16)         "[([ 1, -1],) ([123, 1],)]")
246: (8) struct_2dint = np.array([(0, 1), (2, 3)], dtype=[('B', 'i4', (2, 2))])
247: (16) assert_equal(np.array2string(struct_2dint),
248: (8)         "[([ 0, 1], [ 2, 3]), ([12, 0], [ 0, 0]),]")
249: (16) )
250: (4) def test_structure_format_float(self):
251: (8) array_scalar = np.array(
252: (16)         (1., 2.1234567890123456789, 3.), dtype=('f8,f8,f8'))
253: (8) assert_equal(np.array2string(array_scalar), "(1., 2.12345679, 3.)")
254: (4) def test_unstructured_void_repr(self):
255: (8) a = np.array([27, 91, 50, 75, 7, 65, 10, 8,
256: (22)         27, 91, 51, 49, 109, 82, 101, 100], dtype='u1').view('V8')
257: (8) assert_equal(repr(a[0]), r"void(b'\x1B\x5B\x32\x4B\x07\x41\x0A\x08')")
```

```

258: (8) assert_equal(str(a[0]), r"b'\x1B\x5B\x32\x4B\x07\x41\x0A\x08'")
259: (8) assert_equal(repr(a),
260: (12)     r"array([b'\x1B\x5B\x32\x4B\x07\x41\x0A\x08', " " \n"
261: (12)         r"          b'\x1B\x5B\x33\x31\x6D\x52\x52\x65\x64'], dtype='|V8')")
262: (8) assert_equal(eval(repr(a)), vars(np)), a)
263: (8) assert_equal(eval(repr(a[0])), vars(np)), a[0])
264: (4) def test_edgeitems_kwarg(self):
265: (8)     arr = np.zeros(3, int)
266: (8)     assert_equal(
267: (12)         np.array2string(arr, edgeitems=1, threshold=0),
268: (12)         "[0 ... 0]")
269: (8)
270: (4) def test_summarize_1d(self):
271: (8)     A = np.arange(1001)
272: (8)     strA = '[ 0 1 2 ... 998 999 1000]'
273: (8)     assert_equal(str(A), strA)
274: (8)     reprA = 'array([ 0, 1, 2, ..., 998, 999, 1000])'
275: (8)     assert_equal(repr(A), reprA)
276: (4) def test_summarize_2d(self):
277: (8)     A = np.arange(1002).reshape(2, 501)
278: (8)     strA = '[[ 0 1 2 ... 498 499 500]\n' \
279: (15)         ' [ 501 502 503 ... 999 1000 1001]]'
280: (8)     assert_equal(str(A), strA)
281: (8)     reprA = 'array([[ 0, 1, 2, ..., 498, 499, 500],\n' \
282: (16)         ' [ 501, 502, 503, ..., 999, 1000, 1001]])'
283: (8)     assert_equal(repr(A), reprA)
284: (4) def test_summarize_structure(self):
285: (8)     A = (np.arange(2002, dtype=<i8>).reshape(2, 1001)
286: (13)         .view([('i', '<i8', (1001,))]))
287: (8)     strA = ("[[([ 0, 1, 2, ..., 998, 999, 1000],)],\n" \
288: (16)         " [[(1001, 1002, 1003, ..., 1999, 2000, 2001),]]")
289: (8)     assert_equal(str(A), strA)
290: (8)     reprA = ("array([[([ 0, 1, 2, ..., 998, 999, 1000],)],\n" \
291: (17)         " [[([1001, 1002, 1003, ..., 1999, 2000, 2001],)],\n" \
292: (17)         "         dtype=[('i', '<i8', (1001,))]])")
293: (8)     assert_equal(repr(A), reprA)
294: (8)     B = np.ones(2002, dtype=>i8>).view([('i', '>i8', (2, 1001))])
295: (8)     strB = "[([[[1, 1, 1, ..., 1, 1, 1], [1, 1, 1, ..., 1, 1, 1]]],)]"
296: (8)     assert_equal(str(B), strB)
297: (8)     reprB = (
298: (12)         "array([([[[1, 1, 1, ..., 1, 1, 1], [1, 1, 1, ..., 1, 1, 1]],)],\n" \
299: (12)             "         dtype=[('i', '>i8', (2, 1001))])"
300: (8)
301: (8)     )
302: (8)     assert_equal(repr(B), reprB)
303: (8)     C = (np.arange(22, dtype=<i8>).reshape(2, 11)
304: (13)         .view([('i1', '<i8', ('i10', '<i8', (10,)))]))
305: (8)     strC = "[[( 0, [ 1, ..., 10])]\n [(11, [12, ..., 21])]]"
306: (4)     assert_equal(np.array2string(C, threshold=1, edgeitems=1), strC)
307: (8)     def test_linewidth(self):
308: (8)         a = np.full(6, 1)
309: (12)         def make_str(a, width, **kw):
310: (8)             return np.array2string(a, separator="", max_line_width=width,
311: (8)             **kw)
312: (8)             assert_equal(make_str(a, 8, legacy='1.13'), '[111111]')
313: (8)             assert_equal(make_str(a, 7, legacy='1.13'), '[111111]')
314: (52)             assert_equal(make_str(a, 5, legacy='1.13'), '[111\n' \
315: (8)                 ' 11]')
316: (37)             assert_equal(make_str(a, 8), '[111111]')
317: (8)             assert_equal(make_str(a, 7), '[11111\n' \
318: (37)                 ' 1]')
319: (8)             assert_equal(make_str(a, 5), '[111\n' \
320: (8)                 ' 111]')
321: (8)             b = a[None,None,:]
322: (8)             assert_equal(make_str(b, 12, legacy='1.13'), '[[[111111]]]')
323: (53)             assert_equal(make_str(b, 9, legacy='1.13'), '[[[111111]]]')
324: (8)             assert_equal(make_str(b, 8, legacy='1.13'), '[[[11111\n' \
325: (8)                 ' 1]]]')
324: (8)             assert_equal(make_str(b, 12), '[[[111111]]]')
325: (8)             assert_equal(make_str(b, 9), '[[[111\n' \

```

```

326: (38)                                ' 111]]])')
327: (8)       assert_equal(make_str(b, 8), '[[[11\n'
328: (38)                                ' 11\n'
329: (38)                                ' 11]]])')
330: (4)        def test_wide_element(self):
331: (8)          a = np.array(['xxxxx'])
332: (8)          assert_equal(
333: (12)            np.array2string(a, max_line_width=5),
334: (12)            "['xxxxx']")
335: (8)        )
336: (8)        assert_equal(
337: (12)          np.array2string(a, max_line_width=5, legacy='1.13'),
338: (12)          "[ 'xxxxx']")
339: (8)      )
340: (4)        def test_multiline_repr(self):
341: (8)          class Multiline:
342: (12)            def __repr__(self):
343: (16)              return "Line 1\nLine 2"
344: (8)          a = np.array([[None, MultiLine()], [MultiLine(), None]])
345: (8)          assert_equal(
346: (12)            np.array2string(a),
347: (12)            '[[None Line 1\n'
348: (12)            '    Line 2]\n'
349: (12)            ' [Line 1\n'
350: (12)            '    Line 2 None]]'
351: (8)        )
352: (8)        assert_equal(
353: (12)          np.array2string(a, max_line_width=5),
354: (12)          '[[None\n'
355: (12)          '    Line 1\n'
356: (12)          '    Line 2]\n'
357: (12)          ' [Line 1\n'
358: (12)          '    Line 2\n'
359: (12)          '    None]]'
360: (8)      )
361: (8)      assert_equal(
362: (12)        repr(a),
363: (12)        'array([[None, Line 1\n'
364: (12)        '           Line 2],\n'
365: (12)        '           [Line 1\n'
366: (12)        '             Line 2, None]], dtype=object)'
367: (8)    )
368: (8)    class MultiLineLong:
369: (12)      def __repr__(self):
370: (16)        return "Line 1\nLooooooooooooongestLine2\nLongerLine 3"
371: (8)    a = np.array([[None, MultiLineLong()], [MultiLineLong(), None]])
372: (8)    assert_equal(
373: (12)      repr(a),
374: (12)      'array([[None, Line 1\n'
375: (12)      '           LooooooooooooongestLine2\n'
376: (12)      '           LongerLine 3           ],\n'
377: (12)      '           [Line 1\n'
378: (12)      '             LooooooooooooongestLine2\n'
379: (12)      '             LongerLine 3           , None]], dtype=object)'
380: (8)    )
381: (8)    assert_equal(
382: (12)      np.array_repr(a, 20),
383: (12)      'array([[None,\n'
384: (12)      '           Line 1\n'
385: (12)      '           LooooooooooooongestLine2\n'
386: (12)      '           LongerLine 3           ],\n'
387: (12)      '           [Line 1\n'
388: (12)      '             LooooooooooooongestLine2\n'
389: (12)      '             LongerLine 3           ,\n'
390: (12)      '             None]],\n'
391: (12)      '             dtype=object)'
392: (8)    )
393: (4)      def test_nested_array_repr(self):
394: (8)        a = np.empty((2, 2), dtype=object)

```

```

395: (8)             a[0, 0] = np.eye(2)
396: (8)             a[0, 1] = np.eye(3)
397: (8)             a[1, 0] = None
398: (8)             a[1, 1] = np.ones((3, 1))
399: (8)             assert_equal(
400: (12)                 repr(a),
401: (12)                 'array([[array([[1., 0.],\n'
402: (12)                     '          [0., 1.]])], array([[1., 0., 0.],\n'
403: (12)                         '          [0., 1., 0.],\n'
404: (12)                             '          [0., 0., 1.]])],\n'
405: (12)                               [None, array([[1.],\n'
406: (12)                                   '          [1.],\n'
407: (12)                                       [1.]])]], dtype=object)'
408: (8)             )
409: (4)             @given(hynp.from_dtype(np.dtype("U")))
410: (4)             def test_any_text(self, text):
411: (8)                 a = np.array([text, text, text])
412: (8)                 assert_equal(a[0], text)
413: (8)                 expected_repr = "[{0!r} {0!r}\n {0!r}]".format(text)
414: (8)                 result = np.array2string(a, max_line_width=len(repr(text)) * 2 + 3)
415: (8)                 assert_equal(result, expected_repr)
416: (4)             @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
417: (4)             def test_refcount(self):
418: (8)                 gc.disable()
419: (8)                 a = np.arange(2)
420: (8)                 r1 = sys.getrefcount(a)
421: (8)                 np.array2string(a)
422: (8)                 np.array2string(a)
423: (8)                 r2 = sys.getrefcount(a)
424: (8)                 gc.collect()
425: (8)                 gc.enable()
426: (8)                 assert_(r1 == r2)
427: (0)             class TestPrintOptions:
428: (4)                 """Test getting and setting global print options."""
429: (4)                 def setup_method(self):
430: (8)                     self.oldopts = np.get_printoptions()
431: (4)                 def teardown_method(self):
432: (8)                     np.set_printoptions(**self.oldopts)
433: (4)                 def test_basic(self):
434: (8)                     x = np.array([1.5, 0, 1.234567890])
435: (8)                     assert_equal(repr(x), "array([1.5      , 0.        , 1.23456789])")
436: (8)                     np.set_printoptions(precision=4)
437: (8)                     assert_equal(repr(x), "array([1.5   , 0.    , 1.2346])")
438: (4)                 def test_precision_zero(self):
439: (8)                     np.set_printoptions(precision=0)
440: (8)                     for values, string in (
441: (16)                         ([0.], "0."), ([.3], "0."), ([-.3], "-0."), ([.7], "1."),
442: (16)                         ([1.5], "2."), ([-1.5], "-2."), ([-15.34], "-15."),
443: (16)                         ([100.], "100."), ([.2, -1, 122.51], " 0., -1., 123."),
444: (16)                         ([0], "0"), ([-12], "-12"), ([complex(.3, -.7)], "0.-1.j")):
445: (12)                         x = np.array(values)
446: (12)                         assert_equal(repr(x), "array[%s]" % string)
447: (4)                 def test_formatter(self):
448: (8)                     x = np.arange(3)
449: (8)                     np.set_printoptions(formatter={'all':lambda x: str(x-1)})
450: (8)                     assert_equal(repr(x), "array([-1, 0, 1])")
451: (4)                 def test_formatter_reset(self):
452: (8)                     x = np.arange(3)
453: (8)                     np.set_printoptions(formatter={'all':lambda x: str(x-1)})
454: (8)                     assert_equal(repr(x), "array([-1, 0, 1])")
455: (8)                     np.set_printoptions(formatter={'int':None})
456: (8)                     assert_equal(repr(x), "array([0, 1, 2])")
457: (8)                     np.set_printoptions(formatter={'all':lambda x: str(x-1)})
458: (8)                     assert_equal(repr(x), "array([-1, 0, 1])")
459: (8)                     np.set_printoptions(formatter={'all':None})
460: (8)                     assert_equal(repr(x), "array([0, 1, 2])")
461: (8)                     np.set_printoptions(formatter={'int':lambda x: str(x-1)})
462: (8)                     assert_equal(repr(x), "array([-1, 0, 1])")
463: (8)                     np.set_printoptions(formatter={'int_kind':None})

```

```

464: (8) assert_equal(repr(x), "array([0, 1, 2])")
465: (8) x = np.arange(3.)
466: (8) np.set_printoptions(formatter={'float':lambda x: str(x-1)})
467: (8) assert_equal(repr(x), "array([-1.0, 0.0, 1.0])")
468: (8) np.set_printoptions(formatter={'float_kind':None})
469: (8) assert_equal(repr(x), "array([0., 1., 2.])")
470: (4) def test_0d_arrays(self):
471: (8) assert_equal(str(np.array('cafÃ©', '<U4')), 'cafÃ©')
472: (8) assert_equal(repr(np.array('cafÃ©', '<U4')),
473: (21) "array('cafÃ©', dtype='<U4')")
474: (8) assert_equal(str(np.array('test', np.str_)), 'test')
475: (8) a = np.zeros(1, dtype=[('a', '<i4', (3,))])
476: (8) assert_equal(str(a[0]), '([0, 0, 0],)')
477: (8) assert_equal(repr(np.datetime64('2005-02-25')[...]),
478: (21) "array('2005-02-25', dtype='datetime64[D]')")
479: (8) assert_equal(repr(np.timedelta64('10', 'Y')[...]),
480: (21) "array(10, dtype='timedelta64[Y]')")
481: (8) x = np.array(1)
482: (8) np.set_printoptions(formatter={'all':lambda x: "test"})
483: (8) assert_equal(repr(x), "array(test)")
484: (8) assert_equal(str(x), "1")
485: (8) assert_warns(DeprecationWarning, np.array2string,
486: (41) np.array(1.), style=repr)
487: (8) np.array2string(np.array(1.), style=repr, legacy='1.13')
488: (8) np.array2string(np.array(1.), legacy='1.13')
489: (4) def test_float_spacing(self):
490: (8) x = np.array([1., 2., 3.])
491: (8) y = np.array([1., 2., -10.])
492: (8) z = np.array([100., 2., -1.])
493: (8) w = np.array([-100., 2., 1.])
494: (8) assert_equal(repr(x), 'array([1., 2., 3.])')
495: (8) assert_equal(repr(y), 'array([ 1., 2., -10.])')
496: (8) assert_equal(repr(np.array(y[0])), 'array(1.)')
497: (8) assert_equal(repr(np.array(y[-1])), 'array(-10.)')
498: (8) assert_equal(repr(z), 'array([100., 2., -1.])')
499: (8) assert_equal(repr(w), 'array([-100., 2., 1.])')
500: (8) assert_equal(repr(np.array([np.nan, np.inf])), 'array([nan, inf])')
501: (8) assert_equal(repr(np.array([np.nan, -np.inf])), 'array([ nan, -inf])')
502: (8) x = np.array([np.inf, 100000, 1.1234])
503: (8) y = np.array([np.inf, 100000, -1.1234])
504: (8) z = np.array([np.inf, 1.1234, -1e120])
505: (8) np.set_printoptions(precision=2)
506: (8) assert_equal(repr(x), 'array([      inf, 1.00e+05, 1.12e+00])')
507: (8) assert_equal(repr(y), 'array([      inf, 1.00e+05, -1.12e+00])')
508: (8) assert_equal(repr(z), 'array([      inf, 1.12e+000, -1.00e+120])')
509: (4) def test_bool_spacing(self):
510: (8) assert_equal(repr(np.array([True, True])),
511: (21) "array([ True, True])")
512: (8) assert_equal(repr(np.array([True, False])),
513: (21) "array([ True, False])")
514: (8) assert_equal(repr(np.array([True])),
515: (21) "array([ True])")
516: (8) assert_equal(repr(np.array(True)),
517: (21) "array(True)")
518: (8) assert_equal(repr(np.array(False)),
519: (21) "array(False)")
520: (4) def test_sign_spacing(self):
521: (8) a = np.arange(4.)
522: (8) b = np.array([1.234e9])
523: (8) c = np.array([1.0 + 1.0j, 1.123456789 + 1.123456789j], dtype='c16')
524: (8) assert_equal(repr(a), 'array([0., 1., 2., 3.])')
525: (8) assert_equal(repr(np.array(1.)), 'array(1.)')
526: (8) assert_equal(repr(b), 'array([1.234e+09])')
527: (8) assert_equal(repr(np.array([0.])), 'array([0.])')
528: (8) assert_equal(repr(c),
529: (12) "array([1.          +1.j         , 1.12345679+1.12345679j])")
530: (8) assert_equal(repr(np.array([0., -0.])), 'array([ 0., -0.])')
531: (8) np.set_printoptions(sign=' ')
532: (8) assert_equal(repr(a), 'array([ 0.,  1.,  2.,  3.])')

```

```

533: (8) assert_equal(repr(np.array(1.)), 'array( 1.)')
534: (8) assert_equal(repr(b), 'array([ 1.234e+09])')
535: (8) assert_equal(repr(c),
536: (12) "array([ 1.           +1.j           , 1.12345679+1.12345679j])")
537: (8) assert_equal(repr(np.array([0., -0.])), 'array([ 0., -0.])')
538: (8) np.set_printoptions(sign='+')
539: (8) assert_equal(repr(a), 'array([+0., +1., +2., +3.])')
540: (8) assert_equal(repr(np.array(1.)), 'array(+1.)')
541: (8) assert_equal(repr(b), 'array([+1.234e+09])')
542: (8) assert_equal(repr(c),
543: (12) "array([+1.           +1.j           , +1.12345679+1.12345679j])")
544: (8) np.set_printoptions(legacy='1.13')
545: (8) assert_equal(repr(a), 'array([ 0.,  1.,  2.,  3.])')
546: (8) assert_equal(repr(b), 'array([-1.23400000e+09])')
547: (8) assert_equal(repr(-b), 'array([-1.23400000e+09])')
548: (8) assert_equal(repr(np.array(1.)), 'array(1.0)')
549: (8) assert_equal(repr(np.array([0.])), 'array([ 0.])')
550: (8) assert_equal(repr(c),
551: (12) "array([ 1.00000000+1.j           , 1.12345679+1.12345679j])")
552: (8) assert_equal(str(np.array([-1., 10])), "[ -1. 10.]")
553: (8) assert_raises(TypeError, np.set_printoptions, wrongarg=True)
554: (4) def test_float_overflow_nowarn(self):
555: (8)     repr(np.array([1e4, 0.1], dtype='f2'))
556: (4) def test_sign_spacing_structured(self):
557: (8)     a = np.ones(2, dtype='<f,<f')
558: (8)     assert_equal(repr(a),
559: (12) "array([(1., 1.), (1., 1.)], dtype=[('f0', '<f4'), ('f1',
560: '<f4')])")
561: (8) def test_floatmode(self):
562: (8)     x = np.array([0.6104, 0.922, 0.457, 0.0906, 0.3733, 0.007244,
563: (22)             0.5933, 0.947, 0.2383, 0.4226], dtype=np.float16)
564: (8)     y = np.array([0.2918820979355541, 0.5064172631089138,
565: (22)             0.2848750619642916, 0.4342965294660567,
566: (22)             0.7326538397312751, 0.3459503329096204,
567: (22)             0.0862072768214508, 0.39112753029631175],
568: (22)             dtype=np.float64)
569: (8)     z = np.arange(6, dtype=np.float16)/10
570: (8)     c = np.array([1.0 + 1.0j, 1.123456789 + 1.123456789j], dtype='c16')
571: (8)     w = np.array(['1e{}'.format(i) for i in range(25)], dtype=np.float64)
572: (8)     wp = np.array([1.234e1, 1e2, 1e123])
573: (8)     np.set_printoptions(floatmode='unique')
574: (8)     assert_equal(repr(x),
575: (12) "array([0.6104   , 0.922   , 0.457   , 0.0906  , 0.3733  ,
576: 0.007244,\n")
577: (12) "0.2848750619642916 ,\n"
578: (12) "0.3459503329096204 ,\n"
579: (12) "1.0 + 1.0j, 1.123456789 + 1.123456789j], dtype='c16')")
580: (12) "array([0.5933   , 0.947   , 0.2383  , 0.4226  ], dtype=float16)")
581: (8) assert_equal(repr(y),
582: (12) "array([0.2918820979355541 , 0.5064172631089138 ,
583: (8)             0.4342965294660567 , 0.7326538397312751 ,
584: (12)             0.0862072768214508 , 0.39112753029631175])")
585: (12) assert_equal(repr(z),
586: (12) "array([0. , 0.1, 0.2, 0.3, 0.4, 0.5], dtype=float16)")
587: (12) assert_equal(repr(w),
588: (12) "array([1.e+00, 1.e+01, 1.e+02, 1.e+03, 1.e+04, 1.e+05, 1.e+06,
589: 1.e+07,\n")
590: (12) "1.e+08, 1.e+09, 1.e+10, 1.e+11, 1.e+12, 1.e+13, 1.e+14,
591: (12)             1.e+16, 1.e+17, 1.e+18, 1.e+19, 1.e+20, 1.e+21, 1.e+22,
592: (12)             1.e+24])")
593: (12) assert_equal(repr(wp), "array([1.234e+001, 1.000e+002, 1.000e+123])")
594: (8) assert_equal(repr(c),
595: (12) "array([1.           +1.j           , 1.123456789+1.123456789j])")
596: (8) np.set_printoptions(floatmode='maxprec', precision=8)
597: (8) assert_equal(repr(x),
598: (12) "array([0.6104   , 0.922   , 0.457   , 0.0906  , 0.3733  ,
599: 0.007244,\n")

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

594: (12)                                     "      0.5933 , 0.947 , 0.2383 , 0.4226 ], dtype=float16")
595: (8)          assert_equal(repr(y),
596: (12)            "array([0.2918821 , 0.50641726, 0.28487506, 0.43429653,
0.73265384,\n"
597: (12)              "      0.34595033, 0.08620728, 0.39112753])")
598: (8)          assert_equal(repr(z),
599: (12)            "array([0. , 0.1, 0.2, 0.3, 0.4, 0.5], dtype=float16)")
600: (8)          assert_equal(repr(w[:5]),
601: (12)            "array([1.e+00, 1.e+05, 1.e+10, 1.e+15, 1.e+20]))")
602: (8)          assert_equal(repr(wp), "array([1.234e+001, 1.000e+002, 1.000e+123]))")
603: (8)          assert_equal(repr(c),
604: (12)            "array([1.           +1.j           , 1.12345679+1.12345679j])")
605: (8)          np.set_printoptions(floatmode='fixed', precision=4)
606: (8)          assert_equal(repr(x),
607: (12)            "array([0.6104, 0.9219, 0.4570, 0.0906, 0.3733, 0.0072, 0.5933,
0.9468,\n"
608: (12)              "      0.2383, 0.4226], dtype=float16)")
609: (8)          assert_equal(repr(y),
610: (12)            "array([0.2919, 0.5064, 0.2849, 0.4343, 0.7327, 0.3460, 0.0862,
0.3911]))")
611: (8)          assert_equal(repr(z),
612: (12)            "array([0.0000, 0.1000, 0.2000, 0.3000, 0.3999, 0.5000],
dtype=float16"))
613: (8)          assert_equal(repr(w[:5]),
614: (12)            "array([1.0000e+00, 1.0000e+05, 1.0000e+10, 1.0000e+15,
1.0000e+20]))")
615: (8)          assert_equal(repr(wp), "array([1.2340e+001, 1.0000e+002,
1.0000e+123]))")
616: (8)          assert_equal(repr(np.zeros(3)), "array([0.0000, 0.0000, 0.0000])")
617: (8)          assert_equal(repr(c),
618: (12)            "array([1.0000+1.0000j, 1.1235+1.1235j])")
619: (8)          np.set_printoptions(floatmode='fixed', precision=8)
620: (8)          assert_equal(repr(z),
621: (12)            "array([0.00000000, 0.09997559, 0.19995117, 0.30004883,
0.39990234,\n"
622: (12)              "      0.50000000], dtype=float16)")
623: (8)          np.set_printoptions(floatmode='maxprec_equal', precision=8)
624: (8)          assert_equal(repr(x),
625: (12)            "array([0.610352, 0.921875, 0.457031, 0.090576, 0.373291,
0.007244,\n"
626: (12)              "      0.593262, 0.946777, 0.238281, 0.422607], dtype=float16)")
627: (8)          assert_equal(repr(y),
628: (12)            "array([0.29188210, 0.50641726, 0.28487506, 0.43429653,
0.73265384,\n"
629: (12)              "      0.34595033, 0.08620728, 0.39112753])")
630: (8)          assert_equal(repr(z),
631: (12)            "array([0.0, 0.1, 0.2, 0.3, 0.4, 0.5], dtype=float16)")
632: (8)          assert_equal(repr(w[:5]),
633: (12)            "array([1.e+00, 1.e+05, 1.e+10, 1.e+15, 1.e+20]))")
634: (8)          assert_equal(repr(wp), "array([1.234e+001, 1.000e+002, 1.000e+123]))")
635: (8)          assert_equal(repr(c),
636: (12)            "array([1.00000000+1.00000000j, 1.12345679+1.12345679j])")
637: (8)          a = np.float64.fromhex('-1p-97')
638: (8)          assert_equal(np.float64(np.array2string(a, floatmode='unique')), a)
639: (4)          def test_legacy_mode_scalars(self):
640: (8)            np.set_printoptions(legacy='1.13')
641: (8)            assert_equal(str(np.float64(1.123456789123456789)), '1.12345678912')
642: (8)            assert_equal(str(np.complex128(complex(1, np.nan))), '(1+nan*j)')
643: (8)            np.set_printoptions(legacy=False)
644: (8)            assert_equal(str(np.float64(1.123456789123456789)),
645: (21)              '1.1234567891234568')
646: (8)            assert_equal(str(np.complex128(complex(1, np.nan))), '(1+nanj)')
647: (4)          def test_legacy_stray_comma(self):
648: (8)            np.set_printoptions(legacy='1.13')
649: (8)            assert_equal(str(np.arange(10000)), '[ 0 1 2 ..., 9997 9998
9999]')
650: (8)            np.set_printoptions(legacy=False)
651: (8)            assert_equal(str(np.arange(10000)), '[ 0 1 2 ... 9997 9998
9999]')

```

```

652: (4)
653: (8)
654: (8)
655: (12)
dtype=float32"))
656: (8)
657: (12)
22.],
658: (18)
659: (8)
660: (8)
661: (12)
662: (8)
663: (12)
'1'],
664: (18)
665: (4)
666: (8)
667: (8)
668: (12)
669: (12)
670: (12)
671: (12)
672: (12)
673: (12)
674: (12)
675: (12)
676: (12)
677: (12)
678: (12)
679: (12)
680: (12)
681: (8)
682: (4)
683: (4)
def test_dtype_endianness_repr(self, native):
684: (8)
685: (8)
686: (8)
687: (8)
688: (8)
689: (8)
690: (8)
691: (8)
692: (8)
693: (8)
694: (8)
695: (8)
696: (16)
697: (17)
698: (17)
699: (8)
700: (12)
701: (12)
702: (4)
703: (8)
704: (8)
705: (8)
706: (12)
707: (12)
708: (12)
709: (19)
710: (19)
711: (8)
712: (8)
713: (8)
714: (12)
715: (12)
716: (12)
717: (19)

    def test_dtype_linewidth_wrapping(self):
        np.set_printoptions(linewidth=75)
        assert_equal(repr(np.arange(10, 20., dtype='f4')),
                     "array([10., 11., 12., 13., 14., 15., 16., 17., 18., 19.],"
                     "      dtype=float32)""")
        assert_equal(repr(np.arange(10, 23., dtype='f4')), textwrap.dedent("""\
array([10., 11., 12., 13., 14., 15., 16., 17., 18., 19., 20., 21.,
       dtype=float32)"""))
        styp = '<U4'
        assert_equal(repr(np.ones(3, dtype=styp)),
                     "array(['1', '1', '1'], dtype='{}')".format(styp))
        assert_equal(repr(np.ones(12, dtype=styp)), textwrap.dedent("""\
array(['1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1', '1",
       dtype='{}')""".format(styp)))
    @pytest.mark.parametrize(
        ['native'],
        [
            ('bool',),
            ('uint8',),
            ('uint16',),
            ('uint32',),
            ('uint64',),
            ('int8',),
            ('int16',),
            ('int32',),
            ('int64',),
            ('float16',),
            ('float32',),
            ('float64',),
            ('U1'), # 4-byte width string
        ],
    )
    def test_dtype_endianness_repr(self, native):
        """
        there was an issue where
        repr(array([0], dtype='<u2')) and repr(array([0], dtype='>u2'))
        both returned the same thing:
        array([0], dtype=uint16)
        even though their dtypes have different endianness.
        """
        native_dtype = np.dtype(native)
        non_native_dtype = native_dtype.newbyteorder()
        non_native_repr = repr(np.array([1], non_native_dtype))
        native_repr = repr(np.array([1], native_dtype))
        assert ('dtype' in native_repr) ^ (native_dtype in _typelessdata), \
               ("an array's repr should show dtype if and only if the type "
                "of the array is NOT one of the standard types "
                '(e.g., int32, bool, float64).')
        if non_native_dtype.itemsize > 1:
            assert non_native_repr != native_repr
            assert f"dtype='{non_native_dtype.byteorder}'" in non_native_repr
    def test_linewidth_repr(self):
        a = np.full(7, fill_value=2)
        np.set_printoptions(linewidth=17)
        assert_equal(
            repr(a),
            textwrap.dedent("""\
array([2, 2, 2,
       2, 2, 2,
       2])"""))
    np.set_printoptions(linewidth=17, legacy='1.13')
    assert_equal(
        repr(a),
        textwrap.dedent("""\
array([2, 2, 2,
       2, 2, 2])"""))

```

```
718: (8) )
719: (8)     a = np.full(8, fill_value=2)
720: (8)     np.set_printoptions(linewidth=18, legacy=False)
721: (8)     assert_equal(
722: (12)         repr(a),
723: (12)         textwrap.dedent("""\
724: (12)             array([2, 2, 2,
725: (19)                 2, 2, 2,
726: (19)                 2, 2])"""))
727: (8)     )
728: (8)     np.set_printoptions(linewidth=18, legacy='1.13')
729: (8)     assert_equal(
730: (12)         repr(a),
731: (12)         textwrap.dedent("""\
732: (12)             array([2, 2, 2, 2,
733: (19)                 2, 2, 2, 2])"""))
734: (8)     )
735: (4)     def test_linewidth_str(self):
736: (8)         a = np.full(18, fill_value=2)
737: (8)         np.set_printoptions(linewidth=18)
738: (8)         assert_equal(
739: (12)             str(a),
740: (12)             textwrap.dedent("""\
741: (12)                 [2 2 2 2 2 2 2 2
742: (13)                     2 2 2 2 2 2 2
743: (13)                     2 2]"""))
744: (8)     )
745: (8)     np.set_printoptions(linewidth=18, legacy='1.13')
746: (8)     assert_equal(
747: (12)         str(a),
748: (12)         textwrap.dedent("""\
749: (12)             [2 2 2 2 2 2 2 2
750: (13)                 2 2 2 2 2 2 2 2]"""))
751: (8)     )
752: (4)     def test_edgeitems(self):
753: (8)         np.set_printoptions(edgeitems=1, threshold=1)
754: (8)         a = np.arange(27).reshape((3, 3, 3))
755: (8)         assert_equal(
756: (12)             repr(a),
757: (12)             textwrap.dedent("""\
758: (12)                 array([[ [ 0, ..., 2],
759: (20)                     ...,
760: (20)                         [ 6, ..., 8]],
761: (19)                     ...,
762: (19)                         [[18, ..., 20],
763: (20)                             ...,
764: (20)                               [24, ..., 26]]]]"""))
765: (8)     )
766: (8)     b = np.zeros((3, 3, 1, 1))
767: (8)     assert_equal(
768: (12)         repr(b),
769: (12)         textwrap.dedent("""\
770: (12)             array([[[[0.]],
771: (20)                 ...,
772: (20)                     [[[0.]]],
773: (19)                     ...,
774: (19)                         [[[0.]]],
775: (20)                             ...,
776: (20)                               [[[0.]]]]]]"""))
777: (8)     )
778: (8)     np.set_printoptions(legacy='1.13')
779: (8)     assert_equal(
780: (12)         repr(a),
781: (12)         textwrap.dedent("""\
782: (12)             array([[ [ 0, ..., 2],
783: (20)                     ...,
784: (20)                         [ 6, ..., 8]],
785: (19)                     ...,
786: (19)                         [[18, ..., 20],
```

```

787: (20)
788: (20)
789: (8)
790: (8)
791: (12)
792: (12)
793: (12)
794: (20)
795: (20)
796: (19)
797: (19)
798: (20)
799: (20)
800: (8)
801: (4)
802: (8)
803: (8)
804: (8)
805: (12)
806: (12)
807: (12)
808: (8)
809: (8)
810: (4)
811: (8)
812: (8)
813: (8)
814: (8)
815: (8)
816: (0)
817: (4)
818: (4)
819: (4)
820: (0)
821: (4)
822: (8)
823: (12)
824: (8)
825: (4)
826: (8)
827: (8)
828: (29)
829: (12)
830: (8)
831: (4)
832: (8)
833: (8)
834: (12)
835: (16)
836: (8)
837: (12)
838: (8)
839: (4)
840: (8)
841: (8)
842: (12)
843: (8)

-----
```

## File 78 - test\_array\_coercion.py:

```

1: (0)
2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (0)

    """
Tests for array coercion, mainly through testing `np.array` results directly.
Note that other such tests exist, e.g., in `test_api.py` and many corner-cases
are tested (sometimes indirectly) elsewhere.
"""

from itertools import permutations, product
import pytest
```

```

8: (0)
9: (0)
10: (0)
11: (0)
12: (0)
13: (4)     assert_array_equal, assert_warnings, IS_PYPY)
14: (0)
15: (4)
16: (4)     Generator for functions converting an array into various array-likes.
17: (4)     If full is True (default) it includes array-likes not capable of handling
18: (4)     all dtypes.
19: (4)
20: (4)     def ndarray(a):
21: (8)         return a
22: (4)     yield param(ndarray, id="ndarray")
23: (4)     class MyArr(np.ndarray):
24: (8)         pass
25: (4)     def subclass(a):
26: (8)         return a.view(MyArr)
27: (4)     yield subclass
28: (4)     class _SequenceLike():
29: (8)         def __len__(self):
30: (12)             raise TypeError
31: (8)         def __getitem__(self):
32: (12)             raise TypeError
33: (4)     class ArrayDunder(_SequenceLike):
34: (8)         def __init__(self, a):
35: (12)             self.a = a
36: (8)         def __array__(self, dtype=None):
37: (12)             return self.a
38: (4)         yield param(ArrayDunder, id="__array__")
39: (4)         yield param(memoryview, id="memoryview")
40: (4)     class ArrayInterface:
41: (8)         def __init__(self, a):
42: (12)             self.a = a # need to hold on to keep interface valid
43: (12)             self.__array_interface__ = a.__array_interface__
44: (4)         yield param(ArrayInterface, id="__array_interface__")
45: (4)     class ArrayStruct:
46: (8)         def __init__(self, a):
47: (12)             self.a = a # need to hold on to keep struct valid
48: (12)             self.__array_struct__ = a.__array_struct__
49: (4)         yield param(ArrayStruct, id="__array_struct__")
50: (0)     def scalar_instances(times=True, extended_precision=True, user_dtype=True):
51: (4)         yield param(np.sqrt(np.float16(5)), id="float16")
52: (4)         yield param(np.sqrt(np.float32(5)), id="float32")
53: (4)         yield param(np.sqrt(np.float64(5)), id="float64")
54: (4)         if extended_precision:
55: (8)             yield param(np.sqrt(np.longdouble(5)), id="longdouble")
56: (4)             yield param(np.sqrt(np.complex64(2+3j)), id="complex64")
57: (4)             yield param(np.sqrt(np.complex128(2+3j)), id="complex128")
58: (4)             if extended_precision:
59: (8)                 yield param(np.sqrt(np.longcomplex(2+3j)), id="clongdouble")
60: (4)                 yield param(np.int8(2), id="int8")
61: (4)                 yield param(np.int16(2), id="int16")
62: (4)                 yield param(np.int32(2), id="int32")
63: (4)                 yield param(np.int64(2), id="int64")
64: (4)                 yield param(np.uint8(2), id="uint8")
65: (4)                 yield param(np.uint16(2), id="uint16")
66: (4)                 yield param(np.uint32(2), id="uint32")
67: (4)                 yield param(np.uint64(2), id="uint64")
68: (4)                 if user_dtype:
69: (8)                     yield param(rational(1, 2), id="rational")
70: (4)                     structured = np.array([(1, 3)], "i,i")[0]
71: (4)                     assert isinstance(structured, np.void)
72: (4)                     assert structured.dtype == np.dtype("i,i")
73: (4)                     yield param(structured, id="structured")
74: (4)
75: (8)                     yield param(np.timedelta64(2), id="timedelta64[generic]")
76: (8)                     yield param(np.timedelta64(23, "s"), id="timedelta64[s]")

```

```

77: (8)             yield param(np.timedelta64("NaT", "s"), id="timedelta64[s](NaT)")
78: (8)             yield param(np.datetime64("NaT"), id="datetime64[generic](NaT)")
79: (8)             yield param(np.datetime64("2020-06-07 12:43", "ms"),
80: (4)               id="datetime64[ms]")
81: (4)             yield param(np.bytes_(b"1234"), id="bytes")
82: (4)             yield param(np.str_("2345"), id="unicode")
83: (0)             yield param(np.void(b"4321"), id="unstructured_void")
84: (4)             def is_parametric_dtype(dtype):
85: (4)               """Returns True if the dtype is a parametric legacy dtype (itemsize
86: (4)                 is 0, or a datetime without units)
87: (4)               """
88: (8)               if dtype.itemsize == 0:
89: (8)                 return True
90: (8)               if issubclass(dtype.type, (np.datetime64, np.timedelta64)):
91: (12)                 if dtype.name.endswith("64"):
92: (4)                   return True
93: (0)               return False
94: (4)             class TestStringDiscovery:
95: (12)               @pytest.mark.parametrize("obj",
96: (12)                 [object(), 1.2, 10**43, None, "string"],
97: (4)                   ids=["object", "1.2", "10**43", "None", "string"])
98: (8)               def test_basic_stringlength(self, obj):
99: (8)                 length = len(str(obj))
100: (8)                 expected = np.dtype(f"S{length}")
101: (8)                 assert np.array(obj, dtype="S").dtype == expected
102: (8)                 assert np.array([obj], dtype="S").dtype == expected
103: (8)                 arr = np.array(obj, dtype="O")
104: (8)                 assert np.array(arr, dtype="S").dtype == expected
105: (8)                 assert np.array(arr, dtype=type(expected)).dtype == expected
106: (8)                 assert arr.astype("S").dtype == expected
107: (8)                 assert arr.astype(type(np.dtype("S"))).dtype == expected
108: (12)               @pytest.mark.parametrize("obj",
109: (12)                 [object(), 1.2, 10**43, None, "string"],
110: (4)                   ids=["object", "1.2", "10**43", "None", "string"])
111: (8)               def test_nested_arrays_stringlength(self, obj):
112: (8)                 length = len(str(obj))
113: (8)                 expected = np.dtype(f"S{length}")
114: (8)                 arr = np.array(obj, dtype="O")
115: (4)                 assert np.array([arr, arr], dtype="S").dtype == expected
116: (4)               @pytest.mark.parametrize("arraylike", arraylikes())
117: (4)               def test_unpack_first_level(self, arraylike):
118: (8)                 obj = np.array([None])
119: (8)                 obj[0] = np.array(1.2)
120: (8)                 length = len(str(obj[0]))
121: (8)                 expected = np.dtype(f"S{length}")
122: (8)                 obj = arraylike(obj)
123: (8)                 arr = np.array([obj], dtype="S")
124: (8)                 assert arr.shape == (1, 1)
125: (0)                 assert arr.dtype == expected
126: (4)             class TestScalarDiscovery:
127: (8)               def test_void_special_case(self):
128: (8)                 arr = np.array((1, 2, 3), dtype="i,i,i")
129: (8)                 assert arr.shape == ()
130: (8)                 arr = np.array([(1, 2, 3)], dtype="i,i,i")
131: (8)                 assert arr.shape == (1,)
132: (4)               def test_char_special_case(self):
133: (8)                 arr = np.array("string", dtype="c")
134: (8)                 assert arr.shape == (6,)
135: (8)                 assert arr.dtype.char == "c"
136: (8)                 arr = np.array(["string"], dtype="c")
137: (8)                 assert arr.shape == (1, 6)
138: (8)                 assert arr.dtype.char == "c"
139: (4)               def test_char_special_case_deep(self):
140: (8)                 nested = ["string"] # 2 dimensions (due to string being sequence)
141: (12)                 for i in range(np.MAXDIMS - 2):
142: (8)                   nested = [nested]
143: (8)                   arr = np.array(nested, dtype='c')
144: (8)                   assert arr.shape == (1,) * (np.MAXDIMS - 1) + (6,)
145: (8)                   with pytest.raises(ValueError):

```

```

145: (12)           np.array([nested], dtype="c")
146: (4)            def test_unknown_object(self):
147: (8)              arr = np.array(object())
148: (8)              assert arr.shape == ()
149: (8)              assert arr.dtype == np.dtype("O")
150: (4)          @pytest.mark.parametrize("scalar", scalar_instances())
151: (4)            def test_scalar(self, scalar):
152: (8)              arr = np.array(scalar)
153: (8)              assert arr.shape == ()
154: (8)              assert arr.dtype == scalar.dtype
155: (8)              arr = np.array([[scalar, scalar]])
156: (8)              assert arr.shape == (1, 2)
157: (8)              assert arr.dtype == scalar.dtype
158: (4)          @pytest.mark.filterwarnings("ignore:Promotion of numbers:FutureWarning")
159: (4)            def test_scalar_promotion(self):
160: (8)              for sc1, sc2 in product(scalar_instances(), scalar_instances()):
161: (12)                  sc1, sc2 = sc1.values[0], sc2.values[0]
162: (12)                  try:
163: (16)                      arr = np.array([sc1, sc2])
164: (12)                  except (TypeError, ValueError):
165: (16)                      continue
166: (12)                  assert arr.shape == (2,)
167: (12)                  try:
168: (16)                      dt1, dt2 = sc1.dtype, sc2.dtype
169: (16)                      expected_dtype = np.promote_types(dt1, dt2)
170: (16)                      assert arr.dtype == expected_dtype
171: (12)                  except TypeError as e:
172: (16)                      assert arr.dtype == np.dtype("O")
173: (4)          @pytest.mark.parametrize("scalar", scalar_instances())
174: (4)            def test_scalar_coercion(self, scalar):
175: (8)              if isinstance(scalar, np.inexact):
176: (12)                  scalar = type(scalar)((scalar * 2)**0.5)
177: (8)              if type(scalar) is rational:
178: (12)                  pytest.xfail("Rational to object cast is undefined currently.")
179: (8)              arr = np.array(scalar, dtype=object).astype(scalar.dtype)
180: (8)              arr1 = np.array(scalar).reshape(1)
181: (8)              arr2 = np.array([scalar])
182: (8)              arr3 = np.empty(1, dtype=scalar.dtype)
183: (8)              arr3[0] = scalar
184: (8)              arr4 = np.empty(1, dtype=scalar.dtype)
185: (8)              arr4[:] = [scalar]
186: (8)              assert_array_equal(arr, arr1)
187: (8)              assert_array_equal(arr, arr2)
188: (8)              assert_array_equal(arr, arr3)
189: (8)              assert_array_equal(arr, arr4)
190: (4)          @pytest.mark.xfail(IS_PYPY, reason="`int(np.complex128(3))` fails on
PyPy")
191: (4)          @pytest.mark.filterwarnings("ignore::numpy.ComplexWarning")
192: (4)          @pytest.mark.parametrize("cast_to", scalar_instances())
193: (4)            def test_scalar_coercion_same_as_cast_and_assignment(self, cast_to):
194: (8)                """
195: (8)                Test that in most cases:
196: (11)                    * `np.array(scalar, dtype=dtype)`
197: (11)                    * `np.empty((), dtype=dtype)[()] = scalar`
198: (11)                    * `np.array(scalar).astype(dtype)`
199: (8)                should behave the same. The only exceptions are parametric dtypes
200: (8)                (mainly datetime/timedelta without unit) and void without fields.
201: (8)                """
202: (8)                dtype = cast_to.dtype # use to parametrize only the target dtype
203: (8)                for scalar in scalar_instances(times=False):
204: (12)                    scalar = scalar.values[0]
205: (12)                    if dtype.type == np.void:
206: (15)                        if scalar.dtype.fields is not None and dtype.fields is None:
207: (20)                            with pytest.raises(TypeError):
208: (24)                                np.array(scalar).astype(dtype)
209: (20)                                np.array(scalar, dtype=dtype)
210: (20)                                np.array([scalar], dtype=dtype)
211: (20)                                continue
212: (12)                    try:

```

```

213: (16)           cast = np.array(scalar).astype(dtype)
214: (12)           except (TypeError, ValueError, RuntimeError):
215: (16)               with pytest.raises(Exception):
216: (20)                   np.array(scalar, dtype=dtype)
217: (16)               if (isinstance(scalar, rational) and
218: (24)                   np.issubdtype(dtype, np.signedinteger)):
219: (20)                   return
220: (16)               with pytest.raises(Exception):
221: (20)                   np.array([scalar], dtype=dtype)
222: (16)               res = np.zeros(), dtype=dtype)
223: (16)               with pytest.raises(Exception):
224: (20)                   res[()] = scalar
225: (16)               return
226: (12)           arr = np.array(scalar, dtype=dtype)
227: (12)           assert_array_equal(arr, cast)
228: (12)           ass = np.zeros(), dtype=dtype)
229: (12)           ass[()] = scalar
230: (12)           assert_array_equal(ass, cast)
231: (4) @pytest.mark.parametrize("pyscalar", [10, 10.32, 10.14j, 10**100])
232: (4) def test_pyscalar_subclasses(self, pyscalar):
233: (8)     """NumPy arrays are read/write which means that anything but invariant
234: (8) behaviour is on thin ice. However, we currently are happy to discover
235: (8) subclasses of Python float, int, complex the same as the base classes.
236: (8) This should potentially be deprecated.
237: (8)
238: (8)     class MyScalar(type(pyscalar)):
239: (12)         pass
240: (8)         res = np.array(MyScalar(pyscalar))
241: (8)         expected = np.array(pyscalar)
242: (8)         assert_array_equal(res, expected)
243: (4) @pytest.mark.parametrize("dtype_char", np.typecodes["All"])
244: (4) def test_default_dtype_instance(self, dtype_char):
245: (8)     if dtype_char in "SU":
246: (12)         dtype = np.dtype(dtype_char + "1")
247: (8)     elif dtype_char == "V":
248: (12)         dtype = np.dtype("V8")
249: (8)
250: (12)     else:
251: (8)         dtype = np.dtype(dtype_char)
252: (8)     discovered_dtype, _ = _discover_array_parameters([], type(dtype))
253: (8)     assert discovered_dtype == dtype
254: (8)     assert discovered_dtype.itemsize == dtype.itemsize
255: (4) @pytest.mark.parametrize("dtype", np.typecodes["Integer"])
256: (4) @pytest.mark.parametrize(["scalar", "error"],
257: (12)     [(np.float64(np.nan), ValueError),
258: (13)         (np.array(-1).astype(np.ulonglong)[()], OverflowError)])
259: (4) def test_scalar_to_int_coerce_does_not_cast(self, dtype, scalar, error):
259: (8)
260: (8)     """Signed integers are currently different in that they do not cast other
261: (8) NumPy scalar, but instead use scalar.__int__(). The hardcoded
262: (8) exception to this rule is `np.array(scalar, dtype=integer)` .
263: (8)
264: (8)     dtype = np.dtype(dtype)
265: (8)     with np.errstate(invalid="ignore"):
266: (12)         coerced = np.array(scalar, dtype=dtype)
267: (12)         cast = np.array(scalar).astype(dtype)
268: (8)         assert_array_equal(coerced, cast)
269: (8)         with pytest.raises(error):
270: (12)             np.array([scalar], dtype=dtype)
271: (8)             with pytest.raises(error):
272: (12)                 cast[()] = scalar
273: (0) class TestTimeScalars:
274: (4) @pytest.mark.parametrize("dtype", [np.int64, np.float32])
275: (4) @pytest.mark.parametrize("scalar",
276: (12)     [param(np.timedelta64("NaT", "s"), id="timedelta64[s](NaT)"),
277: (13)         param(np.timedelta64(123, "s"), id="timedelta64[s]"),
278: (13)         param(np.datetime64("Nat", "generic"), id="datetime64[generic]
279: (NaT]"),
279: (13)         param(np.datetime64(1, "D"), id="datetime64[D]")],)
280: (4) def test_coercion_basic(self, dtype, scalar):

```

```

281: (8)             arr = np.array(scalar, dtype=dtype)
282: (8)             cast = np.array(scalar).astype(dtype)
283: (8)             assert_array_equal(arr, cast)
284: (8)             ass = np.ones((), dtype=dtype)
285: (8)             if issubclass(dtype, np.integer):
286: (12)                 with pytest.raises(TypeError):
287: (16)                     ass[()] = scalar
288: (8)             else:
289: (12)                 ass[()] = scalar
290: (12)                 assert_array_equal(ass, cast)
291: (4) @pytest.mark.parametrize("dtype", [np.int64, np.float32])
292: (4) @pytest.mark.parametrize("scalar",
293: (12)     [param(np.timedelta64(123, "ns"), id="timedelta64[ns]"),
294: (13)         param(np.timedelta64(12, "generic"), id="timedelta64[generic]")])
295: (4) def test_coercion_timedelta_convert_to_number(self, dtype, scalar):
296: (8)     arr = np.array(scalar, dtype=dtype)
297: (8)     cast = np.array(scalar).astype(dtype)
298: (8)     ass = np.ones((), dtype=dtype)
299: (8)     ass[()] = scalar # raises, as would np.array([scalar], dtype=dtype)
300: (8)     assert_array_equal(arr, cast)
301: (8)     assert_array_equal(cast, cast)
302: (4) @pytest.mark.parametrize("dtype", ["S6", "U6"])
303: (4) @pytest.mark.parametrize(["val", "unit"],
304: (12)     [param(123, "s", id="[s]"), param(123, "D", id="[D]")])
305: (4) def test_coercion_assignment_datetime(self, val, unit, dtype):
306: (8)     scalar = np.datetime64(val, unit)
307: (8)     dtype = np.dtype(dtype)
308: (8)     cut_string = dtype.type(str(scalar)[:6])
309: (8)     arr = np.array(scalar, dtype=dtype)
310: (8)     assert arr[()] == cut_string
311: (8)     ass = np.ones((), dtype=dtype)
312: (8)     ass[()] = scalar
313: (8)     assert ass[()] == cut_string
314: (8)     with pytest.raises(RuntimeError):
315: (12)         np.array(scalar).astype(dtype)
316: (4) @pytest.mark.parametrize(["val", "unit"],
317: (12)     [param(123, "s", id="[s]"), param(123, "D", id="[D]")])
318: (4) def test_coercion_assignment_timedelta(self, val, unit):
319: (8)     scalar = np.timedelta64(val, unit)
320: (8)     np.array(scalar, dtype="S6")
321: (8)     cast = np.array(scalar).astype("S6")
322: (8)     ass = np.ones((), dtype="S6")
323: (8)     ass[()] = scalar
324: (8)     expected = scalar.astype("S")[:6]
325: (8)     assert cast[()] == expected
326: (8)     assert ass[()] == expected
327: (0) class TestNested:
328: (4)     def test_nested_simple(self):
329: (8)         initial = [1.2]
330: (8)         nested = initial
331: (8)         for i in range(np.MAXDIMS - 1):
332: (12)             nested = [nested]
333: (8)             arr = np.array(nested, dtype="float64")
334: (8)             assert arr.shape == (1,) * np.MAXDIMS
335: (8)             with pytest.raises(ValueError):
336: (12)                 np.array([nested], dtype="float64")
337: (8)             with pytest.raises(ValueError, match=".*would exceed the maximum"):
338: (12)                 np.array([nested]) # user must ask for `object` explicitly
339: (8)                 arr = np.array([nested], dtype=object)
340: (8)                 assert arr.dtype == np.dtype("O")
341: (8)                 assert arr.shape == (1,) * np.MAXDIMS
342: (8)                 assert arr.item() is initial
343: (4)     def test_pathological_self_containing(self):
344: (8)         l = []
345: (8)         l.append(l)
346: (8)         arr = np.array([l, l, l], dtype=object)
347: (8)         assert arr.shape == (3,) + (1,) * (np.MAXDIMS - 1)
348: (8)         arr = np.array([l, [None], l], dtype=object)
349: (8)         assert arr.shape == (3, 1)

```

```

350: (4) @pytest.mark.parametrize("arraylike", arraylikes())
351: (4) def test_nested_arraylikes(self, arraylike):
352: (8)     initial = arraylike(np.ones((1, 1)))
353: (8)     nested = initial
354: (8)     for i in range(np.MAXDIMS - 1):
355: (12)         nested = [nested]
356: (8)     with pytest.raises(ValueError, match=".*would exceed the maximum"):
357: (12)         np.array(nested, dtype="float64")
358: (8)     arr = np.array(nested, dtype=object)
359: (8)     assert arr.shape == (1,) * np.MAXDIMS
360: (8)     assert arr.item() == np.array(initial).item()
361: (4) @pytest.mark.parametrize("arraylike", arraylikes())
362: (4) def test_uneven_depth_ragged(self, arraylike):
363: (8)     arr = np.arange(4).reshape((2, 2))
364: (8)     arr = arraylike(arr)
365: (8)     out = np.array([arr, [arr]], dtype=object)
366: (8)     assert out.shape == (2,)
367: (8)     assert out[0] is arr
368: (8)     assert type(out[1]) is list
369: (8)     with pytest.raises(ValueError):
370: (12)         np.array([arr, [arr, arr]], dtype=object)
371: (4) def test_empty_sequence(self):
372: (8)     arr = np.array([[], [1], [[1]]], dtype=object)
373: (8)     assert arr.shape == (3,)
374: (8)     with pytest.raises(ValueError):
375: (12)         np.array([], np.empty((0, 1)), dtype=object)
376: (4) def test_array_of_different_depths(self):
377: (8)     arr = np.zeros((3, 2))
378: (8)     mismatch_first_dim = np.zeros((1, 2))
379: (8)     mismatch_second_dim = np.zeros((3, 3))
380: (8)     dtype, shape = _discover_array_parameters(
381: (12)         [arr, mismatch_second_dim], dtype=np.dtype("O"))
382: (8)     assert shape == (2, 3)
383: (8)     dtype, shape = _discover_array_parameters(
384: (12)         [arr, mismatch_first_dim], dtype=np.dtype("O"))
385: (8)     assert shape == (2,)
386: (8)     res = np.asarray([arr, mismatch_first_dim], dtype=np.dtype("O"))
387: (8)     assert res[0] is arr
388: (8)     assert res[1] is mismatch_first_dim
389: (0) class TestBadSequences:
390: (4)     def test_growing_list(self):
391: (8)         obj = []
392: (8)         class mylist(list):
393: (12)             def __len__(self):
394: (16)                 obj.append([1, 2])
395: (16)                 return super().__len__()
396: (8)             obj.append(mylist([1, 2]))
397: (8)             with pytest.raises(RuntimeError):
398: (12)                 np.array(obj)
399: (4)     def test_mutated_list(self):
400: (8)         obj = []
401: (8)         class mylist(list):
402: (12)             def __len__(self):
403: (16)                 obj[0] = [2, 3] # replace with a different list.
404: (16)                 return super().__len__()
405: (8)             obj.append([2, 3])
406: (8)             obj.append(mylist([1, 2]))
407: (8)             np.array(obj)
408: (4)     def test_replace_0d_array(self):
409: (8)         obj = []
410: (8)         class baditem:
411: (12)             def __len__(self):
412: (16)                 obj[0][0] = 2 # replace with a different list.
413: (16)                 raise ValueError("not actually a sequence!")
414: (12)             def __getitem__(self):
415: (16)                 pass
416: (8)             obj.append([np.array(2), baditem()])
417: (8)             with pytest.raises(RuntimeError):
418: (12)                 np.array(obj)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

419: (0)
420: (4)
421: (4)
422: (8)
423: (8)
424: (8)
425: (8)
426: (8)
427: (8)
428: (4)
429: (4)
430: (4)
431: (8)
432: (8)
433: (8)
434: (8)
435: (4)
436: (8)
437: (12)
438: (16)
439: (8)
440: (8)
441: (8)
442: (8)
443: (8)
444: (12)
445: (8)
446: (8)
447: (8)
448: (8)
449: (12)
450: (4)
451: (8)
452: (8)
453: (8)
454: (8)
455: (8)
456: (12)
457: (12)
458: (16)
459: (12)
460: (12)
461: (16)
462: (12)
463: (16)
464: (8)
465: (8)
466: (8)
467: (8)
468: (4)
469: (12)
470: (4)
471: (8)
472: (8)
473: (8)
474: (8)
475: (12)
476: (16)
477: (12)
478: (16)
479: (4)
480: (8)
481: (4)
482: (4)
483: (8)
484: (12)
485: (16)
486: (20)
487: (16)

class TestArrayLikes:
    @pytest.mark.parametrize("arraylike", arraylikes())
    def test_0d_object_special_case(self, arraylike):
        arr = np.array(0.)
        obj = arraylike(arr)
        res = np.array(obj, dtype=object)
        assert_array_equal(arr, res)
        res = np.array([obj], dtype=object)
        assert res[0] is obj
    @pytest.mark.parametrize("arraylike", arraylikes())
    @pytest.mark.parametrize("arr", [np.array(0.), np.arange(4)])
    def test_object_assignment_special_case(self, arraylike, arr):
        obj = arraylike(arr)
        empty = np.arange(1, dtype=object)
        empty[:] = [obj]
        assert empty[0] is obj
    def test_0d_generic_special_case(self):
        class ArraySubclass(np.ndarray):
            def __float__(self):
                raise TypeError("e.g. quantities raise on this")
        arr = np.array(0.)
        obj = arr.view(ArraySubclass)
        res = np.array(obj)
        assert_array_equal(arr, res)
        with pytest.raises(TypeError):
            np.array([obj])
        obj = memoryview(arr)
        res = np.array(obj)
        assert_array_equal(arr, res)
        with pytest.raises(ValueError):
            np.array([obj])
    def test_arraylike_classes(self):
        arr = np.array(np.int64)
        assert arr[()] is np.int64
        arr = np.array([np.int64])
        assert arr[0] is np.int64
        class ArrayLike:
            @property
            def __array_interface__(self):
                pass
            @property
            def __array_struct__(self):
                pass
            def __array__(self):
                pass
            arr = np.array(ArrayLike)
            assert arr[()] is ArrayLike
            arr = np.array([ArrayLike])
            assert arr[0] is ArrayLike
    @pytest.mark.skipif(
        np.dtype(np.intp).itemsize < 8, reason="Needs 64bit platform")
    def test_too_large_array_error_paths(self):
        """Test the error paths, including for memory leaks"""
        arr = np.array(0, dtype="uint8")
        arr = np.broadcast_to(arr, 2**62)
        for i in range(5):
            with pytest.raises(MemoryError):
                np.array(arr)
            with pytest.raises(MemoryError):
                np.array([arr])
    @pytest.mark.parametrize("attribute",
                           ["__array_interface__", "__array__", "__array_struct__"])
    @pytest.mark.parametrize("error", [RecursionError, MemoryError])
    def test_bad_array_like_attributes(self, attribute, error):
        class BadInterface:
            def __getattr__(self, attr):
                if attr == attribute:
                    raise error
                super().__getattr__(attr)

```

```

488: (8)
489: (12)
490: (4)
491: (4)
492: (8)
493: (12)
494: (16)
495: (12)
496: (16)
497: (8)
498: (12)
499: (0)
500: (4)
501: (4)
502: (8)
503: (8)
504: (8)
505: (8)
506: (8)
507: (8)
508: (8)
509: (8)
510: (8)
511: (8)
512: (8)
513: (8)
514: (8)
515: (8)
516: (8)
517: (8)
518: (8)
519: (45)
520: (8)
521: (8)
522: (8)
523: (8)
524: (8)
525: (8)
526: (8)
527: (8)
528: (8)
529: (12)
530: (16)
531: (16)
532: (8)
533: (8)
534: (8)
535: (8)
536: (8)
537: (8)
538: (8)
539: (8)
540: (8)
541: (8)
542: (8)
543: (8)
544: (8)
545: (0)
546: (4)
547: (8)
548: (8)
549: (12)
550: (4)
551: (8)
552: (8)
553: (12)
554: (4)
555: (8)
556: (12)

        with pytest.raises(error):
            np.array(BadInterface())
    @pytest.mark.parametrize("error", [RecursionError, MemoryError])
    def test_bad_array_like_bad_length(self, error):
        class BadSequence:
            def __len__(self):
                raise error
            def __getitem__(self):
                return 1
        with pytest.raises(error):
            np.array(BadSequence())
class TestAsArray:
    """Test expected behaviors of ``asarray``."""
    def test_dtype_identity(self):
        """Confirm the intended behavior for *dtype* kwarg.
        The result of ``asarray()`` should have the dtype provided through the keyword argument, when used. This forces unique array handles to be produced for unique np.dtype objects, but (for equivalent dtypes), the underlying data (the base object) is shared with the original array object.
        Ref https://github.com/numpy/numpy/issues/1468
        """
        int_array = np.array([1, 2, 3], dtype='i')
        assert np.asarray(int_array) is int_array
        assert np.asarray(int_array, dtype='i') is int_array
        unequal_type = np.dtype('i', metadata={'spam': True})
        annotated_int_array = np.asarray(int_array, dtype=unequal_type)
        assert annotated_int_array is not int_array
        assert annotated_int_array.base is int_array
        equivalent_requirement = np.dtype('i', metadata={'spam': True})
        annotated_int_array_alt = np.asarray(annotated_int_array,
   dtype=equivalent_requirement)
        assert unequal_type == equivalent_requirement
        assert unequal_type is not equivalent_requirement
        assert annotated_int_array_alt is not annotated_int_array
        assert annotated_int_array_alt.dtype is equivalent_requirement
        integer_type_codes = ('i', 'l', 'q')
        integer_dtypes = [np.dtype(code) for code in integer_type_codes]
        typeA = None
        typeB = None
        for typeA, typeB in permutations(integer_dtypes, r=2):
            if typeA == typeB:
                assert typeA is not typeB
                break
        assert isinstance(typeA, np.dtype) and isinstance(typeB, np.dtype)
        long_int_array = np.asarray(int_array, dtype='l')
        long_long_int_array = np.asarray(int_array, dtype='q')
        assert long_int_array is not int_array
        assert long_long_int_array is not int_array
        assert np.asarray(long_int_array, dtype='q') is not long_int_array
        array_a = np.asarray(int_array, dtype=typeA)
        assert typeA == typeB
        assert typeA is not typeB
        assert array_a.dtype is typeA
        assert array_a is not np.asarray(array_a, dtype=typeB)
        assert np.asarray(array_a, dtype=typeB).dtype is typeB
        assert array_a is np.asarray(array_a, dtype=typeB).base
class TestSpecialAttributeLookupFailure:
    class WeirdArrayLike:
        @property
        def __array__(self):
            raise RuntimeError("oops!")
    class WeirdArrayInterface:
        @property
        def __array_interface__(self):
            raise RuntimeError("oops!")
    def test_DEPRECATED(self):
        with pytest.raises(RuntimeError):
            np.array(self.WeirdArrayLike())

```

```

557: (8)           with pytest.raises(RuntimeError):
558: (12)             np.array(self.WeirdArrayInterface())
559: (0)           def test_subarray_from_array_construction():
560: (4)             arr = np.array([1, 2])
561: (4)             res = arr.astype("(2)i,")
562: (4)             assert_array_equal(res, [[1, 1], [2, 2]])
563: (4)             res = np.array(arr, dtype="(2)i,")
564: (4)             assert_array_equal(res, [[1, 1], [2, 2]])
565: (4)             res = np.array([(1,), (2,)], arr, dtype="(2)i,")
566: (4)             assert_array_equal(res, [[[1, 1], [2, 2]], [[1, 1], [2, 2]]])
567: (4)             arr = np.arange(5 * 2).reshape(5, 2)
568: (4)             expected = np.broadcast_to(arr[:, :, np.newaxis, np.newaxis], (5, 2, 2,
2))
569: (4)             res = arr.astype("(2,2)f")
570: (4)             assert_array_equal(res, expected)
571: (4)             res = np.array(arr, dtype="(2,2)f")
572: (4)             assert_array_equal(res, expected)
573: (0)           def test_empty_string():
574: (4)             res = np.array([""] * 10, dtype="S")
575: (4)             assert_array_equal(res, np.array("\0", "S1"))
576: (4)             assert res.dtype == "S1"
577: (4)             arr = np.array([""] * 10, dtype=object)
578: (4)             res = arr.astype("S")
579: (4)             assert_array_equal(res, b(""))
580: (4)             assert res.dtype == "S1"
581: (4)             res = np.array(arr, dtype="S")
582: (4)             assert_array_equal(res, b(""))
583: (4)             assert res.dtype == f"S{np.dtype('O').itemsize}"
584: (4)             res = np.array([[""] * 10, arr], dtype="S")
585: (4)             assert_array_equal(res, b(""))
586: (4)             assert res.shape == (2, 10)
587: (4)             assert res.dtype == "S1"

```

---

## File 79 - test\_array\_interface.py:

```

1: (0)           import sys
2: (0)           import pytest
3: (0)           import numpy as np
4: (0)           from numpy.testing import extbuild
5: (0)           @pytest.fixture
6: (0)           def get_module(tmp_path):
7: (4)             """ Some codes to generate data and manage temporary buffers use when
8: (4)             sharing with numpy via the array interface protocol.
9: (4)             """
10: (4)            if not sys.platform.startswith('linux'):
11: (8)              pytest.skip('link fails on cygwin')
12: (4)            prologue = '''
13: (8)              NPY_NO_EXPORT
14: (8)              void delete_array_struct(PyObject *cap) {
15: (12)                /* get the array interface structure */
16: (12)                PyArrayInterface *inter = (PyArrayInterface*)
17: (16)                  PyCapsule_GetPointer(cap, NULL);
18: (12)                /* get the buffer by which data was shared */
19: (12)                double *ptr = (double*)PyCapsule_GetContext(cap);
20: (12)                /* for the purposes of the regression test set the elements
21: (15)                  to nan */
22: (12)                for (npy_intp i = 0; i < inter->shape[0]; ++i)
23: (16)                  ptr[i] = nan("");
24: (12)                /* free the shared buffer */
25: (12)                free(ptr);
26: (12)                /* free the array interface structure */
27: (12)                free(inter->shape);
28: (12)                free(inter);
29: (12)                fprintf(stderr, "delete_array_struct\\ncap = %ld inter = %ld"
30: (16)                  " ptr = %ld\\n", (long)cap, (long)inter, (long)ptr);
31: (8)              }
32: (8)              ...

```

```

33: (4)     functions = [
34: (8)         ("new_array_struct", "METH_VARARGS", """
35: (12)             long long n_elem = 0;
36: (12)             double value = 0.0;
37: (12)             if (!PyArg_ParseTuple(args, "Ld", &n_elem, &value)) {
38: (16)                 Py_RETURN_NONE;
39: (12)             }
40: (12)             /* allocate and initialize the data to share with numpy */
41: (12)             long long n_bytes = n_elem*sizeof(double);
42: (12)             double *data = (double*)malloc(n_bytes);
43: (12)             if (!data) {
44: (16)                 PyErr_Format(PyExc_MemoryError,
45: (20)                     "Failed to malloc %lld bytes", n_bytes);
46: (16)                 Py_RETURN_NONE;
47: (12)             }
48: (12)             for (long long i = 0; i < n_elem; ++i) {
49: (16)                 data[i] = value;
50: (12)             }
51: (12)             /* calculate the shape and stride */
52: (12)             int nd = 1;
53: (12)             npy_intp *ss = (npy_intp*)malloc(2*nd*sizeof(npy_intp));
54: (12)             npy_intp *shape = ss;
55: (12)             npy_intp *stride = ss + nd;
56: (12)             shape[0] = n_elem;
57: (12)             stride[0] = sizeof(double);
58: (12)             /* construct the array interface */
59: (12)             PyArrayInterface *inter = (PyArrayInterface*)
60: (16)                 malloc(sizeof(PyArrayInterface));
61: (12)             memset(inter, 0, sizeof(PyArrayInterface));
62: (12)             inter->two = 2;
63: (12)             inter->nd = nd;
64: (12)             inter->typekind = 'f';
65: (12)             inter->itemsize = sizeof(double);
66: (12)             inter->shape = shape;
67: (12)             inter->strides = stride;
68: (12)             inter->data = data;
69: (12)             inter->flags = NPY_ARRAY_WRITEABLE | NPY_ARRAY_NOTSWAPPED |
70: (27)                 NPY_ARRAY_ALIGNED | NPY_ARRAY_C_CONTIGUOUS;
71: (12)             /* package into a capsule */
72: (12)             PyObject *cap = PyCapsule_New(inter, NULL, delete_array_struct);
73: (12)             /* save the pointer to the data */
74: (12)             PyCapsule_SetContext(cap, data);
75: (12)             fprintf(stderr, "new_array_struct\\ncap = %ld inter = %ld"
76: (16)                 " ptr = %ld\\n", (long)cap, (long)inter, (long)data);
77: (12)             return cap;
78: (8)         """
79: (8)     ]
80: (4)     more_init = "import_array();"
81: (4)     try:
82: (8)         import array_interface_testing
83: (8)         return array_interface_testing
84: (4)     except ImportError:
85: (8)         pass
86: (4)     return extbuild.build_and_import_extension('array_interface_testing',
87: (47)             functions,
88: (47)             prologue=prologue,
89: (47)             include_dirs=
[np.get_include()],
90: (47)             build_dir=tmp_path,
91: (47)             more_init=more_init)
92: (0)     @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
93: (0)     @pytest.mark.slow
94: (0)     def test_cstruct(get_module):
95: (4)         class data_source:
96: (8)             """
97: (8)             This class is for testing the timing of the PyCapsule destructor
98: (8)             invoked when numpy release its reference to the shared data as part of
99: (8)             the numpy array interface protocol. If the PyCapsule destructor is
100: (8)             called early the shared data is freed and invalid memory accesses will

```

```

101: (8)          occur.
102: (8)
103: (8)
104: (12)
105: (12)
106: (8)
107: (8)
108: (12)
109: (4)
110: (4)
111: (4)
112: (4)
113: (4)
114: (4)
115: (4)
116: (4)
117: (4)
118: (17)
119: (4)
120: (4)
121: (4)
122: (4)
123: (4)
124: (4)
125: (4)
126: (4)
127: (4)
128: (4)
129: (4)
130: (4)
131: (4)
132: (17)
133: (4)
134: (4)
135: (4)
136: (4)
137: (4)
138: (4)
139: (4)
140: (4)
141: (4)

-----

```

## File 80 - test\_casting\_unittests.py:

```

1: (0)          """
2: (0)          The tests exercise the casting machinery in a more low-level manner.
3: (0)          The reason is mostly to test a new implementation of the casting machinery.
4: (0)          Unlike most tests in NumPy, these are closer to unit-tests rather
5: (0)          than integration tests.
6: (0)
7: (0)          import pytest
8: (0)          import textwrap
9: (0)          import enum
10: (0)         import random
11: (0)         import ctypes
12: (0)         import numpy as np
13: (0)         from numpy.lib.stride_tricks import as_strided
14: (0)         from numpy.testing import assert_array_equal
15: (0)         from numpy.core._multiarray_umath import _get_castingimpl as get_castingimpl
16: (0)         simple_dtypes = "?bhilqBHILQefdFD"
17: (0)         if np.dtype("l").itemsize != np.dtype("q").itemsize:
18: (4)             simple_dtypes = simple_dtypes.replace("l", "").replace("L", "")
19: (0)         simple_dtypes = [type(np.dtype(c)) for c in simple_dtypes]
20: (0)
21: (4)         def simple_dtype_instances():
22: (8)             for dtype_class in simple_dtypes:
23: (8)                 dt = dtype_class()

```

```

24: (8)             if dt.byteorder != "|":
25: (12)            dt = dt.newbyteorder()
26: (12)            yield pytest.param(dt, id=str(dt))
27: (0)             def get_expected_stringlength(dtype):
28: (4)               """Returns the string length when casting the basic dtypes to strings.
29: (4)               """
30: (4)               if dtype == np.bool_:
31: (8)                 return 5
32: (4)               if dtype.kind in "iu":
33: (8)                 if dtype.itemsize == 1:
34: (12)                   length = 3
35: (8)                 elif dtype.itemsize == 2:
36: (12)                     length = 5
37: (8)                 elif dtype.itemsize == 4:
38: (12)                     length = 10
39: (8)                 elif dtype.itemsize == 8:
40: (12)                     length = 20
41: (8)                 else:
42: (12)                     raise AssertionError(f"did not find expected length for {dtype}")
43: (8)               if dtype.kind == "i":
44: (12)                 length += 1 # adds one character for the sign
45: (8)                 return length
46: (4)               if dtype.char == "g":
47: (8)                 return 48
48: (4)               elif dtype.char == "G":
49: (8)                 return 48 * 2
50: (4)               elif dtype.kind == "f":
51: (8)                 return 32 # also for half apparently.
52: (4)               elif dtype.kind == "c":
53: (8)                 return 32 * 2
54: (4)               raise AssertionError(f"did not find expected length for {dtype}")
55: (0)             class Casting(enum.IntEnum):
56: (4)               no = 0
57: (4)               equiv = 1
58: (4)               safe = 2
59: (4)               same_kind = 3
60: (4)               unsafe = 4
61: (0)             def _get_cancast_table():
62: (4)               table = textwrap.dedent("""
63: (8)                 X ? b h i l q B H I L Q e f d g F D G S U V O M m
64: (8)                 ? # = . =
65: (8)                 b . # = = = = . . . . = = = = = = = = = = = = . =
66: (8)                 h . ~ # = = . . . . ~ = = = = = = = = = = = . =
67: (8)                 i . ~ ~ # = = . . . . ~ ~ = = ~ = = = = = = . =
68: (8)                 l . ~ ~ ~ # # . . . . ~ ~ = = ~ = = = = = = . =
69: (8)                 q . ~ ~ ~ # # . . . . ~ ~ = = ~ = = = = = = . =
70: (8)                 B . ~ = = = = # = = = = = = = = = = = = . =
71: (8)                 H . ~ ~ = = = ~ # = = ~ = = = = = = = = = . =
72: (8)                 I . ~ ~ ~ = = ~ ~ # = = ~ ~ = = ~ = = = = = . =
73: (8)                 L . ~ ~ ~ ~ ~ ~ ~ # # ~ ~ = = ~ = = = = = . ~
74: (8)                 Q . ~ ~ ~ ~ ~ ~ ~ # # ~ ~ = = ~ = = = = = . ~
75: (8)                 e . . . . . . . . . . # = = = = = = = = = . .
76: (8)                 f . . . . . . . . . ~ # = = = = = = = = = . .
77: (8)                 d . . . . . . . . . ~ ~ # = ~ = = = = = = . .
78: (8)                 g . . . . . . . . . ~ ~ ~ # ~ ~ = = = = = . .
79: (8)                 F . . . . . . . . . . # = = = = = = = = . .
80: (8)                 D . . . . . . . . . . ~ # = = = = = = = = . .
81: (8)                 G . . . . . . . . . . ~ ~ # = = = = = = . .
82: (8)                 S . . . . . . . . . . . . . # = = = = = = . .
83: (8)                 U . . . . . . . . . . . . . . # = = = = = = . .
84: (8)                 V . . . . . . . . . . . . . . . # = = = = = = . .
85: (8)                 O . . . . . . . . . . . . . . . = # . .
86: (8)                 M . . . . . . . . . . . . . . . = = # . .
87: (8)                 m . . . . . . . . . . . . . . . = = = . #
88: (8)                 """).strip().split("\n")
89: (4)               dtypes = [type(np.dtype(c)) for c in table[0][2::2]]
90: (4)               convert_cast = {".".": Casting.unsafe, "~": Casting.same_kind,
91: (20)                  "=": Casting.safe, "#": Casting.equiv,
92: (20)                  " ": -1}

```

```

93: (4)          cancast = {}
94: (4)          for from_dt, row in zip(dtypes, table[1:]):
95: (8)            cancast[from_dt] = {}
96: (8)            for to_dt, c in zip(dtypes, row[2::2]):
97: (12)              cancast[from_dt][to_dt] = convert_cast[c]
98: (4)          return cancast
99: (0)          CAST_TABLE = _get_cancast_table()
100: (0)         class TestChanges:
101: (4)           """
102: (4)             These test cases exercise some behaviour changes
103: (4)           """
104: (4)           @pytest.mark.parametrize("string", ["S", "U"])
105: (4)           @pytest.mark.parametrize("floating", ["e", "f", "d", "g"])
106: (4)           def test_float_to_string(self, floating, string):
107: (8)             assert np.can_cast(floating, string)
108: (8)             assert np.can_cast(floating, f"{string}100")
109: (4)           def test_to_void(self):
110: (8)             assert np.can_cast("d", "V")
111: (8)             assert np.can_cast("S20", "V")
112: (8)             assert not np.can_cast("d", "V1")
113: (8)             assert not np.can_cast("S20", "V1")
114: (8)             assert not np.can_cast("U1", "V1")
115: (8)             assert np.can_cast("d,i", "V", casting="same_kind")
116: (8)             assert np.can_cast("V3", "V", casting="no")
117: (8)             assert np.can_cast("V0", "V", casting="no")
118: (0)         class TestCasting:
119: (4)           size = 1500 # Best larger than NPY_LOWLEVEL_BUFFER_BLOCKSIZE * itemsize
120: (4)           def get_data(self, dtype1, dtype2):
121: (8)             if dtype2 is None or dtype1.itemsize >= dtype2.itemsize:
122: (12)               length = self.size // dtype1.itemsize
123: (8)
124: (12)               length = self.size // dtype2.itemsize
125: (8)             arr1 = np.empty(length, dtype=dtype1)
126: (8)             assert arr1.flags.c_contiguous
127: (8)             assert arr1.flags.aligned
128: (8)             values = [random.randrange(-128, 128) for _ in range(length)]
129: (8)             for i, value in enumerate(values):
130: (12)               if value < 0 and dtype1.kind == "u":
131: (16)                 value = value + np.iinfo(dtype1).max + 1
132: (12)                 arr1[i] = value
133: (8)             if dtype2 is None:
134: (12)               if dtype1.char == "?":
135: (16)                 values = [bool(v) for v in values]
136: (12)                 return arr1, values
137: (8)             if dtype2.char == "?":
138: (12)               values = [bool(v) for v in values]
139: (8)             arr2 = np.empty(length, dtype=dtype2)
140: (8)             assert arr2.flags.c_contiguous
141: (8)             assert arr2.flags.aligned
142: (8)             for i, value in enumerate(values):
143: (12)               if value < 0 and dtype2.kind == "u":
144: (16)                 value = value + np.iinfo(dtype2).max + 1
145: (12)                 arr2[i] = value
146: (8)             return arr1, arr2, values
147: (4)           def get_data_variation(self, arr1, arr2, aligned=True, contig=True):
148: (8)
149: (8)             """
150: (8)               Returns a copy of arr1 that may be non-contiguous or unaligned, and a
151: (8)               matching array for arr2 (although not a copy).
152: (8)
153: (12)             if contig:
154: (12)               stride1 = arr1.dtype.itemsize
155: (8)               stride2 = arr2.dtype.itemsize
156: (12)             elif aligned:
157: (12)               stride1 = 2 * arr1.dtype.itemsize
158: (8)               stride2 = 2 * arr2.dtype.itemsize
159: (12)             else:
160: (12)               stride1 = arr1.dtype.itemsize + 1
161: (8)               stride2 = arr2.dtype.itemsize + 1
161: (8)               max_size1 = len(arr1) * 3 * arr1.dtype.itemsize + 1

```

```

162: (8)           max_size2 = len(arr2) * 3 * arr2.dtype.itemsize + 1
163: (8)           from_bytes = np.zeros(max_size1, dtype=np.uint8)
164: (8)           to_bytes = np.zeros(max_size2, dtype=np.uint8)
165: (8)           assert stride1 * len(arr1) <= from_bytes.nbytes
166: (8)           assert stride2 * len(arr2) <= to_bytes.nbytes
167: (8)           if aligned:
168: (12)             new1 = as_strided(from_bytes[:-1].view(arr1.dtype),
169: (30)                           arr1.shape, (stride1,))
170: (12)             new2 = as_strided(to_bytes[:-1].view(arr2.dtype),
171: (30)                           arr2.shape, (stride2,))
172: (8)           else:
173: (12)             new1 = as_strided(from_bytes[1:].view(arr1.dtype),
174: (30)                           arr1.shape, (stride1,))
175: (12)             new2 = as_strided(to_bytes[1:].view(arr2.dtype),
176: (30)                           arr2.shape, (stride2,)))
177: (8)           new1[...] = arr1
178: (8)           if not contig:
179: (12)             offset = arr1.dtype.itemsize if aligned else 0
180: (12)             buf = from_bytes[offset::stride1].tobytes()
181: (12)             assert buf.count(b"\0") == len(buf)
182: (8)           if contig:
183: (12)             assert new1.flags.c_contiguous
184: (12)             assert new2.flags.c_contiguous
185: (8)           else:
186: (12)             assert not new1.flags.c_contiguous
187: (12)             assert not new2.flags.c_contiguous
188: (8)           if aligned:
189: (12)             assert new1.flags.aligned
190: (12)             assert new2.flags.aligned
191: (8)           else:
192: (12)             assert not new1.flags.aligned or new1.dtype.alignment == 1
193: (12)             assert not new2.flags.aligned or new2.dtype.alignment == 1
194: (8)           return new1, new2
195: (4)           @pytest.mark.parametrize("from_Dt", simple_dtypes)
196: (4)           def test_simple_cancast(self, from_Dt):
197: (8)             for to_Dt in simple_dtypes:
198: (12)               cast = get_castingimpl(from_Dt, to_Dt)
199: (12)               for from_dt in [from_Dt(), from_Dt().newbyteorder()]:
200: (16)                 default = cast._resolve_descriptors((from_dt, None))[1][1]
201: (16)                 assert default == to_Dt()
202: (16)                 del default
203: (16)                 for to_dt in [to_Dt(), to_Dt().newbyteorder()]:
204: (20)                   casting, (from_res, to_res), view_off =
205: (28)                     cast._resolve_descriptors((from_dt, to_dt))
206: (20)                   assert(type(from_res) == from_Dt)
207: (20)                   assert(type(to_res) == to_Dt)
208: (20)                   if view_off is not None:
209: (24)                     assert casting == Casting.no
210: (24)                     assert Casting.equiv == CAST_TABLE[from_Dt][to_Dt]
211: (24)                     assert from_res.isnative == to_res.isnative
212: (20)                   else:
213: (24)                     if from_Dt == to_Dt:
214: (28)                       assert from_res.isnative != to_res.isnative
215: (24)                       assert casting == CAST_TABLE[from_Dt][to_Dt]
216: (20)                     if from_Dt is to_Dt:
217: (24)                       assert(from_dt is from_res)
218: (24)                       assert(to_dt is to_res)
219: (4)           @pytest.mark.filterwarnings("ignore::numpy.ComplexWarning")
220: (4)           @pytest.mark.parametrize("from_dt", simple_dtype_instances())
221: (4)           def test_simple_direct_casts(self, from_dt):
222: (8)             """
223: (8)             This test checks numeric direct casts for dtypes supported also by the
224: (8)             struct module (plus complex). It tries to be test a wide range of
225: (8)             inputs, but skips over possibly undefined behaviour (e.g. int
rollover).
226: (8)             Longdouble and CLongdouble are tested, but only using double
precision.
227: (8)             If this test creates issues, it should possibly just be simplified
228: (8)             or even removed (checking whether unaligned/non-contiguous casts give

```

```

229: (8)           the same results is useful, though).
230: (8)
231: (8)
232: (12)
233: (12)
234: (12)
235: (16)
236: (12)
237: (16)
238: (12)
239: (12)
240: (12)
241: (12)
242: (12)
243: (12)
244: (12)
245: (12)
246: (12)
247: (12)
248: (20)
249: (16)
250: (12)
251: (12)
252: (12)
253: (12)
254: (12)
255: (12)
256: (12)
257: (12)
258: (12)
259: (4)          @pytest.mark.parametrize("from_Dt", simple_dtypes)
260: (4)          def test_numeric_to_times(self, from_Dt):
261: (8)            from_dt = from_Dt()
262: (8)            time_dtypes = [np.dtype("M8"), np.dtype("M8[ms]"), np.dtype("M8[4D]"),
263: (23)                  np.dtype("m8"), np.dtype("m8[ms]"), np.dtype("m8[4D]")]
264: (8)
265: (12)
266: (12)
267: (16)
268: (12)
269: (12)
270: (12)
271: (12)
272: (12)
273: (12)
274: (12)
275: (12)
276: (12)
277: (12)
278: (16)
279: (16)
280: (16)
281: (20)
282: (16)
283: (12)
284: (12)
285: (12)
286: (12)
287: (12)
288: (12)
289: (4)          @pytest.mark.parametrize(
290: (12)            ["from_dt", "to_dt", "expected_casting", "expected_view_off",
291: (13)              "nom", "denom"],
292: (12)            [("M8[ns]", None, Casting.no, 0, 1, 1),
293: (13)              (str(np.dtype("M8[ns]").newbyteorder()), None,
294: (18)                Casting.equiv, None, 1, 1),
295: (13)              ("M8", "M8[ms]", Casting.safe, 0, 1, 1),
296: (13)              ("M8[ms]", "M8", Casting.unsafe, None, 1, 1),
297: (13)              ("M8[5ms]", "M8[5ms]", Casting.no, 0, 1, 1),

```

```

298: (13)          ("M8[ns]", "M8[ms]", Casting.same_kind, None, 1, 10**6),
299: (13)          ("M8[ms]", "M8[ns]", Casting.safe, None, 10**6, 1),
300: (13)          ("M8[ms]", "M8[7ms]", Casting.same_kind, None, 1, 7),
301: (13)          ("M8[4D]", "M8[1M]", Casting.same_kind, None, None,
302: (18)             [-2**63, 0, -1, 1314, -1315, 564442610]),
303: (13)          ("m8[ns]", None, Casting.no, 0, 1, 1),
304: (13)          (str(np.dtype("m8[ns]").newbyteorder()), None,
305: (18)             Casting.equiv, None, 1, 1),
306: (13)          ("m8", "m8[ms]", Casting.safe, 0, 1, 1),
307: (13)          ("m8[ms]", "m8", Casting.unsafe, None, 1, 1),
308: (13)          ("m8[5ms]", "m8[5ms]", Casting.no, 0, 1, 1),
309: (13)          ("m8[ns]", "m8[ms]", Casting.same_kind, None, 1, 10**6),
310: (13)          ("m8[ms]", "m8[ns]", Casting.safe, None, 10**6, 1),
311: (13)          ("m8[ms]", "m8[7ms]", Casting.same_kind, None, 1, 7),
312: (13)          ("m8[4D]", "m8[1M]", Casting.unsafe, None, None,
313: (18)             [-2**63, 0, 0, 1314, -1315, 564442610]))]
314: (4)           def test_time_to_time(self, from_dt, to_dt,
315: (26)                         expected_casting, expected_view_off,
316: (26)                         nom, denom):
317: (8)             from_dt = np.dtype(from_dt)
318: (8)             if to_dt is not None:
319: (12)                 to_dt = np.dtype(to_dt)
320: (8)             values = np.array([-2**63, 1, 2**63-1, 10000, -10000, 2**32])
321: (8)
values.astype(np.dtype("int64").newbyteorder(from_dt.byteorder))
322: (8)             assert values.dtype.byteorder == from_dt.byteorder
323: (8)             assert np.isnat(values.view(from_dt)[0])
324: (8)             DType = type(from_dt)
325: (8)             cast = get_castingimpl(DType, DType)
326: (8)             casting, (from_res, to_res), view_off = cast._resolve_descriptors(
327: (16)               (from_dt, to_dt))
328: (8)             assert from_res is from_dt
329: (8)             assert to_res is to_dt or to_dt is None
330: (8)             assert casting == expected_casting
331: (8)             assert view_off == expected_view_off
332: (8)             if nom is not None:
333: (12)                 expected_out = (values * nom // denom).view(to_res)
334: (12)                 expected_out[0] = "NaT"
335: (8)             else:
336: (12)                 expected_out = np.empty_like(values)
337: (12)                 expected_out[...] = denom
338: (12)                 expected_out = expected_out.view(to_dt)
339: (8)                 orig_arr = values.view(from_dt)
340: (8)                 orig_out = np.empty_like(expected_out)
341: (8)                 if casting == Casting.unsafe and (to_dt == "m8" or to_dt == "M8"):
342: (12)                     with pytest.raises(ValueError):
343: (16)                         cast._simple_strided_call((orig_arr, orig_out))
344: (12)                     return
345: (8)                 for aligned in [True, True]:
346: (12)                     for contig in [True, True]:
347: (16)                         arr, out = self.get_data_variation(
348: (24)                           orig_arr, orig_out, aligned, contig)
349: (16)                         out[...] = 0
350: (16)                         cast._simple_strided_call((arr, out))
351: (16)                         assert_array_equal(out.view("int64"),
expected_out.view("int64")))
352: (4)             def string_with_modified_length(self, dtype, change_length):
353: (8)                 fact = 1 if dtype.char == "S" else 4
354: (8)                 length = dtype.itemsize // fact + change_length
355: (8)                 return np.dtype(f"{dtype.byteorder}{dtype.char}{length}")
356: (4)             @pytest.mark.parametrize("other_DT", simple_dtypes)
357: (4)             @pytest.mark.parametrize("string_char", ["S", "U"])
358: (4)             def test_string_cancast(self, other_DT, string_char):
359: (8)                 fact = 1 if string_char == "S" else 4
360: (8)                 string_DT = type(np.dtype(string_char))
361: (8)                 cast = get_castingimpl(other_DT, string_DT)
362: (8)                 other_dt = other_DT()
363: (8)                 expected_length = get_expected_stringlength(other_dt)
364: (8)                 string_dt = np.dtype(f"{string_char}{expected_length}")

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

365: (8)           safety, (res_other_dt, res_dt), view_off = cast._resolve_descriptors(
366: (16)             (other_dt, None))
367: (8)           assert res_dt.itemsize == expected_length * fact
368: (8)           assert safety == Casting.safe # we consider to string casts "safe"
369: (8)           assert view_off is None
370: (8)           assert isinstance(res_dt, string_DT)
371: (8)           for change_length in [-1, 0, 1]:
372: (12)             if change_length >= 0:
373: (16)               expected_safety = Casting.safe
374: (12)             else:
375: (16)               expected_safety = Casting.same_kind
376: (12)             to_dt = self.string_with_modified_length(string_dt, change_length)
377: (12)             safety, (_, res_dt), view_off = cast._resolve_descriptors(
378: (20)               (other_dt, to_dt))
379: (12)             assert res_dt is to_dt
380: (12)             assert safety == expected_safety
381: (12)             assert view_off is None
382: (8)           cast = get_castingimpl(string_DT, other_DT)
383: (8)           safety, _, view_off = cast._resolve_descriptors((string_dt, other_dt))
384: (8)           assert safety == Casting.unsafe
385: (8)           assert view_off is None
386: (8)           cast = get_castingimpl(string_DT, other_DT)
387: (8)           safety, (_, res_dt), view_off = cast._resolve_descriptors(
388: (12)             (string_dt, None))
389: (8)           assert safety == Casting.unsafe
390: (8)           assert view_off is None
391: (8)           assert other_dt is res_dt # returns the singleton for simple dtypes
392: (4)           @pytest.mark.parametrize("string_char", ["S", "U"])
393: (4)           @pytest.mark.parametrize("other_dt", simple_dtype_instances())
394: (4)           def test_simple_string_casts_roundtrip(self, other_dt, string_char):
395: (8)             """
396: (8)             Tests casts from and to string by checking the roundtripping property.
397: (8)             The test also covers some string to string casts (but not all).
398: (8)             If this test creates issues, it should possibly just be simplified
399: (8)             or even removed (checking whether unaligned/non-contiguous casts give
400: (8)             the same results is useful, though).
401: (8)             """
402: (8)           string_DT = type(np.dtype(string_char))
403: (8)           cast = get_castingimpl(type(other_dt), string_DT)
404: (8)           cast_back = get_castingimpl(string_DT, type(other_dt))
405: (8)           _, (res_other_dt, string_dt), _ = cast._resolve_descriptors(
406: (16)             (other_dt, None))
407: (8)           if res_other_dt is not other_dt:
408: (12)             assert other_dt.byteorder != res_other_dt.byteorder
409: (12)             return
410: (8)           orig_arr, values = self.get_data(other_dt, None)
411: (8)           str_arr = np.zeros(len(orig_arr), dtype=string_dt)
412: (8)           string_dt_short = self.string_with_modified_length(string_dt, -1)
413: (8)           str_arr_short = np.zeros(len(orig_arr), dtype=string_dt_short)
414: (8)           string_dt_long = self.string_with_modified_length(string_dt, 1)
415: (8)           str_arr_long = np.zeros(len(orig_arr), dtype=string_dt_long)
416: (8)           assert not cast._supports_unaligned # if support is added, should
test
417: (8)           assert not cast_back._supports_unaligned
418: (8)           for contig in [True, False]:
419: (12)             other_arr, str_arr = self.get_data_variation(
420: (16)               orig_arr, str_arr, True, contig)
421: (12)             _, str_arr_short = self.get_data_variation(
422: (16)               orig_arr, str_arr_short.copy(), True, contig)
423: (12)             _, str_arr_long = self.get_data_variation(
424: (16)               orig_arr, str_arr_long, True, contig)
425: (12)             cast._simple_strided_call((other_arr, str_arr))
426: (12)             cast._simple_strided_call((other_arr, str_arr_short))
427: (12)             assert_array_equal(str_arr.astype(string_dt_short), str_arr_short)
428: (12)             cast._simple_strided_call((other_arr, str_arr_long))
429: (12)             assert_array_equal(str_arr, str_arr_long)
430: (12)             if other_dt.kind == "b":
431: (16)               continue
432: (12)             other_arr[...] = 0

```

```

433: (12)           cast_back._simple_strided_call((str_arr, other_arr))
434: (12)           assert_array_equal(orig_arr, other_arr)
435: (12)           other_arr[...] = 0
436: (12)           cast_back._simple_strided_call((str_arr_long, other_arr))
437: (12)           assert_array_equal(orig_arr, other_arr)
438: (4)            @pytest.mark.parametrize("other_dt", ["S8", "<U8", ">U8"])
439: (4)            @pytest.mark.parametrize("string_char", ["S", "U"])
440: (4)            def test_string_to_string_cancast(self, other_dt, string_char):
441: (8)              other_dt = np.dtype(other_dt)
442: (8)              fact = 1 if string_char == "S" else 4
443: (8)              div = 1 if other_dt.char == "S" else 4
444: (8)              string_DT = type(np.dtype(string_char))
445: (8)              cast = get_castingimpl(type(other_dt), string_DT)
446: (8)              expected_length = other_dt.itemsize // div
447: (8)              string_dt = np.dtype(f"{string_char}{expected_length}")
448: (8)              safety, (res_other_dt, res_dt), view_off = cast._resolve_descriptors(
449: (16)                (other_dt, None))
450: (8)              assert res_dt.itemsize == expected_length * fact
451: (8)              assert isinstance(res_dt, string_DT)
452: (8)              expected_view_off = None
453: (8)              if other_dt.char == string_char:
454: (12)                if other_dt.isnative:
455: (16)                  expected_safety = Casting.no
456: (16)                  expected_view_off = 0
457: (12)                else:
458: (16)                  expected_safety = Casting.equiv
459: (8)                elif string_char == "U":
460: (12)                  expected_safety = Casting.safe
461: (8)                else:
462: (12)                  expected_safety = Casting.unsafe
463: (8)                assert view_off == expected_view_off
464: (8)                assert expected_safety == safety
465: (8)                for change_length in [-1, 0, 1]:
466: (12)                  to_dt = self.string_with_modified_length(string_dt, change_length)
467: (12)                  safety, (_, res_dt), view_off = cast._resolve_descriptors(
468: (20)                    (other_dt, to_dt))
469: (12)                  assert res_dt is to_dt
470: (12)                  if change_length <= 0:
471: (16)                      assert view_off == expected_view_off
472: (12)                  else:
473: (16)                      assert view_off is None
474: (12)                  if expected_safety == Casting.unsafe:
475: (16)                      assert safety == expected_safety
476: (12)                  elif change_length < 0:
477: (16)                      assert safety == Casting.same_kind
478: (12)                  elif change_length == 0:
479: (16)                      assert safety == expected_safety
480: (12)                  elif change_length > 0:
481: (16)                      assert safety == Casting.safe
482: (4)            @pytest.mark.parametrize("order1", [">>", "<"])
483: (4)            @pytest.mark.parametrize("order2", [">>", "<"])
484: (4)            def test_unicode_byteswapped_cast(self, order1, order2):
485: (8)              dtype1 = np.dtype(f"{order1}U30")
486: (8)              dtype2 = np.dtype(f"{order2}U30")
487: (8)              data1 = np.empty(30 * 4 + 1, dtype=np.uint8)[1:].view(dtype1)
488: (8)              data2 = np.empty(30 * 4 + 1, dtype=np.uint8)[1:].view(dtype2)
489: (8)              if dtype1.alignment != 1:
490: (12)                  assert not data1.flags.aligned
491: (12)                  assert not data2.flags.aligned
492: (8)                  element = "this is a %nicode string%"
493: (8)                  data1[()] = element
494: (8)                  for data in [data1, data1.copy()]:
495: (12)                      data2[...] = data1
496: (12)                      assert data2[()] == element
497: (12)                      assert data2.copy()[()] == element
498: (4)            def test_void_to_string_special_case(self):
499: (8)              assert np.array([], dtype="V5").astype("S").dtype.itemsize == 5
500: (8)              assert np.array([], dtype="V5").astype("U").dtype.itemsize == 4 * 5
501: (4)            def test_object_to_parametric_internal_error(self):

```

```

502: (8)          object_dtype = type(np.dtype(object))
503: (8)          other_dtype = type(np.dtype(str))
504: (8)          cast = get_castingimpl(object_dtype, other_dtype)
505: (8)          with pytest.raises(TypeError,
506: (20)                  match="casting from object to the parametric DType"):
507: (12)                  cast._resolve_descriptors((np.dtype("0"), None))
508: (4)          @pytest.mark.parametrize("dtype", simple_dtype_instances())
509: (4)          def test_object_and_simple_resolution(self, dtype):
510: (8)              object_dtype = type(np.dtype(object))
511: (8)              cast = get_castingimpl(object_dtype, type(dtype))
512: (8)              safety, (_, res_dt), view_off = cast._resolve_descriptors(
513: (16)                  (np.dtype("0"), dtype))
514: (8)              assert safety == Casting.unsafe
515: (8)              assert view_off is None
516: (8)              assert res_dt is dtype
517: (8)              safety, (_, res_dt), view_off = cast._resolve_descriptors(
518: (16)                  (np.dtype("0"), None))
519: (8)              assert safety == Casting.unsafe
520: (8)              assert view_off is None
521: (8)              assert res_dt == dtype.newbyteorder("=")
522: (4)          @pytest.mark.parametrize("dtype", simple_dtype_instances())
523: (4)          def test_simple_to_object_resolution(self, dtype):
524: (8)              object_dtype = type(np.dtype(object))
525: (8)              cast = get_castingimpl(type(dtype), object_dtype)
526: (8)              safety, (_, res_dt), view_off = cast._resolve_descriptors(
527: (16)                  (dtype, None))
528: (8)              assert safety == Casting.safe
529: (8)              assert view_off is None
530: (8)              assert res_dt is np.dtype("0")
531: (4)          @pytest.mark.parametrize("casting", ["no", "unsafe"])
532: (4)          def test_void_and_structured_with_subarray(self, casting):
533: (8)              dtype = np.dtype([("foo", "<f4", (3, 2))])
534: (8)              expected = casting == "unsafe"
535: (8)              assert np.can_cast("V4", dtype, casting=casting) == expected
536: (8)              assert np.can_cast(dtype, "V4", casting=casting) == expected
537: (4)          @pytest.mark.parametrize(["to_dt", "expected_off"],
538: (12)                  [ # Same as `from_dt` but with both fields shifted:
539: (13)                      (np.dtype({"names": ["a", "b"], "formats": ["i4", "f4"],
540: (24)                          "offsets": [0, 4]}), 2),
541: (13)                      (np.dtype({"names": ["b", "a"], "formats": ["i4", "f4"],
542: (24)                          "offsets": [0, 4]}), 2),
543: (13)                      (np.dtype({"names": ["b", "a"], "formats": ["i4", "f4"],
544: (24)                          "offsets": [0, 6]}), None)])
545: (4)          def test_structured_field_offsets(self, to_dt, expected_off):
546: (8)              from_dt = np.dtype({"names": ["a", "b"],
547: (28)                  "formats": ["i4", "f4"],
548: (28)                  "offsets": [2, 6]})
549: (8)              cast = get_castingimpl(type(from_dt), type(to_dt))
550: (8)              safety, _, view_off = cast._resolve_descriptors((from_dt, to_dt))
551: (8)              if from_dt.names == to_dt.names:
552: (12)                  assert safety == Casting.equiv
553: (8)              else:
554: (12)                  assert safety == Casting.safe
555: (8)              assert view_off == expected_off
556: (4)          @pytest.mark.parametrize(("from_dt", "to_dt", "expected_off"), [
557: (12)                  ("i", "(1,1)i", 0),
558: (12)                  ("(1,1)i", "i", 0),
559: (12)                  ("(2,1)i", "(2,1)i", 0),
560: (12)                  ("i", dict(names=["a"], formats=["i"], offsets=[2]), None),
561: (12)                  (dict(names=["a"], formats=["i"], offsets=[2]), "i", 2),
562: (12)                  ("i", dict(names=["a", "b"], formats=["i", "i"], offsets=[2, 2]),
563: (13)                      None),
564: (12)                  ("i,i", "i,i,i", None),
565: (12)                  ("i4", "V3", 0), # void smaller or equal
566: (12)                  ("i4", "V4", 0), # void smaller or equal
567: (12)                  ("i4", "V10", None), # void is larger (no view)
568: (12)                  ("O", "V4", None), # currently reject objects for view here.
569: (12)                  ("O", "V8", None), # currently reject objects for view here.
570: (12)                  ("V4", "V3", 0),

```

```

571: (12)                 ("V4", "V4", 0),
572: (12)                 ("V3", "V4", None),
573: (12)                 ("V4", "i4", None),
574: (12)                 ("i,i", "i,i,i", None),
575: (8)                  ])
576: (4)      def test_structured_view_offsets_paramteric(
577: (12)          self, from_dt, to_dt, expected_off):
578: (8)          from_dt = np.dtype(from_dt)
579: (8)          to_dt = np.dtype(to_dt)
580: (8)          cast = get_castingimpl(type(from_dt), type(to_dt))
581: (8)          _, _, view_off = cast._resolve_descriptors((from_dt, to_dt))
582: (8)          assert view_off == expected_off
583: (4) @pytest.mark.parametrize("dtype", np.typecodes["All"])
584: (4)      def test_object_casts_NULL_None_equivalence(self, dtype):
585: (8)          arr_normal = np.array([None] * 5)
586: (8)          arr_NULLs = np.empty_like(arr_normal)
587: (8)          ctypes.memset(arr_NULLs.ctypes.data, 0, arr_NULLs.nbytes)
588: (8)          assert arr_NULLs.tobytes() == b"\x00" * arr_NULLs.nbytes
589: (8)          try:
590: (12)              expected = arr_normal.astype(dtype)
591: (8)          except TypeError:
592: (12)              with pytest.raises(TypeError):
593: (16)                  arr_NULLs.astype(dtype),
594: (8)          else:
595: (12)              assert_array_equal(expected, arr_NULLs.astype(dtype))
596: (4) @pytest.mark.parametrize("dtype",
597: (12)      np.typecodes["AllInteger"] + np.typecodes["AllFloat"])
598: (4)      def test_nonstandard_bool_to_other(self, dtype):
599: (8)          nonstandard_bools = np.array([0, 3, -7], dtype=np.int8).view(bool)
600: (8)          res = nonstandard_bools.astype(dtype)
601: (8)          expected = [0, 1, 1]
602: (8)          assert_array_equal(res, expected)

```

---

## File 81 - test\_casting\_floatingpoint\_errors.py:

```

1: (0)          import pytest
2: (0)          from pytest import param
3: (0)          from numpy.testing import IS_WASM
4: (0)          import numpy as np
5: (0)          def values_and_dtypes():
6: (4)              """
7: (4)                  Generate value+dtype pairs that generate floating point errors during
8: (4)                  casts. The invalid casts to integers will generate "invalid" value
9: (4)                  warnings, the float casts all generate "overflow".
10: (4)                  (The Python int/float paths don't need to get tested in all the same
11: (4)                  situations, but it does not hurt.)
12: (4)              """
13: (4)              yield param(70000, "float16", id="int-to-f2")
14: (4)              yield param("70000", "float16", id="str-to-f2")
15: (4)              yield param(70000.0, "float16", id="float-to-f2")
16: (4)              yield param(np.longdouble(70000.), "float16", id="longdouble-to-f2")
17: (4)              yield param(np.float64(70000.), "float16", id="double-to-f2")
18: (4)              yield param(np.float32(70000.), "float16", id="float-to-f2")
19: (4)              yield param(10**100, "float32", id="int-to-f4")
20: (4)              yield param(1e100, "float32", id="float-to-f2")
21: (4)              yield param(np.longdouble(1e300), "float32", id="longdouble-to-f2")
22: (4)              yield param(np.float64(1e300), "float32", id="double-to-f2")
23: (4)              max_ld = np.finfo(np.longdouble).max
24: (4)              spacing = np.spacing(np.nextafter(np.finfo("f8").max, 0))
25: (4)              if max_ld - spacing > np.finfo("f8").max:
26: (8)                  yield param(np.finfo(np.longdouble).max, "float64",
27: (20)                      id="longdouble-to-f8")
28: (4)              yield param(2e300, "complex64", id="float-to-c8")
29: (4)              yield param(2e300+0j, "complex64", id="complex-to-c8")
30: (4)              yield param(2e300j, "complex64", id="complex-to-c8")
31: (4)              yield param(np.longdouble(2e300), "complex64", id="longdouble-to-c8")
32: (4)              with np.errstate(over="ignore"):

```

```

33: (8)           for to_dt in np.typecodes["AllInteger"]:
34: (12)             for value in [np.inf, np.nan]:
35: (16)               for from_dt in np.typecodes["AllFloat"]:
36: (20)                 from_dt = np.dtype(from_dt)
37: (20)                 from_val = from_dt.type(value)
38: (20)                 yield param(from_val, to_dt, id=f"{from_val}-to-{to_dt}")
39: (0)           def check_operations(dtype, value):
40: (4)             """
41: (4)             There are many dedicated paths in NumPy which cast and should check for
42: (4)             floating point errors which occurred during those casts.
43: (4)             """
44: (4)             if dtype.kind != 'i':
45: (8)               def assignment():
46: (12)                 arr = np.empty(3, dtype=dtype)
47: (12)                 arr[0] = value
48: (8)                 yield assignment
49: (8)               def fill():
50: (12)                 arr = np.empty(3, dtype=dtype)
51: (12)                 arr.fill(value)
52: (8)                 yield fill
53: (4)             def copyto_scalar():
54: (8)               arr = np.empty(3, dtype=dtype)
55: (8)               np.copyto(arr, value, casting="unsafe")
56: (4)             yield copyto_scalar
57: (4)             def copyto():
58: (8)               arr = np.empty(3, dtype=dtype)
59: (8)               np.copyto(arr, np.array([value, value, value]), casting="unsafe")
60: (4)             yield copyto
61: (4)             def copyto_scalar_masked():
62: (8)               arr = np.empty(3, dtype=dtype)
63: (8)               np.copyto(arr, value, casting="unsafe",
64: (18)                 where=[True, False, True])
65: (4)             yield copyto_scalar_masked
66: (4)             def copyto_masked():
67: (8)               arr = np.empty(3, dtype=dtype)
68: (8)               np.copyto(arr, np.array([value, value, value]), casting="unsafe",
69: (18)                 where=[True, False, True])
70: (4)             yield copyto_masked
71: (4)             def direct_cast():
72: (8)               np.array([value, value, value]).astype(dtype)
73: (4)             yield direct_cast
74: (4)             def direct_cast_nd_strided():
75: (8)               arr = np.full((5, 5, 5), fill_value=value)[:, ::2, :]
76: (8)               arr.astype(dtype)
77: (4)             yield direct_cast_nd_strided
78: (4)             def boolean_array_assignment():
79: (8)               arr = np.empty(3, dtype=dtype)
80: (8)               arr[[True, False, True]] = np.array([value, value])
81: (4)             yield boolean_array_assignment
82: (4)             def integer_array_assignment():
83: (8)               arr = np.empty(3, dtype=dtype)
84: (8)               values = np.array([value, value])
85: (8)               arr[[0, 1]] = values
86: (4)             yield integer_array_assignment
87: (4)             def integer_array_assignment_with_subspace():
88: (8)               arr = np.empty((5, 3), dtype=dtype)
89: (8)               values = np.array([value, value, value])
90: (8)               arr[[0, 2]] = values
91: (4)             yield integer_array_assignment_with_subspace
92: (4)             def flat_assignment():
93: (8)               arr = np.empty((3,), dtype=dtype)
94: (8)               values = np.array([value, value, value])
95: (8)               arr.flat[:] = values
96: (4)             yield flat_assignment
97: (0) @pytest.mark.skipif(IS_WASM, reason="no wasm fp exception support")
98: (0) @pytest.mark.parametrize(["value", "dtype"], values_and_dtypes())
99: (0) @pytest.mark.filterwarnings("ignore::numpy.ComplexWarning")
100: (0) def test_floatingpoint_errors_casting(dtype, value):
101: (4)   dtype = np.dtype(dtype)

```

```

102: (4)             for operation in check_operations(dtype, value):
103: (8)                 dtype = np.dtype(dtype)
104: (8)                 match = "invalid" if dtype.kind in 'iu' else "overflow"
105: (8)                 with pytest.warns(RuntimeWarning, match=match):
106: (12)                     operation()
107: (8)                 with np.errstate(all="raise"):
108: (12)                     with pytest.raises(FloatingPointError, match=match):
109: (16)                         operation()

-----

```

## File 82 - test\_cpu\_dispatcher.py:

```

1: (0)             from numpy.core._multiarray_umath import __cpu_features__, __cpu_baseline__,
2: (0)             __cpu_dispatch__
3: (0)             from numpy.core import _umath_tests
4: (0)             from numpy.testing import assert_equal
5: (4)             def test_dispatcher():
6: (4)                 """
7: (4)                     Testing the utilities of the CPU dispatcher
8: (4)                 """
9: (8)                 targets = (
10: (8)                     "SSE2", "SSE41", "AVX2",
11: (8)                     "VSX", "VSX2", "VSX3",
12: (8)                     "NEON", "ASIMD", "ASIMDHP",
13: (8)                     "VX", "VXE"
14: (4)                 )
15: (4)                 highest_sfx = "" # no suffix for the baseline
16: (4)                 all_sfx = []
17: (8)                 for feature in reversed(targets):
18: (12)                     if feature in __cpu_baseline__:
19: (8)                         continue
20: (12)                     if feature not in __cpu_dispatch__ or not __cpu_features__[feature]:
21: (8)                         continue
22: (12)                     if not highest_sfx:
23: (8)                         highest_sfx = "_" + feature
24: (4)                     all_sfx.append("func" + "_" + feature)
25: (4)                     test = _umath_tests.test_dispatch()
26: (4)                     assert_equal(test["func"], "func" + highest_sfx)
27: (4)                     assert_equal(test["var"], "var" + highest_sfx)
28: (8)                     if highest_sfx:
29: (8)                         assert_equal(test["func_xb"], "func" + highest_sfx)
30: (8)                         assert_equal(test["var_xb"], "var" + highest_sfx)
31: (4)                     else:
32: (8)                         assert_equal(test["func_xb"], "nobase")
33: (8)                         assert_equal(test["var_xb"], "nobase")
34: (4)                     all_sfx.append("func") # add the baseline
35: (4)                     assert_equal(test["all"], all_sfx)

-----

```

## File 83 - test\_conversion\_utils.py:

```

1: (0)             """
2: (0)             Tests for numpy/core/src/multiarray/conversion_utils.c
3: (0)             """
4: (0)             import re
5: (0)             import sys
6: (0)             import pytest
7: (0)             import numpy as np
8: (0)             import numpy.core._multiarray_tests as mt
9: (0)             from numpy.testing import assert_warns, IS_PYPY
10: (0)             class StringConverterTestCase:
11: (4)                 allow_bytes = True
12: (4)                 case_insensitive = True
13: (4)                 exact_match = False
14: (4)                 warn = True
15: (4)                 def _check_value_error(self, val):
16: (8)                     pattern = r'\\(got {}\\)'.format(re.escape(repr(val)))


```

```

17: (8)           with pytest.raises(ValueError, match=pattern) as exc:
18: (12)             self.conv(val)
19: (4)           def _check_conv_assert_warn(self, val, expected):
20: (8)             if self.warn:
21: (12)               with assert_warnings(DeprecationWarning) as exc:
22: (16)                 assert self.conv(val) == expected
23: (8)             else:
24: (12)               assert self.conv(val) == expected
25: (4)           def _check(self, val, expected):
26: (8)             """Takes valid non-deprecated inputs for converters,
27: (8)             runs converters on inputs, checks correctness of outputs,
28: (8)             warnings and errors"""
29: (8)             assert self.conv(val) == expected
30: (8)             if self.allow_bytes:
31: (12)               assert self.conv(val.encode('ascii')) == expected
32: (8)             else:
33: (12)               with pytest.raises(TypeError):
34: (16)                 self.conv(val.encode('ascii'))
35: (8)             if len(val) != 1:
36: (12)               if self.exact_match:
37: (16)                 self._check_value_error(val[:1])
38: (16)                 self._check_value_error(val + '\0')
39: (12)               else:
40: (16)                 self._check_conv_assert_warn(val[:1], expected)
41: (8)             if self.case_insensitive:
42: (12)               if val != val.lower():
43: (16)                 self._check_conv_assert_warn(val.lower(), expected)
44: (12)               if val != val.upper():
45: (16)                 self._check_conv_assert_warn(val.upper(), expected)
46: (8)             else:
47: (12)               if val != val.lower():
48: (16)                 self._check_value_error(val.lower())
49: (12)               if val != val.upper():
50: (16)                 self._check_value_error(val.upper())
51: (4)           def test_wrong_type(self):
52: (8)             with pytest.raises(TypeError):
53: (12)               self.conv({})
54: (8)             with pytest.raises(TypeError):
55: (12)               self.conv([])
56: (4)           def test_wrong_value(self):
57: (8)             self._check_value_error('')
58: (8)             self._check_value_error('\N{greek small letter pi}')
59: (8)             if self.allow_bytes:
60: (12)               self._check_value_error(b'')
61: (12)               self._check_value_error(b"\xFF")
62: (8)             if self.exact_match:
63: (12)               self._check_value_error("there's no way this is supported")
64: (0)           class TestByteorderConverter(StringConverterTestCase):
65: (4)             """ Tests of PyArray_ByteorderConverter """
66: (4)             conv = mt.run_byteorder_converter
67: (4)             warn = False
68: (4)             def test_valid(self):
69: (8)               for s in ['big', '>']:
70: (12)                 self._check(s, 'NPY_BIG')
71: (8)               for s in ['little', '<']:
72: (12)                 self._check(s, 'NPY_LITTLE')
73: (8)               for s in ['native', '=']:
74: (12)                 self._check(s, 'NPY_NATIVE')
75: (8)               for s in ['ignore', '|']:
76: (12)                 self._check(s, 'NPY_IGNORE')
77: (8)               for s in ['swap']:
78: (12)                 self._check(s, 'NPY_SWAP')
79: (0)           class TestSortkindConverter(StringConverterTestCase):
80: (4)             """ Tests of PyArray_SortkindConverter """
81: (4)             conv = mt.run_sortkind_converter
82: (4)             warn = False
83: (4)             def test_valid(self):
84: (8)               self._check('quicksort', 'NPY_QUICKSORT')
85: (8)               self._check('heapsort', 'NPY_HEAPSORT')

```

```

86: (8)           self._check('mergesort', 'NPY_STABLESORT') # alias
87: (8)           self._check('stable', 'NPY_STABLESORT')
88: (0) class TestSelectkindConverter(StringConverterTestCase):
89: (4)     """ Tests of PyArray_SelectkindConverter """
90: (4)     conv = mt.run_selectkind_converter
91: (4)     case_insensitive = False
92: (4)     exact_match = True
93: (4)     def test_valid(self):
94: (8)         self._check('introselect', 'NPY_INTROSELECT')
95: (0) class TestSearchsideConverter(StringConverterTestCase):
96: (4)     """ Tests of PyArray_SearchsideConverter """
97: (4)     conv = mt.run_searchside_converter
98: (4)     def test_valid(self):
99: (8)         self._check('left', 'NPY_SEARCHLEFT')
100: (8)        self._check('right', 'NPY_SEARCHRIGHT')
101: (0) class TestOrderConverter(StringConverterTestCase):
102: (4)     """ Tests of PyArray_OrderConverter """
103: (4)     conv = mt.run_order_converter
104: (4)     warn = False
105: (4)     def test_valid(self):
106: (8)         self._check('c', 'NPY_CORDER')
107: (8)         self._check('f', 'NPY_FORTRANORDER')
108: (8)         self._check('a', 'NPY_ANYORDER')
109: (8)         self._check('k', 'NPY_KEEPORDER')
110: (4)     def test_flatten_invalid_order(self):
111: (8)         with pytest.raises(ValueError):
112: (12)             self.conv('Z')
113: (8)         for order in [False, True, 0, 8]:
114: (12)             with pytest.raises(TypeError):
115: (16)                 self.conv(order)
116: (0) class TestClipmodeConverter(StringConverterTestCase):
117: (4)     """ Tests of PyArray_ClipmodeConverter """
118: (4)     conv = mt.run_clipmode_converter
119: (4)     def test_valid(self):
120: (8)         self._check('clip', 'NPY_CLIP')
121: (8)         self._check('wrap', 'NPY_WRAP')
122: (8)         self._check('raise', 'NPY_RAISE')
123: (8)         assert self.conv(np.CLIP) == 'NPY_CLIP'
124: (8)         assert self.conv(np.WRAP) == 'NPY_WRAP'
125: (8)         assert self.conv(np.RAISE) == 'NPY_RAISE'
126: (0) class TestCastingConverter(StringConverterTestCase):
127: (4)     """ Tests of PyArray_CastingConverter """
128: (4)     conv = mt.run_casting_converter
129: (4)     case_insensitive = False
130: (4)     exact_match = True
131: (4)     def test_valid(self):
132: (8)         self._check("no", "NPY_NO_CASTING")
133: (8)         self._check("equiv", "NPY_EQUIV_CASTING")
134: (8)         self._check("safe", "NPY_SAFE_CASTING")
135: (8)         self._check("same_kind", "NPY_SAME_KIND_CASTING")
136: (8)         self._check("unsafe", "NPY_UNSAFE_CASTING")
137: (0) class TestIntpConverter:
138: (4)     """ Tests of PyArray_IntpConverter """
139: (4)     conv = mt.run_intp_converter
140: (4)     def test_basic(self):
141: (8)         assert self.conv(1) == (1,)
142: (8)         assert self.conv((1, 2)) == (1, 2)
143: (8)         assert self.conv([1, 2]) == (1, 2)
144: (8)         assert self.conv(())) == ()
145: (4)     def test_none(self):
146: (8)         with pytest.warns(DeprecationWarning):
147: (12)             assert self.conv(None) == ()
148: (4)         @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
149: (12)                         reason="PyPy bug in error formatting")
150: (4)         def test_float(self):
151: (8)             with pytest.raises(TypeError):
152: (12)                 self.conv(1.0)
153: (8)             with pytest.raises(TypeError):
154: (12)                 self.conv([1, 1.0])

```

```

155: (4)         def test_too_large(self):
156: (8)             with pytest.raises(ValueError):
157: (12)                 self.conv(2**64)
158: (4)         def test_too_many_dims(self):
159: (8)             assert self.conv([1]*32) == (1,)*32
160: (8)             with pytest.raises(ValueError):
161: (12)                 self.conv([1]*33)

```

---

## File 84 - test\_cpu\_features.py:

```

1: (0)         import sys, platform, re, pytest
2: (0)         from numpy.core._multiarray_umath import (
3: (4)             __cpu_features__,
4: (4)             __cpu_baseline__,
5: (4)             __cpu_dispatch__,
6: (0)         )
7: (0)         import numpy as np
8: (0)         import subprocess
9: (0)         import pathlib
10: (0)        import os
11: (0)        import re
12: (0)        def assert_features_equal(actual, desired, fname):
13: (4)            __tracebackhide__ = True # Hide traceback for py.test
14: (4)            actual, desired = str(actual), str(desired)
15: (4)            if actual == desired:
16: (8)                return
17: (4)            detected = str(__cpu_features__).replace("''", "")
18: (4)            try:
19: (8)                with open("/proc/cpuinfo") as fd:
20: (12)                    cpuinfo = fd.read(2048)
21: (4)            except Exception as err:
22: (8)                cpuinfo = str(err)
23: (4)            try:
24: (8)                import subprocess
25: (8)                auxv = subprocess.check_output(['/bin/true'],
env=dict(LD_SHOW_AUXV="1"))
26: (8)                auxv = auxv.decode()
27: (4)            except Exception as err:
28: (8)                auxv = str(err)
29: (4)            import textwrap
30: (4)            error_report = textwrap.indent(
31: (0)                """
32: (0)                -----
33: (0)                --- NumPy Detections
34: (0)                -----
35: (0)                %s
36: (0)                -----
37: (0)                --- SYS / CPUINFO
38: (0)                -----
39: (0)                %s....
40: (0)                -----
41: (0)                --- SYS / AUXV
42: (0)                -----
43: (0)                %s
44: (0)                """ % (detected, cpuinfo, auxv), prefix='\r')
45: (4)            raise AssertionError((
46: (8)                "Failure Detection\n"
47: (8)                " NAME: '%s'\n"
48: (8)                " ACTUAL: %s\n"
49: (8)                " DESIRED: %s\n"
50: (8)                "%s"
51: (4)            ) % (fname, actual, desired, error_report))
52: (0)        def _text_to_list(txt):
53: (4)            out = txt.strip("][\n").replace("''", "").split(', ')
54: (4)            return None if out[0] == "" else out
55: (0)        class AbstractTest:
56: (4)            features = []

```

```

57: (4)             features_groups = {}
58: (4)             features_map = {}
59: (4)             features_flags = set()
60: (4)             def load_flags(self):
61: (8)                 pass
62: (4)             def test_features(self):
63: (8)                 self.load_flags()
64: (8)                 for gname, features in self.features_groups.items():
65: (12)                     test_features = [self.cpu_have(f) for f in features]
66: (12)                     assert_features_equal(__cpu_features__.get(gname),
all(test_features), gname)
67: (8)             for feature_name in self.features:
68: (12)                 cpu_have = self.cpu_have(feature_name)
69: (12)                 npy_have = __cpu_features__.get(feature_name)
70: (12)                 assert_features_equal(npy_have, cpu_have, feature_name)
71: (4)             def cpu_have(self, feature_name):
72: (8)                 map_names = self.features_map.get(feature_name, feature_name)
73: (8)                 if isinstance(map_names, str):
74: (12)                     return map_names in self.features_flags
75: (8)                 for f in map_names:
76: (12)                     if f in self.features_flags:
77: (16)                         return True
78: (8)                 return False
79: (4)             def load_flags_cpuinfo(self, magic_key):
80: (8)                 self.features_flags = self.get_cpuinfo_item(magic_key)
81: (4)             def get_cpuinfo_item(self, magic_key):
82: (8)                 values = set()
83: (8)                 with open('/proc/cpuinfo') as fd:
84: (12)                     for line in fd:
85: (16)                         if not line.startswith(magic_key):
86: (20)                             continue
87: (16)                         flags_value = [s.strip() for s in line.split(':', 1)]
88: (16)                         if len(flags_value) == 2:
89: (20)                             values = values.union(flags_value[1].upper().split())
90: (8)                     return values
91: (4)             def load_flags_auxv(self):
92: (8)                 auxv = subprocess.check_output(['/bin>true'],
env=dict(LD_SHOW_AUXV="1"))
93: (8)
94: (12)
95: (16)
96: (12)
97: (12)
98: (16)
99: (20)
100: (16)
101: (0)             @pytest.mark.skipif(
102: (4)                 sys.platform == 'emscripten',
103: (4)                 reason= (
104: (8)                     "The subprocess module is not available on WASM platforms and"
105: (8)                     " therefore this test class cannot be properly executed."
106: (4)
107: (0)
108: (0)
109: (4)             )
110: (4)             class TestEnvPrivation:
111: (4)                 cwd = pathlib.Path(__file__).parent.resolve()
112: (4)                 env = os.environ.copy()
113: (4)                 _enable = os.environ.pop('NPY_ENABLE_CPU_FEATURES', None)
114: (4)                 _disable = os.environ.pop('NPY_DISABLE_CPU_FEATURES', None)
115: (4)                 SUBPROCESS_ARGS = dict(cwd=cwd, capture_output=True, text=True,
check=True)
116: (4)                 unavailable_feats = [
117: (8)                     feat for feat in __cpu_dispatch__ if not __cpu_features__[feat]
118: (4)                 ]
119: (4)                 UNAVAILABLE_FEAT = (
120: (8)                     None if len(unavailable_feats) == 0
121: (8)                     else unavailable_feats[0]
122: (4)                 )
123: (4)                 BASELINE_FEAT = None if len(__cpu_baseline__) == 0 else
__cpu_baseline__[0]

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

122: (4)           SCRIPT = """
123: (0)           def main():
124: (4)             from numpy.core._multiarray_umath import __cpu_features__,
125: (4)             detected = [feat for feat in __cpu_dispatch__ if __cpu_features__[feat]]
126: (4)             print(detected)
127: (0)           if __name__ == "__main__":
128: (4)             main()
129: (4)             """
130: (4)           @pytest.fixture(autouse=True)
131: (4)           def setup_class(self, tmp_path_factory):
132: (8)             file = tmp_path_factory.mktemp("runtime_test_script")
133: (8)             file /= "_runtime_detect.py"
134: (8)             file.write_text(self.SCRIPT)
135: (8)             self.file = file
136: (8)             return
137: (4)           def _run(self):
138: (8)             return subprocess.run(
139: (12)               [sys.executable, self.file],
140: (12)               env=self.env,
141: (12)               **self.SUBPROCESS_ARGS,
142: (12)               )
143: (4)           def _expect_error(
144: (8)             self,
145: (8)             msg,
146: (8)             err_type,
147: (8)             no_error_msg="Failed to generate error"
148: (4)           ):
149: (8)             try:
150: (12)               self._run()
151: (8)             except subprocess.CalledProcessError as e:
152: (12)               assertion_message = f"Expected: {msg}\nGot: {e.stderr}"
153: (12)               assert re.search(msg, e.stderr), assertion_message
154: (12)               assertion_message = (
155: (16)                 f"Expected error of type: {err_type}; see full "
156: (16)                 f"error:\n{e.stderr}"
157: (12)               )
158: (12)               assert re.search(err_type, e.stderr), assertion_message
159: (8)             else:
160: (12)               assert False, no_error_msg
161: (4)           def setup_method(self):
162: (8)             """Ensure that the environment is reset"""
163: (8)             self.env = os.environ.copy()
164: (8)             return
165: (4)           def test_runtime_feature_selection(self):
166: (8)             """
167: (8)               Ensure that when selecting `NPY_ENABLE_CPU_FEATURES`, only the
168: (8)               features exactly specified are dispatched.
169: (8)             """
170: (8)             out = self._run()
171: (8)             non_baseline_features = _text_to_list(out.stdout)
172: (8)             if non_baseline_features is None:
173: (12)               pytest.skip(
174: (16)                 "No dispatchable features outside of baseline detected."
175: (12)               )
176: (8)             feature = non_baseline_features[0]
177: (8)             self.env['NPY_ENABLE_CPU_FEATURES'] = feature
178: (8)             out = self._run()
179: (8)             enabled_features = _text_to_list(out.stdout)
180: (8)             assert set(enabled_features) == {feature}
181: (8)             if len(non_baseline_features) < 2:
182: (12)               pytest.skip("Only one non-baseline feature detected.")
183: (8)             self.env['NPY_ENABLE_CPU_FEATURES'] = ",".join(non_baseline_features)
184: (8)             out = self._run()
185: (8)             enabled_features = _text_to_list(out.stdout)
186: (8)             assert set(enabled_features) == set(non_baseline_features)
187: (8)             return
188: (4)           @pytest.mark.parametrize("enabled, disabled",
189: (4)             [

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

190: (8)          ("feature", "feature"),
191: (8)          ("feature", "same"),
192: (4)          ]
193: (4)      def test_both_enable_disable_set(self, enabled, disabled):
194: (8)          """
195: (8)          Ensure that when both environment variables are set then an
196: (8)          ImportError is thrown
197: (8)          """
198: (8)          self.env['NPY_ENABLE_CPU_FEATURES'] = enabled
199: (8)          self.env['NPY_DISABLE_CPU_FEATURES'] = disabled
200: (8)          msg = "Both NPY_DISABLE_CPU_FEATURES and NPY_ENABLE_CPU_FEATURES"
201: (8)          err_type = "ImportError"
202: (8)          self._expect_error(msg, err_type)
203: (4)      @pytest.mark.skipif(
204: (8)          not __cpu_dispatch__,
205: (8)          reason=(
206: (12)          "NPY_*_CPU_FEATURES only parsed if "
207: (12)          "`__cpu_dispatch__` is non-empty"
208: (8)          )
209: (4)      )
210: (4)      @pytest.mark.parametrize("action", ["ENABLE", "DISABLE"])
211: (4)      def test_variable_too_long(self, action):
212: (8)          """
213: (8)          Test that an error is thrown if the environment variables are too long
214: (8)          to be processed. Current limit is 1024, but this may change later.
215: (8)          """
216: (8)          MAX_VAR_LENGTH = 1024
217: (8)          self.env[f'NPY_{action}_CPU_FEATURES'] = "t" * MAX_VAR_LENGTH
218: (8)          msg = (
219: (12)          f"Length of environment variable 'NPY_{action}_CPU_FEATURES' is "
220: (12)          f"{MAX_VAR_LENGTH + 1}, only {MAX_VAR_LENGTH} accepted"
221: (8)          )
222: (8)          err_type = "RuntimeError"
223: (8)          self._expect_error(msg, err_type)
224: (4)      @pytest.mark.skipif(
225: (8)          not __cpu_dispatch__,
226: (8)          reason=(
227: (12)          "NPY_*_CPU_FEATURES only parsed if "
228: (12)          "`__cpu_dispatch__` is non-empty"
229: (8)          )
230: (4)      )
231: (4)      def test_impossible_feature_disable(self):
232: (8)          """
233: (8)          Test that a RuntimeError is thrown if an impossible feature-disabling
234: (8)          request is made. This includes disabling a baseline feature.
235: (8)          """
236: (8)          if self.BASELINE_FEAT is None:
237: (12)              pytest.skip("There are no unavailable features to test with")
238: (8)          bad_feature = self.BASELINE_FEAT
239: (8)          self.env['NPY_DISABLE_CPU_FEATURES'] = bad_feature
240: (8)          msg = (
241: (12)          f"You cannot disable CPU feature '{bad_feature}', since it is "
242: (12)          "part of the baseline optimizations"
243: (8)          )
244: (8)          err_type = "RuntimeError"
245: (8)          self._expect_error(msg, err_type)
246: (4)      def test_impossible_feature_enable(self):
247: (8)          """
248: (8)          Test that a RuntimeError is thrown if an impossible feature-enabling
249: (8)          request is made. This includes enabling a feature not supported by the
250: (8)          machine, or disabling a baseline optimization.
251: (8)          """
252: (8)          if self.UNAVAILABLE_FEAT is None:
253: (12)              pytest.skip("There are no unavailable features to test with")
254: (8)          bad_feature = self.UNAVAILABLE_FEAT
255: (8)          self.env['NPY_ENABLE_CPU_FEATURES'] = bad_feature
256: (8)          msg = (
257: (12)          f"You cannot enable CPU features \\\({bad_feature}\\), since "
258: (12)          "they are not supported by your machine."

```

```

259: (8) )
260: (8)     err_type = "RuntimeError"
261: (8)     self._expect_error(msg, err_type)
262: (8)     feats = f"{{bad_feature}}, {{self.BASELINE_FEAT}}"
263: (8)     self.env['NPY_ENABLE_CPU_FEATURES'] = feats
264: (8)     msg = (
265: (12)         f"You cannot enable CPU features \\\{{bad_feature}\\}, since they "
266: (12)         "are not supported by your machine."
267: (8)     )
268: (8)     self._expect_error(msg, err_type)
269: (0)     is_linux = sys.platform.startswith('linux')
270: (0)     is_cygwin = sys.platform.startswith('cygwin')
271: (0)     machine = platform.machine()
272: (0)     is_x86 = re.match("^(amd64|x86|i386|i686)", machine, re.IGNORECASE)
273: (0)     @pytest.mark.skipif(
274: (4)         not (is_linux or is_cygwin) or not is_x86, reason="Only for Linux and x86"
275: (0)     )
276: (0)     class Test_X86_Features(AbstractTest):
277: (4)         features = [
278: (8)             "MMX", "SSE", "SSE2", "SSE3", "SSSE3", "SSE41", "POPCNT", "SSE42",
279: (8)             "AVX", "F16C", "XOP", "FMA4", "FMA3", "AVX2", "AVX512F", "AVX512CD",
280: (8)             "AVX512ER", "AVX512PF", "AVX5124FMAPS", "AVX5124VNNIW",
281: (8)             "AVX512VL", "AVX512BW", "AVX512DQ", "AVX512VNNI", "AVX512IFMA",
282: (8)             "AVX512VBMI", "AVX512VBMI2", "AVX512BITALG", "AVX512FP16",
283: (4)         ]
284: (4)         features_groups = dict(
285: (8)             AVX512_KNL = ["AVX512F", "AVX512CD", "AVX512ER", "AVX512PF"],
286: (8)             AVX512_KNM = ["AVX512F", "AVX512CD", "AVX512ER", "AVX512PF",
287: (22)                 "AVX5124VNNIW", "AVX512VPOPCNTDQ"],
288: (8)             AVX512_SKX = ["AVX512F", "AVX512CD", "AVX512BW", "AVX512DQ",
289: (8)             AVX512_CLX = ["AVX512F", "AVX512CD", "AVX512BW", "AVX512DQ",
290: (8)             AVX512_CNL = ["AVX512F", "AVX512CD", "AVX512BW", "AVX512DQ",
291: (22)                 "AVX512VBMI"],
292: (8)             AVX512_ICL = ["AVX512F", "AVX512CD", "AVX512BW", "AVX512DQ",
293: (22)                 "AVX512VBMI", "AVX512VNNI", "AVX512VBMI2",
294: (8)             AVX512_SPR = ["AVX512F", "AVX512CD", "AVX512BW", "AVX512DQ",
295: (22)                 "AVX512VL", "AVX512IFMA", "AVX512VBMI", "AVX512VNNI",
296: (22)                 "AVX512VBMI2", "AVX512BITALG", "AVX512VPOPCNTDQ",
297: (22)                 "AVX512FP16"],
298: (4)         )
299: (4)         features_map = dict(
300: (8)             SSE3="PNI", SSE41="SSE4_1", SSE42="SSE4_2", FMA3="FMA",
301: (8)             AVX512VNNI="AVX512_VNNI", AVX512BITALG="AVX512_BITALG",
302: (8)             AVX5124FMAPS="AVX512_4FMAPS", AVX5124VNNIW="AVX512_4VNNIW",
303: (8)             AVX512VBMI2="AVX512_VBMI2",
304: (4)         )
305: (4)         def load_flags(self):
306: (8)             self.load_flags_cpuinfo("flags")
307: (0)             is_power = re.match("(powerpc|ppc)64", machine, re.IGNORECASE)
308: (0)             @pytest.mark.skipif(not is_linux or not is_power, reason="Only for Linux and
Power")
309: (0)             class Test_POWER_Features(AbstractTest):
310: (4)                 features = ["VSX", "VSX2", "VSX3", "VSX4"]
311: (4)                 features_map = dict(VSX2="ARCH_2_07", VSX3="ARCH_3_00", VSX4="ARCH_3_1")
312: (4)                 def load_flags(self):
313: (8)                     self.load_flags_auxv()
314: (0)                     is_zarch = re.match("(s390x)", machine, re.IGNORECASE)
315: (0)                     @pytest.mark.skipif(not is_linux or not is_zarch,
316: (20)                         reason="Only for Linux and IBM Z")
317: (0)                     class Test_ZARCH_Features(AbstractTest):

```

```

318: (4)             features = ["VX", "VXE", "VXE2"]
319: (4)             def load_flags(self):
320: (8)                 self.load_flags_auxv()
321: (0)             is_arm = re.match("(^arm|aarch64)", machine, re.IGNORECASE)
322: (0)             @pytest.mark.skipif(not is_linux or not is_arm, reason="Only for Linux and
ARM")
323: (0)             class Test_ARM_Features(AbstractTest):
324: (4)                 features = [
325: (8)                     "NEON", "ASIMD", "FPHP", "ASIMDHP", "ASIMDDP", "ASIMDFHM"
326: (4)                 ]
327: (4)                 features_groups = dict(
328: (8)                     NEON_FP16 = ["NEON", "HALF"],
329: (8)                     NEON_VFPV4 = ["NEON", "VFPV4"],
330: (4)                 )
331: (4)                 def load_flags(self):
332: (8)                     self.load_flags_cpuinfo("Features")
333: (8)                     arch = self.get_cpuinfo_item("CPU architecture")
334: (8)                     is_rootfs_v8 = int('0'+next(iter(arch))) > 7 if arch else 0
335: (8)                     if re.match("(^aarch64|AARCH64)", machine) or is_rootfs_v8:
336: (12)                         self.features_map = dict(
337: (16)                             NEON="ASIMD", HALF="ASIMD", VFPV4="ASIMD"
338: (12)                         )
339: (8)                     else:
340: (12)                         self.features_map = dict(
341: (16)                             ASIMD=("AES", "SHA1", "SHA2", "PMULL", "CRC32")
342: (12)                         )
-----
```

## File 85 - test\_custom\_dtypes.py:

```

1: (0)             import pytest
2: (0)             import numpy as np
3: (0)             from numpy.testing import assert_array_equal
4: (0)             from numpy.core._multiarray_umath import (
5: (4)                 _discover_array_parameters as discover_array_params, _get_sfloat_dtype)
6: (0)             SF = _get_sfloat_dtype()
7: (0)             class TestSFloat:
8: (4)                 def _get_array(self, scaling, aligned=True):
9: (8)                     if not aligned:
10: (12)                         a = np.empty(3*8 + 1, dtype=np.uint8)[1:]
11: (12)                         a = a.view(np.float64)
12: (12)                         a[:] = [1., 2., 3.]
13: (8)                     else:
14: (12)                         a = np.array([1., 2., 3.])
15: (8)                         a *= 1./scaling # the casting code also uses the reciprocal.
16: (8)                         return a.view(SF(scaling))
17: (4)                 def test_sffloat_rescaled(self):
18: (8)                     sf = SF(1.)
19: (8)                     sf2 = sf.scaled_by(2.)
20: (8)                     assert sf2.get_scaling() == 2.
21: (8)                     sf6 = sf2.scaled_by(3.)
22: (8)                     assert sf6.get_scaling() == 6.
23: (4)                 def test_class_discovery(self):
24: (8)                     dt, _ = discover_array_params([1., 2., 3.], dtype=SF)
25: (8)                     assert dt == SF(1.)
26: (4)                     @pytest.mark.parametrize("scaling", [1., -1., 2.])
27: (4)                     def test_scaled_float_from_floats(self, scaling):
28: (8)                         a = np.array([1., 2., 3.], dtype=SF(scaling))
29: (8)                         assert a.dtype.get_scaling() == scaling
30: (8)                         assert_array_equal(scaling * a.view(np.float64), [1., 2., 3.])
31: (4)                     def test_repr(self):
32: (8)                         assert repr(SF(scaling=1.)) == "_ScaledFloatTestDType(scaling=1.0)"
33: (4)                     def test_dtype_name(self):
34: (8)                         assert SF(1.).name == "_ScaledFloatTestDType64"
35: (4)                     @pytest.mark.parametrize("scaling", [1., -1., 2.])
36: (4)                     def test_sffloat_from_float(self, scaling):
37: (8)                         a = np.array([1., 2., 3.]).astype(dtype=SF(scaling))
38: (8)                         assert a.dtype.get_scaling() == scaling
-----
```

```

39: (8)             assert_array_equal(scaling * a.view(np.float64), [1., 2., 3.])
40: (4)             @pytest.mark.parametrize("aligned", [True, False])
41: (4)             @pytest.mark.parametrize("scaling", [1., -1., 2.])
42: (4)             def test_sfloating_getitem(self, aligned, scaling):
43: (8)                 a = self._get_array(1., aligned)
44: (8)                 assert a.tolist() == [1., 2., 3.]
45: (4)             @pytest.mark.parametrize("aligned", [True, False])
46: (4)             def test_sfloating_casts(self, aligned):
47: (8)                 a = self._get_array(1., aligned)
48: (8)                 assert np.can_cast(a, SF(-1.), casting="equiv")
49: (8)                 assert not np.can_cast(a, SF(-1.), casting="no")
50: (8)                 na = a.astype(SF(-1.))
51: (8)                 assert_array_equal(-1 * na.view(np.float64), a.view(np.float64))
52: (8)                 assert np.can_cast(a, SF(2.), casting="same_kind")
53: (8)                 assert not np.can_cast(a, SF(2.), casting="safe")
54: (8)                 a2 = a.astype(SF(2.))
55: (8)                 assert_array_equal(2 * a2.view(np.float64), a.view(np.float64))
56: (4)             @pytest.mark.parametrize("aligned", [True, False])
57: (4)             def test_sfloating_cast_internal_errors(self, aligned):
58: (8)                 a = self._get_array(2e300, aligned)
59: (8)                 with pytest.raises(TypeError,
60: (16)                         match="error raised inside the core-loop: non-finite
factor!"):
61: (12)                     a.astype(SF(2e-300))
62: (4)             def test_sfloating_promotion(self):
63: (8)                 assert np.result_type(SF(2.), SF(3.)) == SF(3.)
64: (8)                 assert np.result_type(SF(3.), SF(2.)) == SF(3.)
65: (8)                 assert np.result_type(SF(3.), np.float64) == SF(3.)
66: (8)                 assert np.result_type(np.float64, SF(0.5)) == SF(1.)
67: (8)                 with pytest.raises(TypeError):
68: (12)                     np.result_type(SF(1.), np.int64)
69: (4)             def test_basic_multiply(self):
70: (8)                 a = self._get_array(2.)
71: (8)                 b = self._get_array(4.)
72: (8)                 res = a * b
73: (8)                 assert res.dtype.get_scaling() == 8.
74: (8)                 expected_view = a.view(np.float64) * b.view(np.float64)
75: (8)                 assert_array_equal(res.view(np.float64), expected_view)
76: (4)             def test_possible_and_impossible_reduce(self):
77: (8)                 a = self._get_array(2.)
78: (8)                 res = np.add.reduce(a, initial=0.)
79: (8)                 assert res == a.astype(np.float64).sum()
80: (8)                 with pytest.raises(TypeError,
81: (16)                         match="the resolved dtypes are not compatible"):
82: (12)                     np.multiply.reduce(a)
83: (4)             def test_basic_ufunc_at(self):
84: (8)                 float_a = np.array([1., 2., 3.])
85: (8)                 b = self._get_array(2.)
86: (8)                 float_b = b.view(np.float64).copy()
87: (8)                 np.multiply.at(float_b, [1, 1, 1], float_a)
88: (8)                 np.multiply.at(b, [1, 1, 1], float_a)
89: (8)                 assert_array_equal(b.view(np.float64), float_b)
90: (4)             def test_basic_multiply_promotion(self):
91: (8)                 float_a = np.array([1., 2., 3.])
92: (8)                 b = self._get_array(2.)
93: (8)                 res1 = float_a * b
94: (8)                 res2 = b * float_a
95: (8)                 assert res1.dtype == res2.dtype == b.dtype
96: (8)                 expected_view = float_a * b.view(np.float64)
97: (8)                 assert_array_equal(res1.view(np.float64), expected_view)
98: (8)                 assert_array_equal(res2.view(np.float64), expected_view)
99: (8)                 np.multiply(b, float_a, out=res2)
100: (8)                 with pytest.raises(TypeError):
101: (12)                     np.multiply(b, float_a, out=np.arange(3))
102: (4)             def test_basic_addition(self):
103: (8)                 a = self._get_array(2.)
104: (8)                 b = self._get_array(4.)
105: (8)                 res = a + b
106: (8)                 assert res.dtype == np.result_type(a.dtype, b.dtype)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

107: (8)           expected_view = (a.astype(res.dtype).view(np.float64) +
108: (25)             b.astype(res.dtype).view(np.float64))
109: (8)           assert_array_equal(res.view(np.float64), expected_view)
110: (4)           def test_addition_cast_safety(self):
111: (8)             """The addition method is special for the scaled float, because it
112: (8)             includes the "cast" between different factors, thus cast-safety
113: (8)             is influenced by the implementation.
114: (8)
115: (8)             """
116: (8)             a = self._get_array(2.)
117: (8)             b = self._get_array(-2.)
118: (8)             c = self._get_array(3.)
119: (8)             np.add(a, b, casting="equiv")
120: (12)            with pytest.raises(TypeError):
121: (8)              np.add(a, b, casting="no")
122: (12)            with pytest.raises(TypeError):
123: (8)              np.add(a, c, casting="safe")
124: (12)            with pytest.raises(TypeError):
125: (8)              np.add(a, a, out=c, casting="safe")
126: (4)           @pytest.mark.parametrize("ufunc",
127: (12)             [np.logical_and, np.logical_or, np.logical_xor])
128: (4)           def test_logical_ufuncs_casts_to_bool(self, ufunc):
129: (8)             a = self._get_array(2.)
130: (8)             a[0] = 0. # make sure first element is considered False.
131: (8)             float_equiv = a.astype(float)
132: (8)             expected = ufunc(float_equiv, float_equiv)
133: (8)             res = ufunc(a, a)
134: (8)             assert_array_equal(res, expected)
135: (8)             expected = ufunc.reduce(float_equiv)
136: (8)             res = ufunc.reduce(a)
137: (8)             assert_array_equal(res, expected)
138: (12)            with pytest.raises(TypeError):
139: (4)              ufunc(a, a, out=np.empty(a.shape, dtype=int), casting="equiv")
140: (8)           def test_wrapped_and_wrapped_reductions(self):
141: (8)             a = self._get_array(2.)
142: (8)             float_equiv = a.astype(float)
143: (8)             expected = np.hypot(float_equiv, float_equiv)
144: (8)             res = np.hypot(a, a)
145: (8)             assert res.dtype == a.dtype
146: (8)             res_float = res.view(np.float64) * 2
147: (8)             assert_array_equal(res_float, expected)
148: (8)             res = np.hypot.reduce(a, keepdims=True)
149: (8)             assert res.dtype == a.dtype
150: (8)             expected = np.hypot.reduce(float_equiv, keepdims=True)
151: (4)           def test_astype_class(self):
152: (8)             arr = np.array([1., 2., 3.], dtype=object)
153: (8)             res = arr.astype(SF) # passing the class class
154: (8)             expected = arr.astype(SF(1.)) # above will have discovered 1. scaling
155: (8)             assert_array_equal(res.view(np.float64), expected.view(np.float64))
156: (4)           def test_creation_class(self):
157: (8)             arr1 = np.array([1., 2., 3.], dtype=SF)
158: (8)             assert arr1.dtype == SF(1.)
159: (8)             arr2 = np.array([1., 2., 3.], dtype=SF(1.))
160: (8)             assert_array_equal(arr1.view(np.float64), arr2.view(np.float64))
161: (0)           def test_type_pickle():
162: (4)             import pickle
163: (4)             np._ScaledFloatTestDType = SF
164: (4)             s = pickle.dumps(SF)
165: (4)             res = pickle.loads(s)
166: (4)             assert res is SF
167: (4)             del np._ScaledFloatTestDType
168: (0)           def test_is_numeric():
169: (4)             assert SF._is_numeric

```

-----  
File 86 - test\_cython.py:

```
1: (0)           import os
```

```
2: (0) import shutil
3: (0) import subprocess
4: (0) import sys
5: (0) import pytest
6: (0) import numpy as np
7: (0) from numpy.testing import IS_WASM
8: (0) try:
9: (4)     import cython
10: (4)     from Cython.Compiler.Version import version as cython_version
11: (0) except ImportError:
12: (4)     cython = None
13: (0) else:
14: (4)     from numpy._utils import _pep440
15: (4)     required_version = "0.29.30"
16: (4)     if _pep440.parse(cython_version) < _pep440.Version(required_version):
17: (8)         cython = None
18: (0)     pytestmark = pytest.mark.skipif(cython is None, reason="requires cython")
19: (0)     @pytest.fixture(scope='module')
20: (0)     def install_temp(tmpdir_factory):
21: (4)         if IS_WASM:
22: (8)             pytest.skip("No subprocess")
23: (4)         srcdir = os.path.join(os.path.dirname(__file__), 'examples', 'cython')
24: (4)         build_dir = tmpdir_factory.mktemp("cython_test") / "build"
25: (4)         os.makedirs(build_dir, exist_ok=True)
26: (4)         try:
27: (8)             subprocess.check_call(["meson", "--version"])
28: (4)         except FileNotFoundError:
29: (8)             pytest.skip("No usable 'meson' found")
30: (4)         if sys.platform == "win32":
31: (8)             subprocess.check_call(["meson", "setup",
32: (31)                 "--buildtype=release",
33: (31)                 "--venv", str(srcdir)],
34: (30)                 cwd=build_dir,
35: (30)             )
36: (4)         else:
37: (8)             subprocess.check_call(["meson", "setup", str(srcdir)],
38: (30)                 cwd=build_dir
39: (30)             )
40: (4)             subprocess.check_call(["meson", "compile", "-vv"], cwd=build_dir)
41: (4)             sys.path.append(str(build_dir))
42: (0)     def test_is_timedelta64_object(install_temp):
43: (4)         import checks
44: (4)         assert checks.is_td64(np.timedelta64(1234))
45: (4)         assert checks.is_td64(np.timedelta64(1234, "ns"))
46: (4)         assert checks.is_td64(np.timedelta64("NaT", "ns"))
47: (4)         assert not checks.is_td64(1)
48: (4)         assert not checks.is_td64(None)
49: (4)         assert not checks.is_td64("foo")
50: (4)         assert not checks.is_td64(np.datetime64("now", "s"))
51: (0)     def test_is_datetime64_object(install_temp):
52: (4)         import checks
53: (4)         assert checks.is_dt64(np.datetime64(1234, "ns"))
54: (4)         assert checks.is_dt64(np.datetime64("NaT", "ns"))
55: (4)         assert not checks.is_dt64(1)
56: (4)         assert not checks.is_dt64(None)
57: (4)         assert not checks.is_dt64("foo")
58: (4)         assert not checks.is_dt64(np.timedelta64(1234))
59: (0)     def test_get_datetime64_value(install_temp):
60: (4)         import checks
61: (4)         dt64 = np.datetime64("2016-01-01", "ns")
62: (4)         result = checks.get_dt64_value(dt64)
63: (4)         expected = dt64.view("i8")
64: (4)         assert result == expected
65: (0)     def test_get_timedelta64_value(install_temp):
66: (4)         import checks
67: (4)         td64 = np.timedelta64(12345, "h")
68: (4)         result = checks.get_td64_value(td64)
69: (4)         expected = td64.view("i8")
70: (4)         assert result == expected
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

71: (0)    def test_get_datetime64_unit(install_temp):
72: (4)        import checks
73: (4)        dt64 = np.datetime64("2016-01-01", "ns")
74: (4)        result = checks.get_dt64_unit(dt64)
75: (4)        expected = 10
76: (4)        assert result == expected
77: (4)        td64 = np.timedelta64(12345, "h")
78: (4)        result = checks.get_dt64_unit(td64)
79: (4)        expected = 5
80: (4)        assert result == expected
81: (0)    def test_abstract_scalars(install_temp):
82: (4)        import checks
83: (4)        assert checks.is_integer(1)
84: (4)        assert checks.is_integer(np.int8(1))
85: (4)        assert checks.is_integer(np.uint64(1))
86: (0)    def test_conv_intp(install_temp):
87: (4)        import checks
88: (4)        class myint:
89: (8)            def __int__(self):
90: (12)                return 3
91: (4)        assert checks.conv_intp(3.) == 3
92: (4)        assert checks.conv_intp(myint()) == 3

```

---

File 87 - test\_datetime.py:

```

1: (0)    import numpy
2: (0)    import numpy as np
3: (0)    import datetime
4: (0)    import pytest
5: (0)    from numpy.testing import (
6: (4)        IS_WASM,
7: (4)        assert_, assert_equal, assert_raises, assert_warns, suppress_warnings,
8: (4)        assert_raises_regex, assert_array_equal,
9: (4)    )
10: (0)   from numpy.compat import pickle
11: (0)   try:
12: (4)       from pytz import timezone as tz
13: (4)       _has_pytz = True
14: (0)   except ImportError:
15: (4)       _has_pytz = False
16: (0)   try:
17: (4)       RecursionError
18: (0)   except NameError:
19: (4)       RecursionError = RuntimeError # python < 3.5
20: (0)   class TestDateTime:
21: (4)       def test_datetime_dtype_creation(self):
22: (8)           for unit in ['Y', 'M', 'W', 'D',
23: (21)               'h', 'm', 's', 'ms', 'us',
24: (21)               '1s', # alias for us
25: (21)               'ns', 'ps', 'fs', 'as']:
26: (12)           dt1 = np.dtype('M8[750s]' % unit)
27: (12)           assert_(dt1 == np.dtype('datetime64[750s]' % unit))
28: (12)           dt2 = np.dtype('m8[%s]' % unit)
29: (12)           assert_(dt2 == np.dtype('timedelta64[%s]' % unit))
30: (8)           assert_equal(str(np.dtype("M8")), "datetime64")
31: (8)           assert_equal(np.dtype("=M8"), np.dtype("M8"))
32: (8)           assert_equal(np.dtype("=M8[s]"), np.dtype("M8[s]"))
33: (8)           assert_(np.dtype(">M8") == np.dtype("M8") or
34: (16)               np.dtype("<M8") == np.dtype("M8"))
35: (8)           assert_(np.dtype(">M8[D]") == np.dtype("M8[D]") or
36: (16)               np.dtype("<M8[D]") == np.dtype("M8[D]"))
37: (8)           assert_(np.dtype(">M8") != np.dtype("<M8"))
38: (8)           assert_equal(np.dtype("=m8"), np.dtype("m8"))
39: (8)           assert_equal(np.dtype("=m8[s]"), np.dtype("m8[s]"))
40: (8)           assert_(np.dtype(">m8") == np.dtype("m8") or
41: (16)               np.dtype("<m8") == np.dtype("m8"))
42: (8)           assert_(np.dtype(">m8[D]") == np.dtype("m8[D]") or

```

```

43: (16)           np.dtype("<m8[D]") == np.dtype("m8[D]"))
44: (8)           assert_(np.dtype(">m8") != np.dtype("<m8"))
45: (8)           assert_raises(TypeError, np.dtype, 'M8[badunit]')
46: (8)           assert_raises(TypeError, np.dtype, 'm8[badunit]')
47: (8)           assert_raises(TypeError, np.dtype, 'M8[YY]')
48: (8)           assert_raises(TypeError, np.dtype, 'm8[YY]')
49: (8)           assert_raises(TypeError, np.dtype, 'm4')
50: (8)           assert_raises(TypeError, np.dtype, 'M7')
51: (8)           assert_raises(TypeError, np.dtype, 'm7')
52: (8)           assert_raises(TypeError, np.dtype, 'M16')
53: (8)           assert_raises(TypeError, np.dtype, 'm16')
54: (8)           assert_raises(TypeError, np.dtype, 'M8[3000000000ps]')
55: (4) def test_datetime_casting_rules(self):
56: (8)     assert_(not np.can_cast('m8', 'M8', casting='same_kind'))
57: (8)     assert_(not np.can_cast('M8', 'm8', casting='same_kind'))
58: (8)     assert_(not np.can_cast('m8', 'M8', casting='safe'))
59: (8)     assert_(not np.can_cast('M8', 'm8', casting='safe'))
60: (8)     assert_(np.can_cast('i8', 'm8', casting='same_kind'))
61: (8)     assert_(np.can_cast('i8', 'm8', casting='safe'))
62: (8)     assert_(np.can_cast('i4', 'm8', casting='same_kind'))
63: (8)     assert_(np.can_cast('i4', 'm8', casting='safe'))
64: (8)     assert_(np.can_cast('u4', 'm8', casting='same_kind'))
65: (8)     assert_(np.can_cast('u4', 'm8', casting='safe'))
66: (8)     assert_(np.can_cast('u8', 'm8', casting='same_kind'))
67: (8)     assert_(not np.can_cast('u8', 'm8', casting='safe'))
68: (8)     assert_(not np.can_cast('f4', 'm8', casting='same_kind'))
69: (8)     assert_(not np.can_cast('f4', 'm8', casting='safe'))
70: (8)     assert_(not np.can_cast('i8', 'M8', casting='same_kind'))
71: (8)     assert_(not np.can_cast('i8', 'M8', casting='safe'))
72: (8)     assert_(not np.can_cast('b1', 'M8', casting='same_kind'))
73: (8)     assert_(not np.can_cast('b1', 'M8', casting='safe'))
74: (8)     assert_(np.can_cast('b1', 'm8', casting='same_kind'))
75: (8)     assert_(np.can_cast('b1', 'm8', casting='safe'))
76: (8)     assert_(np.can_cast('M8[M]', 'M8[D]', casting='safe'))
77: (8)     assert_(np.can_cast('M8[Y]', 'M8[D]', casting='safe'))
78: (8)     assert_(not np.can_cast('m8[M]', 'm8[D]', casting='safe'))
79: (8)     assert_(not np.can_cast('m8[Y]', 'm8[D]', casting='safe'))
80: (8)     assert_(np.can_cast('M8[M]', 'M8[D]', casting='same_kind'))
81: (8)     assert_(np.can_cast('M8[Y]', 'M8[D]', casting='same_kind'))
82: (8)     assert_(not np.can_cast('m8[M]', 'm8[D]', casting='same_kind'))
83: (8)     assert_(not np.can_cast('m8[Y]', 'm8[D]', casting='same_kind'))
84: (8)     assert_(np.can_cast('M8[D]', 'M8[h]', casting='same_kind'))
85: (8)     assert_(np.can_cast('m8[D]', 'm8[h]', casting='same_kind'))
86: (8)     assert_(np.can_cast('m8[h]', 'm8[D]', casting='same_kind'))
87: (8)     assert_(not np.can_cast('M8[7h]', 'M8[3h]', casting='safe'))
88: (8)     assert_(not np.can_cast('M8[3h]', 'M8[6h]', casting='safe'))
89: (8)     assert_(np.can_cast('M8[7h]', 'M8[3h]', casting='same_kind'))
90: (8)     assert_(np.can_cast('M8[6h]', 'M8[3h]', casting='safe'))
91: (8)     assert_(np.can_cast('m8', 'm8[h]', casting='same_kind'))
92: (8)     assert_(np.can_cast('m8', 'm8[h]', casting='safe'))
93: (8)     assert_(np.can_cast('M8', 'M8[h]', casting='same_kind'))
94: (8)     assert_(np.can_cast('M8', 'M8[h]', casting='safe'))
95: (8)     assert_(not np.can_cast('m8[h]', 'm8', casting='same_kind'))
96: (8)     assert_(not np.can_cast('m8[h]', 'm8', casting='safe'))
97: (8)     assert_(not np.can_cast('M8[h]', 'M8', casting='same_kind'))
98: (8)     assert_(not np.can_cast('M8[h]', 'M8', casting='safe'))
99: (4) def test_datetime_prefix_conversions(self):
100: (8)    smaller_units = ['M8[7000ms]',
101: (25)          'M8[2000us]',
102: (25)          'M8[1000ns]',
103: (25)          'M8[5000ns]',
104: (25)          'M8[2000ps]',
105: (25)          'M8[9000fs]',
106: (25)          'M8[1000as]',
107: (25)          'M8[2000000ps]',
108: (25)          'M8[1000000as]',
109: (25)          'M8[2000000000ps]',
110: (25)          'M8[1000000000as]']
111: (8)    larger_units = ['M8[7s]',
```

```

112: (24)                                'M8[2ms]',
113: (24)                                'M8[us]',
114: (24)                                'M8[5us]',
115: (24)                                'M8[2ns]',
116: (24)                                'M8[9ps]',
117: (24)                                'M8[1fs]',
118: (24)                                'M8[2us]',
119: (24)                                'M8[1ps]',
120: (24)                                'M8[2ms]',
121: (24)                                'M8[1ns]']
122: (8)                                 for larger_unit, smaller_unit in zip(larger_units, smaller_units):
123: (12)                               assert np.can_cast(larger_unit, smaller_unit, casting='safe')
124: (12)                               assert np.can_cast(smaller_unit, larger_unit, casting='safe')
125: (4) @pytest.mark.parametrize("unit", [
126: (8)   "s", "ms", "us", "ns", "ps", "fs", "as"])
127: (4) def test_prohibit_negative_datetime(self, unit):
128: (8)   with assert_raises(TypeError):
129: (12)     np.array([1], dtype=f"M8[-1{unit}]")
130: (4) def test_compare_generic_nat(self):
131: (8)   assert_(np.datetime64('NaT') !=
132: (16)     np.datetime64('2000') + np.timedelta64('NaT'))
133: (8)   assert_(np.datetime64('NaT') != np.datetime64('NaT', 'us'))
134: (8)   assert_(np.datetime64('NaT', 'us') != np.datetime64('NaT'))
135: (4) @pytest.mark.parametrize("size", [
136: (8)   3, 21, 217, 1000])
137: (4) def test_datetime_nat_argsort_stability(self, size):
138: (8)   expected = np.arange(size)
139: (8)   arr = np.tile(np.datetime64('NaT'), size)
140: (8)   assert_equal(np.argsort(arr, kind='mergesort'), expected)
141: (4) @pytest.mark.parametrize("size", [
142: (8)   3, 21, 217, 1000])
143: (4) def test_timedelta_nat_argsort_stability(self, size):
144: (8)   expected = np.arange(size)
145: (8)   arr = np.tile(np.timedelta64('NaT'), size)
146: (8)   assert_equal(np.argsort(arr, kind='mergesort'), expected)
147: (4) @pytest.mark.parametrize("arr, expected", [
148: (8)   ([['NaT', 1, 2, 3],
149: (9)     [1, 2, 3, 'NaT']],
150: (8)     [['NaT', 9, 'NaT', -707],
151: (9)       [-707, 9, 'NaT', 'NaT']],
152: (8)     [[1, -2, 3, 'NaT'],
153: (9)       [-2, 1, 3, 'NaT']],
154: (8)     ([[51, -220, 'NaT'],
155: (10)       [-17, 'NaT', -90]],
156: (9)       [[-220, 51, 'NaT'],
157: (10)         [-90, -17, 'NaT']])),
158: (8)   ])
159: (4) @pytest.mark.parametrize("dtype", [
160: (8)   'M8[ns]', 'M8[us]',
161: (8)   'm8[ns]', 'm8[us]')
162: (4) def test_datetime_timedelta_sort_nat(self, arr, expected, dtype):
163: (8)   arr = np.array(arr, dtype=dtype)
164: (8)   expected = np.array(expected, dtype=dtype)
165: (8)   arr.sort()
166: (8)   assert_equal(arr, expected)
167: (4) def test_datetime_scalar_construction(self):
168: (8)   assert_equal(np.datetime64('1950-03-12', 'D'),
169: (21)     np.datetime64('1950-03-12'))
170: (8)   assert_equal(np.datetime64('1950-03-12T13', 's'),
171: (21)     np.datetime64('1950-03-12T13', 'm'))
172: (8)   assert_equal(np.datetime64(), np.datetime64('NaT'))
173: (8)   assert_equal(str(np.datetime64('NaT')), 'NaT')
174: (8)   assert_equal(repr(np.datetime64('NaT')),
175: (21)     "numpy.datetime64('NaT')")
176: (8)   assert_equal(str(np.datetime64('2011-02')), '2011-02')
177: (8)   assert_equal(repr(np.datetime64('2011-02')),
178: (21)     "numpy.datetime64('2011-02')")
179: (8)   assert_equal(np.datetime64(None), np.datetime64('NaT'))
180: (8)   assert_equal(np.datetime64().dtype, np.dtype('M8'))

```

```

181: (8) assert_equal(np.datetime64('NaT').dtype, np.dtype('M8'))
182: (8) assert_raises(ValueError, np.datetime64, 17)
183: (8) a = np.datetime64('2000-03-18T16', 'h')
184: (8) b = np.array('2000-03-18T16', dtype='M8[h]')
185: (8) assert_equal(a.dtype, np.dtype('M8[h]'))
186: (8) assert_equal(b.dtype, np.dtype('M8[h]'))
187: (8) assert_equal(np.datetime64(a), a)
188: (8) assert_equal(np.datetime64(a).dtype, np.dtype('M8[h]'))
189: (8) assert_equal(np.datetime64(b), a)
190: (8) assert_equal(np.datetime64(b).dtype, np.dtype('M8[h]'))
191: (8) assert_equal(np.datetime64(a, 's'), a)
192: (8) assert_equal(np.datetime64(a, 's').dtype, np.dtype('M8[s]'))
193: (8) assert_equal(np.datetime64(b, 's'), a)
194: (8) assert_equal(np.datetime64(b, 's').dtype, np.dtype('M8[s]'))
195: (8) assert_equal(np.datetime64('1945-03-25'),
196: (21)           np.datetime64(datetime.date(1945, 3, 25)))
197: (8) assert_equal(np.datetime64('2045-03-25', 'D'),
198: (21)           np.datetime64(datetime.date(2045, 3, 25), 'D'))
199: (8) assert_equal(np.datetime64('1980-01-25T14:36:22.5'),
200: (21)           np.datetime64(datetime.datetime(1980, 1, 25,
201: (48)             14, 36, 22, 500000)))
202: (8) assert_equal(np.datetime64('1920-03-13', 'h'),
203: (21)           np.datetime64('1920-03-13T00'))
204: (8) assert_equal(np.datetime64('1920-03', 'm'),
205: (21)           np.datetime64('1920-03-01T00:00'))
206: (8) assert_equal(np.datetime64('1920', 's'),
207: (21)           np.datetime64('1920-01-01T00:00:00'))
208: (8) assert_equal(np.datetime64(datetime.date(2045, 3, 25), 'ms'),
209: (21)           np.datetime64('2045-03-25T00:00:00.000'))
210: (8) assert_equal(np.datetime64('1920-03-13T18', 'D'),
211: (21)           np.datetime64('1920-03-13'))
212: (8) assert_equal(np.datetime64('1920-03-13T18:33:12', 'M'),
213: (21)           np.datetime64('1920-03'))
214: (8) assert_equal(np.datetime64('1920-03-13T18:33:12.5', 'Y'),
215: (21)           np.datetime64('1920'))
216: (4) def test_datetime_scalar_construction_timezone(self):
217: (8)     with assert_warnings(DeprecationWarning):
218: (12)         assert_equal(np.datetime64('2000-01-01T00Z'),
219: (25)               np.datetime64('2000-01-01T00'))
220: (8)     with assert_warnings(DeprecationWarning):
221: (12)         assert_equal(np.datetime64('2000-01-01T00-08'),
222: (25)               np.datetime64('2000-01-01T08'))
223: (4) def test_datetime_array_find_type(self):
224: (8)     dt = np.datetime64('1970-01-01', 'M')
225: (8)     arr = np.array([dt])
226: (8)     assert_equal(arr.dtype, np.dtype('M8[M]'))
227: (8)     dt = datetime.date(1970, 1, 1)
228: (8)     arr = np.array([dt])
229: (8)     assert_equal(arr.dtype, np.dtype('O'))
230: (8)     dt = datetime.datetime(1970, 1, 1, 12, 30, 40)
231: (8)     arr = np.array([dt])
232: (8)     assert_equal(arr.dtype, np.dtype('O'))
233: (8)     b = np.bool_(True)
234: (8)     dm = np.datetime64('1970-01-01', 'M')
235: (8)     d = datetime.date(1970, 1, 1)
236: (8)     dt = datetime.datetime(1970, 1, 1, 12, 30, 40)
237: (8)     arr = np.array([b, dm])
238: (8)     assert_equal(arr.dtype, np.dtype('O'))
239: (8)     arr = np.array([b, d])
240: (8)     assert_equal(arr.dtype, np.dtype('O'))
241: (8)     arr = np.array([b, dt])
242: (8)     assert_equal(arr.dtype, np.dtype('O'))
243: (8)     arr = np.array([d, d]).astype('datetime64')
244: (8)     assert_equal(arr.dtype, np.dtype('M8[D]'))
245: (8)     arr = np.array([dt, dt]).astype('datetime64')
246: (8)     assert_equal(arr.dtype, np.dtype('M8[us]'))
247: (4)     @pytest.mark.parametrize("unit", [
248: (4)         ("Y"), ("M"), ("W"), ("D"), ("h"), ("m"),
249: (4)         ("s"), ("ms"), ("us"), ("ns"), ("ps"),

```

```

250: (4)
251: (4)
252: (8)
253: (12)
254: (25)
255: (8)
256: (12)
257: (25)
258: (4)
259: (8)
260: (21)
261: (8)
262: (21)
263: (8)
264: (8)
265: (8)
266: (8)
267: (21)
268: (8)
269: (8)
270: (21)
271: (8)
272: (21)
273: (8)
274: (8)
275: (8)
276: (8)
277: (8)
278: (8)
279: (8)
280: (8)
281: (8)
282: (8)
283: (8)
284: (8)
285: (8)
286: (8)
287: (21)
288: (8)
289: (21)
290: (8)
291: (21)
292: (44)
293: (8)
294: (21)
295: (44)
296: (8)
297: (21)
298: (8)
299: (21)
300: (8)
301: (21)
302: (8)
303: (21)
304: (8)
305: (8)
306: (8)
307: (8)
308: (8)
309: (8)
310: (8)
311: (8)
312: (8)
313: (8)
314: (8)
315: (8)
316: (8)
317: (8)
318: (8)

        ("fs"), ("as"), ("generic") ])
def test_timedelta_np_int_construction(self, unit):
    if unit != "generic":
        assert_equal(np.timedelta64(np.int64(123), unit),
                    np.timedelta64(123, unit))
    else:
        assert_equal(np.timedelta64(np.int64(123)),
                    np.timedelta64(123))
def test_timedelta_scalar_construction(self):
    assert_equal(np.timedelta64(7, 'D'),
                np.timedelta64(1, 'W'))
    assert_equal(np.timedelta64(120, 's'),
                np.timedelta64(2, 'm'))
    assert_equal(np.timedelta64(0), np.timedelta64(0))
    assert_equal(np.timedelta64(None), np.timedelta64('NaT'))
    assert_equal(str(np.timedelta64('NaT')), 'NaT')
    assert_equal(repr(np.timedelta64('NaT')),
                "numpy.timedelta64('NaT')")
    assert_equal(str(np.timedelta64(3, 's')), '3 seconds')
    assert_equal(repr(np.timedelta64(-3, 's')),
                "numpy.timedelta64(-3,'s')")
    assert_equal(repr(np.timedelta64(12)),
                "numpy.timedelta64(12)")
    assert_equal(np.timedelta64(12).dtype, np.dtype('m8'))
    a = np.timedelta64(2, 'h')
    b = np.array(2, dtype='m8[h]')
    assert_equal(a.dtype, np.dtype('m8[h]'))
    assert_equal(b.dtype, np.dtype('m8[h]'))
    assert_equal(np.timedelta64(a), a)
    assert_equal(np.timedelta64(a).dtype, np.dtype('m8[h]'))
    assert_equal(np.timedelta64(b), a)
    assert_equal(np.timedelta64(b).dtype, np.dtype('m8[h]'))
    assert_equal(np.timedelta64(a, 's'), a)
    assert_equal(np.timedelta64(a, 's').dtype, np.dtype('m8[s]'))
    assert_equal(np.timedelta64(b, 's'), a)
    assert_equal(np.timedelta64(b, 's').dtype, np.dtype('m8[s]'))
    assert_equal(np.timedelta64(5, 'D'),
                np.timedelta64(datetime.timedelta(days=5)))
    assert_equal(np.timedelta64(102347621, 's'),
                np.timedelta64(datetime.timedelta(seconds=102347621)))
    assert_equal(np.timedelta64(-1023476000, 'us'),
                np.timedelta64(datetime.timedelta(
                    microseconds=-1023476000)))
    assert_equal(np.timedelta64(1023476000, 'us'),
                np.timedelta64(datetime.timedelta(
                    microseconds=1023476000)))
    assert_equal(np.timedelta64(1023476, 'ms'),
                np.timedelta64(datetime.timedelta(milliseconds=1023476)))
    assert_equal(np.timedelta64(10, 'm'),
                np.timedelta64(datetime.timedelta(minutes=10)))
    assert_equal(np.timedelta64(281, 'h'),
                np.timedelta64(datetime.timedelta(hours=281)))
    assert_equal(np.timedelta64(28, 'W'),
                np.timedelta64(datetime.timedelta(weeks=28)))
    a = np.timedelta64(3, 's')
    assert_raises(TypeError, np.timedelta64, a, 'M')
    assert_raises(TypeError, np.timedelta64, a, 'Y')
    a = np.timedelta64(6, 'M')
    assert_raises(TypeError, np.timedelta64, a, 'D')
    assert_raises(TypeError, np.timedelta64, a, 'h')
    a = np.timedelta64(1, 'Y')
    assert_raises(TypeError, np.timedelta64, a, 'D')
    assert_raises(TypeError, np.timedelta64, a, 'm')
    a = datetime.timedelta(seconds=3)
    assert_raises(TypeError, np.timedelta64, a, 'M')
    assert_raises(TypeError, np.timedelta64, a, 'Y')
    a = datetime.timedelta(weeks=3)
    assert_raises(TypeError, np.timedelta64, a, 'M')
    assert_raises(TypeError, np.timedelta64, a, 'Y')

```

```

319: (8)           a = datetime.timedelta()
320: (8)           assert_raises(TypeError, np.timedelta64, a, 'M')
321: (8)           assert_raises(TypeError, np.timedelta64, a, 'Y')
322: (4)           def test_timedelta_object_array_conversion(self):
323: (8)             inputs = [datetime.timedelta(28),
324: (18)               datetime.timedelta(30),
325: (18)               datetime.timedelta(31)]
326: (8)             expected = np.array([28, 30, 31], dtype='timedelta64[D]')
327: (8)             actual = np.array(inputs, dtype='timedelta64[D]')
328: (8)             assert_equal(expected, actual)
329: (4)           def test_timedelta_0_dim_object_array_conversion(self):
330: (8)             test = np.array(datetime.timedelta(seconds=20))
331: (8)             actual = test.astype(np.timedelta64)
332: (8)             expected = np.array(datetime.timedelta(seconds=20),
333: (28)               np.timedelta64)
334: (8)             assert_equal(actual, expected)
335: (4)           def test_timedelta_nat_format(self):
336: (8)             assert_equal('NaT', '{0}'.format(np.timedelta64('nat')))
337: (4)           def test_timedelta_scalar_construction_units(self):
338: (8)             assert_equal(np.datetime64('2010').dtype,
339: (21)               np.dtype('M8[Y]'))
340: (8)             assert_equal(np.datetime64('2010-03').dtype,
341: (21)               np.dtype('M8[M]'))
342: (8)             assert_equal(np.datetime64('2010-03-12').dtype,
343: (21)               np.dtype('M8[D]'))
344: (8)             assert_equal(np.datetime64('2010-03-12T17').dtype,
345: (21)               np.dtype('M8[h]'))
346: (8)             assert_equal(np.datetime64('2010-03-12T17:15').dtype,
347: (21)               np.dtype('M8[m]'))
348: (8)             assert_equal(np.datetime64('2010-03-12T17:15:08').dtype,
349: (21)               np.dtype('M8[s]'))
350: (8)             assert_equal(np.datetime64('2010-03-12T17:15:08.1').dtype,
351: (21)               np.dtype('M8[ms]'))
352: (8)             assert_equal(np.datetime64('2010-03-12T17:15:08.12').dtype,
353: (21)               np.dtype('M8[ms]'))
354: (8)             assert_equal(np.datetime64('2010-03-12T17:15:08.123').dtype,
355: (21)               np.dtype('M8[ms]'))
356: (8)             assert_equal(np.datetime64('2010-03-12T17:15:08.1234').dtype,
357: (21)               np.dtype('M8[us]'))
358: (8)             assert_equal(np.datetime64('2010-03-12T17:15:08.12345').dtype,
359: (21)               np.dtype('M8[us]'))
360: (8)             assert_equal(np.datetime64('2010-03-12T17:15:08.123456').dtype,
361: (21)               np.dtype('M8[us]'))
362: (8)             assert_equal(np.datetime64('1970-01-01T00:00:02.1234567').dtype,
363: (21)               np.dtype('M8[ns]'))
364: (8)             assert_equal(np.datetime64('1970-01-01T00:00:02.12345678').dtype,
365: (21)               np.dtype('M8[ns]'))
366: (8)             assert_equal(np.datetime64('1970-01-01T00:00:02.123456789').dtype,
367: (21)               np.dtype('M8[ns]'))
368: (8)             assert_equal(np.datetime64('1970-01-01T00:00:02.1234567890').dtype,
369: (21)               np.dtype('M8[ps]'))
370: (8)             assert_equal(np.datetime64('1970-01-01T00:00:02.12345678901').dtype,
371: (21)               np.dtype('M8[ps]'))
372: (8)             assert_equal(np.datetime64('1970-01-01T00:00:02.123456789012').dtype,
373: (21)               np.dtype('M8[ps]'))
374: (8)             assert_equal(np.datetime64(
375: (21)               '1970-01-01T00:00:02.1234567890123').dtype,
376: (21)               np.dtype('M8[fs]'))
377: (8)             assert_equal(np.datetime64(
378: (21)               '1970-01-01T00:00:02.12345678901234').dtype,
379: (21)               np.dtype('M8[fs]'))
380: (8)             assert_equal(np.datetime64(
381: (21)               '1970-01-01T00:00:02.123456789012345').dtype,
382: (21)               np.dtype('M8[fs]'))
383: (8)             assert_equal(np.datetime64(
384: (20)               '1970-01-01T00:00:02.1234567890123456').dtype,
385: (21)               np.dtype('M8[as]'))
386: (8)             assert_equal(np.datetime64(
387: (20)               '1970-01-01T00:00:02.12345678901234567').dtype,

```

```

388: (21)           np.dtype('M8[as]'))
389: (8)            assert_equal(np.datetime64(
390: (20)              '1970-01-01T00:00:02.123456789012345678').dtype,
391: (21)              np.dtype('M8[as]'))
392: (8)            assert_equal(np.datetime64(datetime.date(2010, 4, 16)).dtype,
393: (21)              np.dtype('M8[D]'))
394: (8)            assert_equal(np.datetime64(
395: (24)              datetime.datetime(2010, 4, 16, 13, 45, 18)).dtype,
396: (21)              np.dtype('M8[us]'))
397: (8)            assert_equal(np.datetime64('today').dtype,
398: (21)              np.dtype('M8[D]'))
399: (8)            assert_equal(np.datetime64('now').dtype,
400: (21)              np.dtype('M8[s]'))
401: (4) def test_datetime_nat_casting(self):
402: (8)     a = np.array('NaT', dtype='M8[D]')
403: (8)     b = np.datetime64('NaT', '[D]')
404: (8)     assert_equal(a.astype('M8[s]'), np.array('NaT', dtype='M8[s]'))
405: (8)     assert_equal(a.astype('M8[ms]'), np.array('NaT', dtype='M8[ms]'))
406: (8)     assert_equal(a.astype('M8[M]'), np.array('NaT', dtype='M8[M]'))
407: (8)     assert_equal(a.astype('M8[Y]'), np.array('NaT', dtype='M8[Y]'))
408: (8)     assert_equal(a.astype('M8[W]'), np.array('NaT', dtype='M8[W]'))
409: (8)     assert_equal(np.datetime64(b, '[s)'), np.datetime64('NaT', '[s']))
410: (8)     assert_equal(np.datetime64(b, '[ms)'), np.datetime64('NaT', '[ms']))
411: (8)     assert_equal(np.datetime64(b, '[M)'), np.datetime64('NaT', '[M]'))
412: (8)     assert_equal(np.datetime64(b, '[Y)'), np.datetime64('NaT', '[Y]'))
413: (8)     assert_equal(np.datetime64(b, '[W)'), np.datetime64('NaT', '[W]'))
414: (8)     assert_equal(np.datetime64(a, '[s)'), np.datetime64('NaT', '[s]'))
415: (8)     assert_equal(np.datetime64(a, '[ms)'), np.datetime64('NaT', '[ms]'))
416: (8)     assert_equal(np.datetime64(a, '[M)'), np.datetime64('NaT', '[M]'))
417: (8)     assert_equal(np.datetime64(a, '[Y)'), np.datetime64('NaT', '[Y]'))
418: (8)     assert_equal(np.datetime64(a, '[W)'), np.datetime64('NaT', '[W]'))
419: (8)     nan = np.array([np.nan] * 8)
420: (8)     fnan = nan.astype('f')
421: (8)     lnan = nan.astype('g')
422: (8)     cnan = nan.astype('D')
423: (8)     cfnan = nan.astype('F')
424: (8)     clnan = nan.astype('G')
425: (8)     nat = np.array([np.datetime64('NaT')] * 8)
426: (8)     assert_equal(nan.astype('M8[ns]'), nat)
427: (8)     assert_equal(fnan.astype('M8[ns]'), nat)
428: (8)     assert_equal(lnan.astype('M8[ns]'), nat)
429: (8)     assert_equal(cnan.astype('M8[ns]'), nat)
430: (8)     assert_equal(cfnan.astype('M8[ns]'), nat)
431: (8)     assert_equal(clnan.astype('M8[ns]'), nat)
432: (8)     nat = np.array([np.timedelta64('NaT')] * 8)
433: (8)     assert_equal(nan.astype('timedelta64[ns]'), nat)
434: (8)     assert_equal(fnan.astype('timedelta64[ns]'), nat)
435: (8)     assert_equal(lnan.astype('timedelta64[ns]'), nat)
436: (8)     assert_equal(cnan.astype('timedelta64[ns]'), nat)
437: (8)     assert_equal(cfnan.astype('timedelta64[ns]'), nat)
438: (8)     assert_equal(clnan.astype('timedelta64[ns]'), nat)
439: (4) def test_days_creation(self):
440: (8)     assert_equal(np.array('1599', dtype='M8[D]').astype('i8'),
441: (16)             (1600-1970)*365 - (1972-1600)/4 + 3 - 365)
442: (8)     assert_equal(np.array('1600', dtype='M8[D]').astype('i8'),
443: (16)             (1600-1970)*365 - (1972-1600)/4 + 3)
444: (8)     assert_equal(np.array('1601', dtype='M8[D]').astype('i8'),
445: (16)             (1600-1970)*365 - (1972-1600)/4 + 3 + 366)
446: (8)     assert_equal(np.array('1900', dtype='M8[D]').astype('i8'),
447: (16)             (1900-1970)*365 - (1970-1900)//4)
448: (8)     assert_equal(np.array('1901', dtype='M8[D]').astype('i8'),
449: (16)             (1900-1970)*365 - (1970-1900)//4 + 365)
450: (8)     assert_equal(np.array('1967', dtype='M8[D]').astype('i8'), -3*365 - 1)
451: (8)     assert_equal(np.array('1968', dtype='M8[D]').astype('i8'), -2*365 - 1)
452: (8)     assert_equal(np.array('1969', dtype='M8[D]').astype('i8'), -1*365)
453: (8)     assert_equal(np.array('1970', dtype='M8[D]').astype('i8'), 0*365)
454: (8)     assert_equal(np.array('1971', dtype='M8[D]').astype('i8'), 1*365)
455: (8)     assert_equal(np.array('1972', dtype='M8[D]').astype('i8'), 2*365)
456: (8)     assert_equal(np.array('1973', dtype='M8[D]').astype('i8'), 3*365 + 1)

```

```

457: (8) assert_equal(np.array('1974', dtype='M8[D]').astype('i8'), 4*365 + 1)
458: (8) assert_equal(np.array('2000', dtype='M8[D]').astype('i8'),
459: (17) (2000 - 1970)*365 + (2000 - 1972)//4)
460: (8) assert_equal(np.array('2001', dtype='M8[D]').astype('i8'),
461: (17) (2000 - 1970)*365 + (2000 - 1972)//4 + 366)
462: (8) assert_equal(np.array('2400', dtype='M8[D]').astype('i8'),
463: (17) (2400 - 1970)*365 + (2400 - 1972)//4 - 3)
464: (8) assert_equal(np.array('2401', dtype='M8[D]').astype('i8'),
465: (17) (2400 - 1970)*365 + (2400 - 1972)//4 - 3 + 366)
466: (8) assert_equal(np.array('1600-02-29', dtype='M8[D]').astype('i8'),
467: (16) (1600-1970)*365 - (1972-1600)//4 + 3 + 31 + 28)
468: (8) assert_equal(np.array('1600-03-01', dtype='M8[D]').astype('i8'),
469: (16) (1600-1970)*365 - (1972-1600)//4 + 3 + 31 + 29)
470: (8) assert_equal(np.array('2000-02-29', dtype='M8[D]').astype('i8'),
471: (17) (2000 - 1970)*365 + (2000 - 1972)//4 + 31 + 28)
472: (8) assert_equal(np.array('2000-03-01', dtype='M8[D]').astype('i8'),
473: (17) (2000 - 1970)*365 + (2000 - 1972)//4 + 31 + 29)
474: (8) assert_equal(np.array('2001-03-22', dtype='M8[D]').astype('i8'),
475: (17) (2000 - 1970)*365 + (2000 - 1972)//4 + 366 + 31 + 28 + 21)
476: (4) def test_days_to_pydate(self):
477: (8) assert_equal(np.array('1599', dtype='M8[D]').astype('O'),
478: (20) datetime.date(1599, 1, 1))
479: (8) assert_equal(np.array('1600', dtype='M8[D]').astype('O'),
480: (20) datetime.date(1600, 1, 1))
481: (8) assert_equal(np.array('1601', dtype='M8[D]').astype('O'),
482: (20) datetime.date(1601, 1, 1))
483: (8) assert_equal(np.array('1900', dtype='M8[D]').astype('O'),
484: (20) datetime.date(1900, 1, 1))
485: (8) assert_equal(np.array('1901', dtype='M8[D]').astype('O'),
486: (20) datetime.date(1901, 1, 1))
487: (8) assert_equal(np.array('2000', dtype='M8[D]').astype('O'),
488: (20) datetime.date(2000, 1, 1))
489: (8) assert_equal(np.array('2001', dtype='M8[D]').astype('O'),
490: (20) datetime.date(2001, 1, 1))
491: (8) assert_equal(np.array('1600-02-29', dtype='M8[D]').astype('O'),
492: (20) datetime.date(1600, 2, 29))
493: (8) assert_equal(np.array('1600-03-01', dtype='M8[D]').astype('O'),
494: (20) datetime.date(1600, 3, 1))
495: (8) assert_equal(np.array('2001-03-22', dtype='M8[D]').astype('O'),
496: (20) datetime.date(2001, 3, 22)))
497: (4) def test_dtype_comparison(self):
498: (8) assert_(not (np.dtype('M8[us]') == np.dtype('M8[ms]')))
499: (8) assert_(np.dtype('M8[us]') != np.dtype('M8[ms]'))
500: (8) assert_(np.dtype('M8[2D]') != np.dtype('M8[D]'))
501: (8) assert_(np.dtype('M8[D]') != np.dtype('M8[2D]'))
502: (4) def test_pydatetime_creation(self):
503: (8) a = np.array(['1960-03-12', datetime.date(1960, 3, 12)],
504: (8) assert_equal(a[0], a[1])
505: (8) a = np.array(['1999-12-31', datetime.date(1999, 12, 31)],
506: (8) assert_equal(a[0], a[1])
507: (8) a = np.array(['2000-01-01', datetime.date(2000, 1, 1)], dtype='M8[D]')
508: (8) assert_equal(a[0], a[1])
509: (8) a = np.array(['today', datetime.date.today()], dtype='M8[D]')
510: (8) assert_equal(a[0], a[1])
511: (8) assert_equal(np.array(datetime.date(1960, 3, 12), dtype='M8[s']),
512: (21) np.array(np.datetime64('1960-03-12T00:00:00')))
513: (4) def test_datetime_string_conversion(self):
514: (8) a = ['2011-03-16', '1920-01-01', '2013-05-19']
515: (8) str_a = np.array(a, dtype='S')
516: (8) uni_a = np.array(a, dtype='U')
517: (8) dt_a = np.array(a, dtype='M')
518: (8) assert_equal(dt_a, str_a.astype('M'))
519: (8) assert_equal(dt_a.dtype, str_a.astype('M').dtype)
520: (8) dt_b = np.empty_like(dt_a)
521: (8) dt_b[...] = str_a
522: (8) assert_equal(dt_a, dt_b)
523: (8) assert_equal(str_a, dt_a.astype('S0'))

```

```

524: (8)     str_b = np.empty_like(str_a)
525: (8)     str_b[...] = dt_a
526: (8)     assert_equal(str_a, str_b)
527: (8)     assert_equal(dt_a, uni_a.astype('M'))
528: (8)     assert_equal(dt_a.dtype, uni_a.astype('M').dtype)
529: (8)     dt_b = np.empty_like(dt_a)
530: (8)     dt_b[...] = uni_a
531: (8)     assert_equal(dt_a, dt_b)
532: (8)     assert_equal(uni_a, dt_a.astype('U'))
533: (8)     uni_b = np.empty_like(uni_a)
534: (8)     uni_b[...] = dt_a
535: (8)     assert_equal(uni_a, uni_b)
536: (8)     assert_equal(str_a, dt_a.astype((np.bytes_, 128)))
537: (8)     str_b = np.empty(str_a.shape, dtype=(np.bytes_, 128))
538: (8)     str_b[...] = dt_a
539: (8)     assert_equal(str_a, str_b)
540: (4) @pytest.mark.parametrize("time_dtype", ["m8[D]", "M8[Y]"])
541: (4) def test_time_byteswapping(self, time_dtype):
542: (8)     times = np.array(["2017", "NaT"], dtype=time_dtype)
543: (8)     times_swapped = times.astype(times.dtype.newbyteorder())
544: (8)     assert_array_equal(times, times_swapped)
545: (8)     unswapped = times_swapped.view(np.int64).newbyteorder()
546: (8)     assert_array_equal(unswapped, times.view(np.int64))
547: (4) @pytest.mark.parametrize(["time1", "time2"],
548: (12)         [("M8[s]", "M8[D]"), ("m8[s]", "m8[ns]")])
549: (4) def test_time_byteswapped_cast(self, time1, time2):
550: (8)     dtype1 = np.dtype(time1)
551: (8)     dtype2 = np.dtype(time2)
552: (8)     times = np.array(["2017", "NaT"], dtype=dtype1)
553: (8)     expected = times.astype(dtype2)
554: (8)     res = times.astype(dtype1.newbyteorder()).astype(dtype2)
555: (8)     assert_array_equal(res, expected)
556: (8)     res = times.astype(dtype2.newbyteorder())
557: (8)     assert_array_equal(res, expected)
558: (8)     res =
559: (8)         times.astype(dtype1.newbyteorder()).astype(dtype2.newbyteorder())
560: (8)         assert_array_equal(res, expected)
561: (4) @pytest.mark.parametrize("time_dtype", ["m8[D]", "M8[Y]"])
562: (4) @pytest.mark.parametrize("str_dtype", ["U", "S"])
563: (4) def test_datetime_conversions_byteorders(self, str_dtype, time_dtype):
564: (8)     times = np.array(["2017", "NaT"], dtype=time_dtype)
565: (8)     from_strings = np.array(["2017", "NaT"], dtype=str_dtype)
566: (8)     to_strings = times.astype(str_dtype) # assume this is correct
567: (8)     times_swapped = times.astype(times.dtype.newbyteorder())
568: (8)     res = times_swapped.astype(str_dtype)
569: (8)     assert_array_equal(res, to_strings)
570: (8)     res = times_swapped.astype(to_strings.dtype.newbyteorder())
571: (8)     assert_array_equal(res, to_strings)
572: (8)     res = times.astype(to_strings.dtype.newbyteorder())
573: (8)     assert_array_equal(res, to_strings)
574: (16)     from_strings_swapped = from_strings.astype(
575: (8)         from_strings.dtype.newbyteorder())
576: (8)     res = from_strings_swapped.astype(time_dtype)
577: (8)     assert_array_equal(res, times)
578: (8)     res = from_strings_swapped.astype(times.dtype.newbyteorder())
579: (8)     assert_array_equal(res, times)
580: (8)     res = from_strings.astype(times.dtype.newbyteorder())
581: (8)     assert_array_equal(res, times)
582: (4) def test_datetime_array_str(self):
583: (8)     a = np.array(['2011-03-16', '1920-01-01', '2013-05-19'], dtype='M')
584: (8)     assert_equal(str(a), "[ '2011-03-16' '1920-01-01' '2013-05-19' ]")
585: (8)     a = np.array(['2011-03-16T13:55', '1920-01-01T03:12'], dtype='M')
586: (20)     assert_equal(np.array2string(a, separator=', ',
587: (28)                     formatter={'datetime': lambda x:
588: (21)                         "'%s'" % np.datetime_as_string(x,
589: (8)                         timezone='UTC')}),
590: (8)                         "[ '2011-03-16T13:55Z', '1920-01-01T03:12Z' ]")

```

```

591: (4)
592: (8)
593: (8)
594: (8)
595: (8)
596: (8)
597: (8)
598: (8)
599: (8)
600: (8)
601: (8)
602: (8)
603: (8)
604: (4)
605: (8)
606: (12)
607: (12)
608: (12)
609: (12)
610: (12)
611: (12)
612: (25)
613: (12)
614: (12)
615: (25)
616: (8)
617: (14)
618: (14)
619: (8)
620: (8)
621: (14)
622: (14)
623: (8)
624: (8)
625: (14)
626: (14)
627: (8)
628: (4)
629: (8)
630: (8)
631: (8)
-1, -1, 0, 1))
632: (8)
633: (8)
-1, -1, 0, ({}, 'xxx'))
634: (8)
635: (4)
636: (8)
637: (12)
638: (16)
639: (16)
640: (12)
641: (16)
np.dtype(mM+'8[15Y'])),
642: (16)
643: (12)
644: (16)
np.dtype(mM+'8[24M'])),
645: (16)
646: (12)
647: (16)
648: (16)
649: (12)
650: (16)
651: (16)
652: (12)
653: (16)
np.dtype(mM+'8[49s'])),
654: (16)

      def test_timedelta_array_str(self):
          a = np.array([-1, 0, 100], dtype='m')
          assert_equal(str(a), "[ -1 0 100]")
          a = np.array(['NaT', 'NaT'], dtype='m')
          assert_equal(str(a), "[ 'NaT' 'NaT']")
          a = np.array([-1, 'NaT', 0], dtype='m')
          assert_equal(str(a), "[ -1 'NaT' 0]")
          a = np.array([-1, 'NaT', 1234567], dtype='m')
          assert_equal(str(a), "[ -1 'NaT' 1234567]")
          a = np.array([-1, 'NaT', 1234567], dtype='>m')
          assert_equal(str(a), "[ -1 'NaT' 1234567]")
          a = np.array([-1, 'NaT', 1234567], dtype='<m')
          assert_equal(str(a), "[ -1 'NaT' 1234567]")
      def test_pickle(self):
          for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
              dt = np.dtype('M8[7D]')
              assert_equal(pickle.loads(pickle.dumps(dt, protocol=proto)), dt)
              dt = np.dtype('M8[W]')
              assert_equal(pickle.loads(pickle.dumps(dt, protocol=proto)), dt)
              scalar = np.datetime64('2016-01-01T00:00:00.000000000')
              assert_equal(pickle.loads(pickle.dumps(scalar, protocol=proto)),
                           scalar)
              delta = scalar - np.datetime64('2015-01-01T00:00:00.000000000')
              assert_equal(pickle.loads(pickle.dumps(delta, protocol=proto)),
                           delta)
              pk1 = b"numpy\ndtype\np0\\n(S'M8'\\np1\\nI0\\nI1\\ntp2\\nRp3\\n" + \
                     b"(I4\\nS'<'\\np4\\nNNNI-1\\nI-1\\nI0\\n((dp5\\n(S'D'\\np6\\n" + \
                     b"I7\\nI1\\nI1\\ntp7\\ntp8\\ntp9\\nb."
              assert_equal(pickle.loads(pk1), np.dtype('<M8[7D]'))
              pk1 = b"numpy\ndtype\np0\\n(S'M8'\\np1\\nI0\\nI1\\ntp2\\nRp3\\n" + \
                     b"(I4\\nS'<'\\np4\\nNNNI-1\\nI-1\\nI0\\n((dp5\\n(S'W'\\np6\\n" + \
                     b"I1\\nI1\\nI1\\ntp7\\ntp8\\ntp9\\nb."
              assert_equal(pickle.loads(pk1), np.dtype('<M8[W]'))
              pk1 = b"numpy\ndtype\np0\\n(S'M8'\\np1\\nI0\\nI1\\ntp2\\nRp3\\n" + \
                     b"(I4\\nS'>'\\np4\\nNNNI-1\\nI-1\\nI0\\n((dp5\\n(S'us'\\np6\\n" + \
                     b"I1\\nI1\\nI1\\ntp7\\ntp8\\ntp9\\nb."
              assert_equal(pickle.loads(pk1), np.dtype('>M8[us]'))
      def test_setstate(self):
          "Verify that datetime dtype __setstate__ can handle bad arguments"
          dt = np.dtype('>M8[us]')
          assert_raises(ValueError, dt.__setstate__, (4, '>', None, None, None,
-1, -1, 0, 1))
          assert_(dt.__reduce__()[2] == np.dtype('>M8[us]').__reduce__()[2])
          assert_raises(TypeError, dt.__setstate__, (4, '>', None, None, None,
-1, -1, 0, ({}, 'xxx')))
          assert_(dt.__reduce__()[2] == np.dtype('>M8[us]').__reduce__()[2])
      def test_dtype_promotion(self):
          for mM in ['m', 'M']:
              assert_equal(
                  np.promote_types(np.dtype(mM+'8[2Y'])), np.dtype(mM+'8[2Y'])),
                  np.dtype(mM+'8[2Y'])))
              assert_equal(
                  np.promote_types(np.dtype(mM+'8[12Y'])),
                  np.dtype(mM+'8[3Y'])))
              assert_equal(
                  np.promote_types(np.dtype(mM+'8[62M'])),
                  np.dtype(mM+'8[2M'])))
              assert_equal(
                  np.promote_types(np.dtype(mM+'8[1W'])), np.dtype(mM+'8[2D'])),
                  np.dtype(mM+'8[1D'])))
              assert_equal(
                  np.promote_types(np.dtype(mM+'8[W']), np.dtype(mM+'8[13s'])),
                  np.dtype(mM+'8[s'])))
              assert_equal(
                  np.promote_types(np.dtype(mM+'8[13W']), np.dtype(mM+'8[7s']')))
```

```

655: (8)
656: (28)
657: (8)
658: (28)
659: (8)
660: (8)
661: (8)
662: (8)
663: (8)
664: (28)
665: (8)
666: (28)
667: (4)
668: (8)
669: (12)
<M8[D]"')
670: (8)
671: (8)
672: (12)
673: (8)
674: (4)
675: (8)
676: (22)
677: (56)
678: (8)
679: (12)
680: (12)
681: (12)
682: (12)
683: (12)
684: (12)
685: (12)
686: (12)
687: (12)
688: (12)
689: (12)
690: (28)
691: (8)
692: (21)
693: (12)
694: (12)
695: (12)
696: (12)
697: (12)
698: (12)
699: (12)
700: (12)
701: (12)
702: (12)
703: (12)
704: (28)
705: (4)
706: (8)
707: (21)
708: (8)
709: (13)
710: (8)
711: (13)
712: (8)
713: (13)
714: (8)
715: (13)
716: (4)
717: (8)
718: (12)
719: (12)
720: (16)
721: (16)
722: (29)

        assert_raises(TypeError, np.promote_types,
                      np.dtype('m8[Y)'), np.dtype('m8[D]'))
        assert_raises(TypeError, np.promote_types,
                      np.dtype('m8[M)'), np.dtype('m8[W]'))
        assert_raises(TypeError, np.promote_types, "float32", "m8")
        assert_raises(TypeError, np.promote_types, "m8", "float32")
        assert_raises(TypeError, np.promote_types, "uint64", "m8")
        assert_raises(TypeError, np.promote_types, "m8", "uint64")
        assert_raises(OverflowError, np.promote_types,
                     np.dtype('m8[W)'), np.dtype('m8[fs]'))
        assert_raises(OverflowError, np.promote_types,
                     np.dtype('m8[s)'), np.dtype('m8[as]'))

    def test_cast_overflow(self):
        def cast():
            numpy.datetime64("1971-01-01 00:00:00.0000000000000000").astype(
                "<M8[D]")
            assert_raises(OverflowError, cast)
        def cast2():
            numpy.datetime64("2014").astype("<M8[fs]")
            assert_raises(OverflowError, cast2)

    def test_pyobject_roundtrip(self):
        a = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0,
                     -1020040340, -2942398, -1, 0, 1, 234523453, 1199164176],
                     dtype=np.int64)
        for unit in ['M8[D]', 'M8[W]', 'M8[M]', 'M8[Y]']:
            b = a.copy().view(dtype=unit)
            b[0] = '-0001-01-01'
            b[1] = '-0001-12-31'
            b[2] = '0000-01-01'
            b[3] = '0001-01-01'
            b[4] = '1969-12-31'
            b[5] = '1970-01-01'
            b[6] = '9999-12-31'
            b[7] = '10000-01-01'
            b[8] = 'NaT'
            assert_equal(b.astype(object).astype(unit), b,
                        "Error roundtripping unit %s" % unit)
        for unit in ['M8[as]', 'M8[16fs]', 'M8[ps]', 'M8[us]',
                     'M8[300as]', 'M8[20us]']:
            b = a.copy().view(dtype=unit)
            b[0] = '-0001-01-01T00'
            b[1] = '-0001-12-31T00'
            b[2] = '0000-01-01T00'
            b[3] = '0001-01-01T00'
            b[4] = '1969-12-31T23:59:59.999999'
            b[5] = '1970-01-01T00'
            b[6] = '9999-12-31T23:59:59.999999'
            b[7] = '10000-01-01T00'
            b[8] = 'NaT'
            assert_equal(b.astype(object).astype(unit), b,
                        "Error roundtripping unit %s" % unit)

    def test_month_truncation(self):
        assert_equal(np.array('1945-03-01', dtype='M8[M]'),
                    np.array('1945-03-31', dtype='M8[M]'))
        assert_equal(np.array('1969-11-01', dtype='M8[M]'),
                    np.array('1969-11-30T23:59:59.999999', dtype='M').astype('M8[M]'))
        assert_equal(np.array('1969-12-01', dtype='M8[M]'),
                    np.array('1969-12-31T23:59:59.999999', dtype='M').astype('M8[M]'))
        assert_equal(np.array('1970-01-01', dtype='M8[M]'),
                    np.array('1970-01-31T23:59:59.999999', dtype='M').astype('M8[M]'))
        assert_equal(np.array('1980-02-01', dtype='M8[M]'),
                    np.array('1980-02-29T23:59:59.999999', dtype='M').astype('M8[M]'))

    def test_different_unit_comparison(self):
        for unit1 in ['Y', 'M', 'D']:
            dt1 = np.dtype('M8[%s]' % unit1)
            for unit2 in ['Y', 'M', 'D']:
                dt2 = np.dtype('M8[%s]' % unit2)
                assert_equal(np.array('1945', dtype=dt1),
                            np.array('1945', dtype=dt2))

```

```

723: (16) assert_equal(np.array('1970', dtype=dt1),
724: (29)         np.array('1970', dtype=dt2))
725: (16) assert_equal(np.array('9999', dtype=dt1),
726: (29)         np.array('9999', dtype=dt2))
727: (16) assert_equal(np.array('10000', dtype=dt1),
728: (29)         np.array('10000-01-01', dtype=dt2))
729: (16) assert_equal(np.datetime64('1945', unit1),
730: (29)         np.datetime64('1945', unit2))
731: (16) assert_equal(np.datetime64('1970', unit1),
732: (29)         np.datetime64('1970', unit2))
733: (16) assert_equal(np.datetime64('9999', unit1),
734: (29)         np.datetime64('9999', unit2))
735: (16) assert_equal(np.datetime64('10000', unit1),
736: (29)         np.datetime64('10000-01-01', unit2))
737: (8) for unit1 in ['6h', 'h', 'm', 's', '10ms', 'ms', 'us']:
738: (12)     dt1 = np.dtype('M8[%s]' % unit1)
739: (12)     for unit2 in ['h', 'm', 's', 'ms', 'us']:
740: (16)         dt2 = np.dtype('M8[%s]' % unit2)
741: (16)         assert_equal(np.array('1945-03-12T18', dtype=dt1),
742: (29)                 np.array('1945-03-12T18', dtype=dt2))
743: (16)         assert_equal(np.array('1970-03-12T18', dtype=dt1),
744: (29)                 np.array('1970-03-12T18', dtype=dt2))
745: (16)         assert_equal(np.array('9999-03-12T18', dtype=dt1),
746: (29)                 np.array('9999-03-12T18', dtype=dt2))
747: (16)         assert_equal(np.array('10000-01-01T00', dtype=dt1),
748: (29)                 np.array('10000-01-01T00', dtype=dt2))
749: (16)         assert_equal(np.datetime64('1945-03-12T18', unit1),
750: (29)                 np.datetime64('1945-03-12T18', unit2))
751: (16)         assert_equal(np.datetime64('1970-03-12T18', unit1),
752: (29)                 np.datetime64('1970-03-12T18', unit2))
753: (16)         assert_equal(np.datetime64('9999-03-12T18', unit1),
754: (29)                 np.datetime64('9999-03-12T18', unit2))
755: (16)         assert_equal(np.datetime64('10000-01-01T00', unit1),
756: (29)                 np.datetime64('10000-01-01T00', unit2))
757: (8) for unit1 in ['D', '12h', 'h', 'm', 's', '4s', 'ms', 'us']:
758: (12)     dt1 = np.dtype('M8[%s]' % unit1)
759: (12)     for unit2 in ['D', 'h', 'm', 's', 'ms', 'us']:
760: (16)         dt2 = np.dtype('M8[%s]' % unit2)
761: (16)         assert_(np.equal(np.array('1932-02-17',
762: (21)                         np.array('1932-02-17T00:00:00', dtype='M').astype(dt2),
763: (21)                         casting='unsafe'))
764: (16)         assert_(np.equal(np.array('10000-04-27',
765: (21)                         np.array('10000-04-27T00:00:00', dtype='M').astype(dt2),
766: (21)                         casting='unsafe'))
767: (8)         a = np.array('2012-12-21', dtype='M8[D]')
768: (8)         b = np.array(3, dtype='m8[D]')
769: (8)         assert_raises(TypeError, np.less, a, b)
770: (8)         assert_raises(TypeError, np.less, a, b, casting='unsafe')
771: (4)     def test_datetime_like(self):
772: (8)         a = np.array([3], dtype='m8[4D]')
773: (8)         b = np.array(['2012-12-21'], dtype='M8[D]')
774: (8)         assert_equal(np.ones_like(a).dtype, a.dtype)
775: (8)         assert_equal(np.zeros_like(a).dtype, a.dtype)
776: (8)         assert_equal(np.empty_like(a).dtype, a.dtype)
777: (8)         assert_equal(np.ones_like(b).dtype, b.dtype)
778: (8)         assert_equal(np.zeros_like(b).dtype, b.dtype)
779: (8)         assert_equal(np.empty_like(b).dtype, b.dtype)
780: (4)     def test_datetime_unary(self):
781: (8)         for tda, tdb, tdzero, tdone, tdmone in \
782: (16)             [
783: (17)                 (np.array([3], dtype='m8[D]'),
784: (18)                     np.array([-3], dtype='m8[D]'),
785: (18)                     np.array([0], dtype='m8[D]'),
786: (18)                     np.array([1], dtype='m8[D]'),
787: (18)                     np.array([-1], dtype='m8[D'])),
788: (17)                 (np.timedelta64(3, '[D]'),
789: (18)                     np.timedelta64(-3, '[D]'),

```

```

790: (18) np.timedelta64(0, '[D]'),
791: (18) np.timedelta64(1, '[D]'),
792: (18) np.timedelta64(-1, '[D]'))]:
793: (12) assert_equal(-tdb, tda)
794: (12) assert_equal((-tdb).dtype, tda.dtype)
795: (12) assert_equal(np.negative(tdb), tda)
796: (12) assert_equal(np.negative(tdb).dtype, tda.dtype)
797: (12) assert_equal(np.positive(tda), tda)
798: (12) assert_equal(np.positive(tda).dtype, tda.dtype)
799: (12) assert_equal(np.positive(tdb), tdb)
800: (12) assert_equal(np.positive(tdb).dtype, tdb.dtype)
801: (12) assert_equal(np.absolute(tdb), tda)
802: (12) assert_equal(np.absolute(tdb).dtype, tda.dtype)
803: (12) assert_equal(np.sign(tda), tdone)
804: (12) assert_equal(np.sign(tdb), tdmone)
805: (12) assert_equal(np.sign(tdzero), tdzero)
806: (12) assert_equal(np.sign(tda).dtype, tda.dtype)
807: (12) assert_
808: (4) def test_datetime_add(self):
809: (8)     for dta, dtb, dtc, dtnat, tda, tdb, tdc in \
810: (20)         [
811: (21)             (np.array(['2012-12-21'], dtype='M8[D']),
812: (22)                 np.array(['2012-12-24'], dtype='M8[D']),
813: (22)                 np.array(['2012-12-21T11'], dtype='M8[h']),
814: (22)                 np.array(['NaT'], dtype='M8[D']),
815: (22)                 np.array([3], dtype='m8[D']),
816: (22)                 np.array([11], dtype='m8[h']),
817: (22)                 np.array([3*24 + 11], dtype='m8[h'])),
818: (21)             (np.datetime64('2012-12-21', '[D']),
819: (22)                 np.datetime64('2012-12-24', '[D']),
820: (22)                 np.datetime64('2012-12-21T11', '[h']),
821: (22)                 np.datetime64('NaT', '[D']),
822: (22)                 np.timedelta64(3, '[D']),
823: (22)                 np.timedelta64(11, '[h']),
824: (22)                 np.timedelta64(3*24 + 11, '[h']))]:
825: (12)             assert_equal(tda + tdb, tdc)
826: (12)             assert_equal((tda + tdb).dtype, np.dtype('m8[h']))
827: (12)             assert_equal(tdb + True, tdb + 1)
828: (12)             assert_equal((tdb + True).dtype, np.dtype('m8[h']))
829: (12)             assert_equal(tdb + 3*24, tdc)
830: (12)             assert_equal((tdb + 3*24).dtype, np.dtype('m8[h']))
831: (12)             assert_equal(False + tdb, tdb)
832: (12)             assert_equal((False + tdb).dtype, np.dtype('m8[h']))
833: (12)             assert_equal(3*24 + tdb, tdc)
834: (12)             assert_equal((3*24 + tdb).dtype, np.dtype('m8[h']))
835: (12)             assert_equal(dta + True, dta + 1)
836: (12)             assert_equal(dtnat + True, dtnat)
837: (12)             assert_equal((dta + True).dtype, np.dtype('M8[D']))
838: (12)             assert_equal(dta + 3, dtb)
839: (12)             assert_equal(dtnat + 3, dtnat)
840: (12)             assert_equal((dta + 3).dtype, np.dtype('M8[D']))
841: (12)             assert_equal(False + dta, dta)
842: (12)             assert_equal(False + dtnat, dtnat)
843: (12)             assert_equal((False + dta).dtype, np.dtype('M8[D']))
844: (12)             assert_equal(3 + dta, dtb)
845: (12)             assert_equal(3 + dtnat, dtnat)
846: (12)             assert_equal((3 + dta).dtype, np.dtype('M8[D']))
847: (12)             assert_equal(dta + tda, dtb)
848: (12)             assert_equal(dtnat + tda, dtnat)
849: (12)             assert_equal((dta + tda).dtype, np.dtype('M8[D']))
850: (12)             assert_equal(tda + dta, dtb)
851: (12)             assert_equal(tda + dtnat, dtnat)
852: (12)             assert_equal((tda + dta).dtype, np.dtype('M8[D']))
853: (12)             assert_equal(np.add(dta, tdb, casting='unsafe'), dtc)
854: (12)             assert_equal(np.add(dta, tdb, casting='unsafe').dtype,
855: (25)                             np.dtype('M8[h']))
856: (12)             assert_equal(np.add(tdb, dta, casting='unsafe'), dtc)
857: (12)             assert_equal(np.add(tdb, dta, casting='unsafe').dtype,
858: (25)                             np.dtype('M8[h']))

```

```

859: (12)             assert_raises(TypeError, np.add, dta, dtb)
860: (4)              def test_datetime_subtract(self):
861: (8)                for dta, dtb, dtc, dtd, dte, dtnat, tda, tdb, tdc in \
862: (20)                  [
863: (21)                    (np.array(['2012-12-21'], dtype='M8[D']),
864: (22)                      np.array(['2012-12-24'], dtype='M8[D']),
865: (22)                      np.array(['1940-12-24'], dtype='M8[D']),
866: (22)                      np.array(['1940-12-24T00'], dtype='M8[h']),
867: (22)                      np.array(['1940-12-23T13'], dtype='M8[h']),
868: (22)                      np.array(['NaT'], dtype='M8[D']),
869: (22)                      np.array([3], dtype='m8[D']),
870: (22)                      np.array([11], dtype='m8[h']),
871: (22)                      np.array([3*24 - 11], dtype='m8[h'])),
872: (21)                    (np.datetime64('2012-12-21', '[D']),
873: (22)                      np.datetime64('2012-12-24', '[D']),
874: (22)                      np.datetime64('1940-12-24', '[D']),
875: (22)                      np.datetime64('1940-12-24T00', '[h']),
876: (22)                      np.datetime64('1940-12-23T13', '[h']),
877: (22)                      np.datetime64('NaT', '[D']),
878: (22)                      np.timedelta64(3, '[D']),
879: (22)                      np.timedelta64(11, '[h']),
880: (22)                      np.timedelta64(3*24 - 11, '[h'))]:
881: (12)            assert_equal(tda - tdb, tdc)
882: (12)            assert_equal((tda - tdb).dtype, np.dtype('m8[h']))
883: (12)            assert_equal(tdb - tda, -tdc)
884: (12)            assert_equal((tdb - tda).dtype, np.dtype('m8[h']))
885: (12)            assert_equal(tdc - True, tdc - 1)
886: (12)            assert_equal((tdc - True).dtype, np.dtype('m8[h']))
887: (12)            assert_equal(tdc - 3*24, -tdb)
888: (12)            assert_equal((tdc - 3*24).dtype, np.dtype('m8[h']))
889: (12)            assert_equal(False - tdb, -tdb)
890: (12)            assert_equal((False - tdb).dtype, np.dtype('m8[h']))
891: (12)            assert_equal(3*24 - tdb, tdc)
892: (12)            assert_equal((3*24 - tdb).dtype, np.dtype('m8[h']))
893: (12)            assert_equal(dtb - True, dtb - 1)
894: (12)            assert_equal(dtnat - True, dtnat)
895: (12)            assert_equal((dtb - True).dtype, np.dtype('M8[D']))
896: (12)            assert_equal(dtb - 3, dta)
897: (12)            assert_equal(dtnat - 3, dtnat)
898: (12)            assert_equal((dtb - 3).dtype, np.dtype('M8[D']))
899: (12)            assert_equal(dtb - tda, dta)
900: (12)            assert_equal(dtnat - tda, dtnat)
901: (12)            assert_equal((dtb - tda).dtype, np.dtype('M8[D']))
902: (12)            assert_equal(np.subtract(dtc, tdb, casting='unsafe'), dte)
903: (12)            assert_equal(np.subtract(dtc, tdb, casting='unsafe').dtype,
904: (25)                          np.dtype('M8[h']))
905: (12)            assert_equal(np.subtract(dtc, dtd, casting='unsafe'),
906: (25)                          np.timedelta64(0, 'h'))
907: (12)            assert_equal(np.subtract(dtc, dtd, casting='unsafe').dtype,
908: (25)                          np.dtype('m8[h']))
909: (12)            assert_equal(np.subtract(dtd, dtc, casting='unsafe'),
910: (25)                          np.timedelta64(0, 'h'))
911: (12)            assert_equal(np.subtract(dtd, dtc, casting='unsafe').dtype,
912: (25)                          np.dtype('m8[h']))
913: (12)            assert_raises(TypeError, np.subtract, tda, dta)
914: (12)            assert_raises(TypeError, np.subtract, False, dta)
915: (12)            assert_raises(TypeError, np.subtract, 3, dta)
916: (4)              def test_datetime_multiply(self):
917: (8)                for dta, tda, tdb, tdc in \
918: (20)                  [
919: (21)                    (np.array(['2012-12-21'], dtype='M8[D']),
920: (22)                      np.array([6], dtype='m8[h']),
921: (22)                      np.array([9], dtype='m8[h']),
922: (22)                      np.array([12], dtype='m8[h'])),
923: (21)                    (np.datetime64('2012-12-21', '[D']),
924: (22)                      np.timedelta64(6, '[h']),
925: (22)                      np.timedelta64(9, '[h']),
926: (22)                      np.timedelta64(12, '[h'))]:
927: (12)            assert_equal(tda * 2, tdc)

```

```

928: (12) assert_equal((tda * 2).dtype, np.dtype('m8[h']))
929: (12) assert_equal(2 * tda, tdc)
930: (12) assert_equal((2 * tda).dtype, np.dtype('m8[h]'))
931: (12) assert_equal(tda * 1.5, tdb)
932: (12) assert_equal((tda * 1.5).dtype, np.dtype('m8[h]'))
933: (12) assert_equal(1.5 * tda, tdb)
934: (12) assert_equal((1.5 * tda).dtype, np.dtype('m8[h]'))
935: (12) assert_raises(TypeError, np.multiply, tda, tdb)
936: (12) assert_raises(TypeError, np.multiply, dta, tda)
937: (12) assert_raises(TypeError, np.multiply, tda, dta)
938: (12) assert_raises(TypeError, np.multiply, dta, 2)
939: (12) assert_raises(TypeError, np.multiply, 2, dta)
940: (12) assert_raises(TypeError, np.multiply, dta, 1.5)
941: (12) assert_raises(TypeError, np.multiply, 1.5, dta)
942: (8) with suppress_warnings() as sup:
943: (12)     sup.filter(RuntimeWarning, "invalid value encountered in
944: multiply")
945: (12)
946: (16) def check(a, b, res):
947: (16)     assert_equal(a * b, res)
948: (12)     assert_equal(b * a, res)
949: (16)     for tp in (int, float):
950: (16)         check(nat, tp(2), nat)
951: (12)         check(nat, tp(0), nat)
952: (16)         for f in (float('inf'), float('nan')):
953: (16)             check(np.timedelta64(1), f, nat)
954: (16)             check(np.timedelta64(0), f, nat)
955: (16)             check(nat, f, nat)
956: (4) @pytest.mark.parametrize("op1, op2, exp", [
957: (8)     (np.timedelta64(7, 's'),
958: (9)         np.timedelta64(4, 's'),
959: (9)         1),
960: (8)     (np.timedelta64(7, 's'),
961: (9)         np.timedelta64(-4, 's'),
962: (9)         -2),
963: (8)     (np.timedelta64(8, 's'),
964: (9)         np.timedelta64(-4, 's'),
965: (9)         -2),
966: (8)     (np.timedelta64(1, 'm'),
967: (9)         np.timedelta64(31, 's'),
968: (9)         1),
969: (8)     (np.timedelta64(1890),
970: (9)         np.timedelta64(31,
971: (9)         60),
972: (8)     (np.timedelta64(2, 'Y'),
973: (9)         np.timedelta64('13', 'M'),
974: (9)         1),
975: (8)     (np.array([1, 2, 3], dtype='m8'),
976: (9)         np.array([2], dtype='m8'),
977: (8)         np.array([0, 1, 1], dtype=np.int64)),
978: (4) )
979: (8) def test_timedelta_floor_divide(self, op1, op2, exp):
980: (4)     assert_equal(op1 // op2, exp)
981: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
982: (4) @pytest.mark.parametrize("op1, op2", [
983: (8)     (np.timedelta64(10, 'us'),
984: (9)         np.timedelta64(0, 'us')),
985: (8)     (np.timedelta64('NaT'),
986: (9)         np.timedelta64(50, 'us')),
987: (8)     (np.timedelta64(np.iinfo(np.int64).min),
988: (9)         np.timedelta64(-1)),
989: (8) )
990: (8) def test_timedelta_floor_div_warnings(self, op1, op2):
991: (12)     with assert_warns(RuntimeWarning):
992: (12)         actual = op1 // op2
993: (12)         assert_equal(actual, 0)
994: (12)         assert_equal(actual.dtype, np.int64)
995: (4) @pytest.mark.parametrize("val1, val2", [
996: (8)     (9007199254740993, 1),

```

```

996: (8)                               (9007199254740999, -2),
997: (8)
998: (4)
999: (8)
1000: (8)
1001: (8)
1002: (8)
1003: (8)
1004: (4)
1005: (8)
1006: (9)
1007: (8)
1008: (9)
1009: (8)
1010: (4)
1011: (8)
1012: (12)
1013: (4)
1014: (8)
1015: (9)
1016: (8)
1017: (9)
1018: (8)
1019: (9)
1020: (8)
1021: (9)
1022: (8)
1023: (9)
1024: (8)
1025: (9)
1026: (8)
1027: (9)
1028: (8)
1029: (4)
1030: (8)
1031: (8)
1032: (4)
1033: (4)
1034: (8)
1035: (9)
1036: (8)
1037: (9)
1038: (8)
1039: (9)
1040: (8)
1041: (4)
1042: (8)
1043: (12)
1044: (8)
1045: (12)
1046: (8)
1047: (4)
1048: (8)
1049: (20)
1050: (21)
1051: (22)
1052: (22)
1053: (22)
1054: (22)
1055: (21)
1056: (22)
1057: (22)
1058: (22)
1059: (22)
1060: (12)
1061: (12)
1062: (12)
1063: (12)
1064: (12)

        ])
    def test_timedelta_floor_div_precision(self, val1, val2):
        op1 = np.timedelta64(val1)
        op2 = np.timedelta64(val2)
        actual = op1 // op2
        expected = val1 // val2
        assert_equal(actual, expected)
    @pytest.mark.parametrize("val1, val2", [
        (np.timedelta64(7, 'Y'),
         np.timedelta64(3, 's')),
        (np.timedelta64(7, 'M'),
         np.timedelta64(1, 'D')),
    ])
    def test_timedelta_floor_div_error(self, val1, val2):
        with assert_raises_regex(TypeError, "common metadata divisor"):
            val1 // val2
    @pytest.mark.parametrize("op1, op2", [
        (np.timedelta64(7, 's'),
         np.timedelta64(4, 's')),
        (np.timedelta64(7, 's'),
         np.timedelta64(-4, 's')),
        (np.timedelta64(8, 's'),
         np.timedelta64(-4, 's')),
        (np.timedelta64(1, 'm'),
         np.timedelta64(31, 's')),
        (np.timedelta64(1890),
         np.timedelta64(31)),
        (np.timedelta64(2, 'Y'),
         np.timedelta64('13', 'M')),
        (np.array([1, 2, 3], dtype='m8'),
         np.array([2], dtype='m8')),
    ])
    def test_timedelta_divmod(self, op1, op2):
        expected = (op1 // op2, op1 % op2)
        assert_equal(divmod(op1, op2), expected)
    @pytest.mark.skipif(IS_WASM, reason="does not work in wasm")
    @pytest.mark.parametrize("op1, op2", [
        (np.timedelta64(10, 'us'),
         np.timedelta64(0, 'us')),
        (np.timedelta64('NaT'),
         np.timedelta64(50, 'us')),
        (np.timedelta64(np.iinfo(np.int64).min),
         np.timedelta64(-1)),
    ])
    def test_timedelta_divmod_warnings(self, op1, op2):
        with assert_warns(RuntimeWarning):
            expected = (op1 // op2, op1 % op2)
        with assert_warns(RuntimeWarning):
            actual = divmod(op1, op2)
        assert_equal(actual, expected)
    def test_datetime_divide(self):
        for dta, tda, tdb, tdc, tdd in \
            [
                (np.array(['2012-12-21'], dtype='M8[D']),
                 np.array([6], dtype='m8[h']),
                 np.array([9], dtype='m8[h']),
                 np.array([12], dtype='m8[h']),
                 np.array([6], dtype='m8[m'])),
                (np.datetime64('2012-12-21', '[D]'),
                 np.timedelta64(6, '[h]'),
                 np.timedelta64(9, '[h]'),
                 np.timedelta64(12, '[h]'),
                 np.timedelta64(6, '[m']))]:
            assert_equal(tdc / 2, tda)
            assert_equal((tdc / 2).dtype, np.dtype('m8[h']))
            assert_equal(tda / 0.5, tdc)
            assert_equal((tda / 0.5).dtype, np.dtype('m8[h']))
            assert_equal(tda / tdb, 6 / 9)

```

```

1065: (12) assert_equal(np.divide(tda, tdb), 6 / 9)
1066: (12) assert_equal(np.true_divide(tda, tdb), 6 / 9)
1067: (12) assert_equal(tdb / tda, 9 / 6)
1068: (12) assert_equal((tda / tdb).dtype, np.dtype('f8'))
1069: (12) assert_equal(tda / tdd, 60)
1070: (12) assert_equal(tdd / tda, 1 / 60)
1071: (12) assert_raises(TypeError, np.divide, 2, tdb)
1072: (12) assert_raises(TypeError, np.divide, 0.5, tdb)
1073: (12) assert_raises(TypeError, np.divide, dta, tda)
1074: (12) assert_raises(TypeError, np.divide, tda, dta)
1075: (12) assert_raises(TypeError, np.divide, dta, 2)
1076: (12) assert_raises(TypeError, np.divide, 2, dta)
1077: (12) assert_raises(TypeError, np.divide, dta, 1.5)
1078: (12) assert_raises(TypeError, np.divide, 1.5, dta)
1079: (8) with suppress_warnings() as sup:
1080: (12)     sup.filter(RuntimeWarning, r".*encountered in divide")
1081: (12)     nat = np.timedelta64('NaT')
1082: (12)     for tp in (int, float):
1083: (16)         assert_equal(np.timedelta64(1) / tp(0), nat)
1084: (16)         assert_equal(np.timedelta64(0) / tp(0), nat)
1085: (16)         assert_equal(nat / tp(0), nat)
1086: (16)         assert_equal(nat / tp(2), nat)
1087: (12)         assert_equal(np.timedelta64(1) / float('inf'), np.timedelta64(0))
1088: (12)         assert_equal(np.timedelta64(0) / float('inf'), np.timedelta64(0))
1089: (12)         assert_equal(nat / float('inf'), nat)
1090: (12)         assert_equal(np.timedelta64(1) / float('nan'), nat)
1091: (12)         assert_equal(np.timedelta64(0) / float('nan'), nat)
1092: (12)         assert_equal(nat / float('nan'), nat)
1093: (4) def test_datetime_compare(self):
1094: (8)     a = np.datetime64('2000-03-12T18:00:00.000000')
1095: (8)     b = np.array(['2000-03-12T18:00:00.000000',
1096: (22)                 '2000-03-12T17:59:59.999999',
1097: (22)                 '2000-03-12T18:00:00.000001',
1098: (22)                 '1970-01-11T12:00:00.909090',
1099: (22)                 '2016-01-11T12:00:00.909090'],
1100: (22)                 dtype='datetime64[us]')
1101: (8)     assert_equal(np.equal(a, b), [1, 0, 0, 0, 0])
1102: (8)     assert_equal(np.not_equal(a, b), [0, 1, 1, 1, 1])
1103: (8)     assert_equal(np.less(a, b), [0, 0, 1, 0, 1])
1104: (8)     assert_equal(np.less_equal(a, b), [1, 0, 1, 0, 1])
1105: (8)     assert_equal(np.greater(a, b), [0, 1, 0, 1, 0])
1106: (8)     assert_equal(np.greater_equal(a, b), [1, 1, 0, 1, 0])
1107: (4) def test_datetime_compare_nat(self):
1108: (8)     dt_nat = np.datetime64('NaT', 'D')
1109: (8)     dt_other = np.datetime64('2000-01-01')
1110: (8)     td_nat = np.timedelta64('NaT', 'h')
1111: (8)     td_other = np.timedelta64(1, 'h')
1112: (8)     for op in [np.equal, np.less, np.less_equal,
1113: (19)                 np.greater, np.greater_equal]:
1114: (12)         assert_(not op(dt_nat, dt_nat))
1115: (12)         assert_(not op(dt_nat, dt_other))
1116: (12)         assert_(not op(dt_other, dt_nat))
1117: (12)         assert_(not op(td_nat, td_nat))
1118: (12)         assert_(not op(td_nat, td_other))
1119: (12)         assert_(not op(td_other, td_nat))
1120: (8)         assert_(np.not_equal(dt_nat, dt_nat))
1121: (8)         assert_(np.not_equal(dt_nat, dt_other))
1122: (8)         assert_(np.not_equal(dt_other, dt_nat))
1123: (8)         assert_(np.not_equal(td_nat, td_nat))
1124: (8)         assert_(np.not_equal(td_nat, td_other))
1125: (8)         assert_(np.not_equal(td_other, td_nat))
1126: (4) def test_datetime_minmax(self):
1127: (8)     a = np.array('1999-03-12T13', dtype='M8[2m]')
1128: (8)     b = np.array('1999-03-12T12', dtype='M8[s]')
1129: (8)     assert_equal(np.minimum(a, b), b)
1130: (8)     assert_equal(np.minimum(a, b).dtype, np.dtype('M8[s]'))
1131: (8)     assert_equal(np.fmin(a, b), b)
1132: (8)     assert_equal(np.fmin(a, b).dtype, np.dtype('M8[s]'))
1133: (8)     assert_equal(np.maximum(a, b), a)

```

```

1134: (8) assert_equal(np.maximum(a, b).dtype, np.dtype('M8[s]'))
1135: (8) assert_equal(np.fmax(a, b), a)
1136: (8) assert_equal(np.fmax(a, b).dtype, np.dtype('M8[s]'))
1137: (8) assert_equal(np.minimum(a.view('i8'), b.view('i8')), a.view('i8'))
1138: (8) a = np.array('1999-03-12T13', dtype='M8[2m]')
1139: (8) dtnat = np.array('Nat', dtype='M8[h]')
1140: (8) assert_equal(np.minimum(a, dtnat), dtnat)
1141: (8) assert_equal(np.minimum(dtnat, a), dtnat)
1142: (8) assert_equal(np.maximum(a, dtnat), dtnat)
1143: (8) assert_equal(np.maximum(dtnat, a), dtnat)
1144: (8) assert_equal(np.fmin(dtnat, a), a)
1145: (8) assert_equal(np.fmin(a, dtnat), a)
1146: (8) assert_equal(np.fmax(dtnat, a), a)
1147: (8) assert_equal(np.fmax(a, dtnat), a)
1148: (8) a = np.array(3, dtype='m8[h]')
1149: (8) b = np.array(3*3600 - 3, dtype='m8[s]')
1150: (8) assert_equal(np.minimum(a, b), b)
1151: (8) assert_equal(np.minimum(a, b).dtype, np.dtype('m8[s]'))
1152: (8) assert_equal(np.fmin(a, b), b)
1153: (8) assert_equal(np.fmin(a, b).dtype, np.dtype('m8[s]'))
1154: (8) assert_equal(np.maximum(a, b), a)
1155: (8) assert_equal(np.maximum(a, b).dtype, np.dtype('m8[s]'))
1156: (8) assert_equal(np.fmax(a, b), a)
1157: (8) assert_equal(np.fmax(a, b).dtype, np.dtype('m8[s]'))
1158: (8) assert_equal(np.minimum(a.view('i8'), b.view('i8')), a.view('i8'))
1159: (8) a = np.array(3, dtype='m8[h]')
1160: (8) b = np.array('1999-03-12T12', dtype='M8[s]')
1161: (8) assert_raises(TypeError, np.minimum, a, b, casting='same_kind')
1162: (8) assert_raises(TypeError, np.maximum, a, b, casting='same_kind')
1163: (8) assert_raises(TypeError, np.fmin, a, b, casting='same_kind')
1164: (8) assert_raises(TypeError, np.fmax, a, b, casting='same_kind')
1165: (4) def test_hours(self):
1166: (8) t = np.ones(3, dtype='M8[s]')
1167: (8) t[0] = 60*60*24 + 60*60*10
1168: (8) assert_(t[0].item().hour == 10)
1169: (4) def test_divisor_conversion_year(self):
1170: (8) assert_(np.dtype('M8[Y/4]') == np.dtype('M8[3M]'))
1171: (8) assert_(np.dtype('M8[Y/13]') == np.dtype('M8[4W]'))
1172: (8) assert_(np.dtype('M8[3Y/73]') == np.dtype('M8[15D]'))
1173: (4) def test_divisor_conversion_month(self):
1174: (8) assert_(np.dtype('M8[M/2]') == np.dtype('M8[2W]'))
1175: (8) assert_(np.dtype('M8[M/15]') == np.dtype('M8[2D]'))
1176: (8) assert_(np.dtype('M8[3M/40]') == np.dtype('M8[54h]'))
1177: (4) def test_divisor_conversion_week(self):
1178: (8) assert_(np.dtype('m8[W/7]') == np.dtype('m8[D]'))
1179: (8) assert_(np.dtype('m8[3W/14]') == np.dtype('m8[36h]'))
1180: (8) assert_(np.dtype('m8[5W/140]') == np.dtype('m8[360m]'))
1181: (4) def test_divisor_conversion_day(self):
1182: (8) assert_(np.dtype('M8[D/12]') == np.dtype('M8[2h]'))
1183: (8) assert_(np.dtype('M8[D/120]') == np.dtype('M8[12m]'))
1184: (8) assert_(np.dtype('M8[3D/960]') == np.dtype('M8[270s]'))
1185: (4) def test_divisor_conversion_hour(self):
1186: (8) assert_(np.dtype('m8[h/30]') == np.dtype('m8[2m]'))
1187: (8) assert_(np.dtype('m8[3h/300]') == np.dtype('m8[36s]'))
1188: (4) def test_divisor_conversion_minute(self):
1189: (8) assert_(np.dtype('m8[m/30]') == np.dtype('m8[2s]'))
1190: (8) assert_(np.dtype('m8[3m/300]') == np.dtype('m8[600ms]'))
1191: (4) def test_divisor_conversion_second(self):
1192: (8) assert_(np.dtype('m8[s/100]') == np.dtype('m8[10ms]'))
1193: (8) assert_(np.dtype('m8[3s/10000]') == np.dtype('m8[300us]'))
1194: (4) def test_divisor_conversion_fs(self):
1195: (8) assert_(np.dtype('M8[fs/100]') == np.dtype('M8[10as]'))
1196: (8) assert_raises(ValueError, lambda: np.dtype('M8[3fs/10000]'))
1197: (4) def test_divisor_conversion_as(self):
1198: (8) assert_raises(ValueError, lambda: np.dtype('M8[as/10]'))
1199: (4) def test_string_parser_variants(self):
1200: (8) assert_equal(np.array(['1980-02-29T01:02:03']), np.dtype('M8[s']))
1201: (21) np.array(['1980-02-29 01:02:03'], np.dtype('M8[s]'))
1202: (8) assert_equal(np.array(['+1980-02-29T01:02:03']), np.dtype('M8[s]')),

```

```

1203: (21) np.array(['+1980-02-29 01:02:03'], np.dtype('M8[s]')))
1204: (8) assert_equal(np.array(['-1980-02-29T01:02:03'], np.dtype('M8[s]')),
1205: (21) np.array(['-1980-02-29 01:02:03'], np.dtype('M8[s]')))
1206: (8) with assert_warnings(DeprecationWarning):
1207: (12)     assert_equal(
1208: (16)         np.array(['+1980-02-29T01:02:03'], np.dtype('M8[s]')),
1209: (16)         np.array(['+1980-02-29 01:02:03Z'], np.dtype('M8[s]')))
1210: (8) with assert_warnings(DeprecationWarning):
1211: (12)     assert_equal(
1212: (16)         np.array(['-1980-02-29T01:02:03'], np.dtype('M8[s]')),
1213: (16)         np.array(['-1980-02-29 01:02:03Z'], np.dtype('M8[s]')))
1214: (8) with assert_warnings(DeprecationWarning):
1215: (12)     assert_equal(
1216: (16)         np.array(['1980-02-29T02:02:03'], np.dtype('M8[s]')),
1217: (16)         np.array(['1980-02-29 00:32:03-0130'], np.dtype('M8[s]')))
1218: (8) with assert_warnings(DeprecationWarning):
1219: (12)     assert_equal(
1220: (16)         np.array(['1980-02-28T22:32:03'], np.dtype('M8[s]')),
1221: (16)         np.array(['1980-02-29 00:02:03+01:30'], np.dtype('M8[s]')))
1222: (8) with assert_warnings(DeprecationWarning):
1223: (12)     assert_equal(
1224: (16)         np.array(['1980-02-29T02:32:03.506'], np.dtype('M8[s]')),
1225: (16)         np.array(['1980-02-29 00:32:03.506-02'], np.dtype('M8[s]')))
1226: (8) with assert_warnings(DeprecationWarning):
1227: (12)     assert_equal(np.datetime64('1977-03-02T12:30-0230'),
1228: (25)             np.datetime64('1977-03-02T15:00'))
1229: (4) def test_string_parser_error_check(self):
1230: (8)     assert_raises(ValueError, np.array, ['badvalue'], np.dtype('M8[us]'))
1231: (8)     assert_raises(ValueError, np.array, ['1980X'], np.dtype('M8[us]'))
1232: (8)     assert_raises(ValueError, np.array, ['1980-'], np.dtype('M8[us]'))
1233: (8)     assert_raises(ValueError, np.array, ['1980-00'], np.dtype('M8[us]'))
1234: (8)     assert_raises(ValueError, np.array, ['1980-13'], np.dtype('M8[us]'))
1235: (8)     assert_raises(ValueError, np.array, ['1980-1'], np.dtype('M8[us]'))
1236: (8)     assert_raises(ValueError, np.array, ['1980-1-02'], np.dtype('M8[us]'))
1237: (8)     assert_raises(ValueError, np.array, ['1980-Mor'], np.dtype('M8[us]'))
1238: (8)     assert_raises(ValueError, np.array, ['1980-01-'], np.dtype('M8[us]'))
1239: (8)     assert_raises(ValueError, np.array, ['1980-01-0'], np.dtype('M8[us]'))
1240: (8)     assert_raises(ValueError, np.array, ['1980-01-00'],
1241: (8) np.dtype('M8[us]'))
1242: (8) np.dtype('M8[us]'))
1243: (8) np.dtype('M8[us]'))
1244: (8) np.dtype('M8[us]'))
1245: (8) np.dtype('M8[us]'))
1246: (8) np.dtype('M8[us]'))
1247: (8) np.dtype('M8[us]'))
1248: (8) np.dtype('M8[us]'))
1249: (8) np.dtype('M8[us]'))
1250: (8) np.dtype('M8[us]'))
1251: (8) np.dtype('M8[us]'))
1252: (8) np.dtype('M8[us]'))
1253: (8) np.dtype('M8[us]'))
1254: (8) np.dtype('M8[us]'))
1255: (56) assert_raises(ValueError, np.array, ['1980-02-03%'],
1256: (8) np.dtype('M8[us]'))
1257: (56) assert_raises(ValueError, np.array, ['1980-02-03 q'],
1258: (8) np.dtype('M8[us]'))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1258: (8) assert_raises(ValueError, np.array, ['1980-02-03 25'],
1259: (56)                                     np.dtype('M8[us]'))
1260: (8) assert_raises(ValueError, np.array, ['1980-02-03T25'],
1261: (56)                                     np.dtype('M8[us]'))
1262: (8) assert_raises(ValueError, np.array, ['1980-02-03 24:01'],
1263: (56)                                     np.dtype('M8[us]'))
1264: (8) assert_raises(ValueError, np.array, ['1980-02-03T24:01'],
1265: (56)                                     np.dtype('M8[us]'))
1266: (8) assert_raises(ValueError, np.array, ['1980-02-03 -1'],
1267: (56)                                     np.dtype('M8[us]'))
1268: (8) assert_raises(ValueError, np.array, ['1980-02-03 01:'],
1269: (56)                                     np.dtype('M8[us]'))
1270: (8) assert_raises(ValueError, np.array, ['1980-02-03 01:-1'],
1271: (56)                                     np.dtype('M8[us]'))
1272: (8) assert_raises(ValueError, np.array, ['1980-02-03 01:60'],
1273: (56)                                     np.dtype('M8[us]'))
1274: (8) assert_raises(ValueError, np.array, ['1980-02-03 01:60:'],
1275: (56)                                     np.dtype('M8[us]'))
1276: (8) assert_raises(ValueError, np.array, ['1980-02-03 01:10:-1'],
1277: (56)                                     np.dtype('M8[us]'))
1278: (8) assert_raises(ValueError, np.array, ['1980-02-03 01:01:60'],
1279: (56)                                     np.dtype('M8[us]'))
1280: (8) with assert_warnings(DeprecationWarning):
1281: (12)     assert_raises(ValueError, np.array, ['1980-02-03 01:01:00+0661'],
1282: (60)
np.dtype('M8[us]'))
1283: (8) with assert_warnings(DeprecationWarning):
1284: (12)     assert_raises(ValueError, np.array, ['1980-02-03 01:01:00+2500'],
1285: (60)
np.dtype('M8[us]'))
1286: (8) with assert_warnings(DeprecationWarning):
1287: (12)     assert_raises(ValueError, np.array, ['1980-02-03 01:01:00-0070'],
1288: (60)
np.dtype('M8[us]'))
1289: (8) with assert_warnings(DeprecationWarning):
1290: (12)     assert_raises(ValueError, np.array, ['1980-02-03 01:01:00-3000'],
1291: (60)
np.dtype('M8[us]'))
1292: (8) with assert_warnings(DeprecationWarning):
1293: (12)     assert_raises(ValueError, np.array, ['1980-02-03 01:01:00-25:00'],
1294: (60)
np.dtype('M8[us]'))
1295: (4) def test_creation_overflow(self):
1296: (8)     date = '1980-03-23 20:00:00'
1297: (8)     timesteps = np.array([date], dtype='datetime64[s]')
1298: (8)     for unit in ['ms', 'us', 'ns']:
1299: (12)         timesteps *= 1000
1300: (12)         x = np.array([date], dtype='datetime64[%s]' % unit)
1301: (12)         assert_equal(timesteps, x[0].astype(np.int64),
1302: (25)             err_msg='Datetime conversion error for unit %s' %
unit)
1303: (8)         assert_equal(x[0].astype(np.int64), 3226896000000000000)
1304: (8)         with pytest.raises(OverflowError):
1305: (12)             np.datetime64(2**64, 'D')
1306: (8)             with pytest.raises(OverflowError):
1307: (12)                 np.timedelta64(2**64, 'D')
1308: (4) def test_datetime_as_string(self):
1309: (8)     date = '1959-10-13'
1310: (8)     datetime = '1959-10-13T12:34:56.789012345678901234'
1311: (8)     assert_equal(np.datetime_as_string(np.datetime64(date, 'Y')),
1312: (21)         '1959')
1313: (8)     assert_equal(np.datetime_as_string(np.datetime64(date, 'M')),
1314: (21)         '1959-10')
1315: (8)     assert_equal(np.datetime_as_string(np.datetime64(date, 'D')),
1316: (21)         '1959-10-13')
1317: (8)     assert_equal(np.datetime_as_string(np.datetime64(datetime, 'h')),
1318: (21)         '1959-10-13T12')
1319: (8)     assert_equal(np.datetime_as_string(np.datetime64(datetime, 'm')),
```

```
1320: (21)                                '1959-10-13T12:34')
1321: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 's')),
1322: (21)   '1959-10-13T12:34:56')
1323: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'ms')),
1324: (21)   '1959-10-13T12:34:56.789')
1325: (8)       for us in ['us', 'Î%$', b'us']: # check non-ascii and bytes too
1326: (12)           assert_equal(np.datetime_as_string(np.datetime64(datetime, us)),
1327: (25)   '1959-10-13T12:34:56.789012')
1328: (8)       datetime = '1969-12-31T23:34:56.789012345678901234'
1329: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'ns')),
1330: (21)   '1969-12-31T23:34:56.789012345')
1331: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'ps')),
1332: (21)   '1969-12-31T23:34:56.789012345678')
1333: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'fs')),
1334: (21)   '1969-12-31T23:34:56.789012345678901')
1335: (8)       datetime = '1969-12-31T23:59:57.789012345678901234'
1336: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'as')),
1337: (21)   datetime)
1338: (8)       datetime = '1970-01-01T00:34:56.789012345678901234'
1339: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'ns')),
1340: (21)   '1970-01-01T00:34:56.789012345')
1341: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'ps')),
1342: (21)   '1970-01-01T00:34:56.789012345678')
1343: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'fs')),
1344: (21)   '1970-01-01T00:34:56.789012345678901')
1345: (8)       datetime = '1970-01-01T00:00:05.789012345678901234'
1346: (8)       assert_equal(np.datetime_as_string(np.datetime64(datetime, 'as')),
1347: (21)   datetime)
1348: (8)       a = np.datetime64('2032-07-18T12:23:34.123456', 'us')
1349: (8)       assert_equal(np.datetime_as_string(a, unit='Y', casting='unsafe'),
1350: (28)   '2032')
1351: (8)       assert_equal(np.datetime_as_string(a, unit='M', casting='unsafe'),
1352: (28)   '2032-07')
1353: (8)       assert_equal(np.datetime_as_string(a, unit='W', casting='unsafe'),
1354: (28)   '2032-07-18')
1355: (8)       assert_equal(np.datetime_as_string(a, unit='D', casting='unsafe'),
1356: (28)   '2032-07-18')
1357: (8)       assert_equal(np.datetime_as_string(a, unit='h'), '2032-07-18T12')
1358: (8)       assert_equal(np.datetime_as_string(a, unit='m'),
1359: (28)   '2032-07-18T12:23')
1360: (8)       assert_equal(np.datetime_as_string(a, unit='s'),
1361: (28)   '2032-07-18T12:23:34')
1362: (8)       assert_equal(np.datetime_as_string(a, unit='ms'),
1363: (28)   '2032-07-18T12:23:34.123')
1364: (8)       assert_equal(np.datetime_as_string(a, unit='us'),
1365: (28)   '2032-07-18T12:23:34.123456')
1366: (8)       assert_equal(np.datetime_as_string(a, unit='ns'),
1367: (28)   '2032-07-18T12:23:34.123456000000')
1368: (8)       assert_equal(np.datetime_as_string(a, unit='ps'),
1369: (28)   '2032-07-18T12:23:34.1234560000000000')
1370: (8)       assert_equal(np.datetime_as_string(a, unit='fs'),
1371: (28)   '2032-07-18T12:23:34.123456000000000000')
1372: (8)       assert_equal(np.datetime_as_string(a, unit='as'),
1373: (28)   '2032-07-18T12:23:34.123456000000000000')
1374: (8)       assert_equal(np.datetime_as_string(
1375: (16)   np.datetime64('2032-07-18T12:23:34.123456', 'us'),
unit='auto'),
1376: (16)   '2032-07-18T12:23:34.123456')
1377: (8)       assert_equal(np.datetime_as_string(
1378: (16)   np.datetime64('2032-07-18T12:23:34.12', 'us'), unit='auto'),
1379: (16)   '2032-07-18T12:23:34.120')
1380: (8)       assert_equal(np.datetime_as_string(
1381: (16)   np.datetime64('2032-07-18T12:23:34', 'us'), unit='auto'),
1382: (16)   '2032-07-18T12:23:34')
1383: (8)       assert_equal(np.datetime_as_string(
1384: (16)   np.datetime64('2032-07-18T12:23:00', 'us'), unit='auto'),
1385: (16)   '2032-07-18T12:23')
1386: (8)       assert_equal(np.datetime_as_string(
1387: (16)   np.datetime64('2032-07-18T12:00:00', 'us'), unit='auto'),
```

```

1388: (16)                                '2032-07-18T12:00')
1389: (8)       assert_equal(np.datetime_as_string(
1390: (16)           np.datetime64('2032-07-18T00:00:00', 'us'), unit='auto'),
1391: (16)           '2032-07-18')
1392: (8)       assert_equal(np.datetime_as_string(
1393: (16)           np.datetime64('2032-07-01T00:00:00', 'us'), unit='auto'),
1394: (16)           '2032-07-01')
1395: (8)       assert_equal(np.datetime_as_string(
1396: (16)           np.datetime64('2032-01-01T00:00:00', 'us'), unit='auto'),
1397: (16)           '2032-01-01')
1398: (4) @pytest.mark.skipif(not _has_pytz, reason="The pytz module is not
available.")
1399: (4)
1400: (8)
1401: (8)
1402: (16) def test_datetime_as_string_timezone(self):
1403: (8)     a = np.datetime64('2010-03-15T06:30', 'm')
1404: (8)     assert_equal(np.datetime_as_string(a),
1405: (16)         '2010-03-15T06:30')
1406: (8)     assert_equal(np.datetime_as_string(a, timezone='naive'),
1407: (16)         '2010-03-15T06:30')
1408: (8)     assert_equal(np.datetime_as_string(a, timezone='UTC'),
1409: (16)         '2010-03-15T06:30Z')
1410: (8)     assert_(np.datetime_as_string(a, timezone='local') !=
1411: (21)         '2010-03-15T06:30')
1412: (8)     b = np.datetime64('2010-02-15T06:30', 'm')
1413: (21)     assert_equal(np.datetime_as_string(a, timezone=tz('US/Central')),
1414: (8)         '2010-03-15T01:30-0500')
1415: (21)     assert_equal(np.datetime_as_string(a, timezone=tz('US/Eastern')),
1416: (8)         '2010-03-15T02:30-0400')
1417: (21)     assert_equal(np.datetime_as_string(a, timezone=tz('US/Pacific')),
1418: (8)         '2010-03-14T23:30-0700')
1419: (21)     assert_equal(np.datetime_as_string(b, timezone=tz('US/Central')),
1420: (8)         '2010-02-15T00:30-0600')
1421: (21)     assert_equal(np.datetime_as_string(b, timezone=tz('US/Eastern')),
1422: (8)         '2010-02-15T01:30-0500')
1423: (27)     assert_equal(np.datetime_as_string(b, timezone=tz('US/Pacific')),
1424: (8)         '2010-02-14T22:30-0800')
1425: (27)     assert_raises(TypeError, np.datetime_as_string, a, unit='D',
1426: (21)         timezone=tz('US/Pacific'))
1427: (8)     assert_equal(np.datetime_as_string(a, unit='D',
1428: (27)         timezone=tz('US/Pacific')), casting='unsafe'),
1429: (21)         '2010-02-15')
1430: (4) def test_datetime_arange(self):
1431: (8)     a = np.arange('2010-01-05', '2010-01-10', dtype='M8[D]')
1432: (8)     assert_equal(a.dtype, np.dtype('M8[D]'))
1433: (8)     assert_equal(a,
1434: (12)         np.array(['2010-01-05', '2010-01-06', '2010-01-07',
1435: (22)             '2010-01-08', '2010-01-09']), dtype='M8[D]')
1436: (8)     a = np.arange('1950-02-10', '1950-02-06', -1, dtype='M8[D]')
1437: (8)     assert_equal(a.dtype, np.dtype('M8[D]'))
1438: (8)     assert_equal(a,
1439: (12)         np.array(['1950-02-10', '1950-02-09', '1950-02-08',
1440: (22)             '1950-02-07']), dtype='M8[D]')
1441: (8)     a = np.arange('1969-05', '1970-05', 2, dtype='M8')
1442: (8)     assert_equal(a.dtype, np.dtype('M8[M]'))
1443: (8)     assert_equal(a,
1444: (12)         np.datetime64('1969-05') + np.arange(12, step=2))
1445: (8)     a = np.arange('1969', 18, 3, dtype='M8')
1446: (8)     assert_equal(a.dtype, np.dtype('M8[Y]'))
1447: (8)     assert_equal(a,
1448: (12)         np.datetime64('1969') + np.arange(18, step=3))
1449: (8)     a = np.arange('1969-12-19', 22, np.timedelta64(2), dtype='M8')
1450: (8)     assert_equal(a.dtype, np.dtype('M8[D]'))
1451: (8)     assert_equal(a,
1452: (12)         np.datetime64('1969-12-19') + np.arange(22, step=2))
1453: (8)     assert_raises(ValueError, np.arange, np.datetime64('today'),
1454: (32)             np.datetime64('today') + 3, 0)
1455: (8)     assert_raises(TypeError, np.arange, np.datetime64('2011-03-01', 'D'),

```

```

1456: (32)
1457: (8)
1458: (32)
1459: (32)
1460: (4)
1461: (8)
1462: (8)
1463: (8)
1464: (4)
1465: (8)
1466: (8)
1467: (8)
1468: (8)
1469: (8)
1470: (8)
1471: (8)
1472: (32)
1473: (8)
1474: (32)
1475: (8)
1476: (32)
1477: (4)
1478: (8)
1479: (9)
1480: (9)
1481: (8)
1482: (9)
1483: (9)
1484: (8)
1485: (9)
1486: (9)
1487: (8)
1488: (9)
1489: (9)
1490: (8)
1491: (9)
1492: (9)
1493: (8)
1494: (9)
1495: (9)
1496: (8)
1497: (9)
1498: (9)
1499: (8)
1500: (9)
1501: (9)
1502: (8)
1503: (9)
1504: (9)
1505: (8)
1506: (4)
1507: (8)
1508: (4)
1509: (8)
1510: (9)
1511: (8)
1512: (9)
1513: (8)
1514: (4)
1515: (8)
1516: (12)
1517: (4)
1518: (4)
1519: (8)
1520: (12)
1521: (12)
1522: (4)
1523: (8)
1524: (9)

        np.timedelta64(5, 'M'))
    assert_raises(TypeError, np.arange,
                  np.datetime64('2012-02-03T14', 's'),
                  np.timedelta64(5, 'Y'))
def test_datetime_arange_no_dtype(self):
    d = np.array('2010-01-04', dtype="M8[D]")
    assert_equal(np.arange(d, d + 1), d)
    assert_raises(ValueError, np.arange, d)
def test_timedelta_arange(self):
    a = np.arange(3, 10, dtype='m8')
    assert_equal(a.dtype, np.dtype('m8'))
    assert_equal(a, np.timedelta64(0) + np.arange(3, 10))
    a = np.arange(np.timedelta64(3, 's'), 10, 2, dtype='m8')
    assert_equal(a.dtype, np.dtype('m8[s]'))
    assert_equal(a, np.timedelta64(0, 's') + np.arange(3, 10, 2))
    assert_raises(ValueError, np.arange, np.timedelta64(0,
   np.timedelta64(5), 0))
    assert_raises(TypeError, np.arange, np.timedelta64(0, 'D'),
                  np.timedelta64(5, 'M'))
    assert_raises(TypeError, np.arange, np.timedelta64(0, 'Y'),
                  np.timedelta64(5, 'D'))
@pytest.mark.parametrize("val1, val2, expected", [
    (np.timedelta64(7, 's'),
     np.timedelta64(3, 's'),
     np.timedelta64(1, 's')),
    (np.timedelta64(3, 's'),
     np.timedelta64(-2, 's'),
     np.timedelta64(-1, 's')),
    (np.timedelta64(-3, 's'),
     np.timedelta64(2, 's'),
     np.timedelta64(1, 's')),
    (np.timedelta64(17, 's'),
     np.timedelta64(22, 's'),
     np.timedelta64(17, 's')),
    (np.timedelta64(22, 's'),
     np.timedelta64(17, 's'),
     np.timedelta64(5, 's')),
    (np.timedelta64(1, 'm'),
     np.timedelta64(57, 's'),
     np.timedelta64(3, 's')),
    (np.timedelta64(1, 'us'),
     np.timedelta64(727, 'ns'),
     np.timedelta64(273, 'ns')),
    (np.timedelta64('NaT'),
     np.timedelta64(50, 'ns'),
     np.timedelta64('NaT')),
    (np.timedelta64(2, 'Y'),
     np.timedelta64(22, 'M'),
     np.timedelta64(2, 'M')),
    ])
]
def test_timedelta_modulus(self, val1, val2, expected):
    assert_equal(val1 % val2, expected)
@pytest.mark.parametrize("val1, val2", [
    (np.timedelta64(7, 'Y'),
     np.timedelta64(3, 's')),
    (np.timedelta64(7, 'M'),
     np.timedelta64(1, 'D')),
])
def test_timedelta_modulus_error(self, val1, val2):
    with assert_raises_regex(TypeError, "common metadata divisor"):
        val1 % val2
@pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
def test_timedelta_modulus_div_by_zero(self):
    with assert_warns(RuntimeWarning):
        actual = np.timedelta64(10, 's') % np.timedelta64(0, 's')
        assert_equal(actual, np.timedelta64('NaT'))
@pytest.mark.parametrize("val1, val2", [
    (np.timedelta64(7, 'Y'),
     15,),
])

```

```

1525: (8)           (7.5,
1526: (9)             np.timedelta64(1, 'D')),
1527: (8)           ])
1528: (4)     def test_timedelta_modulus_type_resolution(self, val1, val2):
1529: (8)       with assert_raises_regex(TypeError,
1530: (33)                     "'remainder' cannot use operands with
types"):
1531: (12)           val1 % val2
1532: (4)     def test_timedelta_arange_no_dtype(self):
1533: (8)       d = np.array(5, dtype="m8[D]")
1534: (8)       assert_equal(np.arange(d, d + 1), d)
1535: (8)       assert_equal(np.arange(d), np.arange(0, d))
1536: (4)     def test_datetime_maximum_reduce(self):
1537: (8)       a = np.array(['2010-01-02', '1999-03-14', '1833-03'], dtype='M8[D]')
1538: (8)       assert_equal(np.maximum.reduce(a).dtype, np.dtype('M8[D]'))
1539: (8)       assert_equal(np.maximum.reduce(a),
1540: (21)                     np.datetime64('2010-01-02'))
1541: (8)       a = np.array([1, 4, 0, 7, 2], dtype='m8[s]')
1542: (8)       assert_equal(np.maximum.reduce(a).dtype, np.dtype('m8[s]'))
1543: (8)       assert_equal(np.maximum.reduce(a),
1544: (21)                     np.timedelta64(7, 's'))
1545: (4)     def test_timedelta_correct_mean(self):
1546: (8)       a = np.arange(1000, dtype="m8[s]")
1547: (8)       assert_array_equal(a.mean(), a.sum() / len(a))
1548: (4)     def test_datetime_no_subtract_reducelike(self):
1549: (8)       arr = np.array(["2021-12-02", "2019-05-12"], dtype="M8[ms]")
1550: (8)       msg = r"the resolved dtypes are not compatible"
1551: (8)       with pytest.raises(TypeError, match=msg):
1552: (12)           np.subtract.reduce(arr)
1553: (8)       with pytest.raises(TypeError, match=msg):
1554: (12)           np.subtract.accumulate(arr)
1555: (8)       with pytest.raises(TypeError, match=msg):
1556: (12)           np.subtract.reduceat(arr, [0])
1557: (4)     def test_datetime_busday_offset(self):
1558: (8)       assert_equal(
1559: (12)           np.busday_offset('2011-06', 0, roll='forward', weekmask='Mon'),
1560: (12)           np.datetime64('2011-06-06'))
1561: (8)       assert_equal(
1562: (12)           np.busday_offset('2011-07', -1, roll='forward', weekmask='Mon'),
1563: (12)           np.datetime64('2011-06-27'))
1564: (8)       assert_equal(
1565: (12)           np.busday_offset('2011-07', -1, roll='forward', weekmask='Mon'),
1566: (12)           np.datetime64('2011-06-27'))
1567: (8)       assert_equal(np.busday_offset('2010-08', 0, roll='backward'),
1568: (21)                     np.datetime64('2010-07-30'))
1569: (8)       assert_equal(np.busday_offset('2010-08', 0, roll='preceding'),
1570: (21)                     np.datetime64('2010-07-30'))
1571: (8)       assert_equal(np.busday_offset('2010-08', 0, roll='modifiedpreceding'),
1572: (21)                     np.datetime64('2010-08-02'))
1573: (8)       assert_equal(np.busday_offset('2010-08', 0, roll='modifiedfollowing'),
1574: (21)                     np.datetime64('2010-08-02'))
1575: (8)       assert_equal(np.busday_offset('2010-08', 0, roll='forward'),
1576: (21)                     np.datetime64('2010-08-02'))
1577: (8)       assert_equal(np.busday_offset('2010-08', 0, roll='following'),
1578: (21)                     np.datetime64('2010-08-02'))
1579: (8)       assert_equal(np.busday_offset('2010-10-30', 0, roll='following'),
1580: (21)                     np.datetime64('2010-11-01'))
1581: (8)       assert_equal(
1582: (16)           np.busday_offset('2010-10-30', 0, roll='modifiedfollowing'),
1583: (16)           np.datetime64('2010-10-29'))
1584: (8)       assert_equal(
1585: (16)           np.busday_offset('2010-10-30', 0, roll='modifiedpreceding'),
1586: (16)           np.datetime64('2010-10-29'))
1587: (8)       assert_equal(
1588: (16)           np.busday_offset('2010-10-16', 0, roll='modifiedfollowing'),
1589: (16)           np.datetime64('2010-10-18'))
1590: (8)       assert_equal(
1591: (16)           np.busday_offset('2010-10-16', 0, roll='modifiedpreceding'),
1592: (16)           np.datetime64('2010-10-15'))

```

```

1593: (8) assert_raises(ValueError, np.busday_offset, '2011-06-04', 0)
1594: (8) assert_equal(np.busday_offset('2006-02-01', 25),
1595: (21)           np.datetime64('2006-03-08'))
1596: (8) assert_equal(np.busday_offset('2006-03-08', -25),
1597: (21)           np.datetime64('2006-02-01'))
1598: (8) assert_equal(np.busday_offset('2007-02-25', 11, weekmask='SatSun'),
1599: (21)           np.datetime64('2007-04-07'))
1600: (8) assert_equal(np.busday_offset('2007-04-07', -11, weekmask='SatSun'),
1601: (21)           np.datetime64('2007-02-25'))
1602: (8) assert_equal(np.busday_offset(np.datetime64('NaT'), 1, roll='nat'),
1603: (21)           np.datetime64('NaT'))
1604: (8) assert_equal(np.busday_offset(np.datetime64('NaT'), 1,
1605: (21)             roll='following'),
1606: (8)             np.datetime64('NaT'))
1607: (21)             assert_equal(np.busday_offset(np.datetime64('NaT'), 1,
1608: (4)               np.datetime64('NaT'))
1609: (8) def test_datetime_busdaycalendar(self):
1610: (12)     bdd = np.busdaycalendar(
1611: (23)       holidays=['NaT', '2011-01-17', '2011-03-06', 'NaT',
1612: (8)           '2011-12-26', '2011-05-30', '2011-01-17'])
1613: (12)     assert_equal(bdd.holidays,
1614: (8)       np.array(['2011-01-17', '2011-05-30', '2011-12-26'], dtype='M8'))
1615: (8)     assert_equal(bdd.weekmask, np.array([1, 1, 1, 1, 1, 0, 0], dtype='?'))
1616: (8)     bdd = np.busdaycalendar(weekmask="Sun TueWed Thu\tFri")
1617: (8)     assert_equal(bdd.weekmask, np.array([0, 1, 1, 1, 1, 0, 1], dtype='?'))
1618: (8)     bdd = np.busdaycalendar(weekmask="0011001")
1619: (8)     assert_equal(bdd.weekmask, np.array([0, 0, 1, 1, 0, 0, 1], dtype='?'))
1620: (8)     bdd = np.busdaycalendar(weekmask="Mon Tue")
1621: (8)     assert_equal(bdd.weekmask, np.array([1, 1, 0, 0, 0, 0, 0], dtype='?'))
1622: (8)     assert_raises(ValueError, np.busdaycalendar, weekmask=[0, 0, 0, 0, 0,
1623: (8)             0, 0])
1624: (8)     assert_raises(ValueError, np.busdaycalendar, weekmask="")
1625: (8)     assert_raises(ValueError, np.busdaycalendar, weekmask="Mon Tue We")
1626: (8)     assert_raises(ValueError, np.busdaycalendar, weekmask="Max")
1627: (4)     assert_raises(ValueError, np.busdaycalendar, weekmask="Monday Tue")
1628: (8) def test_datetime_busday_holidays_offset(self):
1629: (12)     assert_equal(
1630: (12)       np.busday_offset('2011-11-10', 1, holidays=['2011-11-11']),
1631: (8)       np.datetime64('2011-11-14'))
1632: (12)     assert_equal(
1633: (12)       np.busday_offset('2011-11-04', 5, holidays=['2011-11-11']),
1634: (8)       np.datetime64('2011-11-14'))
1635: (12)     assert_equal(
1636: (12)       np.busday_offset('2011-11-10', 5, holidays=['2011-11-11']),
1637: (8)       np.datetime64('2011-11-18'))
1638: (12)     assert_equal(
1639: (12)       np.busday_offset('2011-11-14', -1, holidays=['2011-11-11']),
1640: (8)       np.datetime64('2011-11-10'))
1641: (12)     assert_equal(
1642: (12)       np.busday_offset('2011-11-18', -5, holidays=['2011-11-11']),
1643: (8)       np.datetime64('2011-11-10'))
1644: (12)     assert_equal(
1645: (12)       np.busday_offset('2011-11-14', -5, holidays=['2011-11-11']),
1646: (8)       np.datetime64('2011-11-04'))
1647: (12)     assert_equal(
1648: (16)       np.busday_offset('2011-11-10', 1,
1649: (12)           holidays=['2011-11-11', '2011-11-11']),
1650: (8)           np.datetime64('2011-11-14'))
1651: (12)     assert_equal(
1652: (16)       np.busday_offset('2011-11-14', -1,
1653: (12)           holidays=['2011-11-11', '2011-11-11']),
1654: (8)           np.datetime64('2011-11-10'))
1655: (12)     assert_equal(
1656: (16)       np.busday_offset('2011-11-10', 1,
1657: (12)           holidays=['2011-11-11', 'NaT']),
1658: (8)           np.datetime64('2011-11-14'))
1659: (8)     assert_equal(

```

```

1659: (12) np.busday_offset('2011-11-14', -1,
1660: (16)     holidays=['NaT', '2011-11-11']),
1661: (12)     np.datetime64('2011-11-10'))
1662: (8) assert_equal(
1663: (12)     np.busday_offset('2011-11-10', 1,
1664: (16)     holidays=['2011-11-11', '2011-11-24']),
1665: (12)     np.datetime64('2011-11-14'))
1666: (8) assert_equal(
1667: (12)     np.busday_offset('2011-11-14', -1,
1668: (16)     holidays=['2011-11-11', '2011-11-24']),
1669: (12)     np.datetime64('2011-11-10'))
1670: (8) assert_equal(
1671: (12)     np.busday_offset('2011-11-10', 1,
1672: (16)     holidays=['2011-10-10', '2011-11-11']),
1673: (12)     np.datetime64('2011-11-14'))
1674: (8) assert_equal(
1675: (12)     np.busday_offset('2011-11-14', -1,
1676: (16)     holidays=['2011-10-10', '2011-11-11']),
1677: (12)     np.datetime64('2011-11-10'))
1678: (8) assert_equal(
1679: (12)     np.busday_offset('2011-11-10', 1,
1680: (16)     holidays=['2011-10-10', '2011-11-11', '2011-11-24']),
1681: (12)     np.datetime64('2011-11-14'))
1682: (8) assert_equal(
1683: (12)     np.busday_offset('2011-11-14', -1,
1684: (16)     holidays=['2011-10-10', '2011-11-11', '2011-11-24']),
1685: (12)     np.datetime64('2011-11-10'))
1686: (8) holidays = ['2011-10-10', '2011-11-11', '2011-11-24',
1687: (18)             '2011-12-25', '2011-05-30', '2011-02-21',
1688: (18)             '2011-12-26', '2012-01-02']
1689: (8) bdd = np.busdaycalendar(weekmask='1111100', holidays=holidays)
1690: (8) assert_equal(
1691: (12)     np.busday_offset('2011-10-03', 4, holidays=holidays),
1692: (12)     np.busday_offset('2011-10-03', 4))
1693: (8) assert_equal(
1694: (12)     np.busday_offset('2011-10-03', 5, holidays=holidays),
1695: (12)     np.busday_offset('2011-10-03', 5 + 1))
1696: (8) assert_equal(
1697: (12)     np.busday_offset('2011-10-03', 27, holidays=holidays),
1698: (12)     np.busday_offset('2011-10-03', 27 + 1))
1699: (8) assert_equal(
1700: (12)     np.busday_offset('2011-10-03', 28, holidays=holidays),
1701: (12)     np.busday_offset('2011-10-03', 28 + 2))
1702: (8) assert_equal(
1703: (12)     np.busday_offset('2011-10-03', 35, holidays=holidays),
1704: (12)     np.busday_offset('2011-10-03', 35 + 2))
1705: (8) assert_equal(
1706: (12)     np.busday_offset('2011-10-03', 36, holidays=holidays),
1707: (12)     np.busday_offset('2011-10-03', 36 + 3))
1708: (8) assert_equal(
1709: (12)     np.busday_offset('2011-10-03', 56, holidays=holidays),
1710: (12)     np.busday_offset('2011-10-03', 56 + 3))
1711: (8) assert_equal(
1712: (12)     np.busday_offset('2011-10-03', 57, holidays=holidays),
1713: (12)     np.busday_offset('2011-10-03', 57 + 4))
1714: (8) assert_equal(
1715: (12)     np.busday_offset('2011-10-03', 60, holidays=holidays),
1716: (12)     np.busday_offset('2011-10-03', 60 + 4))
1717: (8) assert_equal(
1718: (12)     np.busday_offset('2011-10-03', 61, holidays=holidays),
1719: (12)     np.busday_offset('2011-10-03', 61 + 5))
1720: (8) assert_equal(
1721: (12)     np.busday_offset('2011-10-03', 61, busdaycal=bdd),
1722: (12)     np.busday_offset('2011-10-03', 61 + 5))
1723: (8) assert_equal(
1724: (12)     np.busday_offset('2012-01-03', -1, holidays=holidays),
1725: (12)     np.busday_offset('2012-01-03', -1 - 1))
1726: (8) assert_equal(
1727: (12)     np.busday_offset('2012-01-03', -4, holidays=holidays),

```

```

1728: (12) np.busday_offset('2012-01-03', -4 - 1))
1729: (8) assert_equal(
1730: (12)     np.busday_offset('2012-01-03', -5, holidays=holidays),
1731: (12)     np.busday_offset('2012-01-03', -5 - 2))
1732: (8) assert_equal(
1733: (12)     np.busday_offset('2012-01-03', -25, holidays=holidays),
1734: (12)     np.busday_offset('2012-01-03', -25 - 2))
1735: (8) assert_equal(
1736: (12)     np.busday_offset('2012-01-03', -26, holidays=holidays),
1737: (12)     np.busday_offset('2012-01-03', -26 - 3))
1738: (8) assert_equal(
1739: (12)     np.busday_offset('2012-01-03', -33, holidays=holidays),
1740: (12)     np.busday_offset('2012-01-03', -33 - 3))
1741: (8) assert_equal(
1742: (12)     np.busday_offset('2012-01-03', -34, holidays=holidays),
1743: (12)     np.busday_offset('2012-01-03', -34 - 4))
1744: (8) assert_equal(
1745: (12)     np.busday_offset('2012-01-03', -56, holidays=holidays),
1746: (12)     np.busday_offset('2012-01-03', -56 - 4))
1747: (8) assert_equal(
1748: (12)     np.busday_offset('2012-01-03', -57, holidays=holidays),
1749: (12)     np.busday_offset('2012-01-03', -57 - 5))
1750: (8) assert_equal(
1751: (12)     np.busday_offset('2012-01-03', -57, busdaycal=bdd),
1752: (12)     np.busday_offset('2012-01-03', -57 - 5))
1753: (8) assert_raises(ValueError, np.busday_offset, '2012-01-03', -15,
1754: (24)             weekmask='1111100', busdaycal=bdd)
1755: (8) assert_raises(ValueError, np.busday_offset, '2012-01-03', -15,
1756: (24)             holidays=holidays, busdaycal=bdd)
1757: (8) assert_equal(
1758: (12)     np.busday_offset('2011-12-25', 0,
1759: (16)         roll='forward', holidays=holidays),
1760: (12)     np.datetime64('2011-12-27'))
1761: (8) assert_equal(
1762: (12)     np.busday_offset('2011-12-26', 0,
1763: (16)         roll='forward', holidays=holidays),
1764: (12)     np.datetime64('2011-12-27'))
1765: (8) assert_equal(
1766: (12)     np.busday_offset('2011-12-26', 0,
1767: (16)         roll='backward', holidays=holidays),
1768: (12)     np.datetime64('2011-12-23'))
1769: (8) assert_equal(
1770: (12)     np.busday_offset('2012-02-27', 0,
1771: (16)         roll='modifiedfollowing',
1772: (16)         holidays=['2012-02-27', '2012-02-26', '2012-02-28',
1773: (26)             '2012-03-01', '2012-02-29']),
1774: (12)     np.datetime64('2012-02-24'))
1775: (8) assert_equal(
1776: (12)     np.busday_offset('2012-03-06', 0,
1777: (16)         roll='modifiedpreceding',
1778: (16)         holidays=['2012-03-02', '2012-03-03', '2012-03-01',
1779: (26)             '2012-03-05', '2012-03-07', '2012-03-06']),
1780: (12)     np.datetime64('2012-03-08'))
1781: (4) def test_datetime_busday_holidays_count(self):
1782: (8)     holidays = ['2011-01-01', '2011-10-10', '2011-11-11', '2011-11-24',
1783: (20)         '2011-12-25', '2011-05-30', '2011-02-21', '2011-01-17',
1784: (20)         '2011-12-26', '2012-01-02', '2011-02-21', '2011-05-30',
1785: (20)         '2011-07-01', '2011-07-04', '2011-09-05', '2011-10-10']
1786: (8)     bdd = np.busdaycalendar(weekmask='1111100', holidays=holidays)
1787: (8)     dates = np.busday_offset('2011-01-01', np.arange(366),
1788: (24)         roll='forward', busdaycal=bdd)
1789: (8)     assert_equal(np.busday_count('2011-01-01', dates, busdaycal=bdd),
1790: (21)             np.arange(366))
1791: (8)     assert_equal(np.busday_count(dates, '2011-01-01', busdaycal=bdd),
1792: (21)             -np.arange(366) - 1)
1793: (8)     dates = np.busday_offset('2011-12-31', -np.arange(366),
1794: (24)         roll='forward', busdaycal=bdd)
1795: (8)     expected = np.arange(366)
1796: (8)     expected[0] = -1

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1797: (8) assert_equal(np.busday_count(dates, '2011-12-31', busdaycal=bdd),
1798: (21)           expected)
1799: (8)           expected = -np.arange(366)+1
1800: (8)           expected[0] = 0
1801: (8)           assert_equal(np.busday_count('2011-12-31', dates, busdaycal=bdd),
1802: (21)           expected)
1803: (8)           assert_raises(ValueError, np.busday_offset, '2012-01-03', '2012-02-
03',
1804: (24)           weekmask='1111100', busdaycal=bdd)
1805: (8)           assert_raises(ValueError, np.busday_offset, '2012-01-03', '2012-02-
03',
1806: (24)           holidays=holidays, busdaycal=bdd)
1807: (8)           assert_equal(np.busday_count('2011-03', '2011-04', weekmask='Mon'), 4)
1808: (8)           assert_equal(np.busday_count('2011-04', '2011-03', weekmask='Mon'),
-4)
1809: (8)           sunday = np.datetime64('2023-03-05')
1810: (8)           monday = sunday + 1
1811: (8)           friday = sunday + 5
1812: (8)           saturday = sunday + 6
1813: (8)           assert_equal(np.busday_count(sunday, monday), 0)
1814: (8)           assert_equal(np.busday_count(monday, sunday), -1)
1815: (8)           assert_equal(np.busday_count(friday, saturday), 1)
1816: (8)           assert_equal(np.busday_count(saturday, friday), 0)
1817: (4) def test_datetime_is_busday(self):
1818: (8)     holidays = ['2011-01-01', '2011-10-10', '2011-11-11', '2011-11-24',
1819: (20)           '2011-12-25', '2011-05-30', '2011-02-21', '2011-01-17',
1820: (20)           '2011-12-26', '2012-01-02', '2011-02-21', '2011-05-30',
1821: (20)           '2011-07-01', '2011-07-04', '2011-09-05', '2011-10-10',
1822: (20)           'NaT']
1823: (8)     bdd = np.busdaycalendar(weekmask='1111100', holidays=holidays)
1824: (8)     assert_equal(np.is_busday('2011-01-01'), False)
1825: (8)     assert_equal(np.is_busday('2011-01-02'), False)
1826: (8)     assert_equal(np.is_busday('2011-01-03'), True)
1827: (8)     assert_equal(np.is_busday(holidays, busdaycal=bdd),
1828: (21)           np.zeros(len(holidays), dtype='?'))
1829: (4) def test_datetime_y2038(self):
1830: (8)     a = np.datetime64('2038-01-19T03:14:07')
1831: (8)     assert_equal(a.view(np.int64), 2**31 - 1)
1832: (8)     a = np.datetime64('2038-01-19T03:14:08')
1833: (8)     assert_equal(a.view(np.int64), 2**31)
1834: (8)     with assert_warnings(DeprecationWarning):
1835: (12)         a = np.datetime64('2038-01-19T04:14:07+0100')
1836: (12)         assert_equal(a.view(np.int64), 2**31 - 1)
1837: (8)     with assert_warnings(DeprecationWarning):
1838: (12)         a = np.datetime64('2038-01-19T04:14:08+0100')
1839: (12)         assert_equal(a.view(np.int64), 2**31)
1840: (8)     a = np.datetime64('2038-01-20T13:21:14')
1841: (8)     assert_equal(str(a), '2038-01-20T13:21:14')
1842: (4) def test_isnat(self):
1843: (8)     assert_(np.isnat(np.datetime64('NaT', 'ms')))
1844: (8)     assert_(np.isnat(np.datetime64('NaT', 'ns')))
1845: (8)     assert_(not np.isnat(np.datetime64('2038-01-19T03:14:07')))
1846: (8)     assert_(np.isnat(np.timedelta64('NaT', "ms")))
1847: (8)     assert_(not np.isnat(np.timedelta64(34, "ms")))
1848: (8)     res = np.array([False, False, True])
1849: (8)     for unit in ['Y', 'M', 'W', 'D',
1850: (21)           'h', 'm', 's', 'ms', 'us',
1851: (21)           'ns', 'ps', 'fs', 'as']:
1852: (12)         arr = np.array([123, -321, "NaT"], dtype='<datetime64[%s]' % unit)
1853: (12)         assert_equal(np.isnat(arr), res)
1854: (12)         arr = np.array([123, -321, "NaT"], dtype='>datetime64[%s]' % unit)
1855: (12)         assert_equal(np.isnat(arr), res)
1856: (12)         arr = np.array([123, -321, "NaT"], dtype='<timedelta64[%s]' %
unit)
1857: (12)         assert_equal(np.isnat(arr), res)
1858: (12)         arr = np.array([123, -321, "NaT"], dtype='>timedelta64[%s]' %
unit)
1859: (12)         assert_equal(np.isnat(arr), res)
1860: (4) def test_isnat_error(self):

```

```

1861: (8)           for t in np.typecodes["All"]:
1862: (12)          if t in np.typecodes["Datetime"]:
1863: (16)          continue
1864: (12)          assert_raises(TypeError, np.isnat, np.zeros(10, t))
1865: (4)           def test_isfinite_scalar(self):
1866: (8)             assert_(not np.isfinite(np.datetime64('NaT', 'ms')))
1867: (8)             assert_(not np.isfinite(np.datetime64('NaT', 'ns')))
1868: (8)             assert_(np.isfinite(np.datetime64('2038-01-19T03:14:07')))
1869: (8)             assert_(not np.isfinite(np.timedelta64('NaT', "ms")))
1870: (8)             assert_(np.isfinite(np.timedelta64(34, "ms")))
1871: (4)           @pytest.mark.parametrize('unit', ['Y', 'M', 'W', 'D', 'h', 'm', 's', 'ms',
1872: (38)                  'us', 'ns', 'ps', 'fs', 'as'])
1873: (4)           @pytest.mark.parametrize('dstr', ['<datetime64[%s]', '>datetime64[%s]',
1874: (38)                  '<timedelta64[%s]', '>timedelta64[%s]'])
1875: (4)           def test_isfinite_isinf_isnan_units(self, unit, dstr):
1876: (8)             '''check isfinite, isinf, isnan for all units of <M, >M, <m, >m dtypes
1877: (8)             '''
1878: (8)             arr_val = [123, -321, "NaT"]
1879: (8)             arr = np.array(arr_val, dtype= dstr % unit)
1880: (8)             pos = np.array([True, True, False])
1881: (8)             neg = np.array([False, False, True])
1882: (8)             false = np.array([False, False, False])
1883: (8)             assert_equal(np.isfinite(arr), pos)
1884: (8)             assert_equal(np.isinf(arr), false)
1885: (8)             assert_equal(np.isnan(arr), neg)
1886: (4)           def test_assert_equal(self):
1887: (8)             assert_raises(AssertionError, assert_equal,
1888: (16)                   np.datetime64('nat'), np.timedelta64('nat'))
1889: (4)           def test_corecursive_input(self):
1890: (8)             a, b = [], []
1891: (8)             a.append(b)
1892: (8)             b.append(a)
1893: (8)             obj_arr = np.array([None])
1894: (8)             obj_arr[0] = a
1895: (8)             assert_raises(ValueError, obj_arr.astype, 'M8')
1896: (8)             assert_raises(ValueError, obj_arr.astype, 'm8')
1897: (4)           @pytest.mark.parametrize("shape", [(), (1,)])
1898: (4)           def test_discovery_from_object_array(self, shape):
1899: (8)             arr = np.array("2020-10-10", dtype=object).reshape(shape)
2000: (8)             res = np.array("2020-10-10", dtype="M8").reshape(shape)
2001: (8)             assert res.dtype == np.dtype("M8[D]")
2002: (8)             assert_equal(arr.astype("M8"), res)
2003: (8)             arr[...] = np.bytes_("2020-10-10") # try a numpy string type
2004: (8)             assert_equal(arr.astype("M8"), res)
2005: (8)             arr = arr.astype("S")
2006: (8)             assert_equal(arr.astype("S").astype("M8"), res)
2007: (4)           @pytest.mark.parametrize("time_unit", [
2008: (8)             "Y", "M", "W", "D", "h", "m", "s", "ms", "us", "ns", "ps", "fs", "as",
2009: (8)             "10D", "2M",
2010: (4)         ])
2011: (4)           def test_limit_symmetry(self, time_unit):
2012: (8)             """
2013: (8)               Dates should have symmetric limits around the unix epoch at +/- ...
2014: (8)             """
2015: (8)             epoch = np.datetime64(0, time_unit)
2016: (8)             latest = np.datetime64(np.iinfo(np.int64).max, time_unit)
2017: (8)             earliest = np.datetime64(-np.iinfo(np.int64).max, time_unit)
2018: (8)             assert earliest < epoch < latest
2019: (4)           @pytest.mark.parametrize("time_unit", [
2020: (8)             "Y", "M",
2021: (8)             pytest.param("W", marks=pytest.mark.xfail(reason="gh-13197")),
2022: (8)             "D", "h", "m",
2023: (8)             "s", "ms", "us", "ns", "ps", "fs", "as",
2024: (8)             pytest.param("10D", marks=pytest.mark.xfail(reason="similar to gh-
2025: (4)             13197")),
2026: (4)         ])
2027: (4)           @pytest.mark.parametrize("sign", [-1, 1])
2028: (4)           def test_limit_str_roundtrip(self, time_unit, sign):

```

```

1928: (8)           """
1929: (8)           Limits should roundtrip when converted to strings.
1930: (8)           This tests the conversion to and from npy_datetimestruct.
1931: (8)           """
1932: (8)           limit = np.datetime64(np.iinfo(np.int64).max * sign, time_unit)
1933: (8)           limit_via_str = np.datetime64(str(limit), time_unit)
1934: (8)           assert limit_via_str == limit
1935: (0)            class TestDateTimeData:
1936: (4)              def test_basic(self):
1937: (8)                  a = np.array(['1980-03-23'], dtype=np.datetime64)
1938: (8)                  assert_equal(np.datetime_data(a.dtype), ('D', 1))
1939: (4)              def test_bytes(self):
1940: (8)                  dt = np.datetime64('2000', ('b' 'ms', 5))
1941: (8)                  assert np.datetime_data(dt.dtype) == ('ms', 5)
1942: (8)                  dt = np.datetime64('2000', b'5ms')
1943: (8)                  assert np.datetime_data(dt.dtype) == ('ms', 5)
1944: (4)              def test_non_ascii(self):
1945: (8)                  dt = np.datetime64('2000', ('Î% s', 5))
1946: (8)                  assert np.datetime_data(dt.dtype) == ('us', 5)
1947: (8)                  dt = np.datetime64('2000', '5Î% s')
1948: (8)                  assert np.datetime_data(dt.dtype) == ('us', 5)
1949: (0)            def test_comparisons_return_not_implemented():
1950: (4)              class custom:
1951: (8)                  __array_priority__ = 10000
1952: (4)                  obj = custom()
1953: (4)                  dt = np.datetime64('2000', 'ns')
1954: (4)                  td = dt - dt
1955: (4)                  for item in [dt, td]:
1956: (8)                      assert item.__eq__(obj) is NotImplemented
1957: (8)                      assert item.__ne__(obj) is NotImplemented
1958: (8)                      assert item.__le__(obj) is NotImplemented
1959: (8)                      assert item.__lt__(obj) is NotImplemented
1960: (8)                      assert item.__ge__(obj) is NotImplemented
1961: (8)                      assert item.__gt__(obj) is NotImplemented

```

---

## File 88 - test\_defchararray.py:

```

1: (0)          import pytest
2: (0)          import numpy as np
3: (0)          from numpy.core.multiarray import _vec_string
4: (0)          from numpy.testing import (
5: (4)              assert_, assert_equal, assert_array_equal, assert_raises,
6: (4)              assert_raises_regex
7: (4)          )
8: (0)          kw_unicode_true = {'unicode': True} # make 2to3 work properly
9: (0)          kw_unicode_false = {'unicode': False}
10: (0)         class TestBasic:
11: (4)             def test_from_object_array(self):
12: (8)                 A = np.array([['abc', 2],
13: (22)                     ['long ', '0123456789']], dtype='O')
14: (8)                 B = np.char.array(A)
15: (8)                 assert_equal(B.dtype.itemsize, 10)
16: (8)                 assert_array_equal(B, [[b'abc', b'2'],
17: (31)                     [b'long', b'0123456789']])
18: (4)             def test_from_object_array_unicode(self):
19: (8)                 A = np.array([['abc', 'Sigma \u03a3'],
20: (22)                     ['long ', '0123456789']], dtype='O')
21: (8)                 assert_raises(ValueError, np.char.array, (A,))
22: (8)                 B = np.char.array(A, **kw_unicode_true)
23: (8)                 assert_equal(B.dtype.itemsize, 10 * np.array('a', 'U').dtype.itemsize)
24: (8)                 assert_array_equal(B, [[b'abc', 'Sigma \u03a3'],
25: (31)                     [b'long', '0123456789']])
26: (4)             def test_from_string_array(self):
27: (8)                 A = np.array([[b'abc', b'foo'],
28: (22)                     [b'long ', b'0123456789']])
29: (8)                 assert_equal(A.dtype.type, np.bytes_)
30: (8)                 B = np.char.array(A)

```

```

31: (8)             assert_array_equal(B, A)
32: (8)             assert_equal(B.dtype, A.dtype)
33: (8)             assert_equal(B.shape, A.shape)
34: (8)             B[0, 0] = 'changed'
35: (8)             assert_(B[0, 0] != A[0, 0])
36: (8)             C = np.char.asarray(A)
37: (8)             assert_array_equal(C, A)
38: (8)             assert_equal(C.dtype, A.dtype)
39: (8)             C[0, 0] = 'changed again'
40: (8)             assert_(C[0, 0] != B[0, 0])
41: (8)             assert_(C[0, 0] == A[0, 0])
42: (4)             def test_from_unicode_array(self):
43: (8)                 A = np.array([['abc', 'Sigma \u03a3'],
44: (22)                           ['long ', '0123456789']])
45: (8)                 assert_equal(A.dtype.type, np.str_)
46: (8)                 B = np.char.array(A)
47: (8)                 assert_array_equal(B, A)
48: (8)                 assert_equal(B.dtype, A.dtype)
49: (8)                 assert_equal(B.shape, A.shape)
50: (8)                 B = np.char.array(A, **kw_unicode_true)
51: (8)                 assert_array_equal(B, A)
52: (8)                 assert_equal(B.dtype, A.dtype)
53: (8)                 assert_equal(B.shape, A.shape)
54: (8)             def fail():
55: (12)                 np.char.array(A, **kw_unicode_false)
56: (8)                 assert_raises(UnicodeEncodeError, fail)
57: (4)             def test_unicode_upconvert(self):
58: (8)                 A = np.char.array(['abc'])
59: (8)                 B = np.char.array(['\u03a3'])
60: (8)                 assert_(issubclass((A + B).dtype.type, np.str_))
61: (4)             def test_from_string(self):
62: (8)                 A = np.char.array(b'abc')
63: (8)                 assert_equal(len(A), 1)
64: (8)                 assert_equal(len(A[0]), 3)
65: (8)                 assert_(issubclass(A.dtype.type, np.bytes_))
66: (4)             def test_from_unicode(self):
67: (8)                 A = np.char.array('\u03a3')
68: (8)                 assert_equal(len(A), 1)
69: (8)                 assert_equal(len(A[0]), 1)
70: (8)                 assert_equal(A.itemsize, 4)
71: (8)                 assert_(issubclass(A.dtype.type, np.str_))
72: (0)             class TestVecString:
73: (4)                 def test_non_existent_method(self):
74: (8)                     def fail():
75: (12)                         _vec_string('a', np.bytes_, 'bogus')
76: (8)                         assert_raises(AttributeError, fail)
77: (4)                 def test_non_string_array(self):
78: (8)                     def fail():
79: (12)                         _vec_string(1, np.bytes_, 'strip')
80: (8)                         assert_raises(TypeError, fail)
81: (4)                 def test_invalid_args_tuple(self):
82: (8)                     def fail():
83: (12)                         _vec_string(['a'], np.bytes_, 'strip', 1)
84: (8)                         assert_raises(TypeError, fail)
85: (4)                 def test_invalid_type_descr(self):
86: (8)                     def fail():
87: (12)                         _vec_string(['a'], 'BOGUS', 'strip')
88: (8)                         assert_raises(TypeError, fail)
89: (4)                 def test_invalid_function_args(self):
90: (8)                     def fail():
91: (12)                         _vec_string(['a'], np.bytes_, 'strip', (1,))
92: (8)                         assert_raises(TypeError, fail)
93: (4)                 def test_invalid_result_type(self):
94: (8)                     def fail():
95: (12)                         _vec_string(['a'], np.int_, 'strip')
96: (8)                         assert_raises(TypeError, fail)
97: (4)                 def test_broadcast_error(self):
98: (8)                     def fail():
99: (12)                         _vec_string([['abc', 'def']], np.int_, 'find', ([['a', 'd', 'j'],]))
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

100: (8)             assert_raises(ValueError, fail)
101: (0)
102: (4)
103: (8)
104: (27)
105: (8)
106: (27)
107: (4)
108: (8)
109: (8)
110: (8)
111: (8)
112: (8)
113: (8)
114: (0)
115: (4)
116: (8)
117: (4)
118: (8)
119: (8)
120: (0)
121: (4)
122: (8)
123: (27)
124: (8)
125: (27)
126: (4)
127: (8)
128: (4)
129: (8)
False])
130: (4)
131: (8)
132: (4)
133: (8)
134: (4)
135: (8)
136: (4)
137: (8)
138: (4)
139: (8)
140: (8)
141: (8)
142: (8)
143: (0)
144: (4)
145: (4)
146: (8)
147: (8)
148: (27)
149: (0)
150: (4)
151: (4)
152: (8)
153: (8)
154: (27)
155: (0)
156: (4)
157: (8)
158: (27)
159: (27)
160: (8)
161: (27)
162: (27)
163: (4)
164: (8)
165: (8)
166: (8)
167: (4)

    assert_raises(ValueError, fail)
class TestWhitespace:
    def setup_method(self):
        self.A = np.array([['abc ', '123  '],
                          ['789 ', 'xyz ']]).view(np.chararray)
        self.B = np.array([['abc', '123'],
                          ['789', 'xyz']]).view(np.chararray)
    def test1(self):
        assert_(np.all(self.A == self.B))
        assert_(np.all(self.A >= self.B))
        assert_(np.all(self.A <= self.B))
        assert_(not np.any(self.A > self.B))
        assert_(not np.any(self.A < self.B))
        assert_(not np.any(self.A != self.B))
class TestChar:
    def setup_method(self):
        self.A = np.array('abc1', dtype='c').view(np.chararray)
    def test_it(self):
        assert_equal(self.A.shape, (4,))
        assert_equal(self.A.upper()[:2].tobytes(), b'AB')
class TestComparisons:
    def setup_method(self):
        self.A = np.array([['abc', '123'],
                          ['789', 'xyz']]).view(np.chararray)
        self.B = np.array([['efg', '123 '],
                          ['051', 'tuv']]).view(np.chararray)
    def test_not_equal(self):
        assert_array_equal((self.A != self.B), [[True, False], [True, True]])
    def test_equal(self):
        assert_array_equal((self.A == self.B), [[False, True], [False,
False]])
    def test_greater_equal(self):
        assert_array_equal((self.A >= self.B), [[False, True], [True, True]])
    def test_less_equal(self):
        assert_array_equal((self.A <= self.B), [[True, True], [False, False]])
    def test_greater(self):
        assert_array_equal((self.A > self.B), [[False, False], [True, True]])
    def test_less(self):
        assert_array_equal((self.A < self.B), [[True, False], [False, False]])
    def test_type(self):
        out1 = np.char.equal(self.A, self.B)
        out2 = np.char.equal('a', 'a')
        assert_(isinstance(out1, np.ndarray))
        assert_(isinstance(out2, np.ndarray))
class TestComparisonsMixed1(TestComparisons):
    """Ticket #1276"""
    def setup_method(self):
        TestComparisons.setup_method(self)
        self.B = np.array([['efg', '123 '],
                          ['051', 'tuv']], np.str_).view(np.chararray)
class TestComparisonsMixed2(TestComparisons):
    """Ticket #1276"""
    def setup_method(self):
        TestComparisons.setup_method(self)
        self.A = np.array([['abc', '123'],
                          ['789', 'xyz']], np.str_).view(np.chararray)
class TestInformation:
    def setup_method(self):
        self.A = np.array([[' abc ', ''],
                          ['12345', 'MixedCase'],
                          ['123 \t 345 \0 ', 'UPPER']]).view(np.chararray)
        self.B = np.array([['\u03a3 ', ''],
                          ['12345', 'MixedCase'],
                          ['123 \t 345 \0 ', 'UPPER']]).view(np.chararray)
    def test_len(self):
        assert_(issubclass(np.char.str_len(self.A).dtype.type, np.integer))
        assert_array_equal(np.char.str_len(self.A), [[5, 0], [5, 9], [12, 5]])
        assert_array_equal(np.char.str_len(self.B), [[3, 0], [5, 9], [12, 5]])
    def test_count(self):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

168: (8) assert_(issubclass(self.A.count('').dtype.type, np.integer))
169: (8) assert_array_equal(self.A.count('a'), [[1, 0], [0, 1], [0, 0]])
170: (8) assert_array_equal(self.A.count('123'), [[0, 0], [1, 0], [1, 0]])
171: (8) assert_array_equal(self.A.count('a', 0, 2), [[1, 0], [0, 0], [0, 0]])
172: (8) assert_array_equal(self.B.count('a'), [[0, 0], [0, 1], [0, 0]])
173: (8) assert_array_equal(self.B.count('123'), [[0, 0], [1, 0], [1, 0]])
174: (4) def test_endswith(self):
175: (8)     assert_(issubclass(self.A.endswith('').dtype.type, np.bool_))
176: (8)     assert_array_equal(self.A.endswith(' '), [[1, 0], [0, 0], [1, 0]])
177: (8)     assert_array_equal(self.A.endswith('3', 0, 3), [[0, 0], [1, 0], [1, 0], [1,
0]])
178: (8)
179: (12) def fail():
180: (8)     self.A.endswith('3', 'fdjk')
181: (4)     assert_raises(TypeError, fail)
182: (8) def test_find(self):
183: (8)     assert_(issubclass(self.A.find('a').dtype.type, np.integer))
184: (8)     assert_array_equal(self.A.find('a'), [[1, -1], [-1, 6], [-1, -1]])
185: (8)     assert_array_equal(self.A.find('3'), [[-1, -1], [2, -1], [2, -1]])
186: (8)     assert_array_equal(self.A.find('a', 0, 2), [[1, -1], [-1, -1], [-1,
-1]])
187: (8)     assert_array_equal(self.A.find(['1', 'P']), [[-1, -1], [0, -1], [0,
1]])
188: (4) def test_index(self):
189: (8)     def fail():
190: (8)         self.A.index('a')
191: (8)         assert_raises(ValueError, fail)
192: (8)         assert_(np.char.index('abcba', 'b') == 1)
193: (8)         assert_(issubclass(np.char.index('abcba', 'b').dtype.type,
np.integer))
194: (4)
195: (8)     assert_array_equal(self.A.isalnum(), [[False, False], [True, True],
[False, True]])
196: (4)
197: (8)     assert_(issubclass(self.A.isalpha().dtype.type, np.bool_))
198: (8)     assert_array_equal(self.A.isalpha(), [[False, False], [False, True],
[False, True]])
199: (4)
200: (8)     assert_(issubclass(self.A.isdigit().dtype.type, np.bool_))
201: (8)     assert_array_equal(self.A.isdigit(), [[False, False], [True, False],
[False, False]])
202: (4)
203: (8)     assert_(issubclass(self.A.islower().dtype.type, np.bool_))
204: (8)     assert_array_equal(self.A.islower(), [[True, False], [False, False],
[False, False]])
205: (4)
206: (8)     assert_(issubclass(self.A.isspace().dtype.type, np.bool_))
207: (8)     assert_array_equal(self.A.isspace(), [[False, False], [False, False],
[False, False]])
208: (4)
209: (8)     assert_(issubclass(self.A.istitle().dtype.type, np.bool_))
210: (8)     assert_array_equal(self.A.istitle(), [[False, False], [False, False],
[False, False]])
211: (4)
212: (8)     assert_(issubclass(self.A.isupper().dtype.type, np.bool_))
213: (8)     assert_array_equal(self.A.isupper(), [[False, False], [False, False],
[False, False]])
214: (4) def test_rfind(self):
215: (8)     assert_(issubclass(self.A.rfind('a').dtype.type, np.integer))
216: (8)     assert_array_equal(self.A.rfind('a'), [[1, -1], [-1, 6], [-1, -1]])
217: (8)     assert_array_equal(self.A.rfind('3'), [[-1, -1], [2, -1], [6, -1]])
218: (8)     assert_array_equal(self.A.rfind('a', 0, 2), [[1, -1], [-1, -1], [-1,
-1]])
219: (8)     assert_array_equal(self.A.rfind(['1', 'P']), [[-1, -1], [0, -1], [0,
2]])
220: (4)
221: (8)     def fail():
222: (12)         self.A.rindex('a')
223: (8)         assert_raises(ValueError, fail)

```

```

224: (8)
225: (8)
np.integer))
226: (4)
227: (8)
228: (8)
229: (8)
0[])
230: (8)
231: (12)
232: (8)
233: (0)
234: (4)
235: (8)
236: (27)
237: (27)
238: (26)
239: (8)
240: (27)
241: (27)
242: (4)
243: (8)
244: (15)
245: (15)
246: (8)
247: (8)
248: (8)
249: (15)
250: (15)
251: (8)
252: (8)
253: (4)
254: (8)
255: (8)
256: (8)
257: (8)
258: (8)
259: (8)
260: (8)
261: (8)
262: (15)
263: (8)
264: (8)
265: (4)
266: (8)
267: (8)
268: (4)
269: (8)
270: (8)
271: (4)
272: (8)
273: (8)
274: (4)
275: (8)
276: (8)
277: (8)
278: (8)
279: (24)
280: (24)
281: (8)
282: (4)
283: (8)
284: (8)
285: (8)
286: (8)
287: (8)
288: (16)
289: (8)
290: (8)

                        assert_(np.char.rindex('abcba', 'b') == 3)
                        assert_(issubclass(np.char.rindex('abcba', 'b').dtype.type,
np.integer))

def test_startswith(self):
    assert_(issubclass(self.A.startswith('').dtype.type, np.bool_))
    assert_array_equal(self.A.startswith(' '), [[1, 0], [0, 0], [0, 0]])
    assert_array_equal(self.A.startswith('1', 0, 3), [[0, 0], [1, 0], [1,
0]])

def fail():
    self.A.startswith('3', 'fdjk')
    assert_raises(TypeError, fail)

class TestMethods:
    def setup_method(self):
        self.A = np.array([[' abc ', ''],
                          ['12345', 'MixedCase'],
                          ['123 \t 345 \0 ', 'UPPER']],
                         dtype='S').view(np.chararray)
        self.B = np.array([['\u03a3 ', ''],
                          ['12345', 'MixedCase'],
                          ['123 \t 345 \0 ', 'UPPER']]).view(np.chararray)

    def test_capitalize(self):
        tgt = [[b' abc ', b''],
               [b'12345', b'Mixedcase'],
               [b'123 \t 345 \0 ', b'Upper']]
        assert_(issubclass(self.A.capitalize().dtype.type, np.bytes_))
        assert_array_equal(self.A.capitalize(), tgt)

        tgt = [['\u03c3 ', ''],
               ['12345', 'Mixedcase'],
               ['123 \t 345 \0 ', 'Upper']]
        assert_(issubclass(self.B.capitalize().dtype.type, np.str_))
        assert_array_equal(self.B.capitalize(), tgt)

    def test_center(self):
        assert_(issubclass(self.A.center(10).dtype.type, np.bytes_))
        C = self.A.center([10, 20])
        assert_array_equal(np.char.str_len(C), [[10, 20], [10, 20], [12, 20]])
        C = self.A.center(20, b'#')
        assert_(np.all(C.startswith(b'#')))
        assert_(np.all(C.endswith(b'#')))

        C = np.char.center(b'FOO', [10, 20], [15, 8])
        tgt = [[b' FOO ', b' FOO '],
               [b' FOO ', b' FOO ']]
        assert_(issubclass(C.dtype.type, np.bytes_))
        assert_array_equal(C, tgt)

    def test_decode(self):
        A = np.char.array([b'\u03a3'])
        assert_(A.decode('unicode-escape')[0] == '\u03a3')

    def test_encode(self):
        B = self.B.encode('unicode_escape')
        assert_(B[0][0] == str(' \u03a3 ').encode('latin1'))

    def test_expandtabs(self):
        T = self.A.expandtabs()
        assert_(T[2, 0] == b'123      345 \0')

    def test_join(self):
        A0 = self.A.decode('ascii')
        A = np.char.join(['', '#'], A0)
        assert_(issubclass(A.dtype.type, np.str_))
        tgt = np.array([[' ,a,b,c, ',''],
                      ['1,2,3,4,5', 'M#i#x#e#d#C#a#s#e'],
                      ['1,2,3, ,\t, ,3,4,5, ,\x00, ', 'U#P#P#E#R']])
        assert_array_equal(np.char.join(['', '#'], A0), tgt)

    def test_ljust(self):
        assert_(issubclass(self.A.ljust(10).dtype.type, np.bytes_))
        C = self.A.ljust([10, 20])
        assert_array_equal(np.char.str_len(C), [[10, 20], [10, 20], [12, 20]])
        C = self.A.ljust(20, b'#')
        assert_array_equal(C.startswith(b'#'), [
                           [False, True], [False, False], [False, False]])
        assert_(np.all(C.endswith(b'#')))

        C = np.char.ljust(b'FOO', [10, 20], [15, 8])

```

```

291: (8)          tgt = [[b'FOO      ', b'FOO      '],
292: (15)          [b'FOO      ', b'FOO      ']]
293: (8)          assert_(issubclass(C.dtype.type, np.bytes_))
294: (8)          assert_array_equal(C, tgt)
295: (4)          def test_lower(self):
296: (8)              tgt = [[b' abc ', b''],
297: (15)                  [b'12345', b'mixedcase'],
298: (15)                  [b'123 \t 345 \0 ', b'upper']]
299: (8)          assert_(issubclass(self.A.lower().dtype.type, np.bytes_))
300: (8)          assert_array_equal(self.A.lower(), tgt)
301: (8)          tgt = [['\u03c3 ', ''],
302: (15)                  ['12345', 'mixedcase'],
303: (15)                  ['123 \t 345 \0 ', 'upper']]
304: (8)          assert_(issubclass(self.B.lower().dtype.type, np.str_))
305: (8)          assert_array_equal(self.B.lower(), tgt)
306: (4)          def test_lstrip(self):
307: (8)              tgt = [[b'abc ', b''],
308: (15)                  [b'12345', b'MixedCase'],
309: (15)                  [b'123 \t 345 \0 ', b'UPPER']]
310: (8)          assert_(issubclass(self.A.lstrip().dtype.type, np.bytes_))
311: (8)          assert_array_equal(self.A.lstrip(), tgt)
312: (8)          tgt = [[b' abc', b''],
313: (15)                  [b'2345', b'ixedCase'],
314: (15)                  [b'23 \t 345 \x00', b'UPPER']]
315: (8)          assert_array_equal(self.A.lstrip([b'1', b'M']), tgt)
316: (8)          tgt = [['\u03a3 ', ''],
317: (15)                  ['12345', 'MixedCase'],
318: (15)                  ['123 \t 345 \0 ', 'UPPER']]
319: (8)          assert_(issubclass(self.B.lstrip().dtype.type, np.str_))
320: (8)          assert_array_equal(self.B.lstrip(), tgt)
321: (4)          def test_partition(self):
322: (8)              P = self.A.partition([b'3', b'M'])
323: (8)              tgt = [[[b' abc ', b'', b''), (b'', b'', b'')],
324: (15)                  [(b'12', b'3', b'45'), (b'', b'M', b'ixedCase')],
325: (15)                  [(b'12', b'3', b' \t 345 \0 '), (b'UPPER', b'', b'')]]
326: (8)          assert_(issubclass(P.dtype.type, np.bytes_))
327: (8)          assert_array_equal(P, tgt)
328: (4)          def test_replace(self):
329: (8)              R = self.A.replace([b'3', b'a'],
330: (27)                  [b'#####', b'@'])
331: (8)              tgt = [[b' abc ', b''],
332: (15)                  [b'12#####45', b'MixedC@se'],
333: (15)                  [b'12##### \t #####45 \x00', b'UPPER']]
334: (8)          assert_(issubclass(R.dtype.type, np.bytes_))
335: (8)          assert_array_equal(R, tgt)
336: (4)          def test_rjust(self):
337: (8)              assert_(issubclass(self.A.rjust(10).dtype.type, np.bytes_))
338: (8)              C = self.A.rjust([10, 20])
339: (8)              assert_array_equal(np.char.str_len(C), [[10, 20], [10, 20], [12, 20]])
340: (8)              C = self.A.rjust(20, b'#')
341: (8)              assert_(np.all(C.startswith(b'#')))
342: (8)              assert_array_equal(C.endswith(b'#'),
343: (27)                  [[False, True], [False, False], [False, False]])
344: (8)              C = np.char.rjust(b'FOO', [[10, 20], [15, 8]])
345: (8)              tgt = [[b'      FOO', b'      FOO'],
346: (15)                  [b'      FOO', b'      FOO']]
347: (8)          assert_(issubclass(C.dtype.type, np.bytes_))
348: (8)          assert_array_equal(C, tgt)
349: (4)          def test_rpartition(self):
350: (8)              P = self.A.rpartition([b'3', b'M'])
351: (8)              tgt = [[[b'', b'', b' abc'), (b'', b'', b'')],
352: (15)                  [(b'12', b'3', b'45'), (b'', b'M', b'ixedCase')],
353: (15)                  [(b'123 \t ', b'3', b'45 \0 '), (b'', b'', b'UPPER')]]
354: (8)          assert_(issubclass(P.dtype.type, np.bytes_))
355: (8)          assert_array_equal(P, tgt)
356: (4)          def test_rsplit(self):
357: (8)              A = self.A.rsplit(b'3')
358: (8)              tgt = [[[b' abc ', [b'']],
359: (15)                  [[b'12', b'45'], [b'MixedCase']]]]
```

```

360: (15)          [[[b'12', b'\t ', b'45 \x00 '], [b'UPPER']]]
361: (8)           assert_(issubclass(A.dtype.type, np.object_))
362: (8)           assert_equal(A.tolist(), tgt)
363: (4)            def test_rstrip(self):
364: (8)              assert_(issubclass(self.A.rstrip().dtype.type, np.bytes_))
365: (8)              tgt = [[b' abc', b''],
366: (15)                  [b'12345', b'MixedCase'],
367: (15)                  [b'123 \t 345', b'UPPER']]
368: (8)              assert_array_equal(self.A.rstrip(), tgt)
369: (8)              tgt = [[b' abc ', b''],
370: (15)                  [b'1234', b'MixedCase'],
371: (15)                  [b'123 \t 345 \x00', b'UPP']]
372: (15)                  ]
373: (8)              assert_array_equal(self.A.rstrip([b'5', b'ER']), tgt)
374: (8)              tgt = [['\u03a3', ''],
375: (15)                  ['12345', 'MixedCase'],
376: (15)                  ['123 \t 345', 'UPPER']]
377: (8)              assert_(issubclass(self.B.rstrip().dtype.type, np.str_))
378: (8)              assert_array_equal(self.B.rstrip(), tgt)
379: (4)            def test_strip(self):
380: (8)              tgt = [[b'abc', b''],
381: (15)                  [b'12345', b'MixedCase'],
382: (15)                  [b'123 \t 345', b'UPPER']]
383: (8)              assert_(issubclass(self.A.strip().dtype.type, np.bytes_))
384: (8)              assert_array_equal(self.A.strip(), tgt)
385: (8)              tgt = [[b' abc ', b''],
386: (15)                  [b'234', b'ixedCas'],
387: (15)                  [b'23 \t 345 \x00', b'UPP']]
388: (8)              assert_array_equal(self.A.strip([b'15', b'EReM']), tgt)
389: (8)              tgt = [['\u03a3', ''],
390: (15)                  ['12345', 'MixedCase'],
391: (15)                  ['123 \t 345', 'UPPER']]
392: (8)              assert_(issubclass(self.B.strip().dtype.type, np.str_))
393: (8)              assert_array_equal(self.B.strip(), tgt)
394: (4)            def test_split(self):
395: (8)              A = self.A.split(b'3')
396: (8)              tgt = [
397: (15)                  [[b' abc '], [b'']],
398: (15)                  [[b'12', b'45'], [b'MixedCase']],
399: (15)                  [[b'12', b'\t ', b'45 \x00 '], [b'UPPER']]]
400: (8)              assert_(issubclass(A.dtype.type, np.object_))
401: (8)              assert_equal(A.tolist(), tgt)
402: (4)            def test_splitlines(self):
403: (8)              A = np.char.array(['abc\nfds\nwer']).splitlines()
404: (8)              assert_(issubclass(A.dtype.type, np.object_))
405: (8)              assert_(A.shape == (1,))
406: (8)              assert_(len(A[0]) == 3)
407: (4)            def test_swapcase(self):
408: (8)              tgt = [[b' ABC ', b''],
409: (15)                  [b'12345', b'mIXEDcASE'],
410: (15)                  [b'123 \t 345 \0 ', b'upper']]
411: (8)              assert_(issubclass(self.A.swapcase().dtype.type, np.bytes_))
412: (8)              assert_array_equal(self.A.swapcase(), tgt)
413: (8)              tgt = [['\u03c3', ''],
414: (15)                  ['12345', 'mIXEDcASE'],
415: (15)                  ['123 \t 345 \0 ', 'upper']]
416: (8)              assert_(issubclass(self.B.swapcase().dtype.type, np.str_))
417: (8)              assert_array_equal(self.B.swapcase(), tgt)
418: (4)            def test_title(self):
419: (8)              tgt = [[b' Abc ', b''],
420: (15)                  [b'12345', b'Mixedcase'],
421: (15)                  [b'123 \t 345 \0 ', b'Upper']]
422: (8)              assert_(issubclass(self.A.title().dtype.type, np.bytes_))
423: (8)              assert_array_equal(self.A.title(), tgt)
424: (8)              tgt = [['\u03a3', ''],
425: (15)                  ['12345', 'Mixedcase'],
426: (15)                  ['123 \t 345 \0 ', 'Upper']]
427: (8)              assert_(issubclass(self.B.title().dtype.type, np.str_))
428: (8)              assert_array_equal(self.B.title(), tgt)

```

```

429: (4)
430: (8)
431: (15)
432: (15)
433: (8)
434: (8)
435: (8)
436: (15)
437: (15)
438: (8)
439: (8)
440: (4)
441: (8)
442: (12)
443: (8)
444: (8)
445: (8)
446: (16)
447: (4)
448: (8)
449: (12)
450: (8)
451: (8)
452: (8)
453: (16)
454: (0)
455: (4)
456: (8)
457: (27)
458: (8)
459: (27)
460: (4)
461: (8)
462: (23)
463: (8)
464: (8)
465: (4)
466: (8)
467: (23)
468: (8)
469: (4)
470: (8)
471: (8)
472: (12)
473: (27)
474: (12)
475: (8)
476: (12)
477: (37)
478: (16)
479: (4)
480: (8)
481: (8)
482: (12)
483: (27)
484: (12)
485: (8)
486: (12)
487: (37)
488: (16)
489: (4)
490: (8)
491: (8)
492: (8)
493: (8)
494: (23)
495: (8)
496: (8)
497: (8)

        def test_upper(self):
            tgt = [[b' ABC ', b''],
                   [b'12345', b'MIXEDCASE'],
                   [b'123 \t 345 \0 ', b'UPPER']]
            assert_(issubclass(self.A.upper().dtype.type, np.bytes_))
            assert_array_equal(self.A.upper(), tgt)
            tgt = [['\u03a3', ''],
                   ['12345', 'MIXEDCASE'],
                   ['123 \t 345 \0 ', 'UPPER']]
            assert_(issubclass(self.B.upper().dtype.type, np.str_))
            assert_array_equal(self.B.upper(), tgt)

        def test_isnumeric(self):
            def fail():
                self.A.isnumeric()
            assert_raises(TypeError, fail)
            assert_(issubclass(self.B.isnumeric().dtype.type, np.bool_))
            assert_array_equal(self.B.isnumeric(), [
                [False, False], [True, False], [False, False]])

        def test_isdecimal(self):
            def fail():
                self.A.isdecimal()
            assert_raises(TypeError, fail)
            assert_(issubclass(self.B.isdecimal().dtype.type, np.bool_))
            assert_array_equal(self.B.isdecimal(), [
                [False, False], [True, False], [False, False]])

    class TestOperations:
        def setup_method(self):
            self.A = np.array([[['abc', '123'],
                               ['789', 'xyz']]]).view(np.chararray)
            self.B = np.array([[['efg', '456'],
                               ['051', 'tuv']]]).view(np.chararray)

        def test_add(self):
            AB = np.array([[['abcefg', '123456'],
                           ['789051', 'xyztuv']]]).view(np.chararray)
            assert_array_equal(AB, (self.A + self.B))
            assert_(len((self.A + self.B)[0][0]) == 6)

        def test_radd(self):
            QA = np.array([[['qabc', 'q123'],
                           ['q789', 'qxyz']]]).view(np.chararray)
            assert_array_equal(QA, ('q' + self.A))

        def test_mul(self):
            A = self.A
            for r in (2, 3, 5, 7, 197):
                Ar = np.array([[A[0, 0]**r, A[0, 1]**r],
                              [A[1, 0]**r, A[1, 1]**r]]].view(np.chararray)
                assert_array_equal(Ar, (self.A * r))
            for ob in [object(), 'qrs']:
                with assert_raises_regex(ValueError,
   'Can only multiply by integers'):
                    A*ob

        def test_rmul(self):
            A = self.A
            for r in (2, 3, 5, 7, 197):
                Ar = np.array([[A[0, 0]**r, A[0, 1]**r],
                              [A[1, 0]**r, A[1, 1]**r]]).view(np.chararray)
                assert_array_equal(Ar, (r * self.A))
            for ob in [object(), 'qrs']:
                with assert_raises_regex(ValueError,
   'Can only multiply by integers'):
                    ob * A

        def test_mod(self):
            """Ticket #856"""
            F = np.array([[%d, %f], [%s, %r]]).view(np.chararray)
            C = np.array([[3, 7], [19, 1]])
            FC = np.array([[3, '7.000000'],
                          ['19', '1']]).view(np.chararray)
            assert_array_equal(FC, F % C)
            A = np.array([[%.3f, %d], [%s, %r]]).view(np.chararray)
            A1 = np.array([[1.000, 1], ['1', '1']]).view(np.chararray)

```

```

498: (8)             assert_array_equal(A1, (A % 1))
499: (8)             A2 = np.array([[1.000, '2'], [3, '4']]).view(np.chararray)
500: (8)             assert_array_equal(A2, (A % [[1, 2], [3, 4]]))
501: (4)             def test_rmod(self):
502: (8)                 assert_(("%s" % self.A) == str(self.A))
503: (8)                 assert_(("%r" % self.A) == repr(self.A))
504: (8)                 for ob in [42, object()]:
505: (12)                     with assert_raises_regex(
506: (20)                         TypeError, "unsupported operand type.* and 'chararray'"):
507: (16)                         ob % self.A
508: (4)             def test_slice(self):
509: (8)                 """Regression test for https://github.com/numpy/numpy/issues/5982"""
510: (8)                 arr = np.array([['abc ', 'def '], ['geh ', 'ijk ']],
511: (23)                         dtype='S4').view(np.chararray)
512: (8)                 sl1 = arr[:]
513: (8)                 assert_array_equal(sl1, arr)
514: (8)                 assert_(sl1.base is arr)
515: (8)                 assert_(sl1.base.base is arr.base)
516: (8)                 sl2 = arr[:, :]
517: (8)                 assert_array_equal(sl2, arr)
518: (8)                 assert_(sl2.base is arr)
519: (8)                 assert_(sl2.base.base is arr.base)
520: (8)                 assert_(arr[0, 0] == b'abc')
521: (0)             def test_empty_indexing():
522: (4)                 """Regression test for ticket 1948."""
523: (4)                 s = np.chararray((4,))
524: (4)                 assert_(s[[]].size == 0)
525: (0)             @pytest.mark.parametrize(["dt1", "dt2"],
526: (8)                         [("S", "U"), ("U", "S"), ("S", "O"), ("U", "O"),
527: (9)                             ("S", "d"), ("S", "V")])
528: (0)             def test_add_types(dt1, dt2):
529: (4)                 arr1 = np.array([1234234], dtype=dt1)
530: (4)                 arr2 = np.array([b"423"], dtype=dt2)
531: (4)                 with pytest.raises(TypeError,
532: (12)                     match=f".*same dtype kind.*{arr1.dtype}.*{arr2.dtype}"):
533: (8)                     np.char.add(arr1, arr2)

```

---

## File 89 - test\_deprecations.py:

```

1: (0)             """
2: (0)             Tests related to deprecation warnings. Also a convenient place
3: (0)             to document how deprecations should eventually be turned into errors.
4: (0)             """
5: (0)             import datetime
6: (0)             import operator
7: (0)             import warnings
8: (0)             import pytest
9: (0)             import tempfile
10: (0)            import re
11: (0)            import sys
12: (0)            import numpy as np
13: (0)            from numpy.testing import (
14: (4)                assert_raises, assert_warns, assert_, assert_array_equal, SkipTest,
15: (4)                KnownFailureException, break_cycles,
16: (4)            )
17: (0)            from numpy.core._multiarray_tests import fromstring_null_term_c_api
18: (0)            try:
19: (4)                import pytz
20: (4)                _has_pytz = True
21: (0)            except ImportError:
22: (4)                _has_pytz = False
23: (0)            class _DeprecationTestCase:
24: (4)                message = ''
25: (4)                warning_cls = DeprecationWarning
26: (4)                def setup_method(self):
27: (8)                    self.warn_ctx = warnings.catch_warnings(record=True)
28: (8)                    self.log = self.warn_ctx.__enter__()

```

```

29: (8)             warnings.filterwarnings("always", category=self.warning_cls)
30: (8)             warnings.filterwarnings("always", message=self.message,
31: (32)                     category=self.warning_cls)
32: (4)             def teardown_method(self):
33: (8)                 self.warn_ctx.__exit__()
34: (4)             def assert_deprecated(self, function, num=1, ignore_others=False,
35: (26)                     function_fails=False,
36: (26)                     exceptions=np._NoValue,
37: (26)                     args=(), kwargs={}):
38: (8)                 """Test if DeprecationWarnings are given and raised.
39: (8)                 This first checks if the function when called gives `num`
40: (8)                 DeprecationWarnings, after that it tries to raise these
41: (8)                 DeprecationWarnings and compares them with `exceptions`.
42: (8)                 The exceptions can be different for cases where this code path
43: (8)                 is simply not anticipated and the exception is replaced.
44: (8)             Parameters
45: (8)             -----
46: (8)             function : callable
47: (12)                 The function to test
48: (8)             num : int
49: (12)                 Number of DeprecationWarnings to expect. This should normally be
1.
50: (8)             ignore_others : bool
51: (12)                 Whether warnings of the wrong type should be ignored (note that
52: (12)                     the message is not checked)
53: (8)             function_fails : bool
54: (12)                 If the function would normally fail, setting this will check for
55: (12)                     warnings inside a try/except block.
56: (8)             exceptions : Exception or tuple of Exceptions
57: (12)                 Exception to expect when turning the warnings into an error.
58: (12)                 The default checks for DeprecationWarnings. If exceptions is
59: (12)                     empty the function is expected to run successfully.
60: (8)             args : tuple
61: (12)                 Arguments for `function`
62: (8)             kwargs : dict
63: (12)                 Keyword arguments for `function`
64: (8)             """
65: (8)             __tracebackhide__ = True # Hide traceback for py.test
66: (8)             self.log[:] = []
67: (8)             if exceptions is np._NoValue:
68: (12)                 exceptions = (self.warning_cls,)
69: (8)             try:
70: (12)                 function(*args, **kwargs)
71: (8)             except (Exception if function_fails else tuple()):
72: (12)                 pass
73: (8)             num_found = 0
74: (8)             for warning in self.log:
75: (12)                 if warning.category is self.warning_cls:
76: (16)                     num_found += 1
77: (12)                 elif not ignore_others:
78: (16)                     raise AssertionError(
79: (24)                         "expected %s but got: %s" %
80: (24)                         (self.warning_cls.__name__, warning.category))
81: (8)             if num is not None and num_found != num:
82: (12)                 msg = "%i warnings found but %i expected." % (len(self.log), num)
83: (12)                 lst = [str(w) for w in self.log]
84: (12)                 raise AssertionError("\n".join([msg] + lst))
85: (8)             with warnings.catch_warnings():
86: (12)                 warnings.filterwarnings("error", message=self.message,
87: (36)                     category=self.warning_cls)
88: (12)             try:
89: (16)                 function(*args, **kwargs)
90: (16)                 if exceptions != tuple():
91: (20)                     raise AssertionError(
92: (28)                         "No error raised during function call")
93: (12)                 except exceptions:
94: (16)                     if exceptions == tuple():
95: (20)                         raise AssertionError(
96: (28)                             "Error raised during function call")

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

97: (4)
98: (8)
99: (8)
100: (8)
101: (24)
102: (8)
103: (8)
104: (24)
105: (0)
106: (4)
107: (0)
108: (4)
109: (4)
110: (4)
111: (4)
112: (4)
113: (4)
114: (4)
115: (4)
116: (8)
117: (8)
118: (4)
119: (24)
120: (4)
121: (8)
122: (8)
123: (8)
124: (0)
125: (4)
126: (5)
127: (5)
Deprecation-
128: (5)
to
129: (5)
130: (4)
131: (4)
132: (8)
133: (8)
134: (8)
135: (24)
136: (24)
137: (8)
138: (0)
class
TestBinaryReprInsufficientWidthParameterForRepresentation(_DeprecationTestCase):
139: (4)
140: (4)
to
141: (4)
form,
142: (4)
representation
143: (4)
behavior
144: (4)
in the future.
145: (4)
146: (4)
147: (8)
148: (8)
149: (8)
150: (24)
151: (8)
152: (4)
153: (8)
154: (8)
155: (8)
156: (24)
157: (8)

```

def assert\_not\_deprecated(self, function, args=(), kwargs={}):  
 """Test that warnings are not raised.  
 This is just a shorthand for:  
 self.assert\_deprecated(function, num=0, ignore\_others=True,  
 exceptions=tuple(), args=args, kwargs=kwargs)  
 """  
 self.assert\_deprecated(function, num=0, ignore\_others=True,  
 exceptions=tuple(), args=args, kwargs=kwargs)

class \_VisibleDeprecationTestCase(\_DeprecationTestCase):  
 warning\_cls = np.VisibleDeprecationWarning

class TestDatetime64Timezone(\_DeprecationTestCase):  
 """Parsing of datetime64 with timezones deprecated in 1.11.0, because  
 datetime64 is now timezone naive rather than UTC only.  
 It will be quite a while before we can remove this, because, at the very  
 least, a lot of existing code uses the 'Z' modifier to avoid conversion  
 from local time to UTC, even if otherwise it handles time in a timezone  
 naive fashion.  
 """

def test\_string(self):  
 self.assert\_deprecated(np.datetime64, args=('2000-01-01T00+01',))  
 self.assert\_deprecated(np.datetime64, args=('2000-01-01T00Z',))  
@ pytest.mark.skipif(not \_has\_pytz,  
 reason="The pytz module is not available.")

def test\_datetime(self):  
 tz = pytz.timezone('US/Eastern')  
 dt = datetime.datetime(2000, 1, 1, 0, 0, tzinfo=tz)  
 self.assert\_deprecated(np.datetime64, args=(dt,))

class TestArrayDataAttributeAssignmentDeprecation(\_DeprecationTestCase):  
 """Assigning the 'data' attribute of an ndarray is unsafe as pointed  
 out in gh-7093. Eventually, such assignment should NOT be allowed, but  
 in the interests of maintaining backwards compatibility, only a  
 Warning will be raised instead for the time being to give developers time  
 to refactor relevant code.  
 """

def test\_data\_attr\_assignment(self):  
 a = np.arange(10)  
 b = np.linspace(0, 1, 10)  
 self.message = ("Assigning the 'data' attribute is an "  
 "inherently unsafe operation and will "  
 "be removed in the future.")  
 self.assert\_deprecated(a.\_\_setattr\_\_, args=('data', b.data))

class
TestBinaryReprInsufficientWidthParameterForRepresentation(\_DeprecationTestCase):
139: (4)
140: (4)
to
141: (4)
form,
142: (4)
representation
143: (4)
behavior
144: (4)
in the future.
145: (4)
146: (4)
147: (8)
148: (8)
149: (8)
150: (24)
151: (8)
152: (4)
153: (8)
154: (8)
155: (8)
156: (24)
157: (8)

If a 'width' parameter is passed into ``binary\_repr`` that is insufficient  
 represent the number in base 2 (positive) or 2's complement (negative)  
 the function used to silently ignore the parameter and return a  
 using the minimal number of bits needed for the form in question. Such  
 is now considered unsafe from a user perspective and will raise an error  
 """

def test\_insufficient\_width\_positive(self):  
 args = (10,)  
 kwargs = {'width': 2}  
 self.message = ("Insufficient bit width provided. This behavior "  
 "will raise an error in the future.")  
 self.assert\_deprecated(np.binary\_repr, args=args, kwargs=kwargs)

def test\_insufficient\_width\_negative(self):  
 args = (-5,)  
 kwargs = {'width': 2}  
 self.message = ("Insufficient bit width provided. This behavior "  
 "will raise an error in the future.")  
 self.assert\_deprecated(np.binary\_repr, args=args, kwargs=kwargs)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

158: (0)
159: (4)
160: (4)
161: (8)
162: (12)
163: (8)
164: (12)
165: (8)
166: (8)
167: (8)
168: (8)
169: (0)
170: (4)
171: (8)
172: (8)
173: (8)
174: (22)
175: (22)
176: (8)
177: (12)
178: (8)
179: (8)
180: (0)
181: (4)
182: (4)
183: (4)
184: (4)
185: (4)
186: (8)
187: (12)
188: (8)
189: (16)
190: (12)
191: (0)
192: (4)
193: (8)
194: (8)
195: (8)
196: (0)
197: (4)
198: (8)
199: (8)
200: (0)
201: (4)
202: (8)
203: (8)
204: (0)
205: (4)
206: (8)
207: (12)
208: (12)
209: (12)
210: (12)
211: (12)
212: (12)
213: (0)
214: (4)
215: (4)
216: (8)
217: (4)
218: (8)
219: (8)
220: (8)
221: (0)
222: (4)
223: (8)
224: (0)
225: (4)
226: (8)

class TestDTypeAttributeIsDTypeDeprecation(_DeprecationTestCase):
    message = r".*`dtype` attribute"
    def test_deprecation_dtype_attribute_is_dtype(self):
        class dt:
            dtype = "f8"
        class vdt(np.void):
            dtype = "f,f"
        self.assert_deprecated(lambda: np.dtype(dt))
        self.assert_deprecated(lambda: np.dtype(dt()))
        self.assert_deprecated(lambda: np.dtype(vdt))
        self.assert_deprecated(lambda: np.dtype(vdt(1)))
class TestTestDeprecated:
    def test_assert_DEPRECATED(self):
        test_case_instance = _DeprecationTestCase()
        test_case_instance.setup_method()
        assert_raises(AssertionError,
                     test_case_instance.assert_DEPRECATED,
                     lambda: None)
    def foo():
        warnings.warn("foo", category=DeprecationWarning, stacklevel=2)
        test_case_instance.assert_DEPRECATED(foo)
        test_case_instance.teardown_method()
class TestNonNumericConjugate(_DeprecationTestCase):
    """
    Deprecate no-op behavior of ndarray.conjugate on non-numeric dtypes,
    which conflicts with the error behavior of np.conjugate.
    """
    def test_conjugate(self):
        for a in np.array(5), np.array(5j):
            self.assert_not_DEPRECATED(a.conjugate)
        for a in (np.array('s'), np.array('2016', 'M'),
                  np.array((1, 2), [('a', int), ('b', int)])):
            self.assert_DEPRECATED(a.conjugate)
class TestNPY_CHAR(_DeprecationTestCase):
    def test_npy_char_DEPRECATED(self):
        from numpy.core._multiarray_tests import npy_char_DEPRECATED
        self.assert_DEPRECATED(npy_char_DEPRECATED)
        assert_(npy_char_DEPRECATED() == 'S1')
class TestPyArray_AS1D(_DeprecationTestCase):
    def test_npy_pyarrayas1d_DEPRECATED(self):
        from numpy.core._multiarray_tests import npy_pyarrayas1d_DEPRECATED
        assert_raises(NotImplementedError, npy_pyarrayas1d_DEPRECATED)
class TestPyArray_AS2D(_DeprecationTestCase):
    def test_npy_pyarrayas2d_DEPRECATED(self):
        from numpy.core._multiarray_tests import npy_pyarrayas2d_DEPRECATED
        assert_raises(NotImplementedError, npy_pyarrayas2d_DEPRECATED)
class TestDatetimeEvent(_DeprecationTestCase):
    def test_3_tuple(self):
        for cls in (np.datetime64, np.timedelta64):
            self.assert_not_DEPRECATED(cls, args=(1, ('ms', 2)))
            self.assert_not_DEPRECATED(cls, args=(1, ('ms', 2, 1, None)))
            self.assert_DEPRECATED(cls, args=(1, ('ms', 2, 'event')))
            self.assert_DEPRECATED(cls, args=(1, ('ms', 2, 63)))
            self.assert_DEPRECATED(cls, args=(1, ('ms', 2, 1, 'event')))
            self.assert_DEPRECATED(cls, args=(1, ('ms', 2, 1, 63)))
class TestTruthTestingEmptyArrays(_DeprecationTestCase):
    message = '.*truth value of an empty array is ambiguous.*'
    def test_1d(self):
        self.assert_DEPRECATED(bool, args=(np.array([]),))
    def test_2d(self):
        self.assert_DEPRECATED(bool, args=(np.zeros((1, 0)),))
        self.assert_DEPRECATED(bool, args=(np.zeros((0, 1)),))
        self.assert_DEPRECATED(bool, args=(np.zeros((0, 0)),))
class TestBincount(_DeprecationTestCase):
    def test_bincount_minlength(self):
        self.assert_DEPRECATED(lambda: np.bincount([1, 2, 3], minlength=None))
class TestGeneratorSum(_DeprecationTestCase):
    def test_generator_sum(self):
        self.assert_DEPRECATED(np.sum, args=((i for i in range(5)),))

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

227: (0)
228: (4)
229: (8)
230: (0)
231: (4)
232: (4)
233: (4)
234: (8)
235: (8)
236: (12)
237: (12)
238: (12)
239: (12)
240: (12)
241: (12)
242: (12)
243: (16)
244: (16)
245: (16)
246: (16)
247: (4)
248: (4)
249: (8)
250: (8)
251: (8)
252: (8)
253: (8)
254: (8)
255: (8)
256: (12)
257: (12)
258: (8)
259: (12)
260: (12)
261: (12)
262: (0)
263: (4)
264: (8)
265: (8)
266: (8)
267: (8)
268: (0)
269: (4)
270: (4)
271: (8)
272: (0)
273: (4)
274: (8)
275: (8)
276: (0)
277: (4)
278: (4)
279: (8)
280: (8)
281: (4)
282: (8)
283: (8)
284: (8)
285: (12)
286: (8)
287: (0)
288: (4)
289: (4)
290: (8)
291: (8)
292: (8)
293: (8)
294: (4)
295: (4)

class TestFromstring(_DeprecationTestCase):
    def test_fromstring(self):
        self.assert_deprecated(np.fromstring, args=( '\x00'*80,))

class TestFromStringAndFileInvalidData(_DeprecationTestCase):
    message = "string or file could not be read to its end"
    @pytest.mark.parametrize("invalid_str", [",invalid_data", "invalid_sep"])
    def test_deprecate_unparsable_data_file(self, invalid_str):
        x = np.array([1.51, 2, 3.51, 4], dtype=float)
        with tempfile.TemporaryFile(mode="w") as f:
            x.tofile(f, sep=',', format='%.2f')
            f.write(invalid_str)
            f.seek(0)
            self.assert_deprecated(lambda: np.fromfile(f, sep=","))

        f.seek(0)
        self.assert_deprecated(lambda: np.fromfile(f, sep=",", count=5))
        with warnings.catch_warnings():
            warnings.simplefilter("error", DeprecationWarning)
            f.seek(0)
            res = np.fromfile(f, sep=",", count=4)
            assert_array_equal(res, x)

    @pytest.mark.parametrize("invalid_str", [",invalid_data", "invalid_sep"])
    def test_deprecate_unparsable_string(self, invalid_str):
        x = np.array([1.51, 2, 3.51, 4], dtype=float)
        x_str = "1.51,2,3.51,4{}".format(invalid_str)
        self.assert_deprecated(lambda: np.fromstring(x_str, sep=","))

        self.assert_deprecated(lambda: np.fromstring(x_str, sep=",", count=5))
        bytestr = x_str.encode("ascii")
        self.assert_deprecated(lambda: fromstring_null_term_c_api(bytestr))
        with assert_warns(DeprecationWarning):
            res = np.fromstring(x_str, sep=",", count=5)
            assert_array_equal(res[:-1], x)

        with warnings.catch_warnings():
            warnings.simplefilter("error", DeprecationWarning)
            res = np.fromstring(x_str, sep=",", count=4)
            assert_array_equal(res, x)

class Test_GetSet_NumericOps(_DeprecationTestCase):
    def test_get_numeric_ops(self):
        from numpy.core._multiarray_tests import getset_numericops
        self.assert_deprecated(getset_numericops, num=2)
        self.assert_deprecated(np.set_numeric_ops, kwargs={})
        assert_raises(ValueError, np.set_numeric_ops, add='abc')

class TestShape1Fields(_DeprecationTestCase):
    warning_cls = FutureWarning
    def test_shape_1_fields(self):
        self.assert_deprecated(np.dtype, args=[(['a', int, 1]],))

class TestNonZero(_DeprecationTestCase):
    def test_zerod(self):
        self.assert_deprecated(lambda: np.nonzero(np.array(0)))
        self.assert_deprecated(lambda: np.nonzero(np.array(1)))

class TestToString(_DeprecationTestCase):
    message = re.escape("tostring() is deprecated. Use tobytes() instead.")
    def test_tostring(self):
        arr = np.array(list(b"test\xFF"), dtype=np.uint8)
        self.assert_deprecated(arr.tostring)

    def test_tostring_matches_tobytes(self):
        arr = np.array(list(b"test\xFF"), dtype=np.uint8)
        b = arr.tobytes()
        with assert_warns(DeprecationWarning):
            s = arr.tostring()
            assert s == b

class TestDTypeCoercion(_DeprecationTestCase):
    message = "Converting .* to a dtype .*is deprecated"
    deprecated_types = [
        np.generic, np.flexible, np.number,
        np.inexact, np.floating, np.complexfloating,
        np.integer, np.unsignedinteger, np.signedinteger,
        np.character,
    ]
    def test_dtype_coercion(self):

```

```

296: (8)
297: (12)
298: (4)
299: (8)
300: (12)
301: (4)
302: (8)
303: (12)
304: (16)
305: (8)
306: (12)
307: (0)
308: (4)
309: (4)
310: (8)
311: (8)
312: (8)
313: (4)
314: (4)
315: (8)
316: (12)
317: (12)
318: (12)
319: (12)
320: (4)
321: (8)
322: (12)
323: (12)
324: (12)
325: (12)
326: (0)
327: (4)
328: (4)
329: (4)
330: (8)
331: (8)
332: (8)
333: (8)
334: (8)
335: (8)
336: (8)
337: (4)
338: (8)
339: (8)
340: (8)
341: (8)
342: (35)
343: (0)
344: (4)
345: (8)
346: (8)
mode='Cilp'))
347: (8)
side='Random'))
348: (0)
349: (4)
350: (15)
351: (4)
352: (8)
353: (8)
354: (8)
355: (8)
356: (8)
357: (8)
358: (0)
359: (4)
360: (4)

        for scalar_type in self.deprecated_types:
            self.assert_deprecated(np.dtype, args=(scalar_type,))
    def test_array_construction(self):
        for scalar_type in self.deprecated_types:
            self.assert_deprecated(np.array, args=([], scalar_type,))
    def test_not_DEPRECATED(self):
        for group in np.sctypes.values():
            for scalar_type in group:
                self.assert_NOT_DEPRECATED(np.dtype, args=(scalar_type,))
        for scalar_type in [type, dict, list, tuple]:
            self.assert_NOT_DEPRECATED(np.dtype, args=(scalar_type,))
class BuiltInRoundComplexDType(_DeprecationTestCase):
    deprecated_types = [np.csingle, np.cdouble, np.clongdouble]
    not_DEPRECATED_types = [
        np.int8, np.int16, np.int32, np.int64,
        np.uint8, np.uint16, np.uint32, np.uint64,
        np.float16, np.float32, np.float64,
    ]
    def test_DEPRECATED(self):
        for scalar_type in self.deprecated_types:
            scalar = scalar_type(0)
            self.assert_DEPRECATED(round, args=(scalar,))
            self.assert_DEPRECATED(round, args=(scalar, 0))
            self.assert_DEPRECATED(round, args=(scalar,), kwargs={'ndigits': 0})
    def test_not_DEPRECATED(self):
        for scalar_type in self.not_DEPRECATED_types:
            scalar = scalar_type(0)
            self.assert_NOT_DEPRECATED(round, args=(scalar,))
            self.assert_NOT_DEPRECATED(round, args=(scalar, 0))
            self.assert_NOT_DEPRECATED(round, args=(scalar,), kwargs={'ndigits': 0})
class TestIncorrectAdvancedIndexWithEmptyResult(_DeprecationTestCase):
    message = "Out of bound index found. This was previously ignored.*"
    @pytest.mark.parametrize("index", [[(3, 0)], ([0, 0], [3, 0])])
    def test_empty_subspace(self, index):
        arr = np.ones((2, 2, 0))
        self.assert_DEPRECATED(arr.__getitem__, args=(index,))
        self.assert_DEPRECATED(arr.__setitem__, args=(index, 0.))
        arr2 = np.ones((2, 2, 1))
        index2 = (slice(0, 0),) + index
        self.assert_DEPRECATED(arr2.__getitem__, args=(index2,))
        self.assert_DEPRECATED(arr2.__setitem__, args=(index2, 0.))
    def test_empty_index_broadcast_not_DEPRECATED(self):
        arr = np.ones((2, 2, 2))
        index = ([[3], [2]], []) # broadcast to an empty result.
        self.assert_NOT_DEPRECATED(arr.__getitem__, args=(index,))
        self.assert_NOT_DEPRECATED(arr.__setitem__,
                                   args=(index, np.empty((2, 0, 2))))
class TestNonExactMatchDeprecation(_DeprecationTestCase):
    def test_non_exact_match(self):
        arr = np.array([[3, 6, 6], [4, 5, 1]])
        self.assert_DEPRECATED(lambda: np.ravel_multi_index(arr, (7, 6)),
                              self.assert_DEPRECATED(lambda: np.searchsorted(arr[0], 4,
class TestMatrixInOuter(_DeprecationTestCase):
    message = (r"add.outer\(\) was passed a numpy matrix as "
               r"(first|second) argument.")
    def test_DEPRECATED(self):
        arr = np.array([1, 2, 3])
        m = np.array([1, 2, 3]).view(np.matrix)
        self.assert_DEPRECATED(np.add.outer, args=(m, m), num=2)
        self.assert_DEPRECATED(np.add.outer, args=(arr, m))
        self.assert_DEPRECATED(np.add.outer, args=(m, arr))
        self.assert_NOT_DEPRECATED(np.add.outer, args=(arr, arr))
class FlatteningConcatenateUnsafeCast(_DeprecationTestCase):
    message = "concatenate with `axis=None` will use same-kind casting"
    def test_DEPRECATED(self):

```

```

361: (8)
362: (16)
363: (16)
364: (4)
365: (8)
366: (16)
367: (16)
368: (24)
369: (8)
370: (12)
371: (27)
372: (0)
373: (4)
374: (4)
375: (4)
376: (4)
377: (4)
378: (4)
379: (4)
380: (8)
381: (8)
382: (0)
383: (4)
384: (4)
385: (8)
386: (8)
387: (0)
388: (4)
389: (4)
390: (4)
391: (8)
392: (4)
393: (4)
394: (8)
395: (8)
396: (4)
397: (8)
398: (4)
399: (4)
400: (8)
401: (0)
402: (4)
403: (4)
404: (4)
405: (4)
406: (0)
407: (0)
408: (0)
409: (4)
410: (4)
411: (4)
412: (8)
413: (8)
414: (4)
415: (8)
416: (8)
417: (0)
418: (4)
419: (4)
420: (8)
421: (0)
422: (4)
423: (8)
424: (4)
425: (8)
426: (12)
427: (8)
428: (12)
429: (4)

            self.assert_deprecated(np.concatenate,
                                     args=(([0.], [1.])),),
                                     kwargs=dict(axis=None, out=np.empty(2, dtype=np.int64)))
    def test_not_DEPRECATED(self):
        self.assert_not_DEPRECATED(np.concatenate,
                                   args=(([0.], [1.])),),
                                   kwargs={'axis': None, 'out': np.empty(2, dtype=np.int64),
   'casting': "unsafe"})
        with assert_raises(TypeError):
            np.concatenate(([0.], [1.]), out=np.empty(2, dtype=np.int64),
                           casting="same_kind"))
class TestDeprecatedUnpickleObjectScalar(_DeprecationTestCase):
    """
    Technically, it should be impossible to create numpy object scalars,
    but there was an unpickle path that would in theory allow it. That
    path is invalid and must lead to the warning.
    """
    message = "Unpickling a scalar with object dtype is deprecated."
    def test_DEPRECATED(self):
        ctor = np.core.multiarray.scalar
        self.assert_DEPRECATED(lambda: ctor(np.dtype("O"), 1))
class TestSingleElementSignature(_DeprecationTestCase):
    message = r"The use of a length 1"
    def test_DEPRECATED(self):
        self.assert_DEPRECATED(lambda: np.add(1, 2, signature="d"))
        self.assert_DEPRECATED(lambda: np.add(1, 2, sig=(np.dtype("L"),)))
class TestCtypesGetter(_DeprecationTestCase):
    warning_cls = DeprecationWarning
    ctypes = np.array([1]).ctypes
    @pytest.mark.parametrize(
        "name", ["get_data", "get_shape", "get_strides", "get_as_parameter"])
    def test_DEPRECATED(self, name: str) -> None:
        func = getattr(self.ctypes, name)
        self.assert_DEPRECATED(lambda: func())
    @pytest.mark.parametrize(
        "name", ["data", "shape", "strides", "_as_parameter_"])
    def test_not_DEPRECATED(self, name: str) -> None:
        self.assert_not_DEPRECATED(lambda: getattr(self.ctypes, name))
PARTITION_DICT = {
    "partition method": np.arange(10).partition,
    "argpartition method": np.arange(10).argpartition,
    "partition function": lambda kth: np.partition(np.arange(10), kth),
    "argpartition function": lambda kth: np.argpartition(np.arange(10), kth),
}
@pytest.mark.parametrize("func", PARTITION_DICT.values(), ids=PARTITION_DICT)
class TestPartitionBoolIndex(_DeprecationTestCase):
    warning_cls = DeprecationWarning
    message = "Passing booleans as partition index is deprecated"
    def test_DEPRECATED(self, func):
        self.assert_DEPRECATED(lambda: func(True))
        self.assert_DEPRECATED(lambda: func([False, True]))
    def test_not_DEPRECATED(self, func):
        self.assert_not_DEPRECATED(lambda: func(1))
        self.assert_not_DEPRECATED(lambda: func([0, 1]))
class TestMachAr(_DeprecationTestCase):
    warning_cls = DeprecationWarning
    def test_DEPRECATED_module(self):
        self.assert_DEPRECATED(lambda: getattr(np.core, "MachAr"))
class TestQuantileInterpolationDeprecation(_DeprecationTestCase):
    @pytest.mark.parametrize("func",
                           [np.percentile, np.quantile, np.nanpercentile, np.nanquantile])
    def test_DEPRECATED(self, func):
        self.assert_DEPRECATED(
            lambda: func([0., 1.], 0., interpolation="linear"))
        self.assert_DEPRECATED(
            lambda: func([0., 1.], 0., interpolation="nearest"))
    @pytest.mark.parametrize("func",

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

430: (12) [np.percentile, np.quantile, np.nanpercentile, np.nanquantile])
431: (4)
432: (8)
433: (12)
434: (12)
435: (16)
436: (0)
437: (4)
438: (8)
439: (8)
440: (26)
441: (12)
442: (8)
443: (8)
444: (8)
445: (8)
446: (26)
447: (12)
448: (0)
449: (4)
450: (4)
451: (8)
452: (12)
453: (8)
454: (0)
455: (4)
456: (4)
457: (8)
458: (8)
459: (0)
460: (4)
461: (4)
462: (4)
463: (8)
464: (12)
465: (4)
466: (4)
467: (8)
468: (12)
469: (12)
470: (16)
471: (12)
472: (16)
473: (0)
474: (4)
475: (8)
476: (4)
477: (8)
478: (8)
479: (8)
480: (12)
481: (0)
482: (4)
483: (4)
484: (4)
485: (8)
486: (8)
487: (8)
488: (12)
489: (8)
490: (12)
491: (12)
492: (8)
493: (12)
494: (8)
495: (12)
496: (16)
497: (24)
498: (12)

        def test_both_passed(self, func):
            with warnings.catch_warnings():
                warnings.simplefilter("always", DeprecationWarning)
                with pytest.raises(TypeError):
                    func([0., 1.], 0., interpolation="nearest", method="nearest")
    class TestMemEventHook(_DeprecationTestCase):
        def test_mem_seteventhook(self):
            import numpy.core._multiarray_tests as ma_tests
            with pytest.warns(DeprecationWarning,
                              match='PyDataMem_SetEventHook is deprecated'):
                ma_tests.test_pydatamem_seteventhook_start()
            a = np.zeros(1000)
            del a
            break_cycles()
            with pytest.warns(DeprecationWarning,
                              match='PyDataMem_SetEventHook is deprecated'):
                ma_tests.test_pydatamem_seteventhook_end()
    class TestArrayFinalizeNone(_DeprecationTestCase):
        message = "Setting __array_finalize__ = None"
        def test_use_none_is_DEPRECATED(self):
            class NoFinalize(np.ndarray):
                __array_finalize__ = None
            self.assert_deprecated(lambda: np.array(1).view(NoFinalize))
    class TestAxisNotMAXDIMS(_DeprecationTestCase):
        message = r"Using `axis=32` \(\MAXDIMS\) is deprecated"
        def test_DEPRECATED(self):
            a = np.zeros((1,)*32)
            self.assert_deprecated(lambda: np.repeat(a, 1, axis=np.MAXDIMS))
    class TestLoadtxtParseIntsViaFloat(_DeprecationTestCase):
        message = r"loadtxt\(\): Parsing an integer via a float is deprecated.*"
        @pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
        def test_DEPRECATED_warning(self, dtype):
            with pytest.warns(DeprecationWarning, match=self.message):
                np.loadtxt(["10.5"], dtype=dtype)
        @pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
        def test_DEPRECATED_raised(self, dtype):
            with warnings.catch_warnings():
                warnings.simplefilter("error", DeprecationWarning)
                try:
                    np.loadtxt(["10.5"], dtype=dtype)
                except ValueError as e:
                    assert isinstance(e.__cause__, DeprecationWarning)
    class TestScalarConversion(_DeprecationTestCase):
        def test_float_conversion(self):
            self.assert_deprecated(float, args=(np.array([3.14]),))
        def test_behaviour(self):
            b = np.array([[3.14]])
            c = np.zeros(5)
            with pytest.warns(DeprecationWarning):
                c[0] = b
    class TestPyIntConversion(_DeprecationTestCase):
        message = r".*stop allowing conversion of out-of-bound.*"
        @pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
        def test_DEPRECATED_scalar(self, dtype):
            dtype = np.dtype(dtype)
            info = np.iinfo(dtype)
            def scalar(value, dtype):
                dtype.type(value)
            def assign(value, dtype):
                arr = np.array([0, 0, 0], dtype=dtype)
                arr[2] = value
            def create(value, dtype):
                np.array([value], dtype=dtype)
            for creation_func in [scalar, assign, create]:
                try:
                    self.assert_deprecated(
                        lambda: creation_func(info.min - 1, dtype))
                except OverflowError:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

499: (16)           pass # OverflowErrors always happened also before and are OK.
500: (12)          try:
501: (16)            self.assert_deprecated(
502: (24)              lambda: creation_func(info.max + 1, dtype))
503: (12)          except OverflowError:
504: (16)            pass # OverflowErrors always happened also before and are OK.
505: (0)           class TestDeprecatedGlobals(_DeprecationTestCase):
506: (4)             def test_type_aliases(self):
507: (8)               self.assert_deprecated(lambda: np.bool8)
508: (8)               self.assert_deprecated(lambda: np.int0)
509: (8)               self.assert_deprecated(lambda: np.uint0)
510: (8)               self.assert_deprecated(lambda: np.bytes0)
511: (8)               self.assert_deprecated(lambda: np.str0)
512: (8)               self.assert_deprecated(lambda: np.object0)
513: (0)           @pytest.mark.parametrize("name",
514: (8)             ["bool", "long", "ulong", "str", "bytes", "object"])
515: (0)           def test_future_scalar_attributes(name):
516: (4)             assert name not in dir(np) # we may want to not add them
517: (4)             with pytest.warns(FutureWarning,
518: (12)               match=f"In the future .*{name}"):
519: (8)               assert not hasattr(np, name)
520: (4)               np.dtype(name)
521: (4)               name in np.sctypeDict
522: (0)           @pytest.mark.filterwarnings("ignore:In the future:FutureWarning")
523: (0)           class TestRemovedGlobals:
524: (4)             @pytest.mark.parametrize("name",
525: (12)               ["object", "bool", "float", "complex", "str", "int"])
526: (4)             def test_attributeerror_includes_info(self, name):
527: (8)               msg = f".*\n`np.{name}` was a deprecated alias for the builtin"
528: (8)               with pytest.raises(AttributeError, match=msg):
529: (12)                 getattr(np, name)
530: (0)           class TestDeprecatedFinfo(_DeprecationTestCase):
531: (4)             def test_DEPRECATED_none(self):
532: (8)               self.assert_deprecated(np.finfo, args=(None,))
533: (0)           class TestFromnumeric(_DeprecationTestCase):
534: (4)             def test_round_(self):
535: (8)               self.assert_deprecated(lambda: np.round_(np.array([1.5, 2.5, 3.5])))
536: (4)             def test_cumproduct(self):
537: (8)               self.assert_deprecated(lambda: np.cumproduct(np.array([1, 2, 3])))
538: (4)             def test_product(self):
539: (8)               self.assert_deprecated(lambda: np.product(np.array([1, 2, 3])))
540: (4)             def test_sometrue(self):
541: (8)               self.assert_deprecated(lambda: np.sometrue(np.array([True, False])))
542: (4)             def test_alltrue(self):
543: (8)               self.assert_deprecated(lambda: np.alltrue(np.array([True, False])))
544: (0)           class TestMathAlias(_DeprecationTestCase):
545: (4)             def test_DEPRECATED_np_math(self):
546: (8)               self.assert_deprecated(lambda: np.math)
547: (4)             def test_DEPRECATED_np_lib_math(self):
548: (8)               self.assert_deprecated(lambda: np.lib.math)

```

---

File 90 - test\_dlpck.py:

```

1: (0)           import sys
2: (0)           import pytest
3: (0)           import numpy as np
4: (0)           from numpy.testing import assert_array_equal, IS_PYPY
5: (0)           class TestDLPack:
6: (4)             @pytest.mark.skipif(IS_PYPY, reason="PyPy can't get refcounts.")
7: (4)             def test_dunder_dlpck_refcount(self):
8: (8)               x = np.arange(5)
9: (8)               y = x.__dlpack__()
10: (8)              assert sys.getrefcount(x) == 3
11: (8)              del y
12: (8)              assert sys.getrefcount(x) == 2
13: (4)              def test_dunder_dlpck_stream(self):
14: (8)                x = np.arange(5)

```

```

15: (8)           x.___dlpack__(stream=None)
16: (8)           with pytest.raises(RuntimeError):
17: (12)             x.___dlpack__(stream=1)
18: (4)           def test_strides_not_multiple_of_itemsize(self):
19: (8)             dt = np.dtype([('int', np.int32), ('char', np.int8)])
20: (8)             y = np.zeros((5,), dtype=dt)
21: (8)             z = y['int']
22: (8)             with pytest.raises(BufferError):
23: (12)               np.from_dlpak(z)
24: (4)             @pytest.mark.skipif(IS_PYPY, reason="PyPy can't get refcounts.")
25: (4)             def test_from_dlpak_refcount(self):
26: (8)               x = np.arange(5)
27: (8)               y = np.from_dlpak(x)
28: (8)               assert sys.getrefcount(x) == 3
29: (8)               del y
30: (8)               assert sys.getrefcount(x) == 2
31: (4)             @pytest.mark.parametrize("dtype", [
32: (8)               np.bool_,
33: (8)               np.int8, np.int16, np.int32, np.int64,
34: (8)               np.uint8, np.uint16, np.uint32, np.uint64,
35: (8)               np.float16, np.float32, np.float64,
36: (8)               np.complex64, np.complex128
37: (4)             ])
38: (4)             def test_dtype_passthrough(self, dtype):
39: (8)               x = np.arange(5).astype(dtype)
40: (8)               y = np.from_dlpak(x)
41: (8)               assert y.dtype == x.dtype
42: (8)               assert_array_equal(x, y)
43: (4)             def test_invalid_dtype(self):
44: (8)               x = np.asarray(np.datetime64('2021-05-27'))
45: (8)               with pytest.raises(BufferError):
46: (12)                 np.from_dlpak(x)
47: (4)             def test_invalid_byte_swapping(self):
48: (8)               dt = np.dtype('=>i8').newbyteorder()
49: (8)               x = np.arange(5, dtype=dt)
50: (8)               with pytest.raises(BufferError):
51: (12)                 np.from_dlpak(x)
52: (4)             def test_non_contiguous(self):
53: (8)               x = np.arange(25).reshape((5, 5))
54: (8)               y1 = x[0]
55: (8)               assert_array_equal(y1, np.from_dlpak(y1))
56: (8)               y2 = x[:, 0]
57: (8)               assert_array_equal(y2, np.from_dlpak(y2))
58: (8)               y3 = x[1, :]
59: (8)               assert_array_equal(y3, np.from_dlpak(y3))
60: (8)               y4 = x[1]
61: (8)               assert_array_equal(y4, np.from_dlpak(y4))
62: (8)               y5 = np.diagonal(x).copy()
63: (8)               assert_array_equal(y5, np.from_dlpak(y5))
64: (4)             @pytest.mark.parametrize("ndim", range(33))
65: (4)             def test_higher_dims(self, ndim):
66: (8)               shape = (1,) * ndim
67: (8)               x = np.zeros(shape, dtype=np.float64)
68: (8)               assert shape == np.from_dlpak(x).shape
69: (4)             def test_dlpak_device(self):
70: (8)               x = np.arange(5)
71: (8)               assert x.___dlpack_device__() == (1, 0)
72: (8)               y = np.from_dlpak(x)
73: (8)               assert y.___dlpack_device__() == (1, 0)
74: (8)               z = y[::-2]
75: (8)               assert z.___dlpack_device__() == (1, 0)
76: (4)             def dlpak_deleter_exception(self):
77: (8)               x = np.arange(5)
78: (8)               _ = x.___dlpack__()
79: (8)               raise RuntimeError
80: (4)             def test_dlpak_destructor_exception(self):
81: (8)               with pytest.raises(RuntimeError):
82: (12)                 self.dlpak_deleter_exception()
83: (4)             def test_READONLY(self):

```

```

84: (8)          x = np.arange(5)
85: (8)          x.flags.writeable = False
86: (8)          with pytest.raises(BufferError):
87: (12)             x.__dlpack__()
88: (4)          def test_ndim0(self):
89: (8)             x = np.array(1.0)
90: (8)             y = np.from_dlp(x)
91: (8)             assert_array_equal(x, y)
92: (4)          def test_size1dms_arrays(self):
93: (8)             x = np.ndarray(dtype='f8', shape=(10, 5, 1), strides=(8, 80, 4),
94: (23)                           buffer=np.ones(1000, dtype=np.uint8), order='F')
95: (8)             y = np.from_dlp(x)
96: (8)             assert_array_equal(x, y)
-----
```

## File 91 - test\_dtype.py:

```

1: (0)          import sys
2: (0)          import operator
3: (0)          import pytest
4: (0)          import ctypes
5: (0)          import gc
6: (0)          import types
7: (0)          from typing import Any
8: (0)          import numpy as np
9: (0)          import numpy.dtypes
10: (0)         from numpy.core._rational_tests import rational
11: (0)         from numpy.core._multiarray_tests import create_custom_field_dtype
12: (0)         from numpy.testing import (
13: (4)             assert_, assert_equal, assert_array_equal, assert_raises, HAS_REFCOUNT,
14: (4)             IS_PYTHON, _OLD_PROMOTION)
15: (0)         from numpy.compat import pickle
16: (0)         from itertools import permutations
17: (0)         import random
18: (0)         import hypothesis
19: (0)         from hypothesis.extra import numpy as hynp
20: (0)         def assert_dtype_equal(a, b):
21: (4)             assert_equal(a, b)
22: (4)             assert_equal(hash(a), hash(b),
23: (17)                 "two equivalent types do not hash to the same value !")
24: (0)         def assert_dtype_not_equal(a, b):
25: (4)             assert_(a != b)
26: (4)             assert_(hash(a) != hash(b),
27: (12)                 "two different types hash to the same value !")
28: (0)         class TestBuiltin:
29: (4)             @pytest.mark.parametrize('t', [int, float, complex, np.int32, str, object,
30: (35)                           np.compat.unicode])
31: (4)             def test_run(self, t):
32: (8)                 """Only test hash runs at all."""
33: (8)                 dt = np.dtype(t)
34: (8)                 hash(dt)
35: (4)             @pytest.mark.parametrize('t', [int, float])
36: (4)             def test_dtype(self, t):
37: (8)                 dt = np.dtype(t)
38: (8)                 dt2 = dt.newbyteorder("<")
39: (8)                 dt3 = dt.newbyteorder(">")
40: (8)                 if dt == dt2:
41: (12)                     assert_(dt.byteorder != dt2.byteorder, "bogus test")
42: (12)                     assert_dtype_equal(dt, dt2)
43: (8)                 else:
44: (12)                     assert_(dt.byteorder != dt3.byteorder, "bogus test")
45: (12)                     assert_dtype_equal(dt, dt3)
46: (4)             def test_equivalent_dtype_hashing(self):
47: (8)                 uintp = np.dtype(np.uintp)
48: (8)                 if uintp.itemsize == 4:
49: (12)                     left = uintp
50: (12)                     right = np.dtype(np.uint32)
51: (8)                 else:
```

```

52: (12)             left = uintp
53: (12)             right = np.dtype(np.ulonglong)
54: (8)              assert_(left == right)
55: (8)              assert_(hash(left) == hash(right))
56: (4)  def test_invalid_types(self):
57: (8)      assert_raises(TypeError, np.dtype, '03')
58: (8)      assert_raises(TypeError, np.dtype, '05')
59: (8)      assert_raises(TypeError, np.dtype, '07')
60: (8)      assert_raises(TypeError, np.dtype, 'b3')
61: (8)      assert_raises(TypeError, np.dtype, 'h4')
62: (8)      assert_raises(TypeError, np.dtype, 'I5')
63: (8)      assert_raises(TypeError, np.dtype, 'e3')
64: (8)      assert_raises(TypeError, np.dtype, 'f5')
65: (8)      if np.dtype('g').itemsize == 8 or np.dtype('g').itemsize == 16:
66: (12)          assert_raises(TypeError, np.dtype, 'g12')
67: (8)      elif np.dtype('g').itemsize == 12:
68: (12)          assert_raises(TypeError, np.dtype, 'g16')
69: (8)      if np.dtype('l').itemsize == 8:
70: (12)          assert_raises(TypeError, np.dtype, 'l4')
71: (12)          assert_raises(TypeError, np.dtype, 'L4')
72: (8)      else:
73: (12)          assert_raises(TypeError, np.dtype, 'l8')
74: (12)          assert_raises(TypeError, np.dtype, 'L8')
75: (8)      if np.dtype('q').itemsize == 8:
76: (12)          assert_raises(TypeError, np.dtype, 'q4')
77: (12)          assert_raises(TypeError, np.dtype, 'Q4')
78: (8)      else:
79: (12)          assert_raises(TypeError, np.dtype, 'q8')
80: (12)          assert_raises(TypeError, np.dtype, 'Q8')
81: (4)  def test_richcompare_invalid_dtype_equality(self):
82: (8)      assert not np.dtype(np.int32) == 7, "dtype richcompare failed for =="
83: (8)      assert np.dtype(np.int32) != 7, "dtype richcompare failed for !="
84: (4)  @pytest.mark.parametrize(
85: (8)      'operation',
86: (8)      [operator.le, operator.lt, operator.ge, operator.gt])
87: (4)  def test_richcompare_invalid_dtype_comparison(self, operation):
88: (8)      with pytest.raises(TypeError):
89: (12)          operation(np.dtype(np.int32), 7)
90: (4)  @pytest.mark.parametrize("dtype",
91: (13)      ['Bool', 'Bytes0', 'Complex32', 'Complex64',
92: (14)      'Datetime64', 'Float16', 'Float32', 'Float64',
93: (14)      'Int8', 'Int16', 'Int32', 'Int64',
94: (14)      'Object0', 'Str0', 'Timedelta64',
95: (14)      'UInt8', 'UInt16', 'Uint32', 'UInt32',
96: (14)      'Uint64', 'UInt64', 'Void0',
97: (14)      "Float128", "Complex128"])
98: (4)  def test_numeric_style_types_are_invalid(self, dtype):
99: (8)      with assert_raises(TypeError):
100: (12)          np.dtype(dtype)
101: (4)  def test_remaining_dtotypes_with_bad_bytesize(self):
102: (8)      assert np.dtype("int0") is np.dtype("intp")
103: (8)      assert np.dtype("uint0") is np.dtype("uintp")
104: (8)      assert np.dtype("bool8") is np.dtype("bool")
105: (8)      assert np.dtype("bytes0") is np.dtype("bytes")
106: (8)      assert np.dtype("str0") is np.dtype("str")
107: (8)      assert np.dtype("object0") is np.dtype("object")
108: (4)  @pytest.mark.parametrize(
109: (8)      'value',
110: (8)      ['m8', 'M8', 'datetime64', 'timedelta64',
111: (9)      'i4, (2,3)f8, f4', 'a3, 3u8, (3,4)a10',
112: (9)      '>f', '<f', '=f', '|f',
113: (8)      ])
114: (4)  def test_dtype_bytes_str_equivalence(self, value):
115: (8)      bytes_value = value.encode('ascii')
116: (8)      from_bytes = np.dtype(bytes_value)
117: (8)      from_str = np.dtype(value)
118: (8)      assert_dtype_equal(from_bytes, from_str)
119: (4)  def test_dtype_from_bytes(self):
120: (8)      assert_raises(TypeError, np.dtype, b'')
```

```

121: (8) assert_raises(TypeError, np.dtype, b'|')
122: (8) assert_dtype_equal(np.dtype(bytes([0])), np.dtype('bool'))
123: (8) assert_dtype_equal(np.dtype(bytes([17])), np.dtype(object))
124: (8) assert_dtype_equal(np.dtype(b'f'), np.dtype('float32'))
125: (8) assert_raises(TypeError, np.dtype, b'\xff')
126: (8) assert_raises(TypeError, np.dtype, b's\xff')
127: (4) def test_bad_param(self):
128: (8)     assert_raises(ValueError, np.dtype,
129: (24)         {'names': ['f0', 'f1'],
130: (25)             'formats': ['i4', 'i1'],
131: (25)             'offsets': [0, 4],
132: (25)             'itemsize': 4})
133: (8)     assert_raises(ValueError, np.dtype,
134: (24)         {'names': ['f0', 'f1'],
135: (25)             'formats': ['i4', 'i1'],
136: (25)             'offsets': [0, 4],
137: (25)             'itemsize': 9}, align=True)
138: (8)     assert_raises(ValueError, np.dtype,
139: (24)         {'names': ['f0', 'f1'],
140: (25)             'formats': ['i1', 'f4'],
141: (25)             'offsets': [0, 2]}, align=True)
142: (4) def test_field_order_equality(self):
143: (8)     x = np.dtype({'names': ['A', 'B'],
144: (22)         'formats': ['i4', 'f4'],
145: (22)         'offsets': [0, 4]})
146: (8)     y = np.dtype({'names': ['B', 'A'],
147: (22)         'formats': ['i4', 'f4'],
148: (22)         'offsets': [4, 0]})
149: (8)     assert_equal(x == y, False)
150: (8)     assert np.can_cast(x, y, casting="safe")
151: (4) @pytest.mark.parametrize(
152: (8)     ["type_char", "char_size", "scalar_type"],
153: (8)     [{"U": 4, np.str_},
154: (9)     {"S": 1, np.bytes_}])
155: (4) def test_create_string_dtypes_directly(
156: (12)     self, type_char, char_size, scalar_type):
157: (8)     dtype_class = type(np.dtype(type_char))
158: (8)     dtype = dtype_class(8)
159: (8)     assert dtype.type is scalar_type
160: (8)     assert dtype.itemsize == 8*char_size
161: (4) def test_create_invalid_string_errors(self):
162: (8)     one_too_big = np.iinfo(np.intc).max + 1
163: (8)     with pytest.raises(TypeError):
164: (12)         type(np.dtype("U"))(one_too_big // 4)
165: (8)     with pytest.raises(TypeError):
166: (12)         type(np.dtype("U"))(np.iinfo(np.intp).max // 4 + 1)
167: (8)     if one_too_big < sys.maxsize:
168: (12)         with pytest.raises(TypeError):
169: (16)             type(np.dtype("S"))(one_too_big)
170: (8)     with pytest.raises(ValueError):
171: (12)         type(np.dtype("U"))(-1)
172: (0) class TestRecord:
173: (4)     def test_equivalent_record(self):
174: (8)         """Test whether equivalent record dtypes hash the same."""
175: (8)         a = np.dtype([('yo', int)])
176: (8)         b = np.dtype([('yo', int)])
177: (8)         assert_dtype_equal(a, b)
178: (4)     def test_different_names(self):
179: (8)         a = np.dtype([('yo', int)])
180: (8)         b = np.dtype([('ye', int)])
181: (8)         assert_dtype_not_equal(a, b)
182: (4)     def test_different_titles(self):
183: (8)         a = np.dtype({'names': ['r', 'b'],
184: (22)             'formats': ['u1', 'u1'],
185: (22)             'titles': ['Red pixel', 'Blue pixel']})
186: (8)         b = np.dtype({'names': ['r', 'b'],
187: (22)             'formats': ['u1', 'u1'],
188: (22)             'titles': ['RRed pixel', 'Blue pixel']})
189: (8)         assert_dtype_not_equal(a, b)

```

```

190: (4) @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
191: (4) def test_refcount_dictionary_setting(self):
192: (8)     names = ["name1"]
193: (8)     formats = ["f8"]
194: (8)     titles = ["t1"]
195: (8)     offsets = [0]
196: (8)     d = dict(names=names, formats=formats, titles=titles, offsets=offsets)
197: (8)     refcounts = {k: sys.getrefcount(i) for k, i in d.items()}
198: (8)     np.dtype(d)
199: (8)     refcounts_new = {k: sys.getrefcount(i) for k, i in d.items()}
200: (8)     assert refcounts == refcounts_new
201: (4) def test_mutate(self):
202: (8)     a = np.dtype([('yo', int)])
203: (8)     b = np.dtype([('yo', int)])
204: (8)     c = np.dtype([('ye', int)])
205: (8)     assert_dtype_equal(a, b)
206: (8)     assert_dtype_not_equal(a, c)
207: (8)     a.names = ['ye']
208: (8)     assert_dtype_equal(a, c)
209: (8)     assert_dtype_not_equal(a, b)
210: (8)     state = b._reduce_()[2]
211: (8)     a._setstate_(state)
212: (8)     assert_dtype_equal(a, b)
213: (8)     assert_dtype_not_equal(a, c)
214: (4) def test_mutate_error(self):
215: (8)     a = np.dtype("i,i")
216: (8)     with pytest.raises(ValueError, match="must replace all names at
once"):
217: (12)         a.names = ["f0"]
218: (8)     with pytest.raises(ValueError, match=".*and not string"):
219: (12)         a.names = ["f0", b"not a unicode name"]
220: (4) def test_not_lists(self):
221: (8)     """Test if an appropriate exception is raised when passing bad values
to
222: (8)     the dtype constructor.
223: (8) """
224: (8)     assert_raises(TypeError, np.dtype,
225: (22)         dict(names={'A', 'B'}, formats=['f8', 'i4']))
226: (8)     assert_raises(TypeError, np.dtype,
227: (22)         dict(names=['A', 'B'], formats={'f8', 'i4'}))
228: (4) def test_aligned_size(self):
229: (8)     dt = np.dtype('i4, i1', align=True)
230: (8)     assert_equal(dt.itemsize, 8)
231: (8)     dt = np.dtype([('f0', 'i4'), ('f1', 'i1')], align=True)
232: (8)     assert_equal(dt.itemsize, 8)
233: (8)     dt = np.dtype({'names': ['f0', 'f1'],
234: (23)         'formats': ['i4', 'u1'],
235: (23)         'offsets': [0, 4]}, align=True)
236: (8)     assert_equal(dt.itemsize, 8)
237: (8)     dt = np.dtype({'f0': ('i4', 0), 'f1': ('u1', 4)}, align=True)
238: (8)     assert_equal(dt.itemsize, 8)
239: (8)     dt1 = np.dtype([('f0', 'i4'),
240: (23)         ('f1', [('f1', 'i1'), ('f2', 'i4'), ('f3', 'i1')]),
241: (23)         ('f2', 'i1')], align=True)
242: (8)     assert_equal(dt1.itemsize, 20)
243: (8)     dt2 = np.dtype({'names': ['f0', 'f1', 'f2'],
244: (23)         'formats': ['i4',
245: (34)             [('f1', 'i1'), ('f2', 'i4'), ('f3', 'i1')],
246: (34)             'i1'],
247: (23)             'offsets': [0, 4, 16]}, align=True)
248: (8)     assert_equal(dt2.itemsize, 20)
249: (8)     dt3 = np.dtype({'f0': ('i4', 0),
250: (23)         'f1': ([('f1', 'i1'), ('f2', 'i4'), ('f3', 'i1')], 4),
251: (23)         'f2': ('i1', 16)}, align=True)
252: (8)     assert_equal(dt3.itemsize, 20)
253: (8)     assert_equal(dt1, dt2)
254: (8)     assert_equal(dt2, dt3)
255: (8)     dt1 = np.dtype([('f0', 'i4'),
256: (23)         ('f1', [('f1', 'i1'), ('f2', 'i4'), ('f3', 'i1')]),
```

```

257: (23)                                     ('f2', 'i1')], align=False)
258: (8)          assert_equal(dt1.itemsize, 11)
259: (8)          dt2 = np.dtype({'names':['f0', 'f1', 'f2'],
260: (23)                      'formats':['i4',
261: (34)                          [('f1', 'i1'), ('f2', 'i4'), ('f3', 'i1')],
262: (34)                          'i1'],
263: (23)                      'offsets':[0, 4, 10]}, align=False)
264: (8)          assert_equal(dt2.itemsize, 11)
265: (8)          dt3 = np.dtype({'f0': ('i4', 0),
266: (23)                      'f1': ([('f1', 'i1'), ('f2', 'i4'), ('f3', 'i1')], 4),
267: (23)                      'f2': ('i1', 10)}, align=False)
268: (8)          assert_equal(dt3.itemsize, 11)
269: (8)          assert_equal(dt1, dt2)
270: (8)          assert_equal(dt2, dt3)
271: (8)          dt1 = np.dtype([('a', '|i1'),
272: (24)                      ('b', [('f0', '<i2'),
273: (24)                          ('f1', '<f4')], 2)], align=True)
274: (8)          assert_equal(dt1.descr, [('a', '|i1'), ('', '|V3'),
275: (33)                          ('b', [('f0', '<i2'), ('', '|V2'),
276: (33)                          ('f1', '<f4')], (2,))])
277: (4)          def test_union_struct(self):
278: (8)              dt = np.dtype({'names':['f0', 'f1', 'f2'], 'formats':['<u4', '<u2',
279: (<u2'],
280: (24)                      'offsets':[0, 0, 2]}, align=True)
281: (8)              assert_equal(dt.itemsize, 4)
282: (8)              a = np.array([3], dtype='<u4').view(dt)
283: (8)              a['f1'] = 10
284: (8)              a['f2'] = 36
285: (8)              assert_equal(a['f0'], 10 + 36*256*256)
286: (8)              dt = np.dtype({'names':['f0', 'f1', 'f2'], 'formats':['<u4', '<u2',
287: (<u2'],
288: (24)                      'offsets':[4, 0, 2]}, align=True)
289: (8)              assert_equal(dt.itemsize, 8)
290: (24)              dt2 = np.dtype({'names':['f2', 'f0', 'f1'],
291: (8)                      'formats':['<u4', '<u2', '<u2'],
292: (8)                      'offsets':[4, 0, 2]}, align=True)
293: (8)              vals = [(0, 1, 2), (3, 2**15-1, 4)]
294: (8)              vals2 = [(0, 1, 2), (3, 2**15-1, 4)]
295: (8)              a = np.array(vals, dt)
296: (8)              b = np.array(vals2, dt2)
297: (8)              assert_equal(a.astype(dt2), b)
298: (8)              assert_equal(b.astype(dt), a)
299: (8)              assert_equal(a.view(dt2), b)
300: (16)             assert_raises(TypeError, np.dtype,
301: (17)                         {'names':['f0', 'f1'],
302: (17)                         'formats':['0', 'i1'],
303: (8)                         'offsets':[0, 2]})
304: (16)             assert_raises(TypeError, np.dtype,
305: (17)                         {'names':['f0', 'f1'],
306: (17)                         'formats':['i4', '0'],
307: (8)                         'offsets':[0, 3]})
308: (16)             assert_raises(TypeError, np.dtype,
309: (17)                         {'names':['f0', 'f1'],
310: (17)                         'formats':[['a', '0']], 'i1'],
311: (8)                         'offsets':[0, 2]})
312: (16)             assert_raises(TypeError, np.dtype,
313: (17)                         {'names':['f0', 'f1'],
314: (17)                         'formats':[['i4', [('a', '0')]]],
315: (8)                         'offsets':[0, 3]})
316: (23)             dt = np.dtype({'names':['f0', 'f1'],
317: (23)                         'formats':[['i1', '0']],
318: (4)                         'offsets':[np.dtype('intp').itemsize, 0]})

@pytest.mark.parametrize(["obj", "dtype", "expected"],
319: (8)                         [([], ("2)f4,"), np.empty((0, 2), dtype="f4")),
320: (9)                         (3, "(3)f4,"), [3, 3, 3]),
321: (9)                         (np.float64(2), "(2)f4,"), [2, 2]),
322: (9)                         (((0, 1), (1, 2)), ((2,),)), '(2,2)f4', None),
323: (9)                         (["1", "2"], "(2)i,", None)])

```

```

324: (4)
325: (8)
326: (8)
327: (8)
328: (12)
329: (12)
330: (16)
331: (8)
332: (4)
333: (8)
334: (8)
335: (35)
336: (35)
337: (4)
338: (8)
339: (8)
340: (8)
341: (8)
342: (4)
343: (8)
344: (8)
345: (24)
346: (8)
347: (8)
348: (8)
349: (4)
350: (8)
351: (8)
352: (8)
353: (12)
354: (4)
355: (8)
356: (12)
357: (29)
358: (8)
359: (8)
360: (8)
361: (8)
362: (8)
363: (8)
364: (4)
365: (8)
366: (8)
367: (8)
368: (8)
369: (8)
370: (8)
371: (8)
372: (8)
373: (4)
374: (4)
375: (8)
376: (12)
377: (8)
378: (8)
379: (8)
380: (12)
381: (12)
382: (16)
383: (16)
384: (16)
385: (16)
386: (16)
387: (12)
388: (8)
389: (8)
390: (8)
391: (8)
392: (12)

        def test_subarray_list(self, obj, dtype, expected):
            dtype = np.dtype(dtype)
            res = np.array(obj, dtype=dtype)
            if expected is None:
                expected = np.empty(len(obj), dtype=dtype)
                for i in range(len(expected)):
                    expected[i] = obj[i]
            assert_array_equal(res, expected)
        def test_comma_datetime(self):
            dt = np.dtype('M8[D],datetime64[Y],i8')
            assert_equal(dt, np.dtype([('f0', 'M8[D]',),
                                      ('f1', 'datetime64[Y]',),
                                      ('f2', 'i8')]))
        def test_from_dictproxy(self):
            dt = np.dtype({'names': ['a', 'b'], 'formats': ['i4', 'f4']})
            assert_dtype_equal(dt, np.dtype(dt.fields))
            dt2 = np.dtype((np.void, dt.fields))
            assert_equal(dt2.fields, dt.fields)
        def test_from_dict_with_zero_width_field(self):
            dt = np.dtype([('val1', np.float32, (0,)), ('val2', int)])
            dt2 = np.dtype({'names': ['val1', 'val2'],
                           'formats': [(np.float32, (0,)), int]})
            assert_dtype_equal(dt, dt2)
            assert_equal(dt.fields['val1'][0].itemsize, 0)
            assert_equal(dt.itemsize, dt.fields['val2'][0].itemsize)
        def test_bool_commastring(self):
            d = np.dtype('?,?,?') # raises?
            assert_equal(len(d.names), 3)
            for n in d.names:
                assert_equal(d.fields[n][0], np.dtype('?'))
        def test_nonint_offsets(self):
            def make_dtype(off):
                return np.dtype({'names': ['A'], 'formats': ['i4'],
                               'offsets': [off]})

                assert_raises(TypeError, make_dtype, 'ASD')
                assert_raises(OverflowError, make_dtype, 2**70)
                assert_raises(TypeError, make_dtype, 2.3)
                assert_raises(ValueError, make_dtype, -10)
                dt = make_dtype(np.uint32(0))
                np.zeros(1, dtype=dt)[0].item()
            def test_fields_by_index(self):
                dt = np.dtype([('a', np.int8), ('b', np.float32, 3)])
                assert_dtype_equal(dt[0], np.dtype(np.int8))
                assert_dtype_equal(dt[1], np.dtype((np.float32, 3)))
                assert_dtype_equal(dt[-1], dt[1])
                assert_dtype_equal(dt[-2], dt[0])
                assert_raises(IndexError, lambda: dt[-3])
                assert_raises(TypeError, operatorgetitem, dt, 3.0)
                assert_equal(dt[1], dt[np.int8(1)])
            @pytest.mark.parametrize('align_flag',[False, True])
            def test_multifield_index(self, align_flag):
                dt = np.dtype([
                    ('title', 'col1'), '<U20'), ('A', '<f8'), ('B', '<f8')
                ], align=align_flag)
                dt_sub = dt[['B', 'col1']]
                assert_equal(
                    dt_sub,
                    np.dtype({
                        'names': ['B', 'col1'],
                        'formats': ['<f8', '<U20'],
                        'offsets': [88, 0],
                        'titles': [None, 'title'],
                        'itemsize': 96
                    })
                )
                assert_equal(dt_sub.isalignedstruct, align_flag)
                dt_sub = dt[['B']]
                assert_equal(
                    dt_sub,

```

```

393: (12)           np.dtype({
394: (16)             'names': ['B'],
395: (16)             'formats': ['<f8'],
396: (16)             'offsets': [88],
397: (16)             'itemsize': 96
398: (12)         })
399: (8)       )
400: (8)     assert_equal(dt_sub.isalignedstruct, align_flag)
401: (8)     dt_sub = dt[[]]
402: (8)     assert_equal(
403: (12)       dt_sub,
404: (12)       np.dtype({
405: (16)         'names': [],
406: (16)         'formats': [],
407: (16)         'offsets': [],
408: (16)         'itemsize': 96
409: (12)     }))
410: (8)   )
411: (8)     assert_equal(dt_sub.isalignedstruct, align_flag)
412: (8)     assert_raises(TypeError, operatorgetitem, dt, ())
413: (8)     assert_raises(TypeError, operatorgetitem, dt, [1, 2, 3])
414: (8)     assert_raises(TypeError, operatorgetitem, dt, ['col1', 2])
415: (8)     assert_raises(KeyError, operatorgetitem, dt, ['fake'])
416: (8)     assert_raises(KeyError, operatorgetitem, dt, ['title'])
417: (8)     assert_raises(ValueError, operatorgetitem, dt, ['col1', 'col1'])
418: (4)   def test_partial_dict(self):
419: (8)     assert_raises(ValueError, np.dtype,
420: (16)       {'formats': ['i4', 'i4'], 'f0': ('i4', 0), 'f1':('i4', 4)})
421: (4)   def test_fieldless_views(self):
422: (8)     a = np.zeros(2, dtype={'names':[], 'formats':[], 'offsets':[],
423: (31)       'itemsize':8})
424: (8)     assert_raises(ValueError, a.view, np.dtype([]))
425: (8)     d = np.dtype((np.dtype([]), 10))
426: (8)     assert_equal(d.shape, (10,))
427: (8)     assert_equal(d.itemsize, 0)
428: (8)     assert_equal(d.base, np.dtype([]))
429: (8)     arr = np.fromiter(((() for i in range(10)), []))
430: (8)     assert_equal(arr.dtype, np.dtype([]))
431: (8)     assert_raises(ValueError, np.frombuffer, b'', dtype[])
432: (8)     assert_equal(np.frombuffer(b'', dtype[], count=2),
433: (21)       np.empty(2, dtype[]))
434: (8)     assert_raises(ValueError, np.dtype, ([], 'f8'))
435: (8)     assert_raises(ValueError, np.zeros(1, dtype='i4').view, [])
436: (8)     assert_equal(np.zeros(2, dtype[]) == np.zeros(2, dtype[]),
437: (21)       np.ones(2, dtype=bool))
438: (8)     assert_equal(np.zeros((1, 2), dtype[]) == a,
439: (21)       np.ones((1, 2), dtype=bool))
440: (4)   def test_nonstructured_with_object(self):
441: (8)     arr = np.recarray((0,), dtype="O")
442: (8)     assert arr.dtype.names is None # no fields
443: (8)     assert arr.dtype.hasobject # but claims to contain objects
444: (8)     del arr # the deletion failed previously.
445: (0) class TestSubarray:
446: (4)   def test_single_subarray(self):
447: (8)     a = np.dtype((int, (2)))
448: (8)     b = np.dtype((int, (2,)))
449: (8)     assert_dtype_equal(a, b)
450: (8)     assert_equal(type(a.subdtype[1]), tuple)
451: (8)     assert_equal(type(b.subdtype[1]), tuple)
452: (4)   def test_equivalent_record(self):
453: (8)     """Test whether equivalent subarray dtypes hash the same."""
454: (8)     a = np.dtype((int, (2, 3)))
455: (8)     b = np.dtype((int, (2, 3)))
456: (8)     assert_dtype_equal(a, b)
457: (4)   def test_nonequivalent_record(self):
458: (8)     """Test whether different subarray dtypes hash differently."""
459: (8)     a = np.dtype((int, (2, 3)))
460: (8)     b = np.dtype((int, (3, 2)))
461: (8)     assert_dtype_not_equal(a, b)

```

```

462: (8)          a = np.dtype((int, (2, 3)))
463: (8)          b = np.dtype((int, (2, 2)))
464: (8)          assert_dtype_not_equal(a, b)
465: (8)          a = np.dtype((int, (1, 2, 3)))
466: (8)          b = np.dtype((int, (1, 2)))
467: (8)          assert_dtype_not_equal(a, b)
468: (4)          def test_shape_equal(self):
469: (8)              """Test some data types that are equal"""
470: (8)              assert_dtype_equal(np.dtype('f8'), np.dtype(('f8', tuple())))
471: (8)              with pytest.warns(FutureWarning):
472: (12)                  assert_dtype_equal(np.dtype('f8'), np.dtype('f8', 1))
473: (8)                  assert_dtype_equal(np.dtype((int, 2)), np.dtype((int, (2,))))
474: (8)                  assert_dtype_equal(np.dtype('<f4', (3, 2)), np.dtype('<f4', (3,
2)))
475: (8)                  d = [(['a', 'f4', (1, 2)], ('b', 'f8', (3, 1))], (3, 2))
476: (8)                  assert_dtype_equal(np.dtype(d), np.dtype(d))
477: (4)          def test_shape_simple(self):
478: (8)              """Test some simple cases that shouldn't be equal"""
479: (8)              assert_dtype_not_equal(np.dtype('f8'), np.dtype('f8', (1,)))
480: (8)              assert_dtype_not_equal(np.dtype('f8', (1,)), np.dtype('f8', (1,
1)))
481: (8)              assert_dtype_not_equal(np.dtype('f4', (3, 2)), np.dtype('f4', (2,
3)))
482: (4)          def test_shape_monster(self):
483: (8)              """Test some more complicated cases that shouldn't be equal"""
484: (8)              assert_dtype_not_equal(
485: (12)                  np.dtype([(['a', 'f4', (2, 1)], ('b', 'f8', (1, 3))], (2, 2)),
486: (12)                  np.dtype([(['a', 'f4', (1, 2)], ('b', 'f8', (1, 3))], (2, 2)))
487: (8)              assert_dtype_not_equal(
488: (12)                  np.dtype([(['a', 'f4', (2, 1)], ('b', 'f8', (1, 3))], (2, 2)),
489: (12)                  np.dtype([(['a', 'f4', (2, 1)], ('b', 'i8', (1, 3))], (2, 2)))
490: (8)              assert_dtype_not_equal(
491: (12)                  np.dtype([(['a', 'f4', (2, 1)], ('b', 'f8', (1, 3))], (2, 2)),
492: (12)                  np.dtype([(['e', 'f8', (1, 3)], ('d', 'f4', (2, 1))], (2, 2)))
493: (8)              assert_dtype_not_equal(
494: (12)                  np.dtype([(['a', [('a', 'i4', 6)], (2, 1)], ('b', 'f8', (1, 3))],
495: (12)                  np.dtype([(['a', [('a', 'u4', 6)], (2, 1)], ('b', 'f8', (1, 3))],
(2, 2))), (2, 2)))
496: (4)          def test_shape_sequence(self):
497: (8)              a = np.array([1, 2, 3], dtype=np.int16)
498: (8)              l = [1, 2, 3]
499: (8)              dt = np.dtype([('a', 'f4', a)])
500: (8)              assert_(isinstance(dt['a'].shape, tuple))
501: (8)              assert_(isinstance(dt['a'].shape[0], int))
502: (8)              dt = np.dtype([('a', 'f4', 1)])
503: (8)              assert_(isinstance(dt['a'].shape, tuple))
504: (8)              class IntLike:
505: (12)                  def __index__(self):
506: (16)                      return 3
507: (12)                  def __int__(self):
508: (16)                      return 3
509: (8)              dt = np.dtype([('a', 'f4', IntLike())])
510: (8)              assert_(isinstance(dt['a'].shape, tuple))
511: (8)              assert_(isinstance(dt['a'].shape[0], int))
512: (8)              dt = np.dtype([('a', 'f4', (IntLike(),))])
513: (8)              assert_(isinstance(dt['a'].shape, tuple))
514: (8)              assert_(isinstance(dt['a'].shape[0], int))
515: (4)          def test_shape_matches_ndim(self):
516: (8)              dt = np.dtype([('a', 'f4', ())])
517: (8)              assert_equal(dt['a'].shape, ())
518: (8)              assert_equal(dt['a'].ndim, 0)
519: (8)              dt = np.dtype([('a', 'f4')])
520: (8)              assert_equal(dt['a'].shape, ())
521: (8)              assert_equal(dt['a'].ndim, 0)
522: (8)              dt = np.dtype([('a', 'f4', 4)])
523: (8)              assert_equal(dt['a'].shape, (4,))
524: (8)              assert_equal(dt['a'].ndim, 1)
525: (8)              dt = np.dtype([('a', 'f4', (1, 2, 3))])

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

526: (8) assert_equal(dt['a'].shape, (1, 2, 3))
527: (8) assert_equal(dt['a'].ndim, 3)
528: (4) def test_shape_invalid(self):
529: (8)     max_int = np.iinfo(np.intc).max
530: (8)     max_intp = np.iinfo(np.intp).max
531: (8)     assert_raises(ValueError, np.dtype, [('a', 'f4', max_int // 4 + 1)])
532: (8)     assert_raises(ValueError, np.dtype, [('a', 'f4', max_int + 1)])
533: (8)     assert_raises(ValueError, np.dtype, [('a', 'f4', (max_int, 2))])
534: (8)     assert_raises(ValueError, np.dtype, [('a', 'f4', max_intp + 1)])
535: (8)     assert_raises(ValueError, np.dtype, [('a', 'f4', -1)])
536: (8)     assert_raises(ValueError, np.dtype, [('a', 'f4', (-1, -1))])
537: (4) def test_alignment(self):
538: (8)     t1 = np.dtype('(1,)i4', align=True)
539: (8)     t2 = np.dtype('2i4', align=True)
540: (8)     assert_equal(t1.alignment, t2.alignment)
541: (4) def test_aligned_empty(self):
542: (8)     dt = np.dtype([], align=True)
543: (8)     assert dt == np.dtype([])
544: (8)     dt = np.dtype({"names": [], "formats": [], "itemsize": 0}, align=True)
545: (8)     assert dt == np.dtype([])
546: (4) def test_subarray_base_item(self):
547: (8)     arr = np.ones(3, dtype=[("f", "i", 3)])
548: (8)     assert arr["f"].base is arr
549: (8)     item = arr.item(0)
550: (8)     assert type(item) is tuple and len(item) == 1
551: (8)     assert item[0].base is arr
552: (4) def test_subarray_cast_copies(self):
553: (8)     arr = np.ones(3, dtype=[("f", "i", 3)])
554: (8)     cast = arr.astype(object)
555: (8)     for fields in cast:
556: (12)         assert type(fields) == tuple and len(fields) == 1
557: (12)         subarr = fields[0]
558: (12)         assert subarr.base is None
559: (12)         assert subarr.flags.owndata
560: (0) def iter_struct_object_dtotypes():
561: (4) """
562: (4) Iterates over a few complex dtypes and object pattern which
563: (4) fill the array with a given object (defaults to a singleton).
564: (4) Yields
565: (4) -----
566: (4) dtype : dtype
567: (4) pattern : tuple
568: (8)     Structured tuple for use with `np.array` .
569: (4) count : int
570: (8)     Number of objects stored in the dtype.
571: (4) singleton : object
572: (8)     A singleton object. The returned pattern is constructed so that
573: (8)     all objects inside the datatype are set to the singleton.
574: (4) """
575: (4) obj = object()
576: (4) dt = np.dtype([('b', 'O', (2, 3))])
577: (4) p = ([[obj] * 3] * 2,)
578: (4) yield pytest.param(dt, p, 6, obj, id=<subarray>)
579: (4) dt = np.dtype([('a', 'i4'), ('b', 'O', (2, 3))])
580: (4) p = (0, [[obj] * 3] * 2)
581: (4) yield pytest.param(dt, p, 6, obj, id=<subarray in field>)
582: (4) dt = np.dtype([('a', 'i4'),
583: (19)             ('b', [(['ba', 'O'), ('bb', 'i1')], (2, 3))])
584: (4) p = (0, [[(obj, 0)] * 3] * 2)
585: (4) yield pytest.param(dt, p, 6, obj, id=<structured subarray 1>)
586: (4) dt = np.dtype([('a', 'i4'),
587: (19)             ('b', [(['ba', 'O'), ('bb', 'O')], (2, 3))])
588: (4) p = (0, [[(obj, obj)] * 3] * 2)
589: (4) yield pytest.param(dt, p, 12, obj, id=<structured subarray 2>)
590: (0) @pytest.mark.skipif(
591: (4)     sys.version_info >= (3, 12),
592: (4)     reason="Python 3.12 has immortal refcounts, this test will no longer "
593: (11)     "work. See gh-23986"
594: (0) )

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

595: (0)
596: (0)
597: (4)
598: (4)
599: (4)
600: (4)
601: (29)
602: (4)
603: (8)
604: (13)
605: (8)
606: (8)
607: (4)
608: (45)
609: (8)
610: (8)
611: (8)
612: (8)
613: (8)
614: (8)
615: (8)
616: (8)
617: (8)
618: (4)
619: (29)
620: (4)
621: (8)
622: (8)
623: (8)
624: (8)
625: (8)
626: (8)
627: (8)
628: (8)
629: (8)
630: (8)
631: (8)
632: (8)
633: (8)
634: (8)
635: (4)
636: (29)
637: (4)
638: (8)
639: (8)
640: (9)
641: (9)
642: (9)
643: (4)
644: (40)
645: (8)
646: (8)
647: (8)
648: (8)
649: (8)
650: (8)
651: (8)
652: (8)
653: (8)
654: (8)
655: (8)
656: (8)
657: (8)
658: (8)
659: (8)
660: (8)
661: (8)
662: (4)
663: (29)

@pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
class TestStructuredObjectRefCounting:
    """These tests cover various uses of complicated structured types which
    include objects and thus require reference counting.
    """
    @pytest.mark.parametrize(['dt', 'pat', 'count', 'singleton'],
                           iter_struct_object_dt_types())
    @pytest.mark.parametrize(['creation_func', 'creation_obj'], [
        pytest.param(np.empty, None,
                     marks=pytest.mark.skip("unreliable due to python's behaviour")),
        (np.ones, 1),
        (np.zeros, 0)])
    def test_structured_object_create_delete(self, dt, pat, count, singleton,
   creation_func, creation_obj):
        """Structured object reference counting in creation and deletion"""
        gc.collect()
        before = sys.getrefcount(creation_obj)
        arr = creation_func(3, dt)
        now = sys.getrefcount(creation_obj)
        assert now - before == count * 3
        del arr
        now = sys.getrefcount(creation_obj)
        assert now == before

    @pytest.mark.parametrize(['dt', 'pat', 'count', 'singleton'],
                           iter_struct_object_dt_types())
    def test_structured_object_item_setting(self, dt, pat, count, singleton):
        """Structured object reference counting for simple item setting"""
        one = 1
        gc.collect()
        before = sys.getrefcount(singleton)
        arr = np.array([pat] * 3, dt)
        assert sys.getrefcount(singleton) - before == count * 3
        before2 = sys.getrefcount(one)
        arr[...] = one
        after2 = sys.getrefcount(one)
        assert after2 - before2 == count * 3
        del arr
        gc.collect()
        assert sys.getrefcount(one) == before2
        assert sys.getrefcount(singleton) == before

    @pytest.mark.parametrize(['dt', 'pat', 'count', 'singleton'],
                           iter_struct_object_dt_types())
    @pytest.mark.parametrize(['shape', 'index', 'items_changed'],
                           [((), ([0, 2]), 2),
                            ((3, 2), ([0, 2], slice(None)), 4),
                            ((3, 2), ([0, 2], [1]), 2),
                            ((3,), ([True, False, True]), 2)])
    def test_structured_object_indexing(self, shape, index, items_changed,
                                       dt, pat, count, singleton):
        """Structured object reference counting for advanced indexing."""
        val0 = -4
        val1 = -5
        arr = np.full(shape, val0, dt)
        gc.collect()
        before_val0 = sys.getrefcount(val0)
        before_val1 = sys.getrefcount(val1)
        part = arr[index]
        after_val0 = sys.getrefcount(val0)
        assert after_val0 - before_val0 == count * items_changed
        del part
        arr[index] = val1
        gc.collect()
        after_val0 = sys.getrefcount(val0)
        after_val1 = sys.getrefcount(val1)
        assert before_val0 - after_val0 == count * items_changed
        assert after_val1 - before_val1 == count * items_changed

    @pytest.mark.parametrize(['dt', 'pat', 'count', 'singleton'],
                           iter_struct_object_dt_types())

```

```

664: (4)
singleton):
665: (8)         """Structured object reference counting for specialized functions.
666: (8)         The older functions such as take and repeat use different code paths
667: (8)         then item setting (when writing this).
668: (8)
669: (8)         """
670: (8)         indices = [0, 1]
671: (8)         arr = np.array([pat] * 3, dt)
672: (8)         gc.collect()
673: (8)         before = sys.getrefcount(singleton)
674: (8)         res = arr.take(indices)
675: (8)         after = sys.getrefcount(singleton)
676: (8)         assert after - before == count * 2
677: (8)         new = res.repeat(10)
678: (8)         gc.collect()
679: (8)         after_repeat = sys.getrefcount(singleton)
680: (0)         assert after_repeat - after == count * 2 * 10
681: (4) class TestStructuredDtypeSparseFields:
682: (4)     """Tests subarray fields which contain sparse dtypes so that
683: (4)     not all memory is used by the dtype work. Such dtype's should
684: (4)     leave the underlying memory unchanged.
685: (4)     """
686: (29)     dtype = np.dtype([('a', {'names':['aa', 'ab'], 'formats':['f', 'f'],
687: (4)             'offsets':[0, 4]}, (2, 3))])
688: (36)     sparse_dtype = np.dtype([('a', {'names':['ab'], 'formats':['f'],
689: (4)                 'offsets':[4]}, (2, 3))])
690: (8)     def test_sparse_field_assignment(self):
691: (8)         arr = np.zeros(3, self.dtype)
692: (8)         sparse_arr = arr.view(self.sparse_dtype)
693: (8)         sparse_arr[...] = np.finfo(np.float32).max
694: (8)         assert_array_equal(arr["a"]["aa"], np.zeros((3, 2, 3)))
695: (8)     def test_sparse_field_assignment_fancy(self):
696: (8)         arr = np.zeros(3, self.dtype)
697: (8)         sparse_arr = arr.view(self.sparse_dtype)
698: (8)         sparse_arr[[0, 1, 2]] = np.finfo(np.float32).max
699: (0)         assert_array_equal(arr["a"]["aa"], np.zeros((3, 2, 3)))
700: (4) class TestMonsterType:
701: (4)     """Test deeply nested subtypes."""
702: (8)     def test1(self):
703: (12)         simple1 = np.dtype({'names': ['r', 'b'], 'formats': ['u1', 'u1'],
704: (8)             'titles': ['Red pixel', 'Blue pixel']})
705: (12)         a = np.dtype([('yo', int), ('ye', simple1),
706: (8)             ('yi', np.dtype((int, (3, 2))))])
707: (12)         b = np.dtype([('yo', int), ('ye', simple1),
708: (8)             ('yi', np.dtype((int, (3, 2))))])
709: (8)         assert_dtype_equal(a, b)
710: (12)         c = np.dtype([('yo', int), ('ye', simple1),
711: (8)             ('yi', np.dtype((a, (3, 2))))])
712: (12)         d = np.dtype([('yo', int), ('ye', simple1),
713: (8)             ('yi', np.dtype((a, (3, 2))))])
714: (4)         assert_dtype_equal(c, d)
715: (4)         @pytest.mark.skipif(IS_PYSTON, reason="Pyston disables recursion
716: (8)             checking")
717: (8)         def test_list_recursion(self):
718: (8)             l = list()
719: (12)             l.append('f', l)
720: (4)             with pytest.raises(RecursionError):
721: (8)                 np.dtype(l)
722: (8)             @pytest.mark.skipif(IS_PYSTON, reason="Pyston disables recursion
723: (8)                 checking")
724: (12)             def test_tuple_recursion(self):
725: (8)                 d = np.int32
726: (12)                 for i in range(100000):
727: (8)                     d = (d, (1,))
728: (4)                     with pytest.raises(RecursionError):
729: (8)                         np.dtype(d)
730: (4)                     @pytest.mark.skipif(IS_PYSTON, reason="Pyston disables recursion
731: (8)                         checking")
732: (4)                     def test_dict_recursion(self):

```



```

797: (23)
798: (34)
799: (8)
800: (20)
801: (20)
802: (20)
803: (20)
804: (31)
805: (20)
806: (8)
807: (24)
808: (24)
809: (8)
810: (20)
811: (20)
812: (20)
813: (20)
814: (20)
815: (8)
816: (8)
817: (20)
818: (4)
819: (8)
820: (32)
821: (23)
822: (35)
823: (8)
824: (21)
825: (21)
826: (21)
827: (21)
828: (8)
'u1'],
829: (24)
830: (24)
831: (24)
832: (8)
833: (20)
834: (20)
835: (20)
836: (4)
837: (8)
838: (23)
839: (23)
840: (23)
841: (34)
842: (8)
843: (20)
844: (20)
845: (20)
846: (20)
847: (32)
848: (20)
849: (8)
850: (24)
851: (24)
852: (24)
853: (8)
854: (20)
855: (20)
856: (20)
857: (20)
858: (20)
859: (4)
860: (8)
861: (8)
862: (20)
863: (4)
864: (8)

    'titles': ['Color', 'Red pixel',
                'Green pixel', 'Blue pixel']])}

    assert_equal(str(dt),
                 "{'names': ['rgba', 'r', 'g', 'b'],
                  'formats': ['<u4', 'u1', 'u1', 'u1'],
                  'offsets': [0, 0, 1, 2],
                  'titles': ['Color', 'Red pixel',
                             'Green pixel', 'Blue pixel'],
                  'itemsize': 4}}")

dt = np.dtype({'names': ['r', 'b'], 'formats': ['u1', 'u1'],
               'offsets': [0, 2],
               'titles': ['Red pixel', 'Blue pixel']})

assert_equal(str(dt),
             "{'names': ['r', 'b'],
              'formats': ['u1', 'u1'],
              'offsets': [0, 2],
              'titles': ['Red pixel', 'Blue pixel'],
              'itemsize': 3}}")

dt = np.dtype([('a', '<m8[D)'), ('b', '<M8[us]'))]
assert_equal(str(dt),
             "[('a', '<m8[D)'), ('b', '<M8[us]')]]")

def test_repr_structured(self):
    dt = np.dtype([('top', [('tiles', ('>f4', (64, 64)), (1,)),
                           ('rtile', '>f4', (64, 36))], (3,)),
                  ('bottom', [('bleft', ('>f4', (8, 64)), (1,)),
                              ('bright', '>f4', (8, 36))])])

    assert_equal(repr(dt),
                 "dtype([('top', [('tiles', ('>f4', (64, 64)), (1,)),
                               ('rtile', '>f4', (64, 36))], (3,)),
                         ('bottom', [('bleft', ('>f4', (8, 64)), (1,)),
                                     ('bright', '>f4', (8, 36))])])")

    dt = np.dtype({'names': ['r', 'g', 'b'], 'formats': ['u1', 'u1',
'u1'],
                   'offsets': [0, 1, 2],
                   'titles': ['Red pixel', 'Green pixel', 'Blue pixel'],
                   'align=True'})

    assert_equal(repr(dt),
                 "dtype([('Red pixel', 'r'), ('u1'),
                         ('Green pixel', 'g'), ('u1'),
                         ('Blue pixel', 'b'), ('u1')], align=True)")

def test_repr_structured_not_packed(self):
    dt = np.dtype({'names': ['rgba', 'r', 'g', 'b'],
                  'formats': ['<u4', 'u1', 'u1', 'u1'],
                  'offsets': [0, 0, 1, 2],
                  'titles': ['Color', 'Red pixel',
                             'Green pixel', 'Blue pixel']],
                  'align=True')

    assert_equal(repr(dt),
                 "dtype({'names': ['rgba', 'r', 'g', 'b'],
                         'formats': ['<u4', 'u1', 'u1', 'u1'],
                         'offsets': [0, 0, 1, 2],
                         'titles': ['Color', 'Red pixel',
                                    'Green pixel', 'Blue pixel'],
                         'itemsize': 4}, align=True)")

    dt = np.dtype({'names': ['r', 'b'], 'formats': ['u1', 'u1'],
                  'offsets': [0, 2],
                  'titles': ['Red pixel', 'Blue pixel'],
                  'itemsize': 4})

    assert_equal(repr(dt),
                 "dtype({'names': ['r', 'b'],
                         'formats': ['u1', 'u1'],
                         'offsets': [0, 2],
                         'titles': ['Red pixel', 'Blue pixel'],
                         'itemsize': 4}))"

def test_repr_structured_datetime(self):
    dt = np.dtype([('a', '<M8[D)'), ('b', '<m8[us]'))]
    assert_equal(repr(dt),
                 "dtype([('a', '<M8[D)'), ('b', '<m8[us]')])")

def test_repr_str_subarray(self):
    dt = np.dtype('<i2', (1,)))

```

```

865: (8) assert_equal(repr(dt), "dtype('<i2', (1,)))")
866: (8) assert_equal(str(dt), "('<i2', (1,))")
867: (4) def test_base_dtype_with_object_type(self):
868: (8)     np.array(['a'], dtype="O").astype("O", [{"name": "0"}]))
869: (4) def test_empty_string_to_object(self):
870: (8)     np.array(["", ""]).astype(object)
871: (4) def test_void_subclass_unsized(self):
872: (8)     dt = np.dtype(np.record)
873: (8)     assert_equal(repr(dt), "dtype('V')")
874: (8)     assert_equal(str(dt), '|V0')
875: (8)     assert_equal(dt.name, 'record')
876: (4) def test_void_subclass_sized(self):
877: (8)     dt = np.dtype((np.record, 2))
878: (8)     assert_equal(repr(dt), "dtype('V2')")
879: (8)     assert_equal(str(dt), '|V2')
880: (8)     assert_equal(dt.name, 'record16')
881: (4) def test_void_subclass_fields(self):
882: (8)     dt = np.dtype((np.record, [('a', '<u2')])))
883: (8)     assert_equal(repr(dt), "dtype((numpy.record, [(('a', '<u2')])))")
884: (8)     assert_equal(str(dt), "(numpy.record, [(('a', '<u2')])))")
885: (8)     assert_equal(dt.name, 'record16')
886: (0) class TestDtypeAttributeDeletion:
887: (4)     def test_dtype_non_writable_attributes_deletion(self):
888: (8)         dt = np.dtype(np.double)
889: (8)         attr = ["subdtype", "descr", "str", "name", "base", "shape",
890: (16)             "isbuiltin", "isnative", "isalignedstruct", "fields",
891: (16)             "metadata", "hasobject"]
892: (8)         for s in attr:
893: (12)             assert_raises(AttributeError, delattr, dt, s)
894: (4)     def test_dtype_writable_attributes_deletion(self):
895: (8)         dt = np.dtype(np.double)
896: (8)         attr = ["names"]
897: (8)         for s in attr:
898: (12)             assert_raises(AttributeError, delattr, dt, s)
899: (0) class TestDtypeAttributes:
900: (4)     def test_descr_has_trailing_void(self):
901: (8)         dtype = np.dtype({
902: (12)             'names': ['A', 'B'],
903: (12)             'formats': ['f4', 'f4'],
904: (12)             'offsets': [0, 8],
905: (12)             'itemsize': 16})
906: (8)         new_dtype = np.dtype(dtype.descr)
907: (8)         assert_equal(new_dtype.itemsize, 16)
908: (4)     def test_name_dtype_subclass(self):
909: (8)         class user_def_subcls(np.void):
910: (12)             pass
911: (8)             assert_equal(np.dtype(user_def_subcls).name, 'user_def_subcls')
912: (4)     def test_zero_stride(self):
913: (8)         arr = np.ones(1, dtype="i8")
914: (8)         arr = np.broadcast_to(arr, 10)
915: (8)         assert arr.strides == (0,)
916: (8)         with pytest.raises(ValueError):
917: (12)             arr.dtype = "i1"
918: (0) class TestDTypeMakeCanonical:
919: (4)     def check_canonical(self, dtype, canonical):
920: (8)         """
921: (8)             Check most properties relevant to "canonical" versions of a dtype,
922: (8)             which is mainly native byte order for datatypes supporting this.
923: (8)             The main work is checking structured dtypes with fields, where we
924: (8)             reproduce most the actual logic used in the C-code.
925: (8)         """
926: (8)         assert type(dtype) is type(canonical)
927: (8)         assert np.can_cast(dtype, canonical, casting="equiv")
928: (8)         assert np.can_cast(canonical, dtype, casting="equiv")
929: (8)         assert canonical.isnative
930: (8)         assert np.result_type(canonical) == canonical
931: (8)         if not dtype.names:
932: (12)             assert dtype.flags == canonical.flags
933: (12)         return

```

```

934: (8)             assert dtype.flags & 0b10000
935: (8)             assert dtype.fields.keys() == canonical.fields.keys()
936: (8)             def aligned_offset(offset, alignment):
937: (12)                 return - (-offset // alignment) * alignment
938: (8)             totalsize = 0
939: (8)             max_alignment = 1
940: (8)             for name in dtype.names:
941: (12)                 new_field_descr = canonical.fields[name][0]
942: (12)                 self.check_canonical(dtype.fields[name][0], new_field_descr)
943: (12)                 expected = 0b11011 & new_field_descr.flags
944: (12)                 assert (canonical.flags & expected) == expected
945: (12)                 if canonical.isalignedstruct:
946: (16)                     totalsize = aligned_offset(totalsize,
new_field_descr.alignment)
947: (16)                     max_alignment = max(new_field_descr.alignment, max_alignment)
948: (12)                     assert canonical.fields[name][1] == totalsize
949: (12)                     assert dtype.fields[name][2:] == canonical.fields[name][2:]
950: (12)                     totalsize += new_field_descr.itemsize
951: (8)                     if canonical.isalignedstruct:
952: (12)                         totalsize = aligned_offset(totalsize, max_alignment)
953: (8)                         assert canonical.itemsize == totalsize
954: (8)                         assert canonical.alignment == max_alignment
955: (4)             def test_simple(self):
956: (8)                 dt = np.dtype(">i4")
957: (8)                 assert np.result_type(dt).isnative
958: (8)                 assert np.result_type(dt).num == dt.num
959: (8)                 struct_dt = np.dtype(">i4,<i1,i8,V3")[[ "f0", "f2" ]]
960: (8)                 canonical = np.result_type(struct_dt)
961: (8)                 assert canonical.itemsize == 4+8
962: (8)                 assert canonical.isnative
963: (8)                 struct_dt = np.dtype(">i1,<i4,i8,V3", align=True)[["f0", "f2"]]
964: (8)                 canonical = np.result_type(struct_dt)
965: (8)                 assert canonical.isalignedstruct
966: (8)                 assert canonical.itemsize == np.dtype("i8").alignment + 8
967: (8)                 assert canonical.isnative
968: (4)             def test_object_flag_not_inherited(self):
969: (8)                 arr = np.ones(3, "i,0,i")[[ "f0", "f2" ]]
970: (8)                 assert arr.dtype.hasobject
971: (8)                 canonical_dt = np.result_type(arr.dtype)
972: (8)                 assert not canonical_dt.hasobject
973: (4)             @pytest.mark.slow
974: (4)             @hypothesis.given(dtype=hypn.nested_dtypes())
975: (4)             def test_make_canonical_hypothesis(self, dtype):
976: (8)                 canonical = np.result_type(dtype)
977: (8)                 self.check_canonical(dtype, canonical)
978: (8)                 two_arg_result = np.result_type(dtype, dtype)
979: (8)                 assert np.can_cast(two_arg_result, canonical, casting="no")
980: (4)             @pytest.mark.slow
981: (4)             @hypothesis.given(
982: (12)                 dtype=hypothesis.extra.numpy.array_dtypes(
983: (16)                     subtype_strategy=hypothesis.extra.numpy.array_dtypes(),
984: (16)                     min_size=5, max_size=10, allow_subarrays=True))
985: (4)             def test_structured(self, dtype):
986: (8)                 field_subset = random.sample(dtype.names, k=4)
987: (8)                 dtype_with_empty_space = dtype[field_subset]
988: (8)                 assert dtype_with_empty_space.itemsize == dtype.itemsize
989: (8)                 canonicalized = np.result_type(dtype_with_empty_space)
990: (8)                 self.check_canonical(dtype_with_empty_space, canonicalized)
991: (8)                 two_arg_result = np.promote_types(
992: (16)                     dtype_with_empty_space, dtype_with_empty_space)
993: (8)                 assert np.can_cast(two_arg_result, canonicalized, casting="no")
994: (8)                 dtype_aligned = np.dtype(dtype.descr, align=not dtype.isalignedstruct)
995: (8)                 dtype_with_empty_space = dtype_aligned[field_subset]
996: (8)                 assert dtype_with_empty_space.itemsize == dtype_aligned.itemsize
997: (8)                 canonicalized = np.result_type(dtype_with_empty_space)
998: (8)                 self.check_canonical(dtype_with_empty_space, canonicalized)
999: (8)                 two_arg_result = np.promote_types(
1000: (12)                     dtype_with_empty_space, dtype_with_empty_space)
1001: (8)                 assert np.can_cast(two_arg_result, canonicalized, casting="no")

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1002: (0)
1003: (4)
1004: (8)
1005: (12)
1006: (12)
1007: (12)
1008: (12)
1009: (12)
1010: (12)
1011: (12)
1012: (16)
1013: (12)
1014: (12)
1015: (12)
1016: (12)
1017: (4)
1018: (35)
1019: (4)
1020: (8)
1021: (4)
1022: (8)
1023: (8)
1024: (4)
1025: (8)
1026: (8)
1027: (4)
1028: (8)
1029: (8)
1030: (4)
1031: (8)
1032: (12)
1033: (12)
1034: (12)
1035: (12)
1036: (8)
1037: (4)
1038: (8)
1039: (23)
1040: (23)
1041: (8)
1042: (4)
1043: (4)
1044: (38)
1045: (4)
1046: (8)
1047: (8)
1048: (8)
1049: (12)
1050: (12)
1051: (4)
1052: (8)
1053: (8)
1054: (4)
1055: (8)
1056: (8)
1057: (4)
1058: (8)
1059: (12)
1060: (12)
1061: (0)
1062: (4)
1063: (4)
1064: (4)
1065: (4)
1066: (4)
1067: (12)
1068: (13)
1069: (13)
1070: (13)

class TestPickling:
    def check_pickling(self, dtype):
        for proto in range(pickle.HIGHEST_PROTOCOL + 1):
            buf = pickle.dumps(dtype, proto)
            assert b"_DType_reconstruct" not in buf
            assert b"dtype" in buf
            pickled = pickle.loads(buf)
            assert_equal(pickled, dtype)
            assert_equal(pickled.descr, dtype.descr)
            if dtype.metadata is not None:
                assert_equal(pickled.metadata, dtype.metadata)
            x = np.zeros(3, dtype=dtype)
            y = np.zeros(3, dtype=pickled)
            assert_equal(x, y)
            assert_equal(x[0], y[0])
    @pytest.mark.parametrize('t', [int, float, complex, np.int32, str, object,
                                  np.compat_unicode, bool])
    def test_builtin(self, t):
        self.check_pickling(np.dtype(t))
    def test_structured(self):
        dt = np.dtype(([('a', '>f4', (2, 1)), ('b', '<f8', (1, 3))], (2, 2)))
        self.check_pickling(dt)
    def test_structured_aligned(self):
        dt = np.dtype('i4, i1', align=True)
        self.check_pickling(dt)
    def test_structured_unaligned(self):
        dt = np.dtype('i4, i1', align=False)
        self.check_pickling(dt)
    def test_structured_padded(self):
        dt = np.dtype({
            'names': ['A', 'B'],
            'formats': ['f4', 'f4'],
            'offsets': [0, 8],
            'itemsize': 16})
        self.check_pickling(dt)
    def test_structured_titles(self):
        dt = np.dtype({'names': ['r', 'b'],
                      'formats': ['u1', 'u1'],
                      'titles': ['Red pixel', 'Blue pixel']})
        self.check_pickling(dt)
    @pytest.mark.parametrize('base', [m8, M8])
    @pytest.mark.parametrize('unit', ['', 'Y', 'M', 'W', 'D', 'h', 'm', 's',
                                    'ms', 'us', 'ns', 'ps', 'fs', 'as'])
    def test_datetime(self, base, unit):
        dt = np.dtype('%s[%s]' % (base, unit) if unit else base)
        self.check_pickling(dt)
        if unit:
            dt = np.dtype('%s[7s]' % (base, unit))
            self.check_pickling(dt)
    def test_metadata(self):
        dt = np.dtype(int, metadata={'datum': 1})
        self.check_pickling(dt)
    @pytest.mark.parametrize("DType",
                           [type(np.dtype(t)) for t in np.typecodes['All']] +
                           [np.dtype(rational), np.dtype]])
    def test_pickle_types(self, DType):
        for proto in range(pickle.HIGHEST_PROTOCOL + 1):
            roundtrip_DType = pickle.loads(pickle.dumps(DType, proto))
            assert roundtrip_DType is DType

class TestPromotion:
    """Test cases related to more complex DType promotions. Further promotion
    tests are defined in `test_numeric.py`"""
    @np._no_nep50_warning()
    @pytest.mark.parametrize(["other", "expected", "expected_weak"],
                           [(2**16-1, np.complex64, None),
                            (2**32-1, np.complex128, np.complex64),
                            (np.float16(2), np.complex64, None),
                            (np.float32(2), np.complex64, None)])

```

```

1071: (13)             (np.longdouble(2), np.complex64, np.clongdouble),
1072: (13)             (np.longdouble(np.nextafter(1.7e308, 0.)),
1073: (18)                 np.complex128, np.clongdouble),
1074: (13)             (np.longdouble(np.nextafter(1.7e308, np.inf)),
1075: (18)                 np.clongdouble, None),
1076: (13)             (np.complex64(2), np.complex64, None),
1077: (13)             (np.clongdouble(2), np.complex64, np.clongdouble),
1078: (13)             (np.clongdouble(np.nextafter(1.7e308, 0.) * 1j),
1079: (18)                 np.complex128, np.clongdouble),
1080: (13)             (np.clongdouble(np.nextafter(1.7e308, np.inf)),
1081: (18)                 np.clongdouble, None),
1082: (13)         ])
1083: (4)     def test_complex_other_value_based(self,
1084: (12)         weak_promotion, other, expected, expected_weak):
1085: (8)         if weak_promotion and expected_weak is not None:
1086: (12)             expected = expected_weak
1087: (8)             min_complex = np.dtype(np.complex64)
1088: (8)             res = np.result_type(other, min_complex)
1089: (8)             assert res == expected
1090: (8)             res = np.minimum(other, np.ones(3, dtype=min_complex)).dtype
1091: (8)             assert res == expected
1092: (4)     @pytest.mark.parametrize(["other", "expected"],
1093: (17)         [(np.bool_, np.complex128),
1094: (18)             (np.int64, np.complex128),
1095: (18)             (np.float16, np.complex64),
1096: (18)             (np.float32, np.complex64),
1097: (18)             (np.float64, np.complex128),
1098: (18)             (np.longdouble, np.clongdouble),
1099: (18)             (np.complex64, np.complex64),
1100: (18)             (np.complex128, np.complex128),
1101: (18)             (np.clongdouble, np.clongdouble),
1102: (18)         ])
1103: (4)     def test_complex_scalar_value_based(self, other, expected):
1104: (8)         complex_scalar = 1j
1105: (8)         res = np.result_type(other, complex_scalar)
1106: (8)         assert res == expected
1107: (8)         res = np.minimum(np.ones(3, dtype=other), complex_scalar).dtype
1108: (8)         assert res == expected
1109: (4)     def test_complex_pyscalar_promote_rational(self):
1110: (8)         with pytest.raises(TypeError,
1111: (16)             match=r".* no common DType exists for the given inputs"):
1112: (12)             np.result_type(1j, rational)
1113: (8)         with pytest.raises(TypeError,
1114: (16)             match=r".* no common DType exists for the given inputs"):
1115: (12)             np.result_type(1j, rational(1, 2))
1116: (4)     @pytest.mark.parametrize("val", [2, 2**32, 2**63, 2**64, 2**100])
1117: (4)     def test_python_integer_promotion(self, val):
1118: (8)         expected_dtype = np.result_type(np.array(val)).dtype,
1119: (8)         assert np.result_type(val, 0) == expected_dtype
1120: (8)         assert np.result_type(val, np.int8(0)) == expected_dtype
1121: (4)     @pytest.mark.parametrize(["other", "expected"],
1122: (12)         [(1, rational), (1., np.float64)])
1123: (4)     @np._no_nep50_warning()
1124: (4)     def test_float_int_pyscalar_promote_rational(
1125: (12)         self, weak_promotion, other, expected):
1126: (8)         if not weak_promotion and type(other) == float:
1127: (12)             with pytest.raises(TypeError,
1128: (20)                 match=r".* do not have a common DType"):
1129: (16)                 np.result_type(other, rational)
1130: (8)         else:
1131: (12)             assert np.result_type(other, rational) == expected
1132: (8)             assert np.result_type(other, rational(1, 2)) == expected
1133: (4)     @pytest.mark.parametrize(["dtypes", "expected"], [
1134: (13)         ([np.uint16, np.int16, np.float16], np.float32),
1135: (13)         ([np.uint16, np.int8, np.float16], np.float32),
1136: (13)         ([np.uint8, np.int16, np.float16], np.float32),
1137: (13)         ([1, 1, np.float64], np.float64),
1138: (13)         ([1, 1., np.complex128], np.complex128),

```

```

1139: (13) ([1, 1j, np.float64], np.complex128),
1140: (13) ([1., 1., np.int64], np.float64),
1141: (13) ([1., 1j, np.float64], np.complex128),
1142: (13) ([1j, 1j, np.float64], np.complex128),
1143: (13) ([1, True, np.bool_], np.int_),
1144: (12) ])
1145: (4)     def test_permutations_do_not_influence_result(self, dtypes, expected):
1146: (8)         for perm in permutations(dtypes):
1147: (12)             assert np.result_type(*perm) == expected
1148: (0)     def test_rational_dtype():
1149: (4)         a = np.array([1111], dtype=rational).astype
1150: (4)         assert_raises(OverflowError, a, 'int8')
1151: (4)         x = rational(1)
1152: (4)         assert_equal(np.array([x,x]).dtype, np.dtype(rational))
1153: (0)     def test_dtotypes_are_true():
1154: (4)         assert bool(np.dtype('f8'))
1155: (4)         assert bool(np.dtype('i8'))
1156: (4)         assert bool(np.dtype([('a', 'i8'), ('b', 'f4')]))
1157: (0)     def test_invalid_dtype_string():
1158: (4)         assert_raises(TypeError, np.dtype, 'f8,i8,[f8,i8]')
1159: (4)         assert_raises(TypeError, np.dtype, 'F1\xfcge1')
1160: (0)     def test_keyword_argument():
1161: (4)         assert np.dtype(dtype=np.float64) == np.dtype(np.float64)
1162: (0)     def test_ulong_dtype():
1163: (4)         assert np.dtype("ulong") == np.dtype(np.uint)
1164: (0) class TestFromDTypeAttribute:
1165: (4)     def test_simple(self):
1166: (8)         class dt:
1167: (12)             dtype = np.dtype("f8")
1168: (8)             assert np.dtype(dt) == np.float64
1169: (8)             assert np.dtype(dt()) == np.float64
1170: (4) @pytest.mark.skipif(IS_PYTHON, reason="Pyston disables recursion"
1171: (4) checking")
1172: (8)     def test_recursion(self):
1173: (12)         class dt:
1174: (8)             pass
1175: (8)             dt.dtype = dt
1176: (12)             with pytest.raises(RecursionError):
1177: (8)                 np.dtype(dt)
1178: (8)             dt_instance = dt()
1179: (8)             dt_instance.dtype = dt
1180: (12)             with pytest.raises(RecursionError):
1181: (4)                 np.dtype(dt_instance)
1182: (8)     def test_void_subtype(self):
1183: (12)         class dt(np.void):
1184: (8)             dtype = np.dtype("f,f")
1185: (8)             np.dtype(dt)
1186: (8)             np.dtype(dt(1))
1187: (4) @pytest.mark.skipif(IS_PYTHON, reason="Pyston disables recursion"
1188: (8) checking")
1189: (12)     def test_void_subtype_recursion(self):
1190: (8)         class vdt(np.void):
1191: (8)             pass
1192: (12)             vdt.dtype = vdt
1193: (8)             with pytest.raises(RecursionError):
1194: (12)                 np.dtype(vdt)
1195: (0) class TestDTypeClasses:
1196: (4) @pytest.mark.parametrize("dtype", list(np.typecodes['All']) + [rational])
1197: (4)     def test_basic_dtotypes_subclass_properties(self, dtype):
1198: (8)         dtype = np.dtype(dtype)
1199: (8)         assert isinstance(dtype, np.dtype)
1200: (8)         assert type(dtype) is not np.dtype
1201: (8)         if dtype.type.__name__ != "rational":
1202: (12)             dt_name = type(dtype).__name__.lower().removesuffix("dtype")
1203: (12)             if dt_name == "uint" or dt_name == "int":
1204: (16)                 dt_name += "c"
1205: (12)             sc_name = dtype.type.__name__

```

```

1206: (12) assert dt_name == sc_name.strip("_")
1207: (12) assert type(dtype).__module__ == "numpy.dtypes"
1208: (12) assert getattr(numpy.dtypes, type(dtype).__name__) is type(dtype)
1209: (8) else:
1210: (12)     assert type(dtype).__name__ == "dtype[rational]"
1211: (12)     assert type(dtype).__module__ == "numpy"
1212: (8) assert not type(dtype).__abstract__
1213: (8) parametric = (np.void, np.str_, np.bytes_, np.datetime64,
np.timedelta64)
1214: (8) if dtype.type not in parametric:
1215: (12)     assert not type(dtype).__parametric
1216: (12)     assert type(dtype)() is dtype
1217: (8) else:
1218: (12)     assert type(dtype).__parametric
1219: (12)     with assert_raises(TypeError):
1220: (16)         type(dtype)()
1221: (4) def test_dtype_superclass(self):
1222: (8)     assert type(np.dtype) is not type
1223: (8)     assert isinstance(np.dtype, type)
1224: (8)     assert type(np.dtype).__name__ == "_DTypeMeta"
1225: (8)     assert type(np.dtype).__module__ == "numpy"
1226: (8)     assert np.dtype.__abstract__
1227: (4) def test_is_numeric(self):
1228: (8)     all_codes = set(np.typecodes['All'])
1229: (8)     numeric_codes = set(np.typecodes['AllInteger'] +
1230: (28)         np.typecodes['AllFloat'] + '?')
1231: (8)     non_numeric_codes = all_codes - numeric_codes
1232: (8)     for code in numeric_codes:
1233: (12)         assert type(np.dtype(code)).__is_numeric
1234: (8)     for code in non_numeric_codes:
1235: (12)         assert not type(np.dtype(code)).__is_numeric
1236: (4) @pytest.mark.parametrize("int_", ["UInt", "Int"])
1237: (4) @pytest.mark.parametrize("size", [8, 16, 32, 64])
1238: (4) def test_integer_alias_names(self, int_, size):
1239: (8)     DType = getattr(numpy.dtypes, f"{int_}{size}DType")
1240: (8)     sctype = getattr(numpy, f"{int_.lower()}{size}")
1241: (8)     assert DType.type is sctype
1242: (8)     assert DType.__name__.lower().removesuffix("dtype") == sctype.__name__
1243: (4) @pytest.mark.parametrize("name",
1244: (12)     ["Half", "Float", "Double", "CFloat", "CDouble"])
1245: (4) def test_float_alias_names(self, name):
1246: (8)     with pytest.raises(AttributeError):
1247: (12)         setattr(numpy.dtypes, name + "DType") is numpy.dtypes.Float16DType
1248: (0) class TestFromCTypes:
1249: (4)     @staticmethod
1250: (4)     def check(ctype, dtype):
1251: (8)         dtype = np.dtype(dtype)
1252: (8)         assert_equal(np.dtype(ctype), dtype)
1253: (8)         assert_equal(np.dtype(ctype()), dtype)
1254: (4)     def test_array(self):
1255: (8)         c8 = ctypes.c_uint8
1256: (8)         self.check(3 * c8, (np.uint8, (3,)))
1257: (8)         self.check(1 * c8, (np.uint8, (1,)))
1258: (8)         self.check(0 * c8, (np.uint8, (0,)))
1259: (8)         self.check(1 * (3 * c8), ((np.uint8, (3,)), (1,)))
1260: (8)         self.check(3 * (1 * c8), ((np.uint8, (1,)), (3,)))
1261: (4)     def test_padded_structure(self):
1262: (8)         class PaddedStruct(ctypes.Structure):
1263: (12)             _fields_ = [
1264: (16)                 ('a', ctypes.c_uint8),
1265: (16)                 ('b', ctypes.c_uint16)
1266: (12)             ]
1267: (8)             expected = np.dtype([
1268: (12)                 ('a', np.uint8),
1269: (12)                 ('b', np.uint16)
1270: (8)             ], align=True)
1271: (8)             self.check(PaddedStruct, expected)
1272: (4)         def test_bit_fields(self):
1273: (8)             class BitfieldStruct(ctypes.Structure):

```

```

1274: (12)             _fields_ = [
1275: (16)                 ('a', ctypes.c_uint8, 7),
1276: (16)                 ('b', ctypes.c_uint8, 1)
1277: (12)             ]
1278: (8)             assert_raises(TypeError, np.dtype, BitfieldStruct)
1279: (8)             assert_raises(TypeError, np.dtype, BitfieldStruct())
1280: (4)             def test_pointer(self):
1281: (8)                 p_uint8 = ctypes.POINTER(ctypes.c_uint8)
1282: (8)                 assert_raises(TypeError, np.dtype, p_uint8)
1283: (4)             def test_void_pointer(self):
1284: (8)                 self.check(ctypes.c_void_p, np.uintp)
1285: (4)             def test_union(self):
1286: (8)                 class Union(ctypes.Union):
1287: (12)                     _fields_ = [
1288: (16)                         ('a', ctypes.c_uint8),
1289: (16)                         ('b', ctypes.c_uint16),
1290: (12)                     ]
1291: (8)                     expected = np.dtype(dict(
1292: (12)                         names=['a', 'b'],
1293: (12)                         formats=[np.uint8, np.uint16],
1294: (12)                         offsets=[0, 0],
1295: (12)                         itemsize=2
1296: (8)                     ))
1297: (8)                     self.check(Union, expected)
1298: (4)             def test_union_with_struct_packed(self):
1299: (8)                 class Struct(ctypes.Structure):
1300: (12)                     _pack_ = 1
1301: (12)                     _fields_ = [
1302: (16)                         ('one', ctypes.c_uint8),
1303: (16)                         ('two', ctypes.c_uint32)
1304: (12)                     ]
1305: (8)                 class Union(ctypes.Union):
1306: (12)                     _fields_ = [
1307: (16)                         ('a', ctypes.c_uint8),
1308: (16)                         ('b', ctypes.c_uint16),
1309: (16)                         ('c', ctypes.c_uint32),
1310: (16)                         ('d', Struct),
1311: (12)                     ]
1312: (8)                     expected = np.dtype(dict(
1313: (12)                         names=['a', 'b', 'c', 'd'],
1314: (12)                         formats=['u1', np.uint16, np.uint32, [('one', 'u1'), ('two', np.uint32)]],
1315: (12)                         offsets=[0, 0, 0, 0],
1316: (12)                         itemsize=ctypes.sizeof(Union)
1317: (8)                     ))
1318: (8)                     self.check(Union, expected)
1319: (4)             def test_union_packed(self):
1320: (8)                 class Struct(ctypes.Structure):
1321: (12)                     _fields_ = [
1322: (16)                         ('one', ctypes.c_uint8),
1323: (16)                         ('two', ctypes.c_uint32)
1324: (12)                     ]
1325: (12)                     _pack_ = 1
1326: (8)                 class Union(ctypes.Union):
1327: (12)                     _pack_ = 1
1328: (12)                     _fields_ = [
1329: (16)                         ('a', ctypes.c_uint8),
1330: (16)                         ('b', ctypes.c_uint16),
1331: (16)                         ('c', ctypes.c_uint32),
1332: (16)                         ('d', Struct),
1333: (12)                     ]
1334: (8)                     expected = np.dtype(dict(
1335: (12)                         names=['a', 'b', 'c', 'd'],
1336: (12)                         formats=['u1', np.uint16, np.uint32, [('one', 'u1'), ('two', np.uint32)]],
1337: (12)                         offsets=[0, 0, 0, 0],
1338: (12)                         itemsize=ctypes.sizeof(Union)
1339: (8)                     ))
1340: (8)                     self.check(Union, expected)

```

```

1341: (4)
1342: (8)
1343: (12)
1344: (12)
1345: (16)
1346: (16)
1347: (12)
1348: (8)
1349: (12)
1350: (12)
1351: (8)
1352: (8)
1353: (4)
1354: (8)
1355: (12)
1356: (12)
1357: (16)
1358: (16)
1359: (16)
1360: (16)
1361: (16)
1362: (16)
1363: (16)
1364: (16)
1365: (8)
1366: (12)
np.uint32, np.uint8 ],
1367: (12)
1368: (12)
1369: (12)
1370: (8)
1371: (4)
1372: (8)
1373: (12)
1374: (16)
1375: (16)
1376: (12)
1377: (12)
1378: (8)
1379: (8)
1380: (4)
1381: (8)
1382: (12)
1383: (16)
1384: (16)
1385: (12)
1386: (12)
1387: (8)
1388: (8)
1389: (4)
1390: (8)
1391: (12)
1392: (16)
1393: (16)
1394: (12)
1395: (8)
1396: (12)
1397: (12)
1398: (8)
1399: (8)
1400: (4)
1401: (8)
1402: (12)
1403: (16)
1404: (16)
1405: (12)
1406: (8)
1407: (12)
1408: (12)

        def test_packed_structure(self):
            class PackedStructure(ctypes.Structure):
                _pack_ = 1
                _fields_ = [
                    ('a', ctypes.c_uint8),
                    ('b', ctypes.c_uint16)
                ]
            expected = np.dtype([
                ('a', np.uint8),
                ('b', np.uint16)
            ])
            self.check(PackedStructure, expected)
        def test_large_packed_structure(self):
            class PackedStructure(ctypes.Structure):
                _pack_ = 2
                _fields_ = [
                    ('a', ctypes.c_uint8),
                    ('b', ctypes.c_uint16),
                    ('c', ctypes.c_uint8),
                    ('d', ctypes.c_uint16),
                    ('e', ctypes.c_uint32),
                    ('f', ctypes.c_uint32),
                    ('g', ctypes.c_uint8)
                ]
            expected = np.dtype(dict(
                formats=[np.uint8, np.uint16, np.uint8, np.uint16, np.uint32,
np.uint32, np.uint8 ],
                offsets=[0, 2, 4, 6, 8, 12, 16],
                names=['a', 'b', 'c', 'd', 'e', 'f', 'g'],
                itemsize=18))
            self.check(PackedStructure, expected)
        def test_big_endian_structure_packed(self):
            class BigEndStruct(ctypes.BigEndianStructure):
                _fields_ = [
                    ('one', ctypes.c_uint8),
                    ('two', ctypes.c_uint32)
                ]
                _pack_ = 1
            expected = np.dtype([('one', 'u1'), ('two', '>u4')])
            self.check(BigEndStruct, expected)
        def test_little_endian_structure_packed(self):
            class LittleEndStruct(ctypes.LittleEndianStructure):
                _fields_ = [
                    ('one', ctypes.c_uint8),
                    ('two', ctypes.c_uint32)
                ]
                _pack_ = 1
            expected = np.dtype([('one', 'u1'), ('two', '<u4')])
            self.check(LittleEndStruct, expected)
        def test_little_endian_structure(self):
            class PaddedStruct(ctypes.LittleEndianStructure):
                _fields_ = [
                    ('a', ctypes.c_uint8),
                    ('b', ctypes.c_uint16)
                ]
            expected = np.dtype([
                ('a', '<B'),
                ('b', '<H')
            ], align=True)
            self.check(PaddedStruct, expected)
        def test_big_endian_structure(self):
            class PaddedStruct(ctypes.BigEndianStructure):
                _fields_ = [
                    ('a', ctypes.c_uint8),
                    ('b', ctypes.c_uint16)
                ]
            expected = np.dtype([
                ('a', '>B'),
                ('b', '>H')
            ])

```

```

1409: (8)           ], align=True)
1410: (8)           self.check(PaddedStruct, expected)
1411: (4)           def test_simple_endian_types(self):
1412: (8)             self.check(ctypes.c_uint16._ctype_le__, np.dtype('<u2'))
1413: (8)             self.check(ctypes.c_uint16._ctype_be__, np.dtype('>u2'))
1414: (8)             self.check(ctypes.c_uint8._ctype_le__, np.dtype('u1'))
1415: (8)             self.check(ctypes.c_uint8._ctype_be__, np.dtype('u1'))
1416: (4)             all_types = set(np.typecodes['All'])
1417: (4)             all_pairs = permutations(all_types, 2)
1418: (4)             @pytest.mark.parametrize("pair", all_pairs)
1419: (4)             def test_pairs(self, pair):
1420: (8)               """
1421: (8)                 Check that np.dtype('x,y') matches [np.dtype('x'), np.dtype('y')]
1422: (8)                 Example: np.dtype('d,I') -> dtype([('f0', '<f8'), ('f1', '<u4')])
1423: (8)               """
1424: (8)               pair_type = np.dtype('{}{}'.format(*pair))
1425: (8)               expected = np.dtype([('f0', pair[0]), ('f1', pair[1])])
1426: (8)               assert_equal(pair_type, expected)
1427: (0)           class TestUserDType:
1428: (4)             @pytest.mark.leaks_references(reason="dynamically creates custom dtype.")
1429: (4)             def test_custom_structured_dtype(self):
1430: (8)               class mytype:
1431: (12)                 pass
1432: (8)                 blueprint = np.dtype([("field", object)])
1433: (8)                 dt = create_custom_field_dtype(blueprint, mytype, 0)
1434: (8)                 assert dt.type == mytype
1435: (8)                 assert np.dtype(mytype) == np.dtype("O")
1436: (4)             def test_custom_structured_dtype_errors(self):
1437: (8)               class mytype:
1438: (12)                 pass
1439: (8)                 blueprint = np.dtype([("field", object)])
1440: (8)                 with pytest.raises(ValueError):
1441: (12)                   create_custom_field_dtype(blueprint, mytype, 1)
1442: (8)                 with pytest.raises(RuntimeError):
1443: (12)                   create_custom_field_dtype(blueprint, mytype, 2)
1444: (0)           class TestClassGetItem:
1445: (4)             def test_dtype(self) -> None:
1446: (8)               alias = np.dtype[Any]
1447: (8)               assert isinstance(alias, types.GenericAlias)
1448: (8)               assert alias.__origin__ is np.dtype
1449: (4)             @pytest.mark.parametrize("code", np.typecodes["All"])
1450: (4)             def test_dtype_subclass(self, code: str) -> None:
1451: (8)               cls = type(np.dtype(code))
1452: (8)               alias = cls[Any]
1453: (8)               assert isinstance(alias, types.GenericAlias)
1454: (8)               assert alias.__origin__ is cls
1455: (4)             @pytest.mark.parametrize("arg_len", range(4))
1456: (4)             def test_subscript_tuple(self, arg_len: int) -> None:
1457: (8)               arg_tup = (Any,) * arg_len
1458: (8)               if arg_len == 1:
1459: (12)                 assert np.dtype[arg_tup]
1460: (8)               else:
1461: (12)                 with pytest.raises(TypeError):
1462: (16)                   np.dtype[arg_tup]
1463: (4)             def test_subscript_scalar(self) -> None:
1464: (8)               assert np.dtype[Any]
1465: (0)             def test_result_type_integers_and_unitless_timedelta64():
1466: (4)               td = np.timedelta64(4)
1467: (4)               result = np.result_type(0, td)
1468: (4)               assert_dtype_equal(result, td.dtype)
1469: (0)             def test_creating_dtype_with_dtype_class_errors():
1470: (4)               with pytest.raises(TypeError, match="Cannot convert np.dtype into a"):
1471: (8)                 np.array(np.ones(10), dtype=np.dtype)

```

-----  
File 92 - test\_einsum.py:

```
1: (0)           import itertools
```

```

2: (0) import sys
3: (0) import platform
4: (0) import pytest
5: (0) import numpy as np
6: (0) from numpy.testing import (
7: (4)     assert_, assert_equal, assert_array_equal, assert_almost_equal,
8: (4)     assert_raises, suppress_warnings, assert_raises_regex, assert_allclose
9: (4) )
10: (0) try:
11: (4)     COMPILERS = np.show_config(mode="dicts")["Compilers"]
12: (4)     USING_CLANG_CL = COMPILERS["c"]["name"] == "clang-cl"
13: (0) except TypeError:
14: (4)     USING_CLANG_CL = False
15: (0) chars = 'abcdefghijklmnopqrstuvwxyz'
16: (0) sizes = np.array([2, 3, 4, 5, 4, 3, 2, 6, 5, 4, 3])
17: (0) global_size_dict = dict(zip(chars, sizes))
18: (0) class TestEinsum:
19: (4)     def test_einsum_errors(self):
20: (8)         for do_opt in [True, False]:
21: (12)             assert_raises(ValueError, np.einsum, optimize=do_opt)
22: (12)             assert_raises(ValueError, np.einsum, "", optimize=do_opt)
23: (12)             assert_raises(TypeError, np.einsum, 0, 0, optimize=do_opt)
24: (12)             assert_raises(TypeError, np.einsum, "", 0, out='test',
25: (26)                 optimize=do_opt)
26: (12)             assert_raises(ValueError, np.einsum, "", 0, order='W',
27: (26)                 optimize=do_opt)
28: (12)             assert_raises(ValueError, np.einsum, "", 0, casting='blah',
29: (26)                 optimize=do_opt)
30: (12)             assert_raises(TypeError, np.einsum, "", 0, dtype='bad_data_type',
31: (26)                 optimize=do_opt)
32: (12)             assert_raises(TypeError, np.einsum, "", 0, bad_arg=0,
33: (26)                 optimize=do_opt)
34: (12)             assert_raises(TypeError, np.einsum, *(None,) * 63, optimize=do_opt)
35: (12)             assert_raises(ValueError, np.einsum, "", 0, 0, optimize=do_opt)
36: (12)             assert_raises(ValueError, np.einsum, "", 0, [0], [0],
37: (26)                 optimize=do_opt)
38: (12)             assert_raises(ValueError, np.einsum, "", [0], optimize=do_opt)
39: (12)             assert_raises(ValueError, np.einsum, "i", 0, optimize=do_opt)
40: (12)             assert_raises(ValueError, np.einsum, "ij", [0, 0],
41: (12)                 optimize=do_opt)
42: (12)             assert_raises(ValueError, np.einsum, "...i", 0, optimize=do_opt)
43: (12)             assert_raises(ValueError, np.einsum, "i...j", [0, 0],
44: (12)                 optimize=do_opt)
45: (12)             assert_raises(ValueError, np.einsum, "i...", 0, optimize=do_opt)
46: (12)             assert_raises(ValueError, np.einsum, "ij...", [0, 0],
47: (12)                 optimize=do_opt)
48: (12)             assert_raises(ValueError, np.einsum, "i->..j", [0, 0],
49: (12)                 optimize=do_opt)
50: (12)             assert_raises(ValueError, np.einsum, "j->..j", [0, 0],
51: (12)                 optimize=do_opt)
52: (12)             assert_raises(ValueError, np.einsum, "j->.j...", [0, 0],
53: (12)                 optimize=do_opt)
54: (12)             assert_raises(ValueError, np.einsum, "i%...", [0, 0],
55: (12)                 optimize=do_opt)
56: (12)             assert_raises(ValueError, np.einsum, "...j$", [0, 0],
57: (12)                 optimize=do_opt)
58: (12)             assert_raises(ValueError, np.einsum, "i->&", [0, 0],
59: (12)                 optimize=do_opt)
60: (12)             assert_raises(ValueError, np.einsum, "i->ij", [0, 0],
61: (12)                 optimize=do_opt)
62: (12)             assert_raises(ValueError, np.einsum, "ij->jij", [[0, 0], [0, 0]],
63: (26)                 optimize=do_opt)
64: (12)             assert_raises(ValueError, np.einsum, "ii",
65: (26)                 np.arange(6).reshape(2, 3), optimize=do_opt)
66: (12)             assert_raises(ValueError, np.einsum, "ii->i",
67: (26)                 np.arange(6).reshape(2, 3), optimize=do_opt)
68: (12)             assert_raises(ValueError, np.einsum, "i", np.arange(6).reshape(2,
69: (12)                 optimize=do_opt))

```

```

3),
60: (26)                                optimize=do_opt)
61: (12) assert_raises(ValueError, np.einsum, "i->i", [[0, 1], [0, 1]],
62: (26)   out=np.arange(4).reshape(2, 2), optimize=do_opt)
63: (12) with assert_raises_regex(ValueError, "'b'"):
64: (16)     a = np.ones((3, 3, 4, 5, 6))
65: (16)     b = np.ones((3, 4, 5))
66: (16)     np.einsum('aabcb,abc', a, b)
67: (12) assert_raises(ValueError, np.einsum, "i->i",
np.arange(6).reshape(-1, 1),
68: (26)   optimize=do_opt, order='d')
69: (4) def test_einsum_object_errors(self):
70: (8)     class CustomException(Exception):
71: (12)         pass
72: (8)     class DestructoBox:
73: (12)         def __init__(self, value, destruct):
74: (16)             self._val = value
75: (16)             self._destruct = destruct
76: (12)         def __add__(self, other):
77: (16)             tmp = self._val + other._val
78: (16)             if tmp >= self._destruct:
79: (20)                 raise CustomException
80: (16)             else:
81: (20)                 self._val = tmp
82: (20)                 return self
83: (12)         def __radd__(self, other):
84: (16)             if other == 0:
85: (20)                 return self
86: (16)             else:
87: (20)                 return self.__add__(other)
88: (12)         def __mul__(self, other):
89: (16)             tmp = self._val * other._val
90: (16)             if tmp >= self._destruct:
91: (20)                 raise CustomException
92: (16)             else:
93: (20)                 self._val = tmp
94: (20)                 return self
95: (12)         def __rmul__(self, other):
96: (16)             if other == 0:
97: (20)                 return self
98: (16)             else:
99: (20)                 return self.__mul__(other)
100: (8) a = np.array([DestructoBox(i, 5) for i in range(1, 10)],
101: (21)   dtype='object').reshape(3, 3)
102: (8) assert_raises(CustomException, np.einsum, "ij->i", a)
103: (8) b = np.array([DestructoBox(i, 100) for i in range(0, 27)],
104: (21)   dtype='object').reshape(3, 3, 3)
105: (8) assert_raises(CustomException, np.einsum, "i...k->...", b)
106: (8) b = np.array([DestructoBox(i, 55) for i in range(1, 4)],
107: (21)   dtype='object')
108: (8) assert_raises(CustomException, np.einsum, "ij, j", a, b)
109: (8) assert_raises(CustomException, np.einsum, "ij, jh", a, a)
110: (8) assert_raises(CustomException, np.einsum, "ij->", a)
111: (4) def test_einsum_views(self):
112: (8)     for do_opt in [True, False]:
113: (12)         a = np.arange(6)
114: (12)         a.shape = (2, 3)
115: (12)         b = np.einsum("...", a, optimize=do_opt)
116: (12)         assert_(b.base is a)
117: (12)         b = np.einsum(a, [Ellipsis], optimize=do_opt)
118: (12)         assert_(b.base is a)
119: (12)         b = np.einsum("ij", a, optimize=do_opt)
120: (12)         assert_(b.base is a)
121: (12)         assert_equal(b, a)
122: (12)         b = np.einsum(a, [0, 1], optimize=do_opt)
123: (12)         assert_(b.base is a)
124: (12)         assert_equal(b, a)
125: (12)         b = np.einsum("...", a, optimize=do_opt)
126: (12)         assert_(b.flags['WRITEABLE'])
```

```
127: (12)          a.flags['WRITEABLE'] = False
128: (12)          b = np.einsum("...", a, optimize=do_opt)
129: (12)          assert_(not b.flags['WRITEABLE'])
130: (12)          a = np.arange(6)
131: (12)          a.shape = (2, 3)
132: (12)          b = np.einsum("ji", a, optimize=do_opt)
133: (12)          assert_(b.base is a)
134: (12)          assert_equal(b, a.T)
135: (12)          b = np.einsum(a, [1, 0], optimize=do_opt)
136: (12)          assert_(b.base is a)
137: (12)          assert_equal(b, a.T)
138: (12)          a = np.arange(9)
139: (12)          a.shape = (3, 3)
140: (12)          b = np.einsum("ii->i", a, optimize=do_opt)
141: (12)          assert_(b.base is a)
142: (12)          assert_equal(b, [a[i, i] for i in range(3)])
143: (12)          b = np.einsum(a, [0, 0], [0], optimize=do_opt)
144: (12)          assert_(b.base is a)
145: (12)          assert_equal(b, [a[i, i] for i in range(3)])
146: (12)          a = np.arange(27)
147: (12)          a.shape = (3, 3, 3)
148: (12)          b = np.einsum("...ii->...i", a, optimize=do_opt)
149: (12)          assert_(b.base is a)
150: (12)          assert_equal(b, [[x[i, i] for i in range(3)] for x in a])
151: (12)          b = np.einsum(a, [Ellipsis, 0, 0], [Ellipsis, 0], optimize=do_opt)
152: (12)          assert_(b.base is a)
153: (12)          assert_equal(b, [[x[i, i] for i in range(3)] for x in a])
154: (12)          b = np.einsum("i...->...i", a, optimize=do_opt)
155: (12)          assert_(b.base is a)
156: (12)          assert_equal(b, [[x[i, i] for i in range(3)]
157: (29)                      for x in a.transpose(2, 0, 1)])
158: (12)          b = np.einsum(a, [0, 0, Ellipsis], [Ellipsis, 0], optimize=do_opt)
159: (12)          assert_(b.base is a)
160: (12)          assert_equal(b, [[x[i, i] for i in range(3)]
161: (29)                      for x in a.transpose(2, 0, 1)])
162: (12)          b = np.einsum("...ii->i...", a, optimize=do_opt)
163: (12)          assert_(b.base is a)
164: (12)          assert_equal(b, [a[:, i, i] for i in range(3)])
165: (12)          b = np.einsum(a, [Ellipsis, 0, 0], [0, Ellipsis], optimize=do_opt)
166: (12)          assert_(b.base is a)
167: (12)          assert_equal(b, [a[:, i, i] for i in range(3)])
168: (12)          b = np.einsum("jii->ij", a, optimize=do_opt)
169: (12)          assert_(b.base is a)
170: (12)          assert_equal(b, [a[:, i, i] for i in range(3)])
171: (12)          b = np.einsum(a, [1, 0, 0], [0, 1], optimize=do_opt)
172: (12)          assert_(b.base is a)
173: (12)          assert_equal(b, [a[:, i, i] for i in range(3)])
174: (12)          b = np.einsum("i...->i...", a, optimize=do_opt)
175: (12)          assert_(b.base is a)
176: (12)          assert_equal(b, [a.transpose(2, 0, 1)[:, i, i] for i in range(3)])
177: (12)          b = np.einsum(a, [0, 0, Ellipsis], [0, Ellipsis], optimize=do_opt)
178: (12)          assert_(b.base is a)
179: (12)          assert_equal(b, [a.transpose(2, 0, 1)[:, i, i] for i in range(3)])
180: (12)          b = np.einsum("i...i->i...", a, optimize=do_opt)
181: (12)          assert_(b.base is a)
182: (12)          assert_equal(b, [a.transpose(1, 0, 2)[:, i, i] for i in range(3)])
183: (12)          b = np.einsum(a, [0, Ellipsis, 0], [0, Ellipsis], optimize=do_opt)
184: (12)          assert_(b.base is a)
185: (12)          assert_equal(b, [a.transpose(1, 0, 2)[:, i, i] for i in range(3)])
186: (12)          b = np.einsum("i...i->...i", a, optimize=do_opt)
187: (12)          assert_(b.base is a)
188: (12)          assert_equal(b, [[x[i, i] for i in range(3)]
189: (29)                      for x in a.transpose(1, 0, 2)])
190: (12)          b = np.einsum(a, [0, Ellipsis, 0], [Ellipsis, 0], optimize=do_opt)
191: (12)          assert_(b.base is a)
192: (12)          assert_equal(b, [[x[i, i] for i in range(3)]
193: (29)                      for x in a.transpose(1, 0, 2)])
194: (12)          a = np.arange(27)
195: (12)          a.shape = (3, 3, 3)
```

```

196: (12)                                b = np.einsum("iii->i", a, optimize=do_opt)
197: (12)                                assert_(b.base is a)
198: (12)                                assert_equal(b, [a[i, i, i] for i in range(3)])
199: (12)                                b = np.einsum(a, [0, 0, 0], [0], optimize=do_opt)
200: (12)                                assert_(b.base is a)
201: (12)                                assert_equal(b, [a[i, i, i] for i in range(3)])
202: (12)                                a = np.arange(24)
203: (12)                                a.shape = (2, 3, 4)
204: (12)                                b = np.einsum("ijk->jik", a, optimize=do_opt)
205: (12)                                assert_(b.base is a)
206: (12)                                assert_equal(b, a.swapaxes(0, 1))
207: (12)                                b = np.einsum(a, [0, 1, 2], [1, 0, 2], optimize=do_opt)
208: (12)                                assert_(b.base is a)
209: (12)                                assert_equal(b, a.swapaxes(0, 1))

210: (4) @np._no_nep50_warning()
211: (4) def check_einsum_sums(self, dtype, do_opt=False):
212: (8)     dtype = np.dtype(dtype)
213: (8)     for n in range(1, 17):
214: (12)         a = np.arange(n, dtype=dtype)
215: (12)         b = np.sum(a, axis=-1)
216: (12)         if hasattr(b, 'astype'):
217: (16)             b = b.astype(dtype)
218: (12)         assert_equal(np.einsum("i->", a, optimize=do_opt), b)
219: (12)         assert_equal(np.einsum(a, [0], [], optimize=do_opt), b)
220: (8)     for n in range(1, 17):
221: (12)         a = np.arange(2*3*n, dtype=dtype).reshape(2, 3, n)
222: (12)         b = np.sum(a, axis=-1)
223: (12)         if hasattr(b, 'astype'):
224: (16)             b = b.astype(dtype)
225: (12)         assert_equal(np.einsum("...i->...", a, optimize=do_opt), b)
226: (12)         assert_equal(np.einsum(a, [Ellipsis, 0], [Ellipsis],
227: (8)             optimize=do_opt), b)
228: (12)
229: (12)
230: (12)
231: (16)
232: (12)
233: (12)
234: (8) optimize=do_opt), b)
235: (12)
236: (12)
237: (12)
238: (16)
239: (12)
240: (12)
241: (8) optimize=do_opt), b)
242: (12)
243: (12)
244: (12)
245: (16)
246: (12)
247: (12)
248: (12)
249: (12)
250: (12)
251: (8)
252: (8)
253: (12)
254: (12)
255: (12)
256: (25)
257: (12)
258: (25) optimize=do_opt),
259: (8)
260: (12)

```

```

261: (12)                                b = np.arange(n, dtype=dtype)
262: (12)                                assert_equal(np.einsum("...i, ...i", a, b, optimize=do_opt),
263: (12)  assert_equal(np.einsum(a, [Ellipsis, 0], b, [Ellipsis, 0],
264: (25)  np.inner(a, b)))
265: (8)       for n in range(1, 11):
266: (12)           a = np.arange(n * 3 * 2, dtype=dtype).reshape(n, 3, 2)
267: (12)           b = np.arange(n, dtype=dtype)
268: (12)           assert_equal(np.einsum("i..., i...", a, b, optimize=do_opt),
269: (25)             np.inner(a.T, b.T).T)
270: (12)           assert_equal(np.einsum(a, [0, Ellipsis], b, [0, Ellipsis]),
271: (25)             optimize=do_opt),
272: (8)             np.inner(a.T, b.T).T)
273: (12)       for n in range(1, 17):
274: (12)           a = np.arange(3, dtype=dtype)+1
275: (12)           b = np.arange(n, dtype=dtype)+1
276: (25)           assert_equal(np.einsum("i,j", a, b, optimize=do_opt),
277: (12)             np.outer(a, b))
278: (25)           assert_equal(np.einsum(a, [0], b, [1], optimize=do_opt),
279: (8)             np.outer(a, b))
280: (12)       with suppress_warnings() as sup:
281: (12)           sup.filter(np.ComplexWarning)
282: (16)           for n in range(1, 17):
283: (16)               a = np.arange(4*n, dtype=dtype).reshape(4, n)
284: (16)               b = np.arange(n, dtype=dtype)
285: (29)               assert_equal(np.einsum("ij, j", a, b, optimize=do_opt),
286: (16)                 np.dot(a, b))
287: (29)               assert_equal(np.einsum(a, [0, 1], b, [1], optimize=do_opt),
288: (16)                 np.dot(a, b))
289: (16)               c = np.arange(4, dtype=dtype)
290: (26)               np.einsum("ij,j", a, b, out=c,
291: (16)                 dtype='f8', casting='unsafe', optimize=do_opt)
292: (29)               assert_equal(c,
293: (36)                 np.dot(a.astype('f8'),
294: (16)                   b.astype('f8')).astype(dtype))
295: (16)               c[...] = 0
296: (26)               np.einsum(a, [0, 1], b, [1], out=c,
297: (16)                 dtype='f8', casting='unsafe', optimize=do_opt)
298: (29)               assert_equal(c,
299: (36)                 np.dot(a.astype('f8'),
300: (12)                   b.astype('f8')).astype(dtype))
301: (16)       for n in range(1, 17):
302: (16)           a = np.arange(4*n, dtype=dtype).reshape(4, n)
303: (16)           b = np.arange(n, dtype=dtype)
304: (29)           assert_equal(np.einsum("ji,j", a.T, b.T, optimize=do_opt),
305: (16)             np.dot(b.T, a.T))
306: (29)           assert_equal(np.einsum(a.T, [1, 0], b.T, [1],
307: (16)               np.dot(b.T, a.T)))
308: (16)           c = np.arange(4, dtype=dtype)
309: (26)           np.einsum("ji,j", a.T, b.T, out=c,
310: (16)             dtype='f8', casting='unsafe', optimize=do_opt)
311: (29)           assert_equal(c,
312: (36)             np.dot(b.T.astype('f8'),
313: (16)               a.T.astype('f8')).astype(dtype))
314: (16)           c[...] = 0
315: (26)           np.einsum(a.T, [1, 0], b.T, [1], out=c,
316: (16)             dtype='f8', casting='unsafe', optimize=do_opt)
317: (29)           assert_equal(c,
318: (36)             np.dot(b.T.astype('f8'),
319: (12)               a.T.astype('f8')).astype(dtype))
320: (16)       for n in range(1, 17):
321: (20)           if n < 8 or dtype != 'f2':
322: (20)               a = np.arange(4*n, dtype=dtype).reshape(4, n)
323: (20)               b = np.arange(n*6, dtype=dtype).reshape(n, 6)
324: (33)               assert_equal(np.einsum("ij,jk", a, b, optimize=do_opt),
325: (20)                 np.dot(a, b))
326: (20)               assert_equal(np.einsum(a, [0, 1], b, [1, 2],

```

```

optimize=do_opt),
326: (33)                                     np.dot(a, b))
327: (12)                                     for n in range(1, 17):
328: (16)                                     a = np.arange(4*n, dtype=dtype).reshape(4, n)
329: (16)                                     b = np.arange(n*6, dtype=dtype).reshape(n, 6)
330: (16)                                     c = np.arange(24, dtype=dtype).reshape(4, 6)
331: (16)                                     np.einsum("ij,jk", a, b, out=c, dtype='f8', casting='unsafe',
332: (26)   optimize=do_opt)
333: (16)                                     assert_equal(c,
334: (29)   np.dot(a.astype('f8'),
335: (36)   b.astype('f8')).astype(dtype))
336: (16)                                     c[...] = 0
337: (16)                                     np.einsum(a, [0, 1], b, [1, 2], out=c,
338: (26)   dtype='f8', casting='unsafe', optimize=do_opt)
339: (16)                                     assert_equal(c,
340: (29)   np.dot(a.astype('f8'),
341: (36)   b.astype('f8')).astype(dtype))
342: (12)                                     a = np.arange(12, dtype=dtype).reshape(3, 4)
343: (12)                                     b = np.arange(20, dtype=dtype).reshape(4, 5)
344: (12)                                     c = np.arange(30, dtype=dtype).reshape(5, 6)
345: (12)                                     if dtype != 'f2':
346: (16)   assert_equal(np.einsum("ij,jk,kl", a, b, c, optimize=do_opt),
347: (29)   a.dot(b).dot(c))
348: (16)   assert_equal(np.einsum(a, [0, 1], b, [1, 2], c, [2, 3],
349: (39)   optimize=do_opt), a.dot(b).dot(c))
350: (12)                                     d = np.arange(18, dtype=dtype).reshape(3, 6)
351: (12)                                     np.einsum("ij,jk,kl", a, b, c, out=d,
352: (22)   dtype='f8', casting='unsafe', optimize=do_opt)
353: (12)                                     tgt = a.astype('f8').dot(b.astype('f8'))
354: (12)                                     tgt = tgt.dot(c.astype('f8')).astype(dtype)
355: (12)                                     assert_equal(d, tgt)
356: (12)                                     d[...] = 0
357: (12)                                     np.einsum(a, [0, 1], b, [1, 2], c, [2, 3], out=d,
358: (22)   dtype='f8', casting='unsafe', optimize=do_opt)
359: (12)                                     tgt = a.astype('f8').dot(b.astype('f8'))
360: (12)                                     tgt = tgt.dot(c.astype('f8')).astype(dtype)
361: (12)                                     assert_equal(d, tgt)
362: (12)                                     if np.dtype(dtype) != np.dtype('f2'):
363: (16)   a = np.arange(60, dtype=dtype).reshape(3, 4, 5)
364: (16)   b = np.arange(24, dtype=dtype).reshape(4, 3, 2)
365: (16)   assert_equal(np.einsum("ijk, jil -> k1", a, b),
366: (29)   np.tensordot(a, b, axes=[[1, 0], [0, 1]]))
367: (16)   assert_equal(np.einsum(a, [0, 1, 2], b, [1, 0, 3], [2, 3]),
368: (29)   np.tensordot(a, b, axes=[[1, 0], [0, 1]]))
369: (16)   c = np.arange(10, dtype=dtype).reshape(5, 2)
370: (16)   np.einsum("ijk,jil->k1", a, b, out=c,
371: (26)   dtype='f8', casting='unsafe', optimize=do_opt)
372: (16)   assert_equal(c, np.tensordot(a.astype('f8'), b.astype('f8'),
373: (29)   axes=[[1, 0], [0, 1]]).astype(dtype))
374: (16)   c[...] = 0
375: (16)   np.einsum(a, [0, 1, 2], b, [1, 0, 3], [2, 3], out=c,
376: (26)   dtype='f8', casting='unsafe', optimize=do_opt)
377: (16)   assert_equal(c, np.tensordot(a.astype('f8'), b.astype('f8'),
378: (29)   axes=[[1, 0], [0, 1]]).astype(dtype))
379: (8)   neg_val = -2 if dtype.kind != "u" else np.iinfo(dtype).max - 1
380: (8)   a = np.array([1, 3, neg_val, 0, 12, 13, 0, 1], dtype=dtype)
381: (8)   b = np.array([0, 3.5, 0., neg_val, 0, 1, 3, 12],
382: (8)   dtype=dtype)
383: (8)   c = np.array([True, True, False, True, True, False, True, True])
384: (21)   assert_equal(np.einsum("i,i,i->i", a, b, c,
385: (21)   dtype='?', casting='unsafe', optimize=do_opt),
386: (8)   np.logical_and(np.logical_and(a != 0, b != 0), c != 0))
387: (21)   assert_equal(np.einsum(a, [0], b, [0], c, [0], [0],
388: (21)   dtype='?', casting='unsafe'),
389: (8)   np.logical_and(np.logical_and(a != 0, b != 0), c != 0))
390: (8)   a = np.arange(9, dtype=dtype)
391: (8)   assert_equal(np.einsum(",i->", 3, a), 3*np.sum(a))
392: (8)   assert_equal(np.einsum(3, [], a, [0], []), 3*np.sum(a))
393: (8)   assert_equal(np.einsum("i,->", a, 3), 3*np.sum(a))

```

```

393: (8) assert_equal(np.einsum(a, [0], 3, [], []), 3*np.sum(a))
394: (8) for n in range(1, 25):
395: (12)     a = np.arange(n, dtype=dtype)
396: (12)     if np.dtype(dtype).itemsize > 1:
397: (16)         assert_equal(np.einsum("....", a, a, optimize=do_opt),
398: (29)             np.multiply(a, a))
399: (16)         assert_equal(np.einsum("i,i", a, a, optimize=do_opt),
400: (16)             assert_equal(np.einsum("i,->i", a, 2, optimize=do_opt), 2*a)
401: (16)             assert_equal(np.einsum(",i->i", 2, a, optimize=do_opt), 2*a)
402: (16)             assert_equal(np.einsum("i,->", a, 2, optimize=do_opt),
403: (16)                 assert_equal(np.einsum(",i->", 2, a, optimize=do_opt),
404: (16)                     assert_equal(np.einsum("....", a[1:], a[:-1],
405: (29)                         np.multiply(a[1:], a[:-1]))
406: (16)                     assert_equal(np.einsum("i,i", a[1:], a[:-1], optimize=do_opt),
407: (29)                         np.dot(a[1:], a[:-1]))
408: (16)                     assert_equal(np.einsum("i,->i", a[1:], 2, optimize=do_opt),
409: (16)                         assert_equal(np.einsum(",i->i", 2, a[1:], optimize=do_opt),
410: (16)                             assert_equal(np.einsum("i,->", a[1:], 2, optimize=do_opt),
411: (29)                                 2*np.sum(a[1:]))
412: (16)                             assert_equal(np.einsum(",i->", 2, a[1:], optimize=do_opt),
413: (29)                                 2*np.sum(a[1:]))
414: (8)     a = np.arange(9, dtype=object)
415: (8)     b = np.einsum("i->", a, dtype=dtype, casting='unsafe')
416: (8)     assert_equal(b, np.sum(a))
417: (8)     if hasattr(b, "dtype"):
418: (12)         assert_equal(b.dtype, np.dtype(dtype))
419: (8)     b = np.einsum(a, [0], [], dtype=dtype, casting='unsafe')
420: (8)     assert_equal(b, np.sum(a))
421: (8)     if hasattr(b, "dtype"):
422: (12)         assert_equal(b.dtype, np.dtype(dtype))
423: (8)     p = np.arange(2) + 1
424: (8)     q = np.arange(4).reshape(2, 2) + 3
425: (8)     r = np.arange(4).reshape(2, 2) + 7
426: (8)     assert_equal(np.einsum('z,mz,zm->', p, q, r), 253)
427: (8)     p = np.ones((10,2))
428: (8)     q = np.ones((1,2))
429: (8)     assert_array_equal(np.einsum('ij,ij->j', p, q, optimize=True),
430: (27)                     np.einsum('ij,ij->j', p, q, optimize=False))
431: (8)     assert_array_equal(np.einsum('ij,ij->j', p, q, optimize=True),
432: (27)                     [10.] * 2)
433: (8)     x = np.array([2., 3.])
434: (8)     y = np.array([4.])
435: (8)     assert_array_equal(np.einsum("i, i", x, y, optimize=False), 20.)
436: (8)     assert_array_equal(np.einsum("i, i", x, y, optimize=True), 20.)
437: (8)     p = np.ones((1, 5)) / 2
438: (8)     q = np.ones((5, 5)) / 2
439: (8)     for optimize in (True, False):
440: (12)         assert_array_equal(np.einsum("...ij,...jk->...ik", p, p,
441: (41)                         optimize=optimize),
442: (31)                         np.einsum("...ij,...jk->...ik", p, q,
443: (41)                         optimize=optimize))
444: (12)         assert_array_equal(np.einsum("...ij,...jk->...ik", p, q,
445: (41)                         optimize=optimize),
446: (31)                         np.full((1, 5), 1.25))
447: (8)     x = np.eye(2, dtype=dtype)
448: (8)     y = np.ones(2, dtype=dtype)
449: (8)     assert_array_equal(np.einsum("ji,i->", x, y, optimize=optimize),
450: (27)                     [2.]) # contig_contig_outstride0_two
451: (8)     assert_array_equal(np.einsum("i,ij->", y, x, optimize=optimize),
452: (27)                     [2.]) # stride0_contig_outstride0_two
453: (8)     assert_array_equal(np.einsum("ij,i->", x, y, optimize=optimize),
454: (27)                     [2.]) # contig_stride0_outstride0_two
455: (4) def test_einsum_sums_int8(self):

```

```

456: (8)             if (
457: (16)                 (sys.platform == 'darwin' and platform.machine() == 'x86_64')
458: (16)                     or
459: (16)                         USING_CLANG_CL
460: (8)             ):
461: (12)                 pytest.xfail('Fails on macOS x86-64 and when using clang-cl '
462: (25)                             'with Meson, see gh-23838')
463: (8)             self.check_einsum_sums('i1')
464: (4)             def test_einsum_sums_uint8(self):
465: (8)                 if (
466: (16)                     (sys.platform == 'darwin' and platform.machine() == 'x86_64')
467: (16)                         or
468: (16)                             USING_CLANG_CL
469: (8)             ):
470: (12)                 pytest.xfail('Fails on macOS x86-64 and when using clang-cl '
471: (25)                             'with Meson, see gh-23838')
472: (8)             self.check_einsum_sums('u1')
473: (4)             def test_einsum_sums_int16(self):
474: (8)                 self.check_einsum_sums('i2')
475: (4)             def test_einsum_sums_uint16(self):
476: (8)                 self.check_einsum_sums('u2')
477: (4)             def test_einsum_sums_int32(self):
478: (8)                 self.check_einsum_sums('i4')
479: (8)                 self.check_einsum_sums('i4', True)
480: (4)             def test_einsum_sums_uint32(self):
481: (8)                 self.check_einsum_sums('u4')
482: (8)                 self.check_einsum_sums('u4', True)
483: (4)             def test_einsum_sums_int64(self):
484: (8)                 self.check_einsum_sums('i8')
485: (4)             def test_einsum_sums_uint64(self):
486: (8)                 self.check_einsum_sums('u8')
487: (4)             def test_einsum_sums_float16(self):
488: (8)                 self.check_einsum_sums('f2')
489: (4)             def test_einsum_sums_float32(self):
490: (8)                 self.check_einsum_sums('f4')
491: (4)             def test_einsum_sums_float64(self):
492: (8)                 self.check_einsum_sums('f8')
493: (8)                 self.check_einsum_sums('f8', True)
494: (4)             def test_einsum_sums_longdouble(self):
495: (8)                 self.check_einsum_sums(np.longdouble)
496: (4)             def test_einsum_sums_cfloat64(self):
497: (8)                 self.check_einsum_sums('c8')
498: (8)                 self.check_einsum_sums('c8', True)
499: (4)             def test_einsum_sums_cfloat128(self):
500: (8)                 self.check_einsum_sums('c16')
501: (4)             def test_einsum_sums_clongdouble(self):
502: (8)                 self.check_einsum_sums(np.clongdouble)
503: (4)             def test_einsum_sums_object(self):
504: (8)                 self.check_einsum_sums('object')
505: (8)                 self.check_einsum_sums('object', True)
506: (4)             def test_einsum_misc(self):
507: (8)                 a = np.ones((1, 2))
508: (8)                 b = np.ones((2, 2, 1))
509: (8)                 assert_equal(np.einsum('ij...,j...->i...', a, b), [[[2], [2]]])
510: (8)                 assert_equal(np.einsum('ij...,j...->i...', a, b, optimize=True),
511: [[[[2], [2]]]])
512: (8)                 assert_equal(np.einsum('ij...,j...->i...', a, b), [[[2], [2]]]))
513: (8)                 assert_equal(np.einsum('...i,...i', [1, 2, 3], [2, 3, 4]), 20)
514: (31)                 assert_equal(np.einsum('...i,...i', [1, 2, 3], [2, 3, 4],
515: (8)                                 optimize='greedy'), 20)
516: (8)                 a = np.ones((5, 12, 4, 2, 3), np.int64)
517: (8)                 b = np.ones((5, 12, 11), np.int64)
518: (21)                 assert_equal(np.einsum('ijklm,ijn,ijn->', a, b, b),
519: (8)                               np.einsum('ijklm,ijn->', a, b))
520: (21)                 assert_equal(np.einsum('ijklm,ijn,ijn->', a, b, b, optimize=True),
521: (8)                               np.einsum('ijklm,ijn->', a, b, optimize=True))
522: (8)                 a = np.arange(1, 3)
523: (8)                 b = np.arange(1, 5).reshape(2, 2)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

524: (8) assert_equal(np.einsum('x,yx,zx->xzy', a, b, c),
525: (21)           [[[1, 3], [3, 9], [5, 15], [7, 21]],
526: (21)           [[8, 16], [16, 32], [24, 48], [32, 64]]])
527: (8) assert_equal(np.einsum('x,yx,zx->xzy', a, b, c, optimize=True),
528: (21)           [[[1, 3], [3, 9], [5, 15], [7, 21]],
529: (21)           [[8, 16], [16, 32], [24, 48], [32, 64]]])
530: (8) assert_equal(np.einsum('i,j', [1], [2], out=None), [[2]])
531: (4) def test_object_loop(self):
532: (8)     class Mult:
533: (12)         def __mul__(self, other):
534: (16)             return 42
535: (8)     objMult = np.array([Mult()])
536: (8)     objNULL = np.ndarray(buffer = b'\0' * np.intp(0).itemsize, shape=1,
537: (8)     dtype=object)
538: (12) assert_raises(TypeError, np.einsum, "i,j", [1], objNULL)
539: (8) with pytest.raises(TypeError):
540: (12)     np.einsum("i,j", objNULL, [1])
541: (8) assert np.einsum("i,j", objMult, objMult) == 42
542: (4) def test_subscript_range(self):
543: (8)     a = np.ones((2, 3))
544: (8)     b = np.ones((3, 4))
545: (8)     np.einsum(a, [0, 20], b, [20, 2], [0, 2], optimize=False)
546: (8)     np.einsum(a, [0, 27], b, [27, 2], [0, 2], optimize=False)
547: (8)     np.einsum(a, [0, 51], b, [51, 2], [0, 2], optimize=False)
548: (8)     assert_raises(ValueError, lambda: np.einsum(a, [0, 52], b, [52, 2],
549: (8)     [0, 2], optimize=False))
549: (8)     assert_raises(ValueError, lambda: np.einsum(a, [-1, 5], b, [5, 2],
550: (4)     [-1, 2], optimize=False))
550: (4)     def test_einsum_broadcast(self):
551: (8)         A = np.arange(2 * 3 * 4).reshape(2, 3, 4)
552: (8)         B = np.arange(3)
553: (8)         ref = np.einsum('ijk,j->ijk', A, B, optimize=False)
554: (8)         for opt in [True, False]:
555: (12)             assert_equal(np.einsum('ij...,j...->ij...', A, B, optimize=opt),
556: (12)             assert_equal(np.einsum('ij...,...,j->ij...', A, B, optimize=opt),
557: (12)             assert_equal(np.einsum('ij...,j->ij...', A, B, optimize=opt), ref)

# used to raise error
558: (8) A = np.arange(12).reshape((4, 3))
559: (8) B = np.arange(6).reshape((3, 2))
560: (8) ref = np.einsum('ik,kj->ij', A, B, optimize=False)
561: (8) for opt in [True, False]:
562: (12)     assert_equal(np.einsum('ik...,k...->i...', A, B, optimize=opt),
563: (12)     assert_equal(np.einsum('ik...,...,kj->i...j', A, B, optimize=opt),
564: (12)     assert_equal(np.einsum('...k,kj', A, B, optimize=opt), ref) #
565: (12)     assert_equal(np.einsum('ik,k...->i...', A, B, optimize=opt), ref)

# used to raise error
566: (8) dims = [2, 3, 4, 5]
567: (8) a = np.arange(np.prod(dims)).reshape(dims)
568: (8) v = np.arange(dims[2])
569: (8) ref = np.einsum('ijkl,k->ijl', a, v, optimize=False)
570: (8) for opt in [True, False]:
571: (12)     assert_equal(np.einsum('ijkl,k', a, v, optimize=opt), ref)
572: (12)     assert_equal(np.einsum('...kl,k', a, v, optimize=opt), ref) #

573: (12)     assert_equal(np.einsum('...kl,k...', a, v, optimize=opt), ref)
574: (8) J, K, M = 160, 160, 120
575: (8) A = np.arange(J * K * M).reshape(1, 1, 1, J, K, M)
576: (8) B = np.arange(J * K * M * 3).reshape(J, K, M, 3)
577: (8) ref = np.einsum('...lmn,...lmno->...o', A, B, optimize=False)
578: (8) for opt in [True, False]:
579: (12)     assert_equal(np.einsum('...lmn,lmno->...o', A, B,
580: (35)                                     optimize=opt), ref) # used to raise error
581: (4) def test_einsum_fixedstridebug(self):

```

```

582: (8)         A = np.arange(2 * 3).reshape(2, 3).astype(np.float32)
583: (8)         B = np.arange(2 * 3 * 2731).reshape(2, 3, 2731).astype(np.int16)
584: (8)         es = np.einsum('cl, cpx->lp', A, B)
585: (8)         tp = np.tensordot(A, B, axes=(0, 0))
586: (8)         assert_equal(es, tp)
587: (8)         A = np.arange(3 * 3).reshape(3, 3).astype(np.float64)
588: (8)         B = np.arange(3 * 3 * 64 * 64).reshape(3, 3, 64,
64).astype(np.float32)
589: (8)         es = np.einsum('cl, cpxy->lp', A, B)
590: (8)         tp = np.tensordot(A, B, axes=(0, 0))
591: (8)         assert_equal(es, tp)
592: (4)         def test_einsum_fixed_collapsebug(self):
593: (8)             x = np.random.normal(0, 1, (5, 5, 5, 5))
594: (8)             y1 = np.zeros((5, 5))
595: (8)             np.einsum('aabb->ab', x, out=y1)
596: (8)             idx = np.arange(5)
597: (8)             y2 = x[idx[:, None], idx[:, None], idx, idx]
598: (8)             assert_equal(y1, y2)
599: (4)         def test_einsum_failed_on_p9_and_s390x(self):
600: (8)             tensor = np.random.random_sample((10, 10, 10, 10))
601: (8)             x = np.einsum('ijij->', tensor)
602: (8)             y = tensor.trace(axis1=0, axis2=2).trace()
603: (8)             assert_allclose(x, y)
604: (4)         def test_einsum_all_contig_non_contig_output(self):
605: (8)             x = np.ones((5, 5))
606: (8)             out = np.ones(10)[::2]
607: (8)             correct_base = np.ones(10)
608: (8)             correct_base[::2] = 5
609: (8)             np.einsum('mi,mi,mi->m', x, x, x, out=out)
610: (8)             assert_array_equal(out.base, correct_base)
611: (8)             out = np.ones(10)[::2]
612: (8)             np.einsum('im,im,im->m', x, x, x, out=out)
613: (8)             assert_array_equal(out.base, correct_base)
614: (8)             out = np.ones((2, 2, 2))[..., 0]
615: (8)             correct_base = np.ones((2, 2, 2))
616: (8)             correct_base[..., 0] = 2
617: (8)             x = np.ones((2, 2), np.float32)
618: (8)             np.einsum('ij,jk->ik', x, x, out=out)
619: (8)             assert_array_equal(out.base, correct_base)
620: (4)         @pytest.mark.parametrize("dtype",
621: (13)             np.typecodes["AllFloat"] + np.typecodes["AllInteger"])
622: (4)         def test_different_paths(self, dtype):
623: (8)             dtype = np.dtype(dtype)
624: (8)             arr = (np.arange(7) + 0.5).astype(dtype)
625: (8)             scalar = np.array(2, dtype=dtype)
626: (8)             res = np.einsum('i->', arr)
627: (8)             assert res == arr.sum()
628: (8)             res = np.einsum('i,i->i', arr, arr)
629: (8)             assert_array_equal(res, arr * arr)
630: (8)             res = np.einsum('i,i->i', arr.repeat(2)[::2], arr.repeat(2)[::2])
631: (8)             assert_array_equal(res, arr * arr)
632: (8)             assert np.einsum('i,i->', arr, arr) == (arr * arr).sum()
633: (8)             out = np.ones(7, dtype=dtype)
634: (8)             res = np.einsum('i,->i', arr, dtype.type(2), out=out)
635: (8)             assert_array_equal(res, arr * dtype.type(2))
636: (8)             res = np.einsum(',i->i', scalar, arr)
637: (8)             assert_array_equal(res, arr * dtype.type(2))
638: (8)             res = np.einsum(',i->', scalar, arr)
639: (8)             assert res == np.einsum('i->', scalar * arr)
640: (8)             res = np.einsum('i,->', arr, scalar)
641: (8)             assert res == np.einsum('i->', scalar * arr)
642: (8)             arr = np.array([0.5, 0.5, 0.25, 4.5, 3.], dtype=dtype)
643: (8)             res = np.einsum('i,i,i->', arr, arr, arr)
644: (8)             assert_array_equal(res, (arr * arr * arr).sum())
645: (8)             res = np.einsum('i,i,i,i->', arr, arr, arr, arr)
646: (8)             assert_array_equal(res, (arr * arr * arr * arr).sum())
647: (4)         def test_small_boolean_arrays(self):
648: (8)             a = np.zeros((16, 1, 1), dtype=np.bool_)[::2]
649: (8)             a[...] = True

```

```

650: (8)          out = np.zeros((16, 1, 1), dtype=np.bool_)[::2]
651: (8)          tgt = np.ones((2, 1, 1), dtype=np.bool_)
652: (8)          res = np.einsum('...ij,...jk->...ik', a, a, out=out)
653: (8)          assert_equal(res, tgt)
654: (4)          def test_out_is_res(self):
655: (8)              a = np.arange(9).reshape(3, 3)
656: (8)              res = np.einsum('...ij,...jk->...ik', a, a, out=a)
657: (8)              assert res is a
658: (4)          def optimize_compare(self, subscripts, operands=None):
659: (8)              if operands is None:
660: (12)                  args = [subscripts]
661: (12)                  terms = subscripts.split('->')[0].split(',')
662: (12)                  for term in terms:
663: (16)                      dims = [global_size_dict[x] for x in term]
664: (16)                      args.append(np.random.rand(*dims))
665: (8)              else:
666: (12)                  args = [subscripts] + operands
667: (8)              noopt = np.einsum(*args, optimize=False)
668: (8)              opt = np.einsum(*args, optimize='greedy')
669: (8)              assert_almost_equal(opt, noopt)
670: (8)              opt = np.einsum(*args, optimize='optimal')
671: (8)              assert_almost_equal(opt, noopt)
672: (4)          def test_hadamard_like_products(self):
673: (8)              self.optimize_compare('a,ab,abc->abc')
674: (8)              self.optimize_compare('a,b,ab->ab')
675: (4)          def test_index_transformations(self):
676: (8)              self.optimize_compare('ea,fb,gc,hd,abcd->efgh')
677: (8)              self.optimize_compare('ea,fb,abcd,gc,hd->efgh')
678: (8)              self.optimize_compare('abcd,ea,fb,gc,hd->efgh')
679: (4)          def test_complex(self):
680: (8)              self.optimize_compare('acdf,jbje,gihb,hfac,gifabc,hfac')
681: (8)              self.optimize_compare('acdf,jbje,gihb,hfac,gfac,gifabc,hfac')
682: (8)              self.optimize_compare('cd,bdhe,aidb,hgca,gc,hgibcd,hgac')
683: (8)              self.optimize_compare('abhe,hidj,jgba,hiab,gab')
684: (8)              self.optimize_compare('bde,cdh,agdb,hica,ibd,hgicd,hiac')
685: (8)              self.optimize_compare('chd,bde,agbc,hiad,hgc,hgi,hiad')
686: (8)              self.optimize_compare('chd,bde,agbc,hiad,bdi,cgh,agdb')
687: (8)              self.optimize_compare('bdhe,acad,hiab,agac,habd')
688: (4)          def test_collapse(self):
689: (8)              self.optimize_compare('ab,ab,c->')
690: (8)              self.optimize_compare('ab,ab,c->c')
691: (8)              self.optimize_compare('ab,ab,cd,cd->')
692: (8)              self.optimize_compare('ab,ab,cd,cd->ac')
693: (8)              self.optimize_compare('ab,ab,cd,cd->cd')
694: (8)              self.optimize_compare('ab,ab,cd,cd,ef,ef->')
695: (4)          def test_expand(self):
696: (8)              self.optimize_compare('ab,cd,ef->abcdef')
697: (8)              self.optimize_compare('ab,cd,ef->acdf')
698: (8)              self.optimize_compare('ab,cd,de->abcde')
699: (8)              self.optimize_compare('ab,cd,de->be')
700: (8)              self.optimize_compare('ab,bcd,cd->abcd')
701: (8)              self.optimize_compare('ab,bcd,cd->abd')
702: (4)          def test_edge_cases(self):
703: (8)              self.optimize_compare('eb,cb,fb->cef')
704: (8)              self.optimize_compare('dd,fb,be,cdb->cef')
705: (8)              self.optimize_compare('bca,cdb,dbf,afc->')
706: (8)              self.optimize_compare('dcc,fce,ea,dbf->ab')
707: (8)              self.optimize_compare('fdf,cdd,ccd,afe->ae')
708: (8)              self.optimize_compare('abcd,ad')
709: (8)              self.optimize_compare('ed,fcd,ff,bcf->be')
710: (8)              self.optimize_compare('baa,dcf,af,cde->be')
711: (8)              self.optimize_compare('bd,db,eac->ace')
712: (8)              self.optimize_compare('ffff,fae,bef,def->abd')
713: (8)              self.optimize_compare('efc,dbc,acf,fd->abe')
714: (8)              self.optimize_compare('ba,ac,da->bcd')
715: (4)          def test_inner_product(self):
716: (8)              self.optimize_compare('ab,ab')
717: (8)              self.optimize_compare('ab,ba')
718: (8)              self.optimize_compare('abc,abc')

```

```

719: (8)           self.optimize_compare('abc,bac')
720: (8)           self.optimize_compare('abc,cba')
721: (4) def test_random_cases(self):
722: (8)           self.optimize_compare('aab,fa,df,ecc->bde')
723: (8)           self.optimize_compare('ecb,fef,bad,ed->ac')
724: (8)           self.optimize_compare('bcf,bbb,fbf,fc->')
725: (8)           self.optimize_compare('bb,ff,be->e')
726: (8)           self.optimize_compare('bcb,bb,fc,fff->')
727: (8)           self.optimize_compare('fbb,dfd,fc,fc->')
728: (8)           self.optimize_compare('afd,ba,cc,dc->bf')
729: (8)           self.optimize_compare('adb,bc,fa,cfc->d')
730: (8)           self.optimize_compare('bbd,bda,fc,db->acf')
731: (8)           self.optimize_compare('dba,ead,cad->bce')
732: (8)           self.optimize_compare('aef,fbc,dca->bde')
733: (4) def test_combined_views_mapping(self):
734: (8)           a = np.arange(9).reshape(1, 1, 3, 1, 3)
735: (8)           b = np.einsum('bbcde->d', a)
736: (8)           assert_equal(b, [12])
737: (4) def test_broadcasting_dot_cases(self):
738: (8)           a = np.random.rand(1, 5, 4)
739: (8)           b = np.random.rand(4, 6)
740: (8)           c = np.random.rand(5, 6)
741: (8)           d = np.random.rand(10)
742: (8)           self.optimize_compare('ijk,kl,jl', operands=[a, b, c])
743: (8)           self.optimize_compare('ijk,kl,jl,i->i', operands=[a, b, c, d])
744: (8)           e = np.random.rand(1, 1, 5, 4)
745: (8)           f = np.random.rand(7, 7)
746: (8)           self.optimize_compare('abjk,kl,jl', operands=[e, b, c])
747: (8)           self.optimize_compare('abjk,kl,jl,ab->ab', operands=[e, b, c, f])
748: (8)           g = np.arange(64).reshape(2, 4, 8)
749: (8)           self.optimize_compare('obk,ijk->ioj', operands=[g, g])
750: (4) def test_output_order(self):
751: (8)           a = np.ones((2, 3, 5), order='F')
752: (8)           b = np.ones((4, 3), order='F')
753: (8)           for opt in [True, False]:
754: (12)             tmp = np.einsum('...ft, mf->...mt', a, b, order='a', optimize=opt)
755: (12)             assert_(tmp.flags.f_contiguous)
756: (12)             tmp = np.einsum('...ft, mf->...mt', a, b, order='f', optimize=opt)
757: (12)             assert_(tmp.flags.f_contiguous)
758: (12)             tmp = np.einsum('...ft, mf->...mt', a, b, order='c', optimize=opt)
759: (12)             assert_(tmp.flags.c_contiguous)
760: (12)             tmp = np.einsum('...ft, mf->...mt', a, b, order='k', optimize=opt)
761: (12)             assert_(tmp.flags.c_contiguous is False)
762: (12)             assert_(tmp.flags.f_contiguous is False)
763: (12)             tmp = np.einsum('...ft, mf->...mt', a, b, optimize=opt)
764: (12)             assert_(tmp.flags.c_contiguous is False)
765: (12)             assert_(tmp.flags.f_contiguous is False)
766: (8)           c = np.ones((4, 3), order='C')
767: (8)           for opt in [True, False]:
768: (12)             tmp = np.einsum('...ft, mf->...mt', a, c, order='a', optimize=opt)
769: (12)             assert_(tmp.flags.c_contiguous)
770: (8)             d = np.ones((2, 3, 5), order='C')
771: (8)             for opt in [True, False]:
772: (12)               tmp = np.einsum('...ft, mf->...mt', d, c, order='a', optimize=opt)
773: (12)               assert_(tmp.flags.c_contiguous)
774: (0) class TestEinsumPath:
775: (4)     def build_operands(self, string, size_dict=global_size_dict):
776: (8)         operands = [string]
777: (8)         terms = string.split('>')[0].split(',')
778: (8)         for term in terms:
779: (12)             dims = [size_dict[x] for x in term]
780: (12)             operands.append(np.random.rand(*dims))
781: (8)         return operands
782: (4)     def assert_path_equal(self, comp, benchmark):
783: (8)         ret = (len(comp) == len(benchmark))
784: (8)         assert_(ret)
785: (8)         for pos in range(len(comp) - 1):
786: (12)             ret &= isinstance(comp[pos + 1], tuple)
787: (12)             ret &= (comp[pos + 1] == benchmark[pos + 1])

```

```

788: (8)           assert_(ret)
789: (4)           def test_memory_constraints(self):
790: (8)             outer_test = self.build_operands('a,b,c->abc')
791: (8)             path, path_str = np.einsum_path(*outer_test, optimize=('greedy', 0))
792: (8)             self.assert_path_equal(path, ['einsum_path', (0, 1, 2)])
793: (8)             path, path_str = np.einsum_path(*outer_test, optimize=('optimal', 0))
794: (8)             self.assert_path_equal(path, ['einsum_path', (0, 1, 2)])
795: (8)             long_test = self.build_operands('acdf,jbje,gihb,hfac')
796: (8)             path, path_str = np.einsum_path(*long_test, optimize=('greedy', 0))
797: (8)             self.assert_path_equal(path, ['einsum_path', (0, 1, 2, 3)])
798: (8)             path, path_str = np.einsum_path(*long_test, optimize=('optimal', 0))
799: (8)             self.assert_path_equal(path, ['einsum_path', (0, 1, 2, 3)])
800: (4)           def test_long_paths(self):
801: (8)             long_test1 =
self.build_operands('acdf,jbje,gihb,hfac,gifabc,hfac')
802: (8)             path, path_str = np.einsum_path(*long_test1, optimize='greedy')
803: (8)             self.assert_path_equal(path, ['einsum_path',
804: (38)                           (3, 6), (3, 4), (2, 4), (2, 3), (0, 2),
(0, 1)])
805: (8)             path, path_str = np.einsum_path(*long_test1, optimize='optimal')
806: (8)             self.assert_path_equal(path, ['einsum_path',
807: (38)                           (3, 6), (3, 4), (2, 4), (2, 3), (0, 2),
(0, 1)])
808: (8)             long_test2 = self.build_operands('chd,bde,agbc,hiad,bdi,cgh,agdb')
809: (8)             path, path_str = np.einsum_path(*long_test2, optimize='greedy')
810: (8)             self.assert_path_equal(path, ['einsum_path',
811: (38)                           (3, 4), (0, 3), (3, 4), (1, 3), (1, 2),
(0, 1)])
812: (8)             path, path_str = np.einsum_path(*long_test2, optimize='optimal')
813: (8)             self.assert_path_equal(path, ['einsum_path',
814: (38)                           (0, 5), (1, 4), (3, 4), (1, 3), (1, 2),
(0, 1)])
815: (4)           def test_edge_paths(self):
816: (8)             edge_test1 = self.build_operands('eb,cb,fb->cef')
817: (8)             path, path_str = np.einsum_path(*edge_test1, optimize='greedy')
818: (8)             self.assert_path_equal(path, ['einsum_path', (0, 2), (0, 1)])
819: (8)             path, path_str = np.einsum_path(*edge_test1, optimize='optimal')
820: (8)             self.assert_path_equal(path, ['einsum_path', (0, 2), (0, 1)])
821: (8)             edge_test2 = self.build_operands('dd,fb,be,cdb->cef')
822: (8)             path, path_str = np.einsum_path(*edge_test2, optimize='greedy')
823: (8)             self.assert_path_equal(path, ['einsum_path', (0, 3), (0, 1), (0, 1)])
824: (8)             path, path_str = np.einsum_path(*edge_test2, optimize='optimal')
825: (8)             self.assert_path_equal(path, ['einsum_path', (0, 3), (0, 1), (0, 1)])
826: (8)             edge_test3 = self.build_operands('bca,cdb,dbf,afc->')
827: (8)             path, path_str = np.einsum_path(*edge_test3, optimize='greedy')
828: (8)             self.assert_path_equal(path, ['einsum_path', (1, 2), (0, 2), (0, 1)])
829: (8)             path, path_str = np.einsum_path(*edge_test3, optimize='optimal')
830: (8)             self.assert_path_equal(path, ['einsum_path', (1, 2), (0, 2), (0, 1)])
831: (8)             edge_test4 = self.build_operands('dcc,fce,ea,dbf->ab')
832: (8)             path, path_str = np.einsum_path(*edge_test4, optimize='greedy')
833: (8)             self.assert_path_equal(path, ['einsum_path', (1, 2), (0, 1), (0, 1)])
834: (8)             path, path_str = np.einsum_path(*edge_test4, optimize='optimal')
835: (8)             self.assert_path_equal(path, ['einsum_path', (1, 2), (0, 2), (0, 1)])
836: (8)             edge_test4 = self.build_operands('a,ac,ab,ad,cd,bd,bc->',
837: (41)                           size_dict={"a": 20, "b": 20, "c": 20,
"d": 20})
838: (8)             path, path_str = np.einsum_path(*edge_test4, optimize='greedy')
839: (8)             self.assert_path_equal(path, ['einsum_path', (0, 1), (0, 1, 2, 3, 4,
5)])
840: (8)             path, path_str = np.einsum_path(*edge_test4, optimize='optimal')
841: (8)             self.assert_path_equal(path, ['einsum_path', (0, 1), (0, 1, 2, 3, 4,
5)])
842: (4)           def test_path_type_input(self):
843: (8)             path_test = self.build_operands('dcc,fce,ea,dbf->ab')
844: (8)             path, path_str = np.einsum_path(*path_test, optimize=False)
845: (8)             self.assert_path_equal(path, ['einsum_path', (0, 1, 2, 3)])
846: (8)             path, path_str = np.einsum_path(*path_test, optimize=True)
847: (8)             self.assert_path_equal(path, ['einsum_path', (1, 2), (0, 1), (0, 1)])
848: (8)             exp_path = ['einsum_path', (0, 2), (0, 2), (0, 1)]

```

```

849: (8)             path, path_str = np.einsum_path(*path_test, optimize=exp_path)
850: (8)             self.assert_path_equal(path, exp_path)
851: (8)             noopt = np.einsum(*path_test, optimize=False)
852: (8)             opt = np.einsum(*path_test, optimize=exp_path)
853: (8)             assert_almost_equal(noopt, opt)
854: (4)             def test_path_type_input_internal_trace(self):
855: (8)                 path_test = self.build_operands('cab,cdd->ab')
856: (8)                 exp_path = ['einsum_path', (1,), (0, 1)]
857: (8)                 path, path_str = np.einsum_path(*path_test, optimize=exp_path)
858: (8)                 self.assert_path_equal(path, exp_path)
859: (8)                 noopt = np.einsum(*path_test, optimize=False)
860: (8)                 opt = np.einsum(*path_test, optimize=exp_path)
861: (8)                 assert_almost_equal(noopt, opt)
862: (4)             def test_path_type_input_invalid(self):
863: (8)                 path_test = self.build_operands('ab,bc,cd,de->ae')
864: (8)                 exp_path = ['einsum_path', (2, 3), (0, 1)]
865: (8)                 assert_raises(RuntimeError, np.einsum, *path_test, optimize=exp_path)
866: (8)                 assert_raises(
867: (12)                     RuntimeError, np.einsum_path, *path_test, optimize=exp_path)
868: (8)                 path_test = self.build_operands('a,a,a->a')
869: (8)                 exp_path = ['einsum_path', (1,), (0, 1)]
870: (8)                 assert_raises(RuntimeError, np.einsum, *path_test, optimize=exp_path)
871: (8)                 assert_raises(
872: (12)                     RuntimeError, np.einsum_path, *path_test, optimize=exp_path)
873: (4)             def test_spaces(self):
874: (8)                 arr = np.array([[1]])
875: (8)                 for sp in itertools.product(['', ' '], repeat=4):
876: (12)                     np.einsum('{}...a{}->{}...a{}'.format(*sp), arr)
877: (0)             def test_overlap():
878: (4)                 a = np.arange(9, dtype=int).reshape(3, 3)
879: (4)                 b = np.arange(9, dtype=int).reshape(3, 3)
880: (4)                 d = np.dot(a, b)
881: (4)                 c = np.einsum('ij,jk->ik', a, b)
882: (4)                 assert_equal(c, d)
883: (4)                 c = np.einsum('ij,jk->ik', a, b, out=b)
884: (4)                 assert_equal(c, d)
-----
```

## File 93 - test\_errstate.py:

```

1: (0)             import pytest
2: (0)             import sysconfig
3: (0)             import numpy as np
4: (0)             from numpy.testing import assert_, assert_raises, IS_WASM
5: (0)             hosttype = sysconfig.get_config_var('HOST_GNU_TYPE')
6: (0)             arm_softfloat = False if hosttype is None else hosttype.endswith('gnueabi')
7: (0)             class TestErrstate:
8: (4)                 @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
9: (4)                 @pytest.mark.skipif(arm_softfloat,
10: (24)                     reason='platform/cpu issue with FPU (gh-413,-15562)')
11: (4)             def test_invalid(self):
12: (8)                 with np.errstate(all='raise', under='ignore'):
13: (12)                     a = -np.arange(3)
14: (12)                     with np.errstate(invalid='ignore'):
15: (16)                         np.sqrt(a)
16: (12)                     with assert_raises(FloatingPointError):
17: (16)                         np.sqrt(a)
18: (4)                     @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
19: (4)                     @pytest.mark.skipif(arm_softfloat,
20: (24)                         reason='platform/cpu issue with FPU (gh-15562)')
21: (4)             def test_divide(self):
22: (8)                 with np.errstate(all='raise', under='ignore'):
23: (12)                     a = -np.arange(3)
24: (12)                     with np.errstate(divide='ignore'):
25: (16)                         a // 0
26: (12)                     with assert_raises(FloatingPointError):
27: (16)                         a // 0
28: (12)                     with assert_raises(FloatingPointError):
```

```

29: (16)           a // a
30: (4)           def test_errcall(self):
31: (8)           def foo(*args):
32: (12)             print(args)
33: (8)             olderrcall = np.geterrcall()
34: (8)             with np.errstate(call=foo):
35: (12)               assert_(np.geterrcall() is foo, 'call is not foo')
36: (12)               with np.errstate(call=None):
37: (16)                 assert_(np.geterrcall() is None, 'call is not None')
38: (8)                 assert_(np.geterrcall() is olderrcall, 'call is not olderrcall')
39: (4)             def test_errstate_decorator(self):
40: (8)               @np.errstate(all='ignore')
41: (8)               def foo():
42: (12)                 a = -np.arange(3)
43: (12)                 a // 0
44: (8)               foo()

```

---

## File 94 - test\_extint128.py:

```

1: (0)           import itertools
2: (0)           import contextlib
3: (0)           import operator
4: (0)           import pytest
5: (0)           import numpy as np
6: (0)           import numpy.core._multiarray_tests as mt
7: (0)           from numpy.testing import assert_raises, assert_equal
8: (0)           INT64_MAX = np.iinfo(np.int64).max
9: (0)           INT64_MIN = np.iinfo(np.int64).min
10: (0)          INT64_MID = 2**32
11: (0)          INT128_MAX = 2**128 - 1
12: (0)          INT128_MIN = -INT128_MAX
13: (0)          INT128_MID = 2**64
14: (0)          INT64_VALUES = (
15: (4)            [INT64_MIN + j for j in range(20)] +
16: (4)            [INT64_MAX - j for j in range(20)] +
17: (4)            [INT64_MID + j for j in range(-20, 20)] +
18: (4)            [2*INT64_MID + j for j in range(-20, 20)] +
19: (4)            [INT64_MID//2 + j for j in range(-20, 20)] +
20: (4)            list(range(-70, 70))
21: (0)
22: (0)          INT128_VALUES = (
23: (4)            [INT128_MIN + j for j in range(20)] +
24: (4)            [INT128_MAX - j for j in range(20)] +
25: (4)            [INT128_MID + j for j in range(-20, 20)] +
26: (4)            [2*INT128_MID + j for j in range(-20, 20)] +
27: (4)            [INT128_MID//2 + j for j in range(-20, 20)] +
28: (4)            list(range(-70, 70)) +
29: (4)            [False] # negative zero
30: (0)
31: (0)          INT64_POS_VALUES = [x for x in INT64_VALUES if x > 0]
32: (0)          @contextlib.contextmanager
33: (0)          def exc_iter(*args):
34: (4)            """
35: (4)              Iterate over Cartesian product of *args, and if an exception is raised,
36: (4)              add information of the current iterate.
37: (4)              """
38: (4)            value = [None]
39: (4)            def iterate():
40: (8)              for v in itertools.product(*args):
41: (12)                value[0] = v
42: (12)                yield v
43: (4)            try:
44: (8)              yield iterate()
45: (4)            except Exception:
46: (8)              import traceback
47: (8)              msg = "At: %r\n%s" % (repr(value[0]),
48: (30)                            traceback.format_exc())

```

```

49: (8)             raise AssertionError(msg)
50: (0)         def test_safe_binop():
51: (4)             ops = [
52: (8)                 (operator.add, 1),
53: (8)                 (operator.sub, 2),
54: (8)                 (operator.mul, 3)
55: (4)             ]
56: (4)             with exc_iter(ops, INT64_VALUES, INT64_VALUES) as it:
57: (8)                 for xop, a, b in it:
58: (12)                     pyop, op = xop
59: (12)                     c = pyop(a, b)
60: (12)                     if not (INT64_MIN <= c <= INT64_MAX):
61: (16)                         assert_raises(OverflowError, mt.extint_safe_binop, a, b, op)
62: (12)                     else:
63: (16)                         d = mt.extint_safe_binop(a, b, op)
64: (16)                         if c != d:
65: (20)                             assert_equal(d, c)
66: (0)         def test_to_128():
67: (4)             with exc_iter(INT64_VALUES) as it:
68: (8)                 for a, in it:
69: (12)                     b = mt.extint_to_128(a)
70: (12)                     if a != b:
71: (16)                         assert_equal(b, a)
72: (0)         def test_to_64():
73: (4)             with exc_iter(INT128_VALUES) as it:
74: (8)                 for a, in it:
75: (12)                     if not (INT64_MIN <= a <= INT64_MAX):
76: (16)                         assert_raises(OverflowError, mt.extint_to_64, a)
77: (12)                     else:
78: (16)                         b = mt.extint_to_64(a)
79: (16)                         if a != b:
80: (20)                             assert_equal(b, a)
81: (0)         def test_mul_64_64():
82: (4)             with exc_iter(INT64_VALUES, INT64_VALUES) as it:
83: (8)                 for a, b in it:
84: (12)                     c = a * b
85: (12)                     d = mt.extint_mul_64_64(a, b)
86: (12)                     if c != d:
87: (16)                         assert_equal(d, c)
88: (0)         def test_add_128():
89: (4)             with exc_iter(INT128_VALUES, INT128_VALUES) as it:
90: (8)                 for a, b in it:
91: (12)                     c = a + b
92: (12)                     if not (INT128_MIN <= c <= INT128_MAX):
93: (16)                         assert_raises(OverflowError, mt.extint_add_128, a, b)
94: (12)                     else:
95: (16)                         d = mt.extint_add_128(a, b)
96: (16)                         if c != d:
97: (20)                             assert_equal(d, c)
98: (0)         def test_sub_128():
99: (4)             with exc_iter(INT128_VALUES, INT128_VALUES) as it:
100: (8)                 for a, b in it:
101: (12)                     c = a - b
102: (12)                     if not (INT128_MIN <= c <= INT128_MAX):
103: (16)                         assert_raises(OverflowError, mt.extint_sub_128, a, b)
104: (12)                     else:
105: (16)                         d = mt.extint_sub_128(a, b)
106: (16)                         if c != d:
107: (20)                             assert_equal(d, c)
108: (0)         def test_neg_128():
109: (4)             with exc_iter(INT128_VALUES) as it:
110: (8)                 for a, in it:
111: (12)                     b = -a
112: (12)                     c = mt.extint_neg_128(a)
113: (12)                     if b != c:
114: (16)                         assert_equal(c, b)
115: (0)         def test_shl_128():
116: (4)             with exc_iter(INT128_VALUES) as it:
117: (8)                 for a, in it:

```

```

118: (12)             if a < 0:
119: (16)                 b = -(((a) << 1) & (2**128-1))
120: (12)             else:
121: (16)                 b = (a << 1) & (2**128-1)
122: (12)             c = mt.extint_shl_128(a)
123: (12)             if b != c:
124: (16)                 assert_equal(c, b)
125: (0)             def test_shr_128():
126: (4)                 with exc_iter(INT128_VALUES) as it:
127: (8)                     for a, in it:
128: (12)                         if a < 0:
129: (16)                             b = -((-a) >> 1)
130: (12)                         else:
131: (16)                             b = a >> 1
132: (12)                         c = mt.extint_shr_128(a)
133: (12)                         if b != c:
134: (16)                             assert_equal(c, b)
135: (0)             def test_gt_128():
136: (4)                 with exc_iter(INT128_VALUES, INT128_VALUES) as it:
137: (8)                     for a, b in it:
138: (12)                         c = a > b
139: (12)                         d = mt.extint_gt_128(a, b)
140: (12)                         if c != d:
141: (16)                             assert_equal(d, c)
142: (0)             @pytest.mark.slow
143: (0)             def test_divmod_128_64():
144: (4)                 with exc_iter(INT128_VALUES, INT64_POS_VALUES) as it:
145: (8)                     for a, b in it:
146: (12)                         if a >= 0:
147: (16)                             c, cr = divmod(a, b)
148: (12)                         else:
149: (16)                             c, cr = divmod(-a, b)
150: (16)                             c = -c
151: (16)                             cr = -cr
152: (12)                         d, dr = mt.extint_divmod_128_64(a, b)
153: (12)                         if c != d or d != dr or b*d + dr != a:
154: (16)                             assert_equal(d, c)
155: (16)                             assert_equal(dr, cr)
156: (16)                             assert_equal(b*d + dr, a)
157: (0)             def test_floordiv_128_64():
158: (4)                 with exc_iter(INT128_VALUES, INT64_POS_VALUES) as it:
159: (8)                     for a, b in it:
160: (12)                         c = a // b
161: (12)                         d = mt.extint_floordiv_128_64(a, b)
162: (12)                         if c != d:
163: (16)                             assert_equal(d, c)
164: (0)             def test_ceildiv_128_64():
165: (4)                 with exc_iter(INT128_VALUES, INT64_POS_VALUES) as it:
166: (8)                     for a, b in it:
167: (12)                         c = (a + b - 1) // b
168: (12)                         d = mt.extint_ceildiv_128_64(a, b)
169: (12)                         if c != d:
170: (16)                             assert_equal(d, c)

```

---

**File 95 - test\_function\_base.py:**

```

1: (0)             import pytest
2: (0)             from numpy import (
3: (4)                 logspace, linspace, geomspace, dtype, array, sctypes, arange, isnan,
4: (4)                 ndarray, sqrt, nextafter, stack, errstate
5: (4)             )
6: (0)             from numpy.testing import (
7: (4)                 assert_, assert_equal, assert_raises, assert_array_equal, assert_allclose,
8: (4)             )
9: (0)             class PhysicalQuantity(float):
10: (4)                 def __new__(cls, value):
11: (8)                     return float.__new__(cls, value)

```

```

12: (4)           def __add__(self, x):
13: (8)             assert_(isinstance(x, PhysicalQuantity))
14: (8)             return PhysicalQuantity(float(x) + float(self))
15: (4)             __radd__ = __add__
16: (4)           def __sub__(self, x):
17: (8)             assert_(isinstance(x, PhysicalQuantity))
18: (8)             return PhysicalQuantity(float(self) - float(x))
19: (4)           def __rsub__(self, x):
20: (8)             assert_(isinstance(x, PhysicalQuantity))
21: (8)             return PhysicalQuantity(float(x) - float(self))
22: (4)           def __mul__(self, x):
23: (8)             return PhysicalQuantity(float(x) * float(self))
24: (4)             __rmul__ = __mul__
25: (4)           def __div__(self, x):
26: (8)             return PhysicalQuantity(float(self) / float(x))
27: (4)           def __rdiv__(self, x):
28: (8)             return PhysicalQuantity(float(x) / float(self))
29: (0)         class PhysicalQuantity2(ndarray):
30: (4)             __array_priority__ = 10
31: (0)         class TestLogspace:
32: (4)             def test_basic(self):
33: (8)               y = logspace(0, 6)
34: (8)               assert_(len(y) == 50)
35: (8)               y = logspace(0, 6, num=100)
36: (8)               assert_(y[-1] == 10 ** 6)
37: (8)               y = logspace(0, 6, endpoint=False)
38: (8)               assert_(y[-1] < 10 ** 6)
39: (8)               y = logspace(0, 6, num=7)
40: (8)               assert_array_equal(y, [1, 10, 100, 1e3, 1e4, 1e5, 1e6])
41: (4)             def test_start_stop_array(self):
42: (8)               start = array([0., 1.])
43: (8)               stop = array([6., 7.])
44: (8)               t1 = logspace(start, stop, 6)
45: (8)               t2 = stack([logspace(_start, _stop, 6)
46: (20)                 for _start, _stop in zip(start, stop)], axis=1)
47: (8)               assert_equal(t1, t2)
48: (8)               t3 = logspace(start, stop[0], 6)
49: (8)               t4 = stack([logspace(_start, stop[0], 6)
50: (20)                 for _start in start], axis=1)
51: (8)               assert_equal(t3, t4)
52: (8)               t5 = logspace(start, stop, 6, axis=-1)
53: (8)               assert_equal(t5, t2.T)
54: (4)             @pytest.mark.parametrize("axis", [0, 1, -1])
55: (4)             def test_base_array(self, axis: int):
56: (8)               start = 1
57: (8)               stop = 2
58: (8)               num = 6
59: (8)               base = array([1, 2])
60: (8)               t1 = logspace(start, stop, num=num, base=base, axis=axis)
61: (8)               t2 = stack(
62: (12)                 [logspace(start, stop, num=num, base=_base) for _base in base],
63: (12)                 axis=(axis + 1) % t1.ndim,
64: (8)               )
65: (8)               assert_equal(t1, t2)
66: (4)             @pytest.mark.parametrize("axis", [0, 1, -1])
67: (4)             def test_stop_base_array(self, axis: int):
68: (8)               start = 1
69: (8)               stop = array([2, 3])
70: (8)               num = 6
71: (8)               base = array([1, 2])
72: (8)               t1 = logspace(start, stop, num=num, base=base, axis=axis)
73: (8)               t2 = stack(
74: (12)                 [logspace(start, _stop, num=num, base=_base)
75: (13)                   for _stop, _base in zip(stop, base)],
76: (12)                   axis=(axis + 1) % t1.ndim,
77: (8)                 )
78: (8)               assert_equal(t1, t2)
79: (4)             def test_dtype(self):
80: (8)               y = logspace(0, 6, dtype='float32')

```

```

81: (8) assert_equal(y.dtype, dtype('float32'))
82: (8) y = logspace(0, 6, dtype='float64')
83: (8) assert_equal(y.dtype, dtype('float64'))
84: (8) y = logspace(0, 6, dtype='int32')
85: (8) assert_equal(y.dtype, dtype('int32'))
86: (4) def test_physical_quantities(self):
87: (8)     a = PhysicalQuantity(1.0)
88: (8)     b = PhysicalQuantity(5.0)
89: (8)     assert_equal(logspace(a, b), logspace(1.0, 5.0))
90: (4) def test_subclass(self):
91: (8)     a = array(1).view(PhysicalQuantity2)
92: (8)     b = array(7).view(PhysicalQuantity2)
93: (8)     ls = logspace(a, b)
94: (8)     assert type(ls) is PhysicalQuantity2
95: (8)     assert_equal(ls, logspace(1.0, 7.0))
96: (8)     ls = logspace(a, b, 1)
97: (8)     assert type(ls) is PhysicalQuantity2
98: (8)     assert_equal(ls, logspace(1.0, 7.0, 1))
99: (0)
100: (4) class TestGeomspace:
101: (8)     def test_basic(self):
102: (8)         y = geomspace(1, 1e6)
103: (8)         assert_(len(y) == 50)
104: (8)         y = geomspace(1, 1e6, num=100)
105: (8)         assert_(y[-1] == 10 ** 6)
106: (8)         y = geomspace(1, 1e6, endpoint=False)
107: (8)         assert_(y[-1] < 10 ** 6)
108: (8)         y = geomspace(1, 1e6, num=7)
109: (8)         assert_array_equal(y, [1, 10, 100, 1e3, 1e4, 1e5, 1e6])
110: (8)         y = geomspace(8, 2, num=3)
111: (8)         assert_allclose(y, [8, 4, 2])
112: (8)         assert_array_equal(y.imag, 0)
113: (8)         y = geomspace(-1, -100, num=3)
114: (8)         assert_array_equal(y, [-1, -10, -100])
115: (8)         assert_array_equal(y.imag, 0)
116: (8)         y = geomspace(-100, -1, num=3)
117: (8)         assert_array_equal(y, [-100, -10, -1])
118: (4)     def test_boundaries_match_start_and_stop_exactly(self):
119: (8)         start = 0.3
120: (8)         stop = 20.3
121: (8)         y = geomspace(start, stop, num=1)
122: (8)         assert_equal(y[0], start)
123: (8)         y = geomspace(start, stop, num=1, endpoint=False)
124: (8)         assert_equal(y[0], start)
125: (8)         y = geomspace(start, stop, num=3)
126: (8)         assert_equal(y[0], start)
127: (8)         assert_equal(y[-1], stop)
128: (8)         y = geomspace(start, stop, num=3, endpoint=False)
129: (8)         assert_equal(y[0], start)
130: (4)     def test_nan_interior(self):
131: (8)         with errstate(invalid='ignore'):
132: (12)             y = geomspace(-3, 3, num=4)
133: (8)             assert_equal(y[0], -3.0)
134: (8)             assert_(isnan(y[1:-1]).all())
135: (8)             assert_equal(y[3], 3.0)
136: (8)             with errstate(invalid='ignore'):
137: (12)                 y = geomspace(-3, 3, num=4, endpoint=False)
138: (8)                 assert_equal(y[0], -3.0)
139: (8)                 assert_(isnan(y[1:]).all())
140: (4)     def test_complex(self):
141: (8)         y = geomspace(1j, 16j, num=5)
142: (8)         assert_allclose(y, [1j, 2j, 4j, 8j, 16j])
143: (8)         assert_array_equal(y.real, 0)
144: (8)         y = geomspace(-4j, -324j, num=5)
145: (8)         assert_allclose(y, [-4j, -12j, -36j, -108j, -324j])
146: (8)         assert_array_equal(y.real, 0)
147: (8)         y = geomspace(1+1j, 1000+1000j, num=4)
148: (8)         assert_allclose(y, [1+1j, 10+10j, 100+100j, 1000+1000j])
149: (8)         y = geomspace(-1+1j, -1000+1000j, num=4)

```

```

150: (8) assert_allclose(y, [-1+1j, -10+10j, -100+100j, -1000+1000j])
151: (8) y = geomspace(-1, 1, num=3, dtype=complex)
152: (8) assert_allclose(y, [-1, 1j, +1])
153: (8) y = geomspace(0+3j, -3+0j, 3)
154: (8) assert_allclose(y, [0+3j, -3/sqrt(2)+3j/sqrt(2), -3+0j])
155: (8) y = geomspace(0+3j, 3+0j, 3)
156: (8) assert_allclose(y, [0+3j, 3/sqrt(2)+3j/sqrt(2), 3+0j])
157: (8) y = geomspace(-3+0j, 0-3j, 3)
158: (8) assert_allclose(y, [-3+0j, -3/sqrt(2)-3j/sqrt(2), 0-3j])
159: (8) y = geomspace(0+3j, -3+0j, 3)
160: (8) assert_allclose(y, [0+3j, -3/sqrt(2)+3j/sqrt(2), -3+0j])
161: (8) y = geomspace(-2-3j, 5+7j, 7)
162: (8) assert_allclose(y, [-2-3j, -0.29058977-4.15771027j,
163: (28) 2.08885354-4.34146838j, 4.58345529-3.16355218j,
164: (28) 6.41401745-0.55233457j, 6.75707386+3.11795092j,
165: (28) 5+7j])
166: (8) y = geomspace(3j, -5, 2)
167: (8) assert_allclose(y, [3j, -5])
168: (8) y = geomspace(-5, 3j, 2)
169: (8) assert_allclose(y, [-5, 3j])
170: (4) def test_dtype(self):
171: (8) y = geomspace(1, 1e6, dtype='float32')
172: (8) assert_equal(y.dtype, dtype('float32'))
173: (8) y = geomspace(1, 1e6, dtype='float64')
174: (8) assert_equal(y.dtype, dtype('float64'))
175: (8) y = geomspace(1, 1e6, dtype='int32')
176: (8) assert_equal(y.dtype, dtype('int32'))
177: (8) y = geomspace(1, 1e6, dtype=float)
178: (8) assert_equal(y.dtype, dtype('float_'))
179: (8) y = geomspace(1, 1e6, dtype=complex)
180: (8) assert_equal(y.dtype, dtype('complex'))
181: (4) def test_start_stop_array_scalar(self):
182: (8) lim1 = array([120, 100], dtype="int8")
183: (8) lim2 = array([-120, -100], dtype="int8")
184: (8) lim3 = array([1200, 1000], dtype="uint16")
185: (8) t1 = geomspace(lim1[0], lim1[1], 5)
186: (8) t2 = geomspace(lim2[0], lim2[1], 5)
187: (8) t3 = geomspace(lim3[0], lim3[1], 5)
188: (8) t4 = geomspace(120.0, 100.0, 5)
189: (8) t5 = geomspace(-120.0, -100.0, 5)
190: (8) t6 = geomspace(1200.0, 1000.0, 5)
191: (8) assert_allclose(t1, t4, rtol=1e-2)
192: (8) assert_allclose(t2, t5, rtol=1e-2)
193: (8) assert_allclose(t3, t6, rtol=1e-5)
194: (4) def test_start_stop_array(self):
195: (8) start = array([1.e0, 32., 1j, -4j, 1+1j, -1])
196: (8) stop = array([1.e4, 2., 16j, -324j, 10000+10000j, 1])
197: (8) t1 = geomspace(start, stop, 5)
198: (8) t2 = stack([geomspace(_start, _stop, 5)
199: (20) for _start, _stop in zip(start, stop)], axis=1)
200: (8) assert_equal(t1, t2)
201: (8) t3 = geomspace(start, stop[0], 5)
202: (8) t4 = stack([geomspace(_start, stop[0], 5)
203: (20) for _start in start], axis=1)
204: (8) assert_equal(t3, t4)
205: (8) t5 = geomspace(start, stop, 5, axis=-1)
206: (8) assert_equal(t5, t2.T)
207: (4) def test_physical_quantities(self):
208: (8) a = PhysicalQuantity(1.0)
209: (8) b = PhysicalQuantity(5.0)
210: (8) assert_equal(geomspace(a, b), geomspace(1.0, 5.0))
211: (4) def test_subclass(self):
212: (8) a = array(1).view(PhysicalQuantity2)
213: (8) b = array(7).view(PhysicalQuantity2)
214: (8) gs = geomspace(a, b)
215: (8) assert_type(gs) is PhysicalQuantity2
216: (8) assert_equal(gs, geomspace(1.0, 7.0))
217: (8) gs = geomspace(a, b, 1)
218: (8) assert_type(gs) is PhysicalQuantity2

```

```

219: (8)             assert_equal(gs, geomspace(1.0, 7.0, 1))
220: (4)             def test_bounds(self):
221: (8)                 assert_raises(ValueError, geomspace, 0, 10)
222: (8)                 assert_raises(ValueError, geomspace, 10, 0)
223: (8)                 assert_raises(ValueError, geomspace, 0, 0)
224: (0)             class TestLinspace:
225: (4)                 def test_basic(self):
226: (8)                     y = linspace(0, 10)
227: (8)                     assert_(len(y) == 50)
228: (8)                     y = linspace(2, 10, num=100)
229: (8)                     assert_(y[-1] == 10)
230: (8)                     y = linspace(2, 10, endpoint=False)
231: (8)                     assert_(y[-1] < 10)
232: (8)                     assert_raises(ValueError, linspace, 0, 10, num=-1)
233: (4)                 def test_corner(self):
234: (8)                     y = list(linspace(0, 1, 1))
235: (8)                     assert_(y == [0.0], y)
236: (8)                     assert_raises(TypeError, linspace, 0, 1, num=2.5)
237: (4)                 def test_type(self):
238: (8)                     t1 = linspace(0, 1, 0).dtype
239: (8)                     t2 = linspace(0, 1, 1).dtype
240: (8)                     t3 = linspace(0, 1, 2).dtype
241: (8)                     assert_equal(t1, t2)
242: (8)                     assert_equal(t2, t3)
243: (4)                 def test_dtype(self):
244: (8)                     y = linspace(0, 6, dtype='float32')
245: (8)                     assert_equal(y.dtype, dtype('float32'))
246: (8)                     y = linspace(0, 6, dtype='float64')
247: (8)                     assert_equal(y.dtype, dtype('float64'))
248: (8)                     y = linspace(0, 6, dtype='int32')
249: (8)                     assert_equal(y.dtype, dtype('int32'))
250: (4)                 def test_start_stop_array_scalar(self):
251: (8)                     lim1 = array([-120, 100], dtype="int8")
252: (8)                     lim2 = array([120, -100], dtype="int8")
253: (8)                     lim3 = array([1200, 1000], dtype="uint16")
254: (8)                     t1 = linspace(lim1[0], lim1[1], 5)
255: (8)                     t2 = linspace(lim2[0], lim2[1], 5)
256: (8)                     t3 = linspace(lim3[0], lim3[1], 5)
257: (8)                     t4 = linspace(-120.0, 100.0, 5)
258: (8)                     t5 = linspace(120.0, -100.0, 5)
259: (8)                     t6 = linspace(1200.0, 1000.0, 5)
260: (8)                     assert_equal(t1, t4)
261: (8)                     assert_equal(t2, t5)
262: (8)                     assert_equal(t3, t6)
263: (4)                 def test_start_stop_array(self):
264: (8)                     start = array([-120, 120], dtype="int8")
265: (8)                     stop = array([100, -100], dtype="int8")
266: (8)                     t1 = linspace(start, stop, 5)
267: (8)                     t2 = stack([linspace(_start, _stop, 5)
268: (20)                         for _start, _stop in zip(start, stop)], axis=1)
269: (8)                     assert_equal(t1, t2)
270: (8)                     t3 = linspace(start, stop[0], 5)
271: (8)                     t4 = stack([linspace(_start, stop[0], 5)
272: (20)                         for _start in start], axis=1)
273: (8)                     assert_equal(t3, t4)
274: (8)                     t5 = linspace(start, stop, 5, axis=-1)
275: (8)                     assert_equal(t5, t2.T)
276: (4)                 def test_complex(self):
277: (8)                     lim1 = linspace(1 + 2j, 3 + 4j, 5)
278: (8)                     t1 = array([1.0+2.j, 1.5+2.5j, 2.0+3j, 2.5+3.5j, 3.0+4j])
279: (8)                     lim2 = linspace(1j, 10, 5)
280: (8)                     t2 = array([0.0+1.j, 2.5+0.75j, 5.0+0.5j, 7.5+0.25j, 10.0+0j])
281: (8)                     assert_equal(lim1, t1)
282: (8)                     assert_equal(lim2, t2)
283: (4)                 def test_physical_quantities(self):
284: (8)                     a = PhysicalQuantity(0.0)
285: (8)                     b = PhysicalQuantity(1.0)
286: (8)                     assert_equal(linspace(a, b), linspace(0.0, 1.0))
287: (4)                 def test_subclass(self):

```

```

288: (8)             a = array(0).view(PhysicalQuantity2)
289: (8)             b = array(1).view(PhysicalQuantity2)
290: (8)             ls = linspace(a, b)
291: (8)             assert type(ls) is PhysicalQuantity2
292: (8)             assert_equal(ls, linspace(0.0, 1.0))
293: (8)             ls = linspace(a, b, 1)
294: (8)             assert type(ls) is PhysicalQuantity2
295: (8)             assert_equal(ls, linspace(0.0, 1.0, 1))
296: (4)             def test_array_interface(self):
297: (8)                 class Arrayish:
298: (12)                     """
299: (12)                         A generic object that supports the __array_interface__ and hence
300: (12)                         can in principle be converted to a numeric scalar, but is not
301: (12)                         otherwise recognized as numeric, but also happens to support
302: (12)                         multiplication by floats.
303: (12)                         Data should be an object that implements the buffer interface,
304: (12)                         and contains at least 4 bytes.
305: (12)                     """
306: (12)                     def __init__(self, data):
307: (16)                         self._data = data
308: (12)                     @property
309: (12)                     def __array_interface__(self):
310: (16)                         return {'shape': (), 'typestr': '<i4', 'data': self._data,
311: (24)                             'version': 3}
312: (12)                     def __mul__(self, other):
313: (16)                         return self
314: (8)                     one = Arrayish(array(1, dtype='<i4'))
315: (8)                     five = Arrayish(array(5, dtype='<i4'))
316: (8)                     assert_equal(linspace(one, five), linspace(1, 5))
317: (4)                     def test_denormal_numbers(self):
318: (8)                         for ftype in sctypes['float']:
319: (12)                             stop = nextafter(ftype(0), ftype(1)) * 5 # A denormal number
320: (12)                             assert_(any(linspace(0, stop, 10, endpoint=False, dtype=ftype)))
321: (4)                     def test_equivalent_to_arange(self):
322: (8)                         for j in range(1000):
323: (12)                             assert_equal(linspace(0, j, j+1, dtype=int),
324: (25)                               arange(j+1, dtype=int))
325: (4)                     def test_retstep(self):
326: (8)                         for num in [0, 1, 2]:
327: (12)                             for ept in [False, True]:
328: (16)                                 y = linspace(0, 1, num, endpoint=ept, retstep=True)
329: (16)                                 assert isinstance(y, tuple) and len(y) == 2
330: (16)                                 if num == 2:
331: (20)                                     y0_expect = [0.0, 1.0] if ept else [0.0, 0.5]
332: (20)                                     assert_array_equal(y[0], y0_expect)
333: (20)                                     assert_equal(y[1], y0_expect[1])
334: (16)                                 elif num == 1 and not ept:
335: (20)                                     assert_array_equal(y[0], [0.0])
336: (20)                                     assert_equal(y[1], 1.0)
337: (16)                                 else:
338: (20)                                     assert_array_equal(y[0], [0.0][:num])
339: (20)                                     assert isnan(y[1])
340: (4)                     def test_object(self):
341: (8)                         start = array(1, dtype='O')
342: (8)                         stop = array(2, dtype='O')
343: (8)                         y = linspace(start, stop, 3)
344: (8)                         assert_array_equal(y, array([1., 1.5, 2.]))
345: (4)                     def test_round_negative(self):
346: (8)                         y = linspace(-1, 3, num=8, dtype=int)
347: (8)                         t = array([-1, -1, 0, 0, 1, 1, 2, 3], dtype=int)
348: (8)                         assert_array_equal(y, t)
349: (4)                     def test_any_step_zero_and_not_mult_inplace(self):
350: (8)                         start = array([0.0, 1.0])
351: (8)                         stop = array([2.0, 1.0])
352: (8)                         y = linspace(start, stop, 3)
353: (8)                         assert_array_equal(y, array([[0.0, 1.0], [1.0, 1.0], [2.0, 1.0]]))

```

## File 96 - test\_half.py:

```

1: (0)          import platform
2: (0)          import pytest
3: (0)          import numpy as np
4: (0)          from numpy import uint16, float16, float32, float64
5: (0)          from numpy.testing import assert_, assert_equal, _OLD_PROMOTION, IS_WASM
6: (0)          def assert_raises_fpe(strmatch, callable, *args, **kwargs):
7: (4)              try:
8: (8)                  callable(*args, **kwargs)
9: (4)              except FloatingPointError as exc:
10: (8)                  assert_(str(exc).find(strmatch) >= 0,
11: (16)                      "Did not raise floating point %s error" % strmatch)
12: (4)              else:
13: (8)                  assert_(False,
14: (16)                      "Did not raise floating point %s error" % strmatch)
15: (0)          class TestHalf:
16: (4)              def setup_method(self):
17: (8)                  self.all_f16 = np.arange(0x10000, dtype=uint16)
18: (8)                  self.all_f16.dtype = float16
19: (8)                  with np.errstate(invalid='ignore'):
20: (12)                      self.all_f32 = np.array(self.all_f16, dtype=float32)
21: (12)                      self.all_f64 = np.array(self.all_f16, dtype=float64)
22: (8)                  self.nonan_f16 = np.concatenate(
23: (32)                      (np.arange(0xfc00, 0xffff, -1, dtype=uint16),
24: (33)                          np.arange(0x0000, 0x7c01, 1, dtype=uint16)))
25: (8)                  self.nonan_f16.dtype = float16
26: (8)                  self.nonan_f32 = np.array(self.nonan_f16, dtype=float32)
27: (8)                  self.nonan_f64 = np.array(self.nonan_f16, dtype=float64)
28: (8)                  self.finite_f16 = self.nonan_f16[1:-1]
29: (8)                  self.finite_f32 = self.nonan_f32[1:-1]
30: (8)                  self.finite_f64 = self.nonan_f64[1:-1]
31: (4)              def test_half_conversions(self):
32: (8)                  """Checks that all 16-bit values survive conversion
33: (11)                      to/from 32-bit and 64-bit float"""
34: (8)                  with np.errstate(invalid='ignore'):
35: (12)                      b = np.array(self.all_f32, dtype=float16)
36: (8)                      b_nn = b == b
37: (8)                      assert_equal(self.all_f16[b_nn].view(dtype=uint16),
38: (21)                          b[b_nn].view(dtype=uint16))
39: (8)                      with np.errstate(invalid='ignore'):
40: (12)                          b = np.array(self.all_f64, dtype=float16)
41: (8)                          b_nn = b == b
42: (8)                          assert_equal(self.all_f16[b_nn].view(dtype=uint16),
43: (21)                              b[b_nn].view(dtype=uint16))
44: (8)                          a_ld = np.array(self.nonan_f16, dtype=np.longdouble)
45: (8)                          b = np.array(a_ld, dtype=float16)
46: (8)                          assert_equal(self.nonan_f16.view(dtype=uint16),
47: (21)                              b.view(dtype=uint16))
48: (8)                          i_int = np.arange(-2048, 2049)
49: (8)                          i_f16 = np.array(i_int, dtype=float16)
50: (8)                          j = np.array(i_f16, dtype=int)
51: (8)                          assert_equal(i_int, j)
52: (4)                      @pytest.mark.parametrize("string_dt", ["S", "U"])
53: (4)                      def test_half_conversion_to_string(self, string_dt):
54: (8)                          expected_dt = np.dtype(f"{string_dt}32")
55: (8)                          assert np.promote_types(np.float16, string_dt) == expected_dt
56: (8)                          assert np.promote_types(string_dt, np.float16) == expected_dt
57: (8)                          arr = np.ones(3, dtype=np.float16).astype(string_dt)
58: (8)                          assert arr.dtype == expected_dt
59: (4)                      @pytest.mark.parametrize("string_dt", ["S", "U"])
60: (4)                      def test_half_conversion_from_string(self, string_dt):
61: (8)                          string = np.array("3.1416", dtype=string_dt)
62: (8)                          assert string.astype(np.float16) == np.array(3.1416, dtype=np.float16)
63: (4)                      @pytest.mark.parametrize("offset", [None, "up", "down"])
64: (4)                      @pytest.mark.parametrize("shift", [None, "up", "down"])
65: (4)                      @pytest.mark.parametrize("float_t", [np.float32, np.float64])
66: (4)                      @np._no_nep50_warning()
67: (4)                      def test_half_conversion_rounding(self, float_t, shift, offset):

```

```

68: (8)                         max_pattern = np.float16(np.finfo(np.float16).max).view(np.uint16)
69: (8)                         f16s_patterns = np.arange(0, max_pattern+1, dtype=np.uint16)
70: (8)                         f16s_float = f16s_patterns.view(np.float16).astype(float_t)
71: (8)                         if shift == "up":
72: (12)                         f16s_float = 0.5 * (f16s_float[:-1] + f16s_float[1:])[1:]
73: (8)                         elif shift == "down":
74: (12)                         f16s_float = 0.5 * (f16s_float[:-1] + f16s_float[1:])[::1]
75: (8)                         else:
76: (12)                         f16s_float = f16s_float[1:-1]
77: (8)                         if offset == "up":
78: (12)                         f16s_float = np.nextafter(f16s_float, float_t(np.inf))
79: (8)                         elif offset == "down":
80: (12)                         f16s_float = np.nextafter(f16s_float, float_t(-np.inf))
81: (8)                         res_patterns = f16s_float.astype(np.float16).view(np.uint16)
82: (8)                         cmp_patterns = f16s_patterns[1:-1].copy()
83: (8)                         if shift == "down" and offset != "up":
84: (12)                         shift_pattern = -1
85: (8)                         elif shift == "up" and offset != "down":
86: (12)                         shift_pattern = 1
87: (8)                         else:
88: (12)                         shift_pattern = 0
89: (8)                         if offset is None:
90: (12)                         cmp_patterns[0::2].view(np.int16)[...] += shift_pattern
91: (8)                         else:
92: (12)                         cmp_patterns.view(np.int16)[...] += shift_pattern
93: (8)                         assert_equal(res_patterns, cmp_patterns)
94: (4) @pytest.mark.parametrize(["float_t", "uint_t", "bits"],
95: (29)                         [(np.float32, np.uint32, 23),
96: (30)                         (np.float64, np.uint64, 52)])
97: (4) def test_half_conversion_denormal_round_even(self, float_t, uint_t, bits):
98: (8)                         smallest_value = np.uint16(1).view(np.float16).astype(float_t)
99: (8)                         assert smallest_value == 2**-24
100: (8)                        rounded_to_zero = smallest_value / float_t(2)
101: (8)                        assert rounded_to_zero.astype(np.float16) == 0
102: (8)                        for i in range(bits):
103: (12)                         larger_pattern = rounded_to_zero.view(uint_t) | uint_t(1 << i)
104: (12)                         larger_value = larger_pattern.view(float_t)
105: (12)                         assert larger_value.astype(np.float16) == smallest_value
106: (4) def test_nans_infs(self):
107: (8)                         with np.errstate(all='ignore'):
108: (12)                             assert_equal(np.isnan(self.all_f16), np.isnan(self.all_f32))
109: (12)                             assert_equal(np.isinf(self.all_f16), np.isinf(self.all_f32))
110: (12)                             assert_equal(np.isfinite(self.all_f16), np.isfinite(self.all_f32))
111: (12)                             assert_equal(np.signbit(self.all_f16), np.signbit(self.all_f32))
112: (12)                             assert_equal(np.spacing(float16(65504)), np.inf)
113: (12)                             nan = float16(np.nan)
114: (12)                             assert_(not (self.all_f16 == nan).any())
115: (12)                             assert_(not (nan == self.all_f16).any())
116: (12)                             assert_((self.all_f16 != nan).all())
117: (12)                             assert_((nan != self.all_f16).all())
118: (12)                             assert_(not (self.all_f16 < nan).any())
119: (12)                             assert_(not (nan < self.all_f16).any())
120: (12)                             assert_(not (self.all_f16 <= nan).any())
121: (12)                             assert_(not (nan <= self.all_f16).any())
122: (12)                             assert_(not (self.all_f16 > nan).any())
123: (12)                             assert_(not (nan > self.all_f16).any())
124: (12)                             assert_(not (self.all_f16 >= nan).any())
125: (12)                             assert_(not (nan >= self.all_f16).any())
126: (4) def test_half_values(self):
127: (8)                         """Confirms a small number of known half values"""
128: (8)                         a = np.array([1.0, -1.0,
129: (22)                           2.0, -2.0,
130: (22)                           0.0999755859375, 0.333251953125, # 1/10, 1/3
131: (22)                           65504, -65504, # Maximum magnitude
132: (22)                           2.0**(-14), -2.0**(-14), # Minimum normal
133: (22)                           2.0**(-24), -2.0**(-24), # Minimum subnormal
134: (22)                           0, -1/1e1000, # Signed zeros
135: (22)                           np.inf, -np.inf])
136: (8)                         b = np.array([0x3c00, 0xbc00,

```

```

137: (22)          0x4000, 0xc000,
138: (22)          0x2e66, 0x3555,
139: (22)          0x7bff, 0xfbff,
140: (22)          0x0400, 0x8400,
141: (22)          0x0001, 0x8001,
142: (22)          0x0000, 0x8000,
143: (22)          0x7c00, 0xfc00], dtype=uint16)
144: (8)           b.dtype = float16
145: (8)           assert_equal(a, b)
146: (4)           def test_half_rounding(self):
147: (8)             """Checks that rounding when converting to half is correct"""
148: (8)             a = np.array([2.0**-25 + 2.0**-35, # Rounds to minimum subnormal
149: (22)               2.0**-25,      # Underflows to zero (nearest even mode)
150: (22)               2.0**-26,      # Underflows to zero
151: (22)               1.0+2.0**-11 + 2.0**-16, # rounds to 1.0+2**(-10)
152: (22)               1.0+2.0**-11,    # rounds to 1.0 (nearest even mode)
153: (22)               1.0+2.0**-12,    # rounds to 1.0
154: (22)               65519,        # rounds to 65504
155: (22)               65520],       # rounds to inf
156: (22)             dtype=float64)
157: (8)             rounded = [2.0**-24,
158: (19)               0.0,
159: (19)               0.0,
160: (19)               1.0+2.0**(-10),
161: (19)               1.0,
162: (19)               1.0,
163: (19)               65504,
164: (19)               np.inf]
165: (8)             with np.errstate(over="ignore"):
166: (12)               b = np.array(a, dtype=float16)
167: (8)             assert_equal(b, rounded)
168: (8)             a = np.array(a, dtype=float32)
169: (8)             with np.errstate(over="ignore"):
170: (12)               b = np.array(a, dtype=float16)
171: (8)             assert_equal(b, rounded)
172: (4)           def test_half_correctness(self):
173: (8)             """Take every finite float16, and check the casting functions with
174: (11)               a manual conversion."""
175: (8)             a_bits = self.finite_f16.view(dtype=uint16)
176: (8)             a_sgn = (-1.0)**((a_bits & 0x8000) >> 15)
177: (8)             a_exp = np.array((a_bits & 0x7c00) >> 10, dtype=np.int32) - 15
178: (8)             a_man = (a_bits & 0x03ff) * 2.0**(-10)
179: (8)             a_man[a_exp != -15] += 1
180: (8)             a_exp[a_exp == -15] = -14
181: (8)             a_manual = a_sgn * a_man * 2.0**a_exp
182: (8)             a32_fail = np.nonzero(self.finite_f32 != a_manual)[0]
183: (8)             if len(a32_fail) != 0:
184: (12)               bad_index = a32_fail[0]
185: (12)               assert_equal(self.finite_f32, a_manual,
186: (17)                 "First non-equal is half value 0x%08x -> %g != %g" %
187: (28)                   (a_bits[bad_index],
188: (29)                     self.finite_f32[bad_index],
189: (29)                     a_manual[bad_index]))
190: (8)             a64_fail = np.nonzero(self.finite_f64 != a_manual)[0]
191: (8)             if len(a64_fail) != 0:
192: (12)               bad_index = a64_fail[0]
193: (12)               assert_equal(self.finite_f64, a_manual,
194: (17)                 "First non-equal is half value 0x%016x -> %g != %g" %
195: (28)                   (a_bits[bad_index],
196: (29)                     self.finite_f64[bad_index],
197: (29)                     a_manual[bad_index]))
198: (4)           def test_half_ordering(self):
199: (8)             """Make sure comparisons are working right"""
200: (8)             a = self.nonan_f16[::-1].copy()
201: (8)             b = np.array(a, dtype=float32)
202: (8)             a.sort()
203: (8)             b.sort()
204: (8)             assert_equal(a, b)
205: (8)             assert_((a[:-1] <= a[1:]).all())

```

```

206: (8) assert_(not (a[:-1] > a[1:]).any())
207: (8) assert_((a[1:] >= a[:-1]).all())
208: (8) assert_(not (a[1:] < a[:-1]).any())
209: (8) assert_equal(np.nonzero(a[:-1] < a[1:])[0].size, a.size-2)
210: (8) assert_equal(np.nonzero(a[1:] > a[:-1])[0].size, a.size-2)
211: (4) def test_half_funcs(self):
212: (8)     """Test the various ArrFuncs"""
213: (8)     assert_equal(np.arange(10, dtype=float16),
214: (21)                 np.arange(10, dtype=float32))
215: (8)     a = np.zeros((5,), dtype=float16)
216: (8)     a.fill(1)
217: (8)     assert_equal(a, np.ones((5,), dtype=float16))
218: (8)     a = np.array([0, 0, -1, -1/1e20, 0, 2.0**-24, 7.629e-6],
219: (21)                 dtype=float16)
220: (8)     assert_equal(a.nonzero()[0],
221: (21)                 [2, 5, 6])
222: (8)     a = a.byteswap()
223: (8)     a = a.view(a.dtype.newbyteorder())
224: (21)     assert_equal(a.nonzero()[0],
225: (21)                 [2, 5, 6])
226: (8)     a = np.arange(0, 10, 0.5, dtype=float16)
227: (8)     b = np.ones((20,), dtype=float16)
228: (21)     assert_equal(np.dot(a, b),
229: (21)                 95)
230: (8)     a = np.array([0, -np.inf, -2, 0.5, 12.55, 7.3, 2.1, 12.4],
231: (21)                 dtype=float16)
232: (8)     assert_equal(a.argmax(),
233: (21)                 4)
234: (8)     a = np.array([0, -np.inf, -2, np.inf, 12.55, np.nan, 2.1, 12.4],
235: (21)                 dtype=float16)
236: (8)     assert_equal(a.argmax(),
237: (12)                 5)
238: (4) def test_spacing_nextafter(self):
239: (8)     """Test np.spacing and np.nextafter"""
240: (8)     a = np.arange(0x7c00, dtype=uint16)
241: (8)     hinf = np.array((np.inf,), dtype=float16)
242: (8)     hnan = np.array((np.nan,), dtype=float16)
243: (8)     a_f16 = a.view(dtype=float16)
244: (8)     assert_equal(np.spacing(a_f16[:-1]), a_f16[1:]-a_f16[:-1])
245: (8)     assert_equal(np.nextafter(a_f16[:-1], hinf), a_f16[1:])
246: (8)     assert_equal(np.nextafter(a_f16[0], -hinf), -a_f16[1])
247: (8)     assert_equal(np.nextafter(a_f16[1:], -hinf), a_f16[:-1])
248: (8)     assert_equal(np.nextafter(hinf, a_f16), a_f16[-1])
249: (8)     assert_equal(np.nextafter(-hinf, a_f16), -a_f16[-1])
250: (8)     assert_equal(np.nextafter(hinf, hinf), hinf)
251: (8)     assert_equal(np.nextafter(hinf, -hinf), a_f16[-1])
252: (8)     assert_equal(np.nextafter(-hinf, hinf), -a_f16[-1])
253: (8)     assert_equal(np.nextafter(-hinf, -hinf), -hinf)
254: (8)     assert_equal(np.nextafter(a_f16, hnan), hnan[0])
255: (8)     assert_equal(np.nextafter(hnan, a_f16), hnan[0])
256: (8)     assert_equal(np.nextafter(hnan, hnan), hnan)
257: (8)     assert_equal(np.nextafter(hinf, hnan), hnan)
258: (8)     assert_equal(np.nextafter(hnan, hinf), hnan)
259: (8)     a |= 0x8000
260: (8)     assert_equal(np.spacing(a_f16[0]), np.spacing(a_f16[1]))
261: (8)     assert_equal(np.spacing(a_f16[1:]), a_f16[:-1]-a_f16[1:])
262: (8)     assert_equal(np.nextafter(a_f16[0], hinf), -a_f16[1])
263: (8)     assert_equal(np.nextafter(a_f16[1:], hinf), a_f16[:-1])
264: (8)     assert_equal(np.nextafter(a_f16[:-1], -hinf), a_f16[1:])
265: (8)     assert_equal(np.nextafter(hinf, a_f16), -a_f16[-1])
266: (8)     assert_equal(np.nextafter(-hinf, a_f16), a_f16[-1])
267: (8)     assert_equal(np.nextafter(a_f16, hnan), hnan[0])
268: (8)     assert_equal(np.nextafter(hnan, a_f16), hnan[0])
269: (4) def test_half_ufuncs(self):
270: (8)     """Test the various ufuncs"""
271: (8)     a = np.array([0, 1, 2, 4, 2], dtype=float16)

```

```

272: (8)          b = np.array([-2, 5, 1, 4, 3], dtype=float16)
273: (8)          c = np.array([0, -1, -np.inf, np.nan, 6], dtype=float16)
274: (8)          assert_equal(np.add(a, b), [-2, 6, 3, 8, 5])
275: (8)          assert_equal(np.subtract(a, b), [2, -4, 1, 0, -1])
276: (8)          assert_equal(np.multiply(a, b), [0, 5, 2, 16, 6])
277: (8)          assert_equal(np.divide(a, b), [0, 0.199951171875, 2, 1,
0.66650390625])
278: (8)          assert_equal(np.equal(a, b), [False, False, False, True, False])
279: (8)          assert_equal(np.not_equal(a, b), [True, True, True, False, True])
280: (8)          assert_equal(np.less(a, b), [False, True, False, False, True])
281: (8)          assert_equal(np.less_equal(a, b), [False, True, False, True, True])
282: (8)          assert_equal(np.greater(a, b), [True, False, True, False, False])
283: (8)          assert_equal(np.greater_equal(a, b), [True, False, True, True, False])
284: (8)          assert_equal(np.logical_and(a, b), [False, True, True, True, True])
285: (8)          assert_equal(np.logical_or(a, b), [True, True, True, True, True])
286: (8)          assert_equal(np.logical_xor(a, b), [True, False, False, False, False])
287: (8)          assert_equal(np.logical_not(a), [True, False, False, False, False])
288: (8)          assert_equal(np.isnan(c), [False, False, False, True, False])
289: (8)          assert_equal(np.isinf(c), [False, False, True, False, False])
290: (8)          assert_equal(np.isfinite(c), [True, True, False, False, True])
291: (8)          assert_equal(np.signbit(b), [True, False, False, False, False])
292: (8)          assert_equal(np.copysign(b, a), [2, 5, 1, 4, 3])
293: (8)          assert_equal(np.maximum(a, b), [0, 5, 2, 4, 3])
294: (8)          x = np.maximum(b, c)
295: (8)          assert_(np.isnan(x[3]))
296: (8)          x[3] = 0
297: (8)          assert_equal(x, [0, 5, 1, 0, 6])
298: (8)          assert_equal(np.minimum(a, b), [-2, 1, 1, 4, 2])
299: (8)          x = np.minimum(b, c)
300: (8)          assert_(np.isnan(x[3]))
301: (8)          x[3] = 0
302: (8)          assert_equal(x, [-2, -1, -np.inf, 0, 3])
303: (8)          assert_equal(np.fmax(a, b), [0, 5, 2, 4, 3])
304: (8)          assert_equal(np.fmax(b, c), [0, 5, 1, 4, 6])
305: (8)          assert_equal(np.fmin(a, b), [-2, 1, 1, 4, 2])
306: (8)          assert_equal(np.fmin(b, c), [-2, -1, -np.inf, 4, 3])
307: (8)          assert_equal(np.floor_divide(a, b), [0, 0, 2, 1, 0])
308: (8)          assert_equal(np.remainder(a, b), [0, 1, 0, 0, 2])
309: (8)          assert_equal(np.divmod(a, b), ([0, 0, 2, 1, 0], [0, 1, 0, 0, 2]))
310: (8)          assert_equal(np.square(b), [4, 25, 1, 16, 9])
311: (8)          assert_equal(np.reciprocal(b), [-0.5, 0.199951171875, 1, 0.25,
0.333251953125])
312: (8)          assert_equal(np.ones_like(b), [1, 1, 1, 1, 1])
313: (8)          assert_equal(np.conjugate(b), b)
314: (8)          assert_equal(np.absolute(b), [2, 5, 1, 4, 3])
315: (8)          assert_equal(np.negative(b), [2, -5, -1, -4, -3])
316: (8)          assert_equal(np.positive(b), b)
317: (8)          assert_equal(np.sign(b), [-1, 1, 1, 1, 1])
318: (8)          assert_equal(np.modf(b), ([0, 0, 0, 0, 0], b))
319: (8)          assert_equal(np.frexp(b), ([-0.5, 0.625, 0.5, 0.5, 0.75], [2, 3, 1, 3,
2]))
320: (8)          assert_equal(np.ldexp(b, [0, 1, 2, 4, 2]), [-2, 10, 4, 64, 12])
321: (4)          @np._no_nep50_warning()
322: (4)          def test_half_coercion(self, weak_promotion):
323: (8)          """Test that half gets coerced properly with the other types"""
324: (8)          a16 = np.array((1,), dtype=float16)
325: (8)          a32 = np.array((1,), dtype=float32)
326: (8)          b16 = float16(1)
327: (8)          b32 = float32(1)
328: (8)          assert np.power(a16, 2).dtype == float16
329: (8)          assert np.power(a16, 2.0).dtype == float16
330: (8)          assert np.power(a16, b16).dtype == float16
331: (8)          expected_dt = float32 if weak_promotion else float16
332: (8)          assert np.power(a16, b32).dtype == expected_dt
333: (8)          assert np.power(a16, a16).dtype == float16
334: (8)          assert np.power(a16, a32).dtype == float32
335: (8)          expected_dt = float16 if weak_promotion else float64
336: (8)          assert np.power(b16, 2).dtype == expected_dt
337: (8)          assert np.power(b16, 2.0).dtype == expected_dt

```

```

338: (8) assert np.power(b16, b16).dtype, float16
339: (8) assert np.power(b16, b32).dtype, float32
340: (8) assert np.power(b16, a16).dtype, float16
341: (8) assert np.power(b16, a32).dtype, float32
342: (8) assert np.power(a32, a16).dtype == float32
343: (8) assert np.power(a32, b16).dtype == float32
344: (8) expected_dt = float32 if weak_promotion else float16
345: (8) assert np.power(b32, a16).dtype == expected_dt
346: (8) assert np.power(b32, b16).dtype == float32
347: (4) @pytest.mark.skipif(platform.machine() == "armv5tel",
348: (24) reason="See gh-413.")
349: (4) @pytest.mark.skipif(IS_WASM,
350: (24) reason="fp exceptions don't work in wasm.")
351: (4) def test_half_fpe(self):
352: (8)     with np.errstate(all='raise'):
353: (12)         sx16 = np.array((1e-4,), dtype=float16)
354: (12)         bx16 = np.array((1e4,), dtype=float16)
355: (12)         sy16 = float16(1e-4)
356: (12)         by16 = float16(1e4)
357: (12)         assert_raises_fpe('underflow', lambda a, b:a*b, sx16, sx16)
358: (12)         assert_raises_fpe('underflow', lambda a, b:a*b, sx16, sy16)
359: (12)         assert_raises_fpe('underflow', lambda a, b:a*b, sy16, sx16)
360: (12)         assert_raises_fpe('underflow', lambda a, b:a*b, sy16, sy16)
361: (12)         assert_raises_fpe('underflow', lambda a, b:a/b, sx16, bx16)
362: (12)         assert_raises_fpe('underflow', lambda a, b:a/b, sx16, by16)
363: (12)         assert_raises_fpe('underflow', lambda a, b:a/b, sy16, bx16)
364: (12)         assert_raises_fpe('underflow', lambda a, b:a/b, sy16, by16)
365: (12)         assert_raises_fpe('underflow', lambda a, b:a/b,
366: (45)                         float16(2.**-14), float16(2**11))
367: (12)         assert_raises_fpe('underflow', lambda a, b:a/b,
368: (45)                         float16(-2.**-14),
float16(2**11))
369: (12)         assert_raises_fpe('underflow', lambda a, b:a/b,
370: (45)                         float16(2.**-14+2**-24),
float16(2))
371: (12)         assert_raises_fpe('underflow', lambda a, b:a/b,
372: (45)                         float16(-2.**-14-2**-24),
float16(2))
373: (12)         assert_raises_fpe('underflow', lambda a, b:a/b,
374: (45)                         float16(2.**-14+2**-23),
float16(4))
375: (12)         assert_raises_fpe('overflow', lambda a, b:a*b, bx16, bx16)
376: (12)         assert_raises_fpe('overflow', lambda a, b:a*b, bx16, by16)
377: (12)         assert_raises_fpe('overflow', lambda a, b:a*b, by16, bx16)
378: (12)         assert_raises_fpe('overflow', lambda a, b:a*b, by16, by16)
379: (12)         assert_raises_fpe('overflow', lambda a, b:a/b, bx16, sx16)
380: (12)         assert_raises_fpe('overflow', lambda a, b:a/b, bx16, sy16)
381: (12)         assert_raises_fpe('overflow', lambda a, b:a/b, by16, sx16)
382: (12)         assert_raises_fpe('overflow', lambda a, b:a/b, by16, sy16)
383: (12)         assert_raises_fpe('overflow', lambda a, b:a+b,
384: (45)                         float16(65504), float16(17))
385: (12)         assert_raises_fpe('overflow', lambda a, b:a-b,
386: (45)                         float16(-65504), float16(17))
387: (12)         assert_raises_fpe('overflow', np.nextafter, float16(65504),
float16(np.inf))
388: (12)         assert_raises_fpe('overflow', np.nextafter, float16(-65504),
float16(-np.inf))
389: (12)         assert_raises_fpe('overflow', np.spacing, float16(65504))
390: (12)         assert_raises_fpe('invalid', np.divide, float16(np.inf),
float16(np.inf))
391: (12)         assert_raises_fpe('invalid', np.spacing, float16(np.inf))
392: (12)         assert_raises_fpe('invalid', np.spacing, float16(np.nan))
float16(65472)+float16(32)
float16(2**-13)/float16(2)
float16(2**-14)/float16(2**10)
np.spacing(float16(-65504))
np.nextafter(float16(65504), float16(-np.inf))
np.nextafter(float16(-65504), float16(np.inf))
np.nextafter(float16(np.inf), float16(0))

```

```

400: (12)           np.nextafter(float16(-np.inf), float16(0))
401: (12)           np.nextafter(float16(0), float16(np.nan))
402: (12)           np.nextafter(float16(np.nan), float16(0))
403: (12)           float16(2**-14)/float16(2**10)
404: (12)           float16(-2**-14)/float16(2**10)
405: (12)           float16(2**-14+2**-23)/float16(2)
406: (12)           float16(-2**-14-2**-23)/float16(2)
407: (4)            def test_half_array_interface(self):
408: (8)              """Test that half is compatible with __array_interface__"""
409: (8)              class Dummy:
410: (12)                  pass
411: (8)                  a = np.ones((1,), dtype=float16)
412: (8)                  b = Dummy()
413: (8)                  b.__array_interface__ = a.__array_interface__
414: (8)                  c = np.array(b)
415: (8)                  assert_(c.dtype == float16)
416: (8)                  assert_equal(a, c)

-----

```

## File 97 - test\_getlimits.py:

```

1: (0)          """ Test functions for limits module.
2: (0)          """
3: (0)          import warnings
4: (0)          import numpy as np
5: (0)          import pytest
6: (0)          from numpy.core import finfo, iinfo
7: (0)          from numpy import half, single, double, longdouble
8: (0)          from numpy.testing import assert_equal, assert_, assert_raises
9: (0)          from numpy.core.getlimits import _discovered_machar, _float_ma
10: (0)         class TestPythonFloat:
11: (4)             def test_singleton(self):
12: (8)                 ftype = finfo(float)
13: (8)                 ftype2 = finfo(float)
14: (8)                 assert_equal(id(ftype), id(ftype2))
15: (0)         class TestHalf:
16: (4)             def test_singleton(self):
17: (8)                 ftype = finfo(half)
18: (8)                 ftype2 = finfo(half)
19: (8)                 assert_equal(id(ftype), id(ftype2))
20: (0)         class TestSingle:
21: (4)             def test_singleton(self):
22: (8)                 ftype = finfo(single)
23: (8)                 ftype2 = finfo(single)
24: (8)                 assert_equal(id(ftype), id(ftype2))
25: (0)         class TestDouble:
26: (4)             def test_singleton(self):
27: (8)                 ftype = finfo(double)
28: (8)                 ftype2 = finfo(double)
29: (8)                 assert_equal(id(ftype), id(ftype2))
30: (0)         class TestLongdouble:
31: (4)             def test_singleton(self):
32: (8)                 ftype = finfo(longdouble)
33: (8)                 ftype2 = finfo(longdouble)
34: (8)                 assert_equal(id(ftype), id(ftype2))
35: (0)         def assert_finfo_equal(f1, f2):
36: (4)             for attr in ('bits', 'eps', 'epsneg', 'iexp', 'machep',
37: (17)                 'max', 'maxexp', 'min', 'minexp', 'negep', 'nexp',
38: (17)                 'nmant', 'precision', 'resolution', 'tiny',
39: (17)                 'smallest_normal', 'smallest_subnormal'):
40: (8)                 assert_equal(getattr(f1, attr), getattr(f2, attr),
41: (21)                     f'finfo instances {f1} and {f2} differ on {attr}')
42: (0)         def assert_iinfo_equal(i1, i2):
43: (4)             for attr in ('bits', 'min', 'max'):
44: (8)                 assert_equal(getattr(i1, attr), getattr(i2, attr),
45: (21)                     f'iinfo instances {i1} and {i2} differ on {attr}')
46: (0)         class TestFinfo:
47: (4)             def test_basic(self):

```

```

48: (8)
49: (23)
50: (24)
51: (8)
52: (12)
53: (8)
54: (4)
55: (8)
56: (8)
57: (8)
58: (4)
59: (8)
60: (12)
61: (12)
62: (8)
63: (8)
64: (0)
65: (4)
66: (8)
67: (19)
68: (18)
69: (19)
70: (8)
71: (12)
72: (8)
73: (4)
74: (8)
75: (8)
76: (12)
77: (16)
78: (12)
79: (0)
80: (4)
81: (8)
82: (8)
83: (4)
84: (8)
85: (19)
86: (8)
87: (0)
88: (4)
89: (8)
90: (8)
91: (8)
92: (4)
93: (8)
94: (8)
95: (8)
96: (4)
97: (8)
98: (4)
99: (8)
100: (4)
101: (8)
102: (0)
103: (4)
104: (8)
105: (8)
106: (12)
107: (12)
108: (0)
109: (4)
110: (27)
111: (27)
112: (8)
113: (4)
114: (8)
115: (4)
116: (4)

        dts = list(zip(['f2', 'f4', 'f8', 'c8', 'c16'],
                         [np.float16, np.float32, np.float64, np.complex64,
                          np.complex128]))
        for dt1, dt2 in dts:
            assert_finfo_equal(finfo(dt1), finfo(dt2))
            assert_raises(ValueError, finfo, 'i4')
    def test_regression_gh23108(self):
        f1 = np.finfo(np.float32(1.0))
        f2 = np.finfo(np.float64(1.0))
        assert f1 != f2
    def test_regression_gh23867(self):
        class NonHashableWithDtype:
            __hash__ = None
            dtype = np.dtype('float32')
            x = NonHashableWithDtype()
            assert np.finfo(x) == np.finfo(x.dtype)
    class TestIinfo:
        def test_basic(self):
            dts = list(zip(['i1', 'i2', 'i4', 'i8',
                            'u1', 'u2', 'u4', 'u8'],
                           [np.int8, np.int16, np.int32, np.int64,
                            np.uint8, np.uint16, np.uint32, np.uint64]))
            for dt1, dt2 in dts:
                assert_iinfo_equal(iinfo(dt1), iinfo(dt2))
                assert_raises(ValueError, iinfo, 'f4')
        def test_unsigned_max(self):
            types = np.sctypes['uint']
            for T in types:
                with np.errstate(over="ignore"):
                    max_calculated = T(0) - T(1)
                assert_equal(iinfo(T).max, max_calculated)
    class TestRepr:
        def test_iinfo_repr(self):
            expected = "iinfo(min=-32768, max=32767, dtype=int16)"
            assert_equal(repr(np.iinfo(np.int16)), expected)
        def test_finfo_repr(self):
            expected = "finfo(resolution=1e-06, min=-3.4028235e+38, " + \
                       " max=3.4028235e+38, dtype=float32)"
            assert_equal(repr(np.finfo(np.float32)), expected)
    def test_instances():
        for c in [int, np.int16, np.int32, np.int64]:
            class_iinfo = iinfo(c)
            instance_iinfo = iinfo(c(12))
            assert_iinfo_equal(class_iinfo, instance_iinfo)
        for c in [float, np.float16, np.float32, np.float64]:
            class_finfo = finfo(c)
            instance_finfo = finfo(c(1.2))
            assert_finfo_equal(class_finfo, instance_finfo)
        with pytest.raises(ValueError):
            iinfo(10.)
        with pytest.raises(ValueError):
            iinfo('hi')
        with pytest.raises(ValueError):
            finfo(np.int64(1))
    def assert_ma_equal(discovered, ma_like):
        for key, value in discovered.__dict__.items():
            assert_equal(value, getattr(ma_like, key))
            if hasattr(value, 'shape'):
                assert_equal(value.shape, getattr(ma_like, key).shape)
                assert_equal(value.dtype, getattr(ma_like, key).dtype)
    def test_known_types():
        for ftype, ma_like in ((np.float16, _float_ma[16]),
                               (np.float32, _float_ma[32]),
                               (np.float64, _float_ma[64])):
            assert_ma_equal(_discovered_machar(ftype), ma_like)
        with np.errstate(all='ignore'):
            ld_ma = _discovered_machar(np.longdouble)
            bytes = np.dtype(np.longdouble).itemsize
            if (ld_ma.it, ld_ma.maxexp) == (63, 16384) and bytes in (12, 16):

```

```

117: (8)             assert_ma_equal(ld_ma, _float_ma[80])
118: (4)             elif (ld_ma.it, ld_ma.maxexp) == (112, 16384) and bytes == 16:
119: (8)                 assert_ma_equal(ld_ma, _float_ma[128])
120: (0)             def test_subnormal_warning():
121: (4)                 """Test that the subnormal is zero warning is not being raised."""
122: (4)                 with np.errstate(all='ignore'):
123: (8)                     ld_ma = _discovered_machar(np.longdouble)
124: (4)                     bytes = np.dtype(np.longdouble).itemsize
125: (4)                     with warnings.catch_warnings(record=True) as w:
126: (8)                         warnings.simplefilter('always')
127: (8)                         if (ld_ma.it, ld_ma.maxexp) == (63, 16384) and bytes in (12, 16):
128: (12)                             ld_ma.smallest_subnormal
129: (12)                             assert len(w) == 0
130: (8)                         elif (ld_ma.it, ld_ma.maxexp) == (112, 16384) and bytes == 16:
131: (12)                             ld_ma.smallest_subnormal
132: (12)                             assert len(w) == 0
133: (8)                         else:
134: (12)                             ld_ma.smallest_subnormal
135: (12)                             assert len(w) == 0
136: (0)             def test_plausible_finfo():
137: (4)                 for ftype in np.sctypes['float'] + np.sctypes['complex']:
138: (8)                     info = np.finfo(ftype)
139: (8)                     assert_(info.nmant > 1)
140: (8)                     assert_(info.minexp < -1)
141: (8)                     assert_(info.maxexp > 1)

```

-----

## File 98 - test\_hashtable.py:

```

1: (0)             import pytest
2: (0)             import random
3: (0)             from numpy.core._multiarray_tests import identityhash_tester
4: (0)             @pytest.mark.parametrize("key_length", [1, 3, 6])
5: (0)             @pytest.mark.parametrize("length", [1, 16, 2000])
6: (0)             def test_identity_hashtable(key_length, length):
7: (4)                 pool = [object() for i in range(20)]
8: (4)                 keys_vals = []
9: (4)                 for i in range(length):
10: (8)                     keys = tuple(random.choices(pool, k=key_length))
11: (8)                     keys_vals.append((keys, random.choice(pool)))
12: (4)                     dictionary = dict(keys_vals)
13: (4)                     keys_vals.append(random.choice(keys_vals))
14: (4)                     expected = dictionary[keys_vals[-1][0]]
15: (4)                     res = identityhash_tester(key_length, keys_vals, replace=True)
16: (4)                     assert res is expected
17: (4)                     keys_vals.insert(0, keys_vals[-2])
18: (4)                     with pytest.raises(RuntimeError):
19: (8)                         identityhash_tester(key_length, keys_vals)

```

-----

## File 99 - test\_indexerrors.py:

```

1: (0)             import numpy as np
2: (0)             from numpy.testing import (
3: (8)                 assert_raises, assert_raises_regex,
4: (8)                 )
5: (0)             class TestIndexErrors:
6: (4)                 '''Tests to exercise indexerrors not covered by other tests.'''
7: (4)                 def test_arraytypes_fasttake(self):
8: (8)                     'take from a 0-length dimension'
9: (8)                     x = np.empty((2, 3, 0, 4))
10: (8)                     assert_raises(IndexError, x.take, [0], axis=2)
11: (8)                     assert_raises(IndexError, x.take, [1], axis=2)
12: (8)                     assert_raises(IndexError, x.take, [0], axis=2, mode='wrap')
13: (8)                     assert_raises(IndexError, x.take, [0], axis=2, mode='clip')
14: (4)                 def test_take_from_object(self):
15: (8)                     d = np.zeros(5, dtype=object)

```

```

16: (8)             assert_raises(IndexError, d.take, [6])
17: (8)             d = np.zeros((5, 0), dtype=object)
18: (8)             assert_raises(IndexError, d.take, [1], axis=1)
19: (8)             assert_raises(IndexError, d.take, [0], axis=1)
20: (8)             assert_raises(IndexError, d.take, [0])
21: (8)             assert_raises(IndexError, d.take, [0], mode='wrap')
22: (8)             assert_raises(IndexError, d.take, [0], mode='clip')
23: (4)         def test_multiindex_exceptions(self):
24: (8)             a = np.empty(5, dtype=object)
25: (8)             assert_raises(IndexError, a.item, 20)
26: (8)             a = np.empty((5, 0), dtype=object)
27: (8)             assert_raises(IndexError, a.item, (0, 0))
28: (8)             a = np.empty(5, dtype=object)
29: (8)             assert_raises(IndexError, a.itemset, 20, 0)
30: (8)             a = np.empty((5, 0), dtype=object)
31: (8)             assert_raises(IndexError, a.itemset, (0, 0), 0)
32: (4)         def test_put_exceptions(self):
33: (8)             a = np.zeros((5, 5))
34: (8)             assert_raises(IndexError, a.put, 100, 0)
35: (8)             a = np.zeros((5, 5), dtype=object)
36: (8)             assert_raises(IndexError, a.put, 100, 0)
37: (8)             a = np.zeros((5, 5, 0))
38: (8)             assert_raises(IndexError, a.put, 100, 0)
39: (8)             a = np.zeros((5, 5, 0), dtype=object)
40: (8)             assert_raises(IndexError, a.put, 100, 0)
41: (4)         def test_iterators_exceptions(self):
42: (8)             "cases in iterators.c"
43: (8)             def assign(obj, ind, val):
44: (12)                 obj[ind] = val
45: (8)                 a = np.zeros([1, 2, 3])
46: (8)                 assert_raises(IndexError, lambda: a[0, 5, None, 2])
47: (8)                 assert_raises(IndexError, lambda: a[0, 5, 0, 2])
48: (8)                 assert_raises(IndexError, lambda: assign(a, (0, 5, None, 2), 1))
49: (8)                 assert_raises(IndexError, lambda: assign(a, (0, 5, 0, 2), 1))
50: (8)                 a = np.zeros([1, 0, 3])
51: (8)                 assert_raises(IndexError, lambda: a[0, 0, None, 2])
52: (8)                 assert_raises(IndexError, lambda: assign(a, (0, 0, None, 2), 1))
53: (8)                 a = np.zeros([1, 2, 3])
54: (8)                 assert_raises(IndexError, lambda: a.flat[10])
55: (8)                 assert_raises(IndexError, lambda: assign(a.flat, 10, 5))
56: (8)                 a = np.zeros([1, 0, 3])
57: (8)                 assert_raises(IndexError, lambda: a.flat[10])
58: (8)                 assert_raises(IndexError, lambda: assign(a.flat, 10, 5))
59: (8)                 a = np.zeros([1, 2, 3])
60: (8)                 assert_raises(IndexError, lambda: a.flat[np.array(10)])
61: (8)                 assert_raises(IndexError, lambda: assign(a.flat, np.array(10), 5))
62: (8)                 a = np.zeros([1, 0, 3])
63: (8)                 assert_raises(IndexError, lambda: a.flat[np.array(10)])
64: (8)                 assert_raises(IndexError, lambda: assign(a.flat, np.array(10), 5))
65: (8)                 a = np.zeros([1, 2, 3])
66: (8)                 assert_raises(IndexError, lambda: a.flat[np.array([10])])
67: (8)                 assert_raises(IndexError, lambda: assign(a.flat, np.array([10]), 5))
68: (8)                 a = np.zeros([1, 0, 3])
69: (8)                 assert_raises(IndexError, lambda: a.flat[np.array([10])])
70: (8)                 assert_raises(IndexError, lambda: assign(a.flat, np.array([10]), 5))
71: (4)         def test_mapping(self):
72: (8)             "cases from mapping.c"
73: (8)             def assign(obj, ind, val):
74: (12)                 obj[ind] = val
75: (8)                 a = np.zeros((0, 10))
76: (8)                 assert_raises(IndexError, lambda: a[12])
77: (8)                 a = np.zeros((3, 5))
78: (8)                 assert_raises(IndexError, lambda: a[(10, 20)])
79: (8)                 assert_raises(IndexError, lambda: assign(a, (10, 20), 1))
80: (8)                 a = np.zeros((3, 0))
81: (8)                 assert_raises(IndexError, lambda: a[(1, 0)])
82: (8)                 assert_raises(IndexError, lambda: assign(a, (1, 0), 1))
83: (8)                 a = np.zeros((10,))
84: (8)                 assert_raises(IndexError, lambda: assign(a, 10, 1))

```

```

85: (8)             a = np.zeros((0,))
86: (8)             assert_raises(IndexError, lambda: assign(a, 10, 1))
87: (8)             a = np.zeros((3, 5))
88: (8)             assert_raises(IndexError, lambda: a[(1, [1, 20])])
89: (8)             assert_raises(IndexError, lambda: assign(a, (1, [1, 20]), 1))
90: (8)             a = np.zeros((3, 0))
91: (8)             assert_raises(IndexError, lambda: a[(1, [0, 1])])
92: (8)             assert_raises(IndexError, lambda: assign(a, (1, [0, 1]), 1))
93: (4)             def test_mapping_error_message(self):
94: (8)                 a = np.zeros((3, 5))
95: (8)                 index = (1, 2, 3, 4, 5)
96: (8)                 assert_raises_regex(
97: (16)                     IndexError,
98: (16)                     "too many indices for array: "
99: (16)                     "array is 2-dimensional, but 5 were indexed",
100: (16)                     lambda: a[index])
101: (4)             def test_methods(self):
102: (8)                 "cases from methods.c"
103: (8)                 a = np.zeros((3, 3))
104: (8)                 assert_raises(IndexError, lambda: a.item(100))
105: (8)                 assert_raises(IndexError, lambda: a.itemset(100, 1))
106: (8)                 a = np.zeros((0, 3))
107: (8)                 assert_raises(IndexError, lambda: a.item(100))
108: (8)                 assert_raises(IndexError, lambda: a.itemset(100, 1))

-----

```

## File 100 - test\_indexing.py:

```

1: (0)             import sys
2: (0)             import warnings
3: (0)             import functools
4: (0)             import operator
5: (0)             import pytest
6: (0)             import numpy as np
7: (0)             from numpy.core._multiarray_tests import array_indexing
8: (0)             from itertools import product
9: (0)             from numpy.testing import (
10: (4)                 assert_, assert_equal, assert_raises, assert_raises_regex,
11: (4)                 assert_array_equal, assert_warns, HAS_REFCOUNT, IS_WASM
12: (4)             )
13: (0)             class TestIndexing:
14: (4)                 def test_index_no_floats(self):
15: (8)                     a = np.array([[[5]]])
16: (8)                     assert_raises(IndexError, lambda: a[0.0])
17: (8)                     assert_raises(IndexError, lambda: a[0, 0.0])
18: (8)                     assert_raises(IndexError, lambda: a[0.0, 0])
19: (8)                     assert_raises(IndexError, lambda: a[0.0,:])
20: (8)                     assert_raises(IndexError, lambda: a[:, 0.0])
21: (8)                     assert_raises(IndexError, lambda: a[:, 0.0,:])
22: (8)                     assert_raises(IndexError, lambda: a[0.0,:,:])
23: (8)                     assert_raises(IndexError, lambda: a[0, 0, 0.0])
24: (8)                     assert_raises(IndexError, lambda: a[0.0, 0, 0])
25: (8)                     assert_raises(IndexError, lambda: a[0, 0.0, 0])
26: (8)                     assert_raises(IndexError, lambda: a[-1.4])
27: (8)                     assert_raises(IndexError, lambda: a[0, -1.4])
28: (8)                     assert_raises(IndexError, lambda: a[-1.4, 0])
29: (8)                     assert_raises(IndexError, lambda: a[-1.4,:])
30: (8)                     assert_raises(IndexError, lambda: a[:, -1.4])
31: (8)                     assert_raises(IndexError, lambda: a[:, -1.4,:])
32: (8)                     assert_raises(IndexError, lambda: a[-1.4,:,:])
33: (8)                     assert_raises(IndexError, lambda: a[0, 0, -1.4])
34: (8)                     assert_raises(IndexError, lambda: a[-1.4, 0, 0])
35: (8)                     assert_raises(IndexError, lambda: a[0, -1.4, 0])
36: (8)                     assert_raises(IndexError, lambda: a[0.0:, 0.0])
37: (8)                     assert_raises(IndexError, lambda: a[0.0:, 0.0,:])
38: (4)             def test_slicing_no_floats(self):
39: (8)                 a = np.array([[5]])
40: (8)                 assert_raises(TypeError, lambda: a[0.0:])


```

```

41: (8) assert_raises(TypeError, lambda: a[0:, 0.0:2])
42: (8) assert_raises(TypeError, lambda: a[0.0::2, :0])
43: (8) assert_raises(TypeError, lambda: a[0.0:1:2,:])
44: (8) assert_raises(TypeError, lambda: a[:, 0.0:])
45: (8) assert_raises(TypeError, lambda: a[:0.0])
46: (8) assert_raises(TypeError, lambda: a[:0, 1:2.0])
47: (8) assert_raises(TypeError, lambda: a[:0.0:2, :0])
48: (8) assert_raises(TypeError, lambda: a[:0.0,:])
49: (8) assert_raises(TypeError, lambda: a[:, 0:4.0:2])
50: (8) assert_raises(TypeError, lambda: a[:::1.0])
51: (8) assert_raises(TypeError, lambda: a[0:, :2:2.0])
52: (8) assert_raises(TypeError, lambda: a[1::4.0, :0])
53: (8) assert_raises(TypeError, lambda: a[:::5.0,:])
54: (8) assert_raises(TypeError, lambda: a[:, 0:4:2.0])
55: (8) assert_raises(TypeError, lambda: a[1.0:2:2.0])
56: (8) assert_raises(TypeError, lambda: a[1.0::2.0])
57: (8) assert_raises(TypeError, lambda: a[0:, :2.0:2.0])
58: (8) assert_raises(TypeError, lambda: a[1.0:1:4.0, :0])
59: (8) assert_raises(TypeError, lambda: a[1.0:5.0:5.0,:])
60: (8) assert_raises(TypeError, lambda: a[:, 0.4:4.0:2.0])
61: (8) assert_raises(TypeError, lambda: a[:::0.0])
62: (4) def test_index_no_array_to_index(self):
63: (8) a = np.array([[[1]]])
64: (8) assert_raises(TypeError, lambda: a[a:a:a])
65: (4) def test_none_index(self):
66: (8) a = np.array([1, 2, 3])
67: (8) assert_equal(a[None], a[np.newaxis])
68: (8) assert_equal(a[None].ndim, a.ndim + 1)
69: (4) def test_empty_tuple_index(self):
70: (8) a = np.array([1, 2, 3])
71: (8) assert_equal(a[()], a)
72: (8) assert_(a[()].base is a)
73: (8) a = np.array(0)
74: (8) assert_(isinstance(a[()], np.int_))
75: (4) def test_void_scalar_empty_tuple(self):
76: (8) s = np.zeros(), dtype='V4')
77: (8) assert_equal(s[()].dtype, s.dtype)
78: (8) assert_equal(s[()], s)
79: (8) assert_equal(type(s[...]), np.ndarray)
80: (4) def test_same_kind_index_casting(self):
81: (8) index = np.arange(5)
82: (8) u_index = index.astype(np.uintp)
83: (8) arr = np.arange(10)
84: (8) assert_array_equal(arr[index], arr[u_index])
85: (8) arr[u_index] = np.arange(5)
86: (8) assert_array_equal(arr, np.arange(10))
87: (8) arr = np.arange(10).reshape(5, 2)
88: (8) assert_array_equal(arr[index], arr[u_index])
89: (8) arr[u_index] = np.arange(5)[ :,None]
90: (8) assert_array_equal(arr, np.arange(5)[ :,None].repeat(2, axis=1))
91: (8) arr = np.arange(25).reshape(5, 5)
92: (8) assert_array_equal(arr[u_index, u_index], arr[index, index])
93: (4) def test_empty_fancy_index(self):
94: (8) a = np.array([1, 2, 3])
95: (8) assert_equal(a[[], []])
96: (8) assert_equal(a[[]].dtype, a.dtype)
97: (8) b = np.array([], dtype=np.intp)
98: (8) assert_equal(a[[], []])
99: (8) assert_equal(a[[]].dtype, a.dtype)
100: (8) b = np.array([])
101: (8) assert_raises(IndexError, a.__getitem__, b)
102: (4) def test_ellipsis_index(self):
103: (8) a = np.array([[1, 2, 3],
104: (22) [4, 5, 6],
105: (22) [7, 8, 9]])
106: (8) assert_(a[...] is not a)
107: (8) assert_equal(a[...], a)
108: (8) assert_(a[...].base is a)
109: (8) assert_equal(a[0, ...], a[0])

```

```

110: (8) assert_equal(a[0, ...], a[0,:])
111: (8) assert_equal(a[..., 0], a[:, 0])
112: (8) assert_equal(a[0, ..., 1], np.array(2))
113: (8) b = np.array(1)
114: (8) b[(Ellipsis,)] = 2
115: (8) assert_equal(b, 2)
116: (4) def test_single_int_index(self):
117: (8)     a = np.array([[1, 2, 3],
118: (22)             [4, 5, 6],
119: (22)             [7, 8, 9]])
120: (8)     assert_equal(a[0], [1, 2, 3])
121: (8)     assert_equal(a[-1], [7, 8, 9])
122: (8)     assert_raises(IndexError, a.__getitem__, 1 << 30)
123: (8)     assert_raises(IndexError, a.__getitem__, 1 << 64)
124: (4) def test_single_bool_index(self):
125: (8)     a = np.array([[1, 2, 3],
126: (22)             [4, 5, 6],
127: (22)             [7, 8, 9]])
128: (8)     assert_equal(a[np.array(True)], a[None])
129: (8)     assert_equal(a[np.array(False)], a[None][0:0])
130: (4) def test_boolean_shape_mismatch(self):
131: (8)     arr = np.ones((5, 4, 3))
132: (8)     index = np.array([True])
133: (8)     assert_raises(IndexError, arr.__getitem__, index)
134: (8)     index = np.array([False] * 6)
135: (8)     assert_raises(IndexError, arr.__getitem__, index)
136: (8)     index = np.zeros((4, 4), dtype=bool)
137: (8)     assert_raises(IndexError, arr.__getitem__, index)
138: (8)     assert_raises(IndexError, arr.__getitem__, (slice(None), index))
139: (4) def test_boolean_indexing_onedim(self):
140: (8)     a = np.array([[ 0.,  0.,  0.]])
141: (8)     b = np.array([ True], dtype=bool)
142: (8)     assert_equal(a[b], a)
143: (8)     a[b] = 1.
144: (8)     assert_equal(a, [[1., 1., 1.]])
145: (4) def test_boolean_assignment_value_mismatch(self):
146: (8)     a = np.arange(4)
147: (8)     def f(a, v):
148: (12)         a[a > -1] = v
149: (8)         assert_raises(ValueError, f, a, [])
150: (8)         assert_raises(ValueError, f, a, [1, 2, 3])
151: (8)         assert_raises(ValueError, f, a[:1], [1, 2, 3])
152: (4) def test_boolean_assignment_needs_api(self):
153: (8)     arr = np.zeros(1000)
154: (8)     indx = np.zeros(1000, dtype=bool)
155: (8)     indx[:100] = True
156: (8)     arr[indx] = np.ones(100, dtype=object)
157: (8)     expected = np.zeros(1000)
158: (8)     expected[:100] = 1
159: (8)     assert_array_equal(arr, expected)
160: (4) def test_boolean_indexing_twodim(self):
161: (8)     a = np.array([[1, 2, 3],
162: (22)             [4, 5, 6],
163: (22)             [7, 8, 9]])
164: (8)     b = np.array([[ True, False,  True],
165: (22)                 [False,  True, False],
166: (22)                 [ True, False,  True]])
167: (8)     assert_equal(a[b], [1, 3, 5, 7, 9])
168: (8)     assert_equal(a[b[1]], [[4, 5, 6]])
169: (8)     assert_equal(a[b[0]], a[b[2]])
170: (8)     a[b] = 0
171: (8)     assert_equal(a, [[0, 2, 0],
172: (25)             [4, 0, 6],
173: (25)             [0, 8, 0]])
174: (4) def test_boolean_indexing_list(self):
175: (8)     a = np.array([1, 2, 3])
176: (8)     b = [True, False, True]
177: (8)     assert_equal(a[b], [1, 3])
178: (8)     assert_equal(a[None, b], [[1, 3]])

```

```

179: (4)           def test_reverse_strides_and_subspace_bufferinit(self):
180: (8)             a = np.ones(5)
181: (8)             b = np.zeros(5, dtype=np.intp)[::-1]
182: (8)             c = np.arange(5)[::-1]
183: (8)             a[b] = c
184: (8)             assert_equal(a[0], 0)
185: (8)             a = np.ones((5, 2))
186: (8)             c = np.arange(10).reshape(5, 2)[::-1]
187: (8)             a[b, :] = c
188: (8)             assert_equal(a[0], [0, 1])
189: (4)           def test_reversed_strides_result_allocation(self):
190: (8)             a = np.arange(10)[:, None]
191: (8)             i = np.arange(10)[::-1]
192: (8)             assert_array_equal(a[i], a[i.copy('C')])
193: (8)             a = np.arange(20).reshape(-1, 2)
194: (4)           def test_uncontiguous_subspace_assignment(self):
195: (8)             a = np.full((3, 4, 2), -1)
196: (8)             b = np.full((3, 4, 2), -1)
197: (8)             a[[0, 1]] = np.arange(2 * 4 * 2).reshape(2, 4, 2).T
198: (8)             b[[0, 1]] = np.arange(2 * 4 * 2).reshape(2, 4, 2).T.copy()
199: (8)             assert_equal(a, b)
200: (4)           def test_too_many_fancy_indices_special_case(self):
201: (8)             a = np.ones(1,) * 32 # 32 is NPY_MAXDIMS
202: (8)             assert_raises(IndexError, a.__getitem__, (np.array([0]),) * 32)
203: (4)           def test_scalar_array_bool(self):
204: (8)             a = np.array(1)
205: (8)             assert_equal(a[np.bool_(True)], a[np.array(True)])
206: (8)             assert_equal(a[np.bool_(False)], a[np.array(False)])
207: (4)           def test_everything_returns_views(self):
208: (8)             a = np.arange(5)
209: (8)             assert_(a is not a[()])
210: (8)             assert_(a is not a[...])
211: (8)             assert_(a is not a[:])
212: (4)           def test_broaderrors_indexing(self):
213: (8)             a = np.zeros((5, 5))
214: (8)             assert_raises(IndexError, a.__getitem__, ([0, 1], [0, 1, 2]))
215: (8)             assert_raises(IndexError, a.__setitem__, ([0, 1], [0, 1, 2]), 0)
216: (4)           def test_trivial_fancy_out_of_bounds(self):
217: (8)             a = np.zeros(5)
218: (8)             ind = np.ones(20, dtype=np.intp)
219: (8)             ind[-1] = 10
220: (8)             assert_raises(IndexError, a.__getitem__, ind)
221: (8)             assert_raises(IndexError, a.__setitem__, ind, 0)
222: (8)             ind = np.ones(20, dtype=np.intp)
223: (8)             ind[0] = 11
224: (8)             assert_raises(IndexError, a.__getitem__, ind)
225: (8)             assert_raises(IndexError, a.__setitem__, ind, 0)
226: (4)           def test_trivial_fancy_not_possible(self):
227: (8)             a = np.arange(6)
228: (8)             idx = np.arange(6, dtype=np.intp).reshape(2, 1, 3)[:, :, 0]
229: (8)             assert_array_equal(a[idx], idx)
230: (8)             a[idx] = -1
231: (8)             res = np.arange(6)
232: (8)             res[0] = -1
233: (8)             res[3] = -1
234: (8)             assert_array_equal(a, res)
235: (4)           def test_nonbaseclass_values(self):
236: (8)             class SubClass(np.ndarray):
237: (12)               def __array_finalize__(self, old):
238: (16)                 self.fill(99)
239: (8)             a = np.zeros((5, 5))
240: (8)             s = a.copy().view(type=SubClass)
241: (8)             s.fill(1)
242: (8)             a[[0, 1, 2, 3, 4], :] = s
243: (8)             assert_((a == 1).all())
244: (8)             a[:, [0, 1, 2, 3, 4]] = s
245: (8)             assert_((a == 1).all())
246: (8)             a.fill(0)
247: (8)             a[...] = s

```

```

248: (8)             assert_(a == 1).all())
249: (4)             def test_array_like_values(self):
250: (8)                 a = np.zeros((5, 5))
251: (8)                 s = np.arange(25, dtype=np.float64).reshape(5, 5)
252: (8)                 a[[0, 1, 2, 3, 4], :] = memoryview(s)
253: (8)                 assert_array_equal(a, s)
254: (8)                 a[:, [0, 1, 2, 3, 4]] = memoryview(s)
255: (8)                 assert_array_equal(a, s)
256: (8)                 a[...] = memoryview(s)
257: (8)                 assert_array_equal(a, s)
258: (4)             def test_subclass_writeable(self):
259: (8)                 d = np.rec.array([('NGC1001', 11), ('NGC1002', 1.), ('NGC1003', 1.)],
260: (25)                         dtype=[('target', 'S20'), ('V_mag', '>f4')])
261: (8)                 ind = np.array([False, True, True], dtype=bool)
262: (8)                 assert_(d[ind].flags.writeable)
263: (8)                 ind = np.array([0, 1])
264: (8)                 assert_(d[ind].flags.writeable)
265: (8)                 assert_(d[...].flags.writeable)
266: (8)                 assert_(d[0].flags.writeable)
267: (4)             def test_memory_order(self):
268: (8)                 a = np.arange(10)
269: (8)                 b = np.arange(10).reshape(5,2).T
270: (8)                 assert_(a[b].flags.f_contiguous)
271: (8)                 a = a.reshape(-1, 1)
272: (8)                 assert_(a[b, 0].flags.f_contiguous)
273: (4)             def test_scalar_return_type(self):
274: (8)                 class Zero:
275: (12)                     def __index__(self):
276: (16)                         return 0
277: (8)                 z = Zero()
278: (8)                 class ArrayLike:
279: (12)                     def __array__(self):
280: (16)                         return np.array(0)
281: (8)                 a = np.zeros(())
282: (8)                 assert_(isinstance(a[()], np.float_))
283: (8)                 a = np.zeros(1)
284: (8)                 assert_(isinstance(a[z], np.float_))
285: (8)                 a = np.zeros((1, 1))
286: (8)                 assert_(isinstance(a[z, np.array(0)], np.float_))
287: (8)                 assert_(isinstance(a[z, ArrayLike()], np.float_))
288: (8)                 b = np.array(0)
289: (8)                 a = np.array(0, dtype=object)
290: (8)                 a[()] = b
291: (8)                 assert_(isinstance(a[()], np.ndarray))
292: (8)                 a = np.array([b, None])
293: (8)                 assert_(isinstance(a[z], np.ndarray))
294: (8)                 a = np.array([[b, None]])
295: (8)                 assert_(isinstance(a[z, np.array(0)], np.ndarray))
296: (8)                 assert_(isinstance(a[z, ArrayLike()], np.ndarray))
297: (4)             def test_small_regressions(self):
298: (8)                 a = np.array([0])
299: (8)                 if HAS_REFCOUNT:
300: (12)                     refcount = sys.getrefcount(np.dtype(np.intp))
301: (8)                     a[np.array([0], dtype=np.intp)] = 1
302: (8)                     a[np.array([0], dtype=np.uint8)] = 1
303: (8)                     assert_raises(IndexError, a.__setitem__,
304: (22)                         np.array([1], dtype=np.intp), 1)
305: (8)                     assert_raises(IndexError, a.__setitem__,
306: (22)                         np.array([1], dtype=np.uint8), 1)
307: (8)                     if HAS_REFCOUNT:
308: (12)                         assert_equal(sys.getrefcount(np.dtype(np.intp)), refcount)
309: (4)             def test_unaligned(self):
310: (8)                 v = (np.zeros(64, dtype=np.int8) + ord('a'))[1:-7]
311: (8)                 d = v.view(np.dtype("S8"))
312: (8)                 x = (np.zeros(16, dtype=np.int8) + ord('a'))[1:-7]
313: (8)                 x = x.view(np.dtype("S8"))
314: (8)                 x[...] = np.array("b" * 8, dtype="S")
315: (8)                 b = np.arange(d.size)
316: (8)                 assert_equal(d[b], d)

```

```

317: (8)             d[b] = x
318: (8)             b = np.zeros(d.size + 1).view(np.int8)[1:-(np.intp(0).itemsize - 1)]
319: (8)             b = b.view(np.intp)[:d.size]
320: (8)             b[...] = np.arange(d.size)
321: (8)             assert_equal(d[b.astype(np.int16)], d)
322: (8)             d[b.astype(np.int16)] = x
323: (8)             d[b % 2 == 0]
324: (8)             d[b % 2 == 0] = x[::-2]
325: (4)             def test_tuple_subclass(self):
326: (8)                 arr = np.ones((5, 5))
327: (8)                 class TupleSubclass(tuple):
328: (12)                     pass
329: (8)                 index = ([1], [1])
330: (8)                 index = TupleSubclass(index)
331: (8)                 assert_(arr[index].shape == (1,))
332: (8)                 assert_(arr[index,].shape != (1,))
333: (4)             def test_broken_sequence_not_nd_index(self):
334: (8)                 class SequenceLike:
335: (12)                     def __index__(self):
336: (16)                         return 0
337: (12)                     def __len__(self):
338: (16)                         return 1
339: (12)                     def __getitem__(self, item):
340: (16)                         raise IndexError('Not possible')
341: (8)                     arr = np.arange(10)
342: (8)                     assert_array_equal(arr[SequenceLike()], arr[SequenceLike(),])
343: (8)                     arr = np.zeros((1,), dtype=[('f1', 'i8'), ('f2', 'i8')])
344: (8)                     assert_array_equal(arr[SequenceLike()], arr[SequenceLike(),])
345: (4)             def test_indexing_array_weird_strides(self):
346: (8)                 x = np.ones(10)
347: (8)                 x2 = np.ones((10, 2))
348: (8)                 ind = np.arange(10)[:, None, None, None]
349: (8)                 ind = np.broadcast_to(ind, (10, 55, 4, 4))
350: (8)                 assert_array_equal(x[ind], x[ind.copy()])
351: (8)                 zind = np.zeros(4, dtype=np.intp)
352: (8)                 assert_array_equal(x2[ind, zind], x2[ind.copy(), zind])
353: (4)             def test_indexing_array_negative_strides(self):
354: (8)                 arro = np.zeros((4, 4))
355: (8)                 arr = arro[::-1, ::-1]
356: (8)                 slices = (slice(None), [0, 1, 2, 3])
357: (8)                 arr[slices] = 10
358: (8)                 assert_array_equal(arr, 10.)
359: (4)             def test_character_assignment(self):
360: (8)                 arr = np.zeros((1, 5), dtype="c")
361: (8)                 arr[0] = np.str_("asdfg") # must assign as a sequence
362: (8)                 assert_array_equal(arr[0], np.array("asdfg", dtype="c"))
363: (8)                 assert arr[0, 1] == b"s" # make sure not all were set to "a" for both
364: (4)             @pytest.mark.parametrize("index",
365: (12)                 [True, False, np.array([0])])
366: (4)             @pytest.mark.parametrize("num", [32, 40])
367: (4)             @pytest.mark.parametrize("original_ndim", [1, 32])
368: (4)             def test_too_many_advanced_indices(self, index, num, original_ndim):
369: (8)                 arr = np.ones((1,) * original_ndim)
370: (8)                 with pytest.raises(IndexError):
371: (12)                     arr[(index,) * num]
372: (8)                 with pytest.raises(IndexError):
373: (12)                     arr[(index,) * num] = 1.
374: (4)             @pytest.mark.skipif(IS_WASM, reason="no threading")
375: (4)             def test_structured_advanced_indexing(self):
376: (8)                 from concurrent.futures import ThreadPoolExecutor
377: (8)                 dt = np.dtype([("", "f8")])
378: (8)                 dt = np.dtype([( "", dt)] * 2)
379: (8)                 dt = np.dtype([( "", dt)] * 2)
380: (8)                 arr = np.random.uniform(size=(6000, 8)).view(dt)[:, 0]
381: (8)                 rng = np.random.default_rng()
382: (8)                 def func(arr):
383: (12)                     idx = rng.integers(0, len(arr), size=6000, dtype=np.intp)
384: (12)                     arr[idx]
385: (8)             tpe = ThreadPoolExecutor(max_workers=8)

```

```

386: (8)             futures = [tpe.submit(func, arr) for _ in range(10)]
387: (8)             for f in futures:
388: (12)                 f.result()
389: (8)             assert arr.dtype is dt
390: (4)         def test_nontuple_ndindex(self):
391: (8)             a = np.arange(25).reshape((5, 5))
392: (8)             assert_equal(a[[0, 1]], np.array([a[0], a[1]]))
393: (8)             assert_equal(a[[0, 1], [0, 1]], np.array([0, 6]))
394: (8)             assert_raises(IndexError, a.__getitem__, [slice(None)])
395: (0)     class TestFieldIndexing:
396: (4)         def test_scalar_return_type(self):
397: (8)             a = np.zeros(() , [('a', 'f8')])
398: (8)             assert_(isinstance(a['a'], np.ndarray))
399: (8)             assert_(isinstance(a[['a']], np.ndarray))
400: (0)     class TestBroadcastedAssignments:
401: (4)         def assign(self, a, ind, val):
402: (8)             a[ind] = val
403: (8)             return a
404: (4)         def test_prepending_ones(self):
405: (8)             a = np.zeros((3, 2))
406: (8)             a[...] = np.ones((1, 3, 2))
407: (8)             a[[0, 1, 2], :] = np.ones((1, 3, 2))
408: (8)             a[:, [0, 1]] = np.ones((1, 3, 2))
409: (8)             a[[[0], [1], [2]], [0, 1]] = np.ones((1, 3, 2))
410: (4)         def test_prepend_not_one(self):
411: (8)             assign = self.assign
412: (8)             s_ = np.s_
413: (8)             a = np.zeros(5)
414: (8)             assert_raises(ValueError, assign, a, s_[...], np.ones((2, 1)))
415: (8)             assert_raises(ValueError, assign, a, s_[[1, 2, 3]], np.ones((2, 1)))
416: (8)             assert_raises(ValueError, assign, a, s_[[[1], [2]]], np.ones((2, 2, 1)))
417: (4)         def test_simple_broadcasting_errors(self):
418: (8)             assign = self.assign
419: (8)             s_ = np.s_
420: (8)             a = np.zeros((5, 1))
421: (8)             assert_raises(ValueError, assign, a, s_[...], np.zeros((5, 2)))
422: (8)             assert_raises(ValueError, assign, a, s_[...], np.zeros((5, 0)))
423: (8)             assert_raises(ValueError, assign, a, s_[:, [0]], np.zeros((5, 2)))
424: (8)             assert_raises(ValueError, assign, a, s_[:, [0]], np.zeros((5, 0)))
425: (8)             assert_raises(ValueError, assign, a, s_[[0], :], np.zeros((2, 1)))
426: (4)         @pytest.mark.parametrize("index", [
427: (12)             (... , [1, 2], slice(None)),
428: (12)             ([0, 1], ... , 0),
429: (12)             (... , [1, 2], [1, 2]))
430: (4)         def test_broadcast_error_reports_correct_shape(self, index):
431: (8)             values = np.zeros((100, 100)) # will never broadcast below
432: (8)             arr = np.zeros((3, 4, 5, 6, 7))
433: (8)             shape_str = str(arr[index].shape).replace(" ", "")
434: (8)             with pytest.raises(ValueError) as e:
435: (12)                 arr[index] = values
436: (8)                 assert str(e.value).endswith(shape_str)
437: (4)         def test_index_is_larger(self):
438: (8)             a = np.zeros((5, 5))
439: (8)             a[[[0], [1], [2]], [0, 1, 2]] = [2, 3, 4]
440: (8)             assert_(a[:3, :3] == [2, 3, 4]).all()
441: (4)         def test_broadcast_subspace(self):
442: (8)             a = np.zeros((100, 100))
443: (8)             v = np.arange(100)[ :, None]
444: (8)             b = np.arange(100)[ :-1]
445: (8)             a[b] = v
446: (8)             assert_((a[ :-1] == v).all())
447: (0)     class TestSubclasses:
448: (4)         def test_basic(self):
449: (8)             class SubClass(np.ndarray):
450: (12)                 pass
451: (8)                 a = np.arange(5)
452: (8)                 s = a.view(SubClass)
453: (8)                 s_slice = s[:3]

```

```

454: (8) assert_(type(s_slice) is SubClass)
455: (8) assert_(s_slice.base is s)
456: (8) assert_array_equal(s_slice, a[:3])
457: (8) s_fancy = s[[0, 1, 2]]
458: (8) assert_(type(s_fancy) is SubClass)
459: (8) assert_(s_fancy.base is not s)
460: (8) assert_(type(s_fancy.base) is np.ndarray)
461: (8) assert_array_equal(s_fancy, a[[0, 1, 2]])
462: (8) assert_array_equal(s_fancy.base, a[[0, 1, 2]])
463: (8) s_bool = s[s > 0]
464: (8) assert_(type(s_bool) is SubClass)
465: (8) assert_(s_bool.base is not s)
466: (8) assert_(type(s_bool.base) is np.ndarray)
467: (8) assert_array_equal(s_bool, a[a > 0])
468: (8) assert_array_equal(s_bool.base, a[a > 0])
469: (4) def test_fancy_on_read_only(self):
470: (8)     class SubClass(np.ndarray):
471: (12)         pass
472: (8)         a = np.arange(5)
473: (8)         s = a.view(SubClass)
474: (8)         s.flags.writeable = False
475: (8)         s_fancy = s[[0, 1, 2]]
476: (8)         assert_(s_fancy.flags.writeable)
477: (4)     def test_finalize_gets_full_info(self):
478: (8)         class SubClass(np.ndarray):
479: (12)             def __array_finalize__(self, old):
480: (16)                 self.finalize_status = np.array(self)
481: (16)                 self.old = old
482: (8)                 s = np.arange(10).view(SubClass)
483: (8)                 new_s = s[:3]
484: (8)                 assert_array_equal(new_s.finalize_status, new_s)
485: (8)                 assert_array_equal(new_s.old, s)
486: (8)                 new_s = s[[0,1,2,3]]
487: (8)                 assert_array_equal(new_s.finalize_status, new_s)
488: (8)                 assert_array_equal(new_s.old, s)
489: (8)                 new_s = s[s > 0]
490: (8)                 assert_array_equal(new_s.finalize_status, new_s)
491: (8)                 assert_array_equal(new_s.old, s)
492: (0)             class TestFancyIndexingCast:
493: (4)                 def test_boolean_index_cast_assign(self):
494: (8)                     shape = (8, 63)
495: (8)                     bool_index = np.zeros(shape).astype(bool)
496: (8)                     bool_index[0, 1] = True
497: (8)                     zero_array = np.zeros(shape)
498: (8)                     zero_array[bool_index] = np.array([1])
499: (8)                     assert_equal(zero_array[0, 1], 1)
500: (8)                     assert_warns(np.ComplexWarning,
501: (21)                         zero_array.__setitem__, ([0], [1]), np.array([2 + 1j]))
502: (8)                     assert_equal(zero_array[0, 1], 2) # No complex part
503: (8)                     assert_warns(np.ComplexWarning,
504: (21)                         zero_array.__setitem__, bool_index, np.array([1j]))
505: (8)                     assert_equal(zero_array[0, 1], 0)
506: (0)             class TestFancyIndexingEquivalence:
507: (4)                 def test_object_assign(self):
508: (8)                     a = np.arange(5, dtype=object)
509: (8)                     b = a.copy()
510: (8)                     a[:3] = [1, (1,2), 3]
511: (8)                     b[[0, 1, 2]] = [1, (1,2), 3]
512: (8)                     assert_array_equal(a, b)
513: (8)                     b = np.arange(5, dtype=object)[None, :]
514: (8)                     b[[0], :3] = [[1, (1,2), 3]]
515: (8)                     assert_array_equal(a, b[0])
516: (8)                     b = b.T
517: (8)                     b[:3, [0]] = [[1], [(1,2)], [3]]
518: (8)                     assert_array_equal(a, b[:, 0])
519: (8)                     arr = np.ones((3, 4, 5), dtype=object)
520: (8)                     cmp_arr = arr.copy()
521: (8)                     cmp_arr[:1, ...] = [[[1], [2], [3], [4]]]
522: (8)                     arr[[0], ...] = [[[1], [2], [3], [4]]]

```

```

523: (8)             assert_array_equal(arr, cmp_arr)
524: (8)             arr = arr.copy('F')
525: (8)             arr[[0], ...] = [[[1], [2], [3], [4]]]
526: (8)             assert_array_equal(arr, cmp_arr)
527: (4)             def test_cast_equivalence(self):
528: (8)                 a = np.arange(5)
529: (8)                 b = a.copy()
530: (8)                 a[:3] = np.array(['2', '-3', '-1'])
531: (8)                 b[[0, 2, 1]] = np.array(['2', '-1', '-3'])
532: (8)                 assert_array_equal(a, b)
533: (8)                 b = np.arange(5)[None, :]
534: (8)                 b[[0], :3] = np.array(['2', '-3', '-1'])
535: (8)                 assert_array_equal(a, b[0])
536: (0)             class TestMultiIndexingAutomated:
537: (4)                 """
538: (4)                     These tests use code to mimic the C-Code indexing for selection.
539: (4)                     NOTE:
540: (8)                         * This still lacks tests for complex item setting.
541: (8)                         * If you change behavior of indexing, you might want to modify
542: (10)                            these tests to try more combinations.
543: (8)                         * Behavior was written to match numpy version 1.8. (though a
544: (10)                            first version matched 1.7.)
545: (8)                         * Only tuple indices are supported by the mimicking code.
546: (10)                            (and tested as of writing this)
547: (8)                         * Error types should match most of the time as long as there
548: (10)                            is only one error. For multiple errors, what gets raised
549: (10)                            will usually not be the same one. They are *not* tested.
550: (4)                         Update 2016-11-30: It is probably not worth maintaining this test
551: (4)                         indefinitely and it can be dropped if maintenance becomes a burden.
552: (4)                         """
553: (4)             def setup_method(self):
554: (8)                 self.a = np.arange(np.prod([3, 1, 5, 6])).reshape(3, 1, 5, 6)
555: (8)                 self.b = np.empty((3, 0, 5, 6))
556: (8)                 self.complex_indices = ['skip', Ellipsis,
557: (12)                               0,
558: (12)                               np.array([True, False, False]),
559: (12)                               np.array([[True, False], [False, True]]),
560: (12)                               np.array([[[False, False], [False, False]]]),
561: (12)                               slice(-5, 5, 2),
562: (12)                               slice(1, 1, 100),
563: (12)                               slice(4, -1, -2),
564: (12)                               slice(None, None, -3),
565: (12)                               np.empty((0, 1, 1), dtype=np.intp), # empty and can be broadcast
566: (12)                               np.array([0, 1, -2]),
567: (12)                               np.array([[2], [0], [1]]),
568: (12)                               np.array([[0, -1], [0, 1]]),
569: (12)                               dtype=np.dtype('intp').newbyteorder()),
570: (12)                               np.array([2, -1], dtype=np.int8),
571: (12)                               np.zeros([1]*31, dtype=int), # trigger too large array.
572: (8)                               np.array([0., 1.]) # invalid datatype
573: (31)                               self.simple_indices = [Ellipsis, None, -1, [1], np.array([True]),
574: (8)   'skip']
575: (4)                               self.fill_indices = [slice(None, None), 0]
576: (8)             def _get_multi_index(self, arr, indices):
577: (8)                 """Mimic multi dimensional indexing.
578: (8)                 Parameters
579: (8)                 -----
580: (12)                   arr : ndarray
581: (8)                       Array to be indexed.
582: (8)                   indices : tuple of index objects
583: (8)                   Returns
584: (8)                   -----
585: (12)                   out : ndarray
586: (12)                       An array equivalent to the indexing operation (but always a copy).
587: (8)                       `arr[indices]` should be identical.
588: (12)                   no_copy : bool
589: (12)                       Whether the indexing operation requires a copy. If this is `True`,
590: (12)                           `np.may_share_memory(arr, arr[indices])` should be `True` (with

```

```

591: (8)          Notes
592: (8)
593: (8)          -----
594: (8)          While the function may mostly match the errors of normal indexing this
595: (8)          is generally not the case.
596: (8)          """
597: (8)          in_indices = list(indices)
598: (8)          indices = []
599: (8)          no_copy = True
600: (8)          num_fancy = 0
601: (8)          fancy_dim = 0
602: (8)          error_unless_broadcast_to_empty = False
603: (8)          ndim = 0
604: (8)          ellipsis_pos = None # define here mostly to replace all but first.
605: (12)         for i, indx in enumerate(in_indices):
606: (16)             if indx is None:
607: (12)                 continue
608: (16)             if isinstance(indx, np.ndarray) and indx.dtype == bool:
609: (16)                 no_copy = False
610: (20)                 if indx.ndim == 0:
611: (16)                     raise IndexError
612: (16)                     ndim += indx.ndim
613: (16)                     fancy_dim += indx.ndim
614: (12)                     continue
615: (16)             if indx is Ellipsis:
616: (20)                 if ellipsis_pos is None:
617: (20)                     ellipsis_pos = i
618: (16)                     continue # do not increment ndim counter
619: (12)                     raise IndexError
620: (16)             if isinstance(indx, slice):
621: (16)                 ndim += 1
622: (12)                 continue
623: (16)             if not isinstance(indx, np.ndarray):
624: (20)                 try:
625: (16)                     indx = np.array(indx, dtype=np.intp)
626: (20)                 except ValueError:
627: (16)                     raise IndexError
628: (12)                     in_indices[i] = indx
629: (16)                 elif indx.dtype.kind != 'b' and indx.dtype.kind != 'i':
630: (33)                     raise IndexError('arrays used as indices must be of '
631: (12)                         'integer (or boolean) type')
632: (16)                 if indx.ndim != 0:
633: (12)                     no_copy = False
634: (12)                     ndim += 1
635: (8)                     fancy_dim += 1
636: (12)                 if arr.ndim - ndim < 0:
637: (8)                     raise IndexError
638: (12)                 if ndim == 0 and None not in in_indices:
639: (8)                     return arr.copy(), no_copy
640: (12)                 if ellipsis_pos is not None:
641: (55)                     in_indices[ellipsis_pos:ellipsis_pos+1] = ([slice(None, None)] *
642: (8)                         (arr.ndim - ndim))
643: (12)                 for ax, indx in enumerate(in_indices):
644: (16)                     if isinstance(indx, slice):
645: (16)                         indx = np.arange(*indx.indices(arr.shape[ax]))
646: (16)                         indices.append(['s', indx])
647: (12)                         continue
648: (16)                     elif indx is None:
649: (16)                         indices.append(['n', np.array([0], dtype=np.intp)])
650: (16)                         arr = arr.reshape((arr.shape[:ax] + (1,) + arr.shape[ax:]))
651: (12)                         continue
652: (16)                     if isinstance(indx, np.ndarray) and indx.dtype == bool:
653: (20)                         if indx.shape != arr.shape[ax:ax+indx.ndim]:
654: (16)                             raise IndexError
655: (20)                         try:
656: (36)                             flat_indx = np.ravel_multi_index(np.nonzero(indx),
657: (16)                                 arr.shape[ax:ax+indx.ndim], mode='raise')
658: (20)                         except Exception:
659: (20)                             error_unless_broadcast_to_empty = True
659: (20)                             flat_indx = np.array([0]*indx.sum(), dtype=np.intp)

```

```

660: (16)             if indx.ndim != 0:
661: (20)                 arr = arr.reshape((arr.shape[:ax]
662: (34)                         + (np.prod(arr.shape[ax:ax+indx.ndim]),)
663: (34)                         + arr.shape[ax+indx.ndim:]))
664: (20)                         indx = flat_indx
665: (16)                     else:
666: (20)                         raise IndexError
667: (12)                 else:
668: (16)                     if indx.ndim == 0:
669: (20)                         if indx >= arr.shape[ax] or indx < -arr.shape[ax]:
670: (24)                             raise IndexError
671: (12)                     if indx.ndim == 0:
672: (16)                         if indx >= arr.shape[ax] or indx < - arr.shape[ax]:
673: (20)                             raise IndexError
674: (12)                     if (len(indices) > 0 and
675: (20)                         indices[-1][0] == 'f' and
676: (20)                         ax != ellipsis_pos):
677: (16)                         indices[-1].append(indx)
678: (12)                 else:
679: (16)                     num_fancy += 1
680: (16)                     indices.append(['f', indx])
681: (8)             if num_fancy > 1 and not no_copy:
682: (12)                 new_indices = indices[:]
683: (12)                 axes = list(range(arr.ndim))
684: (12)                 fancy_axes = []
685: (12)                 new_indices.insert(0, ['f'])
686: (12)                 ni = 0
687: (12)                 ai = 0
688: (12)                 for indx in indices:
689: (16)                     ni += 1
690: (16)                     if indx[0] == 'f':
691: (20)                         new_indices[0].extend(indx[1:])
692: (20)                         del new_indices[ni]
693: (20)                         ni -= 1
694: (20)                         for ax in range(ai, ai + len(indx[1:])):
695: (24)                             fancy_axes.append(ax)
696: (24)                             axes.remove(ax)
697: (16)                         ai += len(indx) - 1 # axis we are at
698: (12)             indices = new_indices
699: (12)             arr = arr.transpose(*(fancy_axes + axes))
700: (8)         ax = 0
701: (8)         for indx in indices:
702: (12)             if indx[0] == 'f':
703: (16)                 if len(indx) == 1:
704: (20)                     continue
705: (16)                 orig_shape = arr.shape
706: (16)                 orig_slice = orig_shape[ax:ax + len(indx[1:])]
707: (16)                 arr = arr.reshape((arr.shape[:ax]
708: (36)                                 + (np.prod(orig_slice).astype(int),)
709: (36)                                 + arr.shape[ax + len(indx[1:]):]))
710: (16)                 res = np.broadcast(*indx[1:])
711: (16)                 if res.size != 0:
712: (20)                     if errorUnlessBroadcastToEmpty:
713: (24)                         raise IndexError
714: (20)                     for _indx, _size in zip(indx[1:], orig_slice):
715: (24)                         if _indx.size == 0:
716: (28)                             continue
717: (24)                         if np.any(_indx >= _size) or np.any(_indx < -_size):
718: (32)                             raise IndexError
719: (16)                     if len(indx[1:]) == len(orig_slice):
720: (20)                         if np.prod(orig_slice) == 0:
721: (24)                             try:
722: (28)                                 mi = np.ravel_multi_index(indx[1:], orig_slice,
723: (54)                                     mode='raise')
724: (24)                             except Exception:
725: (28)                                 raise IndexError('invalid index into 0-sized')
726: (20)                         else:
727: (24)                             mi = np.ravel_multi_index(indx[1:], orig_slice,
728: (50)                                     mode='wrap')

```

```

729: (16)
730: (20)
731: (16)
732: (16)
733: (20)
734: (40)
735: (40)
736: (16)
737: (20)
738: (16)
739: (16)
740: (12)
741: (12)
742: (8)
743: (4)
744: (8)
745: (8)
746: (8)
747: (8)
748: (12)
749: (8)
750: (12)
751: (8)
752: (8)
753: (12)
754: (8)
755: (12)
756: (16)
757: (12)
758: (12)
759: (12)
760: (16)
761: (12)
762: (8)
763: (4)
764: (8)
765: (8)
766: (8)
767: (8)
768: (12)
769: (8)
770: (12)
771: (12)
772: (8)
773: (8)
774: (12)
775: (8)
776: (12)
777: (16)
778: (12)
779: (12)
780: (12)
781: (16)
782: (12)
783: (8)
784: (4)
785: (8)
786: (8)
787: (8)
788: (8)
789: (8)
790: (8)
791: (12)
792: (12)
793: (16)
794: (20)
795: (16)
796: (20)
797: (8)

                else:
                    raise ValueError
                arr = arr.take(mi.ravel(), axis=ax)
            try:
                arr = arr.reshape((arr.shape[:ax]
                                   + mi.shape
                                   + arr.shape[ax+1:]))

            except ValueError:
                raise IndexError
            ax += mi.ndim
            continue
        arr = arr.take(indx[1], axis=ax)
        ax += 1
    return arr, no_copy
def _check_multi_index(self, arr, index):
    """Check a multi index item getting and simple setting.
    Parameters
    -----
    arr : ndarray
        Array to be indexed, must be a reshaped arange.
    index : tuple of indexing objects
        Index being tested.
    """
    try:
        mimic_get, no_copy = self._get_multi_index(arr, index)
    except Exception as e:
        if HAS_REFCOUNT:
            prev_refcount = sys.getrefcount(arr)
            assert_raises(type(e), arr.__getitem__, index)
            assert_raises(type(e), arr.__setitem__, index, 0)
        if HAS_REFCOUNT:
            assert_equal(prev_refcount, sys.getrefcount(arr))
        return
    self._compare_index_result(arr, index, mimic_get, no_copy)
def _check_single_index(self, arr, index):
    """Check a single index item getting and simple setting.
    Parameters
    -----
    arr : ndarray
        Array to be indexed, must be an arange.
    index : indexing object
        Index being tested. Must be a single index and not a tuple
        of indexing objects (see also `'_check_multi_index`).
    """
    try:
        mimic_get, no_copy = self._get_multi_index(arr, (index,))
    except Exception as e:
        if HAS_REFCOUNT:
            prev_refcount = sys.getrefcount(arr)
            assert_raises(type(e), arr.__getitem__, index)
            assert_raises(type(e), arr.__setitem__, index, 0)
        if HAS_REFCOUNT:
            assert_equal(prev_refcount, sys.getrefcount(arr))
        return
    self._compare_index_result(arr, index, mimic_get, no_copy)
def _compare_index_result(self, arr, index, mimic_get, no_copy):
    """Compare mimicked result to indexing result.
    """
    arr = arr.copy()
    indexed_arr = arr[index]
    assert_array_equal(indexed_arr, mimic_get)
    if indexed_arr.size != 0 and indexed_arr.ndim != 0:
        assert_(np.may_share_memory(indexed_arr, arr) == no_copy)
        if HAS_REFCOUNT:
            if no_copy:
                assert_equal(sys.getrefcount(arr), 3)
            else:
                assert_equal(sys.getrefcount(arr), 2)
    b = arr.copy()

```

```

798: (8)             b[index] = mimic_get + 1000
799: (8)             if b.size == 0:
800: (12)             return # nothing to compare here...
801: (8)             if no_copy and indexed_arr.ndim != 0:
802: (12)                 indexed_arr += 1000
803: (12)                 assert_array_equal(arr, b)
804: (12)                 return
805: (8)             arr.flat[indexed_arr.ravel()] += 1000
806: (8)             assert_array_equal(arr, b)
807: (4)             def test_boolean(self):
808: (8)                 a = np.array(5)
809: (8)                 assert_equal(a[np.array(True)], 5)
810: (8)                 a[np.array(True)] = 1
811: (8)                 assert_equal(a, 1)
812: (8)                 self._check_multi_index(
813: (12)                     self.a, (np.zeros_like(self.a, dtype=bool),))
814: (8)                 self._check_multi_index(
815: (12)                     self.a, (np.zeros_like(self.a, dtype=bool)[..., 0],))
816: (8)                 self._check_multi_index(
817: (12)                     self.a, (np.zeros_like(self.a, dtype=bool)[None, ...],))
818: (4)             def test_multidim(self):
819: (8)                 with warnings.catch_warnings():
820: (12)                     warnings.filterwarnings('error', '', DeprecationWarning)
821: (12)                     warnings.filterwarnings('error', '', np.VisibleDeprecationWarning)
822: (12)                     def isskip(idx):
823: (16)                         return isinstance(idx, str) and idx == "skip"
824: (12)                     for simple_pos in [0, 2, 3]:
825: (16)                         tocheck = [self.fill_indices, self.complex_indices,
826: (27)                             self.fill_indices, self.fill_indices]
827: (16)                         tocheck[simple_pos] = self.simple_indices
828: (16)                         for index in product(*tocheck):
829: (20)                             index = tuple(i for i in index if not isskip(i))
830: (20)                             self._check_multi_index(self.a, index)
831: (20)                             self._check_multi_index(self.b, index)
832: (8)                             self._check_multi_index(self.a, (0, 0, 0, 0))
833: (8)                             self._check_multi_index(self.b, (0, 0, 0, 0))
834: (8)                             assert_raises(IndexError, self.a.__getitem__, (0, 0, 0, 0, 0))
835: (8)                             assert_raises(IndexError, self.a.__setitem__, (0, 0, 0, 0, 0), 0)
836: (8)                             assert_raises(IndexError, self.a.__getitem__, (0, 0, [1], 0, 0))
837: (8)                             assert_raises(IndexError, self.a.__setitem__, (0, 0, [1], 0, 0), 0)
838: (4)             def test_1d(self):
839: (8)                 a = np.arange(10)
840: (8)                 for index in self.complex_indices:
841: (12)                     self._check_single_index(a, index)
842: (0)             class TestFloatNonIntegerArgument:
843: (4)                 """
844: (4)                     These test that ``TypeError`` is raised when you try to use
845: (4)                     non-integers as arguments to for indexing and slicing e.g. ``a[0.0:5]``
846: (4)                     and ``a[0.5]``, or other functions like ``array.reshape(1., -1)``.
847: (4)                     """
848: (4)             def test_valid_indexing(self):
849: (8)                 a = np.array([[[5]]])
850: (8)                 a[np.array([0])]
851: (8)                 a[[0, 0]]
852: (8)                 a[:, [0, 0]]
853: (8)                 a[:, 0,:]
854: (8)                 a[:, :, :]

855: (4)             def test_valid_slicing(self):
856: (8)                 a = np.array([[[5]]])
857: (8)                 a[:, :]
858: (8)                 a[0,:]
859: (8)                 a[:2]
860: (8)                 a[0:2]
861: (8)                 a[:, 2]
862: (8)                 a[1::2]
863: (8)                 a[:, 2:2]
864: (8)                 a[1:2:2]

865: (4)             def test_non_integer_argument_errors(self):
866: (8)                 a = np.array([[5]])

```

```

867: (8) assert_raises(TypeError, np.reshape, a, (1., 1., -1))
868: (8) assert_raises(TypeError, np.reshape, a, (np.array(1.), -1))
869: (8) assert_raises(TypeError, np.take, a, [0], 1.)
870: (8) assert_raises(TypeError, np.take, a, [0], np.float64(1.))
871: (4) def test_non_integer_sequence_multiplication(self):
872: (8)     def mult(a, b):
873: (12)         return a * b
874: (8)     assert_raises(TypeError, mult, [1], np.float_(3))
875: (8)     mult([1], np.int_(3))
876: (4) def test_reduce_axis_float_index(self):
877: (8)     d = np.zeros((3,3,3))
878: (8)     assert_raises(TypeError, np.min, d, 0.5)
879: (8)     assert_raises(TypeError, np.min, d, (0.5, 1))
880: (8)     assert_raises(TypeError, np.min, d, (1, 2.2))
881: (8)     assert_raises(TypeError, np.min, d, (.2, 1.2))
882: (0) class TestBooleanIndexing:
883: (4)     def test_bool_as_int_argument_errors(self):
884: (8)         a = np.array([[[1]]])
885: (8)         assert_raises(TypeError, np.reshape, a, (True, -1))
886: (8)         assert_raises(TypeError, np.reshape, a, (np.bool_(True), -1))
887: (8)         assert_raises(TypeError, operator.index, np.array(True))
888: (8)         assert_warns(DeprecationWarning, operator.index, np.True_)
889: (8)         assert_raises(TypeError, np.take, args=(a, [0], False))
890: (4)     def test_boolean_indexing_weirdness(self):
891: (8)         a = np.ones((2, 3, 4))
892: (8)         assert a[False, True, ...].shape == (0, 2, 3, 4)
893: (8)         assert a[True, [0, 1], True, True, [1], [[2]]].shape == (1, 2)
894: (8)         assert_raises(IndexError, lambda: a[False, [0, 1], ...])
895: (4)     def test_boolean_indexing_fast_path(self):
896: (8)         a = np.ones((3, 3))
897: (8)         idx1 = np.array([[False]*9])
898: (8)         assert_raises_regex(IndexError,
899: (12)             "boolean index did not match indexed array along dimension 0; "
900: (12)             "dimension is 3 but corresponding boolean dimension is 1",
901: (12)             lambda: a[idx1])
902: (8)         idx2 = np.array([[False]*8 + [True]])
903: (8)         assert_raises_regex(IndexError,
904: (12)             "boolean index did not match indexed array along dimension 0; "
905: (12)             "dimension is 3 but corresponding boolean dimension is 1",
906: (12)             lambda: a[idx2])
907: (8)         idx3 = np.array([[False]*10])
908: (8)         assert_raises_regex(IndexError,
909: (12)             "boolean index did not match indexed array along dimension 0; "
910: (12)             "dimension is 3 but corresponding boolean dimension is 1",
911: (12)             lambda: a[idx3])
912: (8)         a = np.ones((1, 1, 2))
913: (8)         idx = np.array([[[True], [False]]])
914: (8)         assert_raises_regex(IndexError,
915: (12)             "boolean index did not match indexed array along dimension 1; "
916: (12)             "dimension is 1 but corresponding boolean dimension is 2",
917: (12)             lambda: a[idx])
918: (0) class TestArrayToIndexDeprecation:
919: (4)     """Creating an index from array not 0-D is an error.
920: (4)     """
921: (4)     def test_array_to_index_error(self):
922: (8)         a = np.array([[[1]]])
923: (8)         assert_raises(TypeError, operator.index, np.array([1]))
924: (8)         assert_raises(TypeError, np.reshape, a, (a, -1))
925: (8)         assert_raises(TypeError, np.take, a, [0], a)
926: (0) class TestNonIntegerArrayList:
927: (4)     """Tests that array_likes only valid if can safely cast to integer.
928: (4)     For instance, lists give IndexError when they cannot be safely cast to
929: (4)     an integer.
930: (4)     """
931: (4)     def test_basic(self):
932: (8)         a = np.arange(10)
933: (8)         assert_raises(IndexError, a.__getitem__, [0.5, 1.5])
934: (8)         assert_raises(IndexError, a.__getitem__, ('1', '2',))
935: (8)         a.__getitem__([])

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

936: (0)         class TestMultipleEllipsisError:
937: (4)             """An index can only have a single ellipsis.
938: (4)             """
939: (4)             def test_basic(self):
940: (8)                 a = np.arange(10)
941: (8)                 assert_raises(IndexError, lambda: a[..., ...])
942: (8)                 assert_raises(IndexError, a.__getitem__, ((Ellipsis,) * 2,))
943: (8)                 assert_raises(IndexError, a.__getitem__, ((Ellipsis,) * 3,))
944: (0)         class TestCApiAccess:
945: (4)             def test_getitem(self):
946: (8)                 subscript = functools.partial(array_indexing, 0)
947: (8)                 assert_raises(IndexError, subscript, np.ones(()), 0)
948: (8)                 assert_raises(IndexError, subscript, np.ones(10), 11)
949: (8)                 assert_raises(IndexError, subscript, np.ones(10), -11)
950: (8)                 assert_raises(IndexError, subscript, np.ones((10, 10)), 11)
951: (8)                 assert_raises(IndexError, subscript, np.ones((10, 10)), -11)
952: (8)                 a = np.arange(10)
953: (8)                 assert_array_equal(a[4], subscript(a, 4))
954: (8)                 a = a.reshape(5, 2)
955: (8)                 assert_array_equal(a[-4], subscript(a, -4))
956: (4)             def test_setitem(self):
957: (8)                 assign = functools.partial(array_indexing, 1)
958: (8)                 assert_raises(ValueError, assign, np.ones(10), 0)
959: (8)                 assert_raises(IndexError, assign, np.ones(()), 0, 0)
960: (8)                 assert_raises(IndexError, assign, np.ones(10), 11, 0)
961: (8)                 assert_raises(IndexError, assign, np.ones(10), -11, 0)
962: (8)                 assert_raises(IndexError, assign, np.ones((10, 10)), 11, 0)
963: (8)                 assert_raises(IndexError, assign, np.ones((10, 10)), -11, 0)
964: (8)                 a = np.arange(10)
965: (8)                 assign(a, 4, 10)
966: (8)                 assert_(a[4] == 10)
967: (8)                 a = a.reshape(5, 2)
968: (8)                 assign(a, 4, 10)
969: (8)                 assert_array_equal(a[-1], [10, 10])

```

---

File 101 - test\_item\_selection.py:

```

1: (0)         import sys
2: (0)         import pytest
3: (0)         import numpy as np
4: (0)         from numpy.testing import (
5: (4)             assert_, assert_raises, assert_array_equal, HAS_REFCOUNT
6: (4)         )
7: (0)         class TestTake:
8: (4)             def test_simple(self):
9: (8)                 a = [[1, 2], [3, 4]]
10: (8)                a_str = [[b'1', b'2'], [b'3', b'4']]
11: (8)                modes = ['raise', 'wrap', 'clip']
12: (8)                indices = [-1, 4]
13: (8)                index_arrays = [np.empty(0, dtype=np.intp),
14: (24)                    np.empty(tuple(), dtype=np.intp),
15: (24)                    np.empty((1, 1), dtype=np.intp)]
16: (8)                real_indices = {'raise': {-1: 1, 4: IndexError},
17: (24)                    'wrap': {-1: 1, 4: 0},
18: (24)                    'clip': {-1: 0, 4: 1}}
19: (8)                types = int, object, np.dtype([('i1', 'i2', 3)])
20: (8)                for t in types:
21: (12)                    ta = np.array(a if np.issubdtype(t, np.number) else a_str,
22: (12)                      dtype=t)
23: (12)                    tresult = list(ta.T.copy())
24: (16)                    for index_array in index_arrays:
25: (20)                        if index_array.size != 0:
26: (20)                            tresult[0].shape = (2,) + index_array.shape
27: (16)                            tresult[1].shape = (2,) + index_array.shape
28: (20)                            for mode in modes:
29: (24)                                for index in indices:

```

```

30: (24)                     if real_index is IndexError and index_array.size != 0:
31: (28)                         index_array.put(0, index)
32: (28)                         assert_raises(IndexError, ta.take, index_array,
33: (42)   mode=mode, axis=1)
34: (24)                     elif index_array.size != 0:
35: (28)                         index_array.put(0, index)
36: (28)                         res = ta.take(index_array, mode=mode, axis=1)
37: (28)                         assert_array_equal(res, tresult[real_index])
38: (24)                     else:
39: (28)                         res = ta.take(index_array, mode=mode, axis=1)
40: (28)                         assert_(res.shape == (2,) + index_array.shape)
41: (4)             def test_refcounting(self):
42: (8)                 objects = [object() for i in range(10)]
43: (8)                 for mode in ('raise', 'clip', 'wrap'):
44: (12)                     a = np.array(objects)
45: (12)                     b = np.array([2, 2, 4, 5, 3, 5])
46: (12)                     a.take(b, out=a[:6], mode=mode)
47: (12)                     del a
48: (12)                     if HAS_REFCOUNT:
49: (16)                         assert_(all(sys.getrefcount(o) == 3 for o in objects))
50: (12)                     a = np.array(objects * 2)[:2]
51: (12)                     a.take(b, out=a[:6], mode=mode)
52: (12)                     del a
53: (12)                     if HAS_REFCOUNT:
54: (16)                         assert_(all(sys.getrefcount(o) == 3 for o in objects))
55: (4)             def test_unicode_mode(self):
56: (8)                 d = np.arange(10)
57: (8)                 k = b'\xc3\xaa'.decode("UTF8")
58: (8)                 assert_raises(ValueError, d.take, 5, mode=k)
59: (4)             def test_empty_partition(self):
60: (8)                 a_original = np.array([0, 2, 4, 6, 8, 10])
61: (8)                 a = a_original.copy()
62: (8)                 a.partition(np.array([], dtype=np.int16))
63: (8)                 assert_array_equal(a, a_original)
64: (4)             def test_empty_argpartition(self):
65: (8)                 a = np.array([0, 2, 4, 6, 8, 10])
66: (8)                 a = a.argmax(np.array([], dtype=np.int16))
67: (8)                 b = np.array([0, 1, 2, 3, 4, 5])
68: (8)                 assert_array_equal(a, b)
69: (0)         class TestPutMask:
70: (4)             @pytest.mark.parametrize("dtype", list(np.typecodes["All"]) + ["i,0"])
71: (4)             def test_simple(self, dtype):
72: (8)                 if dtype.lower() == "m":
73: (12)                     dtype += "8[ns]"
74: (8)                     vals = np.arange(1001).astype(dtype=dtype)
75: (8)                     mask = np.random.randint(2, size=1000).astype(bool)
76: (8)                     arr = np.zeros(1000, dtype=vals.dtype)
77: (8)                     zeros = arr.copy()
78: (8)                     np.putmask(arr, mask, vals)
79: (8)                     assert_array_equal(arr[mask], vals[:len(mask)][mask])
80: (8)                     assert_array_equal(arr[~mask], zeros[~mask])
81: (4)             @pytest.mark.parametrize("dtype", list(np.typecodes["All"])[1:] + ["i,0"])
82: (4)             @pytest.mark.parametrize("mode", ["raise", "wrap", "clip"])
83: (4)             def test_empty(self, dtype, mode):
84: (8)                 arr = np.zeros(1000, dtype=dtype)
85: (8)                 arr_copy = arr.copy()
86: (8)                 mask = np.random.randint(2, size=1000).astype(bool)
87: (8)                 np.put(arr, mask, [])
88: (8)                 assert_array_equal(arr, arr_copy)
89: (0)         class TestPut:
90: (4)             @pytest.mark.parametrize("dtype", list(np.typecodes["All"])[1:] + ["i,0"])
91: (4)             @pytest.mark.parametrize("mode", ["raise", "wrap", "clip"])
92: (4)             def test_simple(self, dtype, mode):
93: (8)                 if dtype.lower() == "m":
94: (12)                     dtype += "8[ns]"
95: (8)                     vals = np.arange(1001).astype(dtype=dtype)
96: (8)                     arr = np.zeros(1000, dtype=vals.dtype)
97: (8)                     zeros = arr.copy()
98: (8)                     if mode == "clip":
```

```

99: (12)             idx = np.random.permutation(len(arr) - 2)[-500] + 1
100: (12)            idx[-1] = 0
101: (12)            idx[-2] = len(arr) - 1
102: (12)            idx_put = idx.copy()
103: (12)            idx_put[-1] = -1389
104: (12)            idx_put[-2] = 1321
105: (8)             else:
106: (12)                 idx = np.random.permutation(len(arr) - 3)[-500]
107: (12)                 idx_put = idx
108: (12)                 if mode == "wrap":
109: (16)                     idx_put = idx_put + len(arr)
110: (8)             np.put(arr, idx_put, vals, mode=mode)
111: (8)             assert_array_equal(arr[idx], vals[:len(idx)])
112: (8)             untouched = np.ones(len(arr), dtype=bool)
113: (8)             untouched[idx] = False
114: (8)             assert_array_equal(arr[untouched], zeros[:untouched.sum()])
115: (4)             @pytest.mark.parametrize("dtype", list(np.typecodes["All"])[1:] + ["i,O"])
116: (4)             @pytest.mark.parametrize("mode", ["raise", "wrap", "clip"])
117: (4)             def test_empty(self, dtype, mode):
118: (8)                 arr = np.zeros(1000, dtype=dtype)
119: (8)                 arr_copy = arr.copy()
120: (8)                 np.put(arr, [1, 2, 3], [])
121: (8)                 assert_array_equal(arr, arr_copy)

```

-----

## File 102 - test\_limited\_api.py:

```

1: (0)             import os
2: (0)             import shutil
3: (0)             import subprocess
4: (0)             import sys
5: (0)             import sysconfig
6: (0)             import pytest
7: (0)             from numpy.testing import IS_WASM
8: (0)             @pytest.mark.skipif(IS_WASM, reason="Can't start subprocess")
9: (0)             @pytest.mark.xfail(
10: (4)                 sysconfig.get_config_var("Py_DEBUG"),
11: (4)                 reason=(
12: (8)                     "Py_LIMITED_API is incompatible with Py_DEBUG, Py_TRACE_REFS, "
13: (8)                     "and Py_REF_DEBUG"
14: (4)                 ),
15: (0)             )
16: (0)             def test_limited_api(tmp_path):
17: (4)                 """Test building a third-party C extension with the limited API."""
18: (4)                 here = os.path.dirname(__file__)
19: (4)                 ext_dir = os.path.join(here, "examples", "limited_api")
20: (4)                 cytest = str(tmp_path / "limited_api")
21: (4)                 shutil.copytree(ext_dir, cytest)
22: (4)                 install_log = str(tmp_path / "tmp_install_log.txt")
23: (4)                 subprocess.check_output(
24: (8)                     [
25: (12)                         sys.executable,
26: (12)                         "setup.py",
27: (12)                         "build",
28: (12)                         "install",
29: (12)                         "--prefix", str(tmp_path / "installdir"),
30: (12)                         "--single-version-externally-managed",
31: (12)                         "--record",
32: (12)                         install_log,
33: (8)                         ],
34: (8)                         cwd=cytest,
35: (4)                     )

```

-----

## File 103 - test\_longdouble.py:

```
1: (0)             import warnings
```

```

2: (0) import platform
3: (0) import pytest
4: (0) import numpy as np
5: (0) from numpy.testing import (
6: (4)     assert_, assert_equal, assert_raises, assert_warns, assert_array_equal,
7: (4)     temppath, IS_MUSL
8: (4) )
9: (0) from numpy.core.tests._locales import CommaDecimalPointLocale
10: (0) LD_INFO = np.finfo(np.longdouble)
11: (0) longdouble_longer_than_double = (LD_INFO.eps < np.finfo(np.double).eps)
12: (0) _o = 1 + LD_INFO.eps
13: (0) string_to_longdouble_inaccurate = (_o != np.longdouble(repr(_o)))
14: (0) del _o
15: (0) def test_scalar_extraction():
16: (4)     """Confirm that extracting a value doesn't convert to python float"""
17: (4)     o = 1 + LD_INFO.eps
18: (4)     a = np.array([o, o, o])
19: (4)     assert_equal(a[1], o)
20: (0)     repr_precision = len(repr(np.longdouble(0.1)))
21: (0)     @pytest.mark.skipif(IS_MUSL,
22: (20)             reason="test flaky on musllinux")
23: (0)     @pytest.mark.skipif(LD_INFO.precision + 2 >= repr_precision,
24: (20)             reason="repr precision not enough to show eps")
25: (0) def test_repr_roundtrip():
26: (4)     o = 1 + LD_INFO.eps
27: (4)     assert_equal(np.longdouble(repr(o)), o, "repr was %s" % repr(o))
28: (0)     @pytest.mark.skipif(string_to_longdouble_inaccurate, reason="Need strtold_l")
29: (0) def test_repr_roundtrip_bytes():
30: (4)     o = 1 + LD_INFO.eps
31: (4)     assert_equal(np.longdouble(repr(o).encode("ascii")), o)
32: (0)     @pytest.mark.skipif(string_to_longdouble_inaccurate, reason="Need strtold_l")
33: (0)     @pytest.mark.parametrize("strtype", (np.str_, np.bytes_, str, bytes))
34: (0) def test_array_and_stringlike_roundtrip(strtype):
35: (4)     """
36: (4)     Test that string representations of long-double roundtrip both
37: (4)     for array casting and scalar coercion, see also gh-15608.
38: (4) """
39: (4)     o = 1 + LD_INFO.eps
40: (4)     if strtype in (np.bytes_, bytes):
41: (8)         o_str = strtype(repr(o).encode("ascii"))
42: (4)     else:
43: (8)         o_str = strtype(repr(o))
44: (4)     assert o == np.longdouble(o_str)
45: (4)     o_strarr = np.asarray([o] * 3, dtype=strtype)
46: (4)     assert (o == o_strarr.astype(np.longdouble)).all()
47: (4)     assert (o_strarr == o_str).all()
48: (4)     assert (np.asarray([o] * 3).astype(strtype) == o_str).all()
49: (0) def test_bogus_string():
50: (4)     assert_raises(ValueError, np.longdouble, "spam")
51: (4)     assert_raises(ValueError, np.longdouble, "1.0 flub")
52: (0)     @pytest.mark.skipif(string_to_longdouble_inaccurate, reason="Need strtold_l")
53: (0) def test_fromstring():
54: (4)     o = 1 + LD_INFO.eps
55: (4)     s = (" " + repr(o))*5
56: (4)     a = np.array([o]*5)
57: (4)     assert_equal(np.fromstring(s, sep=" ", dtype=np.longdouble), a,
58: (17)           err_msg="reading '%s'" % s)
59: (0) def test_fromstring_complex():
60: (4)     for ctype in ["complex", "cdouble", "cfloat"]:
61: (8)         assert_equal(np.fromstring("1, 2 , 3 ,4", sep=",", dtype=ctype),
62: (21)             np.array([1., 2., 3., 4.]))
63: (8)         assert_equal(np.fromstring("1j, -2j, 3j, 4e1j", sep=",",
64: (21)               np.array([1.j, -2.j, 3.j, 40.j])))
65: (8)         assert_equal(np.fromstring("1+1j,2-2j, -3+3j, -4e1+4j", sep=",",
66: (21)               np.array([1. + 1.j, 2. - 2.j, - 3. + 3.j, - 40. + 4j])))
67: (8)         with assert_warnings(DeprecationWarning):
68: (12)             assert_equal(np.fromstring("1+2 j,3", dtype=ctype, sep=","),
```

```

69: (25)                                np.array([1.]))
70: (8)       with assert_warns(DeprecationWarning):
71: (12)           assert_equal(np.fromstring("1+ 2j,3", dtype=ctype, sep=","),
72: (25)               np.array([1.]))
73: (8)       with assert_warns(DeprecationWarning):
74: (12)           assert_equal(np.fromstring("1 +2j,3", dtype=ctype, sep=","),
75: (25)               np.array([1.]))
76: (8)       with assert_warns(DeprecationWarning):
77: (12)           assert_equal(np.fromstring("1+j", dtype=ctype, sep=","),
78: (25)               np.array([1.]))
79: (8)       with assert_warns(DeprecationWarning):
80: (12)           assert_equal(np.fromstring("1+", dtype=ctype, sep=","),
81: (25)               np.array([1.]))
82: (8)       with assert_warns(DeprecationWarning):
83: (12)           assert_equal(np.fromstring("1j+1", dtype=ctype, sep=","),
84: (25)               np.array([1j])))
85: (0)   def test_fromstring_bogus():
86: (4)       with assert_warns(DeprecationWarning):
87: (8)           assert_equal(np.fromstring("1. 2. 3. flop 4.", dtype=float, sep=" "),
88: (21)               np.array([1., 2., 3.]))
89: (0)   def test_fromstring_empty():
90: (4)       with assert_warns(DeprecationWarning):
91: (8)           assert_equal(np.fromstring("xxxxx", sep="x"),
92: (21)               np.array([])))
93: (0)   def test_fromstring_missing():
94: (4)       with assert_warns(DeprecationWarning):
95: (8)           assert_equal(np.fromstring("1xx3x4x5x6", sep="x"),
96: (21)               np.array([1])))
97: (0)   class TestFileBased:
98: (4)       ldbl = 1 + LD_INFO.eps
99: (4)       tgt = np.array([ldbl]*5)
100: (4)      out = ''.join([repr(t) + '\n' for t in tgt])
101: (4)      def test_fromfile_bogus(self):
102: (8)          with temppath() as path:
103: (12)              with open(path, 'w') as f:
104: (16)                  f.write("1. 2. 3. flop 4.\n")
105: (12)              with assert_warns(DeprecationWarning):
106: (16)                  res = np.fromfile(path, dtype=float, sep=" ")
107: (8)          assert_equal(res, np.array([1., 2., 3.]))
108: (4)      def test_fromfile_complex(self):
109: (8)          for ctype in ["complex", "cdouble", "cfloat"]:
110: (12)              with temppath() as path:
111: (16)                  with open(path, 'w') as f:
112: (20)                      f.write("1, 2 , 3 ,4\n")
113: (16)                      res = np.fromfile(path, dtype=ctype, sep=",")
114: (12)          assert_equal(res, np.array([1., 2., 3., 4.]))
115: (12)          with temppath() as path:
116: (16)              with open(path, 'w') as f:
117: (20)                  f.write("1j, -2j, 3j, 4e1j\n")
118: (16)                  res = np.fromfile(path, dtype=ctype, sep=",")
119: (12)          assert_equal(res, np.array([1.j, -2.j, 3.j, 40.j]))
120: (12)          with temppath() as path:
121: (16)              with open(path, 'w') as f:
122: (20)                  f.write("1+1j,2-2j, -3+3j, -4e1+4j\n")
123: (16)                  res = np.fromfile(path, dtype=ctype, sep=",")
124: (12)          assert_equal(res, np.array([1. + 1.j, 2. - 2.j, - 3. + 3.j, - 40.
+ 4j]))
125: (12)          with temppath() as path:
126: (16)              with open(path, 'w') as f:
127: (20)                  f.write("1+2 j,3\n")
128: (16)                  with assert_warns(DeprecationWarning):
129: (20)                      res = np.fromfile(path, dtype=ctype, sep=",")
130: (12)                  assert_equal(res, np.array([1.]))
131: (12)                  with temppath() as path:
132: (16)                      with open(path, 'w') as f:
133: (20)                          f.write("1+ 2j,3\n")
134: (16)                          with assert_warns(DeprecationWarning):
135: (20)                              res = np.fromfile(path, dtype=ctype, sep=",")
136: (12)                          assert_equal(res, np.array([1.]))

```

```

137: (12)           with temppath() as path:
138: (16)             with open(path, 'w') as f:
139: (20)               f.write("1 +2j,3\n")
140: (16)             with assert_warns(DeprecationWarning):
141: (20)               res = np.fromfile(path, dtype=ctype, sep=",")
142: (12)             assert_equal(res, np.array([1.]))
143: (12)             with temppath() as path:
144: (16)               with open(path, 'w') as f:
145: (20)                 f.write("1+j\n")
146: (16)               with assert_warns(DeprecationWarning):
147: (20)                 res = np.fromfile(path, dtype=ctype, sep=",")
148: (12)               assert_equal(res, np.array([1.]))
149: (12)             with temppath() as path:
150: (16)               with open(path, 'w') as f:
151: (20)                 f.write("1+\n")
152: (16)               with assert_warns(DeprecationWarning):
153: (20)                 res = np.fromfile(path, dtype=ctype, sep=",")
154: (12)               assert_equal(res, np.array([1.]))
155: (12)             with temppath() as path:
156: (16)               with open(path, 'w') as f:
157: (20)                 f.write("1j+1\n")
158: (16)               with assert_warns(DeprecationWarning):
159: (20)                 res = np.fromfile(path, dtype=ctype, sep=",")
160: (12)               assert_equal(res, np.array([1.j]))
161: (4)             @pytest.mark.skipif(string_to_longdouble_inaccurate,
162: (24)                           reason="Need strtold_l")
163: (4)             def test_fromfile(self):
164: (8)               with temppath() as path:
165: (12)                 with open(path, 'w') as f:
166: (16)                   f.write(self.out)
167: (12)                 res = np.fromfile(path, dtype=np.longdouble, sep="\n")
168: (8)                 assert_equal(res, self.tgt)
169: (4)             @pytest.mark.skipif(string_to_longdouble_inaccurate,
170: (24)                           reason="Need strtold_l")
171: (4)             def test_genfromtxt(self):
172: (8)               with temppath() as path:
173: (12)                 with open(path, 'w') as f:
174: (16)                   f.write(self.out)
175: (12)                 res = np.genfromtxt(path, dtype=np.longdouble)
176: (8)                 assert_equal(res, self.tgt)
177: (4)             @pytest.mark.skipif(string_to_longdouble_inaccurate,
178: (24)                           reason="Need strtold_l")
179: (4)             def test_loadtxt(self):
180: (8)               with temppath() as path:
181: (12)                 with open(path, 'w') as f:
182: (16)                   f.write(self.out)
183: (12)                 res = np.loadtxt(path, dtype=np.longdouble)
184: (8)                 assert_equal(res, self.tgt)
185: (4)             @pytest.mark.skipif(string_to_longdouble_inaccurate,
186: (24)                           reason="Need strtold_l")
187: (4)             def test_tofile_roundtrip(self):
188: (8)               with temppath() as path:
189: (12)                 self.tgt.tofile(path, sep=" ")
190: (12)                 res = np.fromfile(path, dtype=np.longdouble, sep=" ")
191: (8)                 assert_equal(res, self.tgt)
192: (0)             def test_repr_exact():
193: (4)               o = 1 + LD_INFO.eps
194: (4)               assert_(repr(o) != '1')
195: (0)             @pytest.mark.skipif(longdouble_longer_than_double, reason="BUG #2376")
196: (0)             @pytest.mark.skipif(string_to_longdouble_inaccurate,
197: (20)                           reason="Need strtold_l")
198: (0)             def test_format():
199: (4)               o = 1 + LD_INFO.eps
200: (4)               assert_("{0:.40g}".format(o) != '1')
201: (0)             @pytest.mark.skipif(longdouble_longer_than_double, reason="BUG #2376")
202: (0)             @pytest.mark.skipif(string_to_longdouble_inaccurate,
203: (20)                           reason="Need strtold_l")
204: (0)             def test_percent():
205: (4)               o = 1 + LD_INFO.eps

```

```

206: (4)           assert_(("%.40g" % o != '1'))
207: (0)           @pytest.mark.skipif(longdouble_longer_than_double,
208: (20)                  reason="array repr problem")
209: (0)           @pytest.mark.skipif(string_to_longdouble_inaccurate,
210: (20)                  reason="Need strtold_l")
211: (0)           def test_array_repr():
212: (4)             o = 1 + LD_INFO.eps
213: (4)             a = np.array([o])
214: (4)             b = np.array([1], dtype=np.longdouble)
215: (4)             if not np.all(a != b):
216: (8)               raise ValueError("precision loss creating arrays")
217: (4)             assert_(repr(a) != repr(b))
218: (0)           class TestCommaDecimalPointLocale(CommaDecimalPointLocale):
219: (4)             def test_repr_roundtrip_foreign(self):
220: (8)               o = 1.5
221: (8)               assert_equal(o, np.longdouble(repr(o)))
222: (4)             def test_fromstring_foreign_repr(self):
223: (8)               f = 1.234
224: (8)               a = np.fromstring(repr(f), dtype=float, sep=" ")
225: (8)               assert_equal(a[0], f)
226: (4)             def test_fromstring_best_effort_float(self):
227: (8)               with assert_warns(DeprecationWarning):
228: (12)                 assert_equal(np.fromstring("1,234", dtype=float, sep=" "),
229: (25)                           np.array([1.]))
230: (4)             def test_fromstring_best_effort(self):
231: (8)               with assert_warns(DeprecationWarning):
232: (12)                 assert_equal(np.fromstring("1,234", dtype=np.longdouble, sep=" "),
233: (25)                           np.array([1.]))
234: (4)             def test_fromstring_foreign(self):
235: (8)               s = "1.234"
236: (8)               a = np.fromstring(s, dtype=np.longdouble, sep=" ")
237: (8)               assert_equal(a[0], np.longdouble(s))
238: (4)             def test_fromstring_foreign_sep(self):
239: (8)               a = np.array([1, 2, 3, 4])
240: (8)               b = np.fromstring("1,2,3,4,", dtype=np.longdouble, sep=",")
241: (8)               assert_array_equal(a, b)
242: (4)             def test_fromstring_foreign_value(self):
243: (8)               with assert_warns(DeprecationWarning):
244: (12)                 b = np.fromstring("1,234", dtype=np.longdouble, sep=" ")
245: (12)                 assert_array_equal(b[0], 1)
246: (0)           @pytest.mark.parametrize("int_val", [
247: (4)             2 ** 1024, 0])
248: (0)           def test_longdouble_from_int(int_val):
249: (4)             str_val = str(int_val)
250: (4)             with warnings.catch_warnings(record=True) as w:
251: (8)               warnings.filterwarnings('always', '', RuntimeWarning)
252: (8)               assert np.longdouble(int_val) == np.longdouble(str_val)
253: (8)               if np.allclose(np.finfo(np.longdouble).max,
254: (23)                     np.finfo(np.double).max) and w:
255: (12)                 assert w[0].category is RuntimeWarning
256: (0)           @pytest.mark.parametrize("bool_val", [
257: (4)             True, False])
258: (0)           def test_longdouble_from_bool(bool_val):
259: (4)             assert np.longdouble(bool_val) == np.longdouble(int(bool_val))
260: (0)           @pytest.mark.skipif(
261: (4)             not (IS_MUSL and platform.machine() == "x86_64"),
262: (4)             reason="only need to run on musllinux_x86_64"
263: (0)         )
264: (0)           def test_musllinux_x86_64_signature():
265: (4)             known_sigs = [b'\xcd\xcc\xcc\xcc\xcc\xcc\xcc\xfb\xbf']
266: (4)             sig = (np.longdouble(-1.0) / np.longdouble(10.0)
267: (11)               ).newbyteorder('<').tobytes()[:10]
268: (4)             assert sig in known_sigs
269: (0)           def test_eps_positive():
270: (4)             assert np.finfo(np.longdouble).eps > 0.

```

-----  
File 104 - test\_machar.py:

```

1: (0)
2: (0) """
3: (0) Test machar. Given recent changes to hardcode type data, we might want to get
4: (0) rid of both MachAr and this test at some point.
5: (0) """
6: (0) from numpy.core._machar import MachAr
7: (0) import numpy.core.numerictypes as ntypes
8: (0) from numpy import errstate, array
9: (0) class TestMachAr:
10: (4)     def _run_machar_highprec(self):
11: (8)         try:
12: (12)             hiprec = ntypes.float96
13: (12)             MachAr(lambda v: array(v, hiprec))
14: (8)         except AttributeError:
15: (12)             "Skipping test: no ntypes.float96 available on this platform."
16: (4)     def test_underflow(self):
17: (8)         with errstate(all='raise'):
18: (12)             try:
19: (16)                 self._run_machar_highprec()
20: (12)             except FloatingPointError as e:
21: (16)                 msg = "Caught %s exception, should not have been raised." % e
22: (16)                 raise AssertionError(msg)

-----

```

File 105 - test\_memmap.py:

```

1: (0)     import sys
2: (0)     import os
3: (0)     import mmap
4: (0)     import pytest
5: (0)     from pathlib import Path
6: (0)     from tempfile import NamedTemporaryFile, TemporaryFile
7: (0)     from numpy import (
8: (4)         memmap, sum, average, prod, ndarray, isscalar, add, subtract, multiply)
9: (0)     from numpy import arange, allclose, asarray
10: (0)    from numpy.testing import (
11: (4)        assert_, assert_equal, assert_array_equal, suppress_warnings, IS_PYPY,
12: (4)        break_cycles
13: (4)    )
14: (0)    class TestMemmap:
15: (4)        def setup_method(self):
16: (8)            self.tmpfp = NamedTemporaryFile(prefix='mmap')
17: (8)            self.shape = (3, 4)
18: (8)            self.dtype = 'float32'
19: (8)            self.data = arange(12, dtype=self.dtype)
20: (8)            self.data.resize(self.shape)
21: (4)        def teardown_method(self):
22: (8)            self.tmpfp.close()
23: (8)            self.data = None
24: (8)            if IS_PYPY:
25: (12)                break_cycles()
26: (12)                break_cycles()
27: (4)        def test_roundtrip(self):
28: (8)            fp = memmap(self.tmpfp, dtype=self.dtype, mode='w+',
29: (20)                            shape=self.shape)
30: (8)            fp[:] = self.data[:]
31: (8)            del fp # Test __del__ machinery, which handles cleanup
32: (8)            newfp = memmap(self.tmpfp, dtype=self.dtype, mode='r',
33: (23)                            shape=self.shape)
34: (8)            assert_(allclose(self.data, newfp))
35: (8)            assert_array_equal(self.data, newfp)
36: (8)            assert_equal(newfp.flags.writeable, False)
37: (4)        def test_open_with_filename(self, tmp_path):
38: (8)            tmpname = tmp_path / 'mmap'
39: (8)            fp = memmap(tmpname, dtype=self.dtype, mode='w+',
40: (23)                            shape=self.shape)
41: (8)            fp[:] = self.data[:]
42: (8)            del fp

```

```

43: (4)
44: (8)
45: (12)
46: (12)
47: (4)
48: (8)
49: (8)
50: (8)
51: (20)
52: (8)
53: (8)
54: (8)
55: (4)
56: (8)
57: (8)
58: (23)
59: (8)
60: (8)
61: (8)
62: (8)
63: (8)
64: (8)
65: (8)
66: (4)
67: (8)
68: (8)
69: (23)
70: (8)
71: (8)
72: (8)
73: (8)
74: (8)
75: (8)
76: (8)
77: (4)
78: (8)
79: (20)
80: (8)
81: (4)
82: (24)
83: (4)
84: (8)
85: (20)
86: (8)
87: (8)
88: (8)
89: (4)
90: (8)
91: (20)
92: (8)
93: (8)
94: (8)
95: (8)
96: (8)
97: (8)
98: (8)
99: (4)
100: (8)
101: (20)
102: (8)
103: (8)
104: (12)
105: (4)
106: (8)
107: (20)
108: (8)
109: (8)
110: (12)
111: (4)

        def test_unnamed_file(self):
            with TemporaryFile() as f:
                fp = memmap(f, dtype=self.dtype, shape=self.shape)
                del fp
        def test_attributes(self):
            offset = 1
            mode = "w+"
            fp = memmap(self.tmpfp, dtype=self.dtype, mode=mode,
                        shape=self.shape, offset=offset)
            assert_equal(offset, fp.offset)
            assert_equal(mode, fp.mode)
            del fp
        def test_filename(self, tmp_path):
            tmpname = tmp_path / "mmap"
            fp = memmap(tmpname, dtype=self.dtype, mode='w+', 
                        shape=self.shape)
            abspath = Path(os.path.abspath(tmpname))
            fp[:] = self.data[:]
            assert_equal(abspath, fp.filename)
            b = fp[:1]
            assert_equal(abspath, b.filename)
            del b
            del fp
        def test_path(self, tmp_path):
            tmpname = tmp_path / "mmap"
            fp = memmap(Path(tmpname), dtype=self.dtype, mode='w+', 
                        shape=self.shape)
            abspath = str(Path(tmpname).resolve())
            fp[:] = self.data[:]
            assert_equal(abspath, str(fp.filename.resolve()))
            b = fp[:1]
            assert_equal(abspath, str(b.filename.resolve()))
            del b
            del fp
        def test_filename_fileobj(self):
            fp = memmap(self.tmpfp, dtype=self.dtype, mode="w+", 
                        shape=self.shape)
            assert_equal(fp.filename, self.tmpfp.name)
            @pytest.mark.skipif(sys.platform == 'gnu0',
                               reason="Known to fail on hurd")
        def test_flush(self):
            fp = memmap(self.tmpfp, dtype=self.dtype, mode='w+', 
                        shape=self.shape)
            fp[:] = self.data[:]
            assert_equal(fp[0], self.data[0])
            fp.flush()
        def test_del(self):
            fp_base = memmap(self.tmpfp, dtype=self.dtype, mode='w+', 
                              shape=self.shape)
            fp_base[0] = 5
            fp_view = fp_base[0:1]
            assert_equal(fp_view[0], 5)
            del fp_view
            assert_equal(fp_base[0], 5)
            fp_base[0] = 6
            assert_equal(fp_base[0], 6)
        def test_arithmetic_drops_references(self):
            fp = memmap(self.tmpfp, dtype=self.dtype, mode='w+', 
                        shape=self.shape)
            tmp = (fp + 10)
            if isinstance(tmp, memmap):
                assert_(tmp._mmap is not fp._mmap)
        def test_indexing_drops_references(self):
            fp = memmap(self.tmpfp, dtype=self.dtype, mode='w+', 
                        shape=self.shape)
            tmp = fp[(1, 2), (2, 3)]
            if isinstance(tmp, memmap):
                assert_(tmp._mmap is not fp._mmap)
        def test_slicing_keeps_references(self):

```

```

112: (8)                     fp = memmap(self.tmpfp, dtype=self.dtype, mode='w+',  

113: (20)                       shape=self.shape)  

114: (8)                     assert_(fp[:2, :2]._mmap is fp._mmap)  

115: (4) def test_view(self):  

116: (8)                     fp = memmap(self.tmpfp, dtype=self.dtype, shape=self.shape)  

117: (8)                     new1 = fp.view()  

118: (8)                     new2 = new1.view()  

119: (8)                     assert_(new1.base is fp)  

120: (8)                     assert_(new2.base is fp)  

121: (8)                     new_array = asarray(fp)  

122: (8)                     assert_(new_array.base is fp)  

123: (4) def test_ufunc_return_ndarray(self):  

124: (8)                     fp = memmap(self.tmpfp, dtype=self.dtype, shape=self.shape)  

125: (8)                     fp[:] = self.data  

126: (8)                     with suppress_warnings() as sup:  

127: (12)                       sup.filter(FutureWarning, "np.average currently does not  

preserve")  

128: (12)                         for unary_op in [sum, average, prod]:  

129: (16)                           result = unary_op(fp)  

130: (16)                           assert_(isscalar(result))  

131: (16)                           assert_(result.__class__ is self.data[0, 0].__class__)  

132: (16)                           assert_(unary_op(fp, axis=0).__class__ is ndarray)  

133: (16)                           assert_(unary_op(fp, axis=1).__class__ is ndarray)  

134: (8)                         for binary_op in [add, subtract, multiply]:  

135: (12)                           assert_(binary_op(fp, self.data).__class__ is ndarray)  

136: (12)                           assert_(binary_op(self.data, fp).__class__ is ndarray)  

137: (12)                           assert_(binary_op(fp, fp).__class__ is ndarray)  

138: (8)                         fp += 1  

139: (8)                         assert(fp.__class__ is memmap)  

140: (8)                         add(fp, 1, out=fp)  

141: (8)                         assert(fp.__class__ is memmap)  

142: (4) def test_getitem(self):  

143: (8)                     fp = memmap(self.tmpfp, dtype=self.dtype, shape=self.shape)  

144: (8)                     fp[:] = self.data  

145: (8)                     assert_(fp[1:, :-1].__class__ is memmap)  

146: (8)                     assert_(fp[[0, 1]].__class__ is ndarray)  

147: (4) def test_memmap_subclass(self):  

148: (8)                     class MemmapSubClass(memmap):  

149: (12)                       pass  

150: (8)                     fp = MemmapSubClass(self.tmpfp, dtype=self.dtype, shape=self.shape)  

151: (8)                     fp[:] = self.data  

152: (8)                     assert_(sum(fp, axis=0).__class__ is MemmapSubClass)  

153: (8)                     assert_(sum(fp).__class__ is MemmapSubClass)  

154: (8)                     assert_(fp[1:, :-1].__class__ is MemmapSubClass)  

155: (8)                     assert_(fp[[0, 1]].__class__ is MemmapSubClass)  

156: (4) def test_mmap_offset_greater_than_allocation_granularity(self):  

157: (8)                     size = 5 * mmap.ALLOCATIONGRANULARITY  

158: (8)                     offset = mmap.ALLOCATIONGRANULARITY + 1  

159: (8)                     fp = memmap(self.tmpfp, shape=size, mode='w+', offset=offset)  

160: (8)                     assert_(fp.offset == offset)  

161: (4) def test_no_shape(self):  

162: (8)                     self.tmpfp.write(b'a'*16)  

163: (8)                     mm = memmap(self.tmpfp, dtype='float64')  

164: (8)                     assert_equal(mm.shape, (2,))  

165: (4) def test_empty_array(self):  

166: (8)                     with pytest.raises(ValueError, match='empty file'):  

167: (12)                       memmap(self.tmpfp, shape=(0,4), mode='w+')  

168: (8)                       self.tmpfp.write(b'\0')  

169: (8)                       memmap(self.tmpfp, shape=(0,4), mode='w+')
-----
```

## File 106 - test\_mem\_overlap.py:

```

1: (0)                     import itertools
2: (0)                     import pytest
3: (0)                     import numpy as np
4: (0)                     from numpy.core._multiarray_tests import solve_diophantine, internal_overlap
5: (0)                     from numpy.core import _umath_tests

```

```

6: (0)
7: (0)
8: (4)
9: (4)
10: (0)
11: (0)
12: (0)
13: (0)
14: (0)
15: (0)
16: (4)
sign."""
17: (4)
18: (8)
19: (4)
20: (4)
21: (8)
22: (12)
23: (12)
24: (12)
25: (4)
26: (0)
27: (4)
28: (4)
29: (4)
30: (8)
31: (8)
"nelems"
32: (4)
33: (0)
34: (4)
pairs."""
35: (4)
36: (4)
37: (0)
38: (4)
39: (4)
40: (4)
41: (4)
42: (4)
43: (4)
44: (12)
45: (0)
46: (4)
47: (4)
48: (8)
49: (8)
50: (8)
51: (0)
52: (0)
53: (4)
54: (4)
55: (4)
56: (8)
57: (8)
58: (8)
59: (8)
60: (12)
61: (12)
62: (12)
63: (12)
64: (12)
65: (22)
66: (12)
67: (22)
68: (12)
69: (12)
70: (12)
71: (16)

from numpy.lib.stride_tricks import as_strided
from numpy.testing import (
    assert_, assert_raises, assert_equal, assert_array_equal
)
ndims = 2
size = 10
shape = tuple([size] * ndims)
MAY_SHARE_BOUNDS = 0
MAY_SHARE_EXACT = -1
def _indices_for_nelems(nelems):
    """Returns slices of length nelems, from start onwards, in direction
    sign."""
    if nelems == 0:
        return [size // 2] # int index
    res = []
    for step in (1, 2):
        for sign in (-1, 1):
            start = size // 2 - nelems * step * sign // 2
            stop = start + nelems * step * sign
            res.append(slice(start, stop, step * sign))
    return res
def _indices_for_axis():
    """Returns (src, dst) pairs of indices."""
    res = []
    for nelems in (0, 2, 3):
        ind = _indices_for_nelems(nelems)
        res.extend(itertools.product(ind, ind)) # all assignments of size
    return res
def _indices(ndims):
    """Returns ((axis0_src, axis0_dst), (axis1_src, axis1_dst), ... ) index
    pairs."""
    ind = _indices_for_axis()
    return itertools.product(ind, repeat=ndims)
def _check_assignment(srcidx, dstidx):
    """Check assignment arr[dstidx] = arr[srcidx] works."""
    arr = np.arange(np.prod(shape)).reshape(shape)
    cpy = arr.copy()
    cpy[dstidx] = arr[srcidx]
    arr[dstidx] = arr[srcidx]
    assert_(np.all(arr == cpy),
            'assigning arr[%s] = arr[%s]' % (dstidx, srcidx))
def test_overlapping_assignments():
    inds = _indices(ndims)
    for ind in inds:
        srcidx = tuple([a[0] for a in ind])
        dstidx = tuple([a[1] for a in ind])
        _check_assignment(srcidx, dstidx)
@pytest.mark.slow
def test_diophantine_fuzz():
    rng = np.random.RandomState(1234)
    max_int = np.iinfo(np.intp).max
    for ndim in range(10):
        feasible_count = 0
        infeasible_count = 0
        min_count = 500//(ndim + 1)
        while min(feasible_count, infeasible_count) < min_count:
            A_max = 1 + rng.randint(0, 11, dtype=np.intp)**6
            U_max = rng.randint(0, 11, dtype=np.intp)**6
            A_max = min(max_int, A_max)
            U_max = min(max_int-1, U_max)
            A = tuple(int(rng.randint(1, A_max+1, dtype=np.intp))
                      for j in range(ndim))
            U = tuple(int(rng.randint(0, U_max+2, dtype=np.intp))
                      for j in range(ndim))
            b_ub = min(max_int-2, sum(a*ub for a, ub in zip(A, U)))
            b = int(rng.randint(-1, b_ub+2, dtype=np.intp))
            if ndim == 0 and feasible_count < min_count:
                b = 0

```

```

72: (12)             X = solve_diophantine(A, U, b)
73: (12)             if X is None:
74: (16)                 X_simplified = solve_diophantine(A, U, b, simplify=1)
75: (16)                 assert_(X_simplified is None, (A, U, b, X_simplified))
76: (16)                 ranges = tuple(range(0, a*ub+1, a) for a, ub in zip(A, U))
77: (16)                 size = 1
78: (16)                 for r in ranges:
79: (20)                     size *= len(r)
80: (16)                 if size < 100000:
81: (20)                     assert_(not any(sum(w) == b for w in
82: (20)                         itertools.product(*ranges)))
82: (20)                     infeasible_count += 1
83: (12)             else:
84: (16)                 X_simplified = solve_diophantine(A, U, b, simplify=1)
85: (16)                 assert_(X_simplified is not None, (A, U, b, X_simplified))
86: (16)                 assert_(sum(a*x for a, x in zip(A, X)) == b)
87: (16)                 assert_(all(0 <= x <= ub for x, ub in zip(X, U)))
88: (16)                 feasible_count += 1
89: (0)             def test_diophantine_overflow():
90: (4)                 max_intp = np.iinfo(np.intp).max
91: (4)                 max_int64 = np.iinfo(np.int64).max
92: (4)                 if max_int64 <= max_intp:
93: (8)                     A = (max_int64//2, max_int64//2 - 10)
94: (8)                     U = (max_int64//2, max_int64//2 - 10)
95: (8)                     b = 2*(max_int64//2) - 10
96: (8)                     assert_equal(solve_diophantine(A, U, b), (1, 1))
97: (0)             def check_may_share_memory_exact(a, b):
98: (4)                 got = np.may_share_memory(a, b, max_work=MAY_SHARE_EXACT)
99: (4)                 assert_equal(np.may_share_memory(a, b),
100: (17)                               np.may_share_memory(a, b, max_work=MAY_SHARE_BOUNDS))
101: (4)                 a.fill(0)
102: (4)                 b.fill(0)
103: (4)                 a.fill(1)
104: (4)                 exact = b.any()
105: (4)                 err_msg = ""
106: (4)                 if got != exact:
107: (8)                     err_msg = "    " + "\n    ".join([
108: (12)                         "base_a - base_b = %r" % (a.__array_interface__['data'][0] -
b.__array_interface__['data'][0],),
109: (12)                         "shape_a = %r" % (a.shape,),
110: (12)                         "shape_b = %r" % (b.shape,),
111: (12)                         "strides_a = %r" % (a.strides,),
112: (12)                         "strides_b = %r" % (b.strides,),
113: (12)                         "size_a = %r" % (a.size,),
114: (12)                         "size_b = %r" % (b.size,))
115: (8)                     ])
116: (4)                     assert_equal(got, exact, err_msg=err_msg)
117: (0)             def test_may_share_memory_manual():
118: (4)                 xs0 = [
119: (8)                     np.zeros([13, 21, 23, 22], dtype=np.int8),
120: (8)                     np.zeros([13, 21, 23*2, 22], dtype=np.int8)[::,:,:,::2,:]
121: (4)                 ]
122: (4)                 xs = []
123: (4)                 for x in xs0:
124: (8)                     for ss in itertools.product(*(([slice(None)], slice(None, None,
-1]),)*4)):
125: (12)                         xp = x[ss]
126: (12)                         xs.append(xp)
127: (4)                 for x in xs:
128: (8)                     assert_(np.may_share_memory(x[:,0,:], x[:,1,:]))
129: (8)                     assert_(np.may_share_memory(x[:,0,:], x[:,1,:], max_work=None))
130: (8)                     check_may_share_memory_exact(x[:,0,:], x[:,1,:])
131: (8)                     check_may_share_memory_exact(x[:,::7], x[:,3::3])
132: (8)                     try:
133: (12)                         xp = x.ravel()
134: (12)                         if xp.flags.owndata:
135: (16)                             continue
136: (12)                         xp = xp.view(np.int16)
137: (8)                     except ValueError:

```

```

138: (12)           continue
139: (8)            check_may_share_memory_exact(x.ravel()[6:6],
140: (37)           xp.reshape(13, 21, 23, 11)[::,:7])
141: (8)            check_may_share_memory_exact(x[:,::7],
142: (37)           xp.reshape(13, 21, 23, 11))
143: (8)            check_may_share_memory_exact(x[:,::7],
144: (37)           xp.reshape(13, 21, 23, 11)[::,3::3])
145: (8)            check_may_share_memory_exact(x.ravel()[6:7],
146: (37)           xp.reshape(13, 21, 23, 11)[::,:7])
147: (4)             x = np.zeros([1], dtype=np.int8)
148: (4)             check_may_share_memory_exact(x, x)
149: (4)             check_may_share_memory_exact(x, x.copy())
150: (0)             def iter_random_view_pairs(x, same_steps=True, equal_size=False):
151: (4)               rng = np.random.RandomState(1234)
152: (4)               if equal_size and same_steps:
153: (8)                 raise ValueError()
154: (4)               def random_slice(n, step):
155: (8)                 start = rng.randint(0, n+1, dtype=np.intp)
156: (8)                 stop = rng.randint(start, n+1, dtype=np.intp)
157: (8)                 if rng.randint(0, 2, dtype=np.intp) == 0:
158: (12)                   stop, start = start, stop
159: (12)                   step *= -1
160: (8)                   return slice(start, stop, step)
161: (4)               def random_slice_fixed_size(n, step, size):
162: (8)                 start = rng.randint(0, n+1 - size*step)
163: (8)                 stop = start + (size-1)*step + 1
164: (8)                 if rng.randint(0, 2) == 0:
165: (12)                   stop, start = start-1, stop-1
166: (12)                   if stop < 0:
167: (16)                     stop = None
168: (12)                     step *= -1
169: (8)                     return slice(start, stop, step)
170: (4)             yield x, x
171: (4)             for j in range(1, 7, 3):
172: (8)               yield x[j:], x[:-j]
173: (8)               yield x[...,:j], x[...,:-j]
174: (4)             strides = list(x.strides)
175: (4)             strides[0] = 0
176: (4)             xp = as_strided(x, shape=x.shape, strides=strides)
177: (4)             yield x, xp
178: (4)             yield xp, xp
179: (4)             strides = list(x.strides)
180: (4)             if strides[0] > 1:
181: (8)               strides[0] = 1
182: (4)             xp = as_strided(x, shape=x.shape, strides=strides)
183: (4)             yield x, xp
184: (4)             yield xp, xp
185: (4)             while True:
186: (8)               steps = tuple(rng.randint(1, 11, dtype=np.intp)
187: (22)                 if rng.randint(0, 5, dtype=np.intp) == 0 else 1
188: (22)                 for j in range(x.ndim))
189: (8)               s1 = tuple(random_slice(p, s) for p, s in zip(x.shape, steps))
190: (8)               t1 = np.arange(x.ndim)
191: (8)               rng.shuffle(t1)
192: (8)               if equal_size:
193: (12)                 t2 = t1
194: (8)               else:
195: (12)                 t2 = np.arange(x.ndim)
196: (12)                 rng.shuffle(t2)
197: (8)               a = x[s1]
198: (8)               if equal_size:
199: (12)                 if a.size == 0:
200: (16)                   continue
201: (12)                 steps2 = tuple(rng.randint(1, max(2, p//(1+pa)))
202: (27)                   if rng.randint(0, 5) == 0 else 1
203: (27)                   for p, pa in zip(x.shape, s1, a.shape))
204: (12)                 s2 = tuple(random_slice_fixed_size(p, s, pa)
205: (23)                   for p, s, pa in zip(x.shape, steps2, a.shape))
206: (8)               elif same_steps:
```

```

207: (12)           steps2 = steps
208: (8)            else:
209: (12)              steps2 = tuple(rng.randint(1, 11, dtype=np.intp)
210: (27)                  if rng.randint(0, 5, dtype=np.intp) == 0 else 1
211: (27)                  for j in range(x.ndim))
212: (8)            if not equal_size:
213: (12)              s2 = tuple(random_slice(p, s) for p, s in zip(x.shape, steps2))
214: (8)              a = a.transpose(t1)
215: (8)              b = x[s2].transpose(t2)
216: (8)              yield a, b
217: (0)            def check_may_share_memory_easy_fuzz(get_max_work, same_steps, min_count):
218: (4)              x = np.zeros([17, 34, 71, 97], dtype=np.int16)
219: (4)              feasible = 0
220: (4)              infeasible = 0
221: (4)              pair_iter = iter_random_view_pairs(x, same_steps)
222: (4)              while min(feasible, infeasible) < min_count:
223: (8)                  a, b = next(pair_iter)
224: (8)                  bounds_overlap = np.may_share_memory(a, b)
225: (8)                  may_share_answer = np.may_share_memory(a, b)
226: (8)                  easy_answer = np.may_share_memory(a, b, max_work=get_max_work(a, b))
227: (8)                  exact_answer = np.may_share_memory(a, b, max_work=MAY_SHARE_EXACT)
228: (8)                  if easy_answer != exact_answer:
229: (12)                      assert_equal(easy_answer, exact_answer)
230: (8)                  if may_share_answer != bounds_overlap:
231: (12)                      assert_equal(may_share_answer, bounds_overlap)
232: (8)                  if bounds_overlap:
233: (12)                      if exact_answer:
234: (16)                          feasible += 1
235: (12)                      else:
236: (16)                          infeasible += 1
237: (0)            @pytest.mark.slow
238: (0)            def test_may_share_memory_easy_fuzz():
239: (4)                check_may_share_memory_easy_fuzz(get_max_work=lambda a, b: 1,
240: (37)                                same_steps=True,
241: (37)                                min_count=2000)
242: (0)            @pytest.mark.slow
243: (0)            def test_may_share_memory_harder_fuzz():
244: (4)                check_may_share_memory_easy_fuzz(get_max_work=lambda a, b: max(a.size,
b.size)//2,
245: (37)
246: (37)
247: (0)            def test_shares_memory_api():
248: (4)                x = np.zeros([4, 5, 6], dtype=np.int8)
249: (4)                assert_equal(np.shares_memory(x, x), True)
250: (4)                assert_equal(np.shares_memory(x, x.copy()), False)
251: (4)                a = x[:, ::2, ::3]
252: (4)                b = x[:, ::3, ::2]
253: (4)                assert_equal(np.shares_memory(a, b), True)
254: (4)                assert_equal(np.shares_memory(a, b, max_work=None), True)
255: (4)                assert_raises(np.TooHardError, np.shares_memory, a, b, max_work=1)
256: (0)            def test_may_share_memory_bad_max_work():
257: (4)                x = np.zeros([1])
258: (4)                assert_raises(OverflowError, np.may_share_memory, x, x, max_work=10**100)
259: (4)                assert_raises(OverflowError, np.shares_memory, x, x, max_work=10**100)
260: (0)            def test_internal_overlap_diophantine():
261: (4)                def check(A, U, exists=None):
262: (8)                    X = solve_diophantine(A, U, 0, require_ub_nontrivial=1)
263: (8)                    if exists is None:
264: (12)                        exists = (X is not None)
265: (8)                    if X is not None:
266: (12)                        assert_(sum(a*x for a, x in zip(A, X)) == sum(a*u//2 for a, u in
zip(A, U)))
267: (12)                        assert_(all(0 <= x <= u for x, u in zip(X, U)))
268: (12)                        assert_(any(x != u//2 for x, u in zip(X, U)))
269: (8)                    if exists:
270: (12)                        assert_(X is not None, repr(X))
271: (8)                    else:
272: (12)                        assert_(X is None, repr(X))
273: (4)            check((3, 2), (2*2, 3*2), exists=True)

```

```

274: (4)           check((3*2, 2), (15*2, (3-1)*2), exists=False)
275: (0)           def test_internal_overlap_slices():
276: (4)             x = np.zeros([17,34,71,97], dtype=np.int16)
277: (4)             rng = np.random.RandomState(1234)
278: (4)             def random_slice(n, step):
279: (8)               start = rng.randint(0, n+1, dtype=np.intp)
280: (8)               stop = rng.randint(start, n+1, dtype=np.intp)
281: (8)               if rng.randint(0, 2, dtype=np.intp) == 0:
282: (12)                 stop, start = start, stop
283: (12)                 step *= -1
284: (8)               return slice(start, stop, step)
285: (4)             cases = 0
286: (4)             min_count = 5000
287: (4)             while cases < min_count:
288: (8)               steps = tuple(rng.randint(1, 11, dtype=np.intp)
289: (22)                     if rng.randint(0, 5, dtype=np.intp) == 0 else 1
290: (22)                         for j in range(x.ndim))
291: (8)               t1 = np.arange(x.ndim)
292: (8)               rng.shuffle(t1)
293: (8)               s1 = tuple(random_slice(p, s) for p, s in zip(x.shape, steps))
294: (8)               a = x[s1].transpose(t1)
295: (8)               assert_(not internal_overlap(a))
296: (8)               cases += 1
297: (0)           def check_internal_overlap(a, manual_expected=None):
298: (4)             got = internal_overlap(a)
299: (4)             m = set()
300: (4)             ranges = tuple(range(n) for n in a.shape)
301: (4)             for v in itertools.product(*ranges):
302: (8)               offset = sum(s*w for s, w in zip(a.strides, v))
303: (8)               if offset in m:
304: (12)                 expected = True
305: (12)                 break
306: (8)               else:
307: (12)                 m.add(offset)
308: (4)             else:
309: (8)               expected = False
310: (4)             if got != expected:
311: (8)               assert_equal(got, expected, err_msg=repr((a.strides, a.shape)))
312: (4)             if manual_expected is not None and expected != manual_expected:
313: (8)               assert_equal(expected, manual_expected)
314: (4)             return got
315: (0)           def test_internal_overlap_manual():
316: (4)             x = np.arange(1).astype(np.int8)
317: (4)             check_internal_overlap(x, False) # 1-dim
318: (4)             check_internal_overlap(x.reshape([]), False) # 0-dim
319: (4)             a = as_strided(x, strides=(3, 4), shape=(4, 4))
320: (4)             check_internal_overlap(a, False)
321: (4)             a = as_strided(x, strides=(3, 4), shape=(5, 4))
322: (4)             check_internal_overlap(a, True)
323: (4)             a = as_strided(x, strides=(0,), shape=(0,))
324: (4)             check_internal_overlap(a, False)
325: (4)             a = as_strided(x, strides=(0,), shape=(1,))
326: (4)             check_internal_overlap(a, False)
327: (4)             a = as_strided(x, strides=(0,), shape=(2,))
328: (4)             check_internal_overlap(a, True)
329: (4)             a = as_strided(x, strides=(0, -9993), shape=(87, 22))
330: (4)             check_internal_overlap(a, True)
331: (4)             a = as_strided(x, strides=(0, -9993), shape=(1, 22))
332: (4)             check_internal_overlap(a, False)
333: (4)             a = as_strided(x, strides=(0, -9993), shape=(0, 22))
334: (4)             check_internal_overlap(a, False)
335: (0)           def test_internal_overlap_fuzz():
336: (4)             x = np.arange(1).astype(np.int8)
337: (4)             overlap = 0
338: (4)             no_overlap = 0
339: (4)             min_count = 100
340: (4)             rng = np.random.RandomState(1234)
341: (4)             while min(overlap, no_overlap) < min_count:
342: (8)               ndim = rng.randint(1, 4, dtype=np.intp)

```

```

343: (8)
344: (24)
345: (8)
346: (22)
347: (8)
348: (8)
349: (8)
350: (12)
351: (8)
352: (12)
353: (0)
354: (4)
355: (8)
356: (12)
357: (8)
358: (8)
359: (12)
360: (4)
361: (8)
362: (12)
363: (8)
364: (12)
365: (4)
366: (8)
367: (8)
368: (8)
369: (8)
370: (8)
371: (0)
372: (4)
373: (4)
374: (4)
375: (4)
376: (4)
377: (4)
378: (0)
379: (4)
380: (4)
381: (4)
382: (4)
383: (4)
384: (4)
385: (4)
386: (4)
387: (4)
388: (4)
389: (4)
390: (4)
391: (4)
392: (8)
393: (0)
394: (4)
395: (4)
396: (4)
397: (4)
398: (29)
399: (8)
400: (8)
401: (8)
402: (12)
403: (12)
404: (12)
405: (12)
406: (12)
407: (16)
408: (16)
409: (16)
410: (16)
411: (20)

        strides = tuple(rng.randint(-1000, 1000, dtype=np.intp)
                         for j in range(ndim))
        shape = tuple(rng.randint(1, 30, dtype=np.intp)
                         for j in range(ndim))
        a = as_strided(x, strides=strides, shape=shape)
        result = check_internal_overlap(a)
        if result:
            overlap += 1
        else:
            no_overlap += 1
    def test_non_ndarray_inputs():
        class MyArray:
            def __init__(self, data):
                self.data = data
            @property
            def __array_interface__(self):
                return self.data.__array_interface__
        class MyArray2:
            def __init__(self, data):
                self.data = data
            def __array__(self):
                return self.data
        for cls in [MyArray, MyArray2]:
            x = np.arange(5)
            assert_(np.may_share_memory(cls(x[::2]), x[1::2]))
            assert_(not np.shares_memory(cls(x[::2]), x[1::2]))
            assert_(np.shares_memory(cls(x[1::3]), x[::2]))
            assert_(np.may_share_memory(cls(x[1::3]), x[::2]))
    def view_element_first_byte(x):
        """Construct an array viewing the first byte of each element of `x`"""
        from numpy.lib.stride_tricks import DummyArray
        interface = dict(x.__array_interface__)
        interface['typestr'] = '|b1'
        interface['descr'] = [((' ', '|b1'))]
        return np.asarray(DummyArray(interface, x))
    def assert_copy_equivalent(operation, args, out, **kwargs):
        """
        Check that operation(*args, out=out) produces results
        equivalent to out[...] = operation(*args, out=out.copy())
        """
        kwargs['out'] = out
        kwargs2 = dict(kwargs)
        kwargs2['out'] = out.copy()
        out_orig = out.copy()
        out[...] = operation(*args, **kwargs2)
        expected = out.copy()
        out[...] = out_orig
        got = operation(*args, **kwargs).copy()
        if (got != expected).any():
            assert_equal(got, expected)
    class TestUFunc:
        """
        Test ufunc call memory overlap handling
        """
        def check_unary_fuzz(self, operation, get_out_axis_size, dtype=np.int16,
                             count=5000):
            shapes = [7, 13, 8, 21, 29, 32]
            rng = np.random.RandomState(1234)
            for ndim in range(1, 6):
                x = rng.randint(0, 2**16, size=shapes[:ndim]).astype(dtype)
                it = iter_random_view_pairs(x, same_steps=False, equal_size=True)
                min_count = count // (ndim + 1)**2
                overlapping = 0
                while overlapping < min_count:
                    a, b = next(it)
                    a_orig = a.copy()
                    b_orig = b.copy()
                    if get_out_axis_size is None:
                        assert_copy_equivalent(operation, [a], out=b)

```

```

412: (20)                     if np.shares_memory(a, b):
413: (24)                         overlapping += 1
414: (16)
415: (20)             else:
416: (24)                 for axis in itertools.chain(range(ndim), [None]):
417: (24)                     a[...] = a_orig
418: (24)                     b[...] = b_orig
419: (24)                     outsize, scalarize = get_out_axis_size(a, b, axis)
420: (28)                     if outsize == 'skip':
421: (24)                         continue
422: (24)                     sl = [slice(None)] * ndim
423: (28)                     if axis is None:
424: (32)                         if outsize is None:
425: (28)                             sl = [slice(0, 1)] + [0]*(ndim - 1)
426: (32)                         else:
427: (24)                             sl = [slice(0, outsize)] + [0]*(ndim - 1)
428: (28)                     else:
429: (32)                         if outsize is None:
430: (32)                             k = b.shape[axis]//2
431: (36)                             if ndim == 1:
432: (32)                                 sl[axis] = slice(k, k + 1)
433: (36)                             else:
434: (28)                                 sl[axis] = k
435: (32)                         else:
436: (32)                             assert b.shape[axis] >= outsize
437: (24)                             sl[axis] = slice(0, outsize)
438: (24)                     b_out = b[tuple(sl)]
439: (28)                     if scalarize:
440: (24)                         b_out = b_out.reshape(())
441: (28)                     if np.shares_memory(a, b_out):
442: (24)                         overlapping += 1
axis=axis)
443: (4)                     assert_copy_equivalent(operation, [a], out=b_out,
444: (4) @pytest.mark.slow
445: (8)             def test_unary_ufunc_call_fuzz(self):
446: (4)                 self.check_unary_fuzz(np.invert, None, np.int16)
447: (4) @pytest.mark.slow
448: (8)             def test_unary_ufunc_call_complex_fuzz(self):
449: (4)                 self.check_unary_fuzz(np.negative, None, np.complex128, count=500)
450: (8)             def test_binary_ufunc_accumulate_fuzz(self):
451: (12)                 def get_out_axis_size(a, b, axis):
452: (16)                     if axis is None:
453: (20)                         if a.ndim == 1:
454: (16)                             return a.size, False
455: (20)                         else:
456: (12)                             return 'skip', False # accumulate doesn't support this
457: (16)                     else:
458: (8)                         return a.shape[axis], False
459: (30)                     self.check_unary_fuzz(np.add.accumulate, get_out_axis_size,
460: (4)                                     dtype=np.int16, count=500)
461: (8)             def test_binary_ufunc_reduce_fuzz(self):
462: (12)                 def get_out_axis_size(a, b, axis):
463: (8)                     return None, (axis is None or a.ndim == 1)
464: (30)                     self.check_unary_fuzz(np.add.reduce, get_out_axis_size,
465: (4)                                     dtype=np.int16, count=500)
466: (8)             def test_binary_ufunc_reduceat_fuzz(self):
467: (12)                 def get_out_axis_size(a, b, axis):
468: (16)                     if axis is None:
469: (20)                         if a.ndim == 1:
470: (16)                             return a.size, False
471: (20)                         else:
472: (12)                             return 'skip', False # reduceat doesn't support this
473: (16)                     else:
474: (8)                         return a.shape[axis], False
475: (12)             def do_reduceat(a, out, axis):
476: (16)                 if axis is None:
477: (16)                     size = len(a)
478: (12)                     step = size//len(out)
479: (16)                 else:

```

```

480: (16)           step = a.shape[axis] // out.shape[axis]
481: (12)           idx = np.arange(0, size, step)
482: (12)           return np.add.reduceat(a, idx, out=out, axis=axis)
483: (8)            self.check_unary_fuzz(do_reduceat, get_out_axis_size,
484:                               dtype=np.int16, count=500)
485: (4)            def test_binary_ufunc_reduceat_manual(self):
486: (8)              def check(ufunc, a, ind, out):
487: (12)                  c1 = ufunc.reduceat(a.copy(), ind.copy(), out=out.copy())
488: (12)                  c2 = ufunc.reduceat(a, ind, out=out)
489: (12)                  assert_array_equal(c1, c2)
490: (8)                  a = np.arange(10000, dtype=np.int16)
491: (8)                  check(np.add, a, a[::-1].copy(), a)
492: (8)                  a = np.arange(10000, dtype=np.int16)
493: (8)                  check(np.add, a, a[::-1], a)
494: (4)            @pytest.mark.slow
495: (4)            def test_unary_gufunc_fuzz(self):
496: (8)              shapes = [7, 13, 8, 21, 29, 32]
497: (8)              gufunc = _umath_tests.euclidean_pdist
498: (8)              rng = np.random.RandomState(1234)
499: (8)              for ndim in range(2, 6):
500: (12)                  x = rng.rand(*shapes[:ndim])
501: (12)                  it = iter_random_view_pairs(x, same_steps=False, equal_size=True)
502: (12)                  min_count = 500 // (ndim + 1)**2
503: (12)                  overlapping = 0
504: (12)                  while overlapping < min_count:
505: (16)                      a, b = next(it)
506: (16)                      if min(a.shape[-2:]) < 2 or min(b.shape[-2:]) < 2 or
a.shape[-1] < 2:
507: (20)                          continue
508: (16)                      if b.shape[-1] > b.shape[-2]:
509: (20)                          b = b[...,0,:]
510: (16)                      else:
511: (20)                          b = b[...,:,0]
512: (16)                      n = a.shape[-2]
513: (16)                      p = n * (n - 1) // 2
514: (16)                      if p <= b.shape[-1] and p > 0:
515: (20)                          b = b[...,:p]
516: (16)                      else:
517: (20)                          n = max(2, int(np.sqrt(b.shape[-1]))//2)
518: (20)                          p = n * (n - 1) // 2
519: (20)                          a = a[..., :n, :]
520: (20)                          b = b[..., :p]
521: (16)                      if np.shares_memory(a, b):
522: (20)                          overlapping += 1
523: (16)                      with np.errstate(over='ignore', invalid='ignore'):
524: (20)                          assert_copy_equivalent(gufunc, [a], out=b)
525: (4)            def test_ufunc_at_manual(self):
526: (8)              def check(ufunc, a, ind, b=None):
527: (12)                  a0 = a.copy()
528: (12)                  if b is None:
529: (16)                      ufunc.at(a0, ind.copy())
530: (16)                      c1 = a0.copy()
531: (16)                      ufunc.at(a, ind)
532: (16)                      c2 = a.copy()
533: (12)                  else:
534: (16)                      ufunc.at(a0, ind.copy(), b.copy())
535: (16)                      c1 = a0.copy()
536: (16)                      ufunc.at(a, ind, b)
537: (16)                      c2 = a.copy()
538: (12)                      assert_array_equal(c1, c2)
539: (8)                      a = np.arange(10000, dtype=np.int16)
540: (8)                      check(np.invert, a[::-1], a)
541: (8)                      a = np.arange(100, dtype=np.int16)
542: (8)                      ind = np.arange(0, 100, 2, dtype=np.int16)
543: (8)                      check(np.add, a, ind, a[25:75])
544: (4)            def test_unary_ufunc_1d_manual(self):
545: (8)              def check(a, b):
546: (12)                  a_orig = a.copy()
547: (12)                  b_orig = b.copy()

```

```

548: (12)          b0 = b.copy()
549: (12)          c1 = ufunc(a, out=b0)
550: (12)          c2 = ufunc(a, out=b)
551: (12)          assert_array_equal(c1, c2)
552: (12)          mask = view_element_first_byte(b).view(np.bool_)
553: (12)          a[...] = a_orig
554: (12)          b[...] = b_orig
555: (12)          c1 = ufunc(a, out=b.copy(), where=mask.copy()).copy()
556: (12)          a[...] = a_orig
557: (12)          b[...] = b_orig
558: (12)          c2 = ufunc(a, out=b, where=mask.copy()).copy()
559: (12)          a[...] = a_orig
560: (12)          b[...] = b_orig
561: (12)          c3 = ufunc(a, out=b, where=mask).copy()
562: (12)          assert_array_equal(c1, c2)
563: (12)          assert_array_equal(c1, c3)
564: (8)           dtypes = [np.int8, np.int16, np.int32, np.int64, np.float32,
565: (18)             np.float64, np.complex64, np.complex128]
566: (8)           dtypes = [np.dtype(x) for x in dtypes]
567: (8)           for dtype in dtypes:
568: (12)             if np.issubdtype(dtype, np.integer):
569: (16)               ufunc = np.invert
570: (12)             else:
571: (16)               ufunc = np.reciprocal
572: (12)             n = 1000
573: (12)             k = 10
574: (12)             indices = [
575: (16)               np.index_exp[:n],
576: (16)               np.index_exp[k:k+n],
577: (16)               np.index_exp[n-1::-1],
578: (16)               np.index_exp[k+n-1:k-1:-1],
579: (16)               np.index_exp[:2*n:2],
580: (16)               np.index_exp[k:k+2*n:2],
581: (16)               np.index_exp[2*n-1::-2],
582: (16)               np.index_exp[k+2*n-1:k-1:-2],
583: (12)             ]
584: (12)             for xi, yi in itertools.product(indices, indices):
585: (16)               v = np.arange(1, 1 + n*2 + k, dtype=dtype)
586: (16)               x = v[xi]
587: (16)               y = v[yi]
588: (16)               with np.errstate(all='ignore'):
589: (20)                 check(x, y)
590: (20)                 check(x[:1], y)
591: (20)                 check(x[-1:], y)
592: (20)                 check(x[:1].reshape([]), y)
593: (20)                 check(x[-1:].reshape([]), y)
594: (4)              def test_unary_ufunc_where_same(self):
595: (8)                ufunc = np.invert
596: (8)                def check(a, out, mask):
597: (12)                  c1 = ufunc(a, out=out.copy(), where=mask.copy())
598: (12)                  c2 = ufunc(a, out=out, where=mask)
599: (12)                  assert_array_equal(c1, c2)
600: (8)                  x = np.arange(100).astype(np.bool_)
601: (8)                  check(x, x, x)
602: (8)                  check(x, x.copy(), x)
603: (8)                  check(x, x, x.copy())
604: (4)                  @pytest.mark.slow
605: (4)                  def test_binary_ufunc_1d_manual(self):
606: (8)                    ufunc = np.add
607: (8)                    def check(a, b, c):
608: (12)                      c0 = c.copy()
609: (12)                      c1 = ufunc(a, b, out=c0)
610: (12)                      c2 = ufunc(a, b, out=c)
611: (12)                      assert_array_equal(c1, c2)
612: (8)                      for dtype in [np.int8, np.int16, np.int32, np.int64,
613: (22)                        np.float32, np.float64, np.complex64, np.complex128]:
614: (12)                        n = 1000
615: (12)                        k = 10
616: (12)                        indices = []

```

```

617: (12)          for p in [1, 2]:
618: (16)            indices.extend([
619: (20)              np.index_exp[:p*n:p],
620: (20)              np.index_exp[k:k+p*n:p],
621: (20)              np.index_exp[p*n-1::-p],
622: (20)              np.index_exp[k+p*n-1:k-1:-p],
623: (16)            ])
624: (12)          for x, y, z in itertools.product(indices, indices, indices):
625: (16)            v = np.arange(6*n).astype(dtype)
626: (16)            x = v[x]
627: (16)            y = v[y]
628: (16)            z = v[z]
629: (16)            check(x, y, z)
630: (16)            check(x[:1], y, z)
631: (16)            check(x[-1:], y, z)
632: (16)            check(x[:1].reshape([]), y, z)
633: (16)            check(x[-1:].reshape([]), y, z)
634: (16)            check(x, y[:1], z)
635: (16)            check(x, y[-1:], z)
636: (16)            check(x, y[:1].reshape([]), z)
637: (16)            check(x, y[-1:].reshape([]), z)
638: (4)          def test_inplace_op_simple_manual(self):
639: (8)            rng = np.random.RandomState(1234)
640: (8)            x = rng.rand(200, 200) # bigger than bufsize
641: (8)            x += x.T
642: (8)            assert_array_equal(x - x.T, 0)

-----

```

## File 107 - test\_mem\_policy.py:

```

1: (0)          import asyncio
2: (0)          import gc
3: (0)          import os
4: (0)          import pytest
5: (0)          import numpy as np
6: (0)          import threading
7: (0)          import warnings
8: (0)          from numpy.testing import extbuild, assert_warnings, IS_WASM
9: (0)          import sys
10: (0)          @pytest.fixture
11: (0)          def get_module(tmp_path):
12: (4)            """ Add a memory policy that returns a false pointer 64 bytes into the
13: (4)            actual allocation, and fill the prefix with some text. Then check at each
14: (4)            memory manipulation that the prefix exists, to make sure all
alloc/realloc/
15: (4)            free/calloc go via the functions here.
16: (4)            """
17: (4)            if sys.platform.startswith('cygwin'):
18: (8)              pytest.skip('link fails on cygwin')
19: (4)            if IS_WASM:
20: (8)              pytest.skip("Can't build module inside Wasm")
21: (4)            functions = [
22: (8)              ("get_default_policy", "METH_NOARGS", """
23: (13)                  Py_INCREF(PyDataMem_DefaultHandler);
24: (13)                  return PyDataMem_DefaultHandler;
25: (9)                  """),
26: (8)              ("set_secret_data_policy", "METH_NOARGS", """
27: (13)                  PyObject *secret_data =
28: (17)                      PyCapsule_New(&secret_data_handler, "mem_handler", NULL);
29: (13)                  if (secret_data == NULL) {
30: (17)                      return NULL;
31: }
32: (13)                  PyObject *old = PyDataMem_SetHandler(secret_data);
33: (13)                  Py_DECREF(secret_data);
34: (13)                  return old;
35: (9)                  """),
36: (8)              ("set_old_policy", "METH_O", """
37: (13)                  PyObject *old;

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

38: (13)             if (args != NULL && PyCapsule_CheckExact(args)) {
39: (17)                 old = PyDataMem_SetHandler(args);
40: (13)             }
41: (13)             else {
42: (17)                 old = PyDataMem_SetHandler(NULL);
43: (13)             }
44: (13)             return old;
45: (9)         """
46: (8)         ("get_array", "METH_NOARGS", """
47: (12)             char *buf = (char *)malloc(20);
48: (12)             npy_intp dims[1];
49: (12)             dims[0] = 20;
50: (12)             PyArray_Descr *descr = PyArray_DescrNewFromType(NPY_UINT8);
51: (12)             return PyArray_NewFromDescr(&PyArray_Type, descr, 1, dims, NULL,
52: (40)                             buf, NPY_ARRAY_WRITEABLE, NULL);
53: (9)         """
54: (8)         ("set_own", "METH_O", """
55: (12)             if (!PyArray_Check(args)) {
56: (16)                 PyErr_SetString(PyExc_ValueError,
57: (29)                     "need an ndarray");
58: (16)             return NULL;
59: (12)         }
60: (12)         PyArray_ENABLEFLAGS((PyArrayObject*)args, NPY_ARRAY_OWNDATA);
61: (12)         // Maybe try this too?
62: (12)         // PyArray_BASE(PyArrayObject *)args) = NULL;
63: (12)         Py_RETURN_NONE;
64: (9)         """
65: (8)         ("get_array_with_base", "METH_NOARGS", """
66: (12)             char *buf = (char *)malloc(20);
67: (12)             npy_intp dims[1];
68: (12)             dims[0] = 20;
69: (12)             PyArray_Descr *descr = PyArray_DescrNewFromType(NPY_UINT8);
70: (12)             PyObject *arr = PyArray_NewFromDescr(&PyArray_Type, descr, 1,
dims,
71: (49)                             NULL, buf,
72: (49)                             NPY_ARRAY_WRITEABLE, NULL);
73: (12)             if (arr == NULL) return NULL;
74: (12)             PyObject *obj = PyCapsule_New(buf, "buf capsule",
75: (42)
(PyCapsule_Destructor)&warn_on_free);
76: (12)             if (obj == NULL) {
77: (16)                 Py_DECREF(arr);
78: (16)             return NULL;
79: (12)
80: (12)             if (PyArray_SetBaseObject((PyArrayObject *)arr, obj) < 0) {
81: (16)                 Py_DECREF(arr);
82: (16)                 Py_DECREF(obj);
83: (16)                 return NULL;
84: (12)
85: (12)             return arr;
86: (9)         """
87: (4)
88: (4)     ]
prologue = '''
89: (8)
90: (9)     * This struct allows the dynamic configuration of the allocator funcs
91: (9)     * of the `secret_data_allocator`. It is provided here for
92: (9)     * demonstration purposes, as a valid `ctx` use-case scenario.
93: (9)
94: (8)     typedef struct {
95: (12)         void *(*malloc)(size_t);
96: (12)         void *(*calloc)(size_t, size_t);
97: (12)         void *(*realloc)(void *, size_t);
98: (12)         void (*free)(void *);
99: (8)     } SecretDataAllocatorFuncs;
100: (8)     NPY_NO_EXPORT void *
101: (8)     shift_alloc(void *ctx, size_t sz) {
102: (12)         SecretDataAllocatorFuncs *funcs = (SecretDataAllocatorFuncs *)ctx;
103: (12)         char *real = (char *)funcs->malloc(sz + 64);
104: (12)         if (real == NULL) {

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

105: (16)                                return NULL;
106: (12)
107: (12)
108: (12)
109: (8)
110: (8)
111: (8)
112: (12)
113: (12)
114: (12)
115: (16)
116: (12)
117: (12)
118: (21)
119: (12)
120: (8)
121: (8)
122: (8)
123: (12)
124: (12)
125: (16)
126: (12)
127: (12)
128: (12)
129: (16)
130: (24)
131: (16)
*/
132: (16)
133: (16)
134: (12)
135: (12)
136: (16)
137: (16)
138: (20)
139: (28)
140: (20)
141: (20)
142: (16)
143: (16)
144: (20)
145: (16)
146: (12)
147: (8)
148: (8)
149: (8)
150: (12)
151: (12)
152: (16)
153: (16)
154: (20)
155: (20)
156: (16)
157: (16)
158: (12)
159: (12)
160: (16)
161: (16)
162: (20)
163: (16)
164: (16)
165: (25)
166: (16)
167: (12)
168: (8)
169: (8)
170: (8)
171: (12)
172: (12)

                    return NULL;
                }
                sprintf(real, 64, "originally allocated %ld", (unsigned long)sz);
                return (void *)(real + 64);
            }
        NPY_NO_EXPORT void *
        shift_zero(void *ctx, size_t sz, size_t cnt) {
            SecretDataAllocatorFuncs *funcs = (SecretDataAllocatorFuncs *)ctx;
            char *real = (char *)funcs->calloc(sz + 64, cnt);
            if (real == NULL) {
                return NULL;
            }
            sprintf(real, 64, "originally allocated %ld via zero",
                    (unsigned long)sz);
            return (void *)(real + 64);
        }
        NPY_NO_EXPORT void
        shift_free(void *ctx, void * p, npy_uintp sz) {
            SecretDataAllocatorFuncs *funcs = (SecretDataAllocatorFuncs *)ctx;
            if (p == NULL) {
                return ;
            }
            char *real = (char *)p - 64;
            if (strncmp(real, "originally allocated", 20) != 0) {
                fprintf(stdout, "uh-oh, unmatched shift_free, "
                        "no appropriate prefix\\n");
                /* Make C runtime crash by calling free on the wrong address
*/
                funcs->free((char *)p + 10);
                /* funcs->free(real); */
            }
            else {
                npy_uintp i = (npy_uintp)atoi(real +20);
                if (i != sz) {
                    fprintf(stderr, "uh-oh, unmatched shift_free"
                            "(ptr, %ld) but allocated %ld\\n", sz, i);
                    /* This happens in some places, only print */
                    funcs->free(real);
                }
                else {
                    funcs->free(real);
                }
            }
        }
        NPY_NO_EXPORT void *
        shift_realloc(void *ctx, void * p, npy_uintp sz) {
            SecretDataAllocatorFuncs *funcs = (SecretDataAllocatorFuncs *)ctx;
            if (p != NULL) {
                char *real = (char *)p - 64;
                if (strncmp(real, "originally allocated", 20) != 0) {
                    fprintf(stdout, "uh-oh, unmatched shift_realloc\\n");
                    return realloc(p, sz);
                }
                return (void *)((char *)funcs->realloc(real, sz + 64) + 64);
            }
            else {
                char *real = (char *)funcs->realloc(p, sz + 64);
                if (real == NULL) {
                    return NULL;
                }
                sprintf(real, 64, "originally allocated "
                        "%ld via realloc", (unsigned long)sz);
                return (void *)(real + 64);
            }
        }
/* As an example, we use the standard {m|c|re}alloc/free funcs. */
static SecretDataAllocatorFuncs secret_data_handler_ctx = {
    malloc,
    calloc,

```

```

173: (12)           realloc,
174: (12)           free
175: (8)        };
176: (8)        static PyDataMem_Handler secret_data_handler = {
177: (12)          "secret_data_allocator",
178: (12)          1,
179: (12)          {
180: (16)            &secret_data_handler_ctx, /* ctx */
181: (16)            shift_alloc,           /* malloc */
182: (16)            shift_zero,            /* calloc */
183: (16)            shift_realloc,         /* realloc */
184: (16)            shift_free             /* free */
185: (12)          }
186: (8)        };
187: (8)        void warn_on_free(void *capsule) {
188: (12)          PyErr_WarnEx(PyExc_UserWarning, "in warn_on_free", 1);
189: (12)          void * obj = PyCapsule_GetPointer(capsule,
190: (46)                                PyCapsule_GetName(capsule));
191: (12)          free(obj);
192: (8)        };
193: (8)        ...
194: (4)        more_init = "import_array();"
195: (4)        try:
196: (8)          import mem_policy
197: (8)          return mem_policy
198: (4)        except ImportError:
199: (8)          pass
200: (4)        return extbuild.build_and_import_extension('mem_policy',
201: (47)                      functions,
202: (47)                      prologue=prologue,
203: (47)                      include_dirs=
[np.get_include()],
204: (47)                      build_dir=tmp_path,
205: (47)                      more_init=more_init)
206: (0)        @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
207: (0)        def test_set_policy(get_module):
208: (4)          get_handler_name = np.core.multiarray.get_handler_name
209: (4)          get_handler_version = np.core.multiarray.get_handler_version
210: (4)          orig_policy_name = get_handler_name()
211: (4)          a = np.arange(10).reshape((2, 5)) # a doesn't own its own data
212: (4)          assert get_handler_name(a) is None
213: (4)          assert get_handler_version(a) is None
214: (4)          assert get_handler_name(a.base) == orig_policy_name
215: (4)          assert get_handler_version(a.base) == 1
216: (4)          orig_policy = get_module.set_secret_data_policy()
217: (4)          b = np.arange(10).reshape((2, 5)) # b doesn't own its own data
218: (4)          assert get_handler_name(b) is None
219: (4)          assert get_handler_version(b) is None
220: (4)          assert get_handler_name(b.base) == 'secret_data_allocator'
221: (4)          assert get_handler_version(b.base) == 1
222: (4)          if orig_policy_name == 'default_allocator':
223: (8)            get_module.set_old_policy(None) # tests PyDataMem_SetHandler(NULL)
224: (8)            assert get_handler_name() == 'default_allocator'
225: (4)          else:
226: (8)            get_module.set_old_policy(orig_policy)
227: (8)            assert get_handler_name() == orig_policy_name
228: (0)        @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
229: (0)        def test_default_policy_singleton(get_module):
230: (4)          get_handler_name = np.core.multiarray.get_handler_name
231: (4)          orig_policy = get_module.set_old_policy(None)
232: (4)          assert get_handler_name() == 'default_allocator'
233: (4)          def_policy_1 = get_module.set_old_policy(None)
234: (4)          assert get_handler_name() == 'default_allocator'
235: (4)          def_policy_2 = get_module.set_old_policy(orig_policy)
236: (4)          assert def_policy_1 is def_policy_2 is get_module.get_default_policy()
237: (0)        @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
238: (0)        def test_policy_propagation(get_module):
239: (4)          class MyArr(np.ndarray):
240: (8)            pass

```

```

241: (4)             get_handler_name = np.core.multiarray.get_handler_name
242: (4)             orig_policy_name = get_handler_name()
243: (4)             a = np.arange(10).view(MyArr).reshape((2, 5))
244: (4)             assert get_handler_name(a) is None
245: (4)             assert a.flags.owndata is False
246: (4)             assert get_handler_name(a.base) is None
247: (4)             assert a.base.flags.owndata is False
248: (4)             assert get_handler_name(a.base.base) == orig_policy_name
249: (4)             assert a.base.base.flags.owndata is True
250: (0)             async def concurrent_context1(get_module, orig_policy_name, event):
251: (4)                 if orig_policy_name == 'default_allocator':
252: (8)                     get_module.set_secret_data_policy()
253: (8)                     assert np.core.multiarray.get_handler_name() ==
254: (4)                         'secret_data_allocator'
255: (8)                     else:
256: (8)                         get_module.set_old_policy(None)
257: (4)                         assert np.core.multiarray.get_handler_name() == 'default_allocator'
258: (0)                         event.set()
259: (4)             async def concurrent_context2(get_module, orig_policy_name, event):
260: (4)                 await event.wait()
261: (4)                 assert np.core.multiarray.get_handler_name() == orig_policy_name
262: (8)                 if orig_policy_name == 'default_allocator':
263: (8)                     get_module.set_secret_data_policy()
264: (4)                     assert np.core.multiarray.get_handler_name() ==
265: (8)                         'secret_data_allocator'
266: (8)                     else:
267: (0)                         get_module.set_old_policy(None)
268: (4)                         assert np.core.multiarray.get_handler_name() == 'default_allocator'
269: (4)             async def async_test_context_locality(get_module):
270: (4)                 orig_policy_name = np.core.multiarray.get_handler_name()
271: (4)                 event = asyncio.Event()
272: (4)                 concurrent_task1 = asyncio.create_task(
273: (8)                     concurrent_context1(get_module, orig_policy_name, event))
274: (4)                 concurrent_task2 = asyncio.create_task(
275: (8)                     concurrent_context2(get_module, orig_policy_name, event))
276: (4)                 await concurrent_task1
277: (4)                 await concurrent_task2
278: (4)                 assert np.core.multiarray.get_handler_name() == orig_policy_name
279: (0)             @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
280: (4)             def test_context_locality(get_module):
281: (12)                 if (sys.implementation.name == 'pypy'
282: (8)                     and sys.pypy_version_info[:3] < (7, 3, 6)):
283: (8)                     pytest.skip('no context-locality support in PyPy < 7.3.6')
284: (4)                 asyncio.run(async_test_context_locality(get_module))
285: (4)             def concurrent_thread1(get_module, event):
286: (4)                 get_module.set_secret_data_policy()
287: (4)                 assert np.core.multiarray.get_handler_name() == 'secret_data_allocator'
288: (4)                 event.set()
289: (4)             def concurrent_thread2(get_module, event):
290: (4)                 event.wait()
291: (4)                 assert np.core.multiarray.get_handler_name() == 'default_allocator'
292: (4)                 get_module.set_secret_data_policy()
293: (0)             @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
294: (0)             def test_thread_locality(get_module):
295: (4)                 orig_policy_name = np.core.multiarray.get_handler_name()
296: (4)                 event = threading.Event()
297: (4)                 concurrent_task1 = threading.Thread(target=concurrent_thread1,
298: (40)                               args=(get_module, event))
299: (4)                 concurrent_task2 = threading.Thread(target=concurrent_thread2,
300: (40)                               args=(get_module, event))
301: (4)                 concurrent_task1.start()
302: (4)                 concurrent_task2.start()
303: (4)                 concurrent_task1.join()
304: (4)                 concurrent_task2.join()
305: (4)                 assert np.core.multiarray.get_handler_name() == orig_policy_name
306: (0)             @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
307: (0)             @pytest.mark.skip(reason="too slow, see gh-23975")
308: (0)             def test_new_policy(get_module):
309: (4)                 a = np.arange(10)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

308: (4)             orig_policy_name = np.core.multiarray.get_handler_name(a)
309: (4)             orig_policy = get_module.set_secret_data_policy()
310: (4)             b = np.arange(10)
311: (4)             assert np.core.multiarray.get_handler_name(b) == 'secret_data_allocator'
312: (4)             if orig_policy_name == 'default_allocator':
313: (8)                 assert np.core.test('full', verbose=1, extra_argv=[])
314: (8)                 assert np.ma.test('full', verbose=1, extra_argv=[])
315: (4)             get_module.set_old_policy(orig_policy)
316: (4)             c = np.arange(10)
317: (4)             assert np.core.multiarray.get_handler_name(c) == orig_policy_name
318: (0) @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
319: (0) @pytest.mark.xfail(sys.implementation.name == "pypy",
320: (19)                     reason=("bad interaction between getenv and "
321: (27)                         "os.environ inside pytest"))
322: (0) @pytest.mark.parametrize("policy", ["0", "1", None])
323: (0) def test_switch_owner(get_module, policy):
324: (4)     a = get_module.get_array()
325: (4)     assert np.core.multiarray.get_handler_name(a) is None
326: (4)     get_module.set_own(a)
327: (4)     if policy is None:
328: (8)         policy = os.getenv("NUMPY_WARN_IF_NO_MEM_POLICY", "0") == "1"
329: (8)         oldval = None
330: (4)     else:
331: (8)         policy = policy == "1"
332: (8)         oldval = np.core._multiarray_umath._set_numpy_warn_if_no_mem_policy(
333: (12)             policy)
334: (4)     try:
335: (8)         if policy:
336: (12)             with assert.warns(RuntimeWarning) as w:
337: (16)                 del a
338: (16)                 gc.collect()
339: (8)             else:
340: (12)                 del a
341: (12)                 gc.collect()
342: (4)             finally:
343: (8)                 if oldval is not None:
344: (12)                     np.core._multiarray_umath._set_numpy_warn_if_no_mem_policy(oldval)
345: (0) @pytest.mark.skipif(sys.version_info >= (3, 12), reason="no numpy.distutils")
346: (0) def test_owner_is_base(get_module):
347: (4)     a = get_module.get_array_with_base()
348: (4)     with pytest.warns(UserWarning, match='warn_on_free'):
349: (8)         del a
350: (8)         gc.collect()
351: (8)         gc.collect()

```

---

File 108 - test\_multiarray.py:

```

1: (0)             from __future__ import annotations
2: (0)             import collections.abc
3: (0)             import tempfile
4: (0)             import sys
5: (0)             import warnings
6: (0)             import operator
7: (0)             import io
8: (0)             import itertools
9: (0)             import functools
10: (0)            import ctypes
11: (0)            import os
12: (0)            import gc
13: (0)            import re
14: (0)            import weakref
15: (0)            import pytest
16: (0)            from contextlib import contextmanager
17: (0)            from numpy.compat import pickle
18: (0)            import pathlib
19: (0)            import builtins
20: (0)            from decimal import Decimal

```

```

21: (0) import mmap
22: (0) import numpy as np
23: (0) import numpy.core._multiarray_tests as _multiarray_tests
24: (0) from numpy.core._rational_tests import rational
25: (0) from numpy.testing import (
26: (4)     assert_, assert_raises, assert_warns, assert_equal, assert_almost_equal,
27: (4)     assert_array_equal, assert_raises_regex, assert_array_almost_equal,
28: (4)     assert_allclose, IS_PYPY, IS_PYTHON, HAS_REFCOUNT, assert_array_less,
29: (4)     runstring, temppath, suppress_warnings, break_cycles, _SUPPORTS_SVE,
30: (4) )
31: (0) from numpy.testing._private.utils import requires_memory, _no_tracing
32: (0) from numpy.core.tests._locales import CommaDecimalPointLocale
33: (0) from numpy.lib.recfunctions import repack_fields
34: (0) from numpy.core.multiarray import _get_ndarray_c_version
35: (0) from datetime import timedelta, datetime
36: (0) def assert_arg_sorted(arr, arg):
37: (4)     assert_equal(arr[arg], np.sort(arr))
38: (4)     assert_equal(np.sort(arg), np.arange(len(arg)))
39: (0) def _aligned_zeros(shape, dtype=float, order="C", align=None):
40: (4) """
41: (4)     Allocate a new ndarray with aligned memory.
42: (4)     The ndarray is guaranteed *not* aligned to twice the requested alignment.
43: (4)     Eg, if align=4, guarantees it is not aligned to 8. If align=None uses
44: (4)     dtype.alignment."""
45: (4)     dtype = np.dtype(dtype)
46: (4)     if dtype == np.dtype(object):
47: (8)         if align is not None:
48: (12)             raise ValueError("object array alignment not supported")
49: (8)         return np.zeros(shape, dtype=dtype, order=order)
50: (4)     if align is None:
51: (8)         align = dtype.alignment
52: (4)     if not hasattr(shape, '__len__'):
53: (8)         shape = (shape,)
54: (4)     size = functools.reduce(operator.mul, shape) * dtype.itemsize
55: (4)     buf = np.empty(size + 2*align + 1, np.uint8)
56: (4)     ptr = buf.__array_interface__['data'][0]
57: (4)     offset = ptr % align
58: (4)     if offset != 0:
59: (8)         offset = align - offset
60: (4)     if (ptr % (2*align)) == 0:
61: (8)         offset += align
62: (4)     buf = buf[offset:offset+size+1][:-1]
63: (4)     buf.fill(0)
64: (4)     data = np.ndarray(shape, dtype, buf, order=order)
65: (4)     return data
66: (0) class TestFlags:
67: (4)     def setup_method(self):
68: (8)         self.a = np.arange(10)
69: (4)     def test_writeable(self):
70: (8)         mydict = locals()
71: (8)         self.a.flags.writeable = False
72: (8)         assert_raises(ValueError, runstring, 'self.a[0] = 3', mydict)
73: (8)         assert_raises(ValueError, runstring, 'self.a[0:1].itemset(3)', mydict)
74: (8)         self.a.flags.writeable = True
75: (8)         self.a[0] = 5
76: (8)         self.a[0] = 0
77: (4)     def test_writeable_any_base(self):
78: (8)         arr = np.arange(10)
79: (8)         class subclass(np.ndarray):
80: (12)             pass
81: (8)             view1 = arr.view(subclass)
82: (8)             view2 = view1[...]
83: (8)             arr.flags.writeable = False
84: (8)             view2.flags.writeable = False
85: (8)             view2.flags.writeable = True # Can be set to True again.
86: (8)             arr = np.arange(10)
87: (8)             class frominterface:
88: (12)                 def __init__(self, arr):
89: (16)                     self.arr = arr

```

```

90: (16)                     self.__array_interface__ = arr.__array_interface__
91: (8)                      view1 = np.asarray(frominterface)
92: (8)                      view2 = view1[...]
93: (8)                      view2.flags.writeable = False
94: (8)                      view2.flags.writeable = True
95: (8)                      view1.flags.writeable = False
96: (8)                      view2.flags.writeable = False
97: (8)                      with assert_raises(ValueError):
98: (12)                         view2.flags.writeable = True
99: (4) def test_writeable_from_READONLY(self):
100: (8)   data = b'\x00' * 100
101: (8)   vals = np.frombuffer(data, 'B')
102: (8)   assert_raises(ValueError, vals.setflags, write=True)
103: (8)   types = np.dtype( [('vals', 'u1'), ('res3', 'S4')]) )
104: (8)   values = np.core.records.fromstring(data, types)
105: (8)   vals = values['vals']
106: (8)   assert_raises(ValueError, vals.setflags, write=True)
107: (4) def test_writeable_from_buffer(self):
108: (8)   data = bytearray(b'\x00' * 100)
109: (8)   vals = np.frombuffer(data, 'B')
110: (8)   assert_(vals.flags.writeable)
111: (8)   vals.setflags(write=False)
112: (8)   assert_(vals.flags.writeable is False)
113: (8)   vals.setflags(write=True)
114: (8)   assert_(vals.flags.writeable)
115: (8)   types = np.dtype( [('vals', 'u1'), ('res3', 'S4')]) )
116: (8)   values = np.core.records.fromstring(data, types)
117: (8)   vals = values['vals']
118: (8)   assert_(vals.flags.writeable)
119: (8)   vals.setflags(write=False)
120: (8)   assert_(vals.flags.writeable is False)
121: (8)   vals.setflags(write=True)
122: (8)   assert_(vals.flags.writeable)
123: (4) @pytest.mark.skipif(IS_PYPY, reason="PyPy always copies")
124: (4) def test_writeable_pickle(self):
125: (8)   import pickle
126: (8)   a = np.arange(1000)
127: (8)   for v in range(pickle.HIGHEST_PROTOCOL):
128: (12)     vals = pickle.loads(pickle.dumps(a, v))
129: (12)     assert_(vals.flags.writeable)
130: (12)     assert_(isinstance(vals.base, bytes))
131: (4) def test_writeable_from_c_data(self):
132: (8)   from numpy.core._multiarray_tests import get_c_wrapping_array
133: (8)   arr_writeable = get_c_wrapping_array(True)
134: (8)   assert not arr_writeable.flags.owndata
135: (8)   assert arr_writeable.flags.writeable
136: (8)   view = arr_writeable[...]
137: (8)   view.flags.writeable = False
138: (8)   assert not view.flags.writeable
139: (8)   view.flags.writeable = True
140: (8)   assert view.flags.writeable
141: (8)   arr_writeable.flags.writeable = False
142: (8)   arr_READONLY = get_c_wrapping_array(False)
143: (8)   assert not arr_READONLY.flags.owndata
144: (8)   assert not arr_READONLY.flags.writeable
145: (8)   for arr in [arr_writeable, arr_READONLY]:
146: (12)     view = arr[...]
147: (12)     view.flags.writeable = False # make sure it is readonly
148: (12)     arr.flags.writeable = False
149: (12)     assert not arr.flags.writeable
150: (12)     with assert_raises(ValueError):
151: (16)       view.flags.writeable = True
152: (12)     with warnings.catch_warnings():
153: (16)       warnings.simplefilter("error", DeprecationWarning)
154: (16)       with assert_raises(DeprecationWarning):
155: (20)         arr.flags.writeable = True
156: (12)       with assert_warns(DeprecationWarning):
157: (16)         arr.flags.writeable = True
158: (4) def test_warnonwrite(self):

```

```

159: (8)             a = np.arange(10)
160: (8)             a.flags._warn_on_write = True
161: (8)             with warnings.catch_warnings(record=True) as w:
162: (12)                 warnings.filterwarnings('always')
163: (12)                 a[1] = 10
164: (12)                 a[2] = 10
165: (12)                 assert_(len(w) == 1)
166: (4) @pytest.mark.parametrize(["flag", "flag_value", "writeable"],
167: (12)     [("writeable", True, True),
168: (13)         ("_warn_on_write", True, False),
169: (13)         ("writeable", False, False)])
170: (4) def test_READONLY_flag_protocols(self, flag, flag_value, writeable):
171: (8)     a = np.arange(10)
172: (8)     setattr(a.flags, flag, flag_value)
173: (8)     class MyArr():
174: (12)         __array_struct__ = a.__array_struct__
175: (8)         assert memoryview(a).readonly is not writeable
176: (8)         assert a.__array_interface__['data'][1] is not writeable
177: (8)         assert np.asarray(MyArr()).flags.writeable is writeable
178: (4) def test_otherflags(self):
179: (8)     assert_equal(self.a.flags.carray, True)
180: (8)     assert_equal(self.a.flags['C'], True)
181: (8)     assert_equal(self.a.flags.farray, False)
182: (8)     assert_equal(self.a.flags.behaved, True)
183: (8)     assert_equal(self.a.flags.fnc, False)
184: (8)     assert_equal(self.a.flags.forc, True)
185: (8)     assert_equal(self.a.flags.owndata, True)
186: (8)     assert_equal(self.a.flags.writeable, True)
187: (8)     assert_equal(self.a.flags.aligned, True)
188: (8)     assert_equal(self.a.flags.writebackifcopy, False)
189: (8)     assert_equal(self.a.flags['X'], False)
190: (8)     assert_equal(self.a.flags['WRITEBACKIFCOPY'], False)
191: (4) def test_string_align(self):
192: (8)     a = np.zeros(4, dtype=np.dtype('|S4'))
193: (8)     assert_(a.flags.aligned)
194: (8)     a = np.zeros(5, dtype=np.dtype('|S4'))
195: (8)     assert_(a.flags.aligned)
196: (4) def test_void_align(self):
197: (8)     a = np.zeros(4, dtype=np.dtype([('a", "i4"), ("b", "i4")]))
198: (8)     assert_(a.flags.aligned)
199: (0) class TestHash:
200: (4)     def test_int(self):
201: (8)         for st, ut, s in [(np.int8, np.uint8, 8),
202: (26)                         (np.int16, np.uint16, 16),
203: (26)                         (np.int32, np.uint32, 32),
204: (26)                         (np.int64, np.uint64, 64)]:
205: (12)             for i in range(1, s):
206: (16)                 assert_equal(hash(st(-2**i)), hash(-2**i),
207: (29)                     err_msg="%r: -2**%d" % (st, i))
208: (16)                 assert_equal(hash(st(2**((i - 1)))), hash(2**((i - 1))),
209: (29)                     err_msg="%r: 2**%d" % (st, i - 1))
210: (16)                 assert_equal(hash(st(2**i - 1)), hash(2**i - 1),
211: (29)                     err_msg="%r: 2**%d - 1" % (st, i))
212: (16)                 i = max(i - 1, 1)
213: (16)                 assert_equal(hash(ut(2**((i - 1)))), hash(2**((i - 1))),
214: (29)                     err_msg="%r: 2**%d" % (ut, i - 1))
215: (16)                 assert_equal(hash(ut(2**i - 1)), hash(2**i - 1),
216: (29)                     err_msg="%r: 2**%d - 1" % (ut, i))
217: (0) class TestAttributes:
218: (4)     def setup_method(self):
219: (8)         self.one = np.arange(10)
220: (8)         self.two = np.arange(20).reshape(4, 5)
221: (8)         self.three = np.arange(60, dtype=np.float64).reshape(2, 5, 6)
222: (4)     def test_attributes(self):
223: (8)         assert_equal(self.one.shape, (10,))
224: (8)         assert_equal(self.two.shape, (4, 5))
225: (8)         assert_equal(self.three.shape, (2, 5, 6))
226: (8)         self.three.shape = (10, 3, 2)
227: (8)         assert_equal(self.three.shape, (10, 3, 2))

```

```

228: (8)           self.three.shape = (2, 5, 6)
229: (8)           assert_equal(self.one.strides, (self.one.itemsize,))
230: (8)           num = self.two.itemsize
231: (8)           assert_equal(self.two.strides, (5*num, num))
232: (8)           num = self.three.itemsize
233: (8)           assert_equal(self.three.strides, (30*num, 6*num, num))
234: (8)           assert_equal(self.one.ndim, 1)
235: (8)           assert_equal(self.two.ndim, 2)
236: (8)           assert_equal(self.three.ndim, 3)
237: (8)           num = self.two.itemsize
238: (8)           assert_equal(self.two.size, 20)
239: (8)           assert_equal(self.two.nbytes, 20*num)
240: (8)           assert_equal(self.two.itemsize, self.two.dtype.itemsize)
241: (8)           assert_equal(self.two.base, np.arange(20))
242: (4) def test_dtypegetattr(self):
243: (8)     assert_equal(self.one.dtype, np.dtype(np.int_))
244: (8)     assert_equal(self.three.dtype, np.dtype(np.float_))
245: (8)     assert_equal(self.one.dtype.char, 'l')
246: (8)     assert_equal(self.three.dtype.char, 'd')
247: (8)     assert_(self.three.dtype.str[0] in '<>')
248: (8)     assert_equal(self.one.dtype.str[1], 'i')
249: (8)     assert_equal(self.three.dtype.str[1], 'f')
250: (4) def test_int_subclassing(self):
251: (8)     numpy_int = np.int_(0)
252: (8)     assert_(not isinstance(numpy_int, int))
253: (4) def test_stridesattr(self):
254: (8)     x = self.one
255: (8)     def make_array(size, offset, strides):
256: (12)         return np.ndarray(size, buffer=x, dtype=int,
257: (30)                 offset=offset*x.itemsize,
258: (30)                 strides=strides*x.itemsize)
259: (8)     assert_equal(make_array(4, 4, -1), np.array([4, 3, 2, 1]))
260: (8)     assert_raises(ValueError, make_array, 4, 4, -2)
261: (8)     assert_raises(ValueError, make_array, 4, 2, -1)
262: (8)     assert_raises(ValueError, make_array, 8, 3, 1)
263: (8)     assert_equal(make_array(8, 3, 0), np.array([3]*8))
264: (8)     assert_raises(ValueError, make_array, (2, 3), 5, np.array([-2, -3]))
265: (8)     make_array(0, 0, 10)
266: (4) def test_set_stridesattr(self):
267: (8)     x = self.one
268: (8)     def make_array(size, offset, strides):
269: (12)         try:
270: (16)             r = np.ndarray([size], dtype=int, buffer=x,
271: (31)                     offset=offset*x.itemsize)
272: (12)         except Exception as e:
273: (16)             raise RuntimeError(e)
274: (12)         r.strides = strides = strides*x.itemsize
275: (12)         return r
276: (8)     assert_equal(make_array(4, 4, -1), np.array([4, 3, 2, 1]))
277: (8)     assert_equal(make_array(7, 3, 1), np.array([3, 4, 5, 6, 7, 8, 9]))
278: (8)     assert_raises(ValueError, make_array, 4, 4, -2)
279: (8)     assert_raises(ValueError, make_array, 4, 2, -1)
280: (8)     assert_raises(RuntimeError, make_array, 8, 3, 1)
281: (8)     x = np.lib.stride_tricks.as_strided(np.arange(1), (10, 10), (0, 0))
282: (8)     def set_strides(arr, strides):
283: (12)         arr.strides = strides
284: (8)     assert_raises(ValueError, set_strides, x, (10*x.itemsize, x.itemsize))
285: (8)     x = np.lib.stride_tricks.as_strided(np.arange(10, dtype=np.int8)[-1],
286: (52)                     shape=(10,), strides=(-1,))
287: (8)     assert_raises(ValueError, set_strides, x[::-1], -1)
288: (8)     a = x[::-1]
289: (8)     a.strides = 1
290: (8)     a[::2].strides = 2
291: (8)     arr_0d = np.array(0)
292: (8)     arr_0d.strides = ()
293: (8)     assert_raises(TypeError, set_strides, arr_0d, None)
294: (4) def test_fill(self):
295: (8)     for t in "?bhilqpBHILQPfdgFDGO":

```

```

296: (12)           x = np.empty((3, 2, 1), t)
297: (12)           y = np.empty((3, 2, 1), t)
298: (12)           x.fill(1)
299: (12)           y[...] = 1
300: (12)           assert_equal(x, y)
301: (4) def test_fill_max_uint64(self):
302: (8)             x = np.empty((3, 2, 1), dtype=np.uint64)
303: (8)             y = np.empty((3, 2, 1), dtype=np.uint64)
304: (8)             value = 2**64 - 1
305: (8)             y[...] = value
306: (8)             x.fill(value)
307: (8)             assert_array_equal(x, y)
308: (4) def test_fill_struct_array(self):
309: (8)             x = np.array([(0, 0.0), (1, 1.0)], dtype='i4,f8')
310: (8)             x.fill(x[0])
311: (8)             assert_equal(x['f1'][1], x['f1'][0])
312: (8)             x = np.zeros(2, dtype=[('a', 'f8'), ('b', 'i4')])
313: (8)             x.fill((3.5, -2))
314: (8)             assert_array_equal(x['a'], [3.5, 3.5])
315: (8)             assert_array_equal(x['b'], [-2, -2])
316: (4) def test_fill_readonly(self):
317: (8)             a = np.zeros(11)
318: (8)             a.setflags(write=False)
319: (8)             with pytest.raises(ValueError, match=".*read-only"):
320: (12)                 a.fill(0)
321: (0) class TestArrayConstruction:
322: (4)     def test_array(self):
323: (8)         d = np.ones(6)
324: (8)         r = np.array([d, d])
325: (8)         assert_equal(r, np.ones((2, 6)))
326: (8)         d = np.ones(6)
327: (8)         tgt = np.ones((2, 6))
328: (8)         r = np.array([d, d])
329: (8)         assert_equal(r, tgt)
330: (8)         tgt[1] = 2
331: (8)         r = np.array([d, d + 1])
332: (8)         assert_equal(r, tgt)
333: (8)         d = np.ones(6)
334: (8)         r = np.array([[d, d]])
335: (8)         assert_equal(r, np.ones((1, 2, 6)))
336: (8)         d = np.ones(6)
337: (8)         r = np.array([[d, d], [d, d]])
338: (8)         assert_equal(r, np.ones((2, 2, 6)))
339: (8)         d = np.ones((6, 6))
340: (8)         r = np.array([d, d])
341: (8)         assert_equal(r, np.ones((2, 6, 6)))
342: (8)         d = np.ones((6, ))
343: (8)         r = np.array([[d, d + 1], d + 2], dtype=object)
344: (8)         assert_equal(len(r), 2)
345: (8)         assert_equal(r[0], [d, d + 1])
346: (8)         assert_equal(r[1], d + 2)
347: (8)         tgt = np.ones((2, 3), dtype=bool)
348: (8)         tgt[0, 2] = False
349: (8)         tgt[1, 0:2] = False
350: (8)         r = np.array([[True, True, False], [False, False, True]])
351: (8)         assert_equal(r, tgt)
352: (8)         r = np.array([[True, False], [True, False], [False, True]])
353: (8)         assert_equal(r, tgt.T)
354: (4) def test_array_empty(self):
355: (8)             assert_raises(TypeError, np.array)
356: (4) def test_0d_array_shape(self):
357: (8)             assert np.ones(np.array(3)).shape == (3,)
358: (4) def test_array_copy_false(self):
359: (8)             d = np.array([1, 2, 3])
360: (8)             e = np.array(d, copy=False)
361: (8)             d[1] = 3
362: (8)             assert_array_equal(e, [1, 3, 3])
363: (8)             e = np.array(d, copy=False, order='F')
364: (8)             d[1] = 4

```

```

365: (8)             assert_array_equal(e, [1, 4, 3])
366: (8)             e[2] = 7
367: (8)             assert_array_equal(d, [1, 4, 7])
368: (4)             def test_array_copy_true(self):
369: (8)                 d = np.array([[1,2,3], [1, 2, 3]])
370: (8)                 e = np.array(d, copy=True)
371: (8)                 d[0, 1] = 3
372: (8)                 e[0, 2] = -7
373: (8)                 assert_array_equal(e, [[1, 2, -7], [1, 2, 3]])
374: (8)                 assert_array_equal(d, [[1, 3, 3], [1, 2, 3]])
375: (8)                 e = np.array(d, copy=True, order='F')
376: (8)                 d[0, 1] = 5
377: (8)                 e[0, 2] = 7
378: (8)                 assert_array_equal(e, [[1, 3, 7], [1, 2, 3]])
379: (8)                 assert_array_equal(d, [[1, 5, 3], [1,2,3]])
380: (4)             def test_array_cont(self):
381: (8)                 d = np.ones(10)[::2]
382: (8)                 assert_(np.ascontiguousarray(d).flags.c_contiguous)
383: (8)                 assert_(np.ascontiguousarray(d).flags.f_contiguous)
384: (8)                 assert_(np.asfortranarray(d).flags.c_contiguous)
385: (8)                 assert_(np.asfortranarray(d).flags.f_contiguous)
386: (8)                 d = np.ones((10, 10))[::2, ::2]
387: (8)                 assert_(np.ascontiguousarray(d).flags.c_contiguous)
388: (8)                 assert_(np.asfortranarray(d).flags.f_contiguous)
389: (4)             @pytest.mark.parametrize("func",
390: (12)                 [np.array,
391: (13)                     np.asarray,
392: (13)                     np.asanyarray,
393: (13)                     np.ascontiguousarray,
394: (13)                     np.asfortranarray])
395: (4)             def test_bad_arguments_error(self, func):
396: (8)                 with pytest.raises(TypeError):
397: (12)                     func(3, dtype="bad dtype")
398: (8)                 with pytest.raises(TypeError):
399: (12)                     func() # missing arguments
400: (8)                 with pytest.raises(TypeError):
401: (12)                     func(1, 2, 3, 4, 5, 6, 7, 8) # too many arguments
402: (4)             @pytest.mark.parametrize("func",
403: (12)                 [np.array,
404: (13)                     np.asarray,
405: (13)                     np.asanyarray,
406: (13)                     np.ascontiguousarray,
407: (13)                     np.asfortranarray])
408: (4)             def test_array_as_keyword(self, func):
409: (8)                 if func is np.array:
410: (12)                     func(object=3)
411: (8)                 else:
412: (12)                     func(a=3)
413: (0)             class TestAssignment:
414: (4)                 def test_assignment_broadcasting(self):
415: (8)                     a = np.arange(6).reshape(2, 3)
416: (8)                     a[...] = np.arange(3)
417: (8)                     assert_equal(a, [[0, 1, 2], [0, 1, 2]])
418: (8)                     a[...] = np.arange(2).reshape(2, 1)
419: (8)                     assert_equal(a, [[0, 0, 0], [1, 1, 1]])
420: (8)                     a[...] = np.arange(6)[::-1].reshape(1, 2, 3)
421: (8)                     assert_equal(a, [[[5, 4, 3], [2, 1, 0]]])
422: (8)                     def assign(a, b):
423: (12)                         a[...] = b
424: (8)                         assert_raises(ValueError, assign, a, np.arange(12).reshape(2, 2, 3))
425: (4)             def test_assignment_errors(self):
426: (8)                 class C:
427: (12)                     pass
428: (8)                     a = np.zeros(1)
429: (8)                     def assign(v):
430: (12)                         a[0] = v
431: (8)                         assert_raises((AttributeError, TypeError), assign, C())
432: (8)                         assert_raises(ValueError, assign, [1])
433: (4)             def test_unicode_assignment(self):

```

```

434: (8)             from numpy.core.numeric import set_string_function
435: (8)             @contextmanager
436: (8)             def inject_str(s):
437: (12)                 """ replace ndarray.__str__ temporarily """
438: (12)                 set_string_function(lambda x: s, repr=False)
439: (12)                 try:
440: (16)                     yield
441: (12)                 finally:
442: (16)                     set_string_function(None, repr=False)
443: (8)             a1d = np.array(['test'])
444: (8)             a0d = np.array('done')
445: (8)             with inject_str('bad'):
446: (12)                 a1d[0] = a0d # previously this would invoke __str__
447: (8)             assert_equal(a1d[0], 'done')
448: (8)             np.array([np.array('\xe5\xe4\xf6')])
449: (4)             def test_stringlike_empty_list(self):
450: (8)                 u = np.array(['done'])
451: (8)                 b = np.array([b'done'])
452: (8)                 class bad_sequence:
453: (12)                     def __getitem__(self): pass
454: (12)                     def __len__(self): raise RuntimeError
455: (8)                     assert_raises(ValueError, operator.setitem, u, 0, [])
456: (8)                     assert_raises(ValueError, operator.setitem, b, 0, [])
457: (8)                     assert_raises(ValueError, operator.setitem, u, 0, bad_sequence())
458: (8)                     assert_raises(ValueError, operator.setitem, b, 0, bad_sequence())
459: (4)             def test_longdouble_assignment(self):
460: (8)                 for dtype in (np.longdouble, np.longcomplex):
461: (12)                     tinyb = np.nextafter(np.longdouble(0), 1).astype(dtype)
462: (12)                     tinya = np.nextafter(np.longdouble(0), -1).astype(dtype)
463: (12)                     tiny1d = np.array([tinya])
464: (12)                     assert_equal(tiny1d[0], tinya)
465: (12)                     tiny1d[0] = tinyb
466: (12)                     assert_equal(tiny1d[0], tinyb)
467: (12)                     tiny1d[0, ...] = tinya
468: (12)                     assert_equal(tiny1d[0], tinya)
469: (12)                     tiny1d[0, ...] = tinyb[...]
470: (12)                     assert_equal(tiny1d[0], tinyb)
471: (12)                     tiny1d[0] = tinyb[...]
472: (12)                     assert_equal(tiny1d[0], tinyb)
473: (12)                     arr = np.array([np.array(tinya)])
474: (12)                     assert_equal(arr[0], tinya)
475: (4)             def test_cast_to_string(self):
476: (8)                 a = np.zeros(1, dtype='S20')
477: (8)                 a[:] = np.array(['1.12345678901234567890'], dtype='f8')
478: (8)                 assert_equal(a[0], b"1.1234567890123457")
479: (0)             class TestDtypedescr:
480: (4)                 def test_construction(self):
481: (8)                     d1 = np.dtype('i4')
482: (8)                     assert_equal(d1, np.dtype(np.int32))
483: (8)                     d2 = np.dtype('f8')
484: (8)                     assert_equal(d2, np.dtype(np.float64))
485: (4)                 def test_byteorders(self):
486: (8)                     assert_(np.dtype('<i4') != np.dtype('>i4'))
487: (8)                     assert_(np.dtype([('a', '<i4')]) != np.dtype([('a', '>i4')]))
488: (4)                 def test_structured_non_void(self):
489: (8)                     fields = [('a', '<i2'), ('b', '<i2')]
490: (8)                     dt_int = np.dtype(('i4', fields))
491: (8)                     assert_equal(str(dt_int), "(numpy.int32, [(('a', '<i2'), ('b', '<i2'))]))")
492: (8)
493: (8)
494: (12)                     arr_int = np.zeros(4, dt_int)
495: (8)                     assert_equal(repr(arr_int),
496: (12)                         "array([0, 0, 0, 0], dtype=(numpy.int32, [(('a', '<i2'), ('b', '<i2')))))")
497: (0)             class TestZeroRank:
498: (4)                 def setup_method(self):
499: (8)                     self.d = np.array(0, np.array('x', object))
500: (8)                 def test_ellipsis_subscript(self):
500: (8)                     a, b = self.d
500: (8)                     assert_equal(a[...], 0)

```

```

501: (8) assert_equal(b[...], 'x')
502: (8) assert_(a[...].base is a) # `a[...]` is a` in numpy <1.9.
503: (8) assert_(b[...].base is b) # `b[...]` is b` in numpy <1.9.
504: (4) def test_empty_subscript(self):
505: (8)     a, b = self.d
506: (8)     assert_equal(a[()], 0)
507: (8)     assert_equal(b[()], 'x')
508: (8)     assert_(type(a[()]) is a.dtype.type)
509: (8)     assert_(type(b[()]) is str)
510: (4) def test_invalid_subscript(self):
511: (8)     a, b = self.d
512: (8)     assert_raises(IndexError, lambda x: x[0], a)
513: (8)     assert_raises(IndexError, lambda x: x[0], b)
514: (8)     assert_raises(IndexError, lambda x: x[np.array([], int)], a)
515: (8)     assert_raises(IndexError, lambda x: x[np.array([], int)], b)
516: (4) def test_ellipsis_subscript_assignment(self):
517: (8)     a, b = self.d
518: (8)     a[...] = 42
519: (8)     assert_equal(a, 42)
520: (8)     b[...] = ''
521: (8)     assert_equal(b.item(), '')
522: (4) def test_empty_subscript_assignment(self):
523: (8)     a, b = self.d
524: (8)     a[()] = 42
525: (8)     assert_equal(a, 42)
526: (8)     b[()] = ''
527: (8)     assert_equal(b.item(), '')
528: (4) def test_invalid_subscript_assignment(self):
529: (8)     a, b = self.d
530: (8)     def assign(x, i, v):
531: (12)         x[i] = v
532: (8)     assert_raises(IndexError, assign, a, 0, 42)
533: (8)     assert_raises(IndexError, assign, b, 0, '')
534: (8)     assert_raises(ValueError, assign, a, (), '')
535: (4) def test_newaxis(self):
536: (8)     a, b = self.d
537: (8)     assert_equal(a[np.newaxis].shape, (1,))
538: (8)     assert_equal(a[..., np.newaxis].shape, (1,))
539: (8)     assert_equal(a[np.newaxis, ...].shape, (1,))
540: (8)     assert_equal(a[..., np.newaxis].shape, (1,))
541: (8)     assert_equal(a[np.newaxis, ..., np.newaxis].shape, (1, 1))
542: (8)     assert_equal(a[..., np.newaxis, np.newaxis].shape, (1, 1))
543: (8)     assert_equal(a[np.newaxis, np.newaxis, ...].shape, (1, 1))
544: (8)     assert_equal(a[(np.newaxis,)*10].shape, (1,)*10)
545: (4) def test_invalid_newaxis(self):
546: (8)     a, b = self.d
547: (8)     def subscript(x, i):
548: (12)         x[i]
549: (8)     assert_raises(IndexError, subscript, a, (np.newaxis, 0))
550: (8)     assert_raises(IndexError, subscript, a, (np.newaxis,)*50)
551: (4) def test_constructor(self):
552: (8)     x = np.ndarray(())
553: (8)     x[()] = 5
554: (8)     assert_equal(x[()], 5)
555: (8)     y = np.ndarray((), buffer=x)
556: (8)     y[()] = 6
557: (8)     assert_equal(x[()], 6)
558: (8)     with pytest.raises(ValueError):
559: (12)         np.ndarray((2,), strides=())
560: (8)     with pytest.raises(ValueError):
561: (12)         np.ndarray((), strides=(2,))
562: (4) def test_output(self):
563: (8)     x = np.array(2)
564: (8)     assert_raises(ValueError, np.add, x, [1], x)
565: (4) def test_real_imag(self):
566: (8)     x = np.array(1j)
567: (8)     xr = x.real
568: (8)     xi = x.imag
569: (8)     assert_equal(xr, np.array(0))

```

```

570: (8) assert_(type(xr) is np.ndarray)
571: (8) assert_equal(xr.flags.contiguous, True)
572: (8) assert_equal(xr.flags.f_contiguous, True)
573: (8) assert_equal(xi, np.array(1))
574: (8) assert_(type(xi) is np.ndarray)
575: (8) assert_equal(xi.flags.contiguous, True)
576: (8) assert_equal(xi.flags.f_contiguous, True)
577: (0) class TestScalarIndexing:
578: (4)     def setup_method(self):
579: (8)         self.d = np.array([0, 1])[0]
580: (4)     def test_ellipsis_subscript(self):
581: (8)         a = self.d
582: (8)         assert_equal(a[...], 0)
583: (8)         assert_equal(a[...].shape, ())
584: (4)     def test_empty_subscript(self):
585: (8)         a = self.d
586: (8)         assert_equal(a[()], 0)
587: (8)         assert_equal(a[()].shape, ())
588: (4)     def test_invalid_subscript(self):
589: (8)         a = self.d
590: (8)         assert_raises(IndexError, lambda x: x[0], a)
591: (8)         assert_raises(IndexError, lambda x: x[np.array([], int)], a)
592: (4)     def test_invalid_subscript_assignment(self):
593: (8)         a = self.d
594: (8)         def assign(x, i, v):
595: (12)             x[i] = v
596: (8)             assert_raises(TypeError, assign, a, 0, 42)
597: (4)     def test_newaxis(self):
598: (8)         a = self.d
599: (8)         assert_equal(a[np.newaxis].shape, (1,))
600: (8)         assert_equal(a[..., np.newaxis].shape, (1,))
601: (8)         assert_equal(a[np.newaxis, ...].shape, (1,))
602: (8)         assert_equal(a[..., np.newaxis].shape, (1,))
603: (8)         assert_equal(a[np.newaxis, ..., np.newaxis].shape, (1, 1))
604: (8)         assert_equal(a[..., np.newaxis, np.newaxis].shape, (1, 1))
605: (8)         assert_equal(a[np.newaxis, np.newaxis, ...].shape, (1, 1))
606: (8)         assert_equal(a[(np.newaxis,)*10].shape, (1,)*10)
607: (4)     def test_invalid_newaxis(self):
608: (8)         a = self.d
609: (8)         def subscript(x, i):
610: (12)             x[i]
611: (8)             assert_raises(IndexError, subscript, a, (np.newaxis, 0))
612: (8)             assert_raises(IndexError, subscript, a, (np.newaxis,)*50)
613: (4)     def test_overlapping_assignment(self):
614: (8)         a = np.arange(4)
615: (8)         a[:-1] = a[1:]
616: (8)         assert_equal(a, [1, 2, 3, 3])
617: (8)         a = np.arange(4)
618: (8)         a[1:] = a[:-1]
619: (8)         assert_equal(a, [0, 0, 1, 2])
620: (8)         a = np.arange(4)
621: (8)         a[:] = a[::-1]
622: (8)         assert_equal(a, [3, 2, 1, 0])
623: (8)         a = np.arange(6).reshape(2, 3)
624: (8)         a[::-1,:] = a[:, ::-1]
625: (8)         assert_equal(a, [[5, 4, 3], [2, 1, 0]])
626: (8)         a = np.arange(6).reshape(2, 3)
627: (8)         a[::-1, ::-1] = a[:, ::-1]
628: (8)         assert_equal(a, [[3, 4, 5], [0, 1, 2]])
629: (8)         a = np.arange(5)
630: (8)         a[:3] = a[2:]
631: (8)         assert_equal(a, [2, 3, 4, 3, 4])
632: (8)         a = np.arange(5)
633: (8)         a[2:] = a[:3]
634: (8)         assert_equal(a, [0, 1, 0, 1, 2])
635: (8)         a = np.arange(5)
636: (8)         a[2::-1] = a[2:]
637: (8)         assert_equal(a, [4, 3, 2, 3, 4])
638: (8)         a = np.arange(5)

```

```

639: (8)             a[2:] = a[2::-1]
640: (8)             assert_equal(a, [0, 1, 2, 1, 0])
641: (8)             a = np.arange(5)
642: (8)             a[2::-1] = a[:1:-1]
643: (8)             assert_equal(a, [2, 3, 4, 3, 4])
644: (8)             a = np.arange(5)
645: (8)             a[:1:-1] = a[2::-1]
646: (8)             assert_equal(a, [0, 1, 0, 1, 2])
647: (0)             class TestCreation:
648: (4)                 """
649: (4)                     Test the np.array constructor
650: (4)                 """
651: (4)             def test_from_attribute(self):
652: (8)                 class x:
653: (12)                     def __array__(self, dtype=None):
654: (16)                         pass
655: (8)                         assert_raises(ValueError, np.array, x())
656: (4)             def test_from_string(self):
657: (8)                 types = np.typecodes['AllInteger'] + np.typecodes['Float']
658: (8)                 nstr = ['123', '123']
659: (8)                 result = np.array([123, 123], dtype=int)
660: (8)                 for type in types:
661: (12)                     msg = 'String conversion for %s' % type
662: (12)                     assert_equal(np.array(nstr, dtype=type), result, err_msg=msg)
663: (4)             def test_void(self):
664: (8)                 arr = np.array([], dtype='V')
665: (8)                 assert arr.dtype == 'V8' # current default
666: (8)                 arr = np.array([b"1234", b"1234"], dtype="V")
667: (8)                 assert arr.dtype == "V4"
668: (8)                 with pytest.raises(TypeError):
669: (12)                     np.array([b"1234", b"12345"], dtype="V")
670: (8)                 with pytest.raises(TypeError):
671: (12)                     np.array([b"12345", b"1234"], dtype="V")
672: (8)                     arr = np.array([b"1234", b"1234"], dtype="O").astype("V")
673: (8)                     assert arr.dtype == "V4"
674: (8)                     with pytest.raises(TypeError):
675: (12)                         np.array([b"1234", b"12345"], dtype="O").astype("V")
676: (4)             @pytest.mark.parametrize("idx",
677: (12)                 [pytest.param(Ellipsis, id="arr"), pytest.param((), id="scalar")])
678: (4)             def test_structured_void_promotion(self, idx):
679: (8)                 arr = np.array(
680: (12)                     [np.array(1, dtype="i,i")[idx], np.array(2, dtype='i,i')[idx]],
681: (12)                     dtype="V")
682: (8)                 assert_array_equal(arr, np.array([(1, 1), (2, 2)], dtype="i,i"))
683: (8)                 with pytest.raises(TypeError):
684: (12)                     np.array(
685: (16)                         [np.array(1, dtype="i,i")[idx], np.array(2, dtype='i,i,i')][idx]],
686: (16)                         dtype="V")
687: (4)             def test_too_big_error(self):
688: (8)                 if np.iinfo('intp').max == 2**31 - 1:
689: (12)                     shape = (46341, 46341)
690: (8)                 elif np.iinfo('intp').max == 2**63 - 1:
691: (12)                     shape = (3037000500, 3037000500)
692: (8)                 else:
693: (12)                     return
694: (8)                 assert_raises(ValueError, np.empty, shape, dtype=np.int8)
695: (8)                 assert_raises(ValueError, np.zeros, shape, dtype=np.int8)
696: (8)                 assert_raises(ValueError, np.ones, shape, dtype=np.int8)
697: (4)             @pytest.mark.skipif(np.dtype(np.intp).itemsize != 8,
698: (24)                             reason="malloc may not fail on 32 bit systems")
699: (4)             def test_malloc_fails(self):
700: (8)                 with assert_raises(np.core._exceptions._ArrayMemoryError):
701: (12)                     np.empty(np.iinfo(np.intp).max, dtype=np.uint8)
702: (4)             def test_zeros(self):
703: (8)                 types = np.typecodes['AllInteger'] + np.typecodes['AllFloat']
704: (8)                 for dt in types:
705: (12)                     d = np.zeros((13,), dtype=dt)
706: (12)                     assert_equal(np.count_nonzero(d), 0)

```

```

707: (12) assert_equal(d.sum(), 0)
708: (12) assert_(not d.any())
709: (12) d = np.zeros(2, dtype='(2,4)i4')
710: (12) assert_equal(np.count_nonzero(d), 0)
711: (12) assert_equal(d.sum(), 0)
712: (12) assert_(not d.any())
713: (12) d = np.zeros(2, dtype='4i4')
714: (12) assert_equal(np.count_nonzero(d), 0)
715: (12) assert_equal(d.sum(), 0)
716: (12) assert_(not d.any())
717: (12) d = np.zeros(2, dtype='(2,4)i4, (2,4)i4')
718: (12) assert_equal(np.count_nonzero(d), 0)
719: (4) @pytest.mark.slow
720: (4) def test_zeros_big(self):
721: (8)     types = np.typecodes['AllInteger'] + np.typecodes['AllFloat']
722: (8)     for dt in types:
723: (12)         d = np.zeros((30 * 1024**2,), dtype=dt)
724: (12)         assert_(not d.any())
725: (12)         del(d)
726: (4) def test_zeros_obj(self):
727: (8)     d = np.zeros((13,), dtype=object)
728: (8)     assert_array_equal(d, [0] * 13)
729: (8)     assert_equal(np.count_nonzero(d), 0)
730: (4) def test_zeros_obj_obj(self):
731: (8)     d = np.zeros(10, dtype=[('k', object, 2)])
732: (8)     assert_array_equal(d['k'], 0)
733: (4) def test_zeros_like_like_zeros(self):
734: (8)     for c in np.typecodes['All']:
735: (12)         if c == 'V':
736: (16)             continue
737: (12)         d = np.zeros((3,3), dtype=c)
738: (12)         assert_array_equal(np.zeros_like(d), d)
739: (12)         assert_equal(np.zeros_like(d).dtype, d.dtype)
740: (8)         d = np.zeros((3,3), dtype='S5')
741: (8)         assert_array_equal(np.zeros_like(d), d)
742: (8)         assert_equal(np.zeros_like(d).dtype, d.dtype)
743: (8)         d = np.zeros((3,3), dtype='U5')
744: (8)         assert_array_equal(np.zeros_like(d), d)
745: (8)         assert_equal(np.zeros_like(d).dtype, d.dtype)
746: (8)         d = np.zeros((3,3), dtype='<i4')
747: (8)         assert_array_equal(np.zeros_like(d), d)
748: (8)         assert_equal(np.zeros_like(d).dtype, d.dtype)
749: (8)         d = np.zeros((3,3), dtype='>i4')
750: (8)         assert_array_equal(np.zeros_like(d), d)
751: (8)         assert_equal(np.zeros_like(d).dtype, d.dtype)
752: (8)         d = np.zeros((3,3), dtype='<M8[s]')
753: (8)         assert_array_equal(np.zeros_like(d), d)
754: (8)         assert_equal(np.zeros_like(d).dtype, d.dtype)
755: (8)         d = np.zeros((3,3), dtype='>M8[s]')
756: (8)         assert_array_equal(np.zeros_like(d), d)
757: (8)         assert_equal(np.zeros_like(d).dtype, d.dtype)
758: (8)         d = np.zeros((3,3), dtype='f4,f4')
759: (8)         assert_array_equal(np.zeros_like(d), d)
760: (8)         assert_equal(np.zeros_like(d).dtype, d.dtype)
761: (4) def test_empty_unicode(self):
762: (8)     for i in range(5, 100, 5):
763: (12)         d = np.empty(i, dtype='U')
764: (12)         str(d)
765: (4) def test_sequence_non_homogeneous(self):
766: (8)     assert_equal(np.array([4, 2**80]).dtype, object)
767: (8)     assert_equal(np.array([4, 2**80, 4]).dtype, object)
768: (8)     assert_equal(np.array([2**80, 4]).dtype, object)
769: (8)     assert_equal(np.array([2**80] * 3).dtype, object)
770: (8)     assert_equal(np.array([[1, 1], [1j, 1j]]).dtype, complex)
771: (8)     assert_equal(np.array([[1j, 1j], [1, 1]]).dtype, complex)
772: (8)     assert_equal(np.array([[1, 1, 1], [1, 1j, 1.], [1, 1, 1]]).dtype, complex)
773: (4) def test_non_sequence_sequence(self):
774: (8)     """Should not segfault.

```

```

775: (8)             Class Fail breaks the sequence protocol for new style classes, i.e.,  

776: (8)             those derived from object. Class Map is a mapping type indicated by  

777: (8)             raising a ValueError. At some point we may raise a warning instead  

778: (8)             of an error in the Fail case.  

779: (8)  

780: (8)             """  

781: (12)             class Fail:  

782: (16)                 def __len__(self):  

783: (12)                     return 1  

784: (16)                     def __getitem__(self, index):  

785: (8)                         raise ValueError()  

786: (12)             class Map:  

787: (16)                 def __len__(self):  

788: (12)                     return 1  

789: (16)                     def __getitem__(self, index):  

790: (8)                         raise KeyError()  

791: (8)                     a = np.array([Map()])  

792: (8)                     assert_(a.shape == (1,))  

793: (8)                     assert_(a.dtype == np.dtype(object))  

794: (4)                     assert_raises(ValueError, np.array, [Fail()])  

def test_no_len_object_type(self):  

    class Point2:  

        def __init__(self):  

            pass  

        def __getitem__(self, ind):  

            if ind in [0, 1]:  

                return ind  

            else:  

                raise IndexError()  

    d = np.array([Point2(), Point2(), Point2()])  

    assert_equal(d.dtype, np.dtype(object))  

def test_false_len_sequence(self):  

    class C:  

        def __getitem__(self, i):  

            raise IndexError  

        def __len__(self):  

            return 42  

    a = np.array(C()) # segfault?  

    assert_equal(len(a), 0)  

def test_false_len_iterable(self):  

    class C:  

        def __getitem__(self, x):  

            raise Exception  

        def __iter__(self):  

            return iter()  

        def __len__(self):  

            return 2  

    a = np.empty(2)  

    with assert_raises(ValueError):  

        a[:] = C() # Segfault!  

    np.array(C()) == list(C())  

def test_failed_len_sequence(self):  

    class A:  

        def __init__(self, data):  

            self._data = data  

        def __getitem__(self, item):  

            return type(self)(self._data[item])  

        def __len__(self):  

            return len(self._data)  

    d = A([1,2,3])  

    assert_equal(len(np.array(d)), 3)  

def test_array_too_big(self):  

    buf = np.zeros(100)  

    max_bytes = np.iinfo(np.intp).max  

    for dtype in ["intp", "S20", "b"]:  

        dtype = np.dtype(dtype)  

        itemsize = dtype.itemsize  

        np.ndarray(buffer=buf, strides=(0,),  

                  shape=(max_bytes//itemsize,), dtype=dtype)  

        assert_raises(ValueError, np.ndarray, buffer=buf, strides=(0,),)

```

```

844: (26)                                     shape=(max_bytes//itemsize + 1,), dtype=dtype)
845: (4)          def _ragged_creation(self, seq):
846: (8)            with pytest.raises(ValueError, match=".detected shape was"):
847: (12)              a = np.array(seq)
848: (8)            return np.array(seq, dtype=object)
849: (4)          def test_ragged_ndim_object(self):
850: (8)            a = self._ragged_creation([[1], 2, 3])
851: (8)            assert_equal(a.shape, (3,))
852: (8)            assert_equal(a.dtype, object)
853: (8)            a = self._ragged_creation([1, [2], 3])
854: (8)            assert_equal(a.shape, (3,))
855: (8)            assert_equal(a.dtype, object)
856: (8)            a = self._ragged_creation([1, 2, [3]])
857: (8)            assert_equal(a.shape, (3,))
858: (8)            assert_equal(a.dtype, object)
859: (4)          def test_ragged_shape_object(self):
860: (8)            a = self._ragged_creation([[1, 1], [2], [3]])
861: (8)            assert_equal(a.shape, (3,))
862: (8)            assert_equal(a.dtype, object)
863: (8)            a = self._ragged_creation([[1], [2, 2], [3]])
864: (8)            assert_equal(a.shape, (3,))
865: (8)            assert_equal(a.dtype, object)
866: (8)            a = self._ragged_creation([[1], [2], [3, 3]])
867: (8)            assert a.shape == (3,)
868: (8)            assert a.dtype == object
869: (4)          def test_array_of_ragged_array(self):
870: (8)            outer = np.array([None, None])
871: (8)            outer[0] = outer[1] = np.array([1, 2, 3])
872: (8)            assert np.array(outer).shape == (2,)
873: (8)            assert np.array([outer]).shape == (1, 2)
874: (8)            outer_ragged = np.array([None, None])
875: (8)            outer_ragged[0] = np.array([1, 2, 3])
876: (8)            outer_ragged[1] = np.array([1, 2, 3, 4])
877: (8)            assert np.array(outer_ragged).shape == (2,)
878: (8)            assert np.array([outer_ragged]).shape == (1, 2,)
879: (4)          def test_deep_nonragged_object(self):
880: (8)            a = np.array([[[Decimal(1)]]])
881: (8)            a = np.array([1, Decimal(1)])
882: (8)            a = np.array([[1], [Decimal(1)]])
883: (4)          @pytest.mark.parametrize("dtype", [object, "0,0", "0,(3)0", "(2,3)0"])
884: (4)          @pytest.mark.parametrize("function", [
885: (12)              np.ndarray, np.empty,
886: (12)              lambda shape, dtype: np.empty_like(np.empty(shape, dtype=dtype))])
887: (4)          def test_object_initialized_to_None(self, function, dtype):
888: (8)            arr = function(3, dtype=dtype)
889: (8)            expected = np.array(None).tobytes()
890: (8)            expected = expected * (arr.nbytes // len(expected))
891: (8)            assert arr.tobytes() == expected
892: (4)          @pytest.mark.parametrize("func", [
893: (8)              np.array, np.asarray, np.asanyarray, np.ascontiguousarray,
894: (8)              np.asfortranarray])
895: (4)          def test_creation_from_dtypemeta(self, func):
896: (8)            dtype = np.dtype('i')
897: (8)            arr1 = func([1, 2, 3], dtype=dtype)
898: (8)            arr2 = func([1, 2, 3], dtype=type(dtype))
899: (8)            assert_array_equal(arr1, arr2)
900: (8)            assert arr2.dtype == dtype
901: (0)          class TestStructured:
902: (4)            def test_subarray_field_access(self):
903: (8)              a = np.zeros((3, 5), dtype=[('a', ('i4', (2, 2)))])
904: (8)              a['a'] = np.arange(60).reshape(3, 5, 2, 2)
905: (8)              assert_array_equal(a.T['a'], a['a'].transpose(1, 0, 2, 3))
906: (8)              b = a.copy(order='F')
907: (8)              assert_equal(a['a'].shape, b['a'].shape)
908: (8)              assert_equal(a.T['a'].shape, a.T.copy()['a'].shape)
909: (4)            def test_subarray_comparison(self):
910: (8)              a = np.rec.fromrecords(
911: (12)                  ([1, 2, 3], 'a', [[1, 2], [3, 4]]), ([3, 3, 3], 'b', [[0, 0], [0, 0]]),
912: (12)                  )

```

```

912: (12)          dtype=[('a', ('f4', 3)), ('b', object), ('c', ('i4', (2, 2)))]
913: (8)           b = a.copy()
914: (8)           assert_equal(a == b, [True, True])
915: (8)           assert_equal(a != b, [False, False])
916: (8)           b[1].b = 'c'
917: (8)           assert_equal(a == b, [True, False])
918: (8)           assert_equal(a != b, [False, True])
919: (8)           for i in range(3):
920: (12)             b[0].a = a[0].a
921: (12)             b[0].a[i] = 5
922: (12)             assert_equal(a == b, [False, False])
923: (12)             assert_equal(a != b, [True, True])
924: (8)           for i in range(2):
925: (12)             for j in range(2):
926: (16)               b = a.copy()
927: (16)               b[0].c[i, j] = 10
928: (16)               assert_equal(a == b, [False, True])
929: (16)               assert_equal(a != b, [True, False])
930: (8)           a = np.array([[(0,)], [(1,)]], dtype=[('a', 'f8')])
931: (8)           b = np.array([(0,), (0,), (1,)], dtype=[('a', 'f8')])
932: (8)           assert_equal(a == b, [[True, True, False], [False, False, True]])
933: (8)           assert_equal(b == a, [[True, True, False], [False, False, True]])
934: (8)           a = np.array([[(0,)], [(1,)]], dtype=[('a', 'f8', (1,))])
935: (8)           b = np.array([(0,), (0,), (1,)], dtype=[('a', 'f8', (1,))])
936: (8)           assert_equal(a == b, [[True, True, False], [False, False, True]])
937: (8)           assert_equal(b == a, [[True, True, False], [False, False, True]])
938: (8)           a = np.array([[[[0, 0],)], [[(1, 1),]]], dtype=[('a', 'f8', (2,))])
939: (8)           b = np.array([[[[0, 0],), ([0, 1],), ([1, 1],)], dtype=[('a', 'f8',
940: (2,))]])
940: (8)             assert_equal(a == b, [[True, False, False], [False, False, True]])
941: (8)             assert_equal(b == a, [[True, False, False], [False, False, True]])
942: (8)           a = np.array([[[[0, 0],)], [[(1, 1),]]], dtype=[('a', 'f8', (2,))],
943: (8)             b = np.array([[[[0, 0],), ([0, 1],), ([1, 1],)], dtype=[('a', 'f8',
944: (2,))]])
944: (8)             assert_equal(a == b, [[True, False, False], [False, False, True]])
945: (8)             assert_equal(b == a, [[True, False, False], [False, False, True]])
946: (8)             x = np.zeros((1,), dtype=[('a', ('f4', (1, 2))), ('b', 'i1')])
947: (8)             y = np.zeros((1,), dtype=[('a', ('f4', (2,))), ('b', 'i1')])
948: (8)             with pytest.raises(TypeError):
949: (12)               x == y
950: (8)             x = np.zeros((1,), dtype=[('a', ('f4', (2, 1))), ('b', 'i1')])
951: (8)             y = np.zeros((1,), dtype=[('a', ('f4', (2,))), ('b', 'i1')])
952: (8)             with pytest.raises(TypeError):
953: (12)               x == y
954: (4)           def test_empty_structured_array_comparison(self):
955: (8)             a = np.zeros(0, [('a', '<f8', (1, 1))])
956: (8)             assert_equal(a, a)
957: (8)             a = np.zeros(0, [('a', '<f8', (1,))])
958: (8)             assert_equal(a, a)
959: (8)             a = np.zeros((0, 0), [('a', '<f8', (1, 1))])
960: (8)             assert_equal(a, a)
961: (8)             a = np.zeros((1, 0, 1), [('a', '<f8', (1, 1))])
962: (8)             assert_equal(a, a)
963: (4)           @pytest.mark.parametrize("op", [operator.eq, operator.ne])
964: (4)           def test_structured_array_comparison_bad_broadcasts(self, op):
965: (8)             a = np.zeros(3, dtype='i,i')
966: (8)             b = np.array([], dtype="i,i")
967: (8)             with pytest.raises(ValueError):
968: (12)               op(a, b)
969: (4)           def test_structured_comparisons_with_promotion(self):
970: (8)             a = np.array([(5, 42), (10, 1)], dtype=[('a', '>i8'), ('b', '<f8')])
971: (8)             b = np.array([(5, 43), (10, 1)], dtype=[('a', '<i8'), ('b', '>f8')])
972: (8)             assert_equal(a == b, [False, True])
973: (8)             assert_equal(a != b, [True, False])
974: (8)             a = np.array([(5, 42), (10, 1)], dtype=[('a', '>f8'), ('b', '<f8')])
975: (8)             b = np.array([(5, 43), (10, 1)], dtype=[('a', '<i8'), ('b', '>i8')])
976: (8)             assert_equal(a == b, [False, True])
977: (8)             assert_equal(a != b, [True, False])

```

```

978: (8)         a = np.array([(5, 42), (10, 1)], dtype=[('a', '10>f8'), ('b',
979: (8)           '5<f8')])                                '5>i8')])
980: (8)         b = np.array([(5, 43), (10, 1)], dtype=[('a', '10<i8'), ('b',
981: (8)           '5>i8')])                                '5>i8'])
982: (4)         assert_equal(a == b, [False, True])
983: (12)        assert_equal(a != b, [True, False])
984: (12)        @pytest.mark.parametrize("op", [
985: (4)          operator.eq, lambda x, y: operator.eq(y, x),
986: (8)          operator.ne, lambda x, y: operator.ne(y, x)])
987: (8)        def test_void_comparison_failures(self, op):
988: (8)          x = np.zeros(3, dtype=[('a', 'i1')])
989: (12)        y = np.zeros(3)
990: (8)          with pytest.raises(TypeError):
991: (8)            op(x, y)
992: (8)          y = np.zeros(3, dtype=[('title', 'a'), 'i1')])
993: (12)        assert np.can_cast(y.dtype, x.dtype)
994: (8)          with pytest.raises(TypeError):
995: (8)            op(x, y)
996: (8)          x = np.zeros(3, dtype="V7")
997: (12)        y = np.zeros(3, dtype="V8")
998: (4)          with pytest.raises(TypeError):
999: (8)            op(x, y)
1000: (8)        def test_casting(self):
1001: (8)          a = np.array([(1,)], dtype=[('a', '<i4')])
1002: (8)          assert_(np.can_cast(a.dtype, [('a', '>i4')], casting='unsafe'))
1003: (8)          b = a.astype([('a', '>i4')])
1004: (8)          assert_equal(b, a.byteswap().newbyteorder())
1005: (8)          assert_equal(a['a'][0], b['a'][0])
1006: (8)          a = np.array([(5, 42), (10, 1)], dtype=[('a', '>i4'), ('b', '<f8')])
1007: (8)          b = np.array([(5, 42), (10, 1)], dtype=[('a', '<i4'), ('b', '>f8')])
1008: (8)          assert_(np.can_cast(a.dtype, b.dtype, casting='equiv'))
1009: (8)          assert_equal(a == b, [True, True])
1010: (8)          assert_(np.can_cast(a.dtype, b.dtype, casting='equiv'))
1011: (8)          c = a.astype(b.dtype, casting='equiv')
1012: (8)          assert_equal(a == c, [True, True])
1013: (8)          t = [('a', '<i8'), ('b', '>f8')]
1014: (8)          assert_(np.can_cast(a.dtype, t, casting='safe'))
1015: (21)        c = a.astype(t, casting='safe')
1016: (8)          assert_equal((c == np.array([(5, 42), (10, 1)], dtype=t)),
1017: (8)                        [True, True])
1018: (8)          t = [('a', '<i4'), ('b', '>f4')]
1019: (8)          assert_(np.can_cast(a.dtype, t, casting='same_kind'))
1020: (21)        c = a.astype(t, casting='same_kind')
1021: (8)          assert_equal((c == np.array([(5, 42), (10, 1)], dtype=t)),
1022: (8)                        [True, True])
1023: (8)          t = [('a', '>i8'), ('b', '<f4')]
1024: (8)          assert_(not np.can_cast(a.dtype, t, casting='safe'))
1025: (8)          assert_raises(TypeError, a.astype, t, casting='safe')
1026: (8)          t = [('a', '>i2'), ('b', '<f8')]
1027: (8)          assert_(not np.can_cast(a.dtype, t, casting='equiv'))
1028: (8)          assert_raises(TypeError, a.astype, t, casting='equiv')
1029: (8)          t = [('a', '>i8'), ('b', '<i2')]
1030: (8)          assert_(not np.can_cast(a.dtype, t, casting='same_kind'))
1031: (8)          assert_raises(TypeError, a.astype, t, casting='same_kind')
1032: (8)          assert_(not np.can_cast(a.dtype, b.dtype, casting='no'))
1033: (12)        assert_raises(TypeError, a.astype, b.dtype, casting='no')
1034: (12)        for casting in ['no', 'safe', 'equiv', 'same_kind']:
1035: (12)          t = [('a', '>i4')]
1036: (12)          assert_(not np.can_cast(a.dtype, t, casting=casting))
1037: (4)          t = [('a', '>i4'), ('b', '<f8'), ('c', 'i4')]
1038: (8)          assert_(not np.can_cast(a.dtype, t, casting=casting))
1039: (8)        def test_objview(self):
1040: (8)          a = np.array([], dtype=[('a', 'f'), ('b', 'f'), ('c', '0')])
1041: (8)          a[['a', 'b']] # TypeError?
1042: (4)          dat2 = np.zeros(3, [('A', 'i'), ('B', '|0')])
1043: (8)          dat2[['B', 'A']] # TypeError?
1044: (8)        def test_setfield(self):

```

```

1045: (8)          x = np.zeros(1, dt)
1046: (8)          x[0]['field'] = np.ones(10, dtype='i4')
1047: (8)          x[0]['struct'] = np.ones(1, dtype=struct_dt)
1048: (8)          assert_equal(x[0]['field'], np.ones(10, dtype='i4'))
1049: (4)          def test_setfield_object(self):
1050: (8)              b = np.zeros(1, dtype=[('x', 'O')])
1051: (8)              b[0]['x'] = np.arange(3)
1052: (8)              assert_equal(b[0]['x'], np.arange(3))
1053: (8)              c = np.zeros(1, dtype=[('x', 'O', 5)])
1054: (8)              def testassign():
1055: (12)                  c[0]['x'] = np.arange(3)
1056: (8)                  assert_raises(ValueError, testassign)
1057: (4)          def test_zero_width_string(self):
1058: (8)              dt = np.dtype([('I', int), ('S', 'S0')])
1059: (8)              x = np.zeros(4, dtype=dt)
1060: (8)              assert_equal(x['S'], [b'', b'', b'', b''])
1061: (8)              assert_equal(x['S'].itemsize, 0)
1062: (8)              x['S'] = ['a', 'b', 'c', 'd']
1063: (8)              assert_equal(x['S'], [b'', b'', b'', b''])
1064: (8)              assert_equal(x['I'], [0, 0, 0, 0])
1065: (8)              x['S'][x['I'] == 0] = 'hello'
1066: (8)              assert_equal(x['S'], [b'', b'', b'', b''])
1067: (8)              assert_equal(x['I'], [0, 0, 0, 0])
1068: (8)              x['S'] = 'A'
1069: (8)              assert_equal(x['S'], [b'', b'', b'', b''])
1070: (8)              assert_equal(x['I'], [0, 0, 0, 0])
1071: (8)              y = np.ndarray(4, dtype=x['S'].dtype)
1072: (8)              assert_equal(y.itemsize, 0)
1073: (8)              assert_equal(x['S'], y)
1074: (8)              assert_equal(np.zeros(4, dtype=[('a', 'S0,S0'),
1075: (40)                                ('b', 'u1')])['a'].itemsize, 0)
1076: (8)              assert_equal(np.empty(3, dtype='S0,S0').itemsize, 0)
1077: (8)              assert_equal(np.zeros(4, dtype='S0,u1')['f0'].itemsize, 0)
1078: (8)              xx = x['S'].reshape((2, 2))
1079: (8)              assert_equal(xx.itemsize, 0)
1080: (8)              assert_equal(xx, [[b'', b''], [b'', b'']])
1081: (8)              assert_equal(xx[:].dtype, xx.dtype)
1082: (8)              assert_array_equal(eval(repr(xx)), dict(array=np.array))
1083: (8)              b = io.BytesIO()
1084: (8)              np.save(b, xx)
1085: (8)              b.seek(0)
1086: (8)              yy = np.load(b)
1087: (8)              assert_equal(yy.itemsize, 0)
1088: (8)              assert_equal(xx, yy)
1089: (8)              with temppath(suffix='.npy') as tmp:
1090: (12)                  np.save(tmp, xx)
1091: (12)                  yy = np.load(tmp)
1092: (12)                  assert_equal(yy.itemsize, 0)
1093: (12)                  assert_equal(xx, yy)
1094: (4)          def test_base_attr(self):
1095: (8)              a = np.zeros(3, dtype='i4,f4')
1096: (8)              b = a[0]
1097: (8)              assert_(b.base is a)
1098: (4)          def test_assignment(self):
1099: (8)              def testassign(arr, v):
1100: (12)                  c = arr.copy()
1101: (12)                  c[0] = v # assign using setitem
1102: (12)                  c[1:] = v # assign using "dtype_transfer" code paths
1103: (12)                  return c
1104: (8)              dt = np.dtype([('foo', 'i8'), ('bar', 'i8')])
1105: (8)              arr = np.ones(2, dt)
1106: (8)              v1 = np.array([(2,3)], dtype=[('foo', 'i8'), ('bar', 'i8')])
1107: (8)              v2 = np.array([(2,3)], dtype=[('bar', 'i8'), ('foo', 'i8')])
1108: (8)              v3 = np.array([(2,3)], dtype=[('bar', 'i8'), ('baz', 'i8')])
1109: (8)              v4 = np.array([(2,)], dtype=[('bar', 'i8')])
1110: (8)              v5 = np.array([(2,3)], dtype=[('foo', 'f8'), ('bar', 'f8')])
1111: (8)              w = arr.view({'names': ['bar'], 'formats': ['i8'], 'offsets': [8]})
1112: (8)              ans = np.array([(2,3),(2,3)], dtype=dt)
1113: (8)              assert_equal(testassign(arr, v1), ans)

```

```

1114: (8) assert_equal(testassign(arr, v2), ans)
1115: (8) assert_equal(testassign(arr, v3), ans)
1116: (8) assert_raises(TypeError, lambda: testassign(arr, v4))
1117: (8) assert_equal(testassign(arr, v5), ans)
1118: (8) w[:] = 4
1119: (8) assert_equal(arr, np.array([(1,4),(1,4)], dtype=dt))
1120: (8) a = np.array([(1,2,3)], dtype=[('foo', 'i8'), ('bar', 'i8'), ('baz', 'f4')])
1121: (21) a[['foo', 'bar']] = a[['bar', 'foo']]
1122: (8) assert_equal(a[0].item(), (2,1,3))
1123: (8) a = np.array([(1,2)], dtype=[('a', 'i4'), ('b', 'i4')])
1124: (8) a[['a', 'b']] = a[['b', 'a']]
1125: (8) assert_equal(a[0].item(), (2,1))
1126: (8) def test_scalar_assignment(self):
1127: (4)     with assert_raises(ValueError):
1128: (8)         arr = np.arange(25).reshape(5, 5)
1129: (12)         arr.itemset(3)
1130: (12)
1131: (4) def test_structuredscalar_indexing(self):
1132: (8)     x = np.empty(shape=1, dtype="(2)3S,(2)3U")
1133: (8)     assert_equal(x[["f0","f1"]][0], x[0][["f0","f1"]])
1134: (8)     assert_equal(x[0], x[0][()])
1135: (4) def test_multiindex_titles(self):
1136: (8)     a = np.zeros(4, dtype=[(('a', 'b'), 'i'), ('c', 'i'), ('d', 'i')])
1137: (8)     assert_raises(KeyError, lambda : a[['a','c']])
1138: (8)     assert_raises(KeyError, lambda : a[['a','a']])
1139: (8)     assert_raises(ValueError, lambda : a[['b','b']]) # field exists, but
repeated
1140: (8)     a[['b','c']] # no exception
1141: (4) def test_structured_cast_promotion_fieldorder(self):
1142: (8)     A = ("a", "<i8")
1143: (8)     B = ("b", ">i8")
1144: (8)     ab = np.array([(1, 2)], dtype=[A, B])
1145: (8)     ba = np.array([(1, 2)], dtype=[B, A])
1146: (8)     assert_raises(TypeError, np.concatenate, ab, ba)
1147: (8)     assert_raises(TypeError, np.result_type, ab.dtype, ba.dtype)
1148: (8)     assert_raises(TypeError, np.promote_types, ab.dtype, ba.dtype)
1149: (8)     assert_equal(np.promote_types(ab.dtype, ba[['a', 'b']].dtype),
1150: (21)             repack_fields(ab.dtype.newbyteorder('N')))
1151: (8)     assert_equal(np.can_cast(ab.dtype, ba.dtype), True)
1152: (8)     assert_equal(ab.astype(ba.dtype).dtype, ba.dtype)
1153: (8)     assert_equal(np.can_cast('f8,i8', [('f0', 'f8'), ('f1', 'i8')]), True)
1154: (8)     assert_equal(np.can_cast('f8,i8', [('f1', 'f8'), ('f0', 'i8')]), True)
1155: (8)     assert_equal(np.can_cast('f8,i8', [('f1', 'i8'), ('f0', 'f8')]),
False)
1156: (8)     assert_equal(np.can_cast('f8,i8', [('f1', 'i8'), ('f0', 'f8')]),
1157: (33)             casting='unsafe'), True)
1158: (8)     ab[:] = ba # make sure assignment still works
1159: (8)     dt1 = np.dtype([('",","i4')])
1160: (8)     dt2 = np.dtype([('",","i8')])
1161: (8)     assert_equal(np.promote_types(dt1, dt2), np.dtype([('f0', 'i8'))))
1162: (8)     assert_equal(np.promote_types(dt2, dt1), np.dtype([('f0', 'i8'))))
1163: (8)     assert_raises(TypeError, np.promote_types, dt1, np.dtype([('",
1164: (8) "V3"))))
1165: (21)     assert_equal(np.promote_types('i4,f8', 'i8,f4'),
1166: (8)             np.dtype([('f0', 'i8'), ('f1', 'f8')))))
1167: (8)     dt1nest = np.dtype([('","dt1')])
1168: (8)     dt2nest = np.dtype([('","dt2')])
1169: (21)     assert_equal(np.promote_types(dt1nest, dt2nest),
1170: (8)             np.dtype([('f0', np.dtype([('f0', 'i8'))))]))
1171: (8)     dt = np.dtype({'names': ['x'], 'formats': ['i4'], 'offsets': [8]})
1172: (8)     a = np.ones(3, dtype=dt)
1173: (4)     assert_equal(np.concatenate([a, a]).dtype, np.dtype([('x', 'i4'))))
1174: (12) @pytest.mark.parametrize("dtype_dict", [
1175: (12)     dict(names=["a", "b"], formats=["i4", "f"], itemsize=100),
1176: (17)     dict(names=["a", "b"], formats=["i4", "f"],
1177: (4)         offsets=[0, 12]))]
1178: (4) @pytest.mark.parametrize("align", [True, False])
1179: (8) def test_structured_promotion_packs(self, dtype_dict, align):
1180: (8)     dtype = np.dtype(dtype_dict, align=align)

```

```

1180: (8)          dtype_dict.pop("itemsize", None)
1181: (8)          dtype_dict.pop("offsets", None)
1182: (8)          expected = np.dtype(dtype_dict, align=align)
1183: (8)          res = np.promote_types(dtype, dtype)
1184: (8)          assert res.itemsize == expected.itemsize
1185: (8)          assert res.fields == expected.fields
1186: (8)          res = np.promote_types(expected, expected)
1187: (8)          assert res is expected
1188: (4)          def test_structured_asarray_is_view(self):
1189: (8)              arr = np.array([1], dtype="i,i")
1190: (8)              scalar = arr[0]
1191: (8)              assert not scalar.flags.owndata # view into the array
1192: (8)              assert np.asarray(scalar).base is scalar
1193: (8)              assert np.asarray(scalar, dtype=scalar.dtype).base is None
1194: (8)              scalar = pickle.loads(pickle.dumps(scalar))
1195: (8)              assert scalar.flags.owndata
1196: (8)              assert np.asarray(scalar).base is None
1197: (0)          class TestBool:
1198: (4)              def test_test_interning(self):
1199: (8)                  a0 = np.bool_(0)
1200: (8)                  b0 = np.bool_(False)
1201: (8)                  assert_(a0 is b0)
1202: (8)                  a1 = np.bool_(1)
1203: (8)                  b1 = np.bool_(True)
1204: (8)                  assert_(a1 is b1)
1205: (8)                  assert_(np.array([True])[0] is a1)
1206: (8)                  assert_(np.array(True)[()] is a1)
1207: (4)          def test_sum(self):
1208: (8)              d = np.ones(101, dtype=bool)
1209: (8)              assert_equal(d.sum(), d.size)
1210: (8)              assert_equal(d[::-2].sum(), d[::-2].size)
1211: (8)              assert_equal(d[:-2].sum(), d[:-2].size)
1212: (8)              d = np.frombuffer(b'\xFF\xFF' * 100, dtype=bool)
1213: (8)              assert_equal(d.sum(), d.size)
1214: (8)              assert_equal(d[::-2].sum(), d[::-2].size)
1215: (8)              assert_equal(d[:-2].sum(), d[:-2].size)
1216: (4)          def check_count_nonzero(self, power, length):
1217: (8)              powers = [2 ** i for i in range(length)]
1218: (8)              for i in range(2**power):
1219: (12)                  l = [(i & x) != 0 for x in powers]
1220: (12)                  a = np.array(l, dtype=bool)
1221: (12)                  c = builtins.sum(l)
1222: (12)                  assert_equal(np.count_nonzero(a), c)
1223: (12)                  av = a.view(np.uint8)
1224: (12)                  av *= 3
1225: (12)                  assert_equal(np.count_nonzero(a), c)
1226: (12)                  av *= 4
1227: (12)                  assert_equal(np.count_nonzero(a), c)
1228: (12)                  av[av != 0] = 0xFF
1229: (12)                  assert_equal(np.count_nonzero(a), c)
1230: (4)          def test_count_nonzero(self):
1231: (8)              self.check_count_nonzero(12, 17)
1232: (4)          @pytest.mark.slow
1233: (4)          def test_count_nonzero_all(self):
1234: (8)              self.check_count_nonzero(17, 17)
1235: (4)          def test_count_nonzero_unaligned(self):
1236: (8)              for o in range(7):
1237: (12)                  a = np.zeros((18,), dtype=bool)[o+1:]
1238: (12)                  a[:o] = True
1239: (12)                  assert_equal(np.count_nonzero(a), builtins.sum(a.tolist()))
1240: (12)                  a = np.ones((18,), dtype=bool)[o+1:]
1241: (12)                  a[:o] = False
1242: (12)                  assert_equal(np.count_nonzero(a), builtins.sum(a.tolist()))
1243: (4)          def _test_cast_from_flexible(self, dtype):
1244: (8)              for n in range(3):
1245: (12)                  v = np.array(b'', (dtype, n))
1246: (12)                  assert_equal(bool(v), False)
1247: (12)                  assert_equal(bool(v[()]), False)
1248: (12)                  assert_equal(v.astype(bool), False)

```

```

1249: (12)             assert_(isinstance(v.astype(bool), np.ndarray))
1250: (12)             assert_(v[()].astype(bool) is np.False_)
1251: (8)              for n in range(1, 4):
1252: (12)                for val in [b'a', b'0', b' ']:
1253: (16)                  v = np.array(val, (dtype, n))
1254: (16)                  assert_equal(bool(v), True)
1255: (16)                  assert_equal(bool(v[()]), True)
1256: (16)                  assert_equal(v.astype(bool), True)
1257: (16)                  assert_(isinstance(v.astype(bool), np.ndarray))
1258: (16)                  assert_(v[()].astype(bool) is np.True_)
1259: (4)               def test_cast_from_void(self):
1260: (8)                 self._test_cast_from_flexible(np.void)
1261: (4) @pytest.mark.xfail(reason="See gh-9847")
1262: (4)               def test_cast_from_unicode(self):
1263: (8)                 self._test_cast_from_flexible(np.str_)
1264: (4) @pytest.mark.xfail(reason="See gh-9847")
1265: (4)               def test_cast_from_bytes(self):
1266: (8)                 self._test_cast_from_flexible(np.bytes_)
1267: (0) class TestZeroSizeFlexible:
1268: (4) @staticmethod
1269: (4)   def _zeros(shape, dtype=str):
1270: (8)     dtype = np.dtype(dtype)
1271: (8)     if dtype == np.void:
1272: (12)       return np.zeros(shape, dtype=(dtype, 0))
1273: (8)     dtype = np.dtype([('x', dtype, 0)])
1274: (8)     return np.zeros(shape, dtype=dtype)['x']
1275: (4)   def test_create(self):
1276: (8)     zs = self._zeros(10, bytes)
1277: (8)     assert_equal(zs.itemsize, 0)
1278: (8)     zs = self._zeros(10, np.void)
1279: (8)     assert_equal(zs.itemsize, 0)
1280: (8)     zs = self._zeros(10, str)
1281: (8)     assert_equal(zs.itemsize, 0)
1282: (4)   def _test_sort_partition(self, name, kinds, **kwargs):
1283: (8)     for dt in [bytes, np.void, str]:
1284: (12)       zs = self._zeros(10, dt)
1285: (12)       sort_method = getattr(zs, name)
1286: (12)       sort_func = getattr(np, name)
1287: (12)       for kind in kinds:
1288: (16)         sort_method(kind=kind, **kwargs)
1289: (16)         sort_func(zs, kind=kind, **kwargs)
1290: (4)   def test_sort(self):
1291: (8)     self._test_sort_partition('sort', kinds='qhs')
1292: (4)   def test_argsort(self):
1293: (8)     self._test_sort_partition('argsort', kinds='qhs')
1294: (4)   def test_partition(self):
1295: (8)     self._test_sort_partition('partition', kinds=['introselect'], kth=2)
1296: (4)   def test_argpartition(self):
1297: (8)     self._test_sort_partition('argpartition', kinds=['introselect'],
1298: (4)   kth=2)
1299: (8)   def test_resize(self):
1300: (12)     for dt in [bytes, np.void, str]:
1301: (12)       zs = self._zeros(10, dt)
1302: (12)       zs.resize(25)
1303: (12)       zs.resize((10, 10))
1304: (4)   def test_view(self):
1305: (8)     for dt in [bytes, np.void, str]:
1306: (12)       zs = self._zeros(10, dt)
1307: (12)       assert_equal(zs.view(dt).dtype, np.dtype(dt))
1308: (12)       assert_equal(zs.view((dt, 1)).shape, (0,))
1309: (4)   def test.dumps(self):
1310: (8)     zs = self._zeros(10, int)
1311: (8)     assert_equal(zs, pickle.loads(zs.dumps()))
1312: (4)   def test_pickle(self):
1313: (8)     for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
1314: (12)       for dt in [bytes, np.void, str]:
1315: (16)         zs = self._zeros(10, dt)
1316: (16)         p = pickle.dumps(zs, protocol=proto)

```

```

1317: (16)                         assert_equal(zs.dtype, zs2.dtype)
1318: (4)  def test_pickle_empty(self):
1319: (8)      """Checking if an empty array pickled and un-pickled will not cause a
1320: (8)      segmentation fault"""
1321: (8)      arr = np.array([]).reshape(999999, 0)
1322: (8)      pk_dmp = pickle.dumps(arr)
1323: (8)      pk_load = pickle.loads(pk_dmp)
1324: (8)      assert pk_load.size == 0
1325: (4)  @pytest.mark.skipif(pickle.HIGHEST_PROTOCOL < 5,
1326: (24)          reason="requires pickle protocol 5")
1327: (4)  def test_pickle_with_bufcallback(self):
1328: (8)      array = np.arange(10)
1329: (8)      buffers = []
1330: (8)      bytes_string = pickle.dumps(array, buffer_callback=buffers.append,
1331: (36)          protocol=5)
1332: (8)      array_from_buffer = pickle.loads(bytes_string, buffers=buffers)
1333: (8)      array[0] = -1
1334: (8)      assert array_from_buffer[0] == -1, array_from_buffer[0]
1335: (0)  class TestMethods:
1336: (4)      sort_kinds = ['quicksort', 'heapsort', 'stable']
1337: (4)  def test_all_where(self):
1338: (8)      a = np.array([[True, False, True],
1339: (22)          [False, False, False],
1340: (22)          [True, True, True]])
1341: (8)      wh_full = np.array([[True, False, True],
1342: (28)          [False, False, False],
1343: (28)          [True, False, True]])
1344: (8)      wh_lower = np.array([[False],
1345: (29)          [False],
1346: (29)          [True]])
1347: (8)      for _ax in [0, None]:
1348: (12)          assert_equal(a.all(axis=_ax, where=wh_lower),
1349: (24)              np.all(a[wh_lower[:,0],:], axis=_ax))
1350: (12)          assert_equal(np.all(a, axis=_ax, where=wh_lower),
1351: (25)              a[wh_lower[:,0],:].all(axis=_ax))
1352: (8)          assert_equal(a.all(where=wh_full), True)
1353: (8)          assert_equal(np.all(a, where=wh_full), True)
1354: (8)          assert_equal(a.all(where=False), True)
1355: (8)          assert_equal(np.all(a, where=False), True)
1356: (4)  def test_any_where(self):
1357: (8)      a = np.array([[True, False, True],
1358: (22)          [False, False, False],
1359: (22)          [True, True, True]])
1360: (8)      wh_full = np.array([[False, True, False],
1361: (28)          [True, True, True],
1362: (28)          [False, False, False]])
1363: (8)      wh_middle = np.array([[False],
1364: (30)          [True],
1365: (30)          [False]])
1366: (8)      for _ax in [0, None]:
1367: (12)          assert_equal(a.any(axis=_ax, where=wh_middle),
1368: (25)              np.any(a[wh_middle[:,0],:], axis=_ax))
1369: (12)          assert_equal(np.any(a, axis=_ax, where=wh_middle),
1370: (25)              a[wh_middle[:,0],:].any(axis=_ax))
1371: (8)          assert_equal(a.any(where=wh_full), False)
1372: (8)          assert_equal(np.any(a, where=wh_full), False)
1373: (8)          assert_equal(a.any(where=False), False)
1374: (8)          assert_equal(np.any(a, where=False), False)
1375: (4)  def test_compress(self):
1376: (8)      tgt = [[5, 6, 7, 8, 9]]
1377: (8)      arr = np.arange(10).reshape(2, 5)
1378: (8)      out = arr.compress([0, 1], axis=0)
1379: (8)      assert_equal(out, tgt)
1380: (8)      tgt = [[1, 3], [6, 8]]
1381: (8)      out = arr.compress([0, 1, 0, 1, 0], axis=1)
1382: (8)      assert_equal(out, tgt)
1383: (8)      tgt = [[1], [6]]
1384: (8)      arr = np.arange(10).reshape(2, 5)
1385: (8)      out = arr.compress([0, 1], axis=1)

```

```

1386: (8)             assert_equal(out, tgt)
1387: (8)             arr = np.arange(10).reshape(2, 5)
1388: (8)             out = arr.compress([0, 1])
1389: (8)             assert_equal(out, 1)
1390: (4)             def test_choose(self):
1391: (8)                 x = 2*np.ones((3,), dtype=int)
1392: (8)                 y = 3*np.ones((3,), dtype=int)
1393: (8)                 x2 = 2*np.ones((2, 3), dtype=int)
1394: (8)                 y2 = 3*np.ones((2, 3), dtype=int)
1395: (8)                 ind = np.array([0, 0, 1])
1396: (8)                 A = ind.choose((x, y))
1397: (8)                 assert_equal(A, [2, 2, 3])
1398: (8)                 A = ind.choose((x2, y2))
1399: (8)                 assert_equal(A, [[2, 2, 3], [2, 2, 3]])
1400: (8)                 A = ind.choose((x, y2))
1401: (8)                 assert_equal(A, [[2, 2, 3], [2, 2, 3]])
1402: (8)                 oned = np.ones(1)
1403: (8)                 assert_raises(TypeError, oned.choose, np.void(0), [oned])
1404: (8)                 out = np.array(0)
1405: (8)                 ret = np.choose(np.array(1), [10, 20, 30], out=out)
1406: (8)                 assert out is ret
1407: (8)                 assert_equal(out[()], 20)
1408: (8)                 x = np.arange(5)
1409: (8)                 y = np.choose([0, 0, 0], [x[:3], x[:3], x[:3]], out=x[1:4], mode='wrap')
1410: (8)                 assert_equal(y, np.array([0, 1, 2]))
1411: (4)             def test_prod(self):
1412: (8)                 ba = [1, 2, 10, 11, 6, 5, 4]
1413: (8)                 ba2 = [[1, 2, 3, 4], [5, 6, 7, 9], [10, 3, 4, 5]]
1414: (8)                 for ctype in [np.int16, np.uint16, np.int32, np.uint32,
1415: (22)                               np.float32, np.float64, np.complex64, np.complex128]:
1416: (12)                   a = np.array(ba, ctype)
1417: (12)                   a2 = np.array(ba2, ctype)
1418: (12)                   if ctype in ['1', 'b']:
1419: (16)                     assert_raises(ArithmeticError, a.prod)
1420: (16)                     assert_raises(ArithmeticError, a2.prod, axis=1)
1421: (12)                   else:
1422: (16)                     assert_equal(a.prod(axis=0), 26400)
1423: (16)                     assert_array_equal(a2.prod(axis=0),
1424: (35)                           np.array([50, 36, 84, 180], ctype))
1425: (16)                     assert_array_equal(a2.prod(axis=-1),
1426: (35)                           np.array([24, 1890, 600], ctype))
1427: (4)             @pytest.mark.parametrize('dtype', [None, object()])
1428: (4)             def test_repeat(self, dtype):
1429: (8)                 m = np.array([1, 2, 3, 4, 5, 6], dtype=dtype)
1430: (8)                 m_rect = m.reshape((2, 3))
1431: (8)                 A = m.repeat([1, 3, 2, 1, 1, 2])
1432: (8)                 assert_equal(A, [1, 2, 2, 2, 3,
1433: (25)                               3, 4, 5, 6, 6])
1434: (8)                 A = m.repeat(2)
1435: (8)                 assert_equal(A, [1, 1, 2, 2, 3, 3,
1436: (25)                               4, 4, 5, 5, 6, 6])
1437: (8)                 A = m_rect.repeat([2, 1], axis=0)
1438: (8)                 assert_equal(A, [[1, 2, 3],
1439: (25)                               [1, 2, 3],
1440: (25)                               [4, 5, 6]])
1441: (8)                 A = m_rect.repeat([1, 3, 2], axis=1)
1442: (8)                 assert_equal(A, [[1, 2, 2, 2, 3, 3],
1443: (25)                               [4, 5, 5, 5, 6, 6]])
1444: (8)                 A = m_rect.repeat(2, axis=0)
1445: (8)                 assert_equal(A, [[1, 2, 3],
1446: (25)                               [1, 2, 3],
1447: (25)                               [4, 5, 6],
1448: (25)                               [4, 5, 6]])
1449: (8)                 A = m_rect.repeat(2, axis=1)
1450: (8)                 assert_equal(A, [[1, 1, 2, 2, 3, 3],
1451: (25)                               [4, 4, 5, 5, 6, 6]])
1452: (4)             def test_reshape(self):
1453: (8)                 arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
1454: (8)                 tgt = [[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]]

```

```

1455: (8) assert_equal(arr.reshape(2, 6), tgt)
1456: (8) tgt = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
1457: (8) assert_equal(arr.reshape(3, 4), tgt)
1458: (8) tgt = [[1, 10, 8, 6], [4, 2, 11, 9], [7, 5, 3, 12]]
1459: (8) assert_equal(arr.reshape((3, 4), order='F'), tgt)
1460: (8) tgt = [[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
1461: (8) assert_equal(arr.T.reshape((3, 4), order='C'), tgt)
1462: (4) def test_round(self):
1463: (8)     def check_round(arr, expected, *round_args):
1464: (12)         assert_equal(arr.round(*round_args), expected)
1465: (12)         out = np.zeros_like(arr)
1466: (12)         res = arr.round(*round_args, out=out)
1467: (12)         assert_equal(out, expected)
1468: (12)         assert out is res
1469: (8)     check_round(np.array([1.2, 1.5]), [1, 2])
1470: (8)     check_round(np.array(1.5), 2)
1471: (8)     check_round(np.array([12.2, 15.5]), [10, 20], -1)
1472: (8)     check_round(np.array([12.15, 15.51]), [12.2, 15.5], 1)
1473: (8)     check_round(np.array([4.5 + 1.5j]), [4 + 2j])
1474: (8)     check_round(np.array([12.5 + 15.5j]), [10 + 20j], -1)
1475: (4) def test_squeeze(self):
1476: (8)     a = np.array([[[1], [2], [3]]])
1477: (8)     assert_equal(a.squeeze(), [1, 2, 3])
1478: (8)     assert_equal(a.squeeze(axis=(0,)), [[1], [2], [3]])
1479: (8)     assert_raises(ValueError, a.squeeze, axis=(1,))
1480: (8)     assert_equal(a.squeeze(axis=(2,)), [[1, 2, 3]])
1481: (4) def test_transpose(self):
1482: (8)     a = np.array([[1, 2], [3, 4]])
1483: (8)     assert_equal(a.transpose(), [[1, 3], [2, 4]])
1484: (8)     assert_raises(ValueError, lambda: a.transpose(0))
1485: (8)     assert_raises(ValueError, lambda: a.transpose(0, 0))
1486: (8)     assert_raises(ValueError, lambda: a.transpose(0, 1, 2))
1487: (4) def test_sort(self):
1488: (8)     msg = "Test real sort order with nans"
1489: (8)     a = np.array([np.nan, 1, 0])
1490: (8)     b = np.sort(a)
1491: (8)     assert_equal(b, a[::-1], msg)
1492: (8)     msg = "Test complex sort order with nans"
1493: (8)     a = np.zeros(9, dtype=np.complex128)
1494: (8)     a.real += [np.nan, np.nan, np.nan, 1, 0, 1, 1, 0, 0]
1495: (8)     a.imag += [np.nan, 1, 0, np.nan, np.nan, 1, 0, 1, 0]
1496: (8)     b = np.sort(a)
1497: (8)     assert_equal(b, a[::-1], msg)
1498: (4) @pytest.mark.parametrize('dtype', [np.uint8, np.uint16, np.uint32,
np.uint64,
1499: (39)   np.float16, np.float32, np.float64,
1500: (39)   np.longdouble])
1501: (4) def test_sort_unsigned(self, dtype):
1502: (8)     a = np.arange(101, dtype=dtype)
1503: (8)     b = a[::-1].copy()
1504: (8)     for kind in self.sort_kinds:
1505: (12)         msg = "scalar sort, kind=%s" % kind
1506: (12)         c = a.copy()
1507: (12)         c.sort(kind=kind)
1508: (12)         assert_equal(c, a, msg)
1509: (12)         c = b.copy()
1510: (12)         c.sort(kind=kind)
1511: (12)         assert_equal(c, a, msg)
1512: (4) @pytest.mark.parametrize('dtype',
1513: (29)   [np.int8, np.int16, np.int32, np.int64,
np.float16,
1514: (30)   np.float32, np.float64, np.longdouble])
1515: (4) def test_sort_signed(self, dtype):
1516: (8)     a = np.arange(-50, 51, dtype=dtype)
1517: (8)     b = a[::-1].copy()
1518: (8)     for kind in self.sort_kinds:
1519: (12)         msg = "scalar sort, kind=%s" % (kind)
1520: (12)         c = a.copy()
1521: (12)         c.sort(kind=kind)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1522: (12) assert_equal(c, a, msg)
1523: (12) c = b.copy()
1524: (12) c.sort(kind=kind)
1525: (12) assert_equal(c, a, msg)
1526: (4) @pytest.mark.parametrize('dtype', [np.float32, np.float64, np.longdouble])
1527: (4) @pytest.mark.parametrize('part', ['real', 'imag'])
1528: (4) def test_sort_complex(self, part, dtype):
1529: (8)     ctype = {
1530: (12)         np.single: np.csingle,
1531: (12)         np.double: np.cdouble,
1532: (12)         np.longdouble: np.clongdouble,
1533: (8)     }[dtype]
1534: (8)     a = np.arange(-50, 51, dtype=dtype)
1535: (8)     b = a[::-1].copy()
1536: (8)     ai = (a * (1+1j)).astype(ctype)
1537: (8)     bi = (b * (1+1j)).astype(ctype)
1538: (8)     setattr(ai, part, 1)
1539: (8)     setattr(bi, part, 1)
1540: (8)     for kind in self.sort_kinds:
1541: (12)         msg = "complex sort, %s part == 1, kind=%s" % (part, kind)
1542: (12)         c = ai.copy()
1543: (12)         c.sort(kind=kind)
1544: (12)         assert_equal(c, ai, msg)
1545: (12)         c = bi.copy()
1546: (12)         c.sort(kind=kind)
1547: (12)         assert_equal(c, ai, msg)
1548: (4) def test_sort_complex_byte_swapping(self):
1549: (8)     for endianness in '<>':
1550: (12)         for dt in np.typecodes['Complex']:
1551: (16)             arr = np.array([1+3.j, 2+2.j, 3+1.j], dtype=endianness + dt)
1552: (16)             c = arr.copy()
1553: (16)             c.sort()
1554: (16)             msg = 'byte-swapped complex sort, dtype={0}'.format(dt)
1555: (16)             assert_equal(c, arr, msg)
1556: (4) @pytest.mark.parametrize('dtype', [np.bytes_, np.str_])
1557: (4) def test_sort_string(self, dtype):
1558: (8)     a = np.array(['aaaaaaaa' + chr(i) for i in range(101)], dtype=dtype)
1559: (8)     b = a[::-1].copy()
1560: (8)     for kind in self.sort_kinds:
1561: (12)         msg = "kind=%s" % kind
1562: (12)         c = a.copy()
1563: (12)         c.sort(kind=kind)
1564: (12)         assert_equal(c, a, msg)
1565: (12)         c = b.copy()
1566: (12)         c.sort(kind=kind)
1567: (12)         assert_equal(c, a, msg)
1568: (4) def test_sort_object(self):
1569: (8)     a = np.empty((101,), dtype=object)
1570: (8)     a[:] = list(range(101))
1571: (8)     b = a[::-1]
1572: (8)     for kind in ['q', 'h', 'm']:
1573: (12)         msg = "kind=%s" % kind
1574: (12)         c = a.copy()
1575: (12)         c.sort(kind=kind)
1576: (12)         assert_equal(c, a, msg)
1577: (12)         c = b.copy()
1578: (12)         c.sort(kind=kind)
1579: (12)         assert_equal(c, a, msg)
1580: (4) @pytest.mark.parametrize("dt", [
1581: (12)     np.dtype([('f', float), ('i', int)]),
1582: (12)     np.dtype([('f', float), ('i', object)])])
1583: (4) @pytest.mark.parametrize("step", [1, 2])
1584: (4) def test_sort_structured(self, dt, step):
1585: (8)     a = np.array([(i, i) for i in range(101*step)], dtype=dt)
1586: (8)     b = a[::-1]
1587: (8)     for kind in ['q', 'h', 'm']:
1588: (12)         msg = "kind=%s" % kind
1589: (12)         c = a.copy()[:step]
1590: (12)         idx = c.argsort(kind=kind)

```

```

1591: (12)             c.sort(kind=kind)
1592: (12)             assert_equal(c, a[::-step], msg)
1593: (12)             assert_equal(a[::-step][indx], a[::-step], msg)
1594: (12)             c = b.copy()[:step]
1595: (12)             indx = c.argsort(kind=kind)
1596: (12)             c.sort(kind=kind)
1597: (12)             assert_equal(c, a[step-1::-step], msg)
1598: (12)             assert_equal(b[::-step][indx], a[step-1::-step], msg)
1599: (4) @pytest.mark.parametrize('dtype', ['datetime64[D]', 'timedelta64[D']])
1600: (4) def test_sort_time(self, dtype):
1601: (8)     a = np.arange(0, 101, dtype=dtype)
1602: (8)     b = a[::-1]
1603: (8)     for kind in ['q', 'h', 'm']:
1604: (12)         msg = "kind=%s" % kind
1605: (12)         c = a.copy()
1606: (12)         c.sort(kind=kind)
1607: (12)         assert_equal(c, a, msg)
1608: (12)         c = b.copy()
1609: (12)         c.sort(kind=kind)
1610: (12)         assert_equal(c, a, msg)
1611: (4) def test_sort_axis(self):
1612: (8)     a = np.array([[3, 2], [1, 0]])
1613: (8)     b = np.array([[1, 0], [3, 2]])
1614: (8)     c = np.array([[2, 3], [0, 1]])
1615: (8)     d = a.copy()
1616: (8)     d.sort(axis=0)
1617: (8)     assert_equal(d, b, "test sort with axis=0")
1618: (8)     d = a.copy()
1619: (8)     d.sort(axis=1)
1620: (8)     assert_equal(d, c, "test sort with axis=1")
1621: (8)     d = a.copy()
1622: (8)     d.sort()
1623: (8)     assert_equal(d, c, "test sort with default axis")
1624: (4) def test_sort_size_0(self):
1625: (8)     a = np.array([])
1626: (8)     a.shape = (3, 2, 1, 0)
1627: (8)     for axis in range(-a.ndim, a.ndim):
1628: (12)         msg = 'test empty array sort with axis={0}'.format(axis)
1629: (12)         assert_equal(np.sort(a, axis=axis), a, msg)
1630: (8)     msg = 'test empty array sort with axis=None'
1631: (8)     assert_equal(np.sort(a, axis=None), a.ravel(), msg)
1632: (4) def test_sort_bad_ordering(self):
1633: (8)     class Boom:
1634: (12)         def __lt__(self, other):
1635: (16)             return True
1636: (8)     a = np.array([Boom()] * 100, dtype=object)
1637: (8)     for kind in self.sort_kinds:
1638: (12)         msg = "kind=%s" % kind
1639: (12)         c = a.copy()
1640: (12)         c.sort(kind=kind)
1641: (12)         assert_equal(c, a, msg)
1642: (4) def test_void_sort(self):
1643: (8)     for i in range(4):
1644: (12)         rand = np.random.randint(256, size=4000, dtype=np.uint8)
1645: (12)         arr = rand.view('V4')
1646: (12)         arr[::-1].sort()
1647: (8)     dt = np.dtype([('val', 'i4', (1,))])
1648: (8)     for i in range(4):
1649: (12)         rand = np.random.randint(256, size=4000, dtype=np.uint8)
1650: (12)         arr = rand.view(dt)
1651: (12)         arr[::-1].sort()
1652: (4) def test_sort_raises(self):
1653: (8)     arr = np.array([0, datetime.now(), 1], dtype=object)
1654: (8)     for kind in self.sort_kinds:
1655: (12)         assert_raises(TypeError, arr.sort, kind=kind)
1656: (8)     class Raiser:
1657: (12)         def raises_anything(*args, **kwargs):
1658: (16)             raise TypeError("SOMETHING ERRORRED")
1659: (12)             __eq__ = __ne__ = __lt__ = __gt__ = __ge__ = __le__ =

```

```

raisesAnything
1660: (8) arr = np.array([[Raiser(), n] for n in range(10)]).reshape(-1)
1661: (8) np.random.shuffle(arr)
1662: (8) for kind in self.sort_kinds:
1663: (12)     assert_raises(TypeError, arr.sort, kind=kind)
1664: (4) def testSortDegraded(self):
1665: (8)     d = np.arange(1000000)
1666: (8)     do = d.copy()
1667: (8)     x = d
1668: (8)     while x.size > 3:
1669: (12)         mid = x.size // 2
1670: (12)         x[mid], x[-2] = x[-2], x[mid]
1671: (12)         x = x[:-2]
1672: (8)     assert_equal(np.sort(d), do)
1673: (8)     assert_equal(d[np.argsort(d)], do)
1674: (4) def testCopy(self):
1675: (8)     def assertFortran(arr):
1676: (12)         assert_(arr.flags.fortran)
1677: (12)         assert_(arr.flags.f_contiguous)
1678: (12)         assert_(not arr.flags.c_contiguous)
1679: (8)     def assertC(arr):
1680: (12)         assert_(not arr.flags.fortran)
1681: (12)         assert_(not arr.flags.f_contiguous)
1682: (12)         assert_(arr.flags.c_contiguous)
1683: (8)     a = np.empty((2, 2), order='F')
1684: (8)     assertC(a.copy())
1685: (8)     assertC(a.copy('C'))
1686: (8)     assert_fortran(a.copy('F'))
1687: (8)     assert_fortran(a.copy('A'))
1688: (8)     a = np.empty((2, 2), order='C')
1689: (8)     assertC(a.copy())
1690: (8)     assertC(a.copy('C'))
1691: (8)     assert_fortran(a.copy('F'))
1692: (8)     assertC(a.copy('A'))
1693: (4) @pytest.mark.parametrize("dtype", ['O', np.int32, 'i,O'])
1694: (4) def testDeepcopy_(self, dtype):
1695: (8)     a = np.empty(4, dtype=dtype)
1696: (8)     ctypes.memset(a.ctypes.data, 0, a.nbytes)
1697: (8)     b = a.__deepcopy__({})
1698: (8)     a[0] = 42
1699: (8)     with pytest.raises(AssertionError):
1700: (12)         assert_array_equal(a, b)
1701: (4) def testDeepcopyCatchesFailure(self):
1702: (8)     class MyObj:
1703: (12)         def __deepcopy__(self, *args, **kwargs):
1704: (16)             raise RuntimeError
1705: (8)         arr = np.array([1, MyObj(), 3], dtype='O')
1706: (8)         with pytest.raises(RuntimeError):
1707: (12)             arr.__deepcopy__({})
1708: (4) def testSortOrder(self):
1709: (8)     x1 = np.array([21, 32, 14])
1710: (8)     x2 = np.array(['my', 'first', 'name'])
1711: (8)     x3 = np.array([3.1, 4.5, 6.2])
1712: (8)     r = np.rec.fromarrays([x1, x2, x3], names='id,word,number')
1713: (8)     r.sort(order=['id'])
1714: (8)     assert_equal(r.id, np.array([14, 21, 32]))
1715: (8)     assert_equal(r.word, np.array(['name', 'my', 'first']))
1716: (8)     assert_equal(r.number, np.array([6.2, 3.1, 4.5]))
1717: (8)     r.sort(order=['word'])
1718: (8)     assert_equal(r.id, np.array([32, 21, 14]))
1719: (8)     assert_equal(r.word, np.array(['first', 'my', 'name']))
1720: (8)     assert_equal(r.number, np.array([4.5, 3.1, 6.2]))
1721: (8)     r.sort(order=['number'])
1722: (8)     assert_equal(r.id, np.array([21, 32, 14]))
1723: (8)     assert_equal(r.word, np.array(['my', 'first', 'name']))
1724: (8)     assert_equal(r.number, np.array([3.1, 4.5, 6.2]))
1725: (8)     assert_raises_regex(ValueError, 'duplicate',
1726: (12)         lambda: r.sort(order=['id', 'id']))
1727: (8)     if sys.byteorder == 'little':

```

```

1728: (12)             strtype = '>i2'
1729: (8)              else:
1730: (12)              strtype = '<i2'
1731: (8)              mydtype = [('name', 'U5'), ('col2', strtype)]
1732: (8)              r = np.array([('a', 1), ('b', 255), ('c', 3), ('d', 258)],
1733: (21)                  dtype=mydtype)
1734: (8)              r.sort(order='col2')
1735: (8)              assert_equal(r['col2'], [1, 3, 255, 258])
1736: (8)              assert_equal(r, np.array([('a', 1), ('c', 3), ('b', 255), ('d', 258)],
1737: (33)                  dtype=mydtype))
1738: (4)               def test_argsort(self):
1739: (8)                   for dtype in [np.int32, np.uint32, np.float32]:
1740: (12)                       a = np.arange(101, dtype=dtype)
1741: (12)                       b = a[::-1].copy()
1742: (12)                       for kind in self.sort_kinds:
1743: (16)                           msg = "scalar argsort, kind=%s, dtype=%s" % (kind, dtype)
1744: (16)                           assert_equal(a.copy().argsort(kind=kind), a, msg)
1745: (16)                           assert_equal(b.copy().argsort(kind=kind), b, msg)
1746: (8)                       ai = a*1j + 1
1747: (8)                       bi = b*1j + 1
1748: (8)                       for kind in self.sort_kinds:
1749: (12)                           msg = "complex argsort, kind=%s" % kind
1750: (12)                           assert_equal(ai.copy().argsort(kind=kind), a, msg)
1751: (12)                           assert_equal(bi.copy().argsort(kind=kind), b, msg)
1752: (8)                       ai = a + 1j
1753: (8)                       bi = b + 1j
1754: (8)                       for kind in self.sort_kinds:
1755: (12)                           msg = "complex argsort, kind=%s" % kind
1756: (12)                           assert_equal(ai.copy().argsort(kind=kind), a, msg)
1757: (12)                           assert_equal(bi.copy().argsort(kind=kind), b, msg)
1758: (8)                       for endianness in '<>':
1759: (12)                           for dt in np.typecodes['Complex']:
1760: (16)                               arr = np.array([1+3.j, 2+2.j, 3+1.j], dtype=endianness + dt)
1761: (16)                               msg = 'byte-swapped complex argsort, dtype={0}'.format(dt)
1762: (16)                               assert_equal(arr.argsort(),
1763: (29)                                   np.arange(len(arr), dtype=np.intp), msg)
1764: (8)                               s = 'aaaaaaaa'
1765: (8)                               a = np.array([s + chr(i) for i in range(101)])
1766: (8)                               b = a[::-1].copy()
1767: (8)                               r = np.arange(101)
1768: (8)                               rr = r[::-1]
1769: (8)                               for kind in self.sort_kinds:
1770: (12)                                   msg = "string argsort, kind=%s" % kind
1771: (12)                                   assert_equal(a.copy().argsort(kind=kind), r, msg)
1772: (12)                                   assert_equal(b.copy().argsort(kind=kind), rr, msg)
1773: (8)                               s = 'aaaaaaaa'
1774: (8)                               a = np.array([s + chr(i) for i in range(101)], dtype=np.str_)
1775: (8)                               b = a[::-1]
1776: (8)                               r = np.arange(101)
1777: (8)                               rr = r[::-1]
1778: (8)                               for kind in self.sort_kinds:
1779: (12)                                   msg = "unicode argsort, kind=%s" % kind
1780: (12)                                   assert_equal(a.copy().argsort(kind=kind), r, msg)
1781: (12)                                   assert_equal(b.copy().argsort(kind=kind), rr, msg)
1782: (8)                               a = np.empty((101,), dtype=object)
1783: (8)                               a[:] = list(range(101))
1784: (8)                               b = a[::-1]
1785: (8)                               r = np.arange(101)
1786: (8)                               rr = r[::-1]
1787: (8)                               for kind in self.sort_kinds:
1788: (12)                                   msg = "object argsort, kind=%s" % kind
1789: (12)                                   assert_equal(a.copy().argsort(kind=kind), r, msg)
1790: (12)                                   assert_equal(b.copy().argsort(kind=kind), rr, msg)
1791: (8)                               dt = np.dtype([('f', float), ('i', int)])
1792: (8)                               a = np.array([(i, i) for i in range(101)], dtype=dt)
1793: (8)                               b = a[::-1]
1794: (8)                               r = np.arange(101)
1795: (8)                               rr = r[::-1]
1796: (8)                               for kind in self.sort_kinds:

```

```

1797: (12)                         msg = "structured array argsort, kind=%s" % kind
1798: (12)                         assert_equal(a.copy().argsort(kind=kind), r, msg)
1799: (12)                         assert_equal(b.copy().argsort(kind=kind), rr, msg)
1800: (8)                          a = np.arange(0, 101, dtype='datetime64[D]')
1801: (8)                          b = a[::-1]
1802: (8)                          r = np.arange(101)
1803: (8)                          rr = r[::-1]
1804: (8)                          for kind in ['q', 'h', 'm']:
1805: (12)                            msg = "datetime64 argsort, kind=%s" % kind
1806: (12)                            assert_equal(a.copy().argsort(kind=kind), r, msg)
1807: (12)                            assert_equal(b.copy().argsort(kind=kind), rr, msg)
1808: (8)                          a = np.arange(0, 101, dtype='timedelta64[D]')
1809: (8)                          b = a[::-1]
1810: (8)                          r = np.arange(101)
1811: (8)                          rr = r[::-1]
1812: (8)                          for kind in ['q', 'h', 'm']:
1813: (12)                            msg = "timedelta64 argsort, kind=%s" % kind
1814: (12)                            assert_equal(a.copy().argsort(kind=kind), r, msg)
1815: (12)                            assert_equal(b.copy().argsort(kind=kind), rr, msg)
1816: (8)                          a = np.array([[3, 2], [1, 0]])
1817: (8)                          b = np.array([[1, 1], [0, 0]])
1818: (8)                          c = np.array([[1, 0], [1, 0]])
1819: (8)                          assert_equal(a.copy().argsort(axis=0), b)
1820: (8)                          assert_equal(a.copy().argsort(axis=1), c)
1821: (8)                          assert_equal(a.copy().argsort(), c)
1822: (8)                          a = np.array([])
1823: (8)                          a.shape = (3, 2, 1, 0)
1824: (8)                          for axis in range(-a.ndim, a.ndim):
1825: (12)                            msg = 'test empty array argsort with axis={0}'.format(axis)
1826: (12)                            assert_equal(np.argsort(a, axis=axis),
1827: (25)                                np.zeros_like(a, dtype=np.intp), msg)
1828: (8)                          msg = 'test empty array argsort with axis=None'
1829: (8)                          assert_equal(np.argsort(a, axis=None),
1830: (21)                                np.zeros_like(a.ravel(), dtype=np.intp), msg)
1831: (8)                          r = np.arange(100)
1832: (8)                          a = np.zeros(100)
1833: (8)                          assert_equal(a.argsort(kind='m'), r)
1834: (8)                          a = np.zeros(100, dtype=complex)
1835: (8)                          assert_equal(a.argsort(kind='m'), r)
1836: (8)                          a = np.array(['aaaaaaaa' for i in range(100)])
1837: (8)                          assert_equal(a.argsort(kind='m'), r)
1838: (8)                          a = np.array(['aaaaaaaa' for i in range(100)], dtype=np.str_)
1839: (8)                          assert_equal(a.argsort(kind='m'), r)
1840: (4)                          def test_sort_unicode_kind(self):
1841: (8)                            d = np.arange(10)
1842: (8)                            k = b'\xc3\xaa'.decode("UTF8")
1843: (8)                            assert_raises(ValueError, d.sort, kind=k)
1844: (8)                            assert_raises(ValueError, d.argsort, kind=k)
1845: (4)                          @pytest.mark.parametrize('a', [
1846: (8)                            np.array([0, 1, np.nan], dtype=np.float16),
1847: (8)                            np.array([0, 1, np.nan], dtype=np.float32),
1848: (8)                            np.array([0, 1, np.nan]),
1849: (4)                          ])
1850: (4)                          def test_searchsorted_floats(self, a):
1851: (8)                            msg = "Test real (%s) searchsorted with nans, side='l'" % a.dtype
1852: (8)                            b = a.searchsorted(a, side='left')
1853: (8)                            assert_equal(b, np.arange(3), msg)
1854: (8)                            msg = "Test real (%s) searchsorted with nans, side='r'" % a.dtype
1855: (8)                            b = a.searchsorted(a, side='right')
1856: (8)                            assert_equal(b, np.arange(1, 4), msg)
1857: (8)                            a.searchsorted(v=1)
1858: (8)                            x = np.array([0, 1, np.nan], dtype='float32')
1859: (8)                            y = np.searchsorted(x, x[-1])
1860: (8)                            assert_equal(y, 2)
1861: (4)                          def test_searchsorted_complex(self):
1862: (8)                            a = np.zeros(9, dtype=np.complex128)
1863: (8)                            a.real += [0, 0, 1, 1, 0, 1, np.nan, np.nan, np.nan]
1864: (8)                            a.imag += [0, 1, 0, 1, np.nan, np.nan, 0, 1, np.nan]
1865: (8)                            msg = "Test complex searchsorted with nans, side='l'"

```

```

1866: (8)          b = a.searchsorted(a, side='left')
1867: (8)          assert_equal(b, np.arange(9), msg)
1868: (8)          msg = "Test complex searchsorted with nans, side='r'"
1869: (8)          b = a.searchsorted(a, side='right')
1870: (8)          assert_equal(b, np.arange(1, 10), msg)
1871: (8)          msg = "Test searchsorted with little endian, side='l'"
1872: (8)          a = np.array([0, 128], dtype='<i4')
1873: (8)          b = a.searchsorted(np.array(128, dtype='<i4'))
1874: (8)          assert_equal(b, 1, msg)
1875: (8)          msg = "Test searchsorted with big endian, side='l'"
1876: (8)          a = np.array([0, 128], dtype='>i4')
1877: (8)          b = a.searchsorted(np.array(128, dtype='>i4'))
1878: (8)          assert_equal(b, 1, msg)
1879: (4)          def test_searchsorted_n_elements(self):
1880: (8)          a = np.ones(0)
1881: (8)          b = a.searchsorted([0, 1, 2], 'left')
1882: (8)          assert_equal(b, [0, 0, 0])
1883: (8)          b = a.searchsorted([0, 1, 2], 'right')
1884: (8)          assert_equal(b, [0, 0, 0])
1885: (8)          a = np.ones(1)
1886: (8)          b = a.searchsorted([0, 1, 2], 'left')
1887: (8)          assert_equal(b, [0, 0, 1])
1888: (8)          b = a.searchsorted([0, 1, 2], 'right')
1889: (8)          assert_equal(b, [0, 1, 1])
1890: (8)          a = np.ones(2)
1891: (8)          b = a.searchsorted([0, 1, 2], 'left')
1892: (8)          assert_equal(b, [0, 0, 2])
1893: (8)          b = a.searchsorted([0, 1, 2], 'right')
1894: (8)          assert_equal(b, [0, 2, 2])
1895: (4)          def test_searchsorted_unaligned_array(self):
1896: (8)          a = np.arange(10)
1897: (8)          aligned = np.empty(a.itemsize * a.size + 1, 'uint8')
1898: (8)          unaligned = aligned[1:].view(a.dtype)
1899: (8)          unaligned[:] = a
1900: (8)          b = unaligned.searchsorted(a, 'left')
1901: (8)          assert_equal(b, a)
1902: (8)          b = unaligned.searchsorted(a, 'right')
1903: (8)          assert_equal(b, a + 1)
1904: (8)          b = a.searchsorted(unaligned, 'left')
1905: (8)          assert_equal(b, a)
1906: (8)          b = a.searchsorted(unaligned, 'right')
1907: (8)          assert_equal(b, a + 1)
1908: (4)          def test_searchsorted_resetting(self):
1909: (8)          a = np.arange(5)
1910: (8)          b = a.searchsorted([6, 5, 4], 'left')
1911: (8)          assert_equal(b, [5, 5, 4])
1912: (8)          b = a.searchsorted([6, 5, 4], 'right')
1913: (8)          assert_equal(b, [5, 5, 5])
1914: (4)          def test_searchsorted_type_specific(self):
1915: (8)          types = ''.join((np.typecodes['AllInteger'], np.typecodes['AllFloat'],
1916: (25)                      np.typecodes['Datetime'], '?0'))
1917: (8)          for dt in types:
1918: (12)            if dt == 'M':
1919: (16)              dt = 'M8[D]'
1920: (12)            if dt == '?':
1921: (16)              a = np.arange(2, dtype=dt)
1922: (16)              out = np.arange(2)
1923: (12)            else:
1924: (16)              a = np.arange(0, 5, dtype=dt)
1925: (16)              out = np.arange(5)
1926: (12)            b = a.searchsorted(a, 'left')
1927: (12)            assert_equal(b, out)
1928: (12)            b = a.searchsorted(a, 'right')
1929: (12)            assert_equal(b, out + 1)
1930: (12)            e = np.ndarray(shape=0, buffer=b'', dtype=dt)
1931: (12)            b = e.searchsorted(a, 'left')
1932: (12)            assert_array_equal(b, np.zeros(len(a), dtype=np.intp))
1933: (12)            b = a.searchsorted(e, 'left')
1934: (12)            assert_array_equal(b, np.zeros(0, dtype=np.intp))

```

```

1935: (4)
1936: (8)
1937: (22)
1938: (22)
1939: (22)
1940: (22)
1941: (22)
1942: (22)
1943: (22)
1944: (22)
1945: (22)
1946: (22)
1947: (22)
1948: (22)
1949: (22)
1950: (21)
1951: (8)
1952: (8)
1953: (8)
1954: (8)
1955: (8)
1956: (4)
1957: (8)
1958: (8)
1959: (8)
1960: (22)
1961: (8)
1962: (8)
1963: (8)
5, 6])
1964: (8)
5])
1965: (8)
3])
1966: (8)
3])
1967: (4)
1968: (8)
1969: (8)
1970: (8)
1971: (8)
1972: (8)
1973: (8)
1974: (8)
1975: (8)
1976: (8)
1977: (8)
1978: (8)
1979: (8)
1980: (8)
1981: (8)
1982: (8)
1983: (8)
1984: (8)
1985: (8)
1986: (8)
1987: (8)
1988: (8)
1989: (8)
1990: (8)
1991: (8)
1992: (8)
1993: (8)
1994: (8)
1995: (8)
1996: (8)
1997: (25)
1998: (8)
1999: (12)

def test_searchsorted_unicode(self):
    a = np.array(['P:\\20x_dapi_cy3\\20x_dapi_cy3_20100185_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100186_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100187_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100189_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100190_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100191_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100192_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100193_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100194_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100195_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100196_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100197_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100198_1',
                  'P:\\20x_dapi_cy3\\20x_dapi_cy3_20100199_1'],
                 dtype=np.str_)
    ind = np.arange(len(a))
    assert_equal([a.searchsorted(v, 'left') for v in a], ind)
    assert_equal([a.searchsorted(v, 'right') for v in a], ind + 1)
    assert_equal([a.searchsorted(a[i], 'left') for i in ind], ind)
    assert_equal([a.searchsorted(a[i], 'right') for i in ind], ind + 1)

def test_searchsorted_with_invalid_sorter(self):
    a = np.array([5, 2, 1, 3, 4])
    s = np.argsort(a)
    assert_raises(TypeError, np.searchsorted, a, 0,
                  sorter=np.array((1, (2, 3)), dtype=object))
    assert_raises(TypeError, np.searchsorted, a, 0, sorter=[1, 1])
    assert_raises(ValueError, np.searchsorted, a, 0, sorter=[1, 2, 3, 4])
    assert_raises(ValueError, np.searchsorted, a, 0, sorter=[1, 2, 3, 4,
5, 6])
    assert_raises(ValueError, np.searchsorted, a, 4, sorter=[0, 1, 2, 3,
5])
    assert_raises(ValueError, np.searchsorted, a, 0, sorter=[-1, 0, 1, 2,
3])
    assert_raises(ValueError, np.searchsorted, a, 0, sorter=[4, 0, -1, 2,
3])

def test_searchsorted_with_sorter(self):
    a = np.random.rand(300)
    s = a.argsort()
    b = np.sort(a)
    k = np.linspace(0, 1, 20)
    assert_equal(b.searchsorted(k), a.searchsorted(k, sorter=s))
    a = np.array([0, 1, 2, 3, 5]*20)
    s = a.argsort()
    k = [0, 1, 2, 3, 5]
    expected = [0, 20, 40, 60, 80]
    assert_equal(a.searchsorted(k, side='left', sorter=s), expected)
    expected = [20, 40, 60, 80, 100]
    assert_equal(a.searchsorted(k, side='right', sorter=s), expected)
    keys = np.arange(10)
    a = keys.copy()
    np.random.shuffle(s)
    s = a.argsort()
    aligned = np.empty(a.itemsize * a.size + 1, 'uint8')
    unaligned = aligned[1:].view(a.dtype)
    unaligned[:] = a
    b = unaligned.searchsorted(keys, 'left', s)
    assert_equal(b, keys)
    b = unaligned.searchsorted(keys, 'right', s)
    assert_equal(b, keys + 1)
    unaligned[:] = keys
    b = a.searchsorted(unaligned, 'left', s)
    assert_equal(b, keys)
    b = a.searchsorted(unaligned, 'right', s)
    assert_equal(b, keys + 1)
    types = ''.join((np.typecodes['AllInteger'], np.typecodes['AllFloat'],
                     np.typecodes['Datetime'], '?0')))
    for dt in types:
        if dt == 'M':

```

```

2000: (16)           dt = 'M8[D]'
2001: (12)          if dt == '?':
2002: (16)            a = np.array([1, 0], dtype=dt)
2003: (16)            s = np.array([1, 0], dtype=np.int16)
2004: (16)            out = np.array([1, 0])
2005: (12)          else:
2006: (16)            a = np.array([3, 4, 1, 2, 0], dtype=dt)
2007: (16)            s = np.array([4, 2, 3, 0, 1], dtype=np.int16)
2008: (16)            out = np.array([3, 4, 1, 2, 0], dtype=np.intp)
2009: (12)          b = a.searchsorted(a, 'left', s)
2010: (12)          assert_equal(b, out)
2011: (12)          b = a.searchsorted(a, 'right', s)
2012: (12)          assert_equal(b, out + 1)
2013: (12)          e = np.ndarray(shape=0, buffer=b'', dtype=dt)
2014: (12)          b = e.searchsorted(a, 'left', s[:0])
2015: (12)          assert_array_equal(b, np.zeros(len(a), dtype=np.intp))
2016: (12)          b = a.searchsorted(e, 'left', s)
2017: (12)          assert_array_equal(b, np.zeros(0, dtype=np.intp))
2018: (8)           a = np.array([3, 4, 1, 2, 0])
2019: (8)           srt = np.empty((10,), dtype=np.intp)
2020: (8)           srt[1::2] = -1
2021: (8)           srt[::2] = [4, 2, 3, 0, 1]
2022: (8)           s = srt[::2]
2023: (8)           out = np.array([3, 4, 1, 2, 0], dtype=np.intp)
2024: (8)           b = a.searchsorted(a, 'left', s)
2025: (8)           assert_equal(b, out)
2026: (8)           b = a.searchsorted(a, 'right', s)
2027: (8)           assert_equal(b, out + 1)
2028: (4)            def test_searchsorted_return_type(self):
2029: (8)              class A(np.ndarray):
2030: (12)                  pass
2031: (8)                  a = np.arange(5).view(A)
2032: (8)                  b = np.arange(1, 3).view(A)
2033: (8)                  s = np.arange(5).view(A)
2034: (8)                  assert_(not isinstance(a.searchsorted(b, 'left'), A))
2035: (8)                  assert_(not isinstance(a.searchsorted(b, 'right'), A))
2036: (8)                  assert_(not isinstance(a.searchsorted(b, 'left', s), A))
2037: (8)                  assert_(not isinstance(a.searchsorted(b, 'right', s), A))
2038: (4)                  @pytest.mark.parametrize("dtype", np.typecodes["All"])
2039: (4)                  def test_argpartition_out_of_range(self, dtype):
2040: (8)                      d = np.arange(10).astype(dtype=dtype)
2041: (8)                      assert_raises(ValueError, d.argmax, 10)
2042: (8)                      assert_raises(ValueError, d.argmax, -11)
2043: (4)                  @pytest.mark.parametrize("dtype", np.typecodes["All"])
2044: (4)                  def test_partition_out_of_range(self, dtype):
2045: (8)                      d = np.arange(10).astype(dtype=dtype)
2046: (8)                      assert_raises(ValueError, d.partition, 10)
2047: (8)                      assert_raises(ValueError, d.partition, -11)
2048: (4)                  def test_argpartition_integer(self):
2049: (8)                      d = np.arange(10)
2050: (8)                      assert_raises(TypeError, d.argmax, 9.)
2051: (8)                      d_obj = np.arange(10, dtype=object)
2052: (8)                      assert_raises(TypeError, d_obj.argmax, 9.)
2053: (4)                  def test_partition_integer(self):
2054: (8)                      d = np.arange(10)
2055: (8)                      assert_raises(TypeError, d.partition, 9.)
2056: (8)                      d_obj = np.arange(10, dtype=object)
2057: (8)                      assert_raises(TypeError, d_obj.partition, 9.)
2058: (4)                  @pytest.mark.parametrize("kth_dtype", np.typecodes["AllInteger"])
2059: (4)                  def test_partition_empty_array(self, kth_dtype):
2060: (8)                      kth = np.array(0, dtype=kth_dtype)[()]
2061: (8)                      a = np.array(())
2062: (8)                      a.shape = (3, 2, 1, 0)
2063: (8)                      for axis in range(-a.ndim, a.ndim):
2064: (12)                        msg = 'test empty array partition with axis={0}'.format(axis)
2065: (12)                        assert_equal(np.partition(a, kth, axis=axis), a, msg)
2066: (8)                        msg = 'test empty array partition with axis=None'
2067: (8)                        assert_equal(np.partition(a, kth, axis=None), a.ravel(), msg)
2068: (4)                  @pytest.mark.parametrize("kth_dtype", np.typecodes["AllInteger"])

```

```

2069: (4)             def test_argpartition_empty_array(self, kth_dtype):
2070: (8)                 kth = np.array(0, dtype=kth_dtype)[()]
2071: (8)                 a = np.array(())
2072: (8)                 a.shape = (3, 2, 1, 0)
2073: (8)                 for axis in range(-a.ndim, a.ndim):
2074: (12)                     msg = 'test empty array argpartition with axis={0}'.format(axis)
2075: (12)                     assert_equal(np.partition(a, kth, axis=axis),
2076: (25)                         np.zeros_like(a, dtype=np.intp), msg)
2077: (8)                     msg = 'test empty array argpartition with axis=None'
2078: (8)                     assert_equal(np.partition(a, kth, axis=None),
2079: (21)                         np.zeros_like(a.ravel(), dtype=np.intp), msg)
2080: (4)             def test_partition(self):
2081: (8)                 d = np.arange(10)
2082: (8)                 assert_raises(TypeError, np.partition, d, 2, kind=1)
2083: (8)                 assert_raises(ValueError, np.partition, d, 2, kind="nonsense")
2084: (8)                 assert_raises(ValueError, np.argpartition, d, 2, kind="nonsense")
2085: (8)                 assert_raises(ValueError, d.partition, 2, axis=0, kind="nonsense")
2086: (8)                 assert_raises(ValueError, d.argpartition, 2, axis=0, kind="nonsense")
2087: (8)             for k in ("introselect",):
2088: (12)                 d = np.array(())
2089: (12)                 assert_array_equal(np.partition(d, 0, kind=k), d)
2090: (12)                 assert_array_equal(np.argpartition(d, 0, kind=k), d)
2091: (12)                 d = np.ones(1)
2092: (12)                 assert_array_equal(np.partition(d, 0, kind=k)[0], d)
2093: (12)                 assert_array_equal(d[np.argpartition(d, 0, kind=k)],
2094: (31)                     np.partition(d, 0, kind=k))
2095: (12)                 kth = np.array([30, 15, 5])
2096: (12)                 okth = kth.copy()
2097: (12)                 np.partition(np.arange(40), kth)
2098: (12)                 assert_array_equal(kth, okth)
2099: (12)                 for r in ([2, 1], [1, 2], [1, 1]):
2100: (16)                     d = np.array(r)
2101: (16)                     tgt = np.sort(d)
2102: (16)                     assert_array_equal(np.partition(d, 0, kind=k)[0], tgt[0])
2103: (16)                     assert_array_equal(np.partition(d, 1, kind=k)[1], tgt[1])
2104: (16)                     assert_array_equal(d[np.argpartition(d, 0, kind=k)],
2105: (35)                         np.partition(d, 0, kind=k))
2106: (16)                     assert_array_equal(d[np.argpartition(d, 1, kind=k)],
2107: (35)                         np.partition(d, 1, kind=k))
2108: (16)                     for i in range(d.size):
2109: (20)                         d[i:].partition(0, kind=k)
2110: (16)                         assert_array_equal(d, tgt)
2111: (12)                     for r in ([3, 2, 1], [1, 2, 3], [2, 1, 3], [2, 3, 1],
2112: (22)                         [1, 1, 1], [1, 2, 2], [2, 2, 1], [1, 2, 1]):
2113: (16)                         d = np.array(r)
2114: (16)                         tgt = np.sort(d)
2115: (16)                         assert_array_equal(np.partition(d, 0, kind=k)[0], tgt[0])
2116: (16)                         assert_array_equal(np.partition(d, 1, kind=k)[1], tgt[1])
2117: (16)                         assert_array_equal(np.partition(d, 2, kind=k)[2], tgt[2])
2118: (16)                         assert_array_equal(d[np.argpartition(d, 0, kind=k)],
2119: (35)                             np.partition(d, 0, kind=k))
2120: (16)                         assert_array_equal(d[np.argpartition(d, 1, kind=k)],
2121: (35)                             np.partition(d, 1, kind=k))
2122: (16)                         assert_array_equal(d[np.argpartition(d, 2, kind=k)],
2123: (35)                             np.partition(d, 2, kind=k))
2124: (16)                         for i in range(d.size):
2125: (20)                             d[i:].partition(0, kind=k)
2126: (16)                             assert_array_equal(d, tgt)
2127: (12)                         d = np.ones(50)
2128: (12)                         assert_array_equal(np.partition(d, 0, kind=k), d)
2129: (12)                         assert_array_equal(d[np.argpartition(d, 0, kind=k)],
2130: (31)                             np.partition(d, 0, kind=k))
2131: (12)                         d = np.arange(49)
2132: (12)                         assert_equal(np.partition(d, 5, kind=k)[5], 5)
2133: (12)                         assert_equal(np.partition(d, 15, kind=k)[15], 15)
2134: (12)                         assert_array_equal(d[np.argpartition(d, 5, kind=k)],
2135: (31)                             np.partition(d, 5, kind=k))
2136: (12)                         assert_array_equal(d[np.argpartition(d, 15, kind=k)],
2137: (31)                             np.partition(d, 15, kind=k))

```

```

2138: (12)             d = np.arange(47)[::-1]
2139: (12)             assert_equal(np.partition(d, 6, kind=k)[6], 6)
2140: (12)             assert_equal(np.partition(d, 16, kind=k)[16], 16)
2141: (12)             assert_array_equal(d[np.argpartition(d, 6, kind=k)],
2142:                               np.partition(d, 6, kind=k))
2143: (12)             assert_array_equal(d[np.argpartition(d, 16, kind=k)],
2144:                               np.partition(d, 16, kind=k))
2145: (12)             assert_array_equal(np.partition(d, -6, kind=k),
2146:                               np.partition(d, 41, kind=k))
2147: (12)             assert_array_equal(np.partition(d, -16, kind=k),
2148:                               np.partition(d, 31, kind=k))
2149: (12)             assert_array_equal(d[np.argpartition(d, -6, kind=k)],
2150:                               np.partition(d, 41, kind=k))
2151: (12)             d = np.arange(1000000)
2152: (12)             x = np.roll(d, d.size // 2)
2153: (12)             mid = x.size // 2 + 1
2154: (12)             assert_equal(np.partition(x, mid)[mid], mid)
2155: (12)             d = np.arange(1000001)
2156: (12)             x = np.roll(d, d.size // 2 + 1)
2157: (12)             mid = x.size // 2 + 1
2158: (12)             assert_equal(np.partition(x, mid)[mid], mid)
2159: (12)             d = np.ones(10)
2160: (12)             d[1] = 4
2161: (12)             assert_equal(np.partition(d, (2, -1))[-1], 4)
2162: (12)             assert_equal(np.partition(d, (2, -1))[2], 1)
2163: (12)             assert_equal(d[np.argpartition(d, (2, -1))][-1], 4)
2164: (12)             assert_equal(d[np.argpartition(d, (2, -1))][2], 1)
2165: (12)             d[1] = np.nan
2166: (12)             assert_(np.isnan(d[np.argpartition(d, (2, -1))][-1]))
2167: (12)             assert_(np.isnan(np.partition(d, (2, -1))[-1]))
2168: (12)             d = np.arange(47) % 7
2169: (12)             tgt = np.sort(np.arange(47) % 7)
2170: (12)             np.random.shuffle(d)
2171: (12)             for i in range(d.size):
2172: (16)                 assert_equal(np.partition(d, i, kind=k)[i], tgt[i])
2173: (12)             assert_array_equal(d[np.argpartition(d, 6, kind=k)],
2174:                               np.partition(d, 6, kind=k))
2175: (12)             assert_array_equal(d[np.argpartition(d, 16, kind=k)],
2176:                               np.partition(d, 16, kind=k))
2177: (12)             for i in range(d.size):
2178: (16)                 d[i:].partition(0, kind=k)
2179: (12)             assert_array_equal(d, tgt)
2180: (12)             d = np.array([0, 1, 2, 3, 4, 5, 7, 7, 7, 7, 7, 7,
2181: (26)                         7, 7, 7, 7, 7, 9])
2182: (12)             kth = [0, 3, 19, 20]
2183: (12)             assert_equal(np.partition(d, kth, kind=k)[kth], (0, 3, 7, 7))
2184: (12)             assert_equal(d[np.argpartition(d, kth, kind=k)][kth], (0, 3, 7,
7))
2185: (12)             d = np.array([2, 1])
2186: (12)             d.partition(0, kind=k)
2187: (12)             assert_raises(ValueError, d.partition, 2)
2188: (12)             assert_raises(np.AxisError, d.partition, 3, axis=1)
2189: (12)             assert_raises(ValueError, np.partition, d, 2)
2190: (12)             assert_raises(np.AxisError, np.partition, d, 2, axis=1)
2191: (12)             assert_raises(ValueError, d.argpartition, 2)
2192: (12)             assert_raises(np.AxisError, d.argpartition, 3, axis=1)
2193: (12)             assert_raises(ValueError, np.argpartition, d, 2)
2194: (12)             assert_raises(np.AxisError, np.argpartition, d, 2, axis=1)
2195: (12)             d = np.arange(10).reshape((2, 5))
2196: (12)             d.partition(1, axis=0, kind=k)
2197: (12)             d.partition(4, axis=1, kind=k)
2198: (12)             np.partition(d, 1, axis=0, kind=k)
2199: (12)             np.partition(d, 4, axis=1, kind=k)
2200: (12)             np.partition(d, 1, axis=None, kind=k)
2201: (12)             np.partition(d, 9, axis=None, kind=k)
2202: (12)             d.argpartition(1, axis=0, kind=k)
2203: (12)             d.argpartition(4, axis=1, kind=k)
2204: (12)             np.argpartition(d, 1, axis=0, kind=k)
2205: (12)             np.argpartition(d, 4, axis=1, kind=k)

```

```

2206: (12) np.argpartition(d, 1, axis=None, kind=k)
2207: (12) np.argpartition(d, 9, axis=None, kind=k)
2208: (12) assert_raises(ValueError, d.partition, 2, axis=0)
2209: (12) assert_raises(ValueError, d.partition, 11, axis=1)
2210: (12) assert_raises(TypeError, d.partition, 2, axis=None)
2211: (12) assert_raises(ValueError, np.partition, d, 9, axis=1)
2212: (12) assert_raises(ValueError, np.partition, d, 11, axis=None)
2213: (12) assert_raises(ValueError, d.argmax, 2, axis=0)
2214: (12) assert_raises(ValueError, d.argmax, 11, axis=1)
2215: (12) assert_raises(ValueError, np.argmax, d, 9, axis=1)
2216: (12) assert_raises(ValueError, np.argmax, d, 11, axis=None)
2217: (12) td = [(dt, s) for dt in [np.int32, np.float32, np.complex64]
2218: (18)         for s in (9, 16)]
2219: (12) for dt, s in td:
2220: (16)     aae = assert_array_equal
2221: (16)     at = assert_
2222: (16)     d = np.arange(s, dtype=dt)
2223: (16)     np.random.shuffle(d)
2224: (16)     d1 = np.tile(np.arange(s, dtype=dt), (4, 1))
2225: (16)     map(np.random.shuffle, d1)
2226: (16)     d0 = np.transpose(d1)
2227: (16)     for i in range(d.size):
2228: (20)         p = np.partition(d, i, kind=k)
2229: (20)         assert_equal(p[i], i)
2230: (20)         assert_array_less(p[:i], p[i])
2231: (20)         assert_array_less(p[i], p[i + 1:])
2232: (20)         aae(p, d[np.argmax(d, i, kind=k)])
2233: (20)         p = np.partition(d1, i, axis=1, kind=k)
2234: (20)         aae(p[:, i], np.array([i] * d1.shape[0], dtype=dt))
2235: (20)         at((p[:, :i].T <= p[:, i]).all(),
2236: (23)             msg="%d: %r <= %r" % (i, p[:, i], p[:, :i].T))
2237: (20)         at((p[:, i + 1:].T > p[:, i]).all(),
2238: (23)             msg="%d: %r < %r" % (i, p[:, i], p[:, i + 1:].T))
2239: (20)         aae(p, d1[np.arange(d1.shape[0])[:, None],
2240: (24)             np.argmax(d1, i, axis=1, kind=k)])
2241: (20)         p = np.partition(d0, i, axis=0, kind=k)
2242: (20)         aae(p[i, :], np.array([i] * d1.shape[0], dtype=dt))
2243: (20)         at((p[:i, :] <= p[i, :]).all(),
2244: (23)             msg="%d: %r <= %r" % (i, p[i, :], p[:i, :]))
2245: (20)         at((p[i + 1:, :] > p[i, :]).all(),
2246: (23)             msg="%d: %r < %r" % (i, p[i, :], p[:, i + 1:]))
2247: (20)         aae(p, d0[np.argmax(d0, i, axis=0, kind=k),
2248: (24)             np.arange(d0.shape[1])[None, :]])
2249: (20)         dc = d.copy()
2250: (20)         dc.partition(i, kind=k)
2251: (20)         assert_equal(dc, np.partition(d, i, kind=k))
2252: (20)         dc = d0.copy()
2253: (20)         dc.partition(i, axis=0, kind=k)
2254: (20)         assert_equal(dc, np.partition(d0, i, axis=0, kind=k))
2255: (20)         dc = d1.copy()
2256: (20)         dc.partition(i, axis=1, kind=k)
2257: (20)         assert_equal(dc, np.partition(d1, i, axis=1, kind=k))
2258: (4) def assert_partitioned(self, d, kth):
2259: (8)     prev = 0
2260: (8)     for k in np.sort(kth):
2261: (12)         assert_array_less(d[prev:k], d[k], err_msg='kth %d' % k)
2262: (12)         assert_((d[k:] >= d[k]).all(),
2263: (20)             msg="kth %d, %r not greater equal %d" % (k, d[k:], d[k]))
2264: (12)         prev = k + 1
2265: (4) def test_partition_iterative(self):
2266: (12)     d = np.arange(17)
2267: (12)     kth = (0, 1, 2, 429, 231)
2268: (12)     assert_raises(ValueError, d.partition, kth)
2269: (12)     assert_raises(ValueError, d.argmax, kth)
2270: (12)     d = np.arange(10).reshape((2, 5))
2271: (12)     assert_raises(ValueError, d.partition, kth, axis=0)
2272: (12)     assert_raises(ValueError, d.partition, kth, axis=1)
2273: (12)     assert_raises(ValueError, np.partition, d, kth, axis=1)
2274: (12)     assert_raises(ValueError, np.partition, d, kth, axis=None)

```

```

2275: (12)             d = np.array([3, 4, 2, 1])
2276: (12)             p = np.partition(d, (0, 3))
2277: (12)             self.assert_partitioned(p, (0, 3))
2278: (12)             self.assert_partitioned(d[np.argpartition(d, (0, 3))], (0, 3))
2279: (12)             assert_array_equal(p, np.partition(d, (-3, -1)))
2280: (12)             assert_array_equal(p, d[np.argpartition(d, (-3, -1))])
2281: (12)             d = np.arange(17)
2282: (12)             np.random.shuffle(d)
2283: (12)             d.partition(range(d.size))
2284: (12)             assert_array_equal(np.arange(17), d)
2285: (12)             np.random.shuffle(d)
2286: (12)             assert_array_equal(np.arange(17),
d[d.argpartition(range(d.size))])
2287: (12)             d = np.arange(17)
2288: (12)             np.random.shuffle(d)
2289: (12)             keys = np.array([1, 3, 8, -2])
2290: (12)             np.random.shuffle(d)
2291: (12)             p = np.partition(d, keys)
2292: (12)             self.assert_partitioned(p, keys)
2293: (12)             p = d[np.argpartition(d, keys)]
2294: (12)             self.assert_partitioned(p, keys)
2295: (12)             np.random.shuffle(keys)
2296: (12)             assert_array_equal(np.partition(d, keys), p)
2297: (12)             assert_array_equal(d[np.argpartition(d, keys)], p)
2298: (12)             d = np.arange(20)[::-1]
2299: (12)             self.assert_partitioned(np.partition(d, [5]*4), [5])
2300: (12)             self.assert_partitioned(np.partition(d, [5]*4 + [6, 13]),
[5]*4 + [6, 13])
2301: (36)             self.assert_partitioned(d[np.argpartition(d, [5]*4)], [5])
2302: (12)             self.assert_partitioned(d[np.argpartition(d, [5]*4 + [6, 13])],
[5]*4 + [6, 13])
2303: (12)
2304: (36)
2305: (12)             d = np.arange(12)
2306: (12)             np.random.shuffle(d)
2307: (12)             d1 = np.tile(np.arange(12), (4, 1))
2308: (12)             map(np.random.shuffle, d1)
2309: (12)             d0 = np.transpose(d1)
2310: (12)             kth = (1, 6, 7, -1)
2311: (12)             p = np.partition(d1, kth, axis=1)
2312: (12)             pa = d1[np.arange(d1.shape[0])[:, None],
d1.argmax(kth, axis=1)]
2313: (20)             assert_array_equal(p, pa)
2314: (12)
2315: (12)             for i in range(d1.shape[0]):
2316: (16)                 self.assert_partitioned(p[i, :], kth)
2317: (12)             p = np.partition(d0, kth, axis=0)
2318: (12)             pa = d0[np.argmax(d0, kth, axis=0),
np.arange(d0.shape[1])[None, :]]
2319: (20)
2320: (12)
2321: (12)
2322: (16)
2323: (4)             self.assert_partitioned(p[:, i], kth)
def test_partition_cdtype(self):
2324: (8)             d = np.array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
2325: (19)                         ('Lancelot', 1.9, 38)],
2326: (18)                         dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
2327: (8)             tgt = np.sort(d, order=['age', 'height'])
2328: (8)             assert_array_equal(np.partition(d, range(d.size)),
2329: (40)                             order=['age', 'height']),
2330: (27)                             tgt)
2331: (8)             assert_array_equal(d[np.argpartition(d, range(d.size)),
2332: (45)                             order=['age', 'height'])],
2333: (27)                             tgt)
2334: (8)             for k in range(d.size):
2335: (12)                 assert_equal(np.partition(d, k, order=['age', 'height'])[k],
2336: (24)                               tgt[k])
2337: (12)                 assert_equal(d[np.argpartition(d, k, order=['age', 'height'])][k],
2338: (25)                               tgt[k])
2339: (8)             d = np.array(['Galahad', 'Arthur', 'zebra', 'Lancelot'])
2340: (8)             tgt = np.sort(d)
2341: (8)             assert_array_equal(np.partition(d, range(d.size)), tgt)
2342: (8)             for k in range(d.size):

```

```

2343: (12)             assert_equal(np.partition(d, k)[k], tgt[k])
2344: (12)             assert_equal(d[np.argpartition(d, k)][k], tgt[k])
2345: (4)  def test_partition_unicode_kind(self):
2346: (8)      d = np.arange(10)
2347: (8)      k = b'\xc3\xaa'.decode("UTF8")
2348: (8)      assert_raises(ValueError, d.partition, 2, kind=k)
2349: (8)      assert_raises(ValueError, d.argpartition, 2, kind=k)
2350: (4)  def test_partition_fuzz(self):
2351: (8)      for j in range(10, 30):
2352: (12)          for i in range(1, j - 2):
2353: (16)              d = np.arange(j)
2354: (16)              np.random.shuffle(d)
2355: (16)              d = d % np.random.randint(2, 30)
2356: (16)              idx = np.random.randint(d.size)
2357: (16)              kth = [0, idx, i, i + 1]
2358: (16)              tgt = np.sort(d)[kth]
2359: (16)              assert_array_equal(np.partition(d, kth)[kth], tgt,
2360: (35)                           err_msg="data: %r\n kth: %r" % (d, kth))
2361: (4) @pytest.mark.parametrize("kth_dtype", np.typecodes["AllInteger"])
2362: (4)  def test_argpartition_gh5524(self, kth_dtype):
2363: (8)      kth = np.array(1, dtype=kth_dtype)[()]
2364: (8)      d = [6, 7, 3, 2, 9, 0]
2365: (8)      p = np.argpartition(d, kth)
2366: (8)      self.assert_partitioned(np.array(d)[p],[1])
2367: (4)  def test_flatten(self):
2368: (8)      x0 = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
2369: (8)      x1 = np.array([[1, 2], [3, 4], [[5, 6], [7, 8]]], np.int32)
2370: (8)      y0 = np.array([1, 2, 3, 4, 5, 6], np.int32)
2371: (8)      y0f = np.array([1, 4, 2, 5, 3, 6], np.int32)
2372: (8)      y1 = np.array([1, 2, 3, 4, 5, 6, 7, 8], np.int32)
2373: (8)      y1f = np.array([1, 5, 3, 7, 2, 6, 4, 8], np.int32)
2374: (8)      assert_equal(x0.flatten(), y0)
2375: (8)      assert_equal(x0.flatten('F'), y0f)
2376: (8)      assert_equal(x0.flatten('F'), x0.T.flatten())
2377: (8)      assert_equal(x1.flatten(), y1)
2378: (8)      assert_equal(x1.flatten('F'), y1f)
2379: (8)      assert_equal(x1.flatten('F'), x1.T.flatten())
2380: (4) @pytest.mark.parametrize('func', (np.dot, np.matmul))
2381: (4)  def test_arr_mult(self, func):
2382: (8)      a = np.array([[1, 0], [0, 1]])
2383: (8)      b = np.array([[0, 1], [1, 0]])
2384: (8)      c = np.array([[9, 1], [1, -9]])
2385: (8)      d = np.arange(24).reshape(4, 6)
2386: (8)      ddt = np.array(
2387: (12)          [[ 55, 145, 235, 325],
2388: (13)          [ 145, 451, 757, 1063],
2389: (13)          [ 235, 757, 1279, 1801],
2390: (13)          [ 325, 1063, 1801, 2539]]
2391: (8)      )
2392: (8)      dtd = np.array(
2393: (12)          [[504, 540, 576, 612, 648, 684],
2394: (13)          [540, 580, 620, 660, 700, 740],
2395: (13)          [576, 620, 664, 708, 752, 796],
2396: (13)          [612, 660, 708, 756, 804, 852],
2397: (13)          [648, 700, 752, 804, 856, 908],
2398: (13)          [684, 740, 796, 852, 908, 964]]
2399: (8)      )
2400: (8)      for et in [np.float32, np.float64, np.complex64, np.complex128]:
2401: (12)          eaf = a.astype(et)
2402: (12)          assert_equal(func(eaf, eaf), eaf)
2403: (12)          assert_equal(func(eaf.T, eaf), eaf)
2404: (12)          assert_equal(func(eaf, eaf.T), eaf)
2405: (12)          assert_equal(func(eaf.T, eaf.T), eaf)
2406: (12)          assert_equal(func(eaf.T.copy(), eaf), eaf)
2407: (12)          assert_equal(func(eaf, eaf.T.copy()), eaf)
2408: (12)          assert_equal(func(eaf.T.copy(), eaf.T.copy()), eaf)
2409: (8)          for et in [np.float32, np.float64, np.complex64, np.complex128]:
2410: (12)              eaf = a.astype(et)
2411: (12)              ebf = b.astype(et)

```

```

2412: (12)             assert_equal(func(ebf, ebf), eaf)
2413: (12)             assert_equal(func(ebf.T, ebf), eaf)
2414: (12)             assert_equal(func(ebf, ebf.T), eaf)
2415: (12)             assert_equal(func(ebf.T, ebf.T), eaf)
2416: (8)              for et in [np.float32, np.float64, np.complex64, np.complex128]:
2417: (12)                  edf = d.astype(et)
2418: (12)                  assert_equal(
2419: (16)                      func(edf[::-1, :], edf.T),
2420: (16)                      func(edf[::-1, :].copy(), edf.T.copy()))
2421: (12)                  )
2422: (12)                  assert_equal(
2423: (16)                      func(edf[:, ::-1], edf.T),
2424: (16)                      func(edf[:, ::-1].copy(), edf.T.copy()))
2425: (12)                  )
2426: (12)                  assert_equal(
2427: (16)                      func(edf, edf[::-1, :].T),
2428: (16)                      func(edf, edf[::-1, :].T.copy()))
2429: (12)                  )
2430: (12)                  assert_equal(
2431: (16)                      func(edf, edf[:, ::-1].T),
2432: (16)                      func(edf, edf[:, ::-1].T.copy()))
2433: (12)                  )
2434: (12)                  assert_equal(
2435: (16)                      func(edf[:edf.shape[0] // 2, :], edf[::-2, :].T),
2436: (16)                      func(edf[:edf.shape[0] // 2, :].copy(), edf[::-2, :].T.copy()))
2437: (12)                  )
2438: (12)                  assert_equal(
2439: (16)                      func(edf[::-2, :], edf[:edf.shape[0] // 2, :].T),
2440: (16)                      func(edf[::-2, :].copy(), edf[:edf.shape[0] // 2, :].T.copy()))
2441: (12)                  )
2442: (8)              for et in [np.float32, np.float64, np.complex64, np.complex128]:
2443: (12)                  edf = d.astype(et)
2444: (12)                  eddtf = ddt.astype(et)
2445: (12)                  edtdf = dtd.astype(et)
2446: (12)                  assert_equal(func(edf, edf.T), eddtf)
2447: (12)                  assert_equal(func(edf.T, edf), edtdf)
2448: (4) @pytest.mark.parametrize('func', (np.dot, np.matmul))
2449: (4) @pytest.mark.parametrize('dtype', 'ifdFD')
2450: (4) def test_no_dgemv(self, func, dtype):
2451: (8)     a = np.arange(8.0, dtype=dtype).reshape(2, 4)
2452: (8)     b = np.broadcast_to(1., (4, 1))
2453: (8)     ret1 = func(a, b)
2454: (8)     ret2 = func(a, b.copy())
2455: (8)     assert_equal(ret1, ret2)
2456: (8)     ret1 = func(b.T, a.T)
2457: (8)     ret2 = func(b.T.copy(), a.T)
2458: (8)     assert_equal(ret1, ret2)
2459: (8)     dt = np.dtype(dtype)
2460: (8)     a = np.zeros(8 * dt.itemsize // 2 + 1, dtype='int16')[1:].view(dtype)
2461: (8)     a = a.reshape(2, 4)
2462: (8)     b = a[0]
2463: (8)     assert_(a.__array_interface__['data'][0] % dt.itemsize != 0)
2464: (8)     ret1 = func(a, b)
2465: (8)     ret2 = func(a.copy(), b.copy())
2466: (8)     assert_equal(ret1, ret2)
2467: (8)     ret1 = func(b.T, a.T)
2468: (8)     ret2 = func(b.T.copy(), a.T.copy())
2469: (8)     assert_equal(ret1, ret2)
2470: (4) def test_dot(self):
2471: (8)     a = np.array([[1, 0], [0, 1]])
2472: (8)     b = np.array([[0, 1], [1, 0]])
2473: (8)     c = np.array([[9, 1], [1, -9]])
2474: (8)     assert_equal(np.dot(a, b), a.dot(b))
2475: (8)     assert_equal(np.dot(np.dot(a, b), c), a.dot(b).dot(c))
2476: (8)     c = np.zeros_like(a)
2477: (8)     a.dot(b, c)
2478: (8)     assert_equal(c, np.dot(a, b))
2479: (8)     c = np.zeros_like(a)
2480: (8)     a.dot(b=b, out=c)

```

```

2481: (8)             assert_equal(c, np.dot(a, b))
2482: (4)             def test_dot_type_mismatch(self):
2483: (8)                 c = 1.
2484: (8)                 A = np.array((1,1), dtype='i,i')
2485: (8)                 assert_raises(TypeError, np.dot, c, A)
2486: (8)                 assert_raises(TypeError, np.dot, A, c)
2487: (4)             def test_dot_out_mem_overlap(self):
2488: (8)                 np.random.seed(1)
2489: (8)                 dtypes = [np.dtype(code) for code in np.typecodes['All']]
2490: (18)                   if code not in 'USVM']
2491: (8)             for dtype in dtypes:
2492: (12)                 a = np.random.rand(3, 3).astype(dtype)
2493: (12)                 b = _aligned_zeros((3, 3), dtype=dtype)
2494: (12)                 b[...] = np.random.rand(3, 3)
2495: (12)                 y = np.dot(a, b)
2496: (12)                 x = np.dot(a, b, out=b)
2497: (12)                 assert_equal(x, y, err_msg=repr(dtype))
2498: (12)                 assert_raises(ValueError, np.dot, a, b, out=b[:2])
2499: (12)                 assert_raises(ValueError, np.dot, a, b, out=b.T)
2500: (4)             def test_dot_matmul_out(self):
2501: (8)                 class Sub(np.ndarray):
2502: (12)                     pass
2503: (8)                     a = np.ones((2, 2)).view(Sub)
2504: (8)                     b = np.ones((2, 2)).view(Sub)
2505: (8)                     out = np.ones((2, 2))
2506: (8)                     np.dot(a, b, out=out)
2507: (8)                     np.matmul(a, b, out=out)
2508: (4)             def test_dot_matmul_inner_array_casting_fails(self):
2509: (8)                 class A:
2510: (12)                     def __array__(self, *args, **kwargs):
2511: (16)                         raise NotImplementedError
2512: (8)                     assert_raises(NotImplementedError, np.dot, A(), A())
2513: (8)                     assert_raises(NotImplementedError, np.matmul, A(), A())
2514: (8)                     assert_raises(NotImplementedError, np.inner, A(), A())
2515: (4)             def test_matmul_out(self):
2516: (8)                 a = np.arange(18).reshape(2, 3, 3)
2517: (8)                 b = np.matmul(a, a)
2518: (8)                 c = np.matmul(a, a, out=a)
2519: (8)                 assert_(c is a)
2520: (8)                 assert_equal(c, b)
2521: (8)                 a = np.arange(18).reshape(2, 3, 3)
2522: (8)                 c = np.matmul(a, a, out=a[::-1, ...])
2523: (8)                 assert_(c.base is a.base)
2524: (8)                 assert_equal(c, b)
2525: (4)             def test_diagonal(self):
2526: (8)                 a = np.arange(12).reshape((3, 4))
2527: (8)                 assert_equal(a.diagonal(), [0, 5, 10])
2528: (8)                 assert_equal(a.diagonal(0), [0, 5, 10])
2529: (8)                 assert_equal(a.diagonal(1), [1, 6, 11])
2530: (8)                 assert_equal(a.diagonal(-1), [4, 9])
2531: (8)                 assert_raises(np.AxisError, a.diagonal, axis1=0, axis2=5)
2532: (8)                 assert_raises(np.AxisError, a.diagonal, axis1=5, axis2=0)
2533: (8)                 assert_raises(np.AxisError, a.diagonal, axis1=5, axis2=5)
2534: (8)                 assert_raises(ValueError, a.diagonal, axis1=1, axis2=1)
2535: (8)                 b = np.arange(8).reshape((2, 2, 2))
2536: (8)                 assert_equal(b.diagonal(), [[0, 6], [1, 7]])
2537: (8)                 assert_equal(b.diagonal(0), [[0, 6], [1, 7]])
2538: (8)                 assert_equal(b.diagonal(1), [[2], [3]])
2539: (8)                 assert_equal(b.diagonal(-1), [[4], [5]])
2540: (8)                 assert_raises(ValueError, b.diagonal, axis1=0, axis2=0)
2541: (8)                 assert_equal(b.diagonal(0, 1, 2), [[0, 3], [4, 7]])
2542: (8)                 assert_equal(b.diagonal(0, 0, 1), [[0, 6], [1, 7]])
2543: (8)                 assert_equal(b.diagonal(offset=1, axis1=0, axis2=2), [[1], [3]])
2544: (8)                 assert_equal(b.diagonal(0, 2, 1), [[0, 3], [4, 7]])
2545: (4)             def test_diagonal_view_notwriteable(self):
2546: (8)                 a = np.eye(3).diagonal()
2547: (8)                 assert_(not a.flags.writeable)
2548: (8)                 assert_(not a.flags.owndata)
2549: (8)                 a = np.diagonal(np.eye(3))

```

```

2550: (8)             assert_(not a.flags.writeable)
2551: (8)             assert_(not a.flags.owndata)
2552: (8)             a = np.diag(np.eye(3))
2553: (8)             assert_(not a.flags.writeable)
2554: (8)             assert_(not a.flags.owndata)
2555: (4)              def test_diagonal_memleak(self):
2556: (8)                  a = np.zeros((100, 100))
2557: (8)                  if HAS_REFCOUNT:
2558: (12)                      assert_(sys.getrefcount(a) < 50)
2559: (8)                  for i in range(100):
2560: (12)                      a.diagonal()
2561: (8)                  if HAS_REFCOUNT:
2562: (12)                      assert_(sys.getrefcount(a) < 50)
2563: (4)              def test_size_zero_memleak(self):
2564: (8)                  a = np.array([], dtype=np.float64)
2565: (8)                  x = np.array(2.0)
2566: (8)                  for _ in range(100):
2567: (12)                      np.dot(a, a, out=x)
2568: (8)                  if HAS_REFCOUNT:
2569: (12)                      assert_(sys.getrefcount(x) < 50)
2570: (4)              def test_trace(self):
2571: (8)                  a = np.arange(12).reshape((3, 4))
2572: (8)                  assert_equal(a.trace(), 15)
2573: (8)                  assert_equal(a.trace(0), 15)
2574: (8)                  assert_equal(a.trace(1), 18)
2575: (8)                  assert_equal(a.trace(-1), 13)
2576: (8)                  b = np.arange(8).reshape((2, 2))
2577: (8)                  assert_equal(b.trace(), [6, 8])
2578: (8)                  assert_equal(b.trace(0), [6, 8])
2579: (8)                  assert_equal(b.trace(1), [2, 3])
2580: (8)                  assert_equal(b.trace(-1), [4, 5])
2581: (8)                  assert_equal(b.trace(0, 0, 1), [6, 8])
2582: (8)                  assert_equal(b.trace(0, 0, 2), [5, 9])
2583: (8)                  assert_equal(b.trace(0, 1, 2), [3, 11])
2584: (8)                  assert_equal(b.trace(offset=1, axis1=0, axis2=2), [1, 3])
2585: (8)                  out = np.array(1)
2586: (8)                  ret = a.trace(out=out)
2587: (8)                  assert ret is out
2588: (4)              def test_trace_subclass(self):
2589: (8)                  class MyArray(np.ndarray):
2590: (12)                      pass
2591: (8)                  b = np.arange(8).reshape((2, 2)).view(MyArray)
2592: (8)                  t = b.trace()
2593: (8)                  assert_(isinstance(t, MyArray))
2594: (4)              def test_put(self):
2595: (8)                  icode = np.typecodes['AllInteger']
2596: (8)                  fcodes = np.typecodes['AllFloat']
2597: (8)                  for dt in icode + fcodes + '0':
2598: (12)                      tgt = np.array([0, 1, 0, 3, 0, 5], dtype=dt)
2599: (12)                      a = np.zeros(6, dtype=dt)
2600: (12)                      a.put([1, 3, 5], [1, 3, 5])
2601: (12)                      assert_equal(a, tgt)
2602: (12)                      a = np.zeros((2, 3), dtype=dt)
2603: (12)                      a.put([1, 3, 5], [1, 3, 5])
2604: (12)                      assert_equal(a, tgt.reshape(2, 3))
2605: (8)                      for dt in '?':
2606: (12)                          tgt = np.array([False, True, False, True, False, True], dtype=dt)
2607: (12)                          a = np.zeros(6, dtype=dt)
2608: (12)                          a.put([1, 3, 5], [True]*3)
2609: (12)                          assert_equal(a, tgt)
2610: (12)                          a = np.zeros((2, 3), dtype=dt)
2611: (12)                          a.put([1, 3, 5], [True]*3)
2612: (12)                          assert_equal(a, tgt.reshape(2, 3))
2613: (8)                      a = np.zeros(6)
2614: (8)                      a.flags.writeable = False
2615: (8)                      assert_raises(ValueError, a.put, [1, 3, 5], [1, 3, 5])
2616: (8)                      bad_array = [1, 2, 3]
2617: (8)                      assert_raises(TypeError, np.put, bad_array, [0, 2], 5)
2618: (4)              def test_ravel(self):

```

```

2619: (8)             a = np.array([[0, 1], [2, 3]])
2620: (8)             assert_equal(a.ravel(), [0, 1, 2, 3])
2621: (8)             assert_(not a.ravel().flags.owndata)
2622: (8)             assert_equal(a.ravel('F'), [0, 2, 1, 3])
2623: (8)             assert_equal(a.ravel(order='C'), [0, 1, 2, 3])
2624: (8)             assert_equal(a.ravel(order='F'), [0, 2, 1, 3])
2625: (8)             assert_equal(a.ravel(order='A'), [0, 1, 2, 3])
2626: (8)             assert_(not a.ravel(order='A').flags.owndata)
2627: (8)             assert_equal(a.ravel(order='K'), [0, 1, 2, 3])
2628: (8)             assert_(not a.ravel(order='K').flags.owndata)
2629: (8)             assert_equal(a.ravel(), a.reshape(-1))
2630: (8)             a = np.array([[0, 1], [2, 3]], order='F')
2631: (8)             assert_equal(a.ravel(), [0, 1, 2, 3])
2632: (8)             assert_equal(a.ravel(order='A'), [0, 2, 1, 3])
2633: (8)             assert_equal(a.ravel(order='K'), [0, 2, 1, 3])
2634: (8)             assert_(not a.ravel(order='A').flags.owndata)
2635: (8)             assert_(not a.ravel(order='K').flags.owndata)
2636: (8)             assert_equal(a.ravel(), a.reshape(-1))
2637: (8)             assert_equal(a.ravel(order='A'), a.reshape(-1, order='A'))
2638: (8)             a = np.array([[0, 1], [2, 3]])[::-1, :]
2639: (8)             assert_equal(a.ravel(), [2, 3, 0, 1])
2640: (8)             assert_equal(a.ravel(order='C'), [2, 3, 0, 1])
2641: (8)             assert_equal(a.ravel(order='F'), [2, 0, 3, 1])
2642: (8)             assert_equal(a.ravel(order='A'), [2, 3, 0, 1])
2643: (8)             assert_equal(a.ravel(order='K'), [2, 3, 0, 1])
2644: (8)             assert_(a.ravel(order='K').flags.owndata)
2645: (8)             a = np.arange(10)[::2]
2646: (8)             assert_(a.ravel('K').flags.owndata)
2647: (8)             assert_(a.ravel('C').flags.owndata)
2648: (8)             assert_(a.ravel('F').flags.owndata)
2649: (8)             a = np.arange(2**3 * 2)[::2]
2650: (8)             a = a.reshape(2, 1, 2, 2).swapaxes(-1, -2)
2651: (8)             strides = list(a.strides)
2652: (8)             strides[1] = 123
2653: (8)             a.strides = strides
2654: (8)             assert_(a.ravel(order='K').flags.owndata)
2655: (8)             assert_equal(a.ravel('K'), np.arange(0, 15, 2))
2656: (8)             a = np.arange(2**3)
2657: (8)             a = a.reshape(2, 1, 2, 2).swapaxes(-1, -2)
2658: (8)             strides = list(a.strides)
2659: (8)             strides[1] = 123
2660: (8)             a.strides = strides
2661: (8)             assert_(np.may_share_memory(a.ravel(order='K'), a))
2662: (8)             assert_equal(a.ravel(order='K'), np.arange(2**3))
2663: (8)             a = np.arange(4)[::-1].reshape(2, 2)
2664: (8)             assert_(a.ravel(order='C').flags.owndata)
2665: (8)             assert_(a.ravel(order='K').flags.owndata)
2666: (8)             assert_equal(a.ravel('C'), [3, 2, 1, 0])
2667: (8)             assert_equal(a.ravel('K'), [3, 2, 1, 0])
2668: (8)             a = np.array([[1]])
2669: (8)             a.strides = (123, 432)
2670: (8)             if np.ones(1).strides == (8,):
2671: (12)                 assert_(np.may_share_memory(a.ravel('K'), a))
2672: (12)                 assert_equal(a.ravel('K').strides, (a.dtype.itemsize,))
2673: (8)             for order in ('C', 'F', 'A', 'K'):
2674: (12)                 a = np.array(0)
2675: (12)                 assert_equal(a.ravel(order), [0])
2676: (12)                 assert_(np.may_share_memory(a.ravel(order), a))
2677: (8)             b = np.arange(2**4 * 2)[::2].reshape(2, 2, 2, 2)
2678: (8)             a = b[..., ::2]
2679: (8)             assert_equal(a.ravel('K'), [0, 4, 8, 12, 16, 20, 24, 28])
2680: (8)             assert_equal(a.ravel('C'), [0, 4, 8, 12, 16, 20, 24, 28])
2681: (8)             assert_equal(a.ravel('A'), [0, 4, 8, 12, 16, 20, 24, 28])
2682: (8)             assert_equal(a.ravel('F'), [0, 16, 8, 24, 4, 20, 12, 28])
2683: (8)             a = b[::-2, ...]
2684: (8)             assert_equal(a.ravel('K'), [0, 2, 4, 6, 8, 10, 12, 14])
2685: (8)             assert_equal(a.ravel('C'), [0, 2, 4, 6, 8, 10, 12, 14])
2686: (8)             assert_equal(a.ravel('A'), [0, 2, 4, 6, 8, 10, 12, 14])
2687: (8)             assert_equal(a.ravel('F'), [0, 8, 4, 12, 2, 10, 6, 14])

```

```

2688: (4)
2689: (8)
2690: (12)
2691: (8)
2692: (8)
2693: (8)
2694: (8)
2695: (8)
2696: (8)
2697: (8)
2698: (8)
2699: (8)
2700: (8)
2701: (4)
2702: (8)
2703: (8)
2704: (8)
2705: (8)
2706: (8)
2707: (8)
2708: (8)
2709: (8)
2710: (8)
2711: (12)
2712: (16)
2713: (20)
2714: (20)
2715: (20)
2716: (20)
2717: (20)
2718: (20)
2719: (20)
2720: (20)
2721: (33)
2722: (33)
2723: (20)
2724: (20)
2725: (24)
2726: (4)
def test_ravel_subclass(self):
    class ArraySubclass(np.ndarray):
        pass
    a = np.arange(10).view(ArraySubclass)
    assert_(isinstance(a.ravel('C'), ArraySubclass))
    assert_(isinstance(a.ravel('F'), ArraySubclass))
    assert_(isinstance(a.ravel('A'), ArraySubclass))
    assert_(isinstance(a.ravel('K'), ArraySubclass))
    a = np.arange(10)[::2].view(ArraySubclass)
    assert_(isinstance(a.ravel('C'), ArraySubclass))
    assert_(isinstance(a.ravel('F'), ArraySubclass))
    assert_(isinstance(a.ravel('A'), ArraySubclass))
    assert_(isinstance(a.ravel('K'), ArraySubclass))
def test_swapaxes(self):
    a = np.arange(1*2*3*4).reshape(1, 2, 3, 4).copy()
    idx = np.indices(a.shape)
    assert_(a.flags['OWNDATA'])
    b = a.copy()
    assert_raises(np.AxisError, a.swapaxes, -5, 0)
    assert_raises(np.AxisError, a.swapaxes, 4, 0)
    assert_raises(np.AxisError, a.swapaxes, 0, -5)
    assert_raises(np.AxisError, a.swapaxes, 0, 4)
    for i in range(-4, 4):
        for j in range(-4, 4):
            for k, src in enumerate((a, b)):
                c = src.swapaxes(i, j)
                shape = list(src.shape)
                shape[i] = src.shape[j]
                shape[j] = src.shape[i]
                assert_equal(c.shape, shape, str((i, j, k)))
                i0, i1, i2, i3 = [dim-1 for dim in c.shape]
                j0, j1, j2, j3 = [dim-1 for dim in src.shape]
                assert_equal(src[idx[j0], idx[j1], idx[j2], idx[j3]],
                            c[idx[i0], idx[i1], idx[i2], idx[i3]],
                            str((i, j, k)))
                assert_(not c.flags['OWNDATA'], str((i, j, k)))
                if k == 1:
                    b = c
def test_conjugate(self):
    a = np.array([1-1j, 1+1j, 23+23.0j])
    ac = a.conj()
    assert_equal(a.real, ac.real)
    assert_equal(a.imag, -ac.imag)
    assert_equal(ac, a.conjugate())
    assert_equal(ac, np.conjugate(a))
    a = np.array([1-1j, 1+1j, 23+23.0j], 'F')
    ac = a.conj()
    assert_equal(a.real, ac.real)
    assert_equal(a.imag, -ac.imag)
    assert_equal(ac, a.conjugate())
    assert_equal(ac, np.conjugate(a))
    a = np.array([1, 2, 3])
    ac = a.conj()
    assert_equal(a, ac)
    assert_equal(ac, a.conjugate())
    assert_equal(ac, np.conjugate(a))
    a = np.array([1.0, 2.0, 3.0])
    ac = a.conj()
    assert_equal(a, ac)
    assert_equal(ac, a.conjugate())
    assert_equal(ac, np.conjugate(a))
    a = np.array([1-1j, 1+1j, 1, 2.0], object)
    ac = a.conj()
    assert_equal(ac, [k.conjugate() for k in a])
    assert_equal(ac, a.conjugate())
    assert_equal(ac, np.conjugate(a))
    a = np.array([1-1j, 1, 2.0, 'f'], object)
    assert_raises(TypeError, lambda: a.conj())
    assert_raises(TypeError, lambda: a.conjugate())

```

```

2757: (4) def test_conjugate_out(self):
2758: (8)     a = np.array([1-1j, 1+1j, 23+23.0j])
2759: (8)     out = np.empty_like(a)
2760: (8)     res = a.conjugate()
2761: (8)     assert res is out
2762: (8)     assert_array_equal(out, a.conjugate())
2763: (4) def test_complex_(self):
2764: (8)     dtypes = ['i1', 'i2', 'i4', 'i8',
2765: (18)         'u1', 'u2', 'u4', 'u8',
2766: (18)         'f', 'd', 'g', 'F', 'D', 'G',
2767: (18)         '?', 'O']
2768: (8)     for dt in dtypes:
2769: (12)         a = np.array(7, dtype=dt)
2770: (12)         b = np.array([7], dtype=dt)
2771: (12)         c = np.array([[[[7]]]], dtype=dt)
2772: (12)         msg = 'dtype: {}'.format(dt)
2773: (12)         ap = complex(a)
2774: (12)         assert_equal(ap, a, msg)
2775: (12)         with assert_warns(DeprecationWarning):
2776: (16)             bp = complex(b)
2777: (12)             assert_equal(bp, b, msg)
2778: (12)             with assert_warns(DeprecationWarning):
2779: (16)                 cp = complex(c)
2780: (12)                 assert_equal(cp, c, msg)
2781: (4) def test_complex_should_not_work(self):
2782: (8)     dtypes = ['i1', 'i2', 'i4', 'i8',
2783: (18)         'u1', 'u2', 'u4', 'u8',
2784: (18)         'f', 'd', 'g', 'F', 'D', 'G',
2785: (18)         '?', 'O']
2786: (8)     for dt in dtypes:
2787: (12)         a = np.array([1, 2, 3], dtype=dt)
2788: (12)         assert_raises(TypeError, complex, a)
2789: (8)         dt = np.dtype([('a', 'f8'), ('b', 'i1')])
2790: (8)         b = np.array((1.0, 3), dtype=dt)
2791: (8)         assert_raises(TypeError, complex, b)
2792: (8)         c = np.array([(1.0, 3), (2e-3, 7)], dtype=dt)
2793: (8)         assert_raises(TypeError, complex, c)
2794: (8)         d = np.array('1+1j')
2795: (8)         assert_raises(TypeError, complex, d)
2796: (8)         e = np.array(['1+1j'], 'U')
2797: (8)         with assert_warns(DeprecationWarning):
2798: (12)             assert_raises(TypeError, complex, e)
2799: (0) class TestCequenceMethods:
2800: (4)     def test_array_contains(self):
2801: (8)         assert_(4.0 in np.arange(16.).reshape(4,4))
2802: (8)         assert_(20.0 not in np.arange(16.).reshape(4,4))
2803: (0) class TestBinop:
2804: (4)     def test_inplace(self):
2805: (8)         assert_array_almost_equal(np.array([0.5]) * np.array([1.0, 2.0]),
2806: (34)                         [0.5, 1.0])
2807: (8)         d = np.array([0.5, 0.5])[:,2:]
2808: (8)         assert_array_almost_equal(d * (d * np.array([1.0, 2.0])), [0.25, 0.5])
2809: (34)         a = np.array([0.5])
2810: (8)         b = np.array([0.5])
2811: (8)         c = a + b
2812: (8)         c = a - b
2813: (8)         c = a * b
2814: (8)         c = a / b
2815: (8)         assert_equal(a, b)
2816: (8)         assert_almost_equal(c, 1.)
2817: (8)         c = a + b * 2. / b * a - a / b
2818: (8)         assert_equal(a, b)
2819: (8)         assert_equal(c, 0.5)
2820: (8)         a = np.array([5])
2821: (8)         b = np.array([3])
2822: (8)         c = (a * a) / b
2823: (8)         assert_almost_equal(c, 25 / 3)
2824: (8)         assert_equal(a, 5)
2825: (8)

```

```

2826: (8)             assert_equal(b, 3)
2827: (4)             @pytest.mark.xfail(IS_PYPY, reason="Bug in pypy3.{9, 10}-v7.3.13, #24862")
2828: (4)             def test_ufunc_binop_interaction(self):
2829: (8)                 ops = {
2830: (12)                     'add':      (np.add, True, float),
2831: (12)                     'sub':      (np.subtract, True, float),
2832: (12)                     'mul':      (np.multiply, True, float),
2833: (12)                     'truediv':  (np.true_divide, True, float),
2834: (12)                     'floordiv': (np.floor_divide, True, float),
2835: (12)                     'mod':      (np.remainder, True, float),
2836: (12)                     'divmod':   (np.divmod, False, float),
2837: (12)                     'pow':      (np.power, True, int),
2838: (12)                     'lshift':   (np.left_shift, True, int),
2839: (12)                     'rshift':   (np.right_shift, True, int),
2840: (12)                     'and':      (np.bitwise_and, True, int),
2841: (12)                     'xor':      (np.bitwise_xor, True, int),
2842: (12)                     'or':       (np.bitwise_or, True, int),
2843: (12)                     'matmul':   (np.matmul, True, float),
2844: (8)             }
2845: (8)             class Coerced(Exception):
2846: (12)                 pass
2847: (8)                 def array_impl(self):
2848: (12)                     raise Coerced
2849: (8)                 def op_impl(self, other):
2850: (12)                     return "forward"
2851: (8)                 def rop_impl(self, other):
2852: (12)                     return "reverse"
2853: (8)                 def iop_impl(self, other):
2854: (12)                     return "in-place"
2855: (8)                 def array_ufunc_impl(self, ufunc, method, *args, **kwargs):
2856: (12)                     return ("__array_ufunc__", ufunc, method, args, kwargs)
2857: (8)                 def make_obj(base, array_priority=False, array_ufunc=False,
2858: (21)                         alleged_module="__main__"):
2859: (12)                     class_namespace = {"__array__": array_impl}
2860: (12)                     if array_priority is not False:
2861: (16)                         class_namespace["__array_priority__"] = array_priority
2862: (12)                     for op in ops:
2863: (16)                         class_namespace["__{0}__".format(op)] = op_impl
2864: (16)                         class_namespace["__r{0}__".format(op)] = rop_impl
2865: (16)                         class_namespace["__i{0}__".format(op)] = iop_impl
2866: (12)                     if array_ufunc is not False:
2867: (16)                         class_namespace["__array_ufunc__"] = array_ufunc
2868: (12)                     eval_namespace = {"base": base,
2869: (30)                         "class_namespace": class_namespace,
2870: (30)                         "__name__": alleged_module,
2871: (30)                         }
2872: (12)                     MyType = eval("type('MyType', (base,), class_namespace)",
2873: (26)                         eval_namespace)
2874: (12)                     if issubclass(MyType, np.ndarray):
2875: (16)                         return np.arange(3, 7).reshape(2, 2).view(MyType)
2876: (12)                     else:
2877: (16)                         return MyType()
2878: (8)                     def check(obj, binop_override_expected, ufunc_override_expected,
2879: (18)                         inplace_override_expected, check_scalar=True):
2880: (12)                         for op, (ufunc, has_inplace, dtype) in ops.items():
2881: (16)                             err_msg = ('op: %s, ufunc: %s, has_inplace: %s, dtype: %s'
2882: (27)                                 % (op, ufunc, has_inplace, dtype))
2883: (16)                             check_objs = [np.arange(3, 7, dtype=dtype).reshape(2, 2)]
2884: (16)                             if check_scalar:
2885: (20)                                 check_objs.append(check_objs[0][0])
2886: (16)                             for arr in check_objs:
2887: (20)                                 arr_method = getattr(arr, "__{0}__".format(op))
2888: (20)                                 def first_out_arg(result):
2889: (24)                                     if op == "divmod":
2890: (28)   assert_(isinstance(result, tuple))
2891: (28)   return result[0]
2892: (24)                                     else:
2893: (28)   return result
2894: (20)                                     if binop_override_expected:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2895: (24) assert_equal(arr_method(obj), NotImplemented, err_msg)
2896: (20) elif ufunc_override_expected:
2897: (24)     assert_equal(arr_method(obj)[0], "__array_ufunc__",
2898: (37)             err_msg)
2899: (20) else:
2900: (24)     if (isinstance(obj, np.ndarray) and
2901: (28)         (type(obj).__array_ufunc__ is
2902: (29)             np.ndarray.__array_ufunc__)):
2903: (28)         res = first_out_arg(arr_method(obj))
2904: (28)         assert_(res.__class__ is obj.__class__, err_msg)
2905: (24)     else:
2906: (28)         assert_raises((TypeError, Coerced),
2907: (42)             arr_method, obj, err_msg=err_msg)
2908: (20) arr_rmethod = getattr(arr, "__r{0}__".format(op))
2909: (20) if ufunc_override_expected:
2910: (24)     res = arr_rmethod(obj)
2911: (24)     assert_equal(res[0], "__array_ufunc__",
2912: (37)             err_msg=err_msg)
2913: (24)     assert_equal(res[1], ufunc, err_msg=err_msg)
2914: (20) else:
2915: (24)     if (isinstance(obj, np.ndarray) and
2916: (32)         (type(obj).__array_ufunc__ is
2917: (33)             np.ndarray.__array_ufunc__)):
2918: (28)         res = first_out_arg(arr_rmethod(obj))
2919: (28)         assert_(res.__class__ is obj.__class__, err_msg)
2920: (24)     else:
2921: (28)         assert_raises((TypeError, Coerced),
2922: (42)             arr_rmethod, obj, err_msg=err_msg)
2923: (20) if has_inplace and isinstance(arr, np.ndarray):
2924: (24)     arr_imethod = getattr(arr, "__i{0}__".format(op))
2925: (24)     if inplace_override_expected:
2926: (28)         assert_equal(arr_method(obj), NotImplemented,
2927: (41)             err_msg=err_msg)
2928: (24)     elif ufunc_override_expected:
2929: (28)         res = arr_imethod(obj)
2930: (28)         assert_equal(res[0], "__array_ufunc__", err_msg)
2931: (28)         assert_equal(res[1], ufunc, err_msg)
2932: (28)         assert_(type(res[-1]["out"]) is tuple, err_msg)
2933: (28)         assert_(res[-1]["out"][0] is arr, err_msg)
2934: (24)     else:
2935: (28)         if (isinstance(obj, np.ndarray) and
2936: (36)             (type(obj).__array_ufunc__ is
2937: (36)                 np.ndarray.__array_ufunc__)):
2938: (32)             assert_(arr_imethod(obj) is arr, err_msg)
2939: (28)         else:
2940: (32)             assert_raises((TypeError, Coerced),
2941: (46)                 arr_imethod, obj,
2942: (46)                 err_msg=err_msg)
2943: (20) op_fn = getattr(operator, op, None)
2944: (20) if op_fn is None:
2945: (24)     op_fn = getattr(operator, op + "_", None)
2946: (20) if op_fn is None:
2947: (24)     op_fn = getattr(builtins, op)
2948: (20) assert_equal(op_fn(obj, arr), "forward", err_msg)
2949: (20) if not isinstance(obj, np.ndarray):
2950: (24)     if binop_override_expected:
2951: (28)         assert_equal(op_fn(arr, obj), "reverse", err_msg)
2952: (24)     elif ufunc_override_expected:
2953: (28)         assert_equal(op_fn(arr, obj)[0],
2954: (41)             err_msg)
2955: (20)     if ufunc_override_expected:
2956: (24)         assert_equal(ufunc(obj, arr)[0], "__array_ufunc__",
2957: (37)             err_msg)
2958: (8) check(make_obj(object), False, False, False)
2959: (8) check(make_obj(object, array_priority=-2**30), False, False, False)
2960: (8) check(make_obj(object, array_priority=1), True, False, True)
2961: (8) check(make_obj(np.ndarray, array_priority=1), False, False, False,
2962: (14)             check_scalar=False)

```

```

2963: (8)
2964: (23)
2965: (8)
2966: (23)
2967: (8)
2968: (8)
2969: (14)
2970: (4)
2971: (4)
2972: (8)
2973: (12)
2974: (12)
2975: (16)
2976: (20)
2977: (16)
2978: (12)
2979: (16)
2980: (8)
2981: (12)
2982: (8)
2983: (8)
2984: (8)
2985: (4)
2986: (8)
2987: (12)
2988: (16)
2989: (8)
2990: (8)
2991: (8)
2992: (8)
2993: (8)
2994: (8)
2995: (8)
2996: (8)
2997: (8)
2998: (4)
2999: (8)
3000: (12)
3001: (16)
3002: (20)
3003: (24)
3004: (16)
3005: (20)
3006: (24)
3007: (8)
3008: (8)
3009: (8)
3010: (8)
3011: (8)
3012: (8)
3013: (8)
3014: (8)
3015: (8)
3016: (8)
3017: (8)
3018: (8)
3019: (8)
3020: (8)
3021: (8)
3022: (8)
3023: (8)
3024: (8)
3025: (12)
3026: (8)
3027: (8)
3028: (8)
3029: (8)
3030: (8)
3031: (8)

    check(make_obj(object, array_priority=1,
                    array_ufunc=array_ufunc_impl), False, True, False)
    check(make_obj(np.ndarray, array_priority=1,
                    array_ufunc=array_ufunc_impl), False, True, False)
    check(make_obj(object, array_ufunc=None), True, False, False)
    check(make_obj(np.ndarray, array_ufunc=None), True, False, False,
          check_scalar=False)
@pytest.mark.parametrize("priority", [None, "runtime error"])
def test_ufunc_binop_bad_array_priority(self, priority):
    class BadPriority:
        @property
        def __array_priority__(self):
            if priority == "runtime error":
                raise RuntimeError("RuntimeError in __array_priority__!")
            return priority
        def __radd__(self, other):
            return "result"
    class LowPriority(np.ndarray):
        __array_priority__ = -1000
    res = np.arange(3).view(LowPriority) + BadPriority()
    assert res.shape == (3,)
    assert res[0] == 'result'

def test_ufunc_override_normalize_signature(self):
    class SomeClass:
        def __array_ufunc__(self, ufunc, method, *inputs, **kw):
            return kw
    a = SomeClass()
    kw = np.add(a, [1])
    assert_('sig' not in kw and 'signature' not in kw)
    kw = np.add(a, [1], sig='ii->i')
    assert_('sig' not in kw and 'signature' in kw)
    assert_equal(kw['signature'], 'ii->i')
    kw = np.add(a, [1], signature='ii->i')
    assert_('sig' not in kw and 'signature' in kw)
    assert_equal(kw['signature'], 'ii->i')

def test_array_ufunc_index(self):
    class CheckIndex:
        def __array_ufunc__(self, ufunc, method, *inputs, **kw):
            for i, a in enumerate(inputs):
                if a is self:
                    return i
            for j, a in enumerate(kw['out']):
                if a is self:
                    return (j,)

    a = CheckIndex()
    dummy = np.arange(2.)
    assert_equal(np.sin(a), 0)
    assert_equal(np.sin(dummy, a), (0,))
    assert_equal(np.sin(dummy, out=a), (0,))
    assert_equal(np.sin(dummy, out=(a,)), (0,))
    assert_equal(np.sin(a, a), 0)
    assert_equal(np.sin(a, out=a), 0)
    assert_equal(np.sin(a, out=(a,)), 0)
    assert_equal(np.modf(dummy, a), (0,))
    assert_equal(np.modf(dummy, None, a), (1,))
    assert_equal(np.modf(dummy, dummy, a), (1,))
    assert_equal(np.modf(dummy, out=(a, None)), (0,))
    assert_equal(np.modf(dummy, out=(a, dummy)), (0,))
    assert_equal(np.modf(dummy, out=(None, a)), (1,))
    assert_equal(np.modf(dummy, out=(dummy, a)), (1,))
    assert_equal(np.modf(a, out=(dummy, a)), 0)
    with assert_raises(TypeError):
        np.modf(dummy, out=a)
    assert_raises(ValueError, np.modf, dummy, out=(a,))
    assert_equal(np.add(a, dummy), 0)
    assert_equal(np.add(dummy, a), 1)
    assert_equal(np.add(dummy, dummy, a), (0,))
    assert_equal(np.add(dummy, a, a), 1)
    assert_equal(np.add(dummy, dummy, out=a), (0,))


```

```

3032: (8)             assert_equal(np.add(dummy, dummy, out=(a,)), (0,))
3033: (8)             assert_equal(np.add(a, dummy, out=a), 0)
3034: (4)             def test_out_override(self):
3035: (8)                 class OutClass(np.ndarray):
3036: (12)                   def __array_ufunc__(self, ufunc, method, *inputs, **kw):
3037: (16)                       if 'out' in kw:
3038: (20)                           tmp_kw = kw.copy()
3039: (20)                           tmp_kw.pop('out')
3040: (20)                           func = getattr(ufunc, method)
3041: (20)                           kw['out'][0][...] = func(*inputs, **tmp_kw)
3042: (8)                           A = np.array([0]).view(OutClass)
3043: (8)                           B = np.array([5])
3044: (8)                           C = np.array([6])
3045: (8)                           np.multiply(C, B, A)
3046: (8)                           assert_equal(A[0], 30)
3047: (8)                           assert_(isinstance(A, OutClass))
3048: (8)                           A[0] = 0
3049: (8)                           np.multiply(C, B, out=A)
3050: (8)                           assert_equal(A[0], 30)
3051: (8)                           assert_(isinstance(A, OutClass))
3052: (4)             def test_pow_override_with_errors(self):
3053: (8)                 class PowerOnly(np.ndarray):
3054: (12)                   def __array_ufunc__(self, ufunc, method, *inputs, **kw):
3055: (16)                       if ufunc is not np.power:
3056: (20)                           raise NotImplementedError
3057: (16)                           return "POWER!"
3058: (8)                           a = np.array(5., dtype=np.float64).view(PowerOnly)
3059: (8)                           assert_equal(a ** 2.5, "POWER!")
3060: (8)                           with assert_raises(NotImplementedError):
3061: (12)                               a ** 0.5
3062: (8)                           with assert_raises(NotImplementedError):
3063: (12)                               a ** 0
3064: (8)                           with assert_raises(NotImplementedError):
3065: (12)                               a ** 1
3066: (8)                           with assert_raises(NotImplementedError):
3067: (12)                               a ** -1
3068: (8)                           with assert_raises(NotImplementedError):
3069: (12)                               a ** 2
3070: (4)             def test_pow_array_object_dtype(self):
3071: (8)                 class SomeClass:
3072: (12)                     def __init__(self, num=None):
3073: (16)                         self.num = num
3074: (12)                     def __mul__(self, other):
3075: (16)                         raise AssertionError('__mul__ should not be called')
3076: (12)                     def __div__(self, other):
3077: (16)                         raise AssertionError('__div__ should not be called')
3078: (12)                     def __pow__(self, exp):
3079: (16)                         return SomeClass(num=self.num ** exp)
3080: (12)                     def __eq__(self, other):
3081: (16)                         if isinstance(other, SomeClass):
3082: (20)                             return self.num == other.num
3083: (12)                         __rpow__ = __pow__
3084: (8)             def pow_for(exp, arr):
3085: (12)                 return np.array([x ** exp for x in arr])
3086: (8)                 obj_arr = np.array([SomeClass(1), SomeClass(2), SomeClass(3)])
3087: (8)                 assert_equal(obj_arr ** 0.5, pow_for(0.5, obj_arr))
3088: (8)                 assert_equal(obj_arr ** 0, pow_for(0, obj_arr))
3089: (8)                 assert_equal(obj_arr ** 1, pow_for(1, obj_arr))
3090: (8)                 assert_equal(obj_arr ** -1, pow_for(-1, obj_arr))
3091: (8)                 assert_equal(obj_arr ** 2, pow_for(2, obj_arr))
3092: (4)             def test_pos_array_ufunc_override(self):
3093: (8)                 class A(np.ndarray):
3094: (12)                     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
3095: (16)                         return getattr(ufunc, method)(*[i.view(np.ndarray) for
3096: (48)                             i in inputs], **kwargs)
3097: (8)                         tst = np.array('foo').view(A)
3098: (8)                         with assert_raises(TypeError):
3099: (12)                             +tst
3100: (0)             class TestTemporaryElide:

```

```

3101: (4) def test_extension_incref_elide(self):
3102: (8)     from numpy.core._multiarray_tests import incref_elide
3103: (8)     d = np.ones(100000)
3104: (8)     orig, res = incref_elide(d)
3105: (8)     d + d
3106: (8)     assert_array_equal(orig, d)
3107: (8)     assert_array_equal(res, d + d)
3108: (4) def test_extension_incref_elide_stack(self):
3109: (8)     from numpy.core._multiarray_tests import incref_elide_l
3110: (8)     l = [1, 1, 1, 1, np.ones(100000)]
3111: (8)     res = incref_elide_l(l)
3112: (8)     assert_array_equal(l[4], np.ones(100000))
3113: (8)     assert_array_equal(res, l[4] + l[4])
3114: (4) def test_temporary_with_cast(self):
3115: (8)     d = np.ones(200000, dtype=np.int64)
3116: (8)     assert_equal(((d + d) + 2**222).dtype, np.dtype('O'))
3117: (8)     r = ((d + d) / 2)
3118: (8)     assert_equal(r.dtype, np.dtype('f8'))
3119: (8)     r = np.true_divide((d + d), 2)
3120: (8)     assert_equal(r.dtype, np.dtype('f8'))
3121: (8)     r = ((d + d) / 2.)
3122: (8)     assert_equal(r.dtype, np.dtype('f8'))
3123: (8)     r = ((d + d) // 2)
3124: (8)     assert_equal(r.dtype, np.dtype(np.int64))
3125: (8)     f = np.ones(100000, dtype=np.float32)
3126: (8)     assert_equal(((f + f) + f.astype(np.float64)).dtype, np.dtype('f8'))
3127: (8)     d = f.astype(np.float64)
3128: (8)     assert_equal(((f + f) + d).dtype, d.dtype)
3129: (8)     l = np.ones(100000, dtype=np.longdouble)
3130: (8)     assert_equal(((d + d) + 1).dtype, l.dtype)
3131: (8)     for dt in (np.complex64, np.complex128, np.clongdouble):
3132: (12)         c = np.ones(100000, dtype=dt)
3133: (12)         r = abs(c * 2.0)
3134: (12)         assert_equal(r.dtype, np.dtype('f%d' % (c.itemsize // 2)))
3135: (4) def test_elide_broadcast(self):
3136: (8)     d = np.ones((2000, 1), dtype=int)
3137: (8)     b = np.ones((2000), dtype=bool)
3138: (8)     r = (1 - d) + b
3139: (8)     assert_equal(r, 1)
3140: (8)     assert_equal(r.shape, (2000, 2000))
3141: (4) def test_elide_scalar(self):
3142: (8)     a = np.bool_()
3143: (8)     assert_(type(~(a & a)) is np.bool_)
3144: (4) def test_elide_scalar_READONLY(self):
3145: (8)     a = np.empty(100000, dtype=np.float64)
3146: (8)     a.imag ** 2
3147: (4) def test_elide_READONLY(self):
3148: (8)     r = np.asarray(np.broadcast_to(np.zeros(1), 100000).flat) * 0.0
3149: (8)     assert_equal(r, 0)
3150: (4) def test_elide_updateifcopy(self):
3151: (8)     a = np.ones(2**20)[::2]
3152: (8)     b = a.flat.__array__() + 1
3153: (8)     del b
3154: (8)     assert_equal(a, 1)
3155: (0) class TestCAPI:
3156: (4)     def test_IsPythonScalar(self):
3157: (8)         from numpy.core._multiarray_tests import IsPythonScalar
3158: (8)         assert_(IsPythonScalar(b'foobar'))
3159: (8)         assert_(IsPythonScalar(1))
3160: (8)         assert_(IsPythonScalar(2**80))
3161: (8)         assert_(IsPythonScalar(2.))
3162: (8)         assert_(IsPythonScalar("a"))
3163: (4)         @pytest.mark.parametrize("converter",
3164: (13)             [_multiarray_tests.run_scalar_intp_converter,
3165: (14)             _multiarray_tests.run_scalar_intp_from_sequence])
3166: (4)     def test_intp_sequence_converters(self, converter):
3167: (8)         assert converter(10) == (10,)
3168: (8)         assert converter(-1) == (-1,)
3169: (8)         assert converter(np.array(123)) == (123,) 
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

3170: (8)
3171: (8)
3172: (4)
3173: (13)
3174: (14)
3175: (4)
3176: (12)
3177: (4)
3178: (8)
3179: (16)
"):
3180: (12)
3181: (8)
3182: (16)
3183: (22)
3184: (12)
3185: (8)
3186: (16)
3187: (12)
3188: (8)
3189: (16)
3190: (12)
3191: (0)
3192: (4)
3193: (8)
3194: (8)
3195: (8)
3196: (0)
3197: (4)
3198: (24)
3199: (32)
3200: (4)
3201: (8)
3202: (4)
3203: (8)
3204: (8)
3205: (16)
3206: (16)
3207: (8)
3208: (16)
3209: (16)
3210: (8)
3211: (12)
3212: (20)
3213: (12)
3214: (20)
3215: (12)
3216: (25)
3217: (12)
3218: (25)
3219: (4)
3220: (24)
3221: (4)
3222: (8)
3223: (8)
3224: (8)
3225: (36)
3226: (8)
3227: (8)
3228: (52)
3229: (8)
3230: (4)
3231: (8)
3232: (8)
3233: (8)
3234: (8)
3235: (12)
3236: (20)
3237: (12)

    assert converter((10,)) == (10,)
    assert converter(np.array([11])) == (11,)
    @pytest.mark.parametrize("converter",
        [_multiarray_tests.run_scalar_intp_converter,
         _multiarray_tests.run_scalar_intp_from_sequence])
    @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
                       reason="PyPy bug in error formatting")
    def test_intp_sequence_converters_errors(self, converter):
        with pytest.raises(TypeError,
                           match="expected a sequence of integers or a single integer",
                           ):
            converter(object())
        with pytest.raises(TypeError,
                           match="expected a sequence of integers or a single integer, "
                                 "got '32.0')":
            converter(32.)
        with pytest.raises(TypeError,
                           match="'float' object cannot be interpreted as an integer"):
            converter([32.])
        with pytest.raises(ValueError,
                           match="Maximum allowed dimension"):
            converter(2**64)

class TestSubscripting:
    def test_test_zero_rank(self):
        x = np.array([1, 2, 3])
        assert_(isinstance(x[0], np.int_))
        assert_(type(x[0, ...]) is np.ndarray)

class TestPickling:
    @pytest.mark.skipif(pickle.HIGHEST_PROTOCOL >= 5,
                       reason=('this tests the error messages when trying to'
                               'protocol 5 although it is not available'))
    def test_correct_protocol5_error_message(self):
        array = np.arange(10)

    def test_record_array_with_object_dtype(self):
        my_object = object()
        arr_with_object = np.array(
            [(my_object, 1, 2.0)],
            dtype=[('a', object), ('b', int), ('c', float)])
        arr_without_object = np.array(
            [('xxx', 1, 2.0)],
            dtype=[('a', str), ('b', int), ('c', float)])
        for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
            depickled_arr_with_object = pickle.loads(
                pickle.dumps(arr_with_object, protocol=proto))
            depickled_arr_without_object = pickle.loads(
                pickle.dumps(arr_without_object, protocol=proto))
            assert_equal(arr_with_object.dtype,
                        depickled_arr_with_object.dtype)
            assert_equal(arr_without_object.dtype,
                        depickled_arr_without_object.dtype)

    @pytest.mark.skipif(pickle.HIGHEST_PROTOCOL < 5,
                       reason="requires pickle protocol 5")
    def test_f_contiguous_array(self):
        f_contiguous_array = np.array([[1, 2, 3], [4, 5, 6]], order='F')
        buffers = []
        bytes_string = pickle.dumps(f_contiguous_array, protocol=5,
                                    buffer_callback=buffers.append)
        assert len(buffers) > 0
        depickled_f_contiguous_array = pickle.loads(bytes_string,
  buffers=buffers)
        assert_equal(f_contiguous_array, depickled_f_contiguous_array)

    def test_non_contiguous_array(self):
        non_contiguous_array = np.arange(12).reshape(3, 4)[:, :2]
        assert not non_contiguous_array.flags.c_contiguous
        assert not non_contiguous_array.flags.f_contiguous
        for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
            depickled_non_contiguous_array = pickle.loads(
                pickle.dumps(non_contiguous_array, protocol=proto))
            assert_equal(non_contiguous_array, depickled_non_contiguous_array)

```

```

3238: (4)             def test_roundtrip(self):
3239: (8)                 for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
3240: (12)                     carray = np.array([[2, 9], [7, 0], [3, 8]])
3241: (12)                     DATA = [
3242: (16)                         carray,
3243: (16)                         np.transpose(carray),
3244: (16)                         np.array([('xxx', 1, 2.0)], dtype=[('a', (str, 3)), ('b',
int),
3245: (51)   ('c', float)])
3246: (12)
3247: (12)                     ]
3248: (12)                     refs = [weakref.ref(a) for a in DATA]
3249: (16)                     for a in DATA:
3250: (24)                         assert_equal(
3251: (24)                             a, pickle.loads(pickle.dumps(a, protocol=proto)),
3252: (12)                             err_msg="%r" % a)
3253: (12)                         del a, DATA, carray
3254: (12)                         break_cycles()
3255: (12)                         for ref in refs:
3256: (16)                             assert ref() is None
3257: (4)             def _loads(self, obj):
3258: (8)                 return pickle.loads(obj, encoding='latin1')
3259: (4)             def test_version0_int8(self):
3260: (8)                 s =
b'\x80\x02cnumpy.core._internal\n_reconstruct\nq\x01cnumpy\nndarray\nq\x02K\x00\x85U\x01b\x87Rq\x0
3(K\x04\x85cnumpy\ndtype\nq\x04U\x02i1K\x00K\x01\x87Rq\x05(U\x01|NNJ\xff\xff\xff\xffJ\xff\xff\xff\x
xfftb\x89U\x04\x01\x02\x03\x04tb.'
3261: (8)                 a = np.array([1, 2, 3, 4], dtype=np.int8)
3262: (8)                 p = self._loads(s)
3263: (8)                 assert_equal(a, p)
3264: (4)             def test_version0_float32(self):
3265: (8)                 s =
b'\x80\x02cnumpy.core._internal\n_reconstruct\nq\x01cnumpy\nndarray\nq\x02K\x00\x85U\x01b\x87Rq\x0
3(K\x04\x85cnumpy\ndtype\nq\x04U\x02f4K\x00K\x01\x87Rq\x05(U\x01<NNJ\xff\xff\xff\xffJ\xff\xff\xff\x
xfftb\x89U\x10\x00\x00\x80?\x00\x00\x00@\x00\x00@@\x00\x00\x80@tb.'
3266: (8)                 a = np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float32)
3267: (8)                 p = self._loads(s)
3268: (8)                 assert_equal(a, p)
3269: (4)             def test_version0_object(self):
3270: (8)                 s =
b'\x80\x02cnumpy.core._internal\n_reconstruct\nq\x01cnumpy\nndarray\nq\x02K\x00\x85U\x01b\x87Rq\x0
3(K\x02\x85cnumpy\ndtype\nq\x04U\x0208K\x00K\x01\x87Rq\x05(U\x01|NNJ\xff\xff\xff\xffJ\xff\xff\xff\x
xfftb\x89]q\x06({q\x07U\x01aK\x01s}q\x08U\x01bK\x02setb.'
3271: (8)                 a = np.array([{ 'a': 1}, { 'b': 2}])
3272: (8)                 p = self._loads(s)
3273: (8)                 assert_equal(a, p)
3274: (4)             def test_version1_int8(self):
3275: (8)                 s =
b'\x80\x02cnumpy.core._internal\n_reconstruct\nq\x01cnumpy\nndarray\nq\x02K\x00\x85U\x01b\x87Rq\x0
3(K\x01K\x04\x85cnumpy\ndtype\nq\x04U\x02i1K\x00K\x01\x87Rq\x05(K\x01U\x01|NNJ\xff\xff\xff\xffJ\xf
f\xff\xff\xfft\t\x89U\x04\x01\x02\x03\x04tb.'
3276: (8)                 a = np.array([1, 2, 3, 4], dtype=np.int8)
3277: (8)                 p = self._loads(s)
3278: (8)                 assert_equal(a, p)
3279: (4)             def test_version1_float32(self):
3280: (8)                 s =
b'\x80\x02cnumpy.core._internal\n_reconstruct\nq\x01cnumpy\nndarray\nq\x02K\x00\x85U\x01b\x87Rq\x0
3(K\x01K\x04\x85cnumpy\ndtype\nq\x04U\x02f4K\x00K\x01\x87Rq\x05(K\x01U\x01<NNJ\xff\xff\xff\xffJ\xf
f\xff\xff\xfft\t\x89U\x10\x00\x00\x80?\x00\x00\x00@\x00\x00@@\x00\x00\x80@tb.'
3281: (8)                 a = np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float32)
3282: (8)                 p = self._loads(s)
3283: (8)                 assert_equal(a, p)
3284: (4)             def test_version1_object(self):
3285: (8)                 s =
b'\x80\x02cnumpy.core._internal\n_reconstruct\nq\x01cnumpy\nndarray\nq\x02K\x00\x85U\x01b\x87Rq\x0
3(K\x01K\x02\x85cnumpy\ndtype\nq\x04U\x0208K\x00K\x01\x87Rq\x05(K\x01U\x01|NNJ\xff\xff\xff\xffJ\xf
f\xff\xff\xfft\t\x89]q\x06({q\x07U\x01aK\x01s}q\x08U\x01bK\x02setb.'
3286: (8)                 a = np.array([{ 'a': 1}, { 'b': 2}])
3287: (8)                 p = self._loads(s)
3288: (8)                 assert_equal(a, p)

```

```

3288: (4)             def test_subarray_int_shape(self):
3289: (8)                 s =
b"numpy.core.multiarray\n_reconstruct\np0\n(numpy\nndarray\np1\n(I0\\ntp2\\nS'b'\\np3\\ntp4\\nRp5\\n(I
1\\n(I1\\ntp6\\ncnumpy\\ndtype\\np7\\n(S'V6'\\np8\\nI0\\nI1\\ntp9\\nRp10\\n(I3\\nS'|'\\np11\\n(S'a'\\np12\\ng3\\ntp
13\\n(dp14\\ng12\\n(g7\\n(S'V4'\\np15\\nI0\\nI1\\ntp16\\nRp17\\n(I3\\nS'|'\\np18\\n(g7\\n(S'i1'\\np19\\nI0\\nI1\\ntp
20\\nRp21\\n(I3\\nS'|'\\np22\\nNNNI-1\\nI-
1\\nI0\\ntp23\\nb(I2\\nI2\\ntp24\\ntp25\\nNNI4\\nI1\\nI0\\ntp26\\nbI0\\ntp27\\nsg3\\n(g7\\n(S'V2'\\np28\\nI0\\nI1\\nt
p29\\nRp30\\n(I3\\nS'|'\\np31\\n(g21\\nI2\\ntp32\\nNNI2\\nI1\\nI0\\ntp33\\nbI4\\ntp34\\nsI6\\nI1\\nI0\\ntp35\\nbI00\\
nS'\\x01\\x01\\x01\\x02'\\np36\\ntp37\\nb."
3290: (8)                 a = np.array([(1, (1, 2))], dtype=[('a', 'i1', (2, 2)), ('b', 'i1',
2)])
3291: (8)                 p = self._loads(s)
3292: (8)                 assert_equal(a, p)
3293: (4)             def test_datetime64_byteorder(self):
3294: (8)                 original = np.array([[2015-02-24T00:00:00.000000000]]),
dtype='datetime64[ns]')
3295: (8)                 original_byte_reversed = original.copy(order='K')
3296: (8)                 original_byte_reversed.dtype =
original_byte_reversed.dtype.newbyteorder('S')
3297: (8)                 original_byte_reversed.byteswap(inplace=True)
3298: (8)                 new = pickle.loads(pickle.dumps(original_byte_reversed))
3299: (8)                 assert_equal(original.dtype, new.dtype)
3300: (0)             class TestFancyIndexing:
3301: (4)                 def test_list(self):
3302: (8)                     x = np.ones((1, 1))
3303: (8)                     x[:, [0]] = 2.0
3304: (8)                     assert_array_equal(x, np.array([[2.0]]))
3305: (8)                     x = np.ones((1, 1, 1))
3306: (8)                     x[:, :, [0]] = 2.0
3307: (8)                     assert_array_equal(x, np.array([[[2.0]]]))
3308: (4)                 def test_tuple(self):
3309: (8)                     x = np.ones((1, 1))
3310: (8)                     x[:, (0,)] = 2.0
3311: (8)                     assert_array_equal(x, np.array([[2.0]]))
3312: (8)                     x = np.ones((1, 1, 1))
3313: (8)                     x[:, :, (0,)] = 2.0
3314: (8)                     assert_array_equal(x, np.array([[[2.0]]]))
3315: (4)                 def test_mask(self):
3316: (8)                     x = np.array([1, 2, 3, 4])
3317: (8)                     m = np.array([0, 1, 0, 0], bool)
3318: (8)                     assert_array_equal(x[m], np.array([2]))
3319: (4)                 def test_mask2(self):
3320: (8)                     x = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
3321: (8)                     m = np.array([0, 1], bool)
3322: (8)                     m2 = np.array([[0, 1, 0, 0], [1, 0, 0, 0]], bool)
3323: (8)                     m3 = np.array([[0, 1, 0, 0], [0, 0, 0, 0]], bool)
3324: (8)                     assert_array_equal(x[m], np.array([[5, 6, 7, 8]]))
3325: (8)                     assert_array_equal(x[m2], np.array([2, 5]))
3326: (8)                     assert_array_equal(x[m3], np.array([2]))
3327: (4)                 def test_assign_mask(self):
3328: (8)                     x = np.array([1, 2, 3, 4])
3329: (8)                     m = np.array([0, 1, 0, 0], bool)
3330: (8)                     x[m] = 5
3331: (8)                     assert_array_equal(x, np.array([1, 5, 3, 4]))
3332: (4)                 def test_assign_mask2(self):
3333: (8)                     xorig = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
3334: (8)                     m = np.array([0, 1], bool)
3335: (8)                     m2 = np.array([[0, 1, 0, 0], [1, 0, 0, 0]], bool)
3336: (8)                     m3 = np.array([[0, 1, 0, 0], [0, 0, 0, 0]], bool)
3337: (8)                     x = xorig.copy()
3338: (8)                     x[m] = 10
3339: (8)                     assert_array_equal(x, np.array([[1, 2, 3, 4], [10, 10, 10, 10]]))
3340: (8)                     x = xorig.copy()
3341: (8)                     x[m2] = 10
3342: (8)                     assert_array_equal(x, np.array([[1, 10, 3, 4], [10, 6, 7, 8]]))
3343: (8)                     x = xorig.copy()
3344: (8)                     x[m3] = 10
3345: (8)                     assert_array_equal(x, np.array([[1, 10, 3, 4], [5, 6, 7, 8]]))
3346: (0)             class TestStringCompare:

```

```

3347: (4)
3348: (8)
3349: (8)
3350: (8)
3351: (8)
3352: (8)
3353: (8)
3354: (8)
3355: (8)
3356: (4)
3357: (8)
3358: (8)
3359: (8)
3360: (8)
3361: (8)
3362: (8)
3363: (8)
3364: (8)
3365: (4)
3366: (8)
3367: (8)
3368: (8)
3369: (8)
3370: (8)
3371: (8)
3372: (8)
3373: (8)
3374: (0)
3375: (4)
3376: (13)
3377: (13)
3378: (13)
3379: (13)
3380: (4)
3381: (8)
3382: (8)
3383: (4)
3384: (4)
3385: (8)
3386: (8)
3387: (12)
3388: (8)
3389: (12)
3390: (12)
3391: (8)
3392: (8)
3393: (8)
3394: (8)
3395: (8)
3396: (8)
3397: (8)
3398: (8)
3399: (28)
3400: (8)
3401: (8)
3402: (8)
3403: (12)
3404: (12)
3405: (16)
3406: (12)
3407: (16)
3408: (12)
3409: (12)
3410: (16)
3411: (24)
3412: (8)
3413: (12)
3414: (8)
3415: (12)

        def test_string(self):
            g1 = np.array(["This", "is", "example"])
            g2 = np.array(["This", "was", "example"])
            assert_array_equal(g1 == g2, [g1[i] == g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 != g2, [g1[i] != g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 <= g2, [g1[i] <= g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 >= g2, [g1[i] >= g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 < g2, [g1[i] < g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 > g2, [g1[i] > g2[i] for i in [0, 1, 2]])

        def test_mixed(self):
            g1 = np.array(["spam", "spa", "spammer", "and eggs"])
            g2 = "spam"
            assert_array_equal(g1 == g2, [x == g2 for x in g1])
            assert_array_equal(g1 != g2, [x != g2 for x in g1])
            assert_array_equal(g1 < g2, [x < g2 for x in g1])
            assert_array_equal(g1 > g2, [x > g2 for x in g1])
            assert_array_equal(g1 <= g2, [x <= g2 for x in g1])
            assert_array_equal(g1 >= g2, [x >= g2 for x in g1])

        def test_unicode(self):
            g1 = np.array(["This", "is", "example"])
            g2 = np.array(["This", "was", "example"])
            assert_array_equal(g1 == g2, [g1[i] == g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 != g2, [g1[i] != g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 <= g2, [g1[i] <= g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 >= g2, [g1[i] >= g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 < g2, [g1[i] < g2[i] for i in [0, 1, 2]])
            assert_array_equal(g1 > g2, [g1[i] > g2[i] for i in [0, 1, 2]])

    class TestArgmaxArgminCommon:
        sizes = [(), (3,), (3, 2), (2, 3),
                 (3, 3), (2, 3, 4), (4, 3, 2),
                 (1, 2, 3, 4), (2, 3, 4, 1),
                 (3, 4, 1, 2), (4, 1, 2, 3),
                 (64,), (128,), (256,)]
        @pytest.mark.parametrize("size, axis", itertools.chain(*[[size, axis]
            for axis in list(range(-len(size), len(size))) + [None]]
            for size in sizes)))
        @pytest.mark.parametrize('method', [np.argmax, np.argmin])
        def test_np_argmin_argmax_keepdims(self, size, axis, method):
            arr = np.random.normal(size=size)
            if axis is None:
                new_shape = [1 for _ in range(len(size))]
            else:
                new_shape = list(size)
                new_shape[axis] = 1
            new_shape = tuple(new_shape)
            _res_orig = method(arr, axis=axis)
            res_orig = _res_orig.reshape(new_shape)
            res = method(arr, axis=axis, keepdims=True)
            assert_equal(res, res_orig)
            assert_(res.shape == new_shape)
            outarray = np.empty(res.shape, dtype=res.dtype)
            res1 = method(arr, axis=axis, out=outarray,
                          keepdims=True)
            assert_(res1 is outarray)
            assert_equal(res, outarray)
            if len(size) > 0:
                wrong_shape = list(new_shape)
                if axis is not None:
                    wrong_shape[axis] = 2
                else:
                    wrong_shape[0] = 2
                wrong_outarray = np.empty(wrong_shape, dtype=res.dtype)
                with pytest.raises(ValueError):
                    method(arr.T, axis=axis,
                           out=wrong_outarray, keepdims=True)
            if axis is None:
                new_shape = [1 for _ in range(len(size))]
            else:
                new_shape = list(size)[::-1]

```

```

3416: (12)
3417: (8)
3418: (8)
3419: (8)
3420: (8)
3421: (8)
3422: (8)
3423: (8)
3424: (8)
3425: (8)
3426: (28)
3427: (8)
3428: (8)
3429: (8)
3430: (12)
3431: (16)
3432: (24)
3433: (8)
3434: (12)
3435: (12)
3436: (16)
3437: (12)
3438: (16)
3439: (12)
3440: (12)
3441: (16)
3442: (24)
3443: (4)
3444: (4)
def test_all(self, method):
3445: (8)
3446: (8)
3447: (8)
3448: (8)
3449: (12)
3450: (12)
3451: (12)
3452: (12)
3453: (12)
3454: (40)
3455: (4)
3456: (4)
3457: (8)
3458: (8)
3459: (8)
3460: (8)
3461: (8)
3462: (8)
3463: (8)
3464: (8)
3465: (8)
3466: (8)
3467: (8)
3468: (4)
3469: (4)
3470: (4)
def test_output_shape(self, method):
3471: (8)
3472: (8)
3473: (8)
3474: (8)
3475: (8)
3476: (4)
3477: (8)
3478: (9)
3479: (4)
3480: (8)
3481: (8)
3482: (8)
3483: (8)
3484: (4)
    new_shape[axis] = 1
    new_shape = tuple(new_shape)
    _res_orig = method(arr.T, axis=axis)
    res_orig = _res_orig.reshape(new_shape)
    res = method(arr.T, axis=axis, keepdims=True)
    assert_equal(res, res_orig)
    assert_(res.shape == new_shape)
    outarray = np.empty(new_shape[::-1], dtype=res.dtype)
    outarray = outarray.T
    res1 = method(arr.T, axis=axis, out=outarray,
                  keepdims=True)
    assert_(res1 is outarray)
    assert_equal(res, outarray)
    if len(size) > 0:
        with pytest.raises(ValueError):
            method(arr[0], axis=axis,
                   out=outarray, keepdims=True)
    if len(size) > 0:
        wrong_shape = list(new_shape)
        if axis is not None:
            wrong_shape[axis] = 2
        else:
            wrong_shape[0] = 2
        wrong_outarray = np.empty(wrong_shape, dtype=res.dtype)
        with pytest.raises(ValueError):
            method(arr.T, axis=axis,
                   out=wrong_outarray, keepdims=True)
@ pytest.mark.parametrize('method', ['max', 'min'])
def test_all(self, method):
    a = np.random.normal(0, 1, (4, 5, 6, 7, 8))
    arg_method = getattr(a, 'arg' + method)
    val_method = getattr(a, method)
    for i in range(a.ndim):
        a_maxmin = val_method(i)
        aarg_maxmin = arg_method(i)
        axes = list(range(a.ndim))
        axes.remove(i)
        assert_(np.all(a_maxmin == aarg_maxmin.choose(
                      *a.transpose(i, *axes))))
@ pytest.mark.parametrize('method', ['argmax', 'argmin'])
def test_output_shape(self, method):
    a = np.ones((10, 5))
    arg_method = getattr(a, method)
    out = np.ones(11, dtype=np.int_)
    assert_raises(ValueError, arg_method, -1, out)
    out = np.ones((2, 5), dtype=np.int_)
    assert_raises(ValueError, arg_method, -1, out)
    out = np.ones((1, 10), dtype=np.int_)
    assert_raises(ValueError, arg_method, -1, out)
    out = np.ones(10, dtype=np.int_)
    arg_method(-1, out=out)
    assert_equal(out, arg_method(-1))
@ pytest.mark.parametrize('ndim', [0, 1])
@ pytest.mark.parametrize('method', ['argmax', 'argmin'])
def test_ret_is_out(self, ndim, method):
    a = np.ones((4,) + (256,)*ndim)
    arg_method = getattr(a, method)
    out = np.empty((256,)*ndim, dtype=np.intp)
    ret = arg_method(axis=0, out=out)
    assert ret is out
@ pytest.mark.parametrize('np_array, method, idx, val',
                        [(np.zeros, 'argmax', 5942, "as"),
                         (np.ones, 'argmin', 6001, "0")])
def test_unicode(self, np_array, method, idx, val):
    d = np_array(6031, dtype='<U9')
    arg_method = getattr(d, method)
    d[idx] = val
    assert_equal(arg_method(), idx)
@ pytest.mark.parametrize('arr_method, np_method',

```

```

3485: (8)          [('argmax', np.argmax),
3486: (9)            ('argmin', np.argmin)])
3487: (4)          def test_np_vs_ndarray(self, arr_method, np_method):
3488: (8)            a = np.random.normal(size=(2, 3))
3489: (8)            arg_method = getattr(a, arr_method)
3490: (8)            out1 = np.zeros(2, dtype=int)
3491: (8)            out2 = np.zeros(2, dtype=int)
3492: (8)            assert_equal(arg_method(1, out1), np_method(a, 1, out2))
3493: (8)            assert_equal(out1, out2)
3494: (8)            out1 = np.zeros(3, dtype=int)
3495: (8)            out2 = np.zeros(3, dtype=int)
3496: (8)            assert_equal(arg_method(out=out1, axis=0),
3497: (21)                  np_method(a, out=out2, axis=0))
3498: (8)            assert_equal(out1, out2)
3499: (4)          @pytest.mark.leaks_references(reason="replaces None with NULL.")
3500: (4)          @pytest.mark.parametrize('method, vals',
3501: (8)            [('argmax', (10, 30)),
3502: (9)              ('argmin', (30, 10))])
3503: (4)          def test_object_with_NULLs(self, method, vals):
3504: (8)            a = np.empty(4, dtype='O')
3505: (8)            arg_method = getattr(a, method)
3506: (8)            ctypes.memset(a.ctypes.data, 0, a.nbytes)
3507: (8)            assert_equal(arg_method(), 0)
3508: (8)            a[3] = vals[0]
3509: (8)            assert_equal(arg_method(), 3)
3510: (8)            a[1] = vals[1]
3511: (8)            assert_equal(arg_method(), 1)
3512: (0)          class TestArgmax:
3513: (4)            usg_data = [
3514: (8)              ([1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0], 0),
3515: (8)              ([3, 3, 3, 3, 2, 2, 2, 2, 2], 0),
3516: (8)              ([0, 1, 2, 3, 4, 5, 6, 7], 7),
3517: (8)              ([7, 6, 5, 4, 3, 2, 1, 0], 0)
3518: (4)            ]
3519: (4)            sg_data = usg_data + [
3520: (8)              ([1, 2, 3, 4, -4, -3, -2, -1], 3),
3521: (8)              ([1, 2, 3, 4, -1, -2, -3, -4], 3)
3522: (4)            ]
3523: (4)            darr = [(np.array(d[0], dtype=t), d[1]) for d, t in (
3524: (8)              itertools.product(usg_data,
3525: (12)                np.uint8, np.uint16, np.uint32, np.uint64
3526: (8)              ))
3527: (4)            ]
3528: (4)            darr = darr + [(np.array(d[0], dtype=t), d[1]) for d, t in (
3529: (8)              itertools.product(sg_data,
3530: (12)                np.int8, np.int16, np.int32, np.int64, np.float32, np.float64
3531: (8)              ))
3532: (4)            ]
3533: (4)            darr = darr + [(np.array(d[0], dtype=t), d[1]) for d, t in (
3534: (8)              itertools.product(
3535: (12)                ([0, 1, 2, 3, np.nan], 4),
3536: (12)                ([0, 1, 2, np.nan, 3], 3),
3537: (12)                ([np.nan, 0, 1, 2, 3], 0),
3538: (12)                ([np.nan, 0, np.nan, 2, 3], 0),
3539: (12)                ([1] * (2**5-1) + [np.nan], 2**5-1),
3540: (12)                ([1] * (4**5-1) + [np.nan], 4**5-1),
3541: (12)                ([1] * (8**5-1) + [np.nan], 8**5-1),
3542: (12)                ([1] * (16**5-1) + [np.nan], 16**5-1),
3543: (12)                ([1] * (32**5-1) + [np.nan], 32**5-1)
3544: (8)              ), (
3545: (12)                np.float32, np.float64
3546: (8)              ))
3547: (4)            )
3548: (4)            nan_arr = darr + [
3549: (8)              ([0, 1, 2, 3, complex(0, np.nan)], 4),
3550: (8)              ([0, 1, 2, 3, complex(np.nan, 0)], 4),
3551: (8)              ([0, 1, 2, complex(np.nan, 0), 3], 3),
3552: (8)              ([0, 1, 2, complex(0, np.nan), 3], 3),
3553: (8)              ([complex(0, np.nan), 0, 1, 2, 3], 0),

```

```

3554: (8)
3555: (8)
3556: (8)
0),
3557: (8)
0),
3558: (8)
3559: (8)
3560: (8)
3561: (8)
3562: (10)
3563: (10)
3564: (10)
3565: (10)
3566: (10)
3567: (8)
3568: (10)
3569: (10)
3570: (10)
3571: (10)
3572: (10)
3573: (8)
3574: (10)
3575: (10)
3576: (10)
3577: (10)
3578: (10)
3579: (8)
3580: (10)
3581: (10)
3582: (10)
3583: (10)
3584: (10)
3585: (8)
3586: (10)
3587: (10)
3588: (10)
3589: (8)
3590: (8)
3591: (10)
3592: (8)
3593: (10)
3594: (8)
3595: (10)
3596: (8)
3597: (8)
3598: (8)
3599: (8)
3600: (4)
3601: (4)
3602: (4)
3603: (8)
3604: (8)
3605: (12)
3606: (24)
3607: (12)
3608: (8)
3609: (8)
3610: (8)
3611: (8)
3612: (8)
3613: (8)
3614: (8)
3615: (8)
3616: (8)
3617: (8)
3618: (8)
3619: (4)
3620: (8)

    ([complex(np.nan, np.nan), 0, 1, 2, 3], 0),
    ([complex(np.nan, 0), complex(np.nan, 2), complex(np.nan, 1)], 0),
    ([complex(np.nan, np.nan), complex(np.nan, 2), complex(np.nan, 1)], 0),
    ([complex(np.nan, 0), complex(np.nan, 2), complex(np.nan, np.nan)], 0),
    ([complex(0, 0), complex(0, 2), complex(0, 1)], 1),
    ([complex(1, 0), complex(0, 2), complex(0, 1)], 0),
    ([complex(1, 0), complex(0, 2), complex(1, 1)], 2),
    ([np.datetime64('1923-04-14T12:43:12'),
      np.datetime64('1994-06-21T14:43:15'),
      np.datetime64('2001-10-15T04:10:32'),
      np.datetime64('1995-11-25T16:02:16'),
      np.datetime64('2005-01-04T03:14:12'),
      np.datetime64('2041-12-03T14:05:03')], 5),
    ([np.datetime64('1935-09-14T04:40:11'),
      np.datetime64('1949-10-12T12:32:11'),
      np.datetime64('2010-01-03T05:14:12'),
      np.datetime64('2015-11-20T12:20:59'),
      np.datetime64('1932-09-23T10:10:13'),
      np.datetime64('2014-10-10T03:50:30')], 3),
    ([np.datetime64('NaT'),
      np.datetime64('NaT'),
      np.datetime64('2010-01-03T05:14:12'),
      np.datetime64('NaT'),
      np.datetime64('2015-09-23T10:10:13'),
      np.datetime64('1932-10-10T03:50:30')], 0),
    ([np.datetime64('2059-03-14T12:43:12'),
      np.datetime64('1996-09-21T14:43:15'),
      np.datetime64('NaT'),
      np.datetime64('2022-12-25T16:02:16'),
      np.datetime64('1963-10-04T03:14:12'),
      np.datetime64('2013-05-08T18:15:23')], 2),
    ([np.timedelta64(2, 's'),
      np.timedelta64(1, 's'),
      np.timedelta64('NaT', 's'),
      np.timedelta64(3, 's')], 2),
    ([np.timedelta64('NaT', 's')] * 3, 0),
    ([timedelta(days=5, seconds=14), timedelta(days=2, seconds=35),
      timedelta(days=-1, seconds=23)], 0),
    ([timedelta(days=1, seconds=43), timedelta(days=10, seconds=5),
      timedelta(days=5, seconds=14)], 1),
    ([timedelta(days=10, seconds=24), timedelta(days=10, seconds=5),
      timedelta(days=10, seconds=43)], 2),
    ([False, False, False, False, True], 4),
    ([False, False, False, True, False], 3),
    ([True, False, False, False, False], 0),
    ([True, False, True, False, False], 0),
    ])

    @pytest.mark.parametrize('data', nan_arr)
def test_combinations(self, data):
    arr, pos = data
    with suppress_warnings() as sup:
        sup.filter(RuntimeWarning,
                   "invalid value encountered in reduce")
        val = np.max(arr)
    assert_equal(np.argmax(arr), pos, err_msg="%r % arr")
    assert_equal(arr[np.argmax(arr)], val, err_msg="%r % arr")
    rarr = np.repeat(arr, 129)
    rpos = pos * 129
    assert_equal(np.argmax(rarr), rpos, err_msg="%r % rarr)
    assert_equal(rarr[np.argmax(rarr)], val, err_msg="%r % rarr)
    padd = np.repeat(np.min(arr), 513)
    rarr = np.concatenate((arr, padd))
    rpos = pos
    assert_equal(np.argmax(rarr), rpos, err_msg="%r % rarr)
    assert_equal(rarr[np.argmax(rarr)], val, err_msg="%r % rarr)

def test_maximum_signed_integers(self):
    a = np.array([1, 2**7 - 1, -2**7], dtype=np.int8)

```

```

3621: (8) assert_equal(np.argmax(a), 1)
3622: (8) a = a.repeat(129)
3623: (8) assert_equal(np.argmax(a), 129)
3624: (8) a = np.array([1, 2**15 - 1, -2**15], dtype=np.int16)
3625: (8) assert_equal(np.argmax(a), 1)
3626: (8) a = a.repeat(129)
3627: (8) assert_equal(np.argmax(a), 129)
3628: (8) a = np.array([1, 2**31 - 1, -2**31], dtype=np.int32)
3629: (8) assert_equal(np.argmax(a), 1)
3630: (8) a = a.repeat(129)
3631: (8) assert_equal(np.argmax(a), 129)
3632: (8) a = np.array([1, 2**63 - 1, -2**63], dtype=np.int64)
3633: (8) assert_equal(np.argmax(a), 1)
3634: (8) a = a.repeat(129)
3635: (8) assert_equal(np.argmax(a), 129)
3636: (0) class TestArgmin:
3637: (4)     usg_data = [
3638: (8)         ([1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0], 8),
3639: (8)         ([3, 3, 3, 3, 2, 2, 2, 2], 4),
3640: (8)         ([0, 1, 2, 3, 4, 5, 6, 7], 0),
3641: (8)         ([7, 6, 5, 4, 3, 2, 1, 0], 7)
3642: (4)     ]
3643: (4)     sg_data = usg_data + [
3644: (8)         ([1, 2, 3, 4, -4, -3, -2, -1], 4),
3645: (8)         ([1, 2, 3, 4, -1, -2, -3, -4], 7)
3646: (4)     ]
3647: (4)     darr = [(np.array(d[0], dtype=t), d[1]) for d, t in (
3648: (8)         itertools.product(usg_data, (
3649: (12)             np.uint8, np.uint16, np.uint32, np.uint64
3650: (8)         )))
3651: (4)     ]
3652: (4)     darr = darr + [(np.array(d[0], dtype=t), d[1]) for d, t in (
3653: (8)         itertools.product(sg_data, (
3654: (12)             np.int8, np.int16, np.int32, np.int64, np.float32, np.float64
3655: (8)         )))
3656: (4)     ]
3657: (4)     darr = darr + [(np.array(d[0], dtype=t), d[1]) for d, t in (
3658: (8)         itertools.product(
3659: (12)             ([0, 1, 2, 3, np.nan], 4),
3660: (12)             ([0, 1, 2, np.nan, 3], 3),
3661: (12)             ([np.nan, 0, 1, 2, 3], 0),
3662: (12)             ([np.nan, 0, np.nan, 2, 3], 0),
3663: (12)             ([1] * (2**5-1) + [np.nan], 2**5-1),
3664: (12)             ([1] * (4**5-1) + [np.nan], 4**5-1),
3665: (12)             ([1] * (8**5-1) + [np.nan], 8**5-1),
3666: (12)             ([1] * (16**5-1) + [np.nan], 16**5-1),
3667: (12)             ([1] * (32**5-1) + [np.nan], 32**5-1)
3668: (8)         ),
3669: (12)             np.float32, np.float64
3670: (8)         )))
3671: (4)     ]
3672: (4)     nan_arr = darr + [
3673: (8)         ([0, 1, 2, 3, complex(0, np.nan)], 4),
3674: (8)         ([0, 1, 2, 3, complex(np.nan, 0)], 4),
3675: (8)         ([0, 1, 2, complex(np.nan, 0), 3], 3),
3676: (8)         ([0, 1, 2, complex(0, np.nan), 3], 3),
3677: (8)         ([complex(0, np.nan), 0, 1, 2, 3], 0),
3678: (8)         ([complex(np.nan, np.nan), 0, 1, 2, 3], 0),
3679: (8)         ([complex(np.nan, 0), complex(np.nan, 2), complex(np.nan, 1)], 0),
3680: (8)         ([complex(np.nan, np.nan), complex(np.nan, 2), complex(np.nan, 1)], 0),
3681: (8)         ([complex(np.nan, 0), complex(np.nan, 2), complex(np.nan, np.nan)], 0),
3682: (8)         ([complex(0, 0), complex(0, 2), complex(0, 1)], 0),
3683: (8)         ([complex(1, 0), complex(0, 2), complex(0, 1)], 2),
3684: (8)         ([complex(1, 0), complex(0, 2), complex(1, 1)], 1),
3685: (8)         ([np.datetime64('1923-04-14T12:43:12'),
3686: (10)             np.datetime64('1994-06-21T14:43:15'),
3687: (10)             np.datetime64('2001-10-15T04:10:32'),

```

```

3688: (10) np.datetime64('1995-11-25T16:02:16'),
3689: (10) np.datetime64('2005-01-04T03:14:12'),
3690: (10) np.datetime64('2041-12-03T14:05:03')], 0),
3691: (8) ([np.datetime64('1935-09-14T04:40:11'),
3692: (10) np.datetime64('1949-10-12T12:32:11'),
3693: (10) np.datetime64('2010-01-03T05:14:12'),
3694: (10) np.datetime64('2014-11-20T12:20:59'),
3695: (10) np.datetime64('2015-09-23T10:10:13'),
3696: (10) np.datetime64('1932-10-10T03:50:30')], 5),
3697: (8) ([np.datetime64('NaT'),
3698: (10) np.datetime64('NaT'),
3699: (10) np.datetime64('2010-01-03T05:14:12'),
3700: (10) np.datetime64('NaT'),
3701: (10) np.datetime64('2015-09-23T10:10:13'),
3702: (10) np.datetime64('1932-10-10T03:50:30')], 0),
3703: (8) ([np.datetime64('2059-03-14T12:43:12'),
3704: (10) np.datetime64('1996-09-21T14:43:15'),
3705: (10) np.datetime64('NaT'),
3706: (10) np.datetime64('2022-12-25T16:02:16'),
3707: (10) np.datetime64('1963-10-04T03:14:12'),
3708: (10) np.datetime64('2013-05-08T18:15:23')], 2),
3709: (8) ([np.timedelta64(2, 's'),
3710: (10) np.timedelta64(1, 's'),
3711: (10) np.timedelta64('NaT', 's'),
3712: (10) np.timedelta64(3, 's')], 2),
3713: (8) ([np.timedelta64('NaT', 's')] * 3, 0),
3714: (8) ([timedelta(days=5, seconds=14), timedelta(days=2, seconds=35),
3715: (10) timedelta(days=-1, seconds=23)], 2),
3716: (8) ([timedelta(days=1, seconds=43), timedelta(days=10, seconds=5),
3717: (10) timedelta(days=5, seconds=14)], 0),
3718: (8) ([timedelta(days=10, seconds=24), timedelta(days=10, seconds=5),
3719: (10) timedelta(days=10, seconds=43)], 1),
3720: (8) ([True, True, True, True, False], 4),
3721: (8) ([True, True, True, False, True], 3),
3722: (8) ([False, True, True, True, True], 0),
3723: (8) ([False, True, False, True, True], 0),
3724: (4)
3725: (4) @pytest.mark.parametrize('data', nan_arr)
3726: (4) def test_combinations(self, data):
3727: (8) arr, pos = data
3728: (8) with suppress_warnings() as sup:
3729: (12)     sup.filter(RuntimeWarning,
3730: (23)             "invalid value encountered in reduce")
3731: (12)     min_val = np.min(arr)
3732: (8)     assert_equal(np.argmin(arr), pos, err_msg="%r" % arr)
3733: (8)     assert_equal(arr[np.argmin(arr)], min_val, err_msg="%r" % arr)
3734: (8)     rarr = np.repeat(arr, 129)
3735: (8)     rpos = pos * 129
3736: (8)     assert_equal(np.argmin(rarr), rpos, err_msg="%r" % rarr)
3737: (8)     assert_equal(rarr[np.argmin(rarr)], min_val, err_msg="%r" % rarr)
3738: (8)     padd = np.repeat(np.max(arr), 513)
3739: (8)     rarr = np.concatenate((arr, padd))
3740: (8)     rpos = pos
3741: (8)     assert_equal(np.argmin(rarr), rpos, err_msg="%r" % rarr)
3742: (8)     assert_equal(rarr[np.argmin(rarr)], min_val, err_msg="%r" % rarr)
3743: (4) def test_minimum_signed_integers(self):
3744: (8)     a = np.array([1, -2**7, -2**7 + 1, 2**7 - 1], dtype=np.int8)
3745: (8)     assert_equal(np.argmin(a), 1)
3746: (8)     a = a.repeat(129)
3747: (8)     assert_equal(np.argmin(a), 129)
3748: (8)     a = np.array([1, -2**15, -2**15 + 1, 2**15 - 1], dtype=np.int16)
3749: (8)     assert_equal(np.argmin(a), 1)
3750: (8)     a = a.repeat(129)
3751: (8)     assert_equal(np.argmin(a), 129)
3752: (8)     a = np.array([1, -2**31, -2**31 + 1, 2**31 - 1], dtype=np.int32)
3753: (8)     assert_equal(np.argmin(a), 1)
3754: (8)     a = a.repeat(129)
3755: (8)     assert_equal(np.argmin(a), 129)
3756: (8)     a = np.array([1, -2**63, -2**63 + 1, 2**63 - 1], dtype=np.int64)

```

```

3757: (8)             assert_equal(np.argmin(a), 1)
3758: (8)             a = a.repeat(129)
3759: (8)             assert_equal(np.argmin(a), 129)
3760: (0)          class TestMinMax:
3761: (4)              def test_scalar(self):
3762: (8)                  assert_raises(np.AxisError, npamax, 1, 1)
3763: (8)                  assert_raises(np.AxisError, npamin, 1, 1)
3764: (8)                  assert_equal(npamax(1, axis=0), 1)
3765: (8)                  assert_equal(npamin(1, axis=0), 1)
3766: (8)                  assert_equal(npamax(1, axis=None), 1)
3767: (8)                  assert_equal(npamin(1, axis=None), 1)
3768: (4)              def test_axis(self):
3769: (8)                  assert_raises(np.AxisError, npamax, [1, 2, 3], 1000)
3770: (8)                  assert_equal(npamax([[1, 2, 3]], axis=1), 3)
3771: (4)              def test_datetime(self):
3772: (8)                  for dtype in ('m8[s]', 'm8[Y]'):
3773: (12)                      a = np.arange(10).astype(dtype)
3774: (12)                      assert_equal(npamin(a), a[0])
3775: (12)                      assert_equal(npamax(a), a[9])
3776: (12)                      a[3] = 'NaT'
3777: (12)                      assert_equal(npamin(a), a[3])
3778: (12)                      assert_equal(npamax(a), a[3])
3779: (0)          class TestNewaxis:
3780: (4)              def test_basic(self):
3781: (8)                  sk = np.array([0, -0.1, 0.1])
3782: (8)                  res = 250*sk[:, np.newaxis]
3783: (8)                  assert_almost_equal(res.ravel(), 250*sk)
3784: (0)          class TestClip:
3785: (4)              def _check_range(self, x, cmin, cmax):
3786: (8)                  assert_(np.all(x >= cmin))
3787: (8)                  assert_(np.all(x <= cmax))
3788: (4)              def _clip_type(self, type_group, array_max,
3789: (19)                  clip_min, clip_max, inplace=False,
3790: (19)                  expected_min=None, expected_max=None):
3791: (8)                  if expected_min is None:
3792: (12)                      expected_min = clip_min
3793: (8)                  if expected_max is None:
3794: (12)                      expected_max = clip_max
3795: (8)                  for T in np.sctypes[type_group]:
3796: (12)                      if sys.byteorder == 'little':
3797: (16)                          byte_orders = ['=', '>']
3798: (12)                      else:
3799: (16)                          byte_orders = ['<', '=']
3800: (12)                  for byteorder in byte_orders:
3801: (16)                      dtype = np.dtype(T).newbyteorder(byteorder)
3802: (16)                      x = (np.random.random(1000) * array_max).astype(dtype)
3803: (16)                      if inplace:
3804: (20)                          x.clip(clip_min, clip_max, x, casting='unsafe')
3805: (16)                      else:
3806: (20)                          x = x.clip(clip_min, clip_max)
3807: (20)                          byteorder = '='
3808: (16)                          if x.dtype.byteorder == '|':
3809: (20)                              byteorder = '|'
3810: (16)                          assert_equal(x.dtype.byteorder, byteorder)
3811: (16)                          self._check_range(x, expected_min, expected_max)
3812: (8)                  return x
3813: (4)              def test_basic(self):
3814: (8)                  for inplace in [False, True]:
3815: (12)                      self._clip_type(
3816: (16)                          'float', 1024, -12.8, 100.2, inplace=inplace)
3817: (12)                      self._clip_type(
3818: (16)                          'float', 1024, 0, 0, inplace=inplace)
3819: (12)                      self._clip_type(
3820: (16)                          'int', 1024, -120, 100, inplace=inplace)
3821: (12)                      self._clip_type(
3822: (16)                          'int', 1024, 0, 0, inplace=inplace)
3823: (12)                      self._clip_type(
3824: (16)                          'uint', 1024, 0, 0, inplace=inplace)
3825: (12)                      self._clip_type(

```

```

3826: (16)             'uint', 1024, -120, 100, inplace=inplace, expected_min=0)
3827: (4)          def test_record_array(self):
3828: (8)              rec = np.array([(-5, 2.0, 3.0), (5.0, 4.0, 3.0)],
3829: (23)                      dtype=[('x', '<f8'), ('y', '>f8'), ('z', '<f8')])
3830: (8)              y = rec['x'].clip(-0.3, 0.5)
3831: (8)              self._check_range(y, -0.3, 0.5)
3832: (4)          def test_max_or_min(self):
3833: (8)              val = np.array([0, 1, 2, 3, 4, 5, 6, 7])
3834: (8)              x = val.clip(3)
3835: (8)              assert_(np.all(x >= 3))
3836: (8)              x = val.clip(min=3)
3837: (8)              assert_(np.all(x >= 3))
3838: (8)              x = val.clip(max=4)
3839: (8)              assert_(np.all(x <= 4))
3840: (4)          def test_nan(self):
3841: (8)              input_arr = np.array([-2., np.nan, 0.5, 3., 0.25, np.nan])
3842: (8)              result = input_arr.clip(-1, 1)
3843: (8)              expected = np.array([-1., np.nan, 0.5, 1., 0.25, np.nan])
3844: (8)              assert_array_equal(result, expected)
3845: (0)      class TestCompress:
3846: (4)          def test_axis(self):
3847: (8)              tgt = [[5, 6, 7, 8, 9]]
3848: (8)              arr = np.arange(10).reshape(2, 5)
3849: (8)              out = np.compress([0, 1], arr, axis=0)
3850: (8)              assert_equal(out, tgt)
3851: (8)              tgt = [[1, 3], [6, 8]]
3852: (8)              out = np.compress([0, 1, 0, 1, 0], arr, axis=1)
3853: (8)              assert_equal(out, tgt)
3854: (4)          def test_truncate(self):
3855: (8)              tgt = [[1], [6]]
3856: (8)              arr = np.arange(10).reshape(2, 5)
3857: (8)              out = np.compress([0, 1], arr, axis=1)
3858: (8)              assert_equal(out, tgt)
3859: (4)          def test_flatten(self):
3860: (8)              arr = np.arange(10).reshape(2, 5)
3861: (8)              out = np.compress([0, 1], arr)
3862: (8)              assert_equal(out, 1)
3863: (0)      class TestPutmask:
3864: (4)          def tst_basic(self, x, T, mask, val):
3865: (8)              np.putmask(x, mask, val)
3866: (8)              assert_equal(x[mask], np.array(val, T))
3867: (4)          def test_ip_types(self):
3868: (8)              unchecked_types = [bytes, str, np.void]
3869: (8)              x = np.random.random(1000)*100
3870: (8)              mask = x < 40
3871: (8)              for val in [-100, 0, 15]:
3872: (12)                  for types in np.sctypes.values():
3873: (16)                      for T in types:
3874: (20)                          if T not in unchecked_types:
3875: (24)                              if val < 0 and np.dtype(T).kind == "u":
3876: (28)                                  val = np.iinfo(T).max - 99
3877: (24)                                  self.tst_basic(x.copy().astype(T), T, mask, val)
3878: (12)                      dt = np.dtype("S3")
3879: (12)                      self.tst_basic(x.astype(dt), dt.type, mask, dt.type(val)[:3])
3880: (4)          def test_mask_size(self):
3881: (8)              assert_raises(ValueError, np.putmask, np.array([1, 2, 3]), [True], 5)
3882: (4) @pytest.mark.parametrize('dtype', ('>i4', '<i4'))
3883: (4)          def test_byteorder(self, dtype):
3884: (8)              x = np.array([1, 2, 3], dtype)
3885: (8)              np.putmask(x, [True, False, True], -1)
3886: (8)              assert_array_equal(x, [-1, 2, -1])
3887: (4)          def test_record_array(self):
3888: (8)              rec = np.array([(-5, 2.0, 3.0), (5.0, 4.0, 3.0)],
3889: (22)                      dtype=[('x', '<f8'), ('y', '>f8'), ('z', '<f8')])
3890: (8)              np.putmask(rec['x'], [True, False], 10)
3891: (8)              assert_array_equal(rec['x'], [10, 5])
3892: (8)              assert_array_equal(rec['y'], [2, 4])
3893: (8)              assert_array_equal(rec['z'], [3, 3])
3894: (8)              np.putmask(rec['y'], [True, False], 11)

```

```

3895: (8) assert_array_equal(rec['x'], [10, 5])
3896: (8) assert_array_equal(rec['y'], [11, 4])
3897: (8) assert_array_equal(rec['z'], [3, 3])
3898: (4) def test_overlaps(self):
3899: (8)     x = np.array([True, False, True, False])
3900: (8)     np.putmask(x[1:4], [True, True, True], x[:3])
3901: (8)     assert_equal(x, np.array([True, True, False, True]))
3902: (8)     x = np.array([True, False, True, False])
3903: (8)     np.putmask(x[1:4], x[:3], [True, False, True])
3904: (8)     assert_equal(x, np.array([True, True, True, True]))
3905: (4) def test_writeable(self):
3906: (8)     a = np.arange(5)
3907: (8)     a.flags.writeable = False
3908: (8)     with pytest.raises(ValueError):
3909: (12)         np.putmask(a, a >= 2, 3)
3910: (4) def test_kwargs(self):
3911: (8)     x = np.array([0, 0])
3912: (8)     np.putmask(x, [0, 1], [-1, -2])
3913: (8)     assert_array_equal(x, [0, -2])
3914: (8)     x = np.array([0, 0])
3915: (8)     np.putmask(x, mask=[0, 1], values=[-1, -2])
3916: (8)     assert_array_equal(x, [0, -2])
3917: (8)     x = np.array([0, 0])
3918: (8)     np.putmask(x, values=[-1, -2], mask=[0, 1])
3919: (8)     assert_array_equal(x, [0, -2])
3920: (8)     with pytest.raises(TypeError):
3921: (12)         np.putmask(a=x, values=[-1, -2], mask=[0, 1])
3922: (0) class TestTake:
3923: (4)     def tst_basic(self, x):
3924: (8)         ind = list(range(x.shape[0]))
3925: (8)         assert_array_equal(x.take(ind, axis=0), x)
3926: (4)     def test_ip_types(self):
3927: (8)         unchecked_types = [bytes, str, np.void]
3928: (8)         x = np.random.random(24)*100
3929: (8)         x.shape = 2, 3, 4
3930: (8)         for types in np.sctypes.values():
3931: (12)             for T in types:
3932: (16)                 if T not in unchecked_types:
3933: (20)                     self.tst_basic(x.copy().astype(T))
3934: (12)                     self.tst_basic(x.astype("S3"))
3935: (4)     def test_raise(self):
3936: (8)         x = np.random.random(24)*100
3937: (8)         x.shape = 2, 3, 4
3938: (8)         assert_raises(IndexError, x.take, [0, 1, 2], axis=0)
3939: (8)         assert_raises(IndexError, x.take, [-3], axis=0)
3940: (8)         assert_array_equal(x.take([-1], axis=0)[0], x[1])
3941: (4)     def test_clip(self):
3942: (8)         x = np.random.random(24)*100
3943: (8)         x.shape = 2, 3, 4
3944: (8)         assert_array_equal(x.take([-1], axis=0, mode='clip')[0], x[0])
3945: (8)         assert_array_equal(x.take([2], axis=0, mode='clip')[0], x[1])
3946: (4)     def test_wrap(self):
3947: (8)         x = np.random.random(24)*100
3948: (8)         x.shape = 2, 3, 4
3949: (8)         assert_array_equal(x.take([-1], axis=0, mode='wrap')[0], x[1])
3950: (8)         assert_array_equal(x.take([2], axis=0, mode='wrap')[0], x[0])
3951: (8)         assert_array_equal(x.take([3], axis=0, mode='wrap')[0], x[1])
3952: (4)         @pytest.mark.parametrize('dtype', ('>i4', '<i4'))
3953: (4)     def test_byteorder(self, dtype):
3954: (8)         x = np.array([1, 2, 3], dtype)
3955: (8)         assert_array_equal(x.take([0, 2, 1]), [1, 3, 2])
3956: (4)     def test_record_array(self):
3957: (8)         rec = np.array([(-5, 2.0, 3.0), (5.0, 4.0, 3.0)],
3958: (22)                         dtype=[('x', '<f8'), ('y', '>f8'), ('z', '<f8')])
3959: (8)         rec1 = rec.take([1])
3960: (8)         assert_(rec1['x'] == 5.0 and rec1['y'] == 4.0)
3961: (4)     def test_out_overlap(self):
3962: (8)         x = np.arange(5)
3963: (8)         y = np.take(x, [1, 2, 3], out=x[2:5], mode='wrap')

```

```

3964: (8)             assert_equal(y, np.array([1, 2, 3]))
3965: (4)             @pytest.mark.parametrize('shape', [(1, 2), (1,), ()])
3966: (4)             def test_ret_is_out(self, shape):
3967: (8)                 x = np.arange(5)
3968: (8)                 inds = np.zeros(shape, dtype=np.intp)
3969: (8)                 out = np.zeros(shape, dtype=x.dtype)
3970: (8)                 ret = np.take(x, inds, out=out)
3971: (8)                 assert ret is out
3972: (0)             class TestLexsort:
3973: (4)                 @pytest.mark.parametrize('dtype',[ 
3974: (8)                     np.uint8, np.uint16, np.uint32, np.uint64,
3975: (8)                     np.int8, np.int16, np.int32, np.int64,
3976: (8)                     np.float16, np.float32, np.float64
3977: (4)                 ])
3978: (4)                 def test_basic(self, dtype):
3979: (8)                     a = np.array([1, 2, 1, 3, 1, 5], dtype=dtype)
3980: (8)                     b = np.array([0, 4, 5, 6, 2, 3], dtype=dtype)
3981: (8)                     idx = np.lexsort((b, a))
3982: (8)                     expected_idx = np.array([0, 4, 2, 1, 3, 5])
3983: (8)                     assert_array_equal(idx, expected_idx)
3984: (8)                     assert_array_equal(a[idx], np.sort(a))
3985: (4)                 def test_mixed(self):
3986: (8)                     a = np.array([1, 2, 1, 3, 1, 5])
3987: (8)                     b = np.array([0, 4, 5, 6, 2, 3], dtype='datetime64[D]')
3988: (8)                     idx = np.lexsort((b, a))
3989: (8)                     expected_idx = np.array([0, 4, 2, 1, 3, 5])
3990: (8)                     assert_array_equal(idx, expected_idx)
3991: (4)                 def test_datetime(self):
3992: (8)                     a = np.array([0,0,0], dtype='datetime64[D]')
3993: (8)                     b = np.array([2,1,0], dtype='datetime64[D]')
3994: (8)                     idx = np.lexsort((b, a))
3995: (8)                     expected_idx = np.array([2, 1, 0])
3996: (8)                     assert_array_equal(idx, expected_idx)
3997: (8)                     a = np.array([0,0,0], dtype='timedelta64[D]')
3998: (8)                     b = np.array([2,1,0], dtype='timedelta64[D]')
3999: (8)                     idx = np.lexsort((b, a))
4000: (8)                     expected_idx = np.array([2, 1, 0])
4001: (8)                     assert_array_equal(idx, expected_idx)
4002: (4)                 def test_object(self): # gh-6312
4003: (8)                     a = np.random.choice(10, 1000)
4004: (8)                     b = np.random.choice(['abc', 'xy', 'wz', 'efghi', 'qwst', 'x'], 1000)
4005: (8)                     for u in a, b:
4006: (12)                         left = np.lexsort((u.astype('O'),))
4007: (12)                         right = np.argsort(u, kind='mergesort')
4008: (12)                         assert_array_equal(left, right)
4009: (8)                     for u, v in (a, b), (b, a):
4010: (12)                         idx = np.lexsort((u, v))
4011: (12)                         assert_array_equal(idx, np.lexsort((u.astype('O'), v)))
4012: (12)                         assert_array_equal(idx, np.lexsort((u, v.astype('O'))))
4013: (12)                         u, v = np.array(u, dtype='object'), np.array(v, dtype='object')
4014: (12)                         assert_array_equal(idx, np.lexsort((u, v)))
4015: (4)                 def test_invalid_axis(self): # gh-7528
4016: (8)                     x = np.linspace(0., 1., 42*3).reshape(42, 3)
4017: (8)                     assert_raises(np.AxisError, np.lexsort, x, axis=2)
4018: (0)             class TestIO:
4019: (4)                 """Test tofile, fromfile, tobytes, and fromstring"""
4020: (4)                 @pytest.fixture()
4021: (4)                 def x(self):
4022: (8)                     shape = (2, 4, 3)
4023: (8)                     rand = np.random.random
4024: (8)                     x = rand(shape) + rand(shape).astype(complex) * 1j
4025: (8)                     x[0, :, 1] = [np.nan, np.inf, -np.inf, np.nan]
4026: (8)                     return x
4027: (4)                 @pytest.fixture(params=["string", "path_obj"])
4028: (4)                 def tmp_filename(self, tmp_path, request):
4029: (8)                     filename = tmp_path / "file"
4030: (8)                     if request.param == "string":
4031: (12)                         filename = str(filename)
4032: (8)                     yield filename

```

```

4033: (4) def test_nofile(self):
4034: (8)     b = io.BytesIO()
4035: (8)     assert_raises(OSError, np.fromfile, b, np.uint8, 80)
4036: (8)     d = np.ones(7)
4037: (8)     assert_raises(OSError, lambda x: x.tofile(b), d)
4038: (4) def test_bool_fromstring(self):
4039: (8)     v = np.array([True, False, True, False], dtype=np.bool_)
4040: (8)     y = np.fromstring('1 0 -2.3 0.0', sep=' ', dtype=np.bool_)
4041: (8)     assert_array_equal(v, y)
4042: (4) def test_uint64_fromstring(self):
4043: (8)     d = np.fromstring("9923372036854775807 104783749223640",
4044: (26)             dtype=np.uint64, sep=' ')
4045: (8)     e = np.array([9923372036854775807, 104783749223640], dtype=np.uint64)
4046: (8)     assert_array_equal(d, e)
4047: (4) def test_int64_fromstring(self):
4048: (8)     d = np.fromstring("-25041670086757 104783749223640",
4049: (26)             dtype=np.int64, sep=' ')
4050: (8)     e = np.array([-25041670086757, 104783749223640], dtype=np.int64)
4051: (8)     assert_array_equal(d, e)
4052: (4) def test_fromstring_count0(self):
4053: (8)     d = np.fromstring("1,2", sep=",", dtype=np.int64, count=0)
4054: (8)     assert d.shape == (0,)
4055: (4) def test_empty_files_text(self, tmp_filename):
4056: (8)     with open(tmp_filename, 'w') as f:
4057: (12)         pass
4058: (8)     y = np.fromfile(tmp_filename)
4059: (8)     assert_(y.size == 0, "Array not empty")
4060: (4) def test_empty_files_binary(self, tmp_filename):
4061: (8)     with open(tmp_filename, 'wb') as f:
4062: (12)         pass
4063: (8)     y = np.fromfile(tmp_filename, sep=" ")
4064: (8)     assert_(y.size == 0, "Array not empty")
4065: (4) def test_roundtrip_file(self, x, tmp_filename):
4066: (8)     with open(tmp_filename, 'wb') as f:
4067: (12)         x.tofile(f)
4068: (8)     with open(tmp_filename, 'rb') as f:
4069: (12)         y = np.fromfile(f, dtype=x.dtype)
4070: (8)     assert_array_equal(y, x.flat)
4071: (4) def test_roundtrip(self, x, tmp_filename):
4072: (8)     x.tofile(tmp_filename)
4073: (8)     y = np.fromfile(tmp_filename, dtype=x.dtype)
4074: (8)     assert_array_equal(y, x.flat)
4075: (4) def test_roundtrip_dump_pathlib(self, x, tmp_filename):
4076: (8)     p = pathlib.Path(tmp_filename)
4077: (8)     x.dump(p)
4078: (8)     y = np.load(p, allow_pickle=True)
4079: (8)     assert_array_equal(y, x)
4080: (4) def test_roundtrip_binary_str(self, x):
4081: (8)     s = x.tobytes()
4082: (8)     y = np.frombuffer(s, dtype=x.dtype)
4083: (8)     assert_array_equal(y, x.flat)
4084: (8)     s = x.tobytes('F')
4085: (8)     y = np.frombuffer(s, dtype=x.dtype)
4086: (8)     assert_array_equal(y, x.flatten('F'))
4087: (4) def test_roundtrip_str(self, x):
4088: (8)     x = x.real.ravel()
4089: (8)     s = "@".join(map(str, x))
4090: (8)     y = np.fromstring(s, sep="@")
4091: (8)     nan_mask = ~np.isfinite(x)
4092: (8)     assert_array_equal(x[nan_mask], y[nan_mask])
4093: (8)     assert_array_almost_equal(x[~nan_mask], y[~nan_mask], decimal=5)
4094: (4) def test_roundtrip_repr(self, x):
4095: (8)     x = x.real.ravel()
4096: (8)     s = "@".join(map(repr, x))
4097: (8)     y = np.fromstring(s, sep="@")
4098: (8)     assert_array_equal(x, y)
4099: (4) def test_unseekable_fromfile(self, x, tmp_filename):
4100: (8)     x.tofile(tmp_filename)
4101: (8)     def fail(*args, **kwargs):

```

```

4102: (12)             raise OSError('Can not tell or seek')
4103: (8)              with io.open(tmp_filename, 'rb', buffering=0) as f:
4104: (12)                f.seek = fail
4105: (12)                f.tell = fail
4106: (12)                assert_raises(OSError, np.fromfile, f, dtype=x.dtype)
4107: (4)                 def test_io_open_unbuffered_fromfile(self, x, tmp_filename):
4108: (8)                   x.tofile(tmp_filename)
4109: (8)                   with io.open(tmp_filename, 'rb', buffering=0) as f:
4110: (12)                     y = np.fromfile(f, dtype=x.dtype)
4111: (12)                     assert_array_equal(y, x.flat)
4112: (4)                 def test_largish_file(self, tmp_filename):
4113: (8)                   d = np.zeros(4 * 1024 ** 2)
4114: (8)                   d.tofile(tmp_filename)
4115: (8)                   assert_equal(os.path.getsize(tmp_filename), d.nbytes)
4116: (8)                   assert_array_equal(d, np.fromfile(tmp_filename))
4117: (8)                   with open(tmp_filename, "r+b") as f:
4118: (12)                     f.seek(d.nbytes)
4119: (12)                     d.tofile(f)
4120: (12)                     assert_equal(os.path.getsize(tmp_filename), d.nbytes * 2)
4121: (8)                   open(tmp_filename, "w").close() # delete file contents
4122: (8)                   with open(tmp_filename, "ab") as f:
4123: (12)                     d.tofile(f)
4124: (8)                     assert_array_equal(d, np.fromfile(tmp_filename))
4125: (8)                   with open(tmp_filename, "ab") as f:
4126: (12)                     d.tofile(f)
4127: (8)                     assert_equal(os.path.getsize(tmp_filename), d.nbytes * 2)
4128: (4)                 def test_io_open_buffered_fromfile(self, x, tmp_filename):
4129: (8)                   x.tofile(tmp_filename)
4130: (8)                   with io.open(tmp_filename, 'rb', buffering=-1) as f:
4131: (12)                     y = np.fromfile(f, dtype=x.dtype)
4132: (8)                     assert_array_equal(y, x.flat)
4133: (4)                 def test_file_position_after_fromfile(self, tmp_filename):
4134: (8)                   sizes = [io.DEFAULT_BUFFER_SIZE//8,
4135: (17)                     io.DEFAULT_BUFFER_SIZE,
4136: (17)                     io.DEFAULT_BUFFER_SIZE*8]
4137: (8)                   for size in sizes:
4138: (12)                     with open(tmp_filename, 'wb') as f:
4139: (16)                       f.seek(size-1)
4140: (16)                       f.write(b'\0')
4141: (12)                     for mode in ['rb', 'r+b']:
4142: (16)                       err_msg = "%d %s" % (size, mode)
4143: (16)                       with open(tmp_filename, mode) as f:
4144: (20)                         f.read(2)
4145: (20)                         np.fromfile(f, dtype=np.float64, count=1)
4146: (20)                         pos = f.tell()
4147: (16)                         assert_equal(pos, 10, err_msg=err_msg)
4148: (4)                 def test_file_position_after_tofile(self, tmp_filename):
4149: (8)                   sizes = [io.DEFAULT_BUFFER_SIZE//8,
4150: (17)                     io.DEFAULT_BUFFER_SIZE,
4151: (17)                     io.DEFAULT_BUFFER_SIZE*8]
4152: (8)                   for size in sizes:
4153: (12)                     err_msg = "%d" % (size,)
4154: (12)                     with open(tmp_filename, 'wb') as f:
4155: (16)                       f.seek(size-1)
4156: (16)                       f.write(b'\0')
4157: (16)                       f.seek(10)
4158: (16)                       f.write(b'12')
4159: (16)                       np.array([0], dtype=np.float64).tofile(f)
4160: (16)                       pos = f.tell()
4161: (12)                         assert_equal(pos, 10 + 2 + 8, err_msg=err_msg)
4162: (12)                         with open(tmp_filename, 'r+b') as f:
4163: (16)                           f.read(2)
4164: (16)                           f.seek(0, 1) # seek between read&write required by ANSI C
4165: (16)                           np.array([0], dtype=np.float64).tofile(f)
4166: (16)                           pos = f.tell()
4167: (12)                         assert_equal(pos, 10, err_msg=err_msg)
4168: (4)                 def test_load_object_array_fromfile(self, tmp_filename):
4169: (8)                   with open(tmp_filename, 'w') as f:
4170: (12)                     pass

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

4171: (8)
4172: (12)
4173: (32)
4174: (8)
4175: (28)
4176: (4)
4177: (8)
4178: (12)
4179: (8)
4180: (12)
4181: (12)
4182: (8)
4183: (12)
4184: (12)
4185: (12)
4186: (12)
4187: (16)
4188: (12)
4189: (12)
4190: (16)
4191: (12)
4192: (12)
4193: (12)
4194: (12)
4195: (8)
4196: (12)
4197: (8)
4198: (12)
4199: (20)
4200: (20)
4201: (20)
4202: (20)
4203: (4)
4204: (4)
4205: (8)
4206: (12)
4207: (8)
4208: (12)
4209: (8)
4210: (8)
4211: (12)
4212: (16)
4213: (16)
4214: (20)
4215: (20)
4216: (8)
4217: (12)
4218: (4)
4219: (8)
4220: (12)
4221: (8)
4222: (12)
4223: (8)
4224: (8)
4225: (12)
4226: (8)
4227: (8)
4228: (4)
4229: (4)
4230: (8)
4231: (8)
4232: (8)
4233: (8)
4234: (12)
4235: (16)
4236: (8)
4237: (12)
4238: (16)
4239: (12)

        with open(tmp_filename, 'rb') as f:
            assert_raises_regex(ValueError, "Cannot read into object array",
                                np.fromfile, f, dtype=object)
            assert_raises_regex(ValueError, "Cannot read into object array",
                                np.fromfile, tmp_filename, dtype=object)

    def test_fromfile_offset(self, x, tmp_filename):
        with open(tmp_filename, 'wb') as f:
            x.tofile(f)
        with open(tmp_filename, 'rb') as f:
            y = np.fromfile(f, dtype=x.dtype, offset=0)
            assert_array_equal(y, x.flat)
        with open(tmp_filename, 'rb') as f:
            count_items = len(x.flat) // 8
            offset_items = len(x.flat) // 4
            offset_bytes = x.dtype.itemsize * offset_items
            y = np.fromfile(
                f, dtype=x.dtype, count=count_items, offset=offset_bytes
            )
            assert_array_equal(
                y, x.flat[offset_items:offset_items+count_items]
            )
            offset_bytes = x.dtype.itemsize
            z = np.fromfile(f, dtype=x.dtype, offset=offset_bytes)
            assert_array_equal(z, x.flat[offset_items+count_items+1:])
        with open(tmp_filename, 'wb') as f:
            x.tofile(f, sep=",")
        with open(tmp_filename, 'rb') as f:
            assert_raises_regex(
                TypeError,
                "'offset' argument only permitted for binary files",
                np.fromfile, tmp_filename, dtype=x.dtype,
                sep=",", offset=1)

@pytest.mark.skipif(IS_PYPY, reason="bug in PyPy's PyNumber_AsSsize_t")
def test_fromfile_bad_dup(self, x, tmp_filename):
    def dup_str(fd):
        return 'abc'
    def dup_bigint(fd):
        return 2**68
    old_dup = os.dup
    try:
        with open(tmp_filename, 'wb') as f:
            x.tofile(f)
            for dup, exc in ((dup_str, TypeError), (dup_bigint, OSError)):
                os.dup = dup
                assert_raises(exc, np.fromfile, f)
    finally:
        os.dup = old_dup

def _check_from(self, s, value, filename, **kw):
    if 'sep' not in kw:
        y = np.frombuffer(s, **kw)
    else:
        y = np.fromstring(s, **kw)
    assert_array_equal(y, value)
    with open(filename, 'wb') as f:
        f.write(s)
    y = np.fromfile(filename, **kw)
    assert_array_equal(y, value)

@pytest.fixture(params=["period", "comma"])
def decimal_sep_localization(self, request):
    """
    Including this fixture in a test will automatically
    execute it with both types of decimal separator.
    So::
        def test_decimal(decimal_sep_localization):
            pass
    is equivalent to the following two tests::
        def test_decimal_period_separator():
            pass
        def test_decimal_comma_separator():
    """


```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

4240: (16)
4241: (20)
4242: (8)
4243: (8)
4244: (12)
4245: (8)
4246: (12)
4247: (16)
4248: (8)
4249: (12)
4250: (4)
4251: (8)
4252: (12)
4253: (12)
4254: (12)
4255: (12)
4256: (4)
4257: (8)
4258: (12)
4259: (12)
4260: (12)
4261: (12)
4262: (4)
4263: (8)
4264: (12)
4265: (12)
4266: (12)
4267: (12)
4268: (4)
4269: (8)
4270: (12)
4271: (12)
4272: (12)
4273: (12)
4274: (4)
4275: (8)
4276: (4)
4277: (8)
4278: (12)
4279: (8)
4280: (12)
4281: (8)
4282: (12)
4283: (4)
4284: (8)
4285: (12)
4286: (4)
4287: (8)
4288: (12)
4289: (12)
4290: (4)
4291: (8)
4292: (12)
4293: (8)
4294: (12)
4295: (4)
4296: (8)
4297: (12)
4298: (16)
4299: (4)
4300: (8)
4301: (12)
4302: (4)
4303: (8)
4304: (8)
4305: (4)
4306: (8)
4307: (8)

        with CommaDecimalPointLocale():
            pass
        """
        if request.param == "period":
            yield
        elif request.param == "comma":
            with CommaDecimalPointLocale():
                yield
        else:
            assert False, request.param
    def test_nan(self, tmp_filename, decimal_sep_localization):
        self._check_from(
            b"nan +nan -nan NaN nan(foo) +NaN(BAR) -NAN(q_u_u_x_)",
            [np.nan, np.nan, np.nan, np.nan, np.nan, np.nan, np.nan],
            tmp_filename,
            sep=' ')
    def test_inf(self, tmp_filename, decimal_sep_localization):
        self._check_from(
            b"inf +inf -inf infinity -Infinity iNFInItY -inF",
            [np.inf, np.inf, -np.inf, np.inf, -np.inf, np.inf, -np.inf],
            tmp_filename,
            sep=' ')
    def test_numbers(self, tmp_filename, decimal_sep_localization):
        self._check_from(
            b"1.234 -1.234 .3 .3e55 -123133.1231e+133",
            [1.234, -1.234, .3, .3e55, -123133.1231e+133],
            tmp_filename,
            sep=' ')
    def test_binary(self, tmp_filename):
        self._check_from(
            b'\x00\x00\x80?\x00\x00\x00@\x00\x00@@\x00\x00\x80@',
            np.array([1, 2, 3, 4]),
            tmp_filename,
            dtype='<f4')
    def test_string(self, tmp_filename):
        self._check_from(b'1,2,3,4', [1., 2., 3., 4.], tmp_filename, sep=',')
    def test_counted_string(self, tmp_filename, decimal_sep_localization):
        self._check_from(
            b'1,2,3,4', [1., 2., 3., 4.], tmp_filename, count=4, sep=',')
        self._check_from(
            b'1,2,3,4', [1., 2., 3.], tmp_filename, count=3, sep=',')
        self._check_from(
            b'1,2,3,4', [1., 2., 3., 4.], tmp_filename, count=-1, sep=',')
    def test_string_with_ws(self, tmp_filename):
        self._check_from(
            b'1 2 3     4  ', [1, 2, 3, 4], tmp_filename, dtype=int, sep=' ')
    def test_counted_string_with_ws(self, tmp_filename):
        self._check_from(
            b'1 2 3     4  ', [1, 2, 3], tmp_filename, count=3, dtype=int,
            sep=' ')
    def test_ascii(self, tmp_filename, decimal_sep_localization):
        self._check_from(
            b'1 , 2 , 3 , 4', [1., 2., 3., 4.], tmp_filename, sep=',')
        self._check_from(
            b'1,2,3,4', [1., 2., 3., 4.], tmp_filename, dtype=float, sep=',')
    def test_malformed(self, tmp_filename, decimal_sep_localization):
        with assert_warns(DeprecationWarning):
            self._check_from(
                b'1.234 1,234', [1.234, 1.], tmp_filename, sep=' ')
    def test_long_sep(self, tmp_filename):
        self._check_from(
            b'1_x_3_x_4_x_5', [1, 3, 4, 5], tmp_filename, sep='_x_')
    def test_dtype(self, tmp_filename):
        v = np.array([1, 2, 3, 4], dtype=np.int_)
        self._check_from(b'1,2,3,4', v, tmp_filename, sep=',', dtype=np.int_)
    def test_dtype_bool(self, tmp_filename):
        v = np.array([True, False, True, False], dtype=np.bool_)
        s = b'1,0,-2.3,0'

```

```

4308: (8)           with open(tmp_filename, 'wb') as f:
4309: (12)             f.write(s)
4310: (8)             y = np.fromfile(tmp_filename, sep=',', dtype=np.bool_)
4311: (8)             assert_(y.dtype == '?')
4312: (8)             assert_array_equal(y, v)
4313: (4)             def test_tofile_sep(self, tmp_filename, decimal_sep_localization):
4314: (8)               x = np.array([1.51, 2, 3.51, 4], dtype=float)
4315: (8)               with open(tmp_filename, 'w') as f:
4316: (12)                 x.tofile(f, sep=',')
4317: (8)               with open(tmp_filename, 'r') as f:
4318: (12)                 s = f.read()
4319: (8)                 y = np.array([float(p) for p in s.split(',')]) 
4320: (8)                 assert_array_equal(x,y)
4321: (4)             def test_tofile_format(self, tmp_filename, decimal_sep_localization):
4322: (8)               x = np.array([1.51, 2, 3.51, 4], dtype=float)
4323: (8)               with open(tmp_filename, 'w') as f:
4324: (12)                 x.tofile(f, sep=',', format='%.2f')
4325: (8)               with open(tmp_filename, 'r') as f:
4326: (12)                 s = f.read()
4327: (8)                 assert_equal(s, '1.51,2.00,3.51,4.00')
4328: (4)             def test_tofile_cleanup(self, tmp_filename):
4329: (8)               x = np.zeros((10), dtype=object)
4330: (8)               with open(tmp_filename, 'wb') as f:
4331: (12)                 assert_raises(OSError, lambda: x.tofile(f, sep=''))
4332: (8)               os.remove(tmp_filename)
4333: (8)               assert_raises(OSError, lambda: x.tofile(tmp_filename))
4334: (8)               os.remove(tmp_filename)
4335: (4)             def test_fromfile_subarray_binary(self, tmp_filename):
4336: (8)               x = np.arange(24, dtype="i4").reshape(2, 3, 4)
4337: (8)               x.tofile(tmp_filename)
4338: (8)               res = np.fromfile(tmp_filename, dtype="(3,4)i4")
4339: (8)               assert_array_equal(x, res)
4340: (8)               x_str = x.tobytes()
4341: (8)               with assert_warns(DeprecationWarning):
4342: (12)                 res = np.fromstring(x_str, dtype="(3,4)i4")
4343: (12)                 assert_array_equal(x, res)
4344: (4)             def test_parsing_subarray_unsupported(self, tmp_filename):
4345: (8)               data = "12,42,13," * 50
4346: (8)               with pytest.raises(ValueError):
4347: (12)                 expected = np.fromstring(data, dtype="(3,)i", sep=",")
4348: (8)               with open(tmp_filename, "w") as f:
4349: (12)                 f.write(data)
4350: (8)               with pytest.raises(ValueError):
4351: (12)                 np.fromfile(tmp_filename, dtype="(3,)i", sep=",")
4352: (4)             def test_read_shorter_than_count_subarray(self, tmp_filename):
4353: (8)               expected = np.arange(511 * 10, dtype="i").reshape(-1, 10)
4354: (8)               binary = expected.tobytes()
4355: (8)               with pytest.raises(ValueError):
4356: (12)                 with pytest.warns(DeprecationWarning):
4357: (16)                   np.fromstring(binary, dtype="(10,)i", count=10000)
4358: (8)               expected.tofile(tmp_filename)
4359: (8)               res = np.fromfile(tmp_filename, dtype="(10,)i", count=10000)
4360: (8)               assert_array_equal(res, expected)
4361: (0)             class TestFromBuffer:
4362: (4)               @pytest.mark.parametrize('byteorder', ['<', '>'])
4363: (4)               @pytest.mark.parametrize('dtype', [float, int, complex])
4364: (4)               def test_basic(self, byteorder, dtype):
4365: (8)                 dt = np.dtype(dtype).newbyteorder(byteorder)
4366: (8)                 x = (np.random.random((4, 7)) * 5).astype(dt)
4367: (8)                 buf = x.tobytes()
4368: (8)                 assert_array_equal(np.frombuffer(buf, dtype=dt), x.flat)
4369: (4)               @pytest.mark.parametrize("obj", [np.arange(10), b"12345678"])
4370: (4)               def test_array_base(self, obj):
4371: (8)                 new = np.frombuffer(obj)
4372: (8)                 assert new.base is obj
4373: (4)               def test_empty(self):
4374: (8)                 assert_array_equal(np.frombuffer(b''), np.array([]))
4375: (4)               @pytest.mark.skipif(IS_PYPY,
4376: (12)                 reason="PyPy's memoryview currently does not track exports. See: "

```

```

4377: (19)                                     "https://foss.heptapod.net/pypy/pypy/-/issues/3724")
4378: (4)          def test_mmap_close(self):
4379: (8)          with tempfile.TemporaryFile(mode='wb') as tmp:
4380: (12)              tmp.write(b"asdf")
4381: (12)              tmp.flush()
4382: (12)              mm = mmap.mmap(tmp.fileno(), 0)
4383: (12)              arr = np.frombuffer(mm, dtype=np.uint8)
4384: (12)              with pytest.raises(BufferError):
4385: (16)                  mm.close() # cannot close while array uses the buffer
4386: (12)              del arr
4387: (12)              mm.close()
4388: (0)          class TestFlat:
4389: (4)              def setup_method(self):
4390: (8)                  a0 = np.arange(20.0)
4391: (8)                  a = a0.reshape(4, 5)
4392: (8)                  a0.shape = (4, 5)
4393: (8)                  a.flags.writeable = False
4394: (8)                  self.a = a
4395: (8)                  self.b = a[::2, ::2]
4396: (8)                  self.a0 = a0
4397: (8)                  self.b0 = a0[::2, ::2]
4398: (4)          def test_contiguous(self):
4399: (8)              testpassed = False
4400: (8)              try:
4401: (12)                  self.a.flat[12] = 100.0
4402: (8)              except ValueError:
4403: (12)                  testpassed = True
4404: (8)              assert_(testpassed)
4405: (8)              assert_(self.a.flat[12] == 12.0)
4406: (4)          def test_discontiguous(self):
4407: (8)              testpassed = False
4408: (8)              try:
4409: (12)                  self.b.flat[4] = 100.0
4410: (8)              except ValueError:
4411: (12)                  testpassed = True
4412: (8)              assert_(testpassed)
4413: (8)              assert_(self.b.flat[4] == 12.0)
4414: (4)          def test_array_(self):
4415: (8)              c = self.a.flat.__array__()
4416: (8)              d = self.b.flat.__array__()
4417: (8)              e = self.a0.flat.__array__()
4418: (8)              f = self.b0.flat.__array__()
4419: (8)              assert_(c.flags.writeable is False)
4420: (8)              assert_(d.flags.writeable is False)
4421: (8)              assert_(e.flags.writeable is True)
4422: (8)              assert_(f.flags.writeable is False)
4423: (8)              assert_(c.flags.writebackifcopy is False)
4424: (8)              assert_(d.flags.writebackifcopy is False)
4425: (8)              assert_(e.flags.writebackifcopy is False)
4426: (8)              assert_(f.flags.writebackifcopy is False)
4427: (4)          @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
4428: (4)          def test_refcount(self):
4429: (8)              inds = [np.intp(0), np.array([True]*self.a.size), np.array([0]), None]
4430: (8)              indtype = np.dtype(np.intp)
4431: (8)              rc_indtype = sys.getrefcount(indtype)
4432: (8)              for ind in inds:
4433: (12)                  rc_ind = sys.getrefcount(ind)
4434: (12)                  for _ in range(100):
4435: (16)                      try:
4436: (20)                          self.a.flat[ind]
4437: (16)                      except IndexError:
4438: (20)                          pass
4439: (12)                      assert_(abs(sys.getrefcount(ind) - rc_ind) < 50)
4440: (12)                      assert_(abs(sys.getrefcount(indtype) - rc_indtype) < 50)
4441: (4)          def test_index_getset(self):
4442: (8)              it = np.arange(10).reshape(2, 1, 5).flat
4443: (8)              with pytest.raises(AttributeError):
4444: (12)                  it.index = 10
4445: (8)                  for _ in it:

```

```

4446: (12)           pass
4447: (8)            assert it.index == it.base.size
4448: (0)             class TestResize:
4449: (4)             @_no_tracing
4450: (4)             def test_basic(self):
4451: (8)               x = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
4452: (8)               if IS_PYPY:
4453: (12)                 x.resize((5, 5), refcheck=False)
4454: (8)               else:
4455: (12)                 x.resize((5, 5))
4456: (8)               assert_array_equal(x.flat[:9],
4457: (16)                 np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]]).flat)
4458: (8)               assert_array_equal(x[9:].flat, 0)
4459: (4)             def test_check_reference(self):
4460: (8)               x = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
4461: (8)               y = x
4462: (8)               assert_raises(ValueError, x.resize, (5, 1))
4463: (8)               del y # avoid pyflakes unused variable warning.
4464: (4)             @_no_tracing
4465: (4)             def test_int_shape(self):
4466: (8)               x = np.eye(3)
4467: (8)               if IS_PYPY:
4468: (12)                 x.resize(3, refcheck=False)
4469: (8)               else:
4470: (12)                 x.resize(3)
4471: (8)               assert_array_equal(x, np.eye(3)[0,:])
4472: (4)             def test_none_shape(self):
4473: (8)               x = np.eye(3)
4474: (8)               x.resize(None)
4475: (8)               assert_array_equal(x, np.eye(3))
4476: (8)               x.resize()
4477: (8)               assert_array_equal(x, np.eye(3))
4478: (4)             def test_0d_shape(self):
4479: (8)               for i in range(10):
4480: (12)                 x = np.empty((1,))
4481: (12)                 x.resize(())
4482: (12)                 assert_equal(x.shape, ())
4483: (12)                 assert_equal(x.size, 1)
4484: (12)                 x = np.empty(())
4485: (12)                 x.resize((1,))
4486: (12)                 assert_equal(x.shape, (1,))
4487: (12)                 assert_equal(x.size, 1)
4488: (4)             def test_invalid_arguments(self):
4489: (8)               assert_raises(TypeError, np.eye(3).resize, 'hi')
4490: (8)               assert_raises(ValueError, np.eye(3).resize, -1)
4491: (8)               assert_raises(TypeError, np.eye(3).resize, order=1)
4492: (8)               assert_raises(TypeError, np.eye(3).resize, refcheck='hi')
4493: (4)             @_no_tracing
4494: (4)             def test_freeform_shape(self):
4495: (8)               x = np.eye(3)
4496: (8)               if IS_PYPY:
4497: (12)                 x.resize(3, 2, 1, refcheck=False)
4498: (8)               else:
4499: (12)                 x.resize(3, 2, 1)
4500: (8)               assert_(x.shape == (3, 2, 1))
4501: (4)             @_no_tracing
4502: (4)             def test_zeros_appended(self):
4503: (8)               x = np.eye(3)
4504: (8)               if IS_PYPY:
4505: (12)                 x.resize(2, 3, 3, refcheck=False)
4506: (8)               else:
4507: (12)                 x.resize(2, 3, 3)
4508: (8)               assert_array_equal(x[0], np.eye(3))
4509: (8)               assert_array_equal(x[1], np.zeros((3, 3)))
4510: (4)             @_no_tracing
4511: (4)             def test_obj_obj(self):
4512: (8)               a = np.ones(10, dtype=[('k', object, 2)])
4513: (8)               if IS_PYPY:
4514: (12)                 a.resize(15, refcheck=False)

```

```

4515: (8)
4516: (12)
4517: (8)
4518: (8)
4519: (8)
4520: (4)
4521: (8)
4522: (8)
4523: (8)
4524: (8)
4525: (4)
4526: (8)
4527: (8)
4528: (8)
4529: (8)
4530: (0)
4531: (4)
4532: (8)
4533: (8)
4534: (8)
4535: (4)
4536: (8)
4537: (12)
4538: (8)
4539: (4)
4540: (8)
4541: (8)
4542: (8)
4543: (8)
4544: (8)
4545: (8)
4546: (8)
4547: (8)
4548: (4)
4549: (8)
4550: (12)
4551: (8)
4552: (4)
4553: (8)
4554: (12)
4555: (8)
4556: (8)
4557: (4)
4558: (8)
4559: (8)
4560: (8)
4561: (8)
4562: (4)
4563: (8)
4564: (34)
4565: (34)
4566: (8)
4567: (8)
4568: (8)
4569: (8)
4570: (8)
4571: (8)
4572: (8)
4573: (8)
4574: (8)
4575: (8)
4576: (8)
4577: (8)
4578: (8)
4579: (8)
4580: (8)
4581: (8)
4582: (8)
4583: (8)

        else:
            a.resize(15, )
        assert_equal(a.shape, (15,))
        assert_array_equal(a['k'][-5:], 0)
        assert_array_equal(a['k'][:-5], 1)
    def test_empty_view(self):
        x = np.zeros((10, 0), int)
        x_view = x[...]
        x_view.resize((0, 10))
        x_view.resize((0, 100))
    def test_check_weakref(self):
        x = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
        xref = weakref.ref(x)
        assert_raises(ValueError, x.resize, (5, 1))
        del xref # avoid pyflakes unused variable warning.
class TestRecord:
    def test_field_rename(self):
        dt = np.dtype([('f', float), ('i', int)])
        dt.names = ['p', 'q']
        assert_equal(dt.names, ['p', 'q'])
    def test_multiple_field_name_occurrence(self):
        def test_dtype_init():
            np.dtype([("A", "f8"), ("B", "f8"), ("A", "f8")])
            assert_raises(ValueError, test_dtype_init)
    def test_bytes_fields(self):
        assert_raises(TypeError, np.dtype, [(b'a', int)])
        assert_raises(TypeError, np.dtype, [((b'b', b'a'), int)])
        dt = np.dtype([(b'a', 'b'), int])
        assert_raises(TypeError, dt.__getitem__, b'a')
        x = np.array([(1,), (2,), (3,)], dtype=dt)
        assert_raises(IndexError, x.__getitem__, b'a')
        y = x[0]
        assert_raises(IndexError, y.__getitem__, b'a')
    def test_multiple_field_name_unicode(self):
        def test_dtype_unicode():
            np.dtype([\("\u20B9", "f8"), ("B", "f8"), ("\u20B9", "f8")])
            assert_raises(ValueError, test_dtype_unicode)
    def test_fromarrays_unicode(self):
        x = np.core.records.fromarrays(
            [[0], [1]], names='a,b', formats='i4,i4')
        assert_equal(x['a'][0], 0)
        assert_equal(x['b'][0], 1)
    def test_unicode_order(self):
        name = 'b'
        x = np.array([1, 3, 2], dtype=[(name, int)])
        x.sort(order=name)
        assert_equal(x['b'], np.array([1, 2, 3]))
    def test_field_names(self):
        a = np.zeros((1,), dtype=[('f1', 'i4'),
                                ('f2', 'i4'),
                                ('f3', [('sf1', 'i4')])])
        assert_raises(IndexError, a.__setitem__, b'f1', 1)
        assert_raises(IndexError, a.__getitem__, b'f1')
        assert_raises(IndexError, a['f1'].__setitem__, b'sf1', 1)
        assert_raises(IndexError, a['f1'].__getitem__, b'sf1')
        b = a.copy()
        fn1 = str('f1')
        b[fn1] = 1
        assert_equal(b[fn1], 1)
        fn1 = str('not at all')
        assert_raises(ValueError, b.__setitem__, fn1, 1)
        assert_raises(ValueError, b.__getitem__, fn1)
        b[0][fn1] = 2
        assert_equal(b[fn1], 2)
        assert_raises(ValueError, b[0].__setitem__, fn1, 1)
        assert_raises(ValueError, b[0].__getitem__, fn1)
        fn3 = str('f3')
        sf1 = str('sf1')
        b[fn3][sf1] = 1

```

```

4584: (8) assert_equal(b[fn3][sfn1], 1)
4585: (8) assert_raises(ValueError, b[fn3].__setitem__, fnn, 1)
4586: (8) assert_raises(ValueError, b[fn3].__getitem__, fnn)
4587: (8) fn2 = str('f2')
4588: (8) b[fn2] = 3
4589: (8) assert_equal(b[['f1', 'f2']][0].tolist(), (2, 3))
4590: (8) assert_equal(b[['f2', 'f1']][0].tolist(), (3, 2))
4591: (8) assert_equal(b[['f1', 'f3']][0].tolist(), (2, (1,)))
4592: (8) assert_raises(ValueError, a.__setitem__, '\u03e0', 1)
4593: (8) assert_raises(ValueError, a.__getitem__, '\u03e0')
4594: (4) def test_record_hash(self):
4595: (8)     a = np.array([(1, 2), (1, 2)], dtype='i1,i2')
4596: (8)     a.flags.writeable = False
4597: (8)     b = np.array([(1, 2), (3, 4)], dtype=[('num1', 'i1'), ('num2', 'i2')])
4598: (8)     b.flags.writeable = False
4599: (8)     c = np.array([(1, 2), (3, 4)], dtype='i1,i2')
4600: (8)     c.flags.writeable = False
4601: (8)     assert_(hash(a[0]) == hash(a[1]))
4602: (8)     assert_(hash(a[0]) == hash(b[0]))
4603: (8)     assert_(hash(a[0]) != hash(b[1]))
4604: (8)     assert_(hash(c[0]) == hash(a[0]) and c[0] == a[0])
4605: (4) def test_record_no_hash(self):
4606: (8)     a = np.array([(1, 2), (1, 2)], dtype='i1,i2')
4607: (8)     assert_raises(TypeError, hash, a[0])
4608: (4) def test_empty_structure_creation(self):
4609: (8)     np.array([()], dtype={'names': [], 'formats': [],
4610: (27)             'offsets': [], 'itemsize': 12})
4611: (8)     np.array([(), (), (), (), ()], dtype={'names': [], 'formats': [],
4612: (43)             'offsets': [], 'itemsize': 12})
4613: (4) def test_multifield_indexing_view(self):
4614: (8)     a = np.ones(3, dtype=[('a', 'i4'), ('b', 'f4'), ('c', 'u4')])
4615: (8)     v = a[['a', 'c']]
4616: (8)     assert_(v.base is a)
4617: (8)     assert_(v.dtype == np.dtype({'names': ['a', 'c'],
4618: (37)             'formats': ['i4', 'u4'],
4619: (37)             'offsets': [0, 8]}))
4620: (8)     v[:] = (4,5)
4621: (8)     assert_equal(a[0].item(), (4, 1, 5))
4622: (0) class TestView:
4623: (4)     def test_basic(self):
4624: (8)         x = np.array([(1, 2, 3, 4), (5, 6, 7, 8)],
4625: (21)             dtype=[('r', np.int8), ('g', np.int8),
4626: (28)                 ('b', np.int8), ('a', np.int8)])
4627: (8)         y = x.view(dtype='<i4')
4628: (8)         z = x.view('<i4')
4629: (8)         assert_array_equal(y, z)
4630: (8)         assert_array_equal(y, [67305985, 134678021])
4631: (0)     def _mean(a, **args):
4632: (4)         return a.mean(**args)
4633: (0)     def _var(a, **args):
4634: (4)         return a.var(**args)
4635: (0)     def _std(a, **args):
4636: (4)         return a.std(**args)
4637: (0) class TestStats:
4638: (4)     funcs = [_mean, _var, _std]
4639: (4)     def setup_method(self):
4640: (8)         np.random.seed(range(3))
4641: (8)         self.rmat = np.random.random((4, 5))
4642: (8)         self.cmat = self.rmat + 1j * self.rmat
4643: (8)         self.omat = np.array([Decimal(repr(r)) for r in self.rmat.flat])
4644: (8)         self.omat = self.omat.reshape(4, 5)
4645: (4)     def test_python_type(self):
4646: (8)         for x in (np.float16(1.), 1, 1., 1+0j):
4647: (12)             assert_equal(np.mean([x]), 1.)
4648: (12)             assert_equal(np.std([x]), 0.)
4649: (12)             assert_equal(np.var([x]), 0.)
4650: (4)     def test_keepdims(self):
4651: (8)         mat = np.eye(3)
4652: (8)         for f in self.funcs:

```

```

4653: (12)
4654: (16)
4655: (16)
4656: (16)
4657: (12)
4658: (16)
4659: (16)
4660: (4)
4661: (8)
4662: (8)
4663: (12)
4664: (12)
4665: (12)
4666: (12)
4667: (12)
4668: (8)
4669: (8)
4670: (8)
4671: (8)
4672: (4)
4673: (8)
4674: (8)
4675: (8)
4676: (12)
4677: (12)
4678: (12)
4679: (12)
4680: (12)
4681: (12)
4682: (8)
4683: (12)
4684: (16)
4685: (16)
4686: (16)
4687: (16)
4688: (16)
4689: (16)
4690: (8)
4691: (12)
4692: (16)
4693: (16)
4694: (16)
4695: (16)
4696: (16)
4697: (16)
4698: (8)
4699: (12)
4700: (16)
4701: (16)
4702: (16)
4703: (16)
4704: (16)
4705: (16)
4706: (4)
4707: (8)
4708: (8)
4709: (12)
4710: (16)
4711: (16)
4712: (16)
4713: (16)
4714: (16)
4715: (4)
4716: (8)
4717: (12)
4718: (16)
4719: (16)
4720: (16)
4721: (8)

        for axis in [0, 1]:
            res = f(mat, axis=axis, keepdims=True)
            assert_(res.ndim == mat.ndim)
            assert_(res.shape[axis] == 1)
        for axis in [None]:
            res = f(mat, axis=axis, keepdims=True)
            assert_(res.shape == (1, 1))

    def test_out(self):
        mat = np.eye(3)
        for f in self.funcs:
            out = np.zeros(3)
            tgt = f(mat, axis=1)
            res = f(mat, axis=1, out=out)
            assert_almost_equal(res, out)
            assert_almost_equal(res, tgt)
            out = np.empty(2)
            assert_raises(ValueError, f, mat, axis=1, out=out)
            out = np.empty((2, 2))
            assert_raises(ValueError, f, mat, axis=1, out=out)

    def test_dtype_from_input(self):
        icode = np.typecodes['AllInteger']
        fcodes = np.typecodes['AllFloat']
        for f in self.funcs:
            mat = np.array([[Decimal(1)]*3]*3)
            tgt = mat.dtype.type
            res = f(mat, axis=1).dtype.type
            assert_(res is tgt)
            res = type(f(mat, axis=None))
            assert_(res is Decimal)
        for f in self.funcs:
            for c in icode:
                mat = np.eye(3, dtype=c)
                tgt = np.float64
                res = f(mat, axis=1).dtype.type
                assert_(res is tgt)
                res = f(mat, axis=None).dtype.type
                assert_(res is tgt)
        for f in [_mean]:
            for c in fcodes:
                mat = np.eye(3, dtype=c)
                tgt = mat.dtype.type
                res = f(mat, axis=1).dtype.type
                assert_(res is tgt)
                res = f(mat, axis=None).dtype.type
                assert_(res is tgt)
        for f in [_var, _std]:
            for c in fcodes:
                mat = np.eye(3, dtype=c)
                tgt = mat.real.dtype.type
                res = f(mat, axis=1).dtype.type
                assert_(res is tgt)
                res = f(mat, axis=None).dtype.type
                assert_(res is tgt)

    def test_dtype_from_dtype(self):
        mat = np.eye(3)
        for f in self.funcs:
            for c in np.typecodes['AllFloat']:
                tgt = np.dtype(c).type
                res = f(mat, axis=1, dtype=c).dtype.type
                assert_(res is tgt)
                res = f(mat, axis=None, dtype=c).dtype.type
                assert_(res is tgt)

    def test_ddof(self):
        for f in [_var]:
            for ddof in range(3):
                dim = self.rmat.shape[1]
                tgt = f(self.rmat, axis=1) * dim
                res = f(self.rmat, axis=1, ddof=ddof) * (dim - ddof)
        for f in [_std]:

```

```

4722: (12)
4723: (16)
4724: (16)
4725: (16)
4726: (16)
4727: (16)
4728: (4)
4729: (8)
4730: (8)
4731: (12)
4732: (16)
4733: (20)
4734: (20)
4735: (20)
4736: (20)
4737: (20)
4738: (4)
4739: (8)
4740: (8)
4741: (12)
4742: (16)
4743: (20)
4744: (20)
4745: (20)
4746: (20)
4747: (12)
4748: (16)
4749: (20)
4750: (20)
4751: (4)
4752: (8)
4753: (12)
4754: (16)
4755: (16)
4756: (16)
4757: (12)
4758: (16)
4759: (16)
4760: (16)
4761: (4)
4762: (8)
4763: (4)
4764: (8)
4765: (12)
4766: (4)
4767: (8)
4768: (8)
4769: (28)
4770: (28)
4771: (28)
4772: (8)
4773: (31)
4774: (31)
4775: (31)
4776: (8)
4777: (18)
4778: (18)
4779: (18)
4780: (8)
4781: (12)
4782: (28)
4783: (12)
4784: (28)
4785: (8)
4786: (8)
4787: (8)
4788: (8)
4789: (24)
4790: (8)

        for ddof in range(3):
            dim = self.rmat.shape[1]
            tgt = f(self.rmat, axis=1) * np.sqrt(dim)
            res = f(self.rmat, axis=1, ddof=ddof) * np.sqrt(dim - ddof)
            assert_almost_equal(res, tgt)
            assert_almost_equal(res, tgt)

    def test_ddof_too_big(self):
        dim = self.rmat.shape[1]
        for f in [_var, _std]:
            for ddof in range(dim, dim + 2):
                with warnings.catch_warnings(record=True) as w:
                    warnings.simplefilter('always')
                    res = f(self.rmat, axis=1, ddof=ddof)
                    assert_(not (res < 0).any())
                    assert_(len(w) > 0)
                    assert_(issubclass(w[0].category, RuntimeWarning))

    def test_empty(self):
        A = np.zeros((0, 3))
        for f in self.funcs:
            for axis in [0, None]:
                with warnings.catch_warnings(record=True) as w:
                    warnings.simplefilter('always')
                    assert_(np.isnan(f(A, axis=axis)).all())
                    assert_(len(w) > 0)
                    assert_(issubclass(w[0].category, RuntimeWarning))
            for axis in [1]:
                with warnings.catch_warnings(record=True) as w:
                    warnings.simplefilter('always')
                    assert_equal(f(A, axis=axis), np.zeros([]))

    def test_mean_values(self):
        for mat in [self.rmat, self.cmat, self.omat]:
            for axis in [0, 1]:
                tgt = mat.sum(axis=axis)
                res = _mean(mat, axis=axis) * mat.shape[axis]
                assert_almost_equal(res, tgt)
            for axis in [None]:
                tgt = mat.sum(axis=axis)
                res = _mean(mat, axis=axis) * np.prod(mat.shape)
                assert_almost_equal(res, tgt)

    def test_mean_float16(self):
        assert(_mean(np.ones(100000, dtype='float16')) == 1)

    def test_mean_axis_error(self):
        with assert_raises(np.exceptions.AxisError):
            np.arange(10).mean(axis=2)

    def test_mean_where(self):
        a = np.arange(16).reshape((4, 4))
        wh_full = np.array([[False, True, False, True],
                           [True, False, True, False],
                           [True, True, False, False],
                           [False, False, True, True]])
        wh_partial = np.array([[False],
                              [True],
                              [True],
                              [False]])
        _cases = [(1, True, [1.5, 5.5, 9.5, 13.5]),
                  (0, wh_full, [6., 5., 10., 9.]),
                  (1, wh_full, [2., 5., 8.5, 14.5]),
                  (0, wh_partial, [6., 7., 8., 9.])]
        for _ax, _wh, _res in _cases:
            assert_allclose(a.mean(axis=_ax, where=_wh),
                            np.array(_res))
            assert_allclose(np.mean(a, axis=_ax, where=_wh),
                            np.array(_res))

        a3d = np.arange(16).reshape((2, 2, 4))
        _wh_partial = np.array([False, True, True, False])
        _res = [[1.5, 5.5], [9.5, 13.5]]
        assert_allclose(a3d.mean(axis=2, where=_wh_partial),
                        np.array(_res))
        assert_allclose(np.mean(a3d, axis=2, where=_wh_partial),

```

```

4791: (24)                         np.array(_res))
4792: (8)                          with pytest.warns(RuntimeWarning) as w:
4793: (12)                         assert_allclose(a.mean(axis=1, where=wh_partial),
4794: (28)                           np.array([np.nan, 5.5, 9.5, np.nan]))
4795: (8)                          with pytest.warns(RuntimeWarning) as w:
4796: (12)                         assert_equal(a.mean(where=False), np.nan)
4797: (8)                          with pytest.warns(RuntimeWarning) as w:
4798: (12)                         assert_equal(np.mean(a, where=False), np.nan)
4799: (4)  def test_var_values(self):
4800: (8)      for mat in [self.rmat, self.cmat, self.omat]:
4801: (12)        for axis in [0, 1, None]:
4802: (16)          msqr = _mean(mat * mat.conj(), axis=axis)
4803: (16)          mean = _mean(mat, axis=axis)
4804: (16)          tgt = msqr - mean * mean.conjugate()
4805: (16)          res = _var(mat, axis=axis)
4806: (16)          assert_almost_equal(res, tgt)
4807: (4)  @pytest.mark.parametrize(('complex_dtype', 'ndec'), (
4808: (8)    ('complex64', 6),
4809: (8)    ('complex128', 7),
4810: (8)    ('clongdouble', 7),
4811: (4)  ))
4812: (4)  def test_var_complex_values(self, complex_dtype, ndec):
4813: (8)    for axis in [0, 1, None]:
4814: (12)      mat = self.cmat.copy().astype(complex_dtype)
4815: (12)      msqr = _mean(mat * mat.conj(), axis=axis)
4816: (12)      mean = _mean(mat, axis=axis)
4817: (12)      tgt = msqr - mean * mean.conjugate()
4818: (12)      res = _var(mat, axis=axis)
4819: (12)      assert_almost_equal(res, tgt, decimal=ndec)
4820: (4)  def test_var_dimensions(self):
4821: (8)    mat = np.stack([self.cmat]*3)
4822: (8)    for axis in [0, 1, 2, -1, None]:
4823: (12)      msqr = _mean(mat * mat.conj(), axis=axis)
4824: (12)      mean = _mean(mat, axis=axis)
4825: (12)      tgt = msqr - mean * mean.conjugate()
4826: (12)      res = _var(mat, axis=axis)
4827: (12)      assert_almost_equal(res, tgt)
4828: (4)  def test_var_complex_byteorder(self):
4829: (8)    cmat = self.cmat.copy().astype('complex128')
4830: (8)    cmat_swapped = cmat.astype(cmat.dtype.newbyteorder())
4831: (8)    assert_almost_equal(cmat.var(), cmat_swapped.var())
4832: (4)  def test_var_axis_error(self):
4833: (8)    with assert_raises(np.exceptions.AxisError):
4834: (12)      np.arange(10).var(axis=2)
4835: (4)  def test_var_where(self):
4836: (8)    a = np.arange(25).reshape((5, 5))
4837: (8)    wh_full = np.array([[False, True, False, True, True],
4838: (28)                  [True, False, True, True, False],
4839: (28)                  [True, True, False, False, True],
4840: (28)                  [False, True, True, False, True],
4841: (28)                  [True, False, True, True, False]])
4842: (8)    wh_partial = np.array([[False],
4843: (31)                  [True],
4844: (31)                  [True],
4845: (31)                  [False],
4846: (31)                  [True]])
4847: (8)    _cases = [(0, True, [50., 50., 50., 50., 50.]),
4848: (18)                  (1, True, [2., 2., 2., 2., 2.])]
4849: (8)    for _ax, _wh, _res in _cases:
4850: (12)      assert_allclose(a.var(axis=_ax, where=_wh),
4851: (28)                      np.array(_res))
4852: (12)      assert_allclose(np.var(a, axis=_ax, where=_wh),
4853: (28)                      np.array(_res))
4854: (8)    a3d = np.arange(16).reshape((2, 2, 4))
4855: (8)    _wh_partial = np.array([False, True, True, False])
4856: (8)    _res = [[0.25, 0.25], [0.25, 0.25]]
4857: (8)    assert_allclose(a3d.var(axis=2, where=_wh_partial),
4858: (24)                      np.array(_res))
4859: (8)    assert_allclose(np.var(a3d, axis=2, where=_wh_partial),

```

```

4860: (24)                               np.array(_res))
4861: (8)                                assert_allclose(np.var(a, axis=1, where=wh_full),
4862: (24)   np.var(a[wh_full].reshape((5, 3)), axis=1))
4863: (8)                                assert_allclose(np.var(a, axis=0, where=wh_partial),
4864: (24)   np.var(a[wh_partial[:, 0]], axis=0))
4865: (8)                                with pytest.warns(RuntimeWarning) as w:
4866: (12)                                     assert_equal(a.var(where=False), np.nan)
4867: (8)                                with pytest.warns(RuntimeWarning) as w:
4868: (12)                                     assert_equal(np.var(a, where=False), np.nan)
4869: (4)                                 def test_std_values(self):
4870: (8)                                     for mat in [self.rmat, self.cmat, self.omat]:
4871: (12)   for axis in [0, 1, None]:
4872: (16)   tgt = np.sqrt(_var(mat, axis=axis))
4873: (16)   res = _std(mat, axis=axis)
4874: (16)   assert_almost_equal(res, tgt)
4875: (4)                                 def test_std_where(self):
4876: (8)                                     a = np.arange(25).reshape((5, 5))[:-1]
4877: (8)                                     whf = np.array([[False, True, False, True, True],
4878: (24)   [True, False, True, False, True],
4879: (24)   [True, True, False, True, False],
4880: (24)   [True, False, True, True, False],
4881: (24)   [False, True, False, True, True]])
4882: (8)                                     whp = np.array([[False],
4883: (24)   [False],
4884: (24)   [True],
4885: (24)   [True],
4886: (24)   [False]])
4887: (8)                                     _cases = [
4888: (12)   (0, True, 7.07106781*np.ones((5))),
4889: (12)   (1, True, 1.41421356*np.ones((5))),
4890: (12)   (0, whf,
4891: (13)   np.array([4.0824829, 8.16496581, 5., 7.39509973, 8.49836586])),
4892: (12)   (0, whp, 2.5*np.ones((5))))
4893: (8)                                     ]
4894: (8)                                     for _ax, _wh, _res in _cases:
4895: (12)   assert_allclose(a.std(axis=_ax, where=_wh), _res)
4896: (12)   assert_allclose(np.std(a, axis=_ax, where=_wh), _res)
4897: (8)                                     a3d = np.arange(16).reshape((2, 2, 4))
4898: (8)                                     _wh_partial = np.array([False, True, True, False])
4899: (8)                                     _res = [[0.5, 0.5], [0.5, 0.5]]
4900: (8)                                     assert_allclose(a3d.std(axis=2, where=_wh_partial),
4901: (24)   np.array(_res))
4902: (8)                                     assert_allclose(np.std(a3d, axis=2, where=_wh_partial),
4903: (24)   np.array(_res))
4904: (8)                                     assert_allclose(a.std(axis=1, where=whf),
4905: (24)   np.std(a[whf].reshape((5, 3)), axis=1))
4906: (8)                                     assert_allclose(np.std(a, axis=1, where=whf),
4907: (24)   (a[whf].reshape((5, 3))).std(axis=1))
4908: (8)                                     assert_allclose(a.std(axis=0, where=whp),
4909: (24)   np.std(a[whp[:, 0]], axis=0))
4910: (8)                                     assert_allclose(np.std(a, axis=0, where=whp),
4911: (24)   (a[whp[:, 0]]).std(axis=0))
4912: (8)                                     with pytest.warns(RuntimeWarning) as w:
4913: (12)   assert_equal(a.std(where=False), np.nan)
4914: (8)                                     with pytest.warns(RuntimeWarning) as w:
4915: (12)   assert_equal(np.std(a, where=False), np.nan)
4916: (4)                                 def test_subclass(self):
4917: (8)                                     class TestArray(np.ndarray):
4918: (12)   def __new__(cls, data, info):
4919: (16)   result = np.array(data)
4920: (16)   result = result.view(cls)
4921: (16)   result.info = info
4922: (16)   return result
4923: (12)   def __array_finalize__(self, obj):
4924: (16)   self.info = getattr(obj, "info", '')
4925: (8)   dat = TestArray([[1, 2, 3, 4], [5, 6, 7, 8]], 'jubba')
4926: (8)   res = dat.mean(1)
4927: (8)   assert_(res.info == dat.info)
4928: (8)   res = dat.std(1)

```

```

4929: (8)             assert_(res.info == dat.info)
4930: (8)             res = dat.var(1)
4931: (8)             assert_(res.info == dat.info)
4932: (0)          class TestVdot:
4933: (4)            def test_basic(self):
4934: (8)              dt_numeric = np.typecodes['AllFloat'] + np.typecodes['AllInteger']
4935: (8)              dt_complex = np.typecodes['Complex']
4936: (8)              a = np.eye(3)
4937: (8)              for dt in dt_numeric + '0':
4938: (12)                b = a.astype(dt)
4939: (12)                res = np.vdot(b, b)
4940: (12)                assert_(np.isscalar(res))
4941: (12)                assert_equal(np.vdot(b, b), 3)
4942: (8)              a = np.eye(3) * 1j
4943: (8)              for dt in dt_complex + '0':
4944: (12)                b = a.astype(dt)
4945: (12)                res = np.vdot(b, b)
4946: (12)                assert_(np.isscalar(res))
4947: (12)                assert_equal(np.vdot(b, b), 3)
4948: (8)              b = np.eye(3, dtype=bool)
4949: (8)              res = np.vdot(b, b)
4950: (8)              assert_(np.isscalar(res))
4951: (8)              assert_equal(np.vdot(b, b), True)
4952: (4)          def test_vdot_array_order(self):
4953: (8)            a = np.array([[1, 2], [3, 4]], order='C')
4954: (8)            b = np.array([[1, 2], [3, 4]], order='F')
4955: (8)            res = np.vdot(a, a)
4956: (8)            assert_equal(np.vdot(a, b), res)
4957: (8)            assert_equal(np.vdot(b, a), res)
4958: (8)            assert_equal(np.vdot(b, b), res)
4959: (4)          def test_vdot_uncontiguous(self):
4960: (8)            for size in [2, 1000]:
4961: (12)              a = np.zeros((size, 2, 2))
4962: (12)              b = np.zeros((size, 2, 2))
4963: (12)              a[:, 0, 0] = np.arange(size)
4964: (12)              b[:, 0, 0] = np.arange(size) + 1
4965: (12)              a = a[..., 0]
4966: (12)              b = b[..., 0]
4967: (12)              assert_equal(np.vdot(a, b),
4968: (25)                            np.vdot(a.flatten(), b.flatten()))
4969: (12)              assert_equal(np.vdot(a, b.copy()),
4970: (25)                            np.vdot(a.flatten(), b.flatten()))
4971: (12)              assert_equal(np.vdot(a.copy(), b),
4972: (25)                            np.vdot(a.flatten(), b.flatten()))
4973: (12)              assert_equal(np.vdot(a.copy('F'), b),
4974: (25)                            np.vdot(a.flatten(), b.flatten()))
4975: (12)              assert_equal(np.vdot(a, b.copy('F')),
4976: (25)                            np.vdot(a.flatten(), b.flatten()))
4977: (0)        class TestDot:
4978: (4)          def setup_method(self):
4979: (8)            np.random.seed(128)
4980: (8)            self.A = np.random.rand(4, 2)
4981: (8)            self.b1 = np.random.rand(2, 1)
4982: (8)            self.b2 = np.random.rand(2)
4983: (8)            self.b3 = np.random.rand(1, 2)
4984: (8)            self.b4 = np.random.rand(4)
4985: (8)            self.N = 7
4986: (4)          def test_dotmatmat(self):
4987: (8)            A = self.A
4988: (8)            res = np.dot(A.transpose(), A)
4989: (8)            tgt = np.array([[1.45046013, 0.86323640],
4990: (24)                          [0.86323640, 0.84934569]])
4991: (8)            assert_almost_equal(res, tgt, decimal=self.N)
4992: (4)          def test_dotmatvec(self):
4993: (8)            A, b1 = self.A, self.b1
4994: (8)            res = np.dot(A, b1)
4995: (8)            tgt = np.array([[0.32114320], [0.04889721],
4996: (24)                          [0.15696029], [0.33612621]])
4997: (8)            assert_almost_equal(res, tgt, decimal=self.N)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

4998: (4)
4999: (8)
5000: (8)
5001: (8)
5002: (8)
5003: (4)
5004: (8)
5005: (8)
5006: (8)
5007: (8)
5008: (4)
5009: (8)
5010: (8)
5011: (8)
5012: (8)
5013: (4)
5014: (8)
5015: (8)
5016: (8)
5017: (8)
5018: (4)
5019: (8)
5020: (8)
5021: (8)
5022: (8)
5023: (4)
5024: (8)
5025: (8)
5026: (8)
5027: (8)
5028: (4)
5029: (8)
5030: (8)
5031: (8)
5032: (8)
5033: (8)
5034: (4)
5035: (8)
5036: (8)
5037: (8)
5038: (8)
5039: (8)
5040: (4)
5041: (8)
5042: (8)
5043: (8)
5044: (8)
5045: (8)
5046: (8)
5047: (4)
5048: (8)
5049: (8)
5050: (8)
5051: (8)
5052: (8)
5053: (8)
5054: (4)
5055: (8)
5056: (8)
5057: (8)
5058: (12)
5059: (12)
5060: (12)
5061: (12)
5062: (12)
5063: (12)
5064: (4)
5065: (8)
5066: (12)

    def test_dotmatvec2(self):
        A, b2 = self.A, self.b2
        res = np.dot(A, b2)
        tgt = np.array([0.29677940, 0.04518649, 0.14468333, 0.31039293])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotvecmat(self):
        A, b4 = self.A, self.b4
        res = np.dot(b4, A)
        tgt = np.array([1.23495091, 1.12222648])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotvecmat2(self):
        b3, A = self.b3, self.A
        res = np.dot(b3, A.transpose())
        tgt = np.array([[0.58793804, 0.08957460, 0.30605758, 0.62716383]])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotvecmat3(self):
        A, b4 = self.A, self.b4
        res = np.dot(A.transpose(), b4)
        tgt = np.array([1.23495091, 1.12222648])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotvecvecouter(self):
        b1, b3 = self.b1, self.b3
        res = np.dot(b1, b3)
        tgt = np.array([[0.20128610, 0.08400440], [0.07190947, 0.03001058]])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotvecvecinner(self):
        b1, b3 = self.b1, self.b3
        res = np.dot(b3, b1)
        tgt = np.array([[ 0.23129668]])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotcolumnvect1(self):
        b1 = np.ones((3, 1))
        b2 = [5.3]
        res = np.dot(b1, b2)
        tgt = np.array([5.3, 5.3, 5.3])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotcolumnvect2(self):
        b1 = np.ones((3, 1)).transpose()
        b2 = [6.2]
        res = np.dot(b2, b1)
        tgt = np.array([6.2, 6.2, 6.2])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotveccscalar(self):
        np.random.seed(100)
        b1 = np.random.rand(1, 1)
        b2 = np.random.rand(1, 4)
        res = np.dot(b1, b2)
        tgt = np.array([[0.15126730, 0.23068496, 0.45905553, 0.00256425]])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_dotveccscalar2(self):
        np.random.seed(100)
        b1 = np.random.rand(4, 1)
        b2 = np.random.rand(1, 1)
        res = np.dot(b1, b2)
        tgt = np.array([[0.00256425], [0.00131359], [0.00200324], [ 0.00398638]])
        assert_almost_equal(res, tgt, decimal=self.N)

    def test_all(self):
        dims = [(), (1,), (1, 1)]
        dout = [(), (1,), (1, 1), (1,), (), (1,), (1, 1), (1,), (1, 1)]
        for dim, (dim1, dim2) in zip(dout, itertools.product(dims, dims)):
            b1 = np.zeros(dim1)
            b2 = np.zeros(dim2)
            res = np.dot(b1, b2)
            tgt = np.zeros(dim)
            assert_(res.shape == tgt.shape)
            assert_almost_equal(res, tgt, decimal=self.N)

    def test_vecobject(self):
        class Vec:
            def __init__(self, sequence=None):

```

```

5067: (16)             if sequence is None:
5068: (20)                 sequence = []
5069: (16)                 self.array = np.array(sequence)
5070: (12)             def __add__(self, other):
5071: (16)                 out = Vec()
5072: (16)                 out.array = self.array + other.array
5073: (16)                 return out
5074: (12)             def __sub__(self, other):
5075: (16)                 out = Vec()
5076: (16)                 out.array = self.array - other.array
5077: (16)                 return out
5078: (12)             def __mul__(self, other): # with scalar
5079: (16)                 out = Vec(self.array.copy())
5080: (16)                 out.array *= other
5081: (16)                 return out
5082: (12)             def __rmul__(self, other):
5083: (16)                 return self*other
5084: (8)             U_non_cont = np.transpose([[1., 1.], [1., 2.]])
5085: (8)             U_cont = np.ascontiguousarray(U_non_cont)
5086: (8)             x = np.array([Vec([1., 0.]), Vec([0., 1.])])
5087: (8)             zeros = np.array([Vec([0., 0.]), Vec([0., 0.])])
5088: (8)             zeros_test = np.dot(U_cont, x) - np.dot(U_non_cont, x)
5089: (8)             assert_equal(zeros[0].array, zeros_test[0].array)
5090: (8)             assert_equal(zeros[1].array, zeros_test[1].array)
5091: (4)             def test_dot_2args(self):
5092: (8)                 from numpy.core.multiarray import dot
5093: (8)                 a = np.array([[1, 2], [3, 4]], dtype=float)
5094: (8)                 b = np.array([[1, 0], [1, 1]], dtype=float)
5095: (8)                 c = np.array([[3, 2], [7, 4]], dtype=float)
5096: (8)                 d = dot(a, b)
5097: (8)                 assert_allclose(c, d)
5098: (4)             def test_dot_3args(self):
5099: (8)                 from numpy.core.multiarray import dot
5100: (8)                 np.random.seed(22)
5101: (8)                 f = np.random.random_sample((1024, 16))
5102: (8)                 v = np.random.random_sample((16, 32))
5103: (8)                 r = np.empty((1024, 32))
5104: (8)                 for i in range(12):
5105: (12)                     dot(f, v, r)
5106: (8)                 if HAS_REFCOUNT:
5107: (12)                     assert_equal(sys.getrefcount(r), 2)
5108: (8)                     r2 = dot(f, v, out=None)
5109: (8)                     assert_array_equal(r2, r)
5110: (8)                     assert_(r is dot(f, v, out=r))
5111: (8)                     v = v[:, 0].copy() # v.shape == (16,)
5112: (8)                     r = r[:, 0].copy() # r.shape == (1024,)
5113: (8)                     r2 = dot(f, v)
5114: (8)                     assert_(r is dot(f, v, r))
5115: (8)                     assert_array_equal(r2, r)
5116: (4)             def test_dot_3args_errors(self):
5117: (8)                 from numpy.core.multiarray import dot
5118: (8)                 np.random.seed(22)
5119: (8)                 f = np.random.random_sample((1024, 16))
5120: (8)                 v = np.random.random_sample((16, 32))
5121: (8)                 r = np.empty((1024, 31))
5122: (8)                 assert_raises(ValueError, dot, f, v, r)
5123: (8)                 r = np.empty((1024,))
5124: (8)                 assert_raises(ValueError, dot, f, v, r)
5125: (8)                 r = np.empty((32,))
5126: (8)                 assert_raises(ValueError, dot, f, v, r)
5127: (8)                 r = np.empty((32, 1024))
5128: (8)                 assert_raises(ValueError, dot, f, v, r)
5129: (8)                 assert_raises(ValueError, dot, f, v, r.T)
5130: (8)                 r = np.empty((1024, 64))
5131: (8)                 assert_raises(ValueError, dot, f, v, r[:, ::2])
5132: (8)                 assert_raises(ValueError, dot, f, v, r[:, :32])
5133: (8)                 r = np.empty((1024, 32), dtype=np.float32)
5134: (8)                 assert_raises(ValueError, dot, f, v, r)
5135: (8)                 r = np.empty((1024, 32), dtype=int)

```

```

5136: (8)             assert_raises(ValueError, dot, f, v, r)
5137: (4)             def test_dot_out_result(self):
5138: (8)                 x = np.ones((), dtype=np.float16)
5139: (8)                 y = np.ones((5,), dtype=np.float16)
5140: (8)                 z = np.zeros((5,), dtype=np.float16)
5141: (8)                 res = x.dot(y, out=z)
5142: (8)                 assert np.array_equal(res, y)
5143: (8)                 assert np.array_equal(z, y)
5144: (4)             def test_dot_out_aliasing(self):
5145: (8)                 x = np.ones((), dtype=np.float16)
5146: (8)                 y = np.ones((5,), dtype=np.float16)
5147: (8)                 z = np.zeros((5,), dtype=np.float16)
5148: (8)                 res = x.dot(y, out=z)
5149: (8)                 z[0] = 2
5150: (8)                 assert np.array_equal(res, z)
5151: (4)             def test_dot_array_order(self):
5152: (8)                 a = np.array([[1, 2], [3, 4]], order='C')
5153: (8)                 b = np.array([[1, 2], [3, 4]], order='F')
5154: (8)                 res = np.dot(a, a)
5155: (8)                 assert_equal(np.dot(a, b), res)
5156: (8)                 assert_equal(np.dot(b, a), res)
5157: (8)                 assert_equal(np.dot(b, b), res)
5158: (4)             def test_accelerate_framework_sgемv_fix(self):
5159: (8)                 def aligned_array(shape, align, dtype, order='C'):
5160: (12)                     d = dtype(0)
5161: (12)                     N = np.prod(shape)
5162: (12)                     tmp = np.zeros(N * d.nbytes + align, dtype=np.uint8)
5163: (12)                     address = tmp.__array_interface__["data"][0]
5164: (12)                     for offset in range(alignment):
5165: (16)                         if (address + offset) % align == 0:
5166: (20)                             break
5167: (12)                     tmp = tmp[offset:offset+N*d.nbytes].view(dtype=dtype)
5168: (12)                     return tmp.reshape(shape, order=order)
5169: (8)             def as_aligned(arr, align, dtype, order='C'):
5170: (12)                 aligned = aligned_array(arr.shape, align, dtype, order)
5171: (12)                 aligned[:] = arr[:]
5172: (12)                 return aligned
5173: (8)             def assert_dot_close(A, X, desired):
5174: (12)                 assert_allclose(np.dot(A, X), desired, rtol=1e-5, atol=1e-7)
5175: (8)                 m = aligned_array(100, 15, np.float32)
5176: (8)                 s = aligned_array((100, 100), 15, np.float32)
5177: (8)                 np.dot(s, m) # this will always segfault if the bug is present
5178: (8)                 testdata = itertools.product((15, 32), (10000,), (200, 89), ('C',
5179: (8) 'F'))
5180: (12)                 for align, m, n, a_order in testdata:
5181: (12)                     A_d = np.random.rand(m, n)
5182: (12)                     X_d = np.random.rand(n)
5183: (12)                     desired = np.dot(A_d, X_d)
5184: (12)                     A_f = as_aligned(A_d, align, np.float32, order=a_order)
5185: (12)                     X_f = as_aligned(X_d, align, np.float32)
5186: (12)                     assert_dot_close(A_f, X_f, desired)
5187: (12)                     A_d_2 = A_d[:, ::2]
5188: (12)                     desired = np.dot(A_d_2, X_d)
5189: (12)                     A_f_2 = A_f[:, ::2]
5190: (12)                     assert_dot_close(A_f_2, X_f, desired)
5191: (12)                     A_d_22 = A_d_2[:, :, ::2]
5192: (12)                     X_d_2 = X_d[:, ::2]
5193: (12)                     desired = np.dot(A_d_22, X_d_2)
5194: (12)                     A_f_22 = A_f_2[:, :, ::2]
5195: (12)                     X_f_2 = X_f[:, ::2]
5196: (12)                     assert_dot_close(A_f_22, X_f_2, desired)
5197: (16)                     if a_order == 'F':
5198: (12)                         assert_equal(A_f_22.strides, (8, 8 * m))
5199: (16)                     else:
5200: (12)                         assert_equal(A_f_22.strides, (8 * n, 8))
5201: (12)                     assert_equal(X_f_2.strides, (8,))
5202: (12)                     X_f_2c = as_aligned(X_f_2, align, np.float32)
5203: (12)                     assert_dot_close(A_f_22, X_f_2c, desired)

```

```

5204: (12)             desired = np.dot(A_d_12, X_d_2)
5205: (12)             A_f_12 = A_f[:, ::2]
5206: (12)             assert_dot_close(A_f_12, X_f_2c, desired)
5207: (12)             assert_dot_close(A_f_12, X_f_2, desired)
5208: (4) @pytest.mark.slow
5209: (4) @pytest.mark.parametrize("dtype", [np.float64, np.complex128])
5210: (4) @requires_memory(free_bytes=18e9) # complex case needs 18GiB+
5211: (4) def test_huge_vectordot(self, dtype):
5212: (8)     data = np.ones(2**30+100, dtype=dtype)
5213: (8)     res = np.dot(data, data)
5214: (8)     assert res == 2**30+100
5215: (4) def test_dtype_discovery_fails(self):
5216: (8)     class BadObject(object):
5217: (12)         def __array__(self):
5218: (16)             raise TypeError("just this tiny mint leaf")
5219: (8)         with pytest.raises(TypeError):
5220: (12)             np.dot(BadObject(), BadObject())
5221: (8)         with pytest.raises(TypeError):
5222: (12)             np.dot(3.0, BadObject())
5223: (0) class MatmulCommon:
5224: (4)     """Common tests for '@' operator and numpy.matmul.
5225: (4) """
5226: (4)     types = "?bhilqBHILQefdgFDGO"
5227: (4) def test_exceptions(self):
5228: (8)     dims = [
5229: (12)         ((1,), (2,)),           # mismatched vector vector
5230: (12)         ((2, 1,), (2,)),      # mismatched matrix vector
5231: (12)         ((2,), (1, 2)),      # mismatched vector matrix
5232: (12)         ((1, 2), (3, 1)),      # mismatched matrix matrix
5233: (12)         ((1,), ()),          # vector scalar
5234: (12)         (((), (1)),),        # scalar vector
5235: (12)         ((1, 1), ()),        # matrix scalar
5236: (12)         (((), (1, 1)),),    # scalar matrix
5237: (12)         ((2, 2, 1), (3, 1, 2)), # cannot broadcast
5238: (12)     ]
5239: (8)     for dt, (dm1, dm2) in itertools.product(self.types, dims):
5240: (12)         a = np.ones(dm1, dtype=dt)
5241: (12)         b = np.ones(dm2, dtype=dt)
5242: (12)         assert_raises(ValueError, self.matmul, a, b)
5243: (4) def test_shapes(self):
5244: (8)     dims = [
5245: (12)         ((1, 1), (2, 1, 1)),    # broadcast first argument
5246: (12)         ((2, 1, 1), (1, 1)),    # broadcast second argument
5247: (12)         ((2, 1, 1), (2, 1, 1)),  # matrix stack sizes match
5248: (12)     ]
5249: (8)     for dt, (dm1, dm2) in itertools.product(self.types, dims):
5250: (12)         a = np.ones(dm1, dtype=dt)
5251: (12)         b = np.ones(dm2, dtype=dt)
5252: (12)         res = self.matmul(a, b)
5253: (12)         assert_(res.shape == (2, 1, 1))
5254: (8)     for dt in self.types:
5255: (12)         a = np.ones((2,), dtype=dt)
5256: (12)         b = np.ones((2,), dtype=dt)
5257: (12)         c = self.matmul(a, b)
5258: (12)         assert_(np.array(c).shape == ())
5259: (4) def test_result_types(self):
5260: (8)     mat = np.ones((1,1))
5261: (8)     vec = np.ones((1,))
5262: (8)     for dt in self.types:
5263: (12)         m = mat.astype(dt)
5264: (12)         v = vec.astype(dt)
5265: (12)         for arg in [(m, v), (v, m), (m, m)]:
5266: (16)             res = self.matmul(*arg)
5267: (16)             assert_(res.dtype == dt)
5268: (12)             if dt != "O":
5269: (16)                 res = self.matmul(v, v)
5270: (16)                 assert_(type(res) is np.dtype(dt).type)
5271: (4) def test_scalar_output(self):
5272: (8)     vec1 = np.array([2])

```

```

5273: (8)           vec2 = np.array([3, 4]).reshape(1, -1)
5274: (8)           tgt = np.array([6, 8])
5275: (8)           for dt in self.types[1:]:
5276: (12)             v1 = vec1.astype(dt)
5277: (12)             v2 = vec2.astype(dt)
5278: (12)             res = self.matmul(v1, v2)
5279: (12)             assert_equal(res, tgt)
5280: (12)             res = self.matmul(v2.T, v1)
5281: (12)             assert_equal(res, tgt)
5282: (8)             vec = np.array([True, True], dtype='?').reshape(1, -1)
5283: (8)             res = self.matmul(vec[:, 0], vec)
5284: (8)             assert_equal(res, True)
5285: (4)             def test_vector_vector_values(self):
5286: (8)               vec1 = np.array([1, 2])
5287: (8)               vec2 = np.array([3, 4]).reshape(-1, 1)
5288: (8)               tgt1 = np.array([11])
5289: (8)               tgt2 = np.array([[3, 6], [4, 8]])
5290: (8)               for dt in self.types[1:]:
5291: (12)                 v1 = vec1.astype(dt)
5292: (12)                 v2 = vec2.astype(dt)
5293: (12)                 res = self.matmul(v1, v2)
5294: (12)                 assert_equal(res, tgt1)
5295: (12)                 res = self.matmul(v2, v1.reshape(1, -1))
5296: (12)                 assert_equal(res, tgt2)
5297: (8)                 vec = np.array([True, True], dtype='?')
5298: (8)                 res = self.matmul(vec, vec)
5299: (8)                 assert_equal(res, True)
5300: (4)             def test_vector_matrix_values(self):
5301: (8)               vec = np.array([1, 2])
5302: (8)               mat1 = np.array([[1, 2], [3, 4]])
5303: (8)               mat2 = np.stack([mat1]*2, axis=0)
5304: (8)               tgt1 = np.array([7, 10])
5305: (8)               tgt2 = np.stack([tgt1]*2, axis=0)
5306: (8)               for dt in self.types[1:]:
5307: (12)                 v = vec.astype(dt)
5308: (12)                 m1 = mat1.astype(dt)
5309: (12)                 m2 = mat2.astype(dt)
5310: (12)                 res = self.matmul(v, m1)
5311: (12)                 assert_equal(res, tgt1)
5312: (12)                 res = self.matmul(v, m2)
5313: (12)                 assert_equal(res, tgt2)
5314: (8)               vec = np.array([True, False])
5315: (8)               mat1 = np.array([[True, False], [False, True]])
5316: (8)               mat2 = np.stack([mat1]*2, axis=0)
5317: (8)               tgt1 = np.array([True, False])
5318: (8)               tgt2 = np.stack([tgt1]*2, axis=0)
5319: (8)               res = self.matmul(vec, mat1)
5320: (8)               assert_equal(res, tgt1)
5321: (8)               res = self.matmul(vec, mat2)
5322: (8)               assert_equal(res, tgt2)
5323: (4)             def test_matrix_vector_values(self):
5324: (8)               vec = np.array([1, 2])
5325: (8)               mat1 = np.array([[1, 2], [3, 4]])
5326: (8)               mat2 = np.stack([mat1]*2, axis=0)
5327: (8)               tgt1 = np.array([5, 11])
5328: (8)               tgt2 = np.stack([tgt1]*2, axis=0)
5329: (8)               for dt in self.types[1:]:
5330: (12)                 v = vec.astype(dt)
5331: (12)                 m1 = mat1.astype(dt)
5332: (12)                 m2 = mat2.astype(dt)
5333: (12)                 res = self.matmul(m1, v)
5334: (12)                 assert_equal(res, tgt1)
5335: (12)                 res = self.matmul(m2, v)
5336: (12)                 assert_equal(res, tgt2)
5337: (8)               vec = np.array([True, False])
5338: (8)               mat1 = np.array([[True, False], [False, True]])
5339: (8)               mat2 = np.stack([mat1]*2, axis=0)
5340: (8)               tgt1 = np.array([True, False])
5341: (8)               tgt2 = np.stack([tgt1]*2, axis=0)

```

```

5342: (8)             res = self.matmul(vec, mat1)
5343: (8)             assert_equal(res, tgt1)
5344: (8)             res = self.matmul(vec, mat2)
5345: (8)             assert_equal(res, tgt2)
5346: (4)             def test_matrix_matrix_values(self):
5347: (8)                 mat1 = np.array([[1, 2], [3, 4]])
5348: (8)                 mat2 = np.array([[1, 0], [1, 1]])
5349: (8)                 mat12 = np.stack([mat1, mat2], axis=0)
5350: (8)                 mat21 = np.stack([mat2, mat1], axis=0)
5351: (8)                 tgt11 = np.array([[7, 10], [15, 22]])
5352: (8)                 tgt12 = np.array([[3, 2], [7, 4]])
5353: (8)                 tgt21 = np.array([[1, 2], [4, 6]])
5354: (8)                 tgt12_21 = np.stack([tgt12, tgt21], axis=0)
5355: (8)                 tgt11_12 = np.stack((tgt11, tgt12), axis=0)
5356: (8)                 tgt11_21 = np.stack((tgt11, tgt21), axis=0)
5357: (8)             for dt in self.types[1:]:
5358: (12)                 m1 = mat1.astype(dt)
5359: (12)                 m2 = mat2.astype(dt)
5360: (12)                 m12 = mat12.astype(dt)
5361: (12)                 m21 = mat21.astype(dt)
5362: (12)                 res = self.matmul(m1, m2)
5363: (12)                 assert_equal(res, tgt12)
5364: (12)                 res = self.matmul(m2, m1)
5365: (12)                 assert_equal(res, tgt21)
5366: (12)                 res = self.matmul(m12, m1)
5367: (12)                 assert_equal(res, tgt11_21)
5368: (12)                 res = self.matmul(m1, m12)
5369: (12)                 assert_equal(res, tgt11_12)
5370: (12)                 res = self.matmul(m12, m21)
5371: (12)                 assert_equal(res, tgt12_21)
5372: (8)                 m1 = np.array([[1, 1], [0, 0]], dtype=np.bool_)
5373: (8)                 m2 = np.array([[1, 0], [1, 1]], dtype=np.bool_)
5374: (8)                 m12 = np.stack([m1, m2], axis=0)
5375: (8)                 m21 = np.stack([m2, m1], axis=0)
5376: (8)                 tgt11 = m1
5377: (8)                 tgt12 = m1
5378: (8)                 tgt21 = np.array([[1, 1], [1, 1]], dtype=np.bool_)
5379: (8)                 tgt12_21 = np.stack([tgt12, tgt21], axis=0)
5380: (8)                 tgt11_12 = np.stack((tgt11, tgt12), axis=0)
5381: (8)                 tgt11_21 = np.stack((tgt11, tgt21), axis=0)
5382: (8)                 res = self.matmul(m1, m2)
5383: (8)                 assert_equal(res, tgt12)
5384: (8)                 res = self.matmul(m2, m1)
5385: (8)                 assert_equal(res, tgt21)
5386: (8)                 res = self.matmul(m12, m1)
5387: (8)                 assert_equal(res, tgt11_21)
5388: (8)                 res = self.matmul(m1, m12)
5389: (8)                 assert_equal(res, tgt11_12)
5390: (8)                 res = self.matmul(m12, m21)
5391: (8)                 assert_equal(res, tgt12_21)
5392: (0)             class TestMatmul(MatmulCommon):
5393: (4)                 matmul = np.matmul
5394: (4)                 def test_out_arg(self):
5395: (8)                     a = np.ones((5, 2), dtype=float)
5396: (8)                     b = np.array([[1, 3], [5, 7]], dtype=float)
5397: (8)                     tgt = np.dot(a, b)
5398: (8)                     msg = "out positional argument"
5399: (8)                     out = np.zeros((5, 2), dtype=float)
5400: (8)                     self.matmul(a, b, out)
5401: (8)                     assert_array_equal(out, tgt, err_msg=msg)
5402: (8)                     msg = "out keyword argument"
5403: (8)                     out = np.zeros((5, 2), dtype=float)
5404: (8)                     self.matmul(a, b, out=out)
5405: (8)                     assert_array_equal(out, tgt, err_msg=msg)
5406: (8)                     msg = "Cannot cast ufunc .* output"
5407: (8)                     out = np.zeros((5, 2), dtype=np.int32)
5408: (8)                     assert_raises_regex(TypeError, msg, self.matmul, a, b, out=out)
5409: (8)                     out = np.zeros((5, 2), dtype=np.complex128)
5410: (8)                     c = self.matmul(a, b, out=out)

```

```

5411: (8)             assert_(c is out)
5412: (8)             with suppress_warnings() as sup:
5413: (12)               sup.filter(np.ComplexWarning, '')
5414: (12)               c = c.astype(tgt.dtype)
5415: (8)               assert_array_equal(c, tgt)
5416: (4)             def test_empty_out(self):
5417: (8)               arr = np.ones((0, 1, 1))
5418: (8)               out = np.ones((1, 1, 1))
5419: (8)               assert self.matmul(arr, arr).shape == (0, 1, 1)
5420: (8)               with pytest.raises(ValueError, match=r"non-broadcastable"):
5421: (12)                 self.matmul(arr, arr, out=out)
5422: (4)             def test_out_contiguous(self):
5423: (8)               a = np.ones((5, 2), dtype=float)
5424: (8)               b = np.array([[1, 3], [5, 7]], dtype=float)
5425: (8)               v = np.array([1, 3], dtype=float)
5426: (8)               tgt = np.dot(a, b)
5427: (8)               tgt_mv = np.dot(a, v)
5428: (8)               out = np.ones((5, 2, 2), dtype=float)
5429: (8)               c = self.matmul(a, b, out=out[..., 0])
5430: (8)               assert c.base is out
5431: (8)               assert_array_equal(c, tgt)
5432: (8)               c = self.matmul(a, v, out=out[:, 0, 0])
5433: (8)               assert_array_equal(c, tgt_mv)
5434: (8)               c = self.matmul(v, a.T, out=out[:, 0, 0])
5435: (8)               assert_array_equal(c, tgt_mv)
5436: (8)               out = np.ones((10, 2), dtype=float)
5437: (8)               c = self.matmul(a, b, out=out[:, :, :])
5438: (8)               assert_array_equal(c, tgt)
5439: (8)               out = np.ones((5, 2), dtype=float)
5440: (8)               c = self.matmul(b.T, a.T, out=out.T)
5441: (8)               assert_array_equal(out, tgt)
5442: (4)             m1 = np.arange(15.).reshape(5, 3)
5443: (4)             m2 = np.arange(21.).reshape(3, 7)
5444: (4)             m3 = np.arange(30.).reshape(5, 6)[:, ::2] # non-contiguous
5445: (4)             vc = np.arange(10.)
5446: (4)             vr = np.arange(6.)
5447: (4)             m0 = np.zeros((3, 0))
5448: (4)             @pytest.mark.parametrize('args', (
5449: (12)               (m1, m2), (m2.T, m1.T), (m2.T.copy(), m1.T), (m2.T, m1.T.copy()),
5450: (12)               (m1, m1.T), (m1.T, m1), (m1, m3.T), (m3, m1.T),
5451: (12)               (m3, m3.T), (m3.T, m3),
5452: (12)               (m3, m2), (m2.T, m3.T), (m2.T.copy(), m3.T),
5453: (12)               (m1, vr[:3]), (vc[:5], m1), (m1.T, vc[:5]), (vr[:3], m1.T),
5454: (12)               (m1, vr[:2]), (vc[:2], m1), (m1.T, vc[:2]), (vr[:2], m1.T),
5455: (12)               (m3, vr[:3]), (vc[:5], m3), (m3.T, vc[:5]), (vr[:3], m3.T),
5456: (12)               (m3, vr[:2]), (vc[:2], m3), (m3.T, vc[:2]), (vr[:2], m3.T),
5457: (12)               (m0, m0.T), (m0.T, m0), (m1, m0), (m0.T, m1.T),
5458: (8)             ))
5459: (4)             def test_dot_equivalent(self, args):
5460: (8)               r1 = np.matmul(*args)
5461: (8)               r2 = np.dot(*args)
5462: (8)               assert_equal(r1, r2)
5463: (8)               r3 = np.matmul(args[0].copy(), args[1].copy())
5464: (8)               assert_equal(r1, r3)
5465: (4)             def test_matmul_object(self):
5466: (8)               import fractions
5467: (8)               f = np.vectorize(fractions.Fraction)
5468: (8)               def random_ints():
5469: (12)                 return np.random.randint(1, 1000, size=(10, 3, 3))
5470: (8)               M1 = f(random_ints(), random_ints())
5471: (8)               M2 = f(random_ints(), random_ints())
5472: (8)               M3 = self.matmul(M1, M2)
5473: (8)               [N1, N2, N3] = [a.astype(float) for a in [M1, M2, M3]]
5474: (8)               assert_allclose(N3, self.matmul(N1, N2))
5475: (4)             def test_matmul_object_type_scalar(self):
5476: (8)               from fractions import Fraction as F
5477: (8)               v = np.array([F(2,3), F(5,7)])
5478: (8)               res = self.matmul(v, v)
5479: (8)               assert_(type(res) is F)

```

```

5480: (4)
5481: (8)
5482: (8)
5483: (8)
5484: (8)
5485: (4)
5486: (8)
5487: (12)
5488: (16)
5489: (8)
5490: (8)
5491: (12)
5492: (4)
5493: (8)
5494: (12)
5495: (16)
5496: (8)
5497: (8)
5498: (12)
5499: (4)
5500: (8)
5501: (8)
5502: (8)
5503: (8)
5504: (8)
5505: (8)
5506: (8)
5507: (8)
5508: (8)
5509: (8)
5510: (8)
5511: (8)
5512: (0)
5513: (4)
5514: (4)
5515: (4)
5516: (8)
5517: (12)
5518: (12)
5519: (16)
5520: (12)
5521: (16)
5522: (8)
5523: (8)
5524: (8)
5525: (8)
5526: (4)
5527: (8)
5528: (8)
np.void(b'abc'))
5529: (8)
5530: (0)
5531: (4)
5532: (4)
5533: (8)
5534: (12)
5535: (16)
5536: (4)
5537: (4)
5538: (8)
5539: (8)
5540: (8)
5541: (8)
5542: (8)
5543: (8)
5544: (8)
5545: (8)
5546: (8)
5547: (12)

def test_matmul_empty(self):
    a = np.empty((3, 0), dtype=object)
    b = np.empty((0, 3), dtype=object)
    c = np.zeros((3, 3))
    assert_array_equal(np.matmul(a, b), c)
def test_matmul_exception_multiply(self):
    class add_not_multiply():
        def __add__(self, other):
            return self
    a = np.full((3, 3), add_not_multiply())
    with assert_raises(TypeError):
        b = np.matmul(a, a)
def test_matmul_exception_add(self):
    class multiply_not_add():
        def __mul__(self, other):
            return self
    a = np.full((3, 3), multiply_not_add())
    with assert_raises(TypeError):
        b = np.matmul(a, a)
def test_matmul_bool(self):
    a = np.array([[1, 0], [1, 1]], dtype=bool)
    assert np.max(a.view(np.uint8)) == 1
    b = np.matmul(a, a)
    assert np.max(b.view(np.uint8)) == 1
    rg = np.random.default_rng(np.random.PCG64(43))
    d = rg.integers(2, size=4*5, dtype=np.int8)
    d = d.reshape(4, 5) > 0
    out1 = np.matmul(d, d.reshape(5, 4))
    out2 = np.dot(d, d.reshape(5, 4))
    assert_equal(out1, out2)
    c = np.matmul(np.zeros((2, 0), dtype=bool), np.zeros(0, dtype=bool))
    assert not np.any(c)
class TestMatmulOperator(MatmulCommon):
    import operator
    matmul = operator.matmul
    def test_array_priority_override(self):
        class A:
            __array_priority__ = 1000
            def __matmul__(self, other):
                return "A"
            def __rmatmul__(self, other):
                return "A"
        a = A()
        b = np.ones(2)
        assert_equal(self.matmul(a, b), "A")
        assert_equal(self.matmul(b, a), "A")
    def test_matmul_raises(self):
        assert_raises(TypeError, self.matmul, np.int8(5), np.int8(5))
        assert_raises(TypeError, self.matmul, np void(b'abc'),
                     assert_raises(TypeError, self.matmul, np.arange(10), np void(b'abc')))
class TestMatmulInplace:
    DTYPES = {}
    for i in MatmulCommon.types:
        for j in MatmulCommon.types:
            if np.can_cast(j, i):
                DTYPES[f"{i}-{j}"] = (np.dtype(i), np.dtype(j))
    @pytest.mark.parametrize("dtype1,dtype2", DTYPES.values(), ids=DTYPES)
    def test_basic(self, dtype1: np.dtype, dtype2: np.dtype) -> None:
        a = np.arange(10).reshape(5, 2).astype(dtype1)
        a_id = id(a)
        b = np.ones((2, 2), dtype=dtype2)
        ref = a @ b
        a @= b
        assert id(a) == a_id
        assert a.dtype == dtype1
        assert a.shape == (5, 2)
        if dtype1.kind in "fc":
            np.testing.assert_allclose(a, ref)

```

```

5548: (8)
5549: (12)
5550: (4)
5551: (8)
5552: (8)
5553: (8)
5554: (8)
5555: (8)
5556: (8)
5557: (8)
5558: (8)
5559: (8)
5560: (8)
5561: (8)
5562: (8)
5563: (8)
5564: (4)
5565: (4)
5566: (4)
5567: (8)
5568: (8)
5569: (8)
5570: (8)
5571: (8)
5572: (8)
5573: (8)
5574: (12)
5575: (16)
5576: (12)
5577: (8)
5578: (12)
5579: (8)
5580: (8)
5581: (8)
5582: (8)
5583: (0)
5584: (4)
5585: (4)
5586: (4)
5587: (4)
5588: (4)
5589: (4)
5590: (4)
5591: (4)
5592: (4)
5593: (0)
5594: (4)
5595: (8)
5596: (8)
5597: (8)
5598: (8)
5599: (4)
5600: (8)
5601: (12)
5602: (12)
5603: (12)
5604: (12)
5605: (12)
5606: (4)
5607: (8)
5608: (8)
5609: (8)
5610: (4)
5611: (8)
5612: (12)
5613: (12)
5614: (12)
5615: (12)
5616: (12)

        else:
            np.testing.assert_array_equal(a, ref)
    SHAPES = {
        "2d_large": ((10**5, 10), (10, 10)),
        "3d_large": ((10**4, 10, 10), (1, 10, 10)),
        "1d": ((3,), (3,)),
        "2d_1d": ((3, 3), (3,)),
        "1d_2d": ((3,), (3, 3)),
        "2d_broadcast": ((3, 3), (3, 1)),
        "2d_broadcast_reverse": ((1, 3), (3, 3)),
        "3d_broadcast1": ((3, 3, 3), (1, 3, 1)),
        "3d_broadcast2": ((3, 3, 3), (1, 3, 3)),
        "3d_broadcast3": ((3, 3, 3), (3, 3, 1)),
        "3d_broadcast_reverse1": ((1, 3, 3), (3, 3, 3)),
        "3d_broadcast_reverse2": ((3, 1, 3), (3, 3, 3)),
        "3d_broadcast_reverse3": ((1, 1, 3), (3, 3, 3)),
    }
    @pytest.mark.parametrize("a_shape,b_shape", SHAPES.values(), ids=SHAPES)
    def test_shapes(self, a_shape: tuple[int, ...], b_shape: tuple[int, ...]):
        a_size = np.prod(a_shape)
        a = np.arange(a_size).reshape(a_shape).astype(np.float64)
        a_id = id(a)
        b_size = np.prod(b_shape)
        b = np.arange(b_size).reshape(b_shape)
        ref = a @ b
        if ref.shape != a_shape:
            with pytest.raises(ValueError):
                a @= b
            return
        else:
            a @= b
            assert id(a) == a_id
            assert a.dtype.type == np.float64
            assert a.shape == a_shape
            np.testing.assert_allclose(a, ref)

    def test_matmul_axes():
        a = np.arange(3*4*5).reshape(3, 4, 5)
        c = np.matmul(a, a, axes=[(-2, -1), (-1, -2), (1, 2)])
        assert c.shape == (3, 4, 4)
        d = np.matmul(a, a, axes=[(-2, -1), (-1, -2), (0, 1)])
        assert d.shape == (4, 4, 3)
        e = np.swapaxes(d, 0, 2)
        assert_array_equal(e, c)
        f = np.matmul(a, np.arange(3), axes=[(1, 0), (0, 0), (0)])
        assert f.shape == (4, 5)

    class TestInner:
        def test_inner_type_mismatch(self):
            c = 1.
            A = np.array((1,1), dtype='i,i')
            assert_raises(TypeError, np.inner, c, A)
            assert_raises(TypeError, np.inner, A, c)
        def test_inner_scalar_and_vector(self):
            for dt in np.typecodes['AllInteger'] + np.typecodes['AllFloat'] + '?':
                sca = np.array(3, dtype=dt)[()]
                vec = np.array([1, 2], dtype=dt)
                desired = np.array([3, 6], dtype=dt)
                assert_equal(np.inner(vec, sca), desired)
                assert_equal(np.inner(sca, vec), desired)
        def test_vecself(self):
            a = np.zeros(shape=(1, 80), dtype=np.float64)
            p = np.inner(a, a)
            assert_almost_equal(p, 0, decimal=14)
        def test_inner_product_with_various_contiguities(self):
            for dt in np.typecodes['AllInteger'] + np.typecodes['AllFloat'] + '?':
                A = np.array([[1, 2], [3, 4]], dtype=dt)
                B = np.array([[1, 3], [2, 4]], dtype=dt)
                C = np.array([1, 1], dtype=dt)
                desired = np.array([4, 6], dtype=dt)
                assert_equal(np.inner(A.T, C), desired)

```

```

5617: (12)             assert_equal(np.inner(C, A.T), desired)
5618: (12)             assert_equal(np.inner(B, C), desired)
5619: (12)             assert_equal(np.inner(C, B), desired)
5620: (12)             desired = np.array([[7, 10], [15, 22]], dtype=dt)
5621: (12)             assert_equal(np.inner(A, B), desired)
5622: (12)             desired = np.array([[5, 11], [11, 25]], dtype=dt)
5623: (12)             assert_equal(np.inner(A, A), desired)
5624: (12)             assert_equal(np.inner(A, A.copy()), desired)
5625: (12)             a = np.arange(5).astype(dt)
5626: (12)             b = a[::-1]
5627: (12)             desired = np.array(10, dtype=dt).item()
5628: (12)             assert_equal(np.inner(b, a), desired)
5629: (4)              def test_3d_tensor(self):
5630: (8)                for dt in np.typecodes['AllInteger'] + np.typecodes['AllFloat'] + '?':
5631: (12)                  a = np.arange(24).reshape(2,3,4).astype(dt)
5632: (12)                  b = np.arange(24, 48).reshape(2,3,4).astype(dt)
5633: (12)                  desired = np.array(
5634: (16)                      [[[[ 158, 182, 206],
5635: (19)                          [ 230, 254, 278]],
5636: (18)                          [[ 566, 654, 742],
5637: (19)                              [ 830, 918, 1006]],
5638: (18)                              [[ 974, 1126, 1278],
5639: (19)                                  [1430, 1582, 1734]]],
5640: (17)                      [[[1382, 1598, 1814],
5641: (19)                          [2030, 2246, 2462]],
5642: (18)                          [[1790, 2070, 2350],
5643: (19)                              [2630, 2910, 3190]],
5644: (18)                              [[2198, 2542, 2886],
5645: (19)                                  [3230, 3574, 3918]]]]
5646: (12)                      ).astype(dt)
5647: (12)                      assert_equal(np.inner(a, b), desired)
5648: (12)                      assert_equal(np.inner(b, a).transpose(2,3,0,1), desired)
5649: (0)           class TestChoose:
5650: (4)             def setup_method(self):
5651: (8)               self.x = 2*np.ones((3,), dtype=int)
5652: (8)               self.y = 3*np.ones((3,), dtype=int)
5653: (8)               self.x2 = 2*np.ones((2, 3), dtype=int)
5654: (8)               self.y2 = 3*np.ones((2, 3), dtype=int)
5655: (8)               self.ind = [0, 0, 1]
5656: (4)             def test_basic(self):
5657: (8)               A = np.choose(self.ind, (self.x, self.y))
5658: (8)               assert_equal(A, [2, 2, 3])
5659: (4)             def test_broadcast1(self):
5660: (8)               A = np.choose(self.ind, (self.x2, self.y2))
5661: (8)               assert_equal(A, [[2, 2, 3], [2, 2, 3]])
5662: (4)             def test_broadcast2(self):
5663: (8)               A = np.choose(self.ind, (self.x, self.y2))
5664: (8)               assert_equal(A, [[2, 2, 3], [2, 2, 3]])
5665: (4)             @pytest.mark.parametrize("ops",
5666: (8)                 [(1000, np.array([1], dtype=np.uint8)),
5667: (9)                     (-1, np.array([1], dtype=np.uint8)),
5668: (9)                     (1., np.float32(3)),
5669: (9)                     (1., np.array([3], dtype=np.float32))],)
5670: (4)             def test_output_dtype(self, ops):
5671: (8)               expected_dt = np.result_type(*ops)
5672: (8)               assert(np.choose([0], ops).dtype == expected_dt)
5673: (0)           class TestRepeat:
5674: (4)             def setup_method(self):
5675: (8)               self.m = np.array([1, 2, 3, 4, 5, 6])
5676: (8)               self.m_rect = self.m.reshape((2, 3))
5677: (4)             def test_basic(self):
5678: (8)               A = np.repeat(self.m, [1, 3, 2, 1, 1, 2])
5679: (8)               assert_equal(A, [1, 2, 2, 2, 3,
5680: (25)                               3, 4, 5, 6])
5681: (4)             def test_broadcast1(self):
5682: (8)               A = np.repeat(self.m, 2)
5683: (8)               assert_equal(A, [1, 1, 2, 2, 3, 3,
5684: (25)                               4, 4, 5, 5, 6, 6])
5685: (4)             def test_axis_spec(self):

```

```

5686: (8)          A = np.repeat(self.m_rect, [2, 1], axis=0)
5687: (8)          assert_equal(A, [[1, 2, 3],
5688: (25)            [1, 2, 3],
5689: (25)            [4, 5, 6]])
5690: (8)          A = np.repeat(self.m_rect, [1, 3, 2], axis=1)
5691: (8)          assert_equal(A, [[1, 2, 2, 2, 3, 3],
5692: (25)            [4, 5, 5, 5, 6, 6]])
5693: (4)          def test_broadcast2(self):
5694: (8)            A = np.repeat(self.m_rect, 2, axis=0)
5695: (8)            assert_equal(A, [[1, 2, 3],
5696: (25)              [1, 2, 3],
5697: (25)              [4, 5, 6],
5698: (25)              [4, 5, 6]])
5699: (8)            A = np.repeat(self.m_rect, 2, axis=1)
5700: (8)            assert_equal(A, [[1, 1, 2, 2, 3, 3],
5701: (25)              [4, 4, 5, 5, 6, 6]])
5702: (0)          NEIGH_MODE = {'zero': 0, 'one': 1, 'constant': 2, 'circular': 3, 'mirror': 4}
5703: (0)          @pytest.mark.parametrize('dt', [float, Decimal], ids=['float', 'object'])
5704: (0)          class TestNeighborhoodIter:
5705: (4)            def test_simple2d(self, dt):
5706: (8)              x = np.array([[0, 1], [2, 3]], dtype=dt)
5707: (8)              r = [np.array([[0, 0, 0], [0, 0, 1]], dtype=dt),
5708: (13)                np.array([[0, 0, 0], [0, 1, 0]], dtype=dt),
5709: (13)                np.array([[0, 0, 1], [0, 2, 3]], dtype=dt),
5710: (13)                np.array([[0, 1, 0], [2, 3, 0]], dtype=dt)]
5711: (8)              l = _multiarray_tests.test_neighborhood_iterator(
5712: (16)                x, [-1, 0, -1, 1], x[0], NEIGH_MODE['zero'])
5713: (8)              assert_array_equal(l, r)
5714: (8)              r = [np.array([[1, 1, 1], [1, 0, 1]], dtype=dt),
5715: (13)                np.array([[1, 1, 1], [0, 1, 1]], dtype=dt),
5716: (13)                np.array([[1, 0, 1], [1, 2, 3]], dtype=dt),
5717: (13)                np.array([[0, 1, 1], [2, 3, 1]], dtype=dt)]
5718: (8)              l = _multiarray_tests.test_neighborhood_iterator(
5719: (16)                x, [-1, 0, -1, 1], x[0], NEIGH_MODE['one'])
5720: (8)              assert_array_equal(l, r)
5721: (8)              r = [np.array([[4, 4, 4], [4, 0, 1]], dtype=dt),
5722: (13)                np.array([[4, 4, 4], [0, 1, 4]], dtype=dt),
5723: (13)                np.array([[4, 0, 1], [4, 2, 3]], dtype=dt),
5724: (13)                np.array([[0, 1, 4], [2, 3, 4]], dtype=dt)]
5725: (8)              l = _multiarray_tests.test_neighborhood_iterator(
5726: (16)                x, [-1, 0, -1, 1], 4, NEIGH_MODE['constant'])
5727: (8)              assert_array_equal(l, r)
5728: (8)              r = [np.array([[4, 0, 1], [4, 2, 3]], dtype=dt),
5729: (13)                np.array([[0, 1, 4], [2, 3, 4]], dtype=dt)]
5730: (8)              l = _multiarray_tests.test_neighborhood_iterator(
5731: (16)                x, [-1, 0, -1, 1], 4, NEIGH_MODE['constant'], 2)
5732: (8)              assert_array_equal(l, r)
5733: (4)          def test_mirror2d(self, dt):
5734: (8)            x = np.array([[0, 1], [2, 3]], dtype=dt)
5735: (8)            r = [np.array([[0, 0, 1], [0, 0, 1]], dtype=dt),
5736: (13)              np.array([[0, 1, 1], [0, 1, 1]], dtype=dt),
5737: (13)              np.array([[0, 0, 1], [2, 2, 3]], dtype=dt),
5738: (13)              np.array([[0, 1, 1], [2, 3, 3]], dtype=dt)]
5739: (8)            l = _multiarray_tests.test_neighborhood_iterator(
5740: (16)              x, [-1, 0, -1, 1], x[0], NEIGH_MODE['mirror'])
5741: (8)            assert_array_equal(l, r)
5742: (4)          def test_simple(self, dt):
5743: (8)            x = np.linspace(1, 5, 5).astype(dt)
5744: (8)            r = [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 0]]
5745: (8)            l = _multiarray_tests.test_neighborhood_iterator(
5746: (16)              x, [-1, 1], x[0], NEIGH_MODE['zero'])
5747: (8)            assert_array_equal(l, r)
5748: (8)            r = [[1, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 1]]
5749: (8)            l = _multiarray_tests.test_neighborhood_iterator(
5750: (16)              x, [-1, 1], x[0], NEIGH_MODE['one'])
5751: (8)            assert_array_equal(l, r)
5752: (8)            r = [[x[4], 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, x[4]]]
5753: (8)            l = _multiarray_tests.test_neighborhood_iterator(
5754: (16)              x, [-1, 1], x[4], NEIGH_MODE['constant'])

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

5755: (8)           assert_array_equal(l, r)
5756: (4)           def test_mirror(self, dt):
5757: (8)             x = np.linspace(1, 5, 5).astype(dt)
5758: (8)             r = np.array([[2, 1, 1, 2, 3], [1, 1, 2, 3, 4], [1, 2, 3, 4, 5],
5759: (16)               [2, 3, 4, 5, 5], [3, 4, 5, 5, 4]], dtype=dt)
5760: (8)             l = _multiarray_tests.test_neighborhood_iterator(
5761: (16)               x, [-2, 2], x[1], NEIGH_MODE['mirror'])
5762: (8)             assert_([i.dtype == dt for i in l])
5763: (8)             assert_array_equal(l, r)
5764: (4)           def test_circular(self, dt):
5765: (8)             x = np.linspace(1, 5, 5).astype(dt)
5766: (8)             r = np.array([[4, 5, 1, 2, 3], [5, 1, 2, 3, 4], [1, 2, 3, 4, 5],
5767: (16)               [2, 3, 4, 5, 1], [3, 4, 5, 1, 2]], dtype=dt)
5768: (8)             l = _multiarray_tests.test_neighborhood_iterator(
5769: (16)               x, [-2, 2], x[0], NEIGH_MODE['circular'])
5770: (8)             assert_array_equal(l, r)
5771: (0)           class TestStackedNeighborhoodIter:
5772: (4)             def test_simple_const(self):
5773: (8)               dt = np.float64
5774: (8)               x = np.array([1, 2, 3], dtype=dt)
5775: (8)               r = [np.array([0], dtype=dt),
5776: (13)                 np.array([0], dtype=dt),
5777: (13)                 np.array([1], dtype=dt),
5778: (13)                 np.array([2], dtype=dt),
5779: (13)                 np.array([3], dtype=dt),
5780: (13)                 np.array([0], dtype=dt),
5781: (13)                 np.array([0], dtype=dt)]
5782: (8)               l = _multiarray_tests.test_neighborhood_iterator_oob(
5783: (16)                 x, [-2, 4], NEIGH_MODE['zero'], [0, 0], NEIGH_MODE['zero'])
5784: (8)               assert_array_equal(l, r)
5785: (8)               r = [np.array([1, 0, 1], dtype=dt),
5786: (13)                 np.array([0, 1, 2], dtype=dt),
5787: (13)                 np.array([1, 2, 3], dtype=dt),
5788: (13)                 np.array([2, 3, 0], dtype=dt),
5789: (13)                 np.array([3, 0, 1], dtype=dt)]
5790: (8)               l = _multiarray_tests.test_neighborhood_iterator_oob(
5791: (16)                 x, [-1, 3], NEIGH_MODE['zero'], [-1, 1], NEIGH_MODE['one'])
5792: (8)               assert_array_equal(l, r)
5793: (4)             def test_simple_mirror(self):
5794: (8)               dt = np.float64
5795: (8)               x = np.array([1, 2, 3], dtype=dt)
5796: (8)               r = [np.array([0, 1, 1], dtype=dt),
5797: (13)                 np.array([1, 1, 2], dtype=dt),
5798: (13)                 np.array([1, 2, 3], dtype=dt),
5799: (13)                 np.array([2, 3, 3], dtype=dt),
5800: (13)                 np.array([3, 3, 0], dtype=dt)]
5801: (8)               l = _multiarray_tests.test_neighborhood_iterator_oob(
5802: (16)                 x, [-1, 3], NEIGH_MODE['mirror'], [-1, 1], NEIGH_MODE['zero'])
5803: (8)               assert_array_equal(l, r)
5804: (8)               x = np.array([1, 2, 3], dtype=dt)
5805: (8)               r = [np.array([1, 0, 0], dtype=dt),
5806: (13)                 np.array([0, 0, 1], dtype=dt),
5807: (13)                 np.array([0, 1, 2], dtype=dt),
5808: (13)                 np.array([1, 2, 3], dtype=dt),
5809: (13)                 np.array([2, 3, 0], dtype=dt)]
5810: (8)               l = _multiarray_tests.test_neighborhood_iterator_oob(
5811: (16)                 x, [-1, 3], NEIGH_MODE['zero'], [-2, 0], NEIGH_MODE['mirror'])
5812: (8)               assert_array_equal(l, r)
5813: (8)               x = np.array([1, 2, 3], dtype=dt)
5814: (8)               r = [np.array([0, 1, 2], dtype=dt),
5815: (13)                 np.array([1, 2, 3], dtype=dt),
5816: (13)                 np.array([2, 3, 0], dtype=dt),
5817: (13)                 np.array([3, 0, 0], dtype=dt),
5818: (13)                 np.array([0, 0, 3], dtype=dt)]
5819: (8)               l = _multiarray_tests.test_neighborhood_iterator_oob(
5820: (16)                 x, [-1, 3], NEIGH_MODE['zero'], [0, 2], NEIGH_MODE['mirror'])
5821: (8)               assert_array_equal(l, r)
5822: (8)               x = np.array([1, 2, 3], dtype=dt)
5823: (8)               r = [np.array([1, 0, 0, 1, 2], dtype=dt),

```

```

5824: (13) np.array([0, 0, 1, 2, 3], dtype=dt),
5825: (13) np.array([0, 1, 2, 3, 0], dtype=dt),
5826: (13) np.array([1, 2, 3, 0, 0], dtype=dt),
5827: (13) np.array([2, 3, 0, 0, 3], dtype=dt)]
5828: (8) l = _multiarray_tests.test_neighborhood_iterator_oob(
5829: (16)     x, [-1, 3], NEIGH_MODE['zero'], [-2, 2], NEIGH_MODE['mirror'])
5830: (8) assert_array_equal(l, r)
5831: (4) def test_simple_circular(self):
5832: (8)     dt = np.float64
5833: (8)     x = np.array([1, 2, 3], dtype=dt)
5834: (8)     r = [np.array([0, 3, 1], dtype=dt),
5835: (13)         np.array([3, 1, 2], dtype=dt),
5836: (13)         np.array([1, 2, 3], dtype=dt),
5837: (13)         np.array([2, 3, 1], dtype=dt),
5838: (13)         np.array([3, 1, 0], dtype=dt)]
5839: (8)     l = _multiarray_tests.test_neighborhood_iterator_oob(
5840: (16)     x, [-1, 3], NEIGH_MODE['circular'], [-1, 1],
NEIGH_MODE['zero'])

5841: (8) assert_array_equal(l, r)
5842: (8) x = np.array([1, 2, 3], dtype=dt)
5843: (8) r = [np.array([3, 0, 0], dtype=dt),
5844: (13)     np.array([0, 0, 1], dtype=dt),
5845: (13)     np.array([0, 1, 2], dtype=dt),
5846: (13)     np.array([1, 2, 3], dtype=dt),
5847: (13)     np.array([2, 3, 0], dtype=dt)]
5848: (8) l = _multiarray_tests.test_neighborhood_iterator_oob(
5849: (16)     x, [-1, 3], NEIGH_MODE['zero'], [-2, 0],
NEIGH_MODE['circular'])

5850: (8) assert_array_equal(l, r)
5851: (8) x = np.array([1, 2, 3], dtype=dt)
5852: (8) r = [np.array([0, 1, 2], dtype=dt),
5853: (13)     np.array([1, 2, 3], dtype=dt),
5854: (13)     np.array([2, 3, 0], dtype=dt),
5855: (13)     np.array([3, 0, 0], dtype=dt),
5856: (13)     np.array([0, 0, 1], dtype=dt)]
5857: (8) l = _multiarray_tests.test_neighborhood_iterator_oob(
5858: (16)     x, [-1, 3], NEIGH_MODE['zero'], [0, 2],
NEIGH_MODE['circular'])

5859: (8) assert_array_equal(l, r)
5860: (8) x = np.array([1, 2, 3], dtype=dt)
5861: (8) r = [np.array([3, 0, 0, 1, 2], dtype=dt),
5862: (13)     np.array([0, 0, 1, 2, 3], dtype=dt),
5863: (13)     np.array([0, 1, 2, 3, 0], dtype=dt),
5864: (13)     np.array([1, 2, 3, 0, 0], dtype=dt),
5865: (13)     np.array([2, 3, 0, 0, 1], dtype=dt)]
5866: (8) l = _multiarray_tests.test_neighborhood_iterator_oob(
5867: (16)     x, [-1, 3], NEIGH_MODE['zero'], [-2, 2],
NEIGH_MODE['circular'])

5868: (8) assert_array_equal(l, r)
5869: (4) def test_simple_strict_within(self):
5870: (8)     dt = np.float64
5871: (8)     x = np.array([1, 2, 3], dtype=dt)
5872: (8)     r = [np.array([1, 2, 3, 0], dtype=dt)]
5873: (8)     l = _multiarray_tests.test_neighborhood_iterator_oob(
5874: (16)     x, [1, 1], NEIGH_MODE['zero'], [-1, 2], NEIGH_MODE['zero'])
5875: (8)     assert_array_equal(l, r)
5876: (8)     x = np.array([1, 2, 3], dtype=dt)
5877: (8)     r = [np.array([1, 2, 3, 3], dtype=dt)]
5878: (8)     l = _multiarray_tests.test_neighborhood_iterator_oob(
5879: (16)     x, [1, 1], NEIGH_MODE['zero'], [-1, 2], NEIGH_MODE['mirror'])
5880: (8)     assert_array_equal(l, r)
5881: (8)     x = np.array([1, 2, 3], dtype=dt)
5882: (8)     r = [np.array([1, 2, 3, 1], dtype=dt)]
5883: (8)     l = _multiarray_tests.test_neighborhood_iterator_oob(
5884: (16)     x, [1, 1], NEIGH_MODE['zero'], [-1, 2],
NEIGH_MODE['circular'])

5885: (8)     assert_array_equal(l, r)
5886: (0)     class TestWarnings:
5887: (4)         def test_complex_warning(self):

```

```

5888: (8)          x = np.array([1, 2])
5889: (8)          y = np.array([1-2j, 1+2j])
5890: (8)          with warnings.catch_warnings():
5891: (12)             warnings.simplefilter("error", np.ComplexWarning)
5892: (12)             assert_raises(np.ComplexWarning, x.__setitem__, slice(None), y)
5893: (12)             assert_equal(x, [1, 2])
5894: (0)          class TestMinScalarType:
5895: (4)              def test_usigned_shortshort(self):
5896: (8)                  dt = np.min_scalar_type(2**8-1)
5897: (8)                  wanted = np.dtype('uint8')
5898: (8)                  assert_equal(wanted, dt)
5899: (4)              def test_usigned_short(self):
5900: (8)                  dt = np.min_scalar_type(2**16-1)
5901: (8)                  wanted = np.dtype('uint16')
5902: (8)                  assert_equal(wanted, dt)
5903: (4)              def test_usigned_int(self):
5904: (8)                  dt = np.min_scalar_type(2**32-1)
5905: (8)                  wanted = np.dtype('uint32')
5906: (8)                  assert_equal(wanted, dt)
5907: (4)              def test_usigned_longlong(self):
5908: (8)                  dt = np.min_scalar_type(2**63-1)
5909: (8)                  wanted = np.dtype('uint64')
5910: (8)                  assert_equal(wanted, dt)
5911: (4)              def test_object(self):
5912: (8)                  dt = np.min_scalar_type(2**64)
5913: (8)                  wanted = np.dtype('O')
5914: (8)                  assert_equal(wanted, dt)
5915: (0)          from numpy.core._internal import _dtype_from_pep3118
5916: (0)          class TestPEP3118Dtype:
5917: (4)              def _check(self, spec, wanted):
5918: (8)                  dt = np.dtype(wanted)
5919: (8)                  actual = _dtype_from_pep3118(spec)
5920: (8)                  assert_equal(actual, dt,
5921: (21)                      err_msg="spec %r != dtype %r" % (spec, wanted))
5922: (4)              def test_native_padding(self):
5923: (8)                  align = np.dtype('i').alignment
5924: (8)                  for j in range(8):
5925: (12)                      if j == 0:
5926: (16)                          s = 'bi'
5927: (12)                      else:
5928: (16)                          s = 'b%dx{i' % j
5929: (12)                  self._check('@'+s, {'f0': ('i1', 0),
5930: (32)                                  'f1': ('i', align*(1 + j//align))})
5931: (12)                  self._check('='+s, {'f0': ('i1', 0),
5932: (32)                                  'f1': ('i', 1+j)})
5933: (4)              def test_native_padding_2(self):
5934: (8)                  self._check('x3T{xi}', {'f0': (({'f0': ('i', 4)}, (3,)), 4)})
5935: (8)                  self._check('^x3T{xi}', {'f0': (({'f0': ('i', 1)}, (3,)), 1)})
5936: (4)              def test_trailing_padding(self):
5937: (8)                  align = np.dtype('i').alignment
5938: (8)                  size = np.dtype('i').itemsize
5939: (8)                  def aligned(n):
5940: (12)                      return align*(1 + (n-1)//align)
5941: (8)                  base = dict(formats=['i'], names=['f0'])
5942: (8)                  self._check('ix', dict(itemsize=aligned(size + 1), **base))
5943: (8)                  self._check('ixx', dict(itemsize=aligned(size + 2), **base))
5944: (8)                  self._check('ixxx', dict(itemsize=aligned(size + 3), **base))
5945: (8)                  self._check('ixxxx', dict(itemsize=aligned(size + 4), **base))
5946: (8)                  self._check('i7x', dict(itemsize=aligned(size + 7), **base))
5947: (8)                  self._check('^ix', dict(itemsize=size + 1, **base))
5948: (8)                  self._check('^ixx', dict(itemsize=size + 2, **base))
5949: (8)                  self._check('^ixxx', dict(itemsize=size + 3, **base))
5950: (8)                  self._check('^ixxxx', dict(itemsize=size + 4, **base))
5951: (8)                  self._check('^i7x', dict(itemsize=size + 7, **base))
5952: (4)              def test_native_padding_3(self):
5953: (8)                  dt = np.dtype(
5954: (16)                      [(['a', 'b'), ('b', 'i'),
5955: (20)                          ('sub', np.dtype('b,i'))), ('c', 'i')], align=True)

```

```

5957: (8)           self._check("T{b:a:xxxi:b:T{b:f0:=i:f1:}:sub:xxxi:c:}", dt)
5958: (8)           dt = np.dtype(
5959: (16)             [ ('a', 'b'), ('b', 'i'), ('c', 'b'), ('d', 'b'),
5960: (20)               ('e', 'b'), ('sub', np.dtype('b,i', align=True))])
5961: (8)           self._check("T{b:a:=i:b:b:c:b:d:b:e:T{b:f0:xxxi:f1:}:sub:}", dt)
5962: (4)           def test_padding_with_array_inside_struct(self):
5963: (8)             dt = np.dtype(
5964: (16)               [ ('a', 'b'), ('b', 'i'), ('c', 'b', (3,)),
5965: (20)                 ('d', 'i')], align=True)
5966: (16)             self._check("T{b:a:xxxi:b:3b:c:xi:d:}", dt)
5967: (8)           def test_byteorder_inside_struct(self):
5968: (4)             self._check('@T{^i}xi', {'f0': {'f0': ('i', 0)}, 'f1': ('i', 5)})
5969: (8)
5970: (33)
5971: (4)           def test_intra_padding(self):
5972: (8)             align = np.dtype('i').alignment
5973: (8)             size = np.dtype('i').itemsize
5974: (8)             def aligned(n):
5975: (12)               return (align*(1 + (n-1)//align))
5976: (8)             self._check('(3)T{ix}', (dict(
5977: (12)               names=['f0'],
5978: (12)               formats=['i'],
5979: (12)               offsets=[0],
5980: (12)               itemsize=aligned(size + 1)
5981: (8)             ), (3,)))
5982: (4)           def test_char_vs_string(self):
5983: (8)             dt = np.dtype('c')
5984: (8)             self._check('c', dt)
5985: (8)             dt = np.dtype([('f0', 'S1', (4,)), ('f1', 'S4')])
5986: (8)             self._check('4c4s', dt)
5987: (4)           def test_field_order(self):
5988: (8)             self._check("(0)I:a:f:b:", [('a', 'I', (0,)), ('b', 'f')])
5989: (8)             self._check("(0)I:b:f:a:", [('b', 'I', (0,)), ('a', 'f')])
5990: (4)           def test_unnamed_fields(self):
5991: (8)             self._check('ii', [('f0', 'i'), ('f1', 'i')])
5992: (8)             self._check('ii:f0:', [('f1', 'i'), ('f0', 'i')])
5993: (8)             self._check('i', 'i')
5994: (8)             self._check('i:f0:', [('f0', 'i')])
5995: (0)           class TestNewBufferProtocol:
5996: (4)             """ Test PEP3118 buffers """
5997: (4)             def _check_roundtrip(self, obj):
5998: (8)               obj = np.asarray(obj)
5999: (8)               x = memoryview(obj)
6000: (8)               y = np.asarray(x)
6001: (8)               y2 = np.array(x)
6002: (8)               assert_(not y.flags.owndata)
6003: (8)               assert_(y2.flags.owndata)
6004: (8)               assert_equal(y.dtype, obj.dtype)
6005: (8)               assert_equal(y.shape, obj.shape)
6006: (8)               assert_array_equal(obj, y)
6007: (8)               assert_equal(y2.dtype, obj.dtype)
6008: (8)               assert_equal(y2.shape, obj.shape)
6009: (8)               assert_array_equal(obj, y2)
6010: (4)             def test_roundtrip(self):
6011: (8)               x = np.array([1, 2, 3, 4, 5], dtype='i4')
6012: (8)               self._check_roundtrip(x)
6013: (8)               x = np.array([[1, 2], [3, 4]], dtype=np.float64)
6014: (8)               self._check_roundtrip(x)
6015: (8)               x = np.zeros((3, 3, 3), dtype=np.float32)[:, 0, :]
6016: (8)               self._check_roundtrip(x)
6017: (8)               dt = [('a', 'b'),
6018: (14)                 ('b', 'h'),
6019: (14)                 ('c', 'i'),
6020: (14)                 ('d', 'l'),
6021: (14)                 ('dx', 'q'),
6022: (14)                 ('e', 'B'),
6023: (14)                 ('f', 'H'),
6024: (14)                 ('g', 'I'),
6025: (14)                 ('h', 'L'),
```

```

6026: (14)          ('hx', 'Q'),
6027: (14)          ('i', np.single),
6028: (14)          ('j', np.double),
6029: (14)          ('k', np.longdouble),
6030: (14)          ('ix', np.csingle),
6031: (14)          ('jx', np.cdouble),
6032: (14)          ('kx', np.clongdouble),
6033: (14)          ('l', 'S4'),
6034: (14)          ('m', 'U4'),
6035: (14)          ('n', 'V3'),
6036: (14)          ('o', '?'),
6037: (14)          ('p', np.half),
6038: (14)          ]
6039: (8)          x = np.array(
6040: (16)            [(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
6041: (20)              b'aaaa', 'bbbb', b'xxx', True, 1.0)],
6042: (16)            dtype=dt)
6043: (8)          self._check_roundtrip(x)
6044: (8)          x = np.array(([1, 2], [3, 4]),), dtype=[('a', (int, (2, 2))))]
6045: (8)          self._check_roundtrip(x)
6046: (8)          x = np.array([1, 2, 3], dtype='>i2')
6047: (8)          self._check_roundtrip(x)
6048: (8)          x = np.array([1, 2, 3], dtype='<i2')
6049: (8)          self._check_roundtrip(x)
6050: (8)          x = np.array([1, 2, 3], dtype='>i4')
6051: (8)          self._check_roundtrip(x)
6052: (8)          x = np.array([1, 2, 3], dtype='<i4')
6053: (8)          self._check_roundtrip(x)
6054: (8)          x = np.array([1, 2, 3], dtype='>q')
6055: (8)          self._check_roundtrip(x)
6056: (8)          if sys.byteorder == 'little':
6057: (12)            x = np.array([1, 2, 3], dtype='>g')
6058: (12)            assert_raises(ValueError, self._check_roundtrip, x)
6059: (12)            x = np.array([1, 2, 3], dtype='<g')
6060: (12)            self._check_roundtrip(x)
6061: (8)          else:
6062: (12)            x = np.array([1, 2, 3], dtype='>g')
6063: (12)            self._check_roundtrip(x)
6064: (12)            x = np.array([1, 2, 3], dtype='<g')
6065: (12)            assert_raises(ValueError, self._check_roundtrip, x)
6066: (4)          def test_roundtrip_half(self):
6067: (8)            half_list = [
6068: (12)              1.0,
6069: (12)              -2.0,
6070: (12)              6.5504 * 10**4, # (max half precision)
6071: (12)              2**-14, # ~= 6.10352 * 10**-5 (minimum positive normal)
6072: (12)              2**-24, # ~= 5.96046 * 10**-8 (minimum strictly positive
subnormal)
6073: (12)              0.0,
6074: (12)              -0.0,
6075: (12)              float('+inf'),
6076: (12)              float('-inf'),
6077: (12)              0.333251953125, # ~= 1/3
6078: (8)            ]
6079: (8)            x = np.array(half_list, dtype='>e')
6080: (8)            self._check_roundtrip(x)
6081: (8)            x = np.array(half_list, dtype='<e')
6082: (8)            self._check_roundtrip(x)
6083: (4)          def test_roundtrip_single_types(self):
6084: (8)            for typ in np.sctypeDict.values():
6085: (12)              dtype = np.dtype(typ)
6086: (12)              if dtype.char in 'Mm':
6087: (16)                continue
6088: (12)              if dtype.char == 'V':
6089: (16)                continue
6090: (12)              x = np.zeros(4, dtype=dtype)
6091: (12)              self._check_roundtrip(x)
6092: (12)              if dtype.char not in 'qQgG':
6093: (16)                dt = dtype.newbyteorder('<')

```

```

6094: (16)           x = np.zeros(4, dtype=dt)
6095: (16)           self._check_roundtrip(x)
6096: (16)           dt = dtype.newbyteorder('>')
6097: (16)           x = np.zeros(4, dtype=dt)
6098: (16)           self._check_roundtrip(x)
6099: (4)            def test_roundtrip_scalar(self):
6100: (8)              self._check_roundtrip(0)
6101: (4)            def test_invalid_buffer_format(self):
6102: (8)              dt = np.dtype([('a', 'uint16'), ('b', 'M8[s]')])
6103: (8)              a = np.empty(3, dt)
6104: (8)              assert_raises((ValueError, BufferError), memoryview, a)
6105: (8)              assert_raises((ValueError, BufferError), memoryview, np.array((3),
6106: 'M8[D'])))
6106: (4)            def test_export_simple_1d(self):
6107: (8)              x = np.array([1, 2, 3, 4, 5], dtype='i')
6108: (8)              y = memoryview(x)
6109: (8)              assert_equal(y.format, 'i')
6110: (8)              assert_equal(y.shape, (5,))
6111: (8)              assert_equal(y.ndim, 1)
6112: (8)              assert_equal(y.strides, (4,))
6113: (8)              assert_equal(y.suboffsets, ())
6114: (8)              assert_equal(y.itemsize, 4)
6115: (4)            def test_export_simple_nd(self):
6116: (8)              x = np.array([[1, 2], [3, 4]], dtype=np.float64)
6117: (8)              y = memoryview(x)
6118: (8)              assert_equal(y.format, 'd')
6119: (8)              assert_equal(y.shape, (2, 2))
6120: (8)              assert_equal(y.ndim, 2)
6121: (8)              assert_equal(y.strides, (16, 8))
6122: (8)              assert_equal(y.suboffsets, ())
6123: (8)              assert_equal(y.itemsize, 8)
6124: (4)            def test_export_discontiguous(self):
6125: (8)              x = np.zeros((3, 3, 3), dtype=np.float32)[:, 0, :]
6126: (8)              y = memoryview(x)
6127: (8)              assert_equal(y.format, 'f')
6128: (8)              assert_equal(y.shape, (3, 3))
6129: (8)              assert_equal(y.ndim, 2)
6130: (8)              assert_equal(y.strides, (36, 4))
6131: (8)              assert_equal(y.suboffsets, ())
6132: (8)              assert_equal(y.itemsize, 4)
6133: (4)            def test_export_record(self):
6134: (8)              dt = [('a', 'b'),
6135: (14)                  ('b', 'h'),
6136: (14)                  ('c', 'i'),
6137: (14)                  ('d', 'l'),
6138: (14)                  ('dx', 'q'),
6139: (14)                  ('e', 'B'),
6140: (14)                  ('f', 'H'),
6141: (14)                  ('g', 'I'),
6142: (14)                  ('h', 'L'),
6143: (14)                  ('hx', 'Q'),
6144: (14)                  ('i', np.single),
6145: (14)                  ('j', np.double),
6146: (14)                  ('k', np.longdouble),
6147: (14)                  ('ix', np.csingle),
6148: (14)                  ('jx', np.cdouble),
6149: (14)                  ('kx', np.clongdouble),
6150: (14)                  ('l', 'S4'),
6151: (14)                  ('m', 'U4'),
6152: (14)                  ('n', 'V3'),
6153: (14)                  ('o', '?'),
6154: (14)                  ('p', np.half),
6155: (14)                  ]
6156: (8)            x = np.array(
6157: (16)                [(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
6158: (20)                    b'aaaa', 'bbbb', b' ', True, 1.0)],
6159: (16)                dtype=dt)
6160: (8)            y = memoryview(x)
6161: (8)            assert_equal(y.shape, (1,))
```

```

6162: (8)             assert_equal(y.ndim, 1)
6163: (8)             assert_equal(y.suboffsets, ())
6164: (8)             sz = sum([np.dtype(b).itemsize for a, b in dt])
6165: (8)             if np.dtype('l').itemsize == 4:
6166: (12)                 assert_equal(y.format,
' T{b:a:=h:b:i:c:l:d:q:dx:B:e:@H:f:=I:g:L:h:Q:hx:f:i:d:j:^g:k:=Zf:ix:Zd:jx:^Zg:kx:4s:l:=4w:m:3x:n?:?
:o:@e:p:}')
6167: (8)             else:
6168: (12)                 assert_equal(y.format,
' T{b:a:=h:b:i:c:q:d:q:dx:B:e:@H:f:=I:g:Q:h:Q:hx:f:i:d:j:^g:k:=Zf:ix:Zd:jx:^Zg:kx:4s:l:=4w:m:3x:n?:?
:o:@e:p:}')
6169: (8)             if not (np.ones(1).strides[0] == np.iinfo(np.intp).max):
6170: (12)                 assert_equal(y.strides, (sz,))
6171: (8)                 assert_equal(y.itemsize, sz)
6172: (4)             def test_export_subarray(self):
6173: (8)                 x = np.array(([1, 2], [3, 4]),), dtype=[('a', ('i', (2, 2)))]
6174: (8)                 y = memoryview(x)
6175: (8)                 assert_equal(y.format, 'T{(2,2)i:a:}')
6176: (8)                 assert_equal(y.shape, ())
6177: (8)                 assert_equal(y.ndim, 0)
6178: (8)                 assert_equal(y.strides, ())
6179: (8)                 assert_equal(y.suboffsets, ())
6180: (8)                 assert_equal(y.itemsize, 16)
6181: (4)             def test_export_endian(self):
6182: (8)                 x = np.array([1, 2, 3], dtype='>i')
6183: (8)                 y = memoryview(x)
6184: (8)                 if sys.byteorder == 'little':
6185: (12)                     assert_equal(y.format, '>i')
6186: (8)                 else:
6187: (12)                     assert_equal(y.format, 'i')
6188: (8)                 x = np.array([1, 2, 3], dtype='<i')
6189: (8)                 y = memoryview(x)
6190: (8)                 if sys.byteorder == 'little':
6191: (12)                     assert_equal(y.format, 'i')
6192: (8)                 else:
6193: (12)                     assert_equal(y.format, '<i')
6194: (4)             def test_export_flags(self):
6195: (8)                 assert_raises(ValueError,
6196: (22)                     _multiarray_tests.get_buffer_info,
6197: (23)                     np.arange(5)[::2], ('SIMPLE',))
6198: (4)             @pytest.mark.parametrize(["obj", "error"], [
6199: (12)                 pytest.param(np.array([1, 2], dtype=rational), ValueError,
id="array"),
6200: (12)                 pytest.param(rational(1, 2), TypeError, id="scalar")])
6201: (4)             def test_export_and_pickle_user_dtype(self, obj, error):
6202: (8)                 with pytest.raises(error):
6203: (12)                     _multiarray_tests.get_buffer_info(obj, ("STRIDED_RO", "FORMAT"))
6204: (8)                     _multiarray_tests.get_buffer_info(obj, ("STRIDED_RO",))
6205: (8)                     pickle_obj = pickle.dumps(obj)
6206: (8)                     res = pickle.loads(pickle_obj)
6207: (8)                     assert_array_equal(res, obj)
6208: (4)             def test_padding(self):
6209: (8)                 for j in range(8):
6210: (12)                     x = np.array([(1,), (2,)], dtype={'f0': (int, j)})
6211: (12)                     self._check_roundtrip(x)
6212: (4)             def test_reference_leak(self):
6213: (8)                 if HAS_REFCOUNT:
6214: (12)                     count_1 = sys.getrefcount(np.core._internal)
6215: (8)                     a = np.zeros(4)
6216: (8)                     b = memoryview(a)
6217: (8)                     c = np.asarray(b)
6218: (8)                     if HAS_REFCOUNT:
6219: (12)                         count_2 = sys.getrefcount(np.core._internal)
6220: (12)                         assert_equal(count_1, count_2)
6221: (8)                     del c # avoid pyflakes unused variable warning.
6222: (4)             def test_padded_struct_array(self):
6223: (8)                 dt1 = np.dtype(
6224: (16)                     [('a', 'b'), ('b', 'i'), ('sub', np.dtype('b,i'))], ('c',
'i')],
```

```

6225: (16)           align=True)
6226: (8)            x1 = np.arange(dt1.itemsize, dtype=np.int8).view(dt1)
6227: (8)            self._check_roundtrip(x1)
6228: (8)            dt2 = np.dtype(
6229: (16)              [ ('a', 'b'), ('b', 'i'), ('c', 'b', (3,)), ('d', 'i') ],
6230: (16)              align=True)
6231: (8)            x2 = np.arange(dt2.itemsize, dtype=np.int8).view(dt2)
6232: (8)            self._check_roundtrip(x2)
6233: (8)            dt3 = np.dtype(
6234: (16)              [ ('a', 'b'), ('b', 'i'), ('c', 'b'), ('d', 'b'),
6235: (20)                ('e', 'b'), ('sub', np.dtype('b,i', align=True))])
6236: (8)            x3 = np.arange(dt3.itemsize, dtype=np.int8).view(dt3)
6237: (8)            self._check_roundtrip(x3)
6238: (4) @pytest.mark.valgrind_error(reason="leaks buffer info cache temporarily.")
6239: (4) def test_relaxed_strides(self, c=np.ones((1, 10, 10), dtype='i8')):
6240: (8)     c.strides = (-1, 80, 8) # strides need to be fixed at export
6241: (8)     assert_(memoryview(c).strides == (800, 80, 8))
6242: (8)     fd = io.BytesIO()
6243: (8)     fd.write(c.data)
6244: (8)     fortran = c.T
6245: (8)     assert_(memoryview(fortran).strides == (8, 80, 800))
6246: (8)     arr = np.ones((1, 10))
6247: (8)     if arr.flags.f_contiguous:
6248: (12)         shape, strides = _multiarray_tests.get_buffer_info(
6249: (20)           arr, ['F_CONTIGUOUS'])
6250: (12)         assert_(strides[0] == 8)
6251: (12)         arr = np.ones((10, 1), order='F')
6252: (12)         shape, strides = _multiarray_tests.get_buffer_info(
6253: (20)           arr, ['C_CONTIGUOUS'])
6254: (12)         assert_(strides[-1] == 8)
6255: (4) @pytest.mark.valgrind_error(reason="leaks buffer info cache temporarily.")
6256: (4) @pytest.mark.skipif(not np.ones((10, 1), order="C").flags.f_contiguous,
6257: (12)               reason="Test is unnecessary (but fails) without relaxed strides.")
6258: (4) def test_relaxed_strides_buffer_info_leak(self, arr=np.ones((1, 10))):
6259: (8)     """Test that alternating export of C- and F-order buffers from
6260: (8)     an array which is both C- and F-order when relaxed strides is
6261: (8)     active works.
6262: (8)     This test defines array in the signature to ensure leaking more
6263: (8)     references every time the test is run (catching the leak with
6264: (8)     pytest-leaks).
6265: (8)     """
6266: (8)     for i in range(10):
6267: (12)         _, s = _multiarray_tests.get_buffer_info(arr, ['F_CONTIGUOUS'])
6268: (12)         assert s == (8, 8)
6269: (12)         _, s = _multiarray_tests.get_buffer_info(arr, ['C_CONTIGUOUS'])
6270: (12)         assert s == (80, 8)
6271: (4) def test_out_of_order_fields(self):
6272: (8)     dt = np.dtype(dict(
6273: (12)       formats=['<i4', '<i4'],
6274: (12)       names=['one', 'two'],
6275: (12)       offsets=[4, 0],
6276: (12)       itemsize=8
6277: (8)     ))
6278: (8)     arr = np.empty(1, dt)
6279: (8)     with assert_raises(ValueError):
6280: (12)         memoryview(arr)
6281: (4) def test_max_dims(self):
6282: (8)     a = np.ones(1,) * 32
6283: (8)     self._check_roundtrip(a)
6284: (4) @pytest.mark.slow
6285: (4) def test_error_too_many_dims(self):
6286: (8)     def make_ctype(shape, scalar_type):
6287: (12)         t = scalar_type
6288: (12)         for dim in shape[::-1]:
6289: (16)             t = dim * t
6290: (12)         return t
6291: (8)     c_u8_33d = make_ctype((1,)*33, ctypes.c_uint8)
6292: (8)     m = memoryview(c_u8_33d())
6293: (8)     assert_equal(m.ndim, 33)

```

```

6294: (8)             assert_raises_regex(
6295: (12)             RuntimeError, "ndim",
6296: (12)             np.array, m)
6297: (8)             del c_u8_33d, m
6298: (8)             for i in range(33):
6299: (12)                 if gc.collect() == 0:
6300: (16)                     break
6301: (4)             def test_error_pointer_type(self):
6302: (8)                 m = memoryview(ctypes.pointer(ctypes.c_uint8()))
6303: (8)                 assert_('&' in m.format)
6304: (8)                 assert_raises_regex(
6305: (12)                     ValueError, "format string",
6306: (12)                     np.array, m)
6307: (4)             def test_error_message_unsupported(self):
6308: (8)                 t = ctypes.c_wchar * 4
6309: (8)                 with assert_raises(ValueError) as cm:
6310: (12)                     np.array(t())
6311: (8)                     exc = cm.exception
6312: (8)                     with assert_raises_regex(
6313: (12)                         NotImplementedError,
6314: (12)                         r"Unrepresentable .* 'u' \(\UCS-2 strings\)"
6315: (8)                     ):
6316: (12)                         raise exc.__cause__
6317: (4)             def test_ctypes_integer_via_memoryview(self):
6318: (8)                 for c_integer in {ctypes.c_int, ctypes.c_long, ctypes.c_longlong}:
6319: (12)                     value = c_integer(42)
6320: (12)                     with warnings.catch_warnings(record=True):
6321: (16)                         warnings.filterwarnings('always', r'.*\bctypes\b',
RuntimeWarning)
6322: (16)                         np.asarray(value)
6323: (4)             def test_ctypes_struct_via_memoryview(self):
6324: (8)                 class foo(ctypes.Structure):
6325: (12)                     _fields_ = [('a', ctypes.c_uint8), ('b', ctypes.c_uint32)]
6326: (8)                     f = foo(a=1, b=2)
6327: (8)                     with warnings.catch_warnings(record=True):
6328: (12)                         warnings.filterwarnings('always', r'.*\bctypes\b', RuntimeWarning)
6329: (12)                         arr = np.asarray(f)
6330: (8)                         assert_equal(arr['a'], 1)
6331: (8)                         assert_equal(arr['b'], 2)
6332: (8)                         f.a = 3
6333: (8)                         assert_equal(arr['a'], 3)
6334: (4)             @pytest.mark.parametrize("obj", [np.ones(3), np.ones(1, dtype="i,i")[]])
6335: (4)             def test_error_if_stored_buffer_info_is_corrupted(self, obj):
6336: (8)                 """
6337: (8)                 If a user extends a NumPy array before 1.20 and then runs it
6338: (8)                 on NumPy 1.20+. A C-subclassed array might in theory modify
6339: (8)                 the new buffer-info field. This checks that an error is raised
6340: (8)                 if this happens (for buffer export), an error is written on delete.
6341: (8)                 This is a sanity check to help users transition to safe code, it
6342: (8)                 may be deleted at any point.
6343: (8)                 """
6344: (8)                 _multiarray_tests.corrupt_or_fix_bufferinfo(obj)
6345: (8)                 name = type(obj)
6346: (8)                 with pytest.raises(RuntimeError,
6347: (20)                     match=f".*{name} appears to be C subclassed"):
6348: (12)                     memoryview(obj)
6349: (8)                     _multiarray_tests.corrupt_or_fix_bufferinfo(obj)
6350: (4)             def test_no_suboffsets(self):
6351: (8)                 try:
6352: (12)                     import _testbuffer
6353: (8)                 except ImportError:
6354: (12)                     raise pytest.skip("_testbuffer is not available")
6355: (8)                 for shape in [(2, 3), (2, 3, 4)]:
6356: (12)                     data = list(range(np.prod(shape)))
6357: (12)                     buffer = _testbuffer.ndarray(data, shape, format='i',
6358: (41)                                     flags=_testbuffer.ND_PIL)
6359: (12)                     msg = "NumPy currently does not support.*suboffsets"
6360: (12)                     with pytest.raises(BufferError, match=msg):
6361: (16)                         np.asarray(buffer)

```

```

6362: (12)
6363: (16)
6364: (12)
6365: (16)
6366: (0)
6367: (4)
6368: (8)
6369: (12)
6370: (4)
6371: (4)
6372: (4)
6373: (8)
6374: (12)
6375: (12)
6376: (12)
6377: (12)
6378: (28)
6379: (12)
6380: (28)
6381: (12)
6382: (28)
6383: (12)
6384: (28)
6385: (12)
6386: (16)
6387: (4)
6388: (8)
6389: (12)
6390: (25)
6391: (12)
6392: (16)
6393: (16)
6394: (20)
6395: (8)
6396: (12)
6397: (16)
6398: (16)
6399: (20)
6400: (20)
6401: (20)
6402: (16)
6403: (20)
6404: (24)
6405: (24)
6406: (20)
6407: (35)
6408: (35)
6409: (20)
6410: (16)
6411: (20)
6412: (24)
6413: (24)
6414: (24)
6415: (20)
6416: (34)
6417: (34)
6418: (20)
6419: (34)
6420: (34)
6421: (4)
6422: (8)
6423: (8)
6424: (8)
6425: (12)
6426: (12)
6427: (8)
6428: (12)
6429: (12)
6430: (8)

        with pytest.raises(BufferError, match=msg):
            np.asarray([buffer])
        with pytest.raises(BufferError):
            np.frombuffer(buffer)

    class TestArrayCreationCopyArgument(object):
        class RaiseOnBool:
            def __bool__(self):
                raise ValueError

        true_vals = [True, np._CopyMode.ALWAYS, np.True_]
        false_vals = [False, np._CopyMode.IF_NEEDED, np.False_]

        def test_scalars(self):
            for dtype in np.typecodes["All"]:
                arr = np.zeros((), dtype=dtype)
                scalar = arr[()]
                pyscalar = arr.item(0)
                assert_raises(ValueError, np.array, scalar,
                              copy=np._CopyMode.NEVER)
                assert_raises(ValueError, np.array, pyscalar,
                              copy=np._CopyMode.NEVER)
                assert_raises(ValueError, np.array, pyscalar,
                              copy=self.RaiseOnBool())
                assert_raises(ValueError, _multiarray_tests.npy_ensurencopy,
                              [1])

            with pytest.raises(ValueError):
                np.array(pyscalar, dtype=np.int64, copy=np._CopyMode.NEVER)

        def test_compatible_cast(self):
            def int_types(bytewrap=False):
                int_types = (np.typecodes["Integer"] +
                            np.typecodes["UnsignedInteger"])
                for int_type in int_types:
                    yield np.dtype(int_type)
                    if bytewrap:
                        yield np.dtype(int_type).newbyteorder()

            for int1 in int_types():
                for int2 in int_types(True):
                    arr = np.arange(10, dtype=int1)
                    for copy in self.true_vals:
                        res = np.array(arr, copy=copy, dtype=int2)
                        assert res is not arr and res.flags.owndata
                        assert_array_equal(res, arr)
                    if int1 == int2:
                        for copy in self.false_vals:
                            res = np.array(arr, copy=copy, dtype=int2)
                            assert res is arr or res.base is arr
                            res = np.array(arr,
  copy=np._CopyMode.NEVER,
  dtype=int2)
                            assert res is arr or res.base is arr
                    else:
                        for copy in self.false_vals:
                            res = np.array(arr, copy=copy, dtype=int2)
                            assert res is not arr and res.flags.owndata
                            assert_array_equal(res, arr)
                            assert_raises(ValueError, np.array,
  arr, copy=np._CopyMode.NEVER,
  dtype=int2)
                        assert_raises(ValueError, np.array,
                                      arr, copy=None,
                                      dtype=int2)

            def test_buffer_interface(self):
                arr = np.arange(10)
                view = memoryview(arr)
                for copy in self.true_vals:
                    res = np.array(view, copy=copy)
                    assert not np.may_share_memory(arr, res)
                for copy in self.false_vals:
                    res = np.array(view, copy=copy)
                    assert np.may_share_memory(arr, res)
                res = np.array(view, copy=np._CopyMode.NEVER)

```

```

6431: (8)           assert np.may_share_memory(arr, res)
6432: (4)           def test_array_interfaces(self):
6433: (8)             base_arr = np.arange(10)
6434: (8)             class ArrayLike:
6435: (12)               __array_interface__ = base_arr.__array_interface__
6436: (8)             arr = ArrayLike()
6437: (8)             for copy, val in [(True, None), (np._CopyMode.ALWAYS, None),
6438: (26)                           (False, arr), (np._CopyMode.IF_NEEDED, arr),
6439: (26)                           (np._CopyMode.NEVER, arr)]:
6440: (12)               res = np.array(arr, copy=copy)
6441: (12)               assert res.base is val
6442: (4)           def test_array__(self):
6443: (8)             base_arr = np.arange(10)
6444: (8)             class ArrayLike:
6445: (12)               def __array__(self):
6446: (16)                 return base_arr
6447: (8)             arr = ArrayLike()
6448: (8)             for copy in self.true_vals:
6449: (12)               res = np.array(arr, copy=copy)
6450: (12)               assert_array_equal(res, base_arr)
6451: (12)               assert res is not base_arr
6452: (8)             for copy in self.false_vals:
6453: (12)               res = np.array(arr, copy=False)
6454: (12)               assert_array_equal(res, base_arr)
6455: (12)               assert res is base_arr # numpy trusts the ArrayLike
6456: (8)             with pytest.raises(ValueError):
6457: (12)               np.array(arr, copy=np._CopyMode.NEVER)
6458: (4)           @pytest.mark.parametrize(
6459: (12)             "arr", [np.ones(), np.arange(81).reshape((9, 9))])
6460: (4)           @pytest.mark.parametrize("order1", ["C", "F", None])
6461: (4)           @pytest.mark.parametrize("order2", ["C", "F", "A", "K"])
6462: (4)           def test_order_mismatch(self, arr, order1, order2):
6463: (8)             arr = arr.copy(order1)
6464: (8)             if order1 == "C":
6465: (12)               assert arr.flags.c_contiguous
6466: (8)             elif order1 == "F":
6467: (12)               assert arr.flags.f_contiguous
6468: (8)             elif arr.ndim != 0:
6469: (12)               arr = arr[::-2, ::2]
6470: (12)               assert not arr.flags.frc
6471: (8)             if order2 == "C":
6472: (12)               no_copy_necessary = arr.flags.c_contiguous
6473: (8)             elif order2 == "F":
6474: (12)               no_copy_necessary = arr.flags.f_contiguous
6475: (8)             else:
6476: (12)               no_copy_necessary = True
6477: (8)             for view in [arr, memoryview(arr)]:
6478: (12)               for copy in self.true_vals:
6479: (16)                 res = np.array(view, copy=copy, order=order2)
6480: (16)                 assert res is not arr and res.flags.owndata
6481: (16)                 assert_array_equal(arr, res)
6482: (12)               if no_copy_necessary:
6483: (16)                 for copy in self.false_vals:
6484: (20)                   res = np.array(view, copy=copy, order=order2)
6485: (20)                   if not IS_PYPY:
6486: (24)                     assert res is arr or res.base.obj is arr
6487: (16)                   res = np.array(view, copy=np._CopyMode.NEVER,
6488: (31)                               order=order2)
6489: (16)                   if not IS_PYPY:
6490: (20)                     assert res is arr or res.base.obj is arr
6491: (12)               else:
6492: (16)                 for copy in self.false_vals:
6493: (20)                   res = np.array(arr, copy=copy, order=order2)
6494: (20)                   assert_array_equal(arr, res)
6495: (16)                   assert_raises(ValueError, np.array,
6496: (30)                         view, copy=np._CopyMode.NEVER,
6497: (30)                         order=order2)
6498: (16)                   assert_raises(ValueError, np.array,
6499: (30)                         view, copy=None,

```

```

6500: (30)                                order=order2)
6501: (4)       def test_striding_not_ok(self):
6502: (8)           arr = np.array([[1, 2, 4], [3, 4, 5]])
6503: (8)           assert_raises(ValueError, np.array,
6504: (22)               arr.T, copy=np._CopyMode.NEVER,
6505: (22)               order='C')
6506: (8)           assert_raises(ValueError, np.array,
6507: (22)               arr.T, copy=np._CopyMode.NEVER,
6508: (22)               order='C', dtype=np.int64)
6509: (8)           assert_raises(ValueError, np.array,
6510: (22)               arr, copy=np._CopyMode.NEVER,
6511: (22)               order='F')
6512: (8)           assert_raises(ValueError, np.array,
6513: (22)               arr, copy=np._CopyMode.NEVER,
6514: (22)               order='F', dtype=np.int64)
6515: (0) class TestArrayAttributeDeletion:
6516: (4)     def test_multiarray_writable_attributes_deletion(self):
6517: (8)         a = np.ones(2)
6518: (8)         attr = ['shape', 'strides', 'data', 'dtype', 'real', 'imag', 'flat']
6519: (8)         with suppress_warnings() as sup:
6520: (12)             sup.filter(DeprecationWarning, "Assigning the 'data' attribute")
6521: (12)             for s in attr:
6522: (16)                 assert_raises(AttributeError, delattr, a, s)
6523: (4)     def test_multiarray_not_writable_attributes_deletion(self):
6524: (8)         a = np.ones(2)
6525: (8)         attr = ['ndim', 'flags', 'itemsize', 'size', ' nbytes', 'base',
6526: (16)             'ctypes', 'T', '__array_interface__', '__array_struct__',
6527: (16)             '__array_priority__', '__array_finalize__']
6528: (8)         for s in attr:
6529: (12)             assert_raises(AttributeError, delattr, a, s)
6530: (4)     def test_multiarray_flags_writable_attribute_deletion(self):
6531: (8)         a = np.ones(2).flags
6532: (8)         attr = ['writebackifcopy', 'updateifcopy', 'aligned', 'writeable']
6533: (8)         for s in attr:
6534: (12)             assert_raises(AttributeError, delattr, a, s)
6535: (4)     def test_multiarray_flags_not_writable_attribute_deletion(self):
6536: (8)         a = np.ones(2).flags
6537: (8)         attr = ['contiguous', 'c_contiguous', 'f_contiguous', 'fortran',
6538: (16)             'owndata', 'fnc', 'forc', 'behaved', 'carray', 'farray',
6539: (16)             'num']
6540: (8)         for s in attr:
6541: (12)             assert_raises(AttributeError, delattr, a, s)
6542: (0) class TestArrayInterface():
6543: (4)     class Foo:
6544: (8)         def __init__(self, value):
6545: (12)             self.value = value
6546: (12)             self iface = {'typestr': 'f8'}
6547: (8)         def __float__(self):
6548: (12)             return float(self.value)
6549: (8)         @property
6550: (8)         def __array_interface__(self):
6551: (12)             return self iface
6552: (4)     f = Foo(0.5)
6553: (4)     @pytest.mark.parametrize('val, iface, expected', [
6554: (8)         (f, {}, 0.5),
6555: (8)         ([f], {}, [0.5]),
6556: (8)         ([f, f], {}, [0.5, 0.5]),
6557: (8)         (f, {'shape': ()}, 0.5),
6558: (8)         (f, {'shape': None}, TypeError),
6559: (8)         (f, {'shape': (1, 1)}, [[0.5]]),
6560: (8)         (f, {'shape': (2,)}, ValueError),
6561: (8)         (f, {'strides': ()}, 0.5),
6562: (8)         (f, {'strides': (2,)}, ValueError),
6563: (8)         (f, {'strides': 16}, TypeError),
6564: (8)     ])
6565: (4)     def test_scalar_interface(self, val, iface, expected):
6566: (8)         self.f iface = {'typestr': 'f8'}
6567: (8)         self.f iface.update(iface)
6568: (8)         if HAS_REFCOUNT:

```

```

6569: (12)           pre_cnt = sys.getrefcount(np.dtype('f8'))
6570: (8)            if isinstance(expected, type):
6571: (12)              assert_raises(expected, np.array, val)
6572: (8)            else:
6573: (12)              result = np.array(val)
6574: (12)              assert_equal(np.array(val), expected)
6575: (12)              assert result.dtype == 'f8'
6576: (12)              del result
6577: (8)            if HAS_REFCOUNT:
6578: (12)              post_cnt = sys.getrefcount(np.dtype('f8'))
6579: (12)              assert_equal(pre_cnt, post_cnt)
6580: (0)             def test_interface_no_shape():
6581: (4)               class ArrayLike:
6582: (8)                 array = np.array(1)
6583: (8)                 __array_interface__ = array.__array_interface__
6584: (4)                 assert_equal(np.array(ArrayLike()), 1)
6585: (0)             def test_array_interface_itemsize():
6586: (4)               my_dtype = np.dtype({'names': ['A', 'B'], 'formats': ['f4', 'f4'],
6587: (25)                   'offsets': [0, 8], 'itemsize': 16})
6588: (4)               a = np.ones(10, dtype=my_dtype)
6589: (4)               descr_t = np.dtype(a.__array_interface__['descr'])
6590: (4)               typestr_t = np.dtype(a.__array_interface__['typestr'])
6591: (4)               assert_equal(descr_t.itemsize, typestr_t.itemsize)
6592: (0)             def test_array_interface_empty_shape():
6593: (4)               arr = np.array([1, 2, 3])
6594: (4)               interface1 = dict(arr.__array_interface__)
6595: (4)               interface1['shape'] = ()
6596: (4)               class DummyArray1:
6597: (8)                 __array_interface__ = interface1
6598: (4)               interface2 = dict(interface1)
6599: (4)               interface2['data'] = arr[0].tobytes()
6600: (4)               class DummyArray2:
6601: (8)                 __array_interface__ = interface2
6602: (4)               arr1 = np.asarray(DummyArray1())
6603: (4)               arr2 = np.asarray(DummyArray2())
6604: (4)               arr3 = arr[:1].reshape(())
6605: (4)               assert_equal(arr1, arr2)
6606: (4)               assert_equal(arr1, arr3)
6607: (0)             def test_array_interface_offset():
6608: (4)               arr = np.array([1, 2, 3], dtype='int32')
6609: (4)               interface = dict(arr.__array_interface__)
6610: (4)               interface['data'] = memoryview(arr)
6611: (4)               interface['shape'] = (2,)
6612: (4)               interface['offset'] = 4
6613: (4)               class DummyArray:
6614: (8)                 __array_interface__ = interface
6615: (4)               arr1 = np.asarray(DummyArray())
6616: (4)               assert_equal(arr1, arr[1:])
6617: (0)             def test_array_interface_unicode_typestr():
6618: (4)               arr = np.array([1, 2, 3], dtype='int32')
6619: (4)               interface = dict(arr.__array_interface__)
6620: (4)               interface['typestr'] = '\N{check mark}'
6621: (4)               class DummyArray:
6622: (8)                 __array_interface__ = interface
6623: (4)               with pytest.raises(TypeError):
6624: (8)                 np.asarray(DummyArray())
6625: (0)             def test_flat_element_deletion():
6626: (4)               it = np.ones(3).flat
6627: (4)               try:
6628: (8)                 del it[1]
6629: (8)                 del it[1:2]
6630: (4)               except TypeError:
6631: (8)                 pass
6632: (4)               except Exception:
6633: (8)                 raise AssertionError
6634: (0)             def test_scalar_element_deletion():
6635: (4)               a = np.zeros(2, dtype=[('x', 'int'), ('y', 'int')])
6636: (4)               assert_raises(ValueError, a[0].__delitem__, 'x')
6637: (0)             class TestMapIter:

```

```

6638: (4)
6639: (8)
6640: (8)
6641: (17)
6642: (8)
6643: (8)
6644: (8)
6645: (25)
6646: (25)
6647: (8)
6648: (8)
6649: (8)
6650: (8)
6651: (8)
6652: (0)
6653: (4)
6654: (8)
6655: (8)
6656: (8)
6657: (4)
6658: (8)
6659: (8)
6660: (8)
6661: (4)
6662: (8)
6663: (8)
6664: (8)
6665: (0)
6666: (4)
6667: (8)
6668: (12)
6669: (12)
6670: (12)
6671: (16)
6672: (24)
6673: (16)
6674: (24)
6675: (8)
6676: (12)
6677: (12)
(dt1,))
6678: (12)
6679: (12)
6680: (16)
6681: (24)
6682: (16)
6683: (24)
6684: (16)
6685: (24)
6686: (8)
6687: (12)
6688: (12)
(dt1,))
6689: (12)
6690: (12)
6691: (16)
6692: (24)
6693: (16)
6694: (24)
6695: (16)
6696: (24)
6697: (4)
6698: (8)
6699: (8)
6700: (8)
6701: (8)
6702: (8)
6703: (12)
6704: (16)

def test_mapiter(self):
    a = np.arange(12).reshape((3, 4)).astype(float)
    index = ([1, 1, 2, 0],
              [0, 0, 2, 3])
    vals = [50, 50, 30, 16]
    _multiarray_tests.test_inplace_increment(a, index, vals)
    assert_equal(a, [[0.0, 1., 2.0, 19.],
                    [104., 5., 6.0, 7.0],
                    [8.0, 9., 40., 11.]])
    b = np.arange(6).astype(float)
    index = (np.array([1, 2, 0]),)
    vals = [50, 4, 100.1]
    _multiarray_tests.test_inplace_increment(b, index, vals)
    assert_equal(b, [100.1, 51., 6., 3., 4., 5.])

class TestAsCArray:
    def test_1darray(self):
        array = np.arange(24, dtype=np.double)
        from_c = _multiarray_tests.test_as_c_array(array, 3)
        assert_equal(array[3], from_c)
    def test_2darray(self):
        array = np.arange(24, dtype=np.double).reshape(3, 8)
        from_c = _multiarray_tests.test_as_c_array(array, 2, 4)
        assert_equal(array[2, 4], from_c)
    def test_3darray(self):
        array = np.arange(24, dtype=np.double).reshape(2, 3, 4)
        from_c = _multiarray_tests.test_as_c_array(array, 1, 2, 3)
        assert_equal(array[1, 2, 3], from_c)

class TestConversion:
    def test_array_scalar_relational_operation(self):
        for dt1 in np.typecodes['AllInteger']:
            assert_(1 > np.array(0, dtype=dt1), "type %s failed" % (dt1,))
            assert_(not 1 < np.array(0, dtype=dt1), "type %s failed" % (dt1,))
            for dt2 in np.typecodes['AllInteger']:
                assert_(np.array(1, dtype=dt1) > np.array(0, dtype=dt2),
                        "type %s and %s failed" % (dt1, dt2))
                assert_(not np.array(1, dtype=dt1) < np.array(0, dtype=dt2),
                        "type %s and %s failed" % (dt1, dt2))
        for dt1 in 'BHILQP':
            assert_(-1 < np.array(1, dtype=dt1), "type %s failed" % (dt1,))
            assert_(not -1 > np.array(1, dtype=dt1), "type %s failed" %
(dt1,))
            assert_(-1 != np.array(1, dtype=dt1), "type %s failed" % (dt1,))
            for dt2 in 'bhilqp':
                assert_(np.array(1, dtype=dt1) > np.array(-1, dtype=dt2),
                        "type %s and %s failed" % (dt1, dt2))
                assert_(not np.array(1, dtype=dt1) < np.array(-1, dtype=dt2),
                        "type %s and %s failed" % (dt1, dt2))
                assert_(np.array(1, dtype=dt1) != np.array(-1, dtype=dt2),
                        "type %s and %s failed" % (dt1, dt2))
        for dt1 in 'bhilqp' + np.typecodes['Float']:
            assert_(1 > np.array(-1, dtype=dt1), "type %s failed" % (dt1,))
            assert_(not 1 < np.array(-1, dtype=dt1), "type %s failed" %
(dt1,))
            assert_(-1 == np.array(-1, dtype=dt1), "type %s failed" % (dt1,))
            for dt2 in 'bhilqp' + np.typecodes['Float']:
                assert_(np.array(1, dtype=dt1) > np.array(-1, dtype=dt2),
                        "type %s and %s failed" % (dt1, dt2))
                assert_(not np.array(1, dtype=dt1) < np.array(-1, dtype=dt2),
                        "type %s and %s failed" % (dt1, dt2))
                assert_(np.array(-1, dtype=dt1) == np.array(-1, dtype=dt2),
                        "type %s and %s failed" % (dt1, dt2))

    def test_to_bool_scalar(self):
        assert_equal(bool(np.array([False])), False)
        assert_equal(bool(np.array([True])), True)
        assert_equal(bool(np.array([[42]])), True)
        assert_raises(ValueError, bool, np.array([1, 2]))
        class NotConvertible:
            def __bool__(self):
                raise NotImplementedError

```

```

6705: (8) assert_raises(NotImplementedError, bool, np.array(NotConvertible()))
6706: (8) assert_raises(NotImplementedError, bool, np.array([NotConvertible()]))
6707: (8) if IS_PYSTON:
6708: (12)     pytest.skip("Pyston disables recursion checking")
6709: (8) selfContaining = np.array([None])
6710: (8) selfContaining[0] = selfContaining
6711: (8) Error = RecursionError
6712: (8) assert_raises(Error, bool, selfContaining) # previously stack
overflow
6713: (8) selfContaining[0] = None # resolve circular reference
6714: (4) def test_to_int_scalar(self):
6715: (8)     int_funcs = (int, lambda x: x.__int__())
6716: (8)     for int_func in int_funcs:
6717: (12)         assert_equal(int_func(np.array(0)), 0)
6718: (12)         with assert_warns(DeprecationWarning):
6719: (16)             assert_equal(int_func(np.array([1])), 1)
6720: (12)         with assert_warns(DeprecationWarning):
6721: (16)             assert_equal(int_func(np.array([[42]])), 42)
6722: (12)         assert_raises(TypeError, int_func, np.array([1, 2]))
6723: (12)         assert_equal(4, int_func(np.array('4')))
6724: (12)         assert_equal(5, int_func(np.bytes_(b'5')))
6725: (12)         assert_equal(6, int_func(np.str_('6')))
6726: (12)         if sys.version_info < (3, 11):
6727: (16)             class HasTrunc:
6728: (20)                 def __trunc__(self):
6729: (24)                     return 3
6730: (16)             assert_equal(3, int_func(np.array(HasTrunc())))
6731: (16)             with assert_warns(DeprecationWarning):
6732: (20)                 assert_equal(3, int_func(np.array([HasTrunc()])))
6733: (12)         else:
6734: (16)             pass
6735: (12)         class NotConvertible:
6736: (16)             def __int__(self):
6737: (20)                 raise NotImplementedError
6738: (12)             assert_raises(NotImplementedError,
6739: (16)                 int_func, np.array(NotConvertible()))
6740: (12)             with assert_warns(DeprecationWarning):
6741: (16)                 assert_raises(NotImplementedError,
6742: (20)                     int_func, np.array([NotConvertible()]))
6743: (0) class TestWhere:
6744: (4)     def test_basic(self):
6745: (8)         dts = [bool, np.int16, np.int32, np.int64, np.double, np.complex128,
6746: (15)             np.longdouble, np.clongdouble]
6747: (8)         for dt in dts:
6748: (12)             c = np.ones(53, dtype=bool)
6749: (12)             assert_equal(np.where(c, dt(0), dt(1)), dt(0))
6750: (12)             assert_equal(np.where(~c, dt(0), dt(1)), dt(1))
6751: (12)             assert_equal(np.where(True, dt(0), dt(1)), dt(0))
6752: (12)             assert_equal(np.where(False, dt(0), dt(1)), dt(1))
6753: (12)             d = np.ones_like(c).astype(dt)
6754: (12)             e = np.zeros_like(d)
6755: (12)             r = d.astype(dt)
6756: (12)             c[7] = False
6757: (12)             r[7] = e[7]
6758: (12)             assert_equal(np.where(c, e, e), e)
6759: (12)             assert_equal(np.where(c, d, e), r)
6760: (12)             assert_equal(np.where(c, d, e[0]), r)
6761: (12)             assert_equal(np.where(c, d[0], e), r)
6762: (12)             assert_equal(np.where(c[::2], d[::2], e[::2]), r[::2])
6763: (12)             assert_equal(np.where(c[1::2], d[1::2], e[1::2]), r[1::2])
6764: (12)             assert_equal(np.where(c[::3], d[::3], e[::3]), r[::3])
6765: (12)             assert_equal(np.where(c[1::3], d[1::3], e[1::3]), r[1::3])
6766: (12)             assert_equal(np.where(c[::2], d[::2], e[::2]), r[::2])
6767: (12)             assert_equal(np.where(c[::3], d[::3], e[::3]), r[::3])
6768: (12)             assert_equal(np.where(c[1::3], d[1::3], e[1::3]), r[1::3])
6769: (4)     def test_exotic(self):
6770: (8)         assert_array_equal(np.where(True, None, None), np.array(None))
6771: (8)         m = np.array([], dtype=bool).reshape(0, 3)
6772: (8)         b = np.array([], dtype=np.float64).reshape(0, 3)

```

```

6773: (8) assert_array_equal(np.where(m, 0, b), np.array([]).reshape(0, 3))
6774: (8) d = np.array([-1.34, -0.16, -0.54, -0.31, -0.08, -0.95, 0.000, 0.313,
6775: (22) 0.547, -0.18, 0.876, 0.236, 1.969, 0.310, 0.699, 1.013,
6776: (22) 1.267, 0.229, -1.39, 0.487])
6777: (8) nan = float('NaN')
6778: (8) e = np.array(['5z', '01', nan, 'Wz', nan, nan, 'Xq', 'cs', nan, nan,
6779: (21) 'QN', nan, nan, 'Fd', nan, nan, 'kp', nan, '36', 'i1'],
6780: (21) dtype=object)
6781: (8) m = np.array([0, 0, 1, 0, 1, 1, 0, 0, 1, 1,
6782: (22) 0, 1, 1, 0, 1, 1, 0, 1, 0, 0], dtype=bool)
6783: (8) r = e[:]
6784: (8) r[np.where(m)] = d[np.where(m)]
6785: (8) assert_array_equal(np.where(m, d, e), r)
6786: (8) r = e[:]
6787: (8) r[np.where(~m)] = d[np.where(~m)]
6788: (8) assert_array_equal(np.where(m, e, d), r)
6789: (8) assert_array_equal(np.where(m, e, e), e)
6790: (8) d = np.array([1., 2.], dtype=np.float32)
6791: (8) e = float('NaN')
6792: (8) assert_equal(np.where(True, d, e).dtype, np.float32)
6793: (8) e = float('Infinity')
6794: (8) assert_equal(np.where(True, d, e).dtype, np.float32)
6795: (8) e = float('-Infinity')
6796: (8) assert_equal(np.where(True, d, e).dtype, np.float32)
6797: (8) e = float(1e150)
6798: (8) assert_equal(np.where(True, d, e).dtype, np.float64)
6799: (4) def test_ndim(self):
6800: (8) c = [True, False]
6801: (8) a = np.zeros((2, 25))
6802: (8) b = np.ones((2, 25))
6803: (8) r = np.where(np.array(c)[:, np.newaxis], a, b)
6804: (8) assert_array_equal(r[0], a[0])
6805: (8) assert_array_equal(r[1], b[0])
6806: (8) a = a.T
6807: (8) b = b.T
6808: (8) r = np.where(c, a, b)
6809: (8) assert_array_equal(r[:, 0], a[:, 0])
6810: (8) assert_array_equal(r[:, 1], b[:, 0])
6811: (4) def test_dtype_mix(self):
6812: (8) c = np.array([False, True, False, False, False, False, True, False,
6813: (21) False, False, True, False])
6814: (8) a = np.uint32(1)
6815: (8) b = np.array([5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.],
6816: (22) dtype=np.float64)
6817: (8) r = np.array([5., 1., 3., 2., -1., -4., 1., -10., 10., 1., 1., 3.],
6818: (21) dtype=np.float64)
6819: (8) assert_equal(np.where(c, a, b), r)
6820: (8) a = a.astype(np.float32)
6821: (8) b = b.astype(np.int64)
6822: (8) assert_equal(np.where(c, a, b), r)
6823: (8) c = c.astype(int)
6824: (8) c[c != 0] = 34242324
6825: (8) assert_equal(np.where(c, a, b), r)
6826: (8) tmpmask = c != 0
6827: (8) c[c == 0] = 41247212
6828: (8) c[tmpmask] = 0
6829: (8) assert_equal(np.where(c, b, a), r)
6830: (4) def test_foreign(self):
6831: (8) c = np.array([False, True, False, False, False, False, True, False,
6832: (21) False, False, True, False])
6833: (8) r = np.array([5., 1., 3., 2., -1., -4., 1., -10., 10., 1., 1., 3.],
6834: (21) dtype=np.float64)
6835: (8) a = np.ones(1, dtype='>i4')
6836: (8) b = np.array([5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.],
6837: (21) dtype=np.float64)
6838: (8) assert_equal(np.where(c, a, b), r)
6839: (8) b = b.astype('>f8')
6840: (8) assert_equal(np.where(c, a, b), r)
6841: (8) a = a.astype('<i4')

```

```

6842: (8)             assert_equal(np.where(c, a, b), r)
6843: (8)             c = c.astype('>i4')
6844: (8)             assert_equal(np.where(c, a, b), r)
6845: (4)             def test_error(self):
6846: (8)                 c = [True, True]
6847: (8)                 a = np.ones((4, 5))
6848: (8)                 b = np.ones((5, 5))
6849: (8)                 assert_raises(ValueError, np.where, c, a, a)
6850: (8)                 assert_raises(ValueError, np.where, c[0], a, b)
6851: (4)             def test_string(self):
6852: (8)                 a = np.array("abc")
6853: (8)                 b = np.array("x" * 753)
6854: (8)                 assert_equal(np.where(True, a, b), "abc")
6855: (8)                 assert_equal(np.where(False, b, a), "abc")
6856: (8)                 a = np.array("abcd")
6857: (8)                 b = np.array("x" * 8)
6858: (8)                 assert_equal(np.where(True, a, b), "abcd")
6859: (8)                 assert_equal(np.where(False, b, a), "abcd")
6860: (4)             def test_empty_result(self):
6861: (8)                 x = np.zeros((1, 1))
6862: (8)                 ibad = np.vstack(np.where(x == 99.))
6863: (8)                 assert_array_equal(ibad,
6864: (27)                               np.atleast_2d(np.array([[[], []]], dtype=np.intp)))
6865: (4)             def test_largedim(self):
6866: (8)                 shape = [10, 2, 3, 4, 5, 6]
6867: (8)                 np.random.seed(2)
6868: (8)                 array = np.random.rand(*shape)
6869: (8)                 for i in range(10):
6870: (12)                     benchmark = array.nonzero()
6871: (12)                     result = array.nonzero()
6872: (12)                     assert_array_equal(benchmark, result)
6873: (4)             def test_kwargs(self):
6874: (8)                 a = np.zeros(1)
6875: (8)                 with assert_raises(TypeError):
6876: (12)                     np.where(a, x=a, y=a)
6877: (0)             if not IS_PYPY:
6878: (4)                 class TestSizeOf:
6879: (8)                     def test_empty_array(self):
6880: (12)                         x = np.array([])
6881: (12)                         assert_(sys.getsizeof(x) > 0)
6882: (8)                     def check_array(self, dtype):
6883: (12)                         elem_size = dtype(0).itemsize
6884: (12)                         for length in [10, 50, 100, 500]:
6885: (16)                             x = np.arange(length, dtype=dtype)
6886: (16)                             assert_(sys.getsizeof(x) > length * elem_size)
6887: (8)                     def test_array_int32(self):
6888: (12)                         self.check_array(np.int32)
6889: (8)                     def test_array_int64(self):
6890: (12)                         self.check_array(np.int64)
6891: (8)                     def test_array_float32(self):
6892: (12)                         self.check_array(np.float32)
6893: (8)                     def test_array_float64(self):
6894: (12)                         self.check_array(np.float64)
6895: (8)                     def test_view(self):
6896: (12)                         d = np.ones(100)
6897: (12)                         assert_(sys.getsizeof(d[...]) < sys.getsizeof(d))
6898: (8)                     def test_reshape(self):
6899: (12)                         d = np.ones(100)
6900: (12)                         assert_(sys.getsizeof(d) < sys.getsizeof(d.reshape(100, 1,
1).copy()))
6901: (8)             @_no_tracing
6902: (8)             def test_resize(self):
6903: (12)                 d = np.ones(100)
6904: (12)                 old = sys.getsizeof(d)
6905: (12)                 d.resize(50)
6906: (12)                 assert_(old > sys.getsizeof(d))
6907: (12)                 d.resize(150)
6908: (12)                 assert_(old < sys.getsizeof(d))
6909: (8)             def test_error(self):

```

```

6910: (12)             d = np.ones(100)
6911: (12)             assert_raises(TypeError, d.__sizeof__, "a")
6912: (0)              class TestHashing:
6913: (4)                def test_arrays_not_hashable(self):
6914: (8)                  x = np.ones(3)
6915: (8)                  assert_raises(TypeError, hash, x)
6916: (4)                def test_collections_hashable(self):
6917: (8)                  x = np.array([])
6918: (8)                  assert_(not isinstance(x, collections.abc.Hashable))
6919: (0)              class TestArrayPriority:
6920: (4)                op = operator
6921: (4)                binary_ops = [
6922: (8)                  op.pow, op.add, op.sub, op.mul, op.floordiv, op.truediv, op.mod,
6923: (8)                  op.and_, op.or_, op.xor, op.lshift, op.rshift, op.mod, op.gt,
6924: (8)                  op.ge, op.lt, op.le, op.ne, op.eq
6925: (8)                ]
6926: (4)                class Foo(np.ndarray):
6927: (8)                  __array_priority__ = 100.
6928: (8)                  def __new__(cls, *args, **kwargs):
6929: (12)                      return np.array(*args, **kwargs).view(cls)
6930: (4)              class Bar(np.ndarray):
6931: (8)                  __array_priority__ = 101.
6932: (8)                  def __new__(cls, *args, **kwargs):
6933: (12)                      return np.array(*args, **kwargs).view(cls)
6934: (4)              class Other:
6935: (8)                  __array_priority__ = 1000.
6936: (8)                  def __all__(self, other):
6937: (12)                      return self.__class__()
6938: (8)                  __add__ = __radd__ = __all
6939: (8)                  __sub__ = __rsub__ = __all
6940: (8)                  __mul__ = __rmul__ = __all
6941: (8)                  __pow__ = __rpow__ = __all
6942: (8)                  __div__ = __rdiv__ = __all
6943: (8)                  __mod__ = __rmod__ = __all
6944: (8)                  __truediv__ = __rtruediv__ = __all
6945: (8)                  __floordiv__ = __rfloordiv__ = __all
6946: (8)                  __and__ = __rand__ = __all
6947: (8)                  __xor__ = __rxor__ = __all
6948: (8)                  __or__ = __ror__ = __all
6949: (8)                  __lshift__ = __rlshift__ = __all
6950: (8)                  __rshift__ = __rrshift__ = __all
6951: (8)                  __eq__ = __all
6952: (8)                  __ne__ = __all
6953: (8)                  __gt__ = __all
6954: (8)                  __ge__ = __all
6955: (8)                  __lt__ = __all
6956: (8)                  __le__ = __all
6957: (4)                  def test_ndarray_subclass(self):
6958: (8)                      a = np.array([1, 2])
6959: (8)                      b = self.Bar([1, 2])
6960: (8)                      for f in self.binary_ops:
6961: (12)                          msg = repr(f)
6962: (12)                          assert_(isinstance(f(a, b), self.Bar), msg)
6963: (12)                          assert_(isinstance(f(b, a), self.Bar), msg)
6964: (4)                  def test_ndarray_other(self):
6965: (8)                      a = np.array([1, 2])
6966: (8)                      b = self.Other()
6967: (8)                      for f in self.binary_ops:
6968: (12)                          msg = repr(f)
6969: (12)                          assert_(isinstance(f(a, b), self.Other), msg)
6970: (12)                          assert_(isinstance(f(b, a), self.Other), msg)
6971: (4)                  def test_subclass_subclass(self):
6972: (8)                      a = self.Foo([1, 2])
6973: (8)                      b = self.Bar([1, 2])
6974: (8)                      for f in self.binary_ops:
6975: (12)                          msg = repr(f)
6976: (12)                          assert_(isinstance(f(a, b), self.Bar), msg)
6977: (12)                          assert_(isinstance(f(b, a), self.Bar), msg)
6978: (4)                  def test_subclass_other(self):

```

```

6979: (8)             a = self.Foo([1, 2])
6980: (8)             b = self.Other()
6981: (8)             for f in self.binary_ops:
6982: (12)                 msg = repr(f)
6983: (12)                 assert_(isinstance(f(a, b), self.Other), msg)
6984: (12)                 assert_(isinstance(f(b, a), self.Other), msg)
6985: (0)             class TestByteStringArrayNonzero:
6986: (4)                 def test_empty_bstring_array_is_falsey(self):
6987: (8)                     assert_(not np.array([''], dtype=str))
6988: (4)                 def test_whitespace_bstring_array_is_falsey(self):
6989: (8)                     a = np.array(['spam'], dtype=str)
6990: (8)                     a[0] = '\0\0'
6991: (8)                     assert_(not a)
6992: (4)                 def test_all_null_bstring_array_is_falsey(self):
6993: (8)                     a = np.array(['spam'], dtype=str)
6994: (8)                     a[0] = '\0\0\0\0'
6995: (8)                     assert_(not a)
6996: (4)                 def test_null_inside_bstring_array_is_truthy(self):
6997: (8)                     a = np.array(['spam'], dtype=str)
6998: (8)                     a[0] = '\0 \0'
6999: (8)                     assert_(a)
7000: (0)             class TestUnicodeEncoding:
7001: (4)                 """
7002: (4)             Tests for encoding related bugs, such as UCS2 vs UCS4, round-tripping
7003: (4)             issues, etc
7004: (4)                 """
7005: (4)             def test_round_trip(self):
7006: (8)                 """ Tests that GETITEM, SETITEM, and PyArray_Scalar roundtrip """
7007: (8)                 arr = np.zeros(shape=(), dtype="U1")
7008: (8)                 for i in range(1, sys.maxunicode + 1):
7009: (12)                     expected = chr(i)
7010: (12)                     arr[()] = expected
7011: (12)                     assert arr[()] == expected
7012: (12)                     assert arr.item() == expected
7013: (4)             def test_assign_scalar(self):
7014: (8)                 l = np.array(['aa', 'bb'])
7015: (8)                 l[:] = np.str_('cc')
7016: (8)                 assert_equal(l, ['cc', 'cc'])
7017: (4)             def test_fill_scalar(self):
7018: (8)                 l = np.array(['aa', 'bb'])
7019: (8)                 l.fill(np.str_('cc'))
7020: (8)                 assert_equal(l, ['cc', 'cc'])
7021: (0)             class TestUnicodeArrayNonzero:
7022: (4)                 def test_empty_ustring_array_is_falsey(self):
7023: (8)                     assert_(not np.array([''], dtype=np.str_))
7024: (4)                 def test_whitespace_ustring_array_is_falsey(self):
7025: (8)                     a = np.array(['eggs'], dtype=np.str_)
7026: (8)                     a[0] = '\0\0'
7027: (8)                     assert_(not a)
7028: (4)                 def test_all_null_ustring_array_is_falsey(self):
7029: (8)                     a = np.array(['eggs'], dtype=np.str_)
7030: (8)                     a[0] = '\0\0\0\0'
7031: (8)                     assert_(not a)
7032: (4)                 def test_null_inside_ustring_array_is_truthy(self):
7033: (8)                     a = np.array(['eggs'], dtype=np.str_)
7034: (8)                     a[0] = '\0 \0'
7035: (8)                     assert_(a)
7036: (0)             class TestFormat:
7037: (4)                 def test_0d(self):
7038: (8)                     a = np.array(np.pi)
7039: (8)                     assert_equal('{:0.3g}'.format(a), '3.14')
7040: (8)                     assert_equal('{:0.3g}'.format(a[()]), '3.14')
7041: (4)                 def test_1d_no_format(self):
7042: (8)                     a = np.array([np.pi])
7043: (8)                     assert_equal('{}'.format(a), str(a))
7044: (4)                 def test_1d_format(self):
7045: (8)                     a = np.array([np.pi])
7046: (8)                     assert_raises(TypeError, '{:30}'.format, a)
7047: (0)             from numpy.testing import IS_PYPY

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

7048: (0)
7049: (4)
7050: (8)
7051: (8)
7052: (8)
7053: (4)
7054: (8)
7055: (8)
7056: (8)
7057: (12)
7058: (12)
7059: (31)
7060: (12)
7061: (8)
7062: (12)
7063: (4)
7064: (8)
7065: (8)
7066: (4)
7067: (8)
7068: (8)
7069: (8)
7070: (8)
7071: (12)
7072: (16)
7073: (16)
7074: (16)
7075: (16)
7076: (12)
7077: (8)
7078: (8)
7079: (8)
7080: (8)
7081: (8)
7082: (4)
7083: (8)
7084: (8)
7085: (8)
7086: (8)
7087: (8)
7088: (8)
7089: (8)
7090: (8)
7091: (4)
7092: (4)
7093: (8)
7094: (8)
7095: (8)
7096: (8)
7097: (8)
7098: (8)
reference")
7099: (8)
7100: (8)
7101: (12)
7102: (8)
reference")
7103: (4)
7104: (8)
7105: (8)
7106: (8)
7107: (8)
7108: (8)
7109: (8)
reference")
7110: (8)
7111: (8)
7112: (12)
7113: (8)

class TestCTypes:
    def test_ctypes_is_available(self):
        test_arr = np.array([[1, 2, 3], [4, 5, 6]])
        assert_equal(ctypes, test_arr.ctypes._ctypes)
        assert_equal(tuple(test_arr.ctypes.shape), (2, 3))
    def test_ctypes_is_not_available(self):
        from numpy.core import _internal
        _internal.ctypes = None
        try:
            test_arr = np.array([[1, 2, 3], [4, 5, 6]])
            assert_(isinstance(test_arr.ctypes._ctypes,
                               _internal._missing_ctypes))
            assert_equal(tuple(test_arr.ctypes.shape), (2, 3))
        finally:
            _internal.ctypes = ctypes
    def _make_READONLY(x):
        x.flags.writeable = False
        return x
    @pytest.mark.parametrize('arr', [
        np.array([1, 2, 3]),
        np.array(['one', 'two'], ['three', 'four']),
        np.array((1, 2), dtype='i4,i4'),
        np.zeros((2,), dtype=
            np.dtype(dict(
                formats=['<i4', '<i4'],
                names=['a', 'b'],
                offsets=[0, 2],
                itemsize=6
            )))
        ],
        np.array([None], dtype=object),
        np.array([]),
        np.empty((0, 0)),
        _make_READONLY(np.array([1, 2, 3])),
    ], ids=[
        '1d',
        '2d',
        'structured',
        'overlapping',
        'object',
        'empty',
        'empty-2d',
        'readonly'
    ])
    def test_ctypes_data_as_holds_reference(self, arr):
        arr = arr.copy()
        arr_ref = weakref.ref(arr)
        ctypes_ptr = arr.ctypes.data_as(ctypes.c_void_p)
        del arr
        break_cycles()
        assert_(arr_ref() is not None, "ctypes pointer did not hold onto a
reference")
        del ctypes_ptr
        if IS_PYPY:
            break_cycles()
        assert_(arr_ref() is None, "unknowable whether ctypes pointer holds a
reference")
    def test_ctypes_as_parameter_holds_reference(self):
        arr = np.array([None]).copy()
        arr_ref = weakref.ref(arr)
        ctypes_ptr = arr.ctypes._as_parameter_
        del arr
        break_cycles()
        assert_(arr_ref() is not None, "ctypes pointer did not hold onto a
reference")
        del ctypes_ptr
        if IS_PYPY:
            break_cycles()
        assert_(arr_ref() is None, "unknowable whether ctypes pointer holds a
reference")

```

```

reference")
7114: (0)
7115: (4)
7116: (8)
7117: (8)
7118: (8)
7119: (8)
7120: (4)
7121: (8)
7122: (8)
7123: (8)
7124: (8)
7125: (4)
7126: (8)
7127: (8)
7128: (8)
7129: (8)
7130: (4)
7131: (8)
7132: (8)
7133: (8)
7134: (4)
7135: (8)
7136: (8)
7137: (8)
7138: (4)
7139: (8)
7140: (8)
7141: (8)
7142: (8)
7143: (4)
7144: (8)
7145: (8)
7146: (8)
7147: (8)
7148: (8)
7149: (36)
7150: (36)
7151: (4)
7152: (8)
7153: (8)
7154: (8)
7155: (8)
7156: (4)
7157: (8)
7158: (8)
7159: (8)
7160: (4)
7161: (8)
npy_resolve
7162: (8)
7163: (8)
7164: (8)
7165: (8)
7166: (8)
7167: (8)
7168: (8)
7169: (8)
7170: (8)
7171: (8)
7172: (8)
7173: (4)
7174: (12)
path.")
7175: (4)
7176: (8)
7177: (12)
7178: (12)
7179: (12)

class TestWritebackIfCopy:
    def test_argmax_with_out(self):
        mat = np.eye(5)
        out = np.empty(5, dtype='i2')
        res = np.argmax(mat, 0, out=out)
        assert_equal(res, range(5))
    def test_argmin_with_out(self):
        mat = -np.eye(5)
        out = np.empty(5, dtype='i2')
        res = np.argmin(mat, 0, out=out)
        assert_equal(res, range(5))
    def test_insert_noncontiguous(self):
        a = np.arange(6).reshape(2,3).T # force non-c-contiguous
        np.place(a, a>2, [44, 55])
        assert_equal(a, np.array([[0, 44], [1, 55], [2, 44]]))
        assert_raises(ValueError, np.place, a, a>20, [])
    def test_put_noncontiguous(self):
        a = np.arange(6).reshape(2,3).T # force non-c-contiguous
        np.put(a, [0, 2], [44, 55])
        assert_equal(a, np.array([[44, 3], [55, 4], [2, 5]]))
    def test_putmask_noncontiguous(self):
        a = np.arange(6).reshape(2,3).T # force non-c-contiguous
        np.putmask(a, a>2, a**2)
        assert_equal(a, np.array([[0, 9], [1, 16], [2, 25]]))
    def test_take_mode_raise(self):
        a = np.arange(6, dtype='int')
        out = np.empty(2, dtype='int')
        np.take(a, [0, 2], out=out, mode='raise')
        assert_equal(out, np.array([0, 2]))
    def test_choose_mod_raise(self):
        a = np.array([[1, 0, 1], [0, 1, 0], [1, 0, 1]])
        out = np.empty((3,3), dtype='int')
        choices = [-10, 10]
        np.choose(a, choices, out=out, mode='raise')
        assert_equal(out, np.array([[ 10, -10,  10],
                                   [-10,  10, -10],
                                   [ 10, -10,  10]]))
    def test_flatiter_array_(self):
        a = np.arange(9).reshape(3,3)
        b = a.T.flat
        c = b.__array__()
        del c
    def test_dot_out(self):
        a = np.arange(9, dtype=float).reshape(3,3)
        b = np.dot(a, a, out=a)
        assert_equal(b, np.array([[15, 18, 21], [42, 54, 66], [69, 90, 111]]))
    def test_view_assign(self):
        from numpy.core._multiarray_tests import npy_create_writebackifcopy,
arr = np.arange(9).reshape(3, 3).T
arr_wb = npy_create_writebackifcopy(arr)
assert_(arr_wb.flags.writebackifcopy)
assert_(arr_wb.base is arr)
arr_wb[...] = -100
npy_resolve(arr_wb)
assert_equal(arr, -100)
assert_(arr_wb.ctypes.data != 0)
assert_equal(arr_wb.base, None)
arr_wb[...] = 100
assert_equal(arr, -100)
@pytest.mark.leaks_references(
    reason="increments self in dealloc; ignore since deprecated
def test_dealloc_warning(self):
    with suppress_warnings() as sup:
        sup.record(RuntimeWarning)
        arr = np.arange(9).reshape(3, 3)
        v = arr.T

```

```

7180: (12)             _multiarray_tests.npy_abuse_writebackifcopy(v)
7181: (12)             assert len(sup.log) == 1
7182: (4)              def test_view_discard_refcount(self):
7183: (8)                from numpy.core._multiarray_tests import npy_create_writebackifcopy,
7184: (8)                arr = np.arange(9).reshape(3, 3).T
7185: (8)                orig = arr.copy()
7186: (8)                if HAS_REFCOUNT:
7187: (12)                  arr_cnt = sys.getrefcount(arr)
7188: (8)                  arr_wb = npy_create_writebackifcopy(arr)
7189: (8)                  assert_(arr_wb.flags.writebackifcopy)
7190: (8)                  assert_(arr_wb.base is arr)
7191: (8)                  arr_wb[...] = -100
7192: (8)                  npy_discard(arr_wb)
7193: (8)                  assert_equal(arr, orig)
7194: (8)                  assert_(arr_wb.ctypes.data != 0)
7195: (8)                  assert_equal(arr_wb.base, None)
7196: (8)                  if HAS_REFCOUNT:
7197: (12)                    assert_equal(arr_cnt, sys.getrefcount(arr))
7198: (8)                    arr_wb[...] = 100
7199: (8)                    assert_equal(arr, orig)
7200: (0)              class TestArange:
7201: (4)                def test_infinite(self):
7202: (8)                  assert_raises_regex(
7203: (12)                      ValueError, "size exceeded",
7204: (12)                      np.arange, 0, np.inf
7205: (8)
7206: (4)
7207: (8)
7208: (12)
7209: (12)
7210: (8)
7211: (4)
7212: (8)
7213: (8)
7214: (8)
7215: (8)
7216: (4)
7217: (8)
7218: (8)
7219: (8)
7220: (8)
7221: (4)
7222: (8)
7223: (8)
7224: (8)
7225: (8)
7226: (8)
7227: (8)
7228: (8)
7229: (4)
7230: (8)
7231: (8)
7232: (8)
7233: (8)
7234: (8)
7235: (8)
7236: (8)
7237: (8)
7238: (8)
7239: (12)
7240: (4)
7241: (4)
7242: (8)
7243: (8)
7244: (8)
7245: (16)
{DType_name}"):
7246: (12)              np.arange(2, dtype=dtype)

@ pytest.mark.parametrize("dtype", ["S3", "U", "5i"])
def test_rejects_bad_dtypes(self, dtype):
    dtype = np.dtype(dtype)
    DType_name = re.escape(str(type(dtype)))
    with pytest.raises(TypeError,
                       match=rf"arange\(\) not supported for inputs .*
{DType_name}"):
        np.arange(2, dtype=dtype)

```

```

7247: (4)             def test_rejects_strings(self):
7248: (8)               DType_name = re.escape(str(type(np.array("a").dtype)))
7249: (8)               with pytest.raises(TypeError,
7250: (16)                 match=rf"arange\(\) not supported for inputs .*"
7251: (12)                   np.arange("a", "b"))
7252: (4)             def test_byteswapped(self):
7253: (8)               res_be = np.arange(1, 1000, dtype=">i4")
7254: (8)               res_le = np.arange(1, 1000, dtype="<i4")
7255: (8)               assert res_be.dtype == ">i4"
7256: (8)               assert res_le.dtype == "<i4"
7257: (8)               assert_array_equal(res_le, res_be)
7258: (4)             @pytest.mark.parametrize("which", [0, 1, 2])
7259: (4)             def test_error_paths_and_promotion(self, which):
7260: (8)               args = [0, 1, 2] # start, stop, and step
7261: (8)               args[which] = np.float64(2.) # should ensure float64 output
7262: (8)               assert np.arange(*args).dtype == np.float64
7263: (8)               args[which] = [None, []]
7264: (8)               with pytest.raises(ValueError):
7265: (12)                 np.arange(*args)
7266: (0)             class TestArrayFinalize:
7267: (4)               """ Tests __array_finalize__ """
7268: (4)               def test_receives_base(self):
7269: (8)                 class SavesBase(np.ndarray):
7270: (12)                   def __array_finalize__(self, obj):
7271: (16)                     self.saved_base = self.base
7272: (8)                     a = np.array(1).view(SavesBase)
7273: (8)                     assert_(a.saved_base is a.base)
7274: (4)               def test_bad_finalize1(self):
7275: (8)                 class BadAttributeArray(np.ndarray):
7276: (12)                   @property
7277: (12)                     def __array_finalize__(self):
7278: (16)                       raise RuntimeError("boohoo!")
7279: (8)                     with pytest.raises(TypeError, match="not callable"):
7280: (12)                       np.arange(10).view(BadAttributeArray)
7281: (4)               def test_bad_finalize2(self):
7282: (8)                 class BadAttributeArray(np.ndarray):
7283: (12)                   def __array_finalize__(self):
7284: (16)                     raise RuntimeError("boohoo!")
7285: (8)                     with pytest.raises(TypeError, match="takes 1 positional"):
7286: (12)                       np.arange(10).view(BadAttributeArray)
7287: (4)               def test_bad_finalize3(self):
7288: (8)                 class BadAttributeArray(np.ndarray):
7289: (12)                   def __array_finalize__(self, obj):
7290: (16)                     raise RuntimeError("boohoo!")
7291: (8)                     with pytest.raises(RuntimeError, match="boohoo!"):
7292: (12)                       np.arange(10).view(BadAttributeArray)
7293: (4)               def test_lifetime_on_error(self):
7294: (8)                 class RaisesInFinalize(np.ndarray):
7295: (12)                   def __array_finalize__(self, obj):
7296: (16)                     raise Exception(self)
7297: (8)                 class Dummy: pass
7298: (8)                 obj_arr = np.array(Dummy())
7299: (8)                 obj_ref = weakref.ref(obj_arr[()])
7300: (8)                 with assert_raises(Exception) as e:
7301: (12)                   obj_arr.view(RaisesInFinalize)
7302: (8)                   obj_subarray = e.exception.args[0]
7303: (8)                   del e
7304: (8)                   assert_(isinstance(obj_subarray, RaisesInFinalize))
7305: (8)                   break_cycles()
7306: (8)                   assert_(obj_ref() is not None, "object should not already be dead")
7307: (8)                   del obj_arr
7308: (8)                   break_cycles()
7309: (8)                   assert_(obj_ref() is not None, "obj_arr should not hold the last
reference")
7310: (8)                   del obj_subarray
7311: (8)                   break_cycles()
7312: (8)                   assert_(obj_ref() is None, "no references should remain")
7313: (4)             def test_can_use_super(self):

```

```

7314: (8)
7315: (12)
7316: (16)
7317: (8)
7318: (8)
7319: (0)
7320: (4)
7321: (4)
7322: (0)
7323: (4)
7324: (8)
7325: (12)
7326: (8)
7327: (12)
7328: (4)
7329: (8)
7330: (4)
7331: (8)
7332: (4)
7333: (4)
7334: (8)
7335: (8)
7336: (8)
7337: (8)
7338: (8)
7339: (0)
7340: (0)
7341: (8)
7342: (8)
7343: (8)
7344: (8)
7345: (0)
7346: (4)
7347: (8)
7348: (8)
7349: (12)
7350: (12)
7351: (4)
7352: (4)
7353: (4)
7354: (4)
7355: (4)
7356: (0)
7357: (8)
7358: (8)
7359: (8)
7360: (8)
7361: (8)
7362: (0)
7363: (0)
7364: (4)
7365: (4)
7366: (4)
7367: (4)
7368: (4)
7369: (4)
7370: (4)
7371: (4)
7372: (4)
7373: (4)
7374: (4)
7375: (4)
7376: (8)
7377: (4)
7378: (8)
7379: (4)
7380: (8)
7381: (0)
7382: (8)

    class SuperFinalize(np.ndarray):
        def __array_finalize__(self, obj):
            self.saved_result = super().__array_finalize__(obj)
            a = np.array(1).view(SuperFinalize)
            assert_(a.saved_result is None)

    def test_orderconverter_with_nonASCII_unicode_ordering():
        a = np.arange(5)
        assert_raises(ValueError, a.flatten, order='\xe2')

    def test_equal_override():
        class MyAlwaysEqual:
            def __eq__(self, other):
                return "eq"
            def __ne__(self, other):
                return "ne"

        class MyAlwaysEqualOld(MyAlwaysEqual):
            __array_priority__ = 10000

        class MyAlwaysEqualNew(MyAlwaysEqual):
            __array_ufunc__ = None

        array = np.array([(0, 1), (2, 3)], dtype='i4,i4')
        for my_always_equal_cls in MyAlwaysEqualOld, MyAlwaysEqualNew:
            my_always_equal = my_always_equal_cls()
            assert_equal(my_always_equal == array, 'eq')
            assert_equal(array == my_always_equal, 'eq')
            assert_equal(my_always_equal != array, 'ne')
            assert_equal(array != my_always_equal, 'ne')

    @pytest.mark.parametrize("op", [operator.eq, operator.ne])
    @pytest.mark.parametrize(["dt1", "dt2"], [
        ([("f", "i")], [("f", "i")]), # structured comparison (successful)
        ("M8", "d"), # impossible comparison: result is all True or False
        ("d", "d"), # valid comparison
    ])
    def test_equal_subclass_no_override(op, dt1, dt2):
        class MyArr(np.ndarray):
            called_wrap = 0
            def __array_wrap__(self, new):
                type(self).called_wrap += 1
                return super().__array_wrap__(new)

        numpy_arr = np.zeros(5, dtype=dt1)
        my_arr = np.zeros(5, dtype=dt2).view(MyArr)
        assert type(op(numpy_arr, my_arr)) is MyArr
        assert type(op(my_arr, numpy_arr)) is MyArr
        assert MyArr.called_wrap == 2

    @pytest.mark.parametrize(["dt1", "dt2"], [
        ("M8[ns]", "d"),
        ("M8[s]", "l"),
        ("m8[ns]", "d"),
        ("M8[s]", "m8[s]"),
        ("S5", "U5"),
    ])
    def test_no_loop_gives_all_true_or_false(dt1, dt2):
        arr1 = np.random.randint(5, size=100).astype(dt1)
        arr2 = np.random.randint(5, size=99)[:, np.newaxis].astype(dt2)
        res = arr1 == arr2
        assert res.shape == (99, 100)
        assert res.dtype == bool
        assert not res.any()
        res = arr1 != arr2
        assert res.shape == (99, 100)
        assert res.dtype == bool
        assert res.all()
        arr2 = np.random.randint(5, size=99).astype(dt2)
        with pytest.raises(ValueError):
            arr1 == arr2
        with pytest.raises(ValueError):
            arr1 != arr2
        with pytest.raises(np.core._exceptions._UFuncNoLoopError):
            arr1 > arr2

    @pytest.mark.parametrize("op", [
        operator.eq, operator.ne, operator.le, operator.lt, operator.ge,
    ])

```

```

7383: (8)             operator.gt])
7384: (0)         def test_comparisons_forwards_error(op):
7385: (4)             class NotArray:
7386: (8)                 def __array__(self):
7387: (12)                     raise TypeError("run you fools")
7388: (4)             with pytest.raises(TypeError, match="run you fools"):
7389: (8)                 op(np.arange(2), NotArray())
7390: (4)             with pytest.raises(TypeError, match="run you fools"):
7391: (8)                 op(NotArray(), np.arange(2))
7392: (0)         def test_richcompare_scalar_boolean_singleton_return():
7393: (4)             assert (np.array(0) == "a") is False
7394: (4)             assert (np.array(0) != "a") is True
7395: (4)             assert (np.int16(0) == "a") is False
7396: (4)             assert (np.int16(0) != "a") is True
7397: (0)         @pytest.mark.parametrize("op", [
7398: (8)             operator.eq, operator.ne, operator.le, operator.lt, operator.ge,
7399: (8)             operator.gt])
7400: (0)         def test_ragged_comparison_fails(op):
7401: (4)             a = np.array([1, np.array([1, 2, 3])], dtype=object)
7402: (4)             b = np.array([1, np.array([1, 2, 3])], dtype=object)
7403: (4)             with pytest.raises(ValueError, match="The truth value.*ambiguous"):
7404: (8)                 op(a, b)
7405: (0)         @pytest.mark.parametrize(
7406: (4)             ["fun", "npyfun"],
7407: (4)             [
7408: (8)                 (_multiarray_tests.npy_cabs, np.absolute),
7409: (8)                 (_multiarray_tests.npy_carg, np.angle)
7410: (4)             ])
7411: (0)
7412: (0)         @pytest.mark.parametrize("x", [1, np.inf, -np.inf, np.nan])
7413: (0)         @pytest.mark.parametrize("y", [1, np.inf, -np.inf, np.nan])
7414: (0)         @pytest.mark.parametrize("test_dtype", np.complexfloating.__subclasses__())
7415: (0)         def test_npymath_complex(fun, npfun, x, y, test_dtype):
7416: (4)             z = test_dtype(complex(x, y))
7417: (4)             with np.errstate(invalid='ignore'):
7418: (8)                 got = fun(z)
7419: (8)                 expected = npfun(z)
7420: (8)                 assert_allclose(got, expected)
7421: (0)         def test_npymath_real():
7422: (4)             from numpy.core._multiarray_tests import (
7423: (8)                 npy_log10, npy_cosh, npy_sinh, npy_tan, npy_tanh)
7424: (4)             funcs = {npy_log10: np.log10,
7425: (13)                 npy_cosh: np.cosh,
7426: (13)                 npy_sinh: np.sinh,
7427: (13)                 npy_tan: np.tan,
7428: (13)                 npy_tanh: np.tanh}
7429: (4)             vals = (1, np.inf, -np.inf, np.nan)
7430: (4)             types = (np.float32, np.float64, np.longdouble)
7431: (4)             with np.errstate(all='ignore'):
7432: (8)                 for fun, npfun in funcs.items():
7433: (12)                     for x, t in itertools.product(vals, types):
7434: (16)                         z = t(x)
7435: (16)                         got = fun(z)
7436: (16)                         expected = npfun(z)
7437: (16)                         assert_allclose(got, expected)
7438: (0)         def test_uintalignment_and_alignment():
7439: (4)             d1 = np.dtype('u1,c8', align=True)
7440: (4)             d2 = np.dtype('u4,c8', align=True)
7441: (4)             d3 = np.dtype({'names': ['a', 'b'], 'formats': ['u1', d1]}, align=True)
7442: (4)             assert_equal(np.zeros(1, dtype=d1)['f1'].flags['ALIGNED'], True)
7443: (4)             assert_equal(np.zeros(1, dtype=d2)['f1'].flags['ALIGNED'], True)
7444: (4)             assert_equal(np.zeros(1, dtype='u1,c8')['f1'].flags['ALIGNED'], False)
7445: (4)             s = _multiarray_tests.get_struct_alignments()
7446: (4)             for d, (alignment, size) in zip([d1, d2, d3], s):
7447: (8)                 assert_equal(d.alignment, alignment)
7448: (8)                 assert_equal(d.itemsize, size)
7449: (4)             src = np.zeros((2,2), dtype=d1)['f1'] # 4-byte aligned, often
7450: (4)             np.exp(src) # assert fails?
7451: (4)             dst = np.zeros((2,2), dtype='c8')

```

```

7452: (4)           dst[:,1] = src[:,1] # assert in lowlevel_strided_loops fails?
7453: (0)
7454: (4)
7455: (8)
7456: (8)
7457: (8)
7458: (12)
7459: (8)
7460: (8)
7461: (12)
7462: (8)
7463: (12)
7464: (8)
7465: (8)
7466: (12)
7467: (8)
7468: (12)
7469: (16)
7470: (8)
7471: (12)
7472: (8)
7473: (12)
7474: (4)
7475: (8)
7476: (12)
7477: (16)
7478: (20)
7479: (24)
7480: (28)
7481: (24)
7482: (28)
7483: (4)
7484: (8)
7485: (12)
7486: (12)
7487: (12)
7488: (12)
7489: (16)
7490: (16)
7491: (12)
7492: (12)
7493: (12)
7494: (12)
7495: (12)
7496: (12)
7497: (12)
7498: (12)
7499: (12)
7500: (12)
7501: (12)
7502: (12)
7503: (0)
7504: (4)
7505: (4)
7506: (8)
7507: (8)
7508: (4)
7509: (8)
7510: (8)
7511: (4)
7512: (4)
7513: (4)
7514: (4)
7515: (4)
7516: (4)
7517: (4)
7518: (0)
7519: (4)
7520: (4)

    class TestAlignment:
        def check(self, shape, dtype, order, align):
            err_msg = repr((shape, dtype, order, align))
            x = _aligned_zeros(shape, dtype, order, align=align)
            if align is None:
                align = np.dtype(dtype).alignment
            assert_equal(x.__array_interface__['data'][0] % align, 0)
            if hasattr(shape, '__len__'):
                assert_equal(x.shape, shape, err_msg)
            else:
                assert_equal(x.shape, (shape,), err_msg)
            assert_equal(x.dtype, dtype)
            if order == "C":
                assert_(x.flags.c_contiguous, err_msg)
            elif order == "F":
                if x.size > 0:
                    assert_(x.flags.f_contiguous, err_msg)
            elif order is None:
                assert_(x.flags.c_contiguous, err_msg)
            else:
                raise ValueError()
        def test_various_alignments(self):
            for align in [1, 2, 3, 4, 8, 12, 16, 32, 64, None]:
                for n in [0, 1, 3, 11]:
                    for order in ["C", "F", None]:
                        for dtype in list(np.typecodes["All"]) + ['i4,i4,i4']:
                            if dtype == 'O':
                                continue
                            for shape in [n, (1, 2, 3, n)]:
                                self.check(shape, np.dtype(dtype), order, align)
        def test_strided_loop_alignments(self):
            for align in [1, 2, 4, 8, 12, 16, None]:
                xf64 = _aligned_zeros(3, np.float64)
                xc64 = _aligned_zeros(3, np.complex64, align=align)
                xf128 = _aligned_zeros(3, np.longdouble, align=align)
                with suppress_warnings() as sup:
                    sup.filter(np.ComplexWarning, "Casting complex values")
                    xc64.astype('f8')
                    xf64.astype(np.complex64)
                    test = xc64 + xf64
                    xf128.astype('f8')
                    xf64.astype(np.longdouble)
                    test = xf128 + xf64
                    test = xf128 + xc64
                    xf64[:] = xf64.copy()
                    xc64[:] = xc64.copy()
                    xf128[:] = xf128.copy()
                    xf64[::2] = xf64[::2].copy()
                    xc64[::2] = xc64[::2].copy()
                    xf128[::2] = xf128[::2].copy()
        def test_getfield():
            a = np.arange(32, dtype='uint16')
            if sys.byteorder == 'little':
                i = 0
                j = 1
            else:
                i = 1
                j = 0
            b = a.getfield('int8', i)
            assert_equal(b, a)
            b = a.getfield('int8', j)
            assert_equal(b, 0)
            pytest.raises(ValueError, a.getfield, 'uint8', -1)
            pytest.raises(ValueError, a.getfield, 'uint8', 16)
            pytest.raises(ValueError, a.getfield, 'uint64', 0)
        class TestViewDtype:
            """
            Verify that making a view of a non-contiguous array works as expected.

```

```

7521: (4)
7522: (4)
7523: (8)
7524: (8)
7525: (27)
7526: (12)
7527: (8)
7528: (8)
7529: (4)
7530: (8)
7531: (8)
7532: (27)
7533: (12)
7534: (8)
7535: (27)
7536: (12)
7537: (8)
7538: (8)
7539: (4)
7540: (8)
7541: (8)
7542: (29)
7543: (8)
7544: (4)
7545: (8)
7546: (8)
7547: (27)
7548: (12)
7549: (8)
7550: (20)
7551: (8)
7552: (4)
7553: (8)
7554: (8)
7555: (27)
7556: (12)
7557: (4)
7558: (8)
7559: (20)
7560: (8)
7561: (20)
7562: (20)
7563: (8)
7564: (0)
7565: (0)
7566: (0)
7567: (0)
7568: (4)
7569: (4)
7570: (4)
7571: (4)
7572: (4)
7573: (4)
7574: (4)
7575: (4)
7576: (4)
7577: (4)
7578: (17)
7579: (4)
7580: (4)
7581: (4)
7582: (0)
7583: (4)
7584: (4)
7585: (4)
7586: (4)
7587: (4)
7588: (4)
7589: (12)

    """
    def test_smaller_dtype_multiple(self):
        x = np.arange(10, dtype='<i4')[::2]
        with pytest.raises(ValueError,
                           match='the last axis must be contiguous'):
            x.view('<i2')
        expected = [[0, 0], [2, 0], [4, 0], [6, 0], [8, 0]]
        assert_array_equal(x[:, np.newaxis].view('<i2'), expected)
    def test_smaller_dtype_not_multiple(self):
        x = np.arange(5, dtype='<i4')[::2]
        with pytest.raises(ValueError,
                           match='the last axis must be contiguous'):
            x.view('S3')
        with pytest.raises(ValueError,
                           match='When changing to a smaller dtype'):
            x[:, np.newaxis].view('S3')
        expected = [[b'', b'\x02', b'\x04']]
        assert_array_equal(x[:, np.newaxis].view('S4'), expected)
    def test_larger_dtype_multiple(self):
        x = np.arange(20, dtype='<i2').reshape(10, 2)[::2, :]
        expected = np.array([[65536, 327684, 589832],
                            [851980, 1114128]], dtype='<i4')
        assert_array_equal(x.view('<i4'), expected)
    def test_larger_dtype_not_multiple(self):
        x = np.arange(20, dtype='<i2').reshape(10, 2)[::2, :]
        with pytest.raises(ValueError,
                           match='When changing to a larger dtype'):
            x.view('S3')
        expected = [[b'\x00\x00\x01'], [b'\x04\x00\x05'], [b'\x08\x00\t'],
                    [b'\x0c\x00\r'], [b'\x10\x00\x11']]
        assert_array_equal(x.view('S4'), expected)
    def test_f_contiguous(self):
        x = np.arange(4 * 3, dtype='<i4').reshape(4, 3).T
        with pytest.raises(ValueError,
                           match='the last axis must be contiguous'):
            x.view('<i2')
    def test_non_c_contiguous(self):
        x = np.arange(2 * 3 * 4, dtype='i1').\
            reshape(2, 3, 4).transpose(1, 0, 2)
        expected = [[[256, 770], [3340, 3854]],
                    [[1284, 1798], [4368, 4882]],
                    [[2312, 2826], [5396, 5910]]]
        assert_array_equal(x.view('<i2'), expected)
    @pytest.mark.xfail(_SUPPORTS_SVE, reason="gh-22982")
    @pytest.mark.parametrize("N", np.arange(1, 512))
    @pytest.mark.parametrize("dtype", ['e', 'f', 'd'])
    def test_sort_float(N, dtype):
        np.random.seed(42)
        arr = -0.5 + np.random.sample(N).astype(dtype)
        arr[np.random.choice(arr.shape[0], 3)] = np.nan
        assert_equal(np.sort(arr, kind='quick'), np.sort(arr, kind='heap'))
        infarr = np.inf*np.ones(N, dtype=dtype)
        infarr[np.random.choice(infarr.shape[0], 5)] = -1.0
        assert_equal(np.sort(infarr, kind='quick'), np.sort(infarr, kind='heap'))
        neginfarr = -np.inf*np.ones(N, dtype=dtype)
        neginfarr[np.random.choice(neginfarr.shape[0], 5)] = 1.0
        assert_equal(np.sort(neginfarr, kind='quick'),
                    np.sort(neginfarr, kind='heap'))
        infarr = np.inf*np.ones(N, dtype=dtype)
        infarr[np.random.choice(infarr.shape[0], (int)(N/2))] = -np.inf
        assert_equal(np.sort(infarr, kind='quick'), np.sort(infarr, kind='heap'))
    def test_sort_float16():
        arr = np.arange(65536, dtype=np.int16)
        temp = np.frombuffer(arr.tobytes(), dtype=np.float16)
        data = np.copy(temp)
        np.random.shuffle(data)
        data_backup = data
        assert_equal(np.sort(data, kind='quick'),
                    np.sort(data_backup, kind='heap'))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

7590: (0) @pytest.mark.parametrize("N", np.arange(1, 512))
7591: (0) @pytest.mark.parametrize("dtype", ['h', 'H', 'i', 'I', 'l', 'L'])
7592: (0) def test_sort_int(N, dtype):
7593: (4)     minv = np.iinfo(dtype).min
7594: (4)     maxv = np.iinfo(dtype).max
7595: (4)     arr = np.random.randint(low=minv, high=maxv-1, size=N, dtype=dtype)
7596: (4)     arr[np.random.choice(arr.shape[0], 10)] = minv
7597: (4)     arr[np.random.choice(arr.shape[0], 10)] = maxv
7598: (4)     assert_equal(np.sort(arr, kind='quick'), np.sort(arr, kind='heap'))
7599: (0) def test_sort_uint():
7600: (4)     rng = np.random.default_rng(42)
7601: (4)     N = 2047
7602: (4)     maxv = np.iinfo(np.uint32).max
7603: (4)     arr = rng.integers(low=0, high=maxv, size=N).astype('uint32')
7604: (4)     arr[np.random.choice(arr.shape[0], 10)] = maxv
7605: (4)     assert_equal(np.sort(arr, kind='quick'), np.sort(arr, kind='heap'))
7606: (0) def test_private_get_ndarray_c_version():
7607: (4)     assert isinstance(_get_ndarray_c_version(), int)
7608: (0) @pytest.mark.parametrize("N", np.arange(1, 512))
7609: (0) @pytest.mark.parametrize("dtype", [np.float32, np.float64])
7610: (0) def test_argsort_float(N, dtype):
7611: (4)     rnd = np.random.RandomState(116112)
7612: (4)     arr = -0.5 + rnd.random(N).astype(dtype)
7613: (4)     arr[rnd.choice(arr.shape[0], 3)] = np.nan
7614: (4)     assert_arg_sorted(arr, np.argsort(arr, kind='quick'))
7615: (4)     arr = -0.5 + rnd.rand(N).astype(dtype)
7616: (4)     arr[N-1] = np.inf
7617: (4)     assert_arg_sorted(arr, np.argsort(arr, kind='quick'))
7618: (0) @pytest.mark.parametrize("N", np.arange(2, 512))
7619: (0) @pytest.mark.parametrize("dtype", [np.int32, np.uint32, np.int64, np.uint64])
7620: (0) def test_argsort_int(N, dtype):
7621: (4)     rnd = np.random.RandomState(1100710816)
7622: (4)     minv = np.iinfo(dtype).min
7623: (4)     maxv = np.iinfo(dtype).max
7624: (4)     arr = rnd.randint(low=minv, high=maxv, size=N, dtype=dtype)
7625: (4)     i, j = rnd.choice(N, 2, replace=False)
7626: (4)     arr[i] = minv
7627: (4)     arr[j] = maxv
7628: (4)     assert_arg_sorted(arr, np.argsort(arr, kind='quick'))
7629: (4)     arr = rnd.randint(low=minv, high=maxv, size=N, dtype=dtype)
7630: (4)     arr[N-1] = maxv
7631: (4)     assert_arg_sorted(arr, np.argsort(arr, kind='quick'))
7632: (0) @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
7633: (0) def test_gh_22683():
7634: (4)     b = 777.68760986
7635: (4)     a = np.array([b] * 10000, dtype=object)
7636: (4)     refc_start = sys.getrefcount(b)
7637: (4)     np.choose(np.zeros(10000, dtype=int), [a], out=a)
7638: (4)     np.choose(np.zeros(10000, dtype=int), [a], out=a)
7639: (4)     refc_end = sys.getrefcount(b)
7640: (4)     assert refc_end - refc_start < 10
7641: (0) def test_gh_24459():
7642: (4)     a = np.zeros((50, 3), dtype=np.float64)
7643: (4)     with pytest.raises(TypeError):
7644: (8)         np.choose(a, [3, -1])

```

---

File 109 - test\_nditer.py:

```

1: (0)         import sys
2: (0)         import pytest
3: (0)         import textwrap
4: (0)         import subprocess
5: (0)         import numpy as np
6: (0)         import numpy.core._multiarray_tests as _multiarray_tests
7: (0)         from numpy import array, arange, nditer, all
8: (0)         from numpy.testing import (
9: (4)             assert_, assert_equal, assert_array_equal, assert_raises,

```

```

10: (4)           IS_WASM, HAS_REFCOUNT, suppress_warnings, break_cycles
11: (4)
12: (0)
13: (4)
14: (4)
15: (8)
16: (8)
17: (4)
18: (0)
19: (4)
20: (4)
21: (8)
22: (8)
23: (4)
24: (0)
25: (4)
26: (4)
27: (8)
28: (8)
29: (4)
30: (0)          @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
31: (0)
32: (4)          def test_iter_refcount():
33: (4)            a = arange(6)
34: (4)            dt = np.dtype('f4').newbyteorder()
35: (4)            rc_a = sys.getrefcount(a)
36: (4)            rc_dt = sys.getrefcount(dt)
37: (16)           with nditer(a, [],
38: (16)             [[['readwrite', 'updateifcopy']]],
39: (16)             casting='unsafe',
40: (8)             op_dtypes=[dt]) as it:
41: (8)               assert_(not it.iterationneedsapi)
42: (8)               assert_(sys.getrefcount(a) > rc_a)
43: (4)               assert_(sys.getrefcount(dt) > rc_dt)
44: (4)               it = None
45: (4)               assert_equal(sys.getrefcount(a), rc_a)
46: (4)               assert_equal(sys.getrefcount(dt), rc_dt)
47: (4)               a = arange(6, dtype='f4')
48: (4)               dt = np.dtype('f4')
49: (4)               rc_a = sys.getrefcount(a)
50: (4)               rc_dt = sys.getrefcount(dt)
51: (4)               it = nditer(a, [],
52: (16)                 [[['readwrite']]],
53: (16)                 op_dtypes=[dt])
54: (4)               rc2_a = sys.getrefcount(a)
55: (4)               rc2_dt = sys.getrefcount(dt)
56: (4)               it2 = it.copy()
57: (4)               assert_(sys.getrefcount(a) > rc2_a)
58: (4)               assert_(sys.getrefcount(dt) > rc2_dt)
59: (4)               it = None
60: (4)               assert_equal(sys.getrefcount(a), rc2_a)
61: (4)               assert_equal(sys.getrefcount(dt), rc2_dt)
62: (4)               it2 = None
63: (4)               assert_equal(sys.getrefcount(a), rc_a)
64: (4)               assert_equal(sys.getrefcount(dt), rc_dt)
65: (0)               del it2 # avoid pyflakes unused variable warning
66: (4)
67: (8)          def test_iter_best_order():
68: (8)            for shape in [(5,), (3, 4), (2, 3, 4), (2, 3, 4, 3), (2, 3, 2, 2, 3)]:
69: (12)              a = arange(np.prod(shape))
70: (12)              for dirs in range(2**len(shape)):
71: (16)                dirs_index = [slice(None)]*len(shape)
72: (20)                for bit in range(len(shape)):
73: (12)                  if ((2**bit) & dirs):
74: (12)                    dirs_index[bit] = slice(None, None, -1)
75: (12)                    dirs_index = tuple(dirs_index)
76: (12)                    aview = a.reshape(shape)[dirs_index]
77: (12)                    i = nditer(aview, [], [[['readonly']]])
78: (12)                    assert_equal([x for x in i], a)
79: (12)                    i = nditer(aview.T, [], [[['readonly']]])
80: (12)                    assert_equal([x for x in i], a)

```

```

79: (12)             if len(shape) > 2:
80: (16)                 i = nditer(aview.swapaxes(0, 1), [], [['readonly']])
81: (16)                     assert_equal([x for x in i], a)
82: (0)             def test_iter_c_order():
83: (4)                 for shape in [(5,), (3, 4), (2, 3, 4), (2, 3, 4, 3), (2, 3, 2, 2, 3)]:
84: (8)                     a = arange(np.prod(shape))
85: (8)                     for dirs in range(2**len(shape)):
86: (12)                         dirs_index = [slice(None)]*len(shape)
87: (12)                         for bit in range(len(shape)):
88: (16)                             if ((2**bit) & dirs):
89: (20)                                 dirs_index[bit] = slice(None, None, -1)
90: (12)                         dirs_index = tuple(dirs_index)
91: (12)                         aview = a.reshape(shape)[dirs_index]
92: (12)                         i = nditer(aview, order='C')
93: (12)                         assert_equal([x for x in i], aview.ravel(order='C'))
94: (12)                         i = nditer(aview.T, order='C')
95: (12)                         assert_equal([x for x in i], aview.T.ravel(order='C'))
96: (12)                         if len(shape) > 2:
97: (16)                             i = nditer(aview.swapaxes(0, 1), order='C')
98: (16)                             assert_equal([x for x in i],
99: (36)   aview.swapaxes(0, 1).ravel(order='C'))
100: (0)             def test_iter_f_order():
101: (4)                 for shape in [(5,), (3, 4), (2, 3, 4), (2, 3, 4, 3), (2, 3, 2, 2, 3)]:
102: (8)                     a = arange(np.prod(shape))
103: (8)                     for dirs in range(2**len(shape)):
104: (12)                         dirs_index = [slice(None)]*len(shape)
105: (12)                         for bit in range(len(shape)):
106: (16)                             if ((2**bit) & dirs):
107: (20)                                 dirs_index[bit] = slice(None, None, -1)
108: (12)                         dirs_index = tuple(dirs_index)
109: (12)                         aview = a.reshape(shape)[dirs_index]
110: (12)                         i = nditer(aview, order='F')
111: (12)                         assert_equal([x for x in i], aview.ravel(order='F'))
112: (12)                         i = nditer(aview.T, order='F')
113: (12)                         assert_equal([x for x in i], aview.T.ravel(order='F'))
114: (12)                         if len(shape) > 2:
115: (16)                             i = nditer(aview.swapaxes(0, 1), order='F')
116: (16)                             assert_equal([x for x in i],
117: (36)   aview.swapaxes(0, 1).ravel(order='F'))
118: (0)             def test_iter_c_or_f_order():
119: (4)                 for shape in [(5,), (3, 4), (2, 3, 4), (2, 3, 4, 3), (2, 3, 2, 2, 3)]:
120: (8)                     a = arange(np.prod(shape))
121: (8)                     for dirs in range(2**len(shape)):
122: (12)                         dirs_index = [slice(None)]*len(shape)
123: (12)                         for bit in range(len(shape)):
124: (16)                             if ((2**bit) & dirs):
125: (20)                                 dirs_index[bit] = slice(None, None, -1)
126: (12)                         dirs_index = tuple(dirs_index)
127: (12)                         aview = a.reshape(shape)[dirs_index]
128: (12)                         i = nditer(aview, order='A')
129: (12)                         assert_equal([x for x in i], aview.ravel(order='A'))
130: (12)                         i = nditer(aview.T, order='A')
131: (12)                         assert_equal([x for x in i], aview.T.ravel(order='A'))
132: (12)                         if len(shape) > 2:
133: (16)                             i = nditer(aview.swapaxes(0, 1), order='A')
134: (16)                             assert_equal([x for x in i],
135: (36)   aview.swapaxes(0, 1).ravel(order='A'))
136: (0)             def test_nditer_multi_index_set():
137: (4)                 a = np.arange(6).reshape(2, 3)
138: (4)                 it = np.nditer(a, flags=['multi_index'])
139: (4)                 it.multi_index = (0, 2,)
140: (4)                 assert_equal([i for i in it], [2, 3, 4, 5])
141: (0)             @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
142: (0)             def test_nditer_multi_index_set_refcount():
143: (4)                 index = 0
144: (4)                 i = np.nditer(np.array([111, 222, 333, 444]), flags=['multi_index'])
145: (4)                 start_count = sys.getrefcount(index)
146: (4)                 i.multi_index = (index,)
147: (4)                 end_count = sys.getrefcount(index)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

148: (4)
149: (0)
150: (4)
151: (4)
152: (4)
153: (4)
154: (4)
155: (0)
156: (4)
157: (4)
158: (4)
(1, 2))
159: (4)
[['readonly']])
160: (4)
(1, 2))
161: (4)
162: (4)
(0, 2))
163: (4)
164: (4)
(1, 0))
165: (4)
166: (4)
(0, 0))
167: (4)
[['readonly']])
168: (4)
(0, 2))
169: (4)
170: (51)
[['readonly']])
171: (4)
(1, 0))
172: (4)
173: (51)
[['readonly']])
174: (4)
(0, 0))
175: (0)
176: (4)
177: (4)
178: (4)
179: (28)
2, 0), (0, 2, 1),
180: (29)
2, 0), (1, 2, 1)])
181: (4)
[['readonly']])
182: (4)
183: (28)
2, 0), (1, 2, 0),
184: (29)
2, 1), (1, 2, 1)])
185: (4)
186: (4)
187: (28)
2, 0), (1, 2, 1),
188: (29)
2, 0), (0, 2, 1)])
189: (4)
190: (4)
191: (28)
0, 0), (0, 0, 1),
192: (29)
0, 0), (1, 0, 1)])
193: (4)
194: (4)
195: (28)

    assert_equal(start_count, end_count)
def test_iter_best_order_multi_index_1d():
    a = arange(4)
    i = nditer(a, ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i), [(0,), (1,), (2,), (3,)])
    i = nditer(a[::-1], ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i), [(3,), (2,), (1,), (0,)])
def test_iter_best_order_multi_index_2d():
    a = arange(6)
    i = nditer(a.reshape(2, 3), ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i), [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1),
(1, 2)])
    i = nditer(a.reshape(2, 3).copy(order='F'), ['multi_index'],
assert_equal(iter_multi_index(i), [(0, 0), (1, 0), (0, 1), (1, 1), (0, 2),
(1, 2)])
    i = nditer(a.reshape(2, 3)[::-1], ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i), [(1, 0), (1, 1), (1, 2), (0, 0), (0, 1),
(0, 2)])
    i = nditer(a.reshape(2, 3)[::, ::-1], ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i), [(0, 2), (0, 1), (0, 0), (1, 2), (1, 1),
(0, 0)])
    i = nditer(a.reshape(2, 3)[::-1, ::-1], ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i), [(1, 2), (1, 1), (1, 0), (0, 2), (0, 1),
(0, 0)])
    i = nditer(a.reshape(2, 3).copy(order='F')[::-1], ['multi_index'],
assert_equal(iter_multi_index(i), [(1, 0), (0, 0), (1, 1), (0, 1), (1, 2),
(0, 2)])
    i = nditer(a.reshape(2, 3).copy(order='F')[::, ::-1],
['multi_index'],
assert_equal(iter_multi_index(i), [(0, 2), (1, 2), (0, 1), (1, 1), (0, 0),
(1, 0)])
    i = nditer(a.reshape(2, 3).copy(order='F')[::-1, ::-1],
['multi_index'],
assert_equal(iter_multi_index(i), [(1, 2), (0, 2), (1, 1), (0, 1), (1, 0),
(0, 0)])
def test_iter_best_order_multi_index_3d():
    a = arange(12)
    i = nditer(a.reshape(2, 3, 2), ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i),
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (0,
(1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1), (1,
i = nditer(a.reshape(2, 3, 2).copy(order='F'), ['multi_index'],
assert_equal(iter_multi_index(i),
[(0, 0, 0), (1, 0, 0), (0, 1, 0), (1, 1, 0), (0,
(0, 0, 1), (1, 0, 1), (0, 1, 1), (1, 1, 1), (0,
i = nditer(a.reshape(2, 3, 2)[::-1], ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i),
[(1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1), (1,
(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (0,
i = nditer(a.reshape(2, 3, 2)[::, ::-1], ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i),
[(0, 2, 0), (0, 2, 1), (0, 1, 0), (0, 1, 1), (0,
(1, 2, 0), (1, 2, 1), (1, 1, 0), (1, 1, 1), (1,
i = nditer(a.reshape(2, 3, 2)[:::, ::-1], ['multi_index'], [['readonly']])
    assert_equal(iter_multi_index(i),
[(0, 0, 1), (0, 0, 0), (0, 1, 1), (0, 1, 0), (0,

```

```

2, 1), (0, 2, 0),
196: (29)
2, 1), (1, 2, 0)])
197: (4)
198: (52)
[['readonly']])
199: (4)
200: (28)
2, 0), (0, 2, 0),
201: (29)
2, 1), (0, 2, 1)])
202: (4)
203: (52)
[['readonly']])
204: (4)
205: (28)
0, 0), (1, 0, 0),
206: (29)
0, 1), (1, 0, 1)])
207: (4)
208: (52)
[['readonly']])
209: (4)
210: (28)
2, 1), (1, 2, 1),
211: (29)
2, 0), (1, 2, 0)])
212: (0)
def test_iter_best_order_c_index_1d():
    a = arange(4)
    i = nditer(a, ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [0, 1, 2, 3])
    i = nditer(a[::-1], ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [3, 2, 1, 0])
def test_iter_best_order_c_index_2d():
    a = arange(6)
    i = nditer(a.reshape(2, 3), ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [0, 1, 2, 3, 4, 5])
    i = nditer(a.reshape(2, 3).copy(order='F'),
               ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [0, 3, 1, 4, 2, 5])
    i = nditer(a.reshape(2, 3)[::-1], ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [3, 4, 5, 0, 1, 2])
    i = nditer(a.reshape(2, 3)[:, ::-1], ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [2, 1, 0, 5, 4, 3])
    i = nditer(a.reshape(2, 3)[::-1, ::-1], ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [5, 4, 3, 2, 1, 0])
    i = nditer(a.reshape(2, 3).copy(order='F')[::-1],
               ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [3, 0, 4, 1, 5, 2])
    i = nditer(a.reshape(2, 3).copy(order='F')[:, ::-1],
               ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [2, 5, 1, 4, 0, 3])
    i = nditer(a.reshape(2, 3).copy(order='F')[::-1, ::-1],
               ['c_index'], [['readonly']])
    assert_equal(iter_indices(i), [5, 2, 4, 1, 3, 0])
def test_iter_best_order_c_index_3d():
    a = arange(12)
    i = nditer(a.reshape(2, 3, 2), ['c_index'], [['readonly']])
    assert_equal(iter_indices(i),
                 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
    i = nditer(a.reshape(2, 3, 2).copy(order='F'),
               ['c_index'], [['readonly']])
    assert_equal(iter_indices(i),
                 [0, 6, 2, 8, 4, 10, 1, 7, 3, 9, 5, 11])
    i = nditer(a.reshape(2, 3, 2)[::-1], ['c_index'], [['readonly']])
    assert_equal(iter_indices(i),
                 [6, 7, 8, 9, 10, 11, 0, 1, 2, 3, 4, 5])
    i = nditer(a.reshape(2, 3, 2)[:, ::-1], ['c_index'], [['readonly']])
    assert_equal(iter_indices(i),

```

```

254: (28) [4, 5, 2, 3, 0, 1, 10, 11, 8, 9, 6, 7])
255: (4) i = nditer(a.reshape(2, 3, 2)[:::, ::-1], ['c_index'], [['readonly']])
256: (4) assert_equal(iter_indices(i),
257: (28) [1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10])
258: (4) i = nditer(a.reshape(2, 3, 2).copy(order='F')[::-1],
259: (36) ['c_index'], [['readonly']])
260: (4) assert_equal(iter_indices(i),
261: (28) [6, 0, 8, 2, 10, 4, 7, 1, 9, 3, 11, 5])
262: (4) i = nditer(a.reshape(2, 3, 2).copy(order='F')[:, ::-1],
263: (36) ['c_index'], [['readonly']])
264: (4) assert_equal(iter_indices(i),
265: (28) [4, 10, 2, 8, 0, 6, 5, 11, 3, 9, 1, 7])
266: (4) i = nditer(a.reshape(2, 3, 2).copy(order='F')[:::, ::-1],
267: (36) ['c_index'], [['readonly']])
268: (4) assert_equal(iter_indices(i),
269: (28) [1, 7, 3, 9, 5, 11, 0, 6, 2, 8, 4, 10])
270: (0) def test_iter_best_order_f_index_1d():
271: (4) a = arange(4)
272: (4) i = nditer(a, ['f_index'], [['readonly']])
273: (4) assert_equal(iter_indices(i), [0, 1, 2, 3])
274: (4) i = nditer(a[::-1], ['f_index'], [['readonly']])
275: (4) assert_equal(iter_indices(i), [3, 2, 1, 0])
276: (0) def test_iter_best_order_f_index_2d():
277: (4) a = arange(6)
278: (4) i = nditer(a.reshape(2, 3), ['f_index'], [['readonly']])
279: (4) assert_equal(iter_indices(i), [0, 2, 4, 1, 3, 5])
280: (4) i = nditer(a.reshape(2, 3).copy(order='F'),
281: (36) ['f_index'], [['readonly']])
282: (4) assert_equal(iter_indices(i), [0, 1, 2, 3, 4, 5])
283: (4) i = nditer(a.reshape(2, 3)[::-1], ['f_index'], [['readonly']])
284: (4) assert_equal(iter_indices(i), [1, 3, 5, 0, 2, 4])
285: (4) i = nditer(a.reshape(2, 3)[:, ::-1], ['f_index'], [['readonly']])
286: (4) assert_equal(iter_indices(i), [4, 2, 0, 5, 3, 1])
287: (4) i = nditer(a.reshape(2, 3)[::-1, ::-1], ['f_index'], [['readonly']])
288: (4) assert_equal(iter_indices(i), [5, 3, 1, 4, 2, 0])
289: (4) i = nditer(a.reshape(2, 3).copy(order='F')[::-1],
290: (36) ['f_index'], [['readonly']])
291: (4) assert_equal(iter_indices(i), [1, 0, 3, 2, 5, 4])
292: (4) i = nditer(a.reshape(2, 3).copy(order='F')[:, ::-1],
293: (36) ['f_index'], [['readonly']])
294: (4) assert_equal(iter_indices(i), [4, 5, 2, 3, 0, 1])
295: (4) i = nditer(a.reshape(2, 3).copy(order='F')[::-1, ::-1],
296: (36) ['f_index'], [['readonly']])
297: (4) assert_equal(iter_indices(i), [5, 4, 3, 2, 1, 0])
298: (0) def test_iter_best_order_f_index_3d():
299: (4) a = arange(12)
300: (4) i = nditer(a.reshape(2, 3, 2), ['f_index'], [['readonly']])
301: (4) assert_equal(iter_indices(i),
302: (28) [0, 6, 2, 8, 4, 10, 1, 7, 3, 9, 5, 11])
303: (4) i = nditer(a.reshape(2, 3, 2).copy(order='F'),
304: (36) ['f_index'], [['readonly']])
305: (4) assert_equal(iter_indices(i),
306: (28) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
307: (4) i = nditer(a.reshape(2, 3, 2)[::-1], ['f_index'], [['readonly']])
308: (4) assert_equal(iter_indices(i),
309: (28) [1, 7, 3, 9, 5, 11, 0, 6, 2, 8, 4, 10])
310: (4) i = nditer(a.reshape(2, 3, 2)[:, ::-1], ['f_index'], [['readonly']])
311: (4) assert_equal(iter_indices(i),
312: (28) [4, 10, 2, 8, 0, 6, 5, 11, 3, 9, 1, 7])
313: (4) i = nditer(a.reshape(2, 3, 2)[:::, ::-1], ['f_index'], [['readonly']])
314: (4) assert_equal(iter_indices(i),
315: (28) [6, 0, 8, 2, 10, 4, 7, 1, 9, 3, 11, 5])
316: (4) i = nditer(a.reshape(2, 3, 2).copy(order='F')[::-1],
317: (36) ['f_index'], [['readonly']])
318: (4) assert_equal(iter_indices(i),
319: (28) [1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10])
320: (4) i = nditer(a.reshape(2, 3, 2).copy(order='F')[:, ::-1],
321: (36) ['f_index'], [['readonly']])
322: (4) assert_equal(iter_indices(i),

```

```

323: (28) [4, 5, 2, 3, 0, 1, 10, 11, 8, 9, 6, 7])
324: (4) i = nditer(a.reshape(2, 3, 2).copy(order='F')[:::, ::-1],
325: (36) ['f_index'], [['readonly']])
326: (4) assert_equal(iter_indices(i),
327: (28) [6, 7, 8, 9, 10, 11, 0, 1, 2, 3, 4, 5])
328: (0) def test_iter_no_inner_full_coalesce():
329: (4) for shape in [(5,), (3, 4), (2, 3, 4), (2, 3, 4, 3), (2, 3, 2, 2, 3)]:
330: (8) size = np.prod(shape)
331: (8) a = arange(size)
332: (8) for dirs in range(2**len(shape)):
333: (12) dirs_index = [slice(None)]*len(shape)
334: (12) for bit in range(len(shape)):
335: (16) if (2**bit) & dirs:
336: (20) dirs_index[bit] = slice(None, None, -1)
337: (12) dirs_index = tuple(dirs_index)
338: (12) aview = a.reshape(shape)[dirs_index]
339: (12) i = nditer(aview, ['external_loop'], [['readonly']])
340: (12) assert_equal(i.ndim, 1)
341: (12) assert_equal(i[0].shape, (size,))
342: (12) i = nditer(aview.T, ['external_loop'], [['readonly']])
343: (12) assert_equal(i.ndim, 1)
344: (12) assert_equal(i[0].shape, (size,))
345: (12) if len(shape) > 2:
346: (16) i = nditer(aview.swapaxes(0, 1),
347: (36) ['external_loop'], [['readonly']])
348: (16) assert_equal(i.ndim, 1)
349: (16) assert_equal(i[0].shape, (size,))
350: (0) def test_iter_no_inner_dim_coalescing():
351: (4) a = arange(24).reshape(2, 3, 4)[:::, ::-1]
352: (4) i = nditer(a, ['external_loop'], [['readonly']])
353: (4) assert_equal(i.ndim, 2)
354: (4) assert_equal(i[0].shape, (3,))
355: (4) a = arange(24).reshape(2, 3, 4)[::, ::-1, ::]
356: (4) i = nditer(a, ['external_loop'], [['readonly']])
357: (4) assert_equal(i.ndim, 2)
358: (4) assert_equal(i[0].shape, (8,))
359: (4) a = arange(24).reshape(2, 3, 4)[::-1, ::, ::]
360: (4) i = nditer(a, ['external_loop'], [['readonly']])
361: (4) assert_equal(i.ndim, 1)
362: (4) assert_equal(i[0].shape, (12,))
363: (4) a = arange(24).reshape(1, 1, 2, 1, 1, 3, 1, 1, 4, 1, 1)
364: (4) i = nditer(a, ['external_loop'], [['readonly']])
365: (4) assert_equal(i.ndim, 1)
366: (4) assert_equal(i[0].shape, (24,))
367: (0) def test_iter_dim_coalescing():
368: (4) a = arange(24).reshape(2, 3, 4)
369: (4) i = nditer(a, ['multi_index'], [['readonly']])
370: (4) assert_equal(i.ndim, 3)
371: (4) a3d = arange(24).reshape(2, 3, 4)
372: (4) i = nditer(a3d, ['c_index'], [['readonly']])
373: (4) assert_equal(i.ndim, 1)
374: (4) i = nditer(a3d.swapaxes(0, 1), ['c_index'], [['readonly']])
375: (4) assert_equal(i.ndim, 3)
376: (4) i = nditer(a3d.T, ['c_index'], [['readonly']])
377: (4) assert_equal(i.ndim, 3)
378: (4) i = nditer(a3d.T, ['f_index'], [['readonly']])
379: (4) assert_equal(i.ndim, 1)
380: (4) i = nditer(a3d.T.swapaxes(0, 1), ['f_index'], [['readonly']])
381: (4) assert_equal(i.ndim, 3)
382: (4) a3d = arange(24).reshape(2, 3, 4)
383: (4) i = nditer(a3d, order='C')
384: (4) assert_equal(i.ndim, 1)
385: (4) i = nditer(a3d.T, order='C')
386: (4) assert_equal(i.ndim, 3)
387: (4) i = nditer(a3d, order='F')
388: (4) assert_equal(i.ndim, 3)
389: (4) i = nditer(a3d.T, order='F')
390: (4) assert_equal(i.ndim, 1)
391: (4) i = nditer(a3d, order='A')

```

```

392: (4)             assert_equal(i.ndim, 1)
393: (4)             i = nditer(a3d.T, order='A')
394: (4)             assert_equal(i.ndim, 1)
395: (0)             def test_iter_broadcasting():
396: (4)                 i = nditer([arange(6), np.int32(2)], ['multi_index'], [['readonly']]*2)
397: (4)                 assert_equal(i.iter_size, 6)
398: (4)                 assert_equal(i.shape, (6,))
399: (4)                 i = nditer([arange(6).reshape(2, 3), np.int32(2)],
400: (24)                         ['multi_index'], [['readonly']]*2)
401: (4)                         assert_equal(i.iter_size, 6)
402: (4)                         assert_equal(i.shape, (2, 3))
403: (4)                         i = nditer([arange(6).reshape(2, 3), arange(3)],
404: (24)                             ['multi_index'], [['readonly']]*2)
405: (4)                             assert_equal(i.iter_size, 6)
406: (4)                             assert_equal(i.shape, (2, 3))
407: (4)                             i = nditer([arange(2).reshape(2, 1), arange(3)],
408: (24)                                 ['multi_index'], [['readonly']]*2)
409: (4)                                 assert_equal(i.iter_size, 6)
410: (4)                                 assert_equal(i.shape, (2, 3))
411: (4)                                 i = nditer([arange(2).reshape(2, 1), arange(3).reshape(1, 3)],
412: (24)                                     ['multi_index'], [['readonly']]*2)
413: (4)                                     assert_equal(i.iter_size, 6)
414: (4)                                     assert_equal(i.shape, (2, 3))
415: (4)                                     i = nditer([np.int32(2), arange(24).reshape(4, 2, 3)],
416: (24)   ['multi_index'], [['readonly']]*2)
417: (4)   assert_equal(i.iter_size, 24)
418: (4)   assert_equal(i.shape, (4, 2, 3))
419: (4)   i = nditer([arange(3), arange(24).reshape(4, 2, 3)],
420: (24)   ['multi_index'], [['readonly']]*2)
421: (4)   assert_equal(i.iter_size, 24)
422: (4)   assert_equal(i.shape, (4, 2, 3))
423: (4)   i = nditer([arange(3), arange(8).reshape(4, 2, 1)],
424: (24)   ['multi_index'], [['readonly']]*2)
425: (4)   assert_equal(i.iter_size, 24)
426: (4)   assert_equal(i.shape, (4, 2, 3))
427: (4)   i = nditer([arange(6).reshape(2, 3), arange(24).reshape(4, 2, 3)],
428: (24)   ['multi_index'], [['readonly']]*2)
429: (4)   assert_equal(i.iter_size, 24)
430: (4)   assert_equal(i.shape, (4, 2, 3))
431: (4)   i = nditer([arange(2).reshape(2, 1), arange(24).reshape(4, 2, 3)],
432: (24)   ['multi_index'], [['readonly']]*2)
433: (4)   assert_equal(i.iter_size, 24)
434: (4)   assert_equal(i.shape, (4, 2, 3))
435: (4)   i = nditer([arange(3).reshape(1, 3), arange(8).reshape(4, 2, 1)],
436: (24)   ['multi_index'], [['readonly']]*2)
437: (4)   assert_equal(i.iter_size, 24)
438: (4)   assert_equal(i.shape, (4, 2, 3))
439: (4)   i = nditer([arange(2).reshape(1, 2, 1), arange(3).reshape(1, 1, 3),
440: (24)   arange(4).reshape(4, 1, 1)],
441: (24)   ['multi_index'], [['readonly']]*3)
442: (4)   assert_equal(i.iter_size, 24)
443: (4)   assert_equal(i.shape, (4, 2, 3))
444: (4)   i = nditer([arange(6).reshape(1, 2, 3), arange(4).reshape(4, 1, 1)],
445: (24)   ['multi_index'], [['readonly']]*2)
446: (4)   assert_equal(i.iter_size, 24)
447: (4)   assert_equal(i.shape, (4, 2, 3))
448: (4)   i = nditer([arange(24).reshape(4, 2, 3), arange(12).reshape(4, 1, 3)],
449: (24)   ['multi_index'], [['readonly']]*2)
450: (4)   assert_equal(i.iter_size, 24)
451: (4)   assert_equal(i.shape, (4, 2, 3))
452: (0)             def test_iter_iitershape():
453: (4)                 a = np.arange(6, dtype='i2').reshape(2, 3)
454: (4)                 i = nditer([a, None], [], [['readonly']], ['writeonly', 'allocate']),
455: (28)                     op_axes=[[0, 1, None], None],
456: (28)                     itershape=(-1, -1, 4))
457: (4)                     assert_equal(i.operands[1].shape, (2, 3, 4))
458: (4)                     assert_equal(i.operands[1].strides, (24, 8, 2))
459: (4)                     i = nditer([a.T, None], [], [['readonly']], ['writeonly', 'allocate']),
460: (28)                         op_axes=[[0, 1, None], None],

```

```

461: (28)                                itershape=(-1, -1, 4))
462: (4)       assert_equal(i.operands[1].shape, (3, 2, 4))
463: (4)       assert_equal(i.operands[1].strides, (8, 24, 2))
464: (4)       i = nditer([a.T, None], [], [['readonly']], ['writeonly', 'allocate']],
465: (28)                     order='F',
466: (28)                     op_axes=[[0, 1, None], None],
467: (28)                     itershape=(-1, -1, 4))
468: (4)       assert_equal(i.operands[1].shape, (3, 2, 4))
469: (4)       assert_equal(i.operands[1].strides, (2, 6, 12))
470: (4)       assert_raises(ValueError, nditer, [a, None], [],
471: (28)                     [['readonly']], ['writeonly', 'allocate']],
472: (28)                     op_axes=[[0, 1, None], None],
473: (28)                     itershape=(-1, 1, 4))
474: (4)       i = np.nditer([np.ones(2), None, None], itershape=(2,))
475: (0) def test_iter_broadcasting_errors():
476: (4)     assert_raises(ValueError, nditer, [arange(2), arange(3)],
477: (20)           [], [['readonly']]**2)
478: (4)     assert_raises(ValueError, nditer,
479: (20)           [arange(6).reshape(2, 3), arange(2)],
480: (20)           [], [['readonly']]**2)
481: (4)     assert_raises(ValueError, nditer,
482: (20)           [arange(6).reshape(2, 3), arange(9).reshape(3, 3)],
483: (20)           [], [['readonly']]**2)
484: (4)     assert_raises(ValueError, nditer,
485: (20)           [arange(6).reshape(2, 3), arange(4).reshape(2, 2)],
486: (20)           [], [['readonly']]**2)
487: (4)     assert_raises(ValueError, nditer,
488: (20)           [arange(36).reshape(3, 3, 4), arange(24).reshape(2, 3,
49: (4)],),
489: (20)           [], [['readonly']]**2)
490: (4)     assert_raises(ValueError, nditer,
491: (20)           [arange(8).reshape(2, 4, 1), arange(24).reshape(2, 3, 4)],
492: (20)           [], [['readonly']]**2)
493: (4) try:
494: (8)     nditer([arange(2).reshape(1, 2, 1),
495: (16)       arange(3).reshape(1, 3),
496: (16)       arange(6).reshape(2, 3)],
497: (15)       [],
498: (15)       [['readonly'], ['readonly'], ['writeonly', 'no_broadcast']])
499: (8)     raise AssertionError('Should have raised a broadcast error')
500: (4) except ValueError as e:
501: (8)     msg = str(e)
502: (8)     assert_(msg.find('(2,3)') >= 0,
503: (16)       'Message "%s" doesn\'t contain operand shape (2,3)' % msg)
504: (8)     assert_(msg.find('(1,2,3)') >= 0,
505: (16)       'Message "%s" doesn\'t contain broadcast shape (1,2,3)' % msg)
506: (4) try:
507: (8)     nditer([arange(6).reshape(2, 3), arange(2)],
508: (15)       [],
509: (15)       [['readonly'], ['readonly']],
510: (15)       op_axes=[[0, 1], [0, np.newaxis]],
511: (15)       itershape=(4, 3))
512: (8)     raise AssertionError('Should have raised a broadcast error')
513: (4) except ValueError as e:
514: (8)     msg = str(e)
515: (8)     assert_(msg.find('(2,3)->(2,3)') >= 0,
516: (12)       'Message "%s" doesn\'t contain operand shape (2,3)->(2,3)' % msg)
517: (8)     assert_(msg.find('(2,)->(2,newaxis)') >= 0,
518: (16)       ('Message "%s" doesn\'t contain remapped operand shape' +
519: (16)         '(2,)->(2,newaxis)') % msg)
520: (8)     assert_(msg.find('(4,3)') >= 0,
521: (16)       'Message "%s" doesn\'t contain itershape parameter (4,3)' %
522: (4) msg)
523: (8) try:
524: (15)     nditer([np.zeros((2, 1, 1)), np.zeros((2,))],
525: (15)       [],
526: (15)       [['writeonly', 'no_broadcast'], ['readonly']])
527: (8)     raise AssertionError('Should have raised a broadcast error')
528: (4) except ValueError as e:
```

```

528: (8)           msg = str(e)
529: (8)           assert_(msg.find('(2,1,1)') >= 0,
530: (12)             'Message "%s" doesn\'t contain operand shape (2,1,1)' % msg)
531: (8)           assert_(msg.find('(2,1,2)') >= 0,
532: (16)             'Message "%s" doesn\'t contain the broadcast shape (2,1,2)' %
533: (0)           msg)
534: (4)           def test_iter_flags_errors():
535: (4)             a = arange(6)
536: (4)             assert_raises(ValueError, nditer, [], [], [])
537: (4)             assert_raises(ValueError, nditer, [a]*100, [], [['readonly']]*100)
538: (4)             assert_raises(ValueError, nditer, [a], ['bad flag'], [['readonly']])
539: (4)             assert_raises(ValueError, nditer, [a], [], [['readonly', 'bad flag']])
540: (4)             assert_raises(ValueError, nditer, [a], [], [['readonly']], order='G')
541: (4)             assert_raises(ValueError, nditer, [a], [], [['readonly']], casting='noon')
542: (4)             assert_raises(ValueError, nditer, [a]**3, [], [[['readonly']]*2)
543: (4)             assert_raises(ValueError, nditer, a,
544: (4)               ['c_index', 'f_index'], [['readonly']])
545: (4)             assert_raises(ValueError, nditer, a,
546: (4)               ['external_loop', 'multi_index'], [['readonly']])
547: (4)             assert_raises(ValueError, nditer, a,
548: (4)               ['external_loop', 'c_index'], [['readonly']])
549: (4)             assert_raises(ValueError, nditer, a,
550: (4)               ['external_loop', 'f_index'], [['readonly']])
551: (4)             assert_raises(ValueError, nditer, a, [])
552: (4)             assert_raises(ValueError, nditer, a, [['readonly', 'writeonly']])
553: (4)             assert_raises(ValueError, nditer, a, [['readonly', 'readwrite']])
554: (4)             assert_raises(ValueError, nditer, a,
555: (4)               [], [['readonly', 'writeonly', 'readwrite']])
556: (4)             assert_raises(TypeError, nditer, 1.5, [], [['writeonly']])
557: (4)             assert_raises(TypeError, nditer, 1.5, [], [['readwrite']])
558: (4)             assert_raises(TypeError, nditer, np.int32(1), [], [['writeonly']])
559: (4)             assert_raises(TypeError, nditer, np.int32(1), [], [['readwrite']])
560: (4)             a.flags.writeable = False
561: (4)             assert_raises(ValueError, nditer, a, [['writeonly']])
562: (4)             assert_raises(ValueError, nditer, a, [['readwrite']])
563: (4)             a.flags.writeable = True
564: (4)             i = nditer(arange(6), [['readonly']])
565: (4)             assert_raises(ValueError, lambda i:i.multi_index, i)
566: (4)             assert_raises(ValueError, lambda i:i.index, i)
567: (4)             def assign_multi_index(i):
568: (8)               i.multi_index = (0,)
569: (4)             def assign_index(i):
570: (8)               i.index = 0
571: (4)             def assign_iterindex(i):
572: (8)               i.iterindex = 0
573: (4)             def assign_iterrange(i):
574: (8)               i.iterrange = (0, 1)
575: (4)             i = nditer(arange(6), ['external_loop'])
576: (4)             assert_raises(ValueError, assign_multi_index, i)
577: (4)             assert_raises(ValueError, assign_index, i)
578: (4)             assert_raises(ValueError, assign_iterindex, i)
579: (4)             assert_raises(ValueError, assign_iterrange, i)
580: (4)             i = nditer(arange(6), ['buffered'])
581: (4)             assert_raises(ValueError, assign_multi_index, i)
582: (4)             assert_raises(ValueError, assign_index, i)
583: (4)             assert_raises(ValueError, assign_iterindex, i)
584: (4)             assert_raises(ValueError, nditer, np.array([]))
585: (0)             def test_iter_slice():
586: (4)               a, b, c = np.arange(3), np.arange(3), np.arange(3.)
587: (4)               i = nditer([a, b, c], [], ['readwrite'])
588: (4)               with i:
589: (8)                 i[0:2] = (3, 3)
590: (8)                 assert_equal(a, [3, 1, 2])
591: (8)                 assert_equal(b, [3, 1, 2])
592: (8)                 assert_equal(c, [0, 1, 2])
593: (8)                 i[1] = 12
594: (8)                 assert_equal(i[0:2], [3, 12])
595: (0)             def test_iter_assign_mapping():

```

```

596: (4)
597: (4)
598: (23)
599: (4)
600: (8)
601: (8)
602: (4)
603: (4)
604: (23)
605: (4)
606: (8)
607: (8)
608: (8)
609: (4)
610: (4)
611: (4)
612: (0)
613: (4)
614: (4)
615: (4)
616: (4)
617: (24)
618: (24)
619: (4)
620: (8)
621: (8)
622: (8)
623: (8)
624: (4)
625: (4)
626: (4)
627: (4)
628: (4)
629: (4)
630: (24)
631: (8)
632: (8)
633: (8)
634: (8)
635: (8)
636: (4)
637: (4)
638: (4)
639: (4)
640: (4)
641: (4)
642: (4)
643: (4)
644: (4)
645: (8)
646: (8)
647: (8)
648: (4)
649: (4)
650: (4)
651: (4)
652: (4)
653: (4)
654: (24)
655: (24)
656: (4)
657: (4)
658: (0)
659: (4)
660: (4)
661: (4)
662: (8)
663: (8)
664: (4)

      a = np.arange(24, dtype='f8').reshape(2, 3, 4).T
      it = np.nditer(a, [], [['readonly', 'updateifcopy']],
                     casting='same_kind', op_dtypes=[np.dtype('f4')])
      with it:
          it.operands[0][...] = 3
          it.operands[0][...] = 14
          assert_equal(a, 14)
      it = np.nditer(a, [], [['readonly', 'updateifcopy']],
                     casting='same_kind', op_dtypes=[np.dtype('f4')])
      with it:
          x = it.operands[0][-1:1]
          x[...] = 14
          it.operands[0][...] = -1234
          assert_equal(a, -1234)
      x = None
      it = None

def test_iter_nbo_align_contig():
    a = np.arange(6, dtype='f4')
    au = a.byteswap().newbyteorder()
    assert_(a.dtype.byteorder != au.dtype.byteorder)
    i = nditer(au, [], [['readonly', 'updateifcopy']],
               casting='equiv',
               op_dtypes=[np.dtype('f4')])
    with i:
        assert_equal(i.dtypes[0].byteorder, a.dtype.byteorder)
        assert_equal(i.operands[0].dtype.byteorder, a.dtype.byteorder)
        assert_equal(i.operands[0], a)
        i.operands[0][:] = 2
    assert_equal(au, [2]*6)
    del i # should not raise a warning
    a = np.arange(6, dtype='f4')
    au = a.byteswap().newbyteorder()
    assert_(a.dtype.byteorder != au.dtype.byteorder)
    with nditer(au, [], [['readonly', 'updateifcopy', 'nbo']],
                casting='equiv') as i:
        assert_equal(i.dtypes[0].byteorder, a.dtype.byteorder)
        assert_equal(i.operands[0].dtype.byteorder, a.dtype.byteorder)
        assert_equal(i.operands[0], a)
        i.operands[0][:] = 12345
        i.operands[0][:] = 2
    assert_equal(au, [2]*6)
    a = np.zeros((6*4+1,), dtype='i1')[1:]
    a.dtype = 'f4'
    a[:] = np.arange(6, dtype='f4')
    assert_(not a.flags.aligned)
    i = nditer(a, [], [['readonly']])
    assert_(not i.operands[0].flags.aligned)
    assert_equal(i.operands[0], a)
    with nditer(a, [], [['readonly', 'updateifcopy', 'aligned']]) as i:
        assert_(i.operands[0].flags.aligned)
        assert_equal(i.operands[0], a)
        i.operands[0][:] = 3
    assert_equal(a, [3]*6)
    a = arange(12)
    i = nditer(a[:6], [], [['readonly']])
    assert_(i.operands[0].flags.contiguous)
    assert_equal(i.operands[0], a[:6])
    i = nditer(a[::2], ['buffered', 'external_loop'],
               [['readonly', 'contig']],
               buffersize=10)
    assert_(i[0].flags.contiguous)
    assert_equal(i[0], a[::2])

def test_iter_array_cast():
    a = np.arange(6, dtype='f4').reshape(2, 3)
    i = nditer(a, [], [['readonly']], op_dtypes=[np.dtype('f4')])
    with i:
        assert_equal(i.operands[0], a)
        assert_equal(i.operands[0].dtype, np.dtype('f4'))
    a = np.arange(6, dtype='<f4').reshape(2, 3)

```

```

665: (4)
666: (12)
667: (12)
668: (8)
669: (8)
670: (4)
671: (4)
672: (12)
673: (12)
674: (4)
675: (4)
676: (4)
677: (4)
678: (4)
679: (12)
680: (12)
681: (4)
682: (4)
683: (4)
684: (4)
685: (4)
686: (12)
687: (12)
688: (12)
689: (8)
690: (8)
691: (8)
692: (8)
693: (8)
694: (4)
695: (4)
696: (4)
697: (12)
698: (12)
699: (12)
700: (8)
701: (8)
702: (8)
703: (4)
704: (0)
def test_iter_array_cast_errors():
    assert_raises(TypeError, nditer, arange(2, dtype='f4'), [],
                 [['readonly']], op_dtypes=[np.dtype('f8')])
    assert_raises(TypeError, nditer, arange(2, dtype='f4'), [],
                 [['readonly', 'copy']], casting='no',
                 op_dtypes=[np.dtype('f8')])
    assert_raises(TypeError, nditer, arange(2, dtype='f4'), [],
                 [['readonly', 'copy']], casting='equiv',
                 op_dtypes=[np.dtype('f8')])
    assert_raises(TypeError, nditer, arange(2, dtype='f8'), [],
                 [['writeonly', 'updateifcopy']],
                 casting='no',
                 op_dtypes=[np.dtype('f4')])
    assert_raises(TypeError, nditer, arange(2, dtype='f8'), [],
                 [['writeonly', 'updateifcopy']],
                 casting='equiv',
                 op_dtypes=[np.dtype('f4')])
    assert_raises(TypeError, nditer, arange(2, dtype='<f4'), [],
                 [['readonly', 'copy']], casting='no',
                 op_dtypes=[np.dtype('>f4')])
    assert_raises(TypeError, nditer, arange(2, dtype='f4'), [],
                 [['readwrite', 'updateifcopy']],
                 casting='safe',
                 op_dtypes=[np.dtype('f8')])
    assert_raises(TypeError, nditer, arange(2, dtype='f8'), [],
                 [['readwrite', 'updateifcopy']],
                 casting='safe',
                 op_dtypes=[np.dtype('f4')])
    assert_raises(TypeError, nditer, arange(2, dtype='f4'), [],
                 [['readonly', 'copy']])

```

```

734: (16)           casting='same_kind',
735: (16)           op_dtypes=[np.dtype('i4')])
736: (4)             assert_raises(TypeError, nditer, arange(2, dtype='i4'), [],
737: (16)                         [['writeonly', 'updateifcopy']]],
738: (16)                         casting='same_kind',
739: (16)                         op_dtypes=[np.dtype('f4')])
740: (0)             def test_iter_scalar_cast():
741: (4)               i = nditer(np.float32(2.5), [], [['readonly']],
742: (20)                 op_dtypes=[np.dtype('f4')])
743: (4)                 assert_equal(i.dtypes[0], np.dtype('f4'))
744: (4)                 assert_equal(i.value.dtype, np.dtype('f4'))
745: (4)                 assert_equal(i.value, 2.5)
746: (4)                 i = nditer(np.float32(2.5), [],
747: (20)                   [['readonly', 'copy']],
748: (20)                     casting='safe',
749: (20)                     op_dtypes=[np.dtype('f8')])
750: (4)                     assert_equal(i.dtypes[0], np.dtype('f8'))
751: (4)                     assert_equal(i.value.dtype, np.dtype('f8'))
752: (4)                     assert_equal(i.value, 2.5)
753: (4)                     i = nditer(np.float64(2.5), [],
754: (20)                       [['readonly', 'copy']],
755: (20)                         casting='same_kind',
756: (20)                         op_dtypes=[np.dtype('f4')])
757: (4)                         assert_equal(i.dtypes[0], np.dtype('f4'))
758: (4)                         assert_equal(i.value.dtype, np.dtype('f4'))
759: (4)                         assert_equal(i.value, 2.5)
760: (4)                         i = nditer(np.float64(3.0), [],
761: (20)                           [['readonly', 'copy']],
762: (20)                             casting='unsafe',
763: (20)                             op_dtypes=[np.dtype('i4')])
764: (4)                             assert_equal(i.dtypes[0], np.dtype('i4'))
765: (4)                             assert_equal(i.value.dtype, np.dtype('i4'))
766: (4)                             assert_equal(i.value, 3)
767: (4)                             i = nditer(3, [], [['readonly']], op_dtypes=[np.dtype('f8')])
768: (4)                             assert_equal(i[0].dtype, np.dtype('f8'))
769: (4)                             assert_equal(i[0], 3.)
770: (0)             def test_iter_scalar_cast_errors():
771: (4)               assert_raises(TypeError, nditer, np.float32(2), [],
772: (16)                 [['readwrite']], op_dtypes=[np.dtype('f8')])
773: (4)               assert_raises(TypeError, nditer, 2.5, [],
774: (16)                 [['readwrite']], op_dtypes=[np.dtype('f4')])
775: (4)               assert_raises(TypeError, nditer, np.float64(1e60), [],
776: (16)                 [['readonly']],
777: (16)                   casting='safe',
778: (16)                   op_dtypes=[np.dtype('f4')])
779: (4)                   assert_raises(TypeError, nditer, np.float32(2), [],
780: (16)                     [['readonly']],
781: (16)                     casting='same_kind',
782: (16)                     op_dtypes=[np.dtype('i4')])
783: (0)             def test_iter_object_arrays_basic():
784: (4)               obj = {'a':3,'b':'d'}
785: (4)               a = np.array([[1, 2, 3], None, obj, None], dtype='O')
786: (4)               if HAS_REFCOUNT:
787: (8)                 rc = sys.getrefcount(obj)
788: (4)                 assert_raises(TypeError, nditer, a)
789: (4)                 if HAS_REFCOUNT:
790: (8)                   assert_equal(sys.getrefcount(obj), rc)
791: (4)                   i = nditer(a, ['refs_ok'], ['readonly'])
792: (4)                   vals = [x_[()] for x_ in i]
793: (4)                   assert_equal(np.array(vals, dtype='O'), a)
794: (4)                   vals, i, x = [None]*3
795: (4)                   if HAS_REFCOUNT:
796: (8)                     assert_equal(sys.getrefcount(obj), rc)
797: (4)                     i = nditer(a.reshape(2, 2).T, ['refs_ok', 'buffered'],
798: (24)                       ['readonly'], order='C')
799: (4)                       assert_(i.iterationneedsapi)
800: (4)                       vals = [x_[()] for x_ in i]
801: (4)                       assert_equal(np.array(vals, dtype='O'), a.reshape(2, 2).ravel(order='F'))
802: (4)                       vals, i, x = [None]*3

```

```

803: (4)
804: (8)
805: (4)
806: (24)
807: (4)
808: (8)
809: (12)
810: (8)
811: (4)
812: (8)
813: (4)
814: (0)
815: (4)
816: (4)
817: (20)
818: (4)
819: (8)
820: (12)
821: (4)
822: (4)
823: (4)
824: (20)
825: (4)
826: (8)
827: (12)
828: (4)
829: (4)
830: (4)
831: (4)
832: (4)
833: (20)
834: (4)
835: (8)
836: (12)
837: (4)
838: (4)
839: (4)
840: (4)
841: (4)
842: (20)
843: (4)
844: (8)
845: (8)
846: (12)
847: (8)
848: (12)
849: (4)
850: (8)
851: (4)
852: (0)
853: (4)
854: (20)
855: (20)
856: (20)
857: (4)
858: (4)
859: (4)
860: (20)
861: (20)
862: (20)
863: (4)
864: (4)
865: (4)
866: (20)
867: (20)
868: (20)
869: (4)
870: (4)
871: (4)

    if HAS_REFCOUNT:
        assert_equal(sys.getrefcount(obj), rc)
    i = nditer(a.reshape(2, 2).T, ['refs_ok', 'buffered'],
               ['readwrite'], order='C')
    with i:
        for x in i:
            x[...] = None
            vals, i, x = [None]*3
    if HAS_REFCOUNT:
        assert_(sys.getrefcount(obj) == rc-1)
    assert_equal(a, np.array([None]*4, dtype='O'))
def test_iter_object_arrays_conversions():
    a = np.arange(6, dtype='O')
    i = nditer(a, ['refs_ok', 'buffered'], ['readwrite'],
               casting='unsafe', op_dtypes='i4')
    with i:
        for x in i:
            x[...] += 1
    assert_equal(a, np.arange(6)+1)
    a = np.arange(6, dtype='i4')
    i = nditer(a, ['refs_ok', 'buffered'], ['readwrite'],
               casting='unsafe', op_dtypes='O')
    with i:
        for x in i:
            x[...] += 1
    assert_equal(a, np.arange(6)+1)
    a = np.zeros((6,), dtype=[('p', 'i1'), ('a', 'O')])
    a = a['a']
    a[:] = np.arange(6)
    i = nditer(a, ['refs_ok', 'buffered'], ['readwrite'],
               casting='unsafe', op_dtypes='i4')
    with i:
        for x in i:
            x[...] += 1
    assert_equal(a, np.arange(6)+1)
    a = np.zeros((6,), dtype=[('p', 'i1'), ('a', 'i4')])
    a = a['a']
    a[:] = np.arange(6) + 98172488
    i = nditer(a, ['refs_ok', 'buffered'], ['readwrite'],
               casting='unsafe', op_dtypes='O')
    with i:
        ob = i[0][()]
        if HAS_REFCOUNT:
            rc = sys.getrefcount(ob)
        for x in i:
            x[...] += 1
        if HAS_REFCOUNT:
            assert_(sys.getrefcount(ob) == rc-1)
    assert_equal(a, np.arange(6)+98172489)
def test_iter_common_dtype():
    i = nditer([array([3], dtype='f4'), array([0], dtype='f8')],
               ['common_dtype'],
               [['readonly', 'copy']] * 2,
               casting='safe')
    assert_equal(i.dtypes[0], np.dtype('f8'))
    assert_equal(i.dtypes[1], np.dtype('f8'))
    i = nditer([array([3], dtype='i4'), array([0], dtype='f4')],
               ['common_dtype'],
               [['readonly', 'copy']] * 2,
               casting='safe')
    assert_equal(i.dtypes[0], np.dtype('f8'))
    assert_equal(i.dtypes[1], np.dtype('f8'))
    i = nditer([array([3], dtype='f4'), array(0, dtype='f8')],
               ['common_dtype'],
               [['readonly', 'copy']] * 2,
               casting='same_kind')
    assert_equal(i.dtypes[0], np.dtype('f4'))
    assert_equal(i.dtypes[1], np.dtype('f4'))
    i = nditer([array([3], dtype='u4'), array(0, dtype='i4')],

```

```

872: (20)                                ['common_dtype'],
873: (20)                                [['readonly', 'copy']] * 2,
874: (20)                                casting='safe')
875: (4)       assert_equal(i.dtypes[0], np.dtype('u4'))
876: (4)       assert_equal(i.dtypes[1], np.dtype('u4'))
877: (4)       i = nditer([array([3], dtype='u4'), array(-12, dtype='i4')],
878: (20)                               ['common_dtype'],
879: (20)                               [['readonly', 'copy']] * 2,
880: (20)                               casting='safe')
881: (4)       assert_equal(i.dtypes[0], np.dtype('i8'))
882: (4)       assert_equal(i.dtypes[1], np.dtype('i8'))
883: (4)       i = nditer([array([3], dtype='u4'), array(-12, dtype='i4'),
884: (17)                               array([2j], dtype='c8'), array([9], dtype='f8')],
885: (20)                               ['common_dtype'],
886: (20)                               [['readonly', 'copy']] * 4,
887: (20)                               casting='safe')
888: (4)       assert_equal(i.dtypes[0], np.dtype('c16'))
889: (4)       assert_equal(i.dtypes[1], np.dtype('c16'))
890: (4)       assert_equal(i.dtypes[2], np.dtype('c16'))
891: (4)       assert_equal(i.dtypes[3], np.dtype('c16'))
892: (4)       assert_equal(i.value, (3, -12, 2j, 9))
893: (4)       i = nditer([array([3], dtype='i4'), None, array([2j], dtype='c16')], [],
894: (20)                               [['readonly', 'copy'],
895: (21)                               ['writeonly', 'allocate'],
896: (21)                               ['writeonly']],
897: (20)                               casting='safe')
898: (4)       assert_equal(i.dtypes[0], np.dtype('i4'))
899: (4)       assert_equal(i.dtypes[1], np.dtype('i4'))
900: (4)       assert_equal(i.dtypes[2], np.dtype('c16'))
901: (4)       i = nditer([array([3], dtype='i4'), None, array([2j], dtype='c16')],
902: (20)                               ['common_dtype'],
903: (20)                               [['readonly', 'copy'],
904: (21)                               ['writeonly', 'allocate'],
905: (21)                               ['writeonly']],
906: (20)                               casting='safe')
907: (4)       assert_equal(i.dtypes[0], np.dtype('c16'))
908: (4)       assert_equal(i.dtypes[1], np.dtype('c16'))
909: (4)       assert_equal(i.dtypes[2], np.dtype('c16'))
910: (0) def test_iter_copy_if_overlap():
911: (4)     for flag in ['readonly', 'writeonly', 'readwrite']:
912: (8)       a = arange(10)
913: (8)       i = nditer([a], ['copy_if_overlap'], [[flag]])
914: (8)       with i:
915: (12)         assert_(i.operands[0] is a)
916: (4)       x = arange(10)
917: (4)       a = x[1:]
918: (4)       b = x[:-1]
919: (4)       with nditer([a, b], ['copy_if_overlap'], [['readonly'], ['readwrite']]) as
920: (8)         i:
921: (4)           assert_(not np.shares_memory(*i.operands))
922: (4)           x = arange(10)
923: (4)           a = x
924: (4)           b = x
925: (45)           i = nditer([a, b], ['copy_if_overlap'], [['readonly',
926: (45)             'overlap_assume_elementwise'],
927: (45)             ['readwrite'],
928: (45)             'overlap_assume_elementwise']])
929: (45)           with i:
930: (8)             assert_(i.operands[0] is a and i.operands[1] is b)
931: (4)             with nditer([a, b], ['copy_if_overlap'], [['readonly'], ['readwrite']]) as
932: (4)               i:
933: (8)                 assert_(i.operands[0] is a and not np.shares_memory(i.operands[1], b))
934: (4)                 x = arange(10)
935: (4)                 a = x[::2]
936: (4)                 b = x[1::2]
937: (4)                 i = nditer([a, b], ['copy_if_overlap'], [['readonly'], ['writeonly']]))
938: (4)                 assert_(i.operands[0] is a and i.operands[1] is b)
939: (4)                 x = arange(4, dtype=np.int8)
940: (4)                 a = x[3:]
```

```

937: (4)          b = x.view(np.int32)[:1]
938: (4)          with nditer([a, b], ['copy_if_overlap'], [['readonly'], ['writeonly']]) as i:
939: (8)              assert_(not np.shares_memory(*i.operands))
940: (4)          for flag in ['writeonly', 'readwrite']:
941: (8)              x = np.ones([10, 10])
942: (8)              a = x
943: (8)              b = x.T
944: (8)              c = x
945: (8)              with nditer([a, b, c], ['copy_if_overlap'],
946: (19)                  [['readonly'], ['readonly'], [flag]]) as i:
947: (12)                      a2, b2, c2 = i.operands
948: (12)                      assert_(not np.shares_memory(a2, c2))
949: (12)                      assert_(not np.shares_memory(b2, c2))
950: (4)          x = np.ones([10, 10])
951: (4)          a = x
952: (4)          b = x.T
953: (4)          c = x
954: (4)          i = nditer([a, b, c], ['copy_if_overlap'],
955: (15)              [['readonly'], ['readonly'], ['readonly']])
956: (4)          a2, b2, c2 = i.operands
957: (4)          assert_(a is a2)
958: (4)          assert_(b is b2)
959: (4)          assert_(c is c2)
960: (4)          x = np.ones([10, 10])
961: (4)          a = x
962: (4)          b = np.ones([10, 10])
963: (4)          c = x.T
964: (4)          i = nditer([a, b, c], ['copy_if_overlap'],
965: (15)              [['readonly'], ['writeonly'], ['readonly']])
966: (4)          a2, b2, c2 = i.operands
967: (4)          assert_(a is a2)
968: (4)          assert_(b is b2)
969: (4)          assert_(c is c2)
970: (4)          x = np.arange(7)
971: (4)          a = x[:3]
972: (4)          b = x[3:6]
973: (4)          c = x[4:7]
974: (4)          i = nditer([a, b, c], ['copy_if_overlap'],
975: (15)              [['readonly'], ['writeonly'], ['writeonly']])
976: (4)          a2, b2, c2 = i.operands
977: (4)          assert_(a is a2)
978: (4)          assert_(b is b2)
979: (4)          assert_(c is c2)
980: (0)      def test_iter_op_axes():
981: (4)          a = arange(6).reshape(2, 3)
982: (4)          i = nditer([a, a.T], [], [['readonly']]*2, op_axes=[[0, 1], [1, 0]])
983: (4)          assert_(all([x == y for (x, y) in i]))
984: (4)          a = arange(24).reshape(2, 3, 4)
985: (4)          i = nditer([a.T, a], [], [['readonly']]*2, op_axes=[[2, 1, 0], None])
986: (4)          assert_(all([x == y for (x, y) in i]))
987: (4)          a = arange(1, 31).reshape(2, 3, 5)
988: (4)          b = arange(1, 3)
989: (4)          i = nditer([a, b], [], [['readonly']]*2, op_axes=[None, [0, -1, -1]])
990: (4)          assert_equal([x*y for (x, y) in i], (a*b.reshape(2, 1, 1)).ravel())
991: (4)          b = arange(1, 4)
992: (4)          i = nditer([a, b], [], [['readonly']]*2, op_axes=[None, [-1, 0, -1]])
993: (4)          assert_equal([x*y for (x, y) in i], (a*b.reshape(1, 3, 1)).ravel())
994: (4)          b = arange(1, 6)
995: (4)          i = nditer([a, b], [], [['readonly']]*2,
996: (28)              op_axes=[None, [np.newaxis, np.newaxis, 0]])
997: (4)          assert_equal([x*y for (x, y) in i], (a*b.reshape(1, 1, 5)).ravel())
998: (4)          a = arange(24).reshape(2, 3, 4)
999: (4)          b = arange(40).reshape(5, 2, 4)
1000: (4)          i = nditer([a, b], ['multi_index'], [['readonly']]*2,
1001: (28)              op_axes=[[0, 1, -1, -1], [-1, -1, 0, 1]])
1002: (4)          assert_equal(i.shape, (2, 3, 5, 2))
1003: (4)          a = arange(12).reshape(3, 4)
1004: (4)          b = arange(20).reshape(4, 5)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1005: (4)
1006: (28)
1007: (4)
1008: (0)
1009: (4)
1010: (4)
1011: (36)
1012: (4)
1013: (36)
1014: (4)
1015: (36)
1016: (4)
1017: (36)
1018: (4)
1019: (36)
1020: (4)
1021: (36)
1022: (4)
1023: (36)
1024: (0)
1025: (4)
1026: (4)
1027: (4)
1028: (4)
1029: (4)
1030: (4)
1031: (4)
1032: (4)
1033: (4)
1034: (4)
1035: (4)
1036: (4)
1037: (4)
1038: (4)
1039: (4)
1040: (4)
1041: (4)
1042: (4)
1043: (4)
1044: (4)
1045: (4)
1046: (4)
1047: (16)
1048: (8)
1049: (4)
1050: (4)
1051: (4)
1052: (16)
1053: (8)
1054: (4)
1055: (0)
1056: (0)
1057: (0)
1058: (0)
1059: (4)
1060: (8)
1061: (4)
1062: (8)
1063: (4)
1064: (4)
1065: (8)
1066: (4)
1067: (8)
1068: (4)
1069: (19)
1070: (4)
1071: (8)
1072: (4)
1073: (4)

    i = nditer([a, b], ['multi_index'], [[['readonly']]*2,
   op_axes=[[0, -1], [-1, 1]]])
    assert_equal(i.shape, (3, 5))
def test_iter_op_axes_errors():
    a = arange(6).reshape(2, 3)
    assert_raises(ValueError, nditer, [a, a], [], [[['readonly']]*2,
  op_axes=[[0, 1], [0, 0]]])
    assert_raises(ValueError, nditer, [a, a], [], [[['readonly']]*2,
  op_axes=[[2, 1], [0, 1]]])
    assert_raises(ValueError, nditer, [a, a], [], [[['readonly']]*2,
  op_axes=[[0, 1], [2, -1]]])
    assert_raises(ValueError, nditer, [a, a], [], [[['readonly']]*2,
  op_axes=[[0, 0], [0, 1]]])
    assert_raises(ValueError, nditer, [a, a], [], [[['readonly']]*2,
  op_axes=[[0, 1], [1, 1]]])
    assert_raises(ValueError, nditer, [a, a], [], [[['readonly']]*2,
  op_axes=[[0, 1], [0, 1, 0]]])
    assert_raises(ValueError, nditer, [a, a], [], [[['readonly']]*2,
  op_axes=[[0, 1], [1, 0]]])

def test_iter_copy():
    a = arange(24).reshape(2, 3, 4)
    i = nditer(a)
    j = i.copy()
    assert_equal([x[()] for x in i], [x[()] for x in j])
    i.iterindex = 3
    j = i.copy()
    assert_equal([x[()] for x in i], [x[()] for x in j])
    i = nditer(a, ['buffered', 'ranged'], order='F', buffersize=3)
    j = i.copy()
    assert_equal([x[()] for x in i], [x[()] for x in j])
    i.iterindex = 3
    j = i.copy()
    assert_equal([x[()] for x in i], [x[()] for x in j])
    i.iterrange = (3, 9)
    j = i.copy()
    assert_equal([x[()] for x in i], [x[()] for x in j])
    i.iterrange = (2, 18)
    next(i)
    next(i)
    j = i.copy()
    assert_equal([x[()] for x in i], [x[()] for x in j])
    with nditer(a, ['buffered'], order='F', casting='unsafe',
                op_dtypes='f8', buffersize=5) as i:
        j = i.copy()
    assert_equal([x[()] for x in j], a.ravel(order='F'))
    a = arange(24, dtype='<i4').reshape(2, 3, 4)
    with nditer(a, ['buffered'], order='F', casting='unsafe',
                op_dtypes='>f8', buffersize=5) as i:
        j = i.copy()
    assert_equal([x[()] for x in j], a.ravel(order='F'))

@pytest.mark.parametrize("dtype", np.typecodes["All"])
@pytest.mark.parametrize("loop_dtype", np.typecodes["All"])
@pytest.mark.filterwarnings("ignore::numpy.ComplexWarning")
def test_iter_copy_casts(dtype, loop_dtype):
    if loop_dtype.lower() == "m":
        loop_dtype = loop_dtype + "[ms]"
    elif np.dtype(loop_dtype).itemsize == 0:
        loop_dtype = loop_dtype + "50"
    arr = np.ones(1000, dtype=np.dtype(dtype).newbyteorder())
    try:
        expected = arr.astype(loop_dtype)
    except Exception:
        return
    it = np.nditer((arr,), ["buffered", "external_loop", "refs_ok"],
                  op_dtypes=[loop_dtype], casting="unsafe")
    if np.issubdtype(np.dtype(loop_dtype), np.number):
        assert_array_equal(expected, np.ones(1000, dtype=loop_dtype))
    it_copy = it.copy()
    res = next(it)

```

```

1074: (4)           del it
1075: (4)           res_copy = next(it_copy)
1076: (4)           del it_copy
1077: (4)           assert_array_equal(res, expected)
1078: (4)           assert_array_equal(res_copy, expected)
1079: (0)           def test_iter_copy_casts_structured():
1080: (4)             in_dtype = np.dtype([("a", np.dtype("i, ")),
1081: (25)                           ("b", np.dtype(">i,<i,>d,S17,>d,(3)f,0,i1"))])
1082: (4)             out_dtype = np.dtype([("a", np.dtype("0")),
1083: (26)                           ("b", np.dtype(">i,>i,S17,>d,>U3,(3)d,i1,0"))])
1084: (4)             arr = np.ones(1000, dtype=in_dtype)
1085: (4)             it = np.nditer((arr,), ["buffered", "external_loop", "refs_ok"],
1086: (19)                           op_dtypes=[out_dtype], casting="unsafe")
1087: (4)             it_copy = it.copy()
1088: (4)             res1 = next(it)
1089: (4)             del it
1090: (4)             res2 = next(it_copy)
1091: (4)             del it_copy
1092: (4)             expected = arr["a"].astype(out_dtype["a"])
1093: (4)             assert_array_equal(res1["a"], expected)
1094: (4)             assert_array_equal(res2["a"], expected)
1095: (4)             for field in in_dtype["b"].names:
1096: (8)               expected = arr["b"][field].astype(out_dtype["b"][field].base)
1097: (8)               assert_array_equal(res1["b"][field], expected)
1098: (8)               assert_array_equal(res2["b"][field], expected)
1099: (0)           def test_iter_copy_casts_structured2():
1100: (4)             in_dtype = np.dtype([("a", np.dtype("0,0")),
1101: (25)                           ("b", np.dtype("(5)0,(3)0,(1,)0,(1,)i,(1,)0"))])
1102: (4)             out_dtype = np.dtype([("a", np.dtype("0")),
1103: (26)                           ("b", np.dtype("0,(3)i,(4)0,(4)0,(4)i"))])
1104: (4)             arr = np.ones(1, dtype=in_dtype)
1105: (4)             it = np.nditer((arr,), ["buffered", "external_loop", "refs_ok"],
1106: (19)                           op_dtypes=[out_dtype], casting="unsafe")
1107: (4)             it_copy = it.copy()
1108: (4)             res1 = next(it)
1109: (4)             del it
1110: (4)             res2 = next(it_copy)
1111: (4)             del it_copy
1112: (4)             for res in res1, res2:
1113: (8)               assert type(res["a"][0]) == tuple
1114: (8)               assert res["a"][0] == (1, 1)
1115: (4)             for res in res1, res2:
1116: (8)               assert_array_equal(res["b"]["f0"][0], np.ones(5, dtype=object))
1117: (8)               assert_array_equal(res["b"]["f1"], np.ones((1, 3), dtype="i"))
1118: (8)               assert res["b"]["f2"].shape == (1, 4)
1119: (8)               assert_array_equal(res["b"]["f2"][0], np.ones(4, dtype=object))
1120: (8)               assert_array_equal(res["b"]["f3"][0], np.ones(4, dtype=object))
1121: (8)               assert_array_equal(res["b"]["f3"][0], np.ones(4, dtype="i"))
1122: (0)           def test_iter_allocate_output_simple():
1123: (4)             a = arange(6)
1124: (4)             i = nditer([a, None], [], [['readonly'], ['writeonly', 'allocate']],
1125: (24)                           op_dtypes=[None, np.dtype('f4')])
1126: (4)             assert_equal(i.operands[1].shape, a.shape)
1127: (4)             assert_equal(i.operands[1].dtype, np.dtype('f4'))
1128: (0)           def test_iter_allocate_output_buffered_readwrite():
1129: (4)             a = arange(6)
1130: (4)             i = nditer([a, None], ['buffered', 'delay_bufalloc'],
1131: (24)                           [['readonly'], ['allocate', 'readwrite']])
1132: (4)             with i:
1133: (8)               i.operands[1][:] = 1
1134: (8)               i.reset()
1135: (8)               for x in i:
1136: (12)                 x[1][...] += x[0][...]
1137: (8)                 assert_equal(i.operands[1], a+1)
1138: (0)           def test_iter_allocate_output_itorder():
1139: (4)             a = arange(6, dtype='i4').reshape(2, 3)
1140: (4)             i = nditer([a, None], [], [['readonly'], ['writeonly', 'allocate']],
1141: (24)                           op_dtypes=[None, np.dtype('f4')])
1142: (4)             assert_equal(i.operands[1].shape, a.shape)

```

```

1143: (4) assert_equal(i.operands[1].strides, a.strides)
1144: (4) assert_equal(i.operands[1].dtype, np.dtype('f4'))
1145: (4) a = arange(24, dtype='i4').reshape(2, 3, 4).T
1146: (4) i = nditer([a, None], [], [['readonly']], ['writeonly', 'allocate']),
1147: (24)                                     op_dtypes=[None, np.dtype('f4')])
1148: (4) assert_equal(i.operands[1].shape, a.shape)
1149: (4) assert_equal(i.operands[1].strides, a.strides)
1150: (4) assert_equal(i.operands[1].dtype, np.dtype('f4'))
1151: (4) a = arange(24, dtype='i4').reshape(2, 3, 4).swapaxes(0, 1)
1152: (4) i = nditer([a, None], [],
1153: (24)                                     [['readonly']], ['writeonly', 'allocate']),
1154: (24)                                     order='C',
1155: (24)                                     op_dtypes=[None, np.dtype('f4')])
1156: (4) assert_equal(i.operands[1].shape, a.shape)
1157: (4) assert_equal(i.operands[1].strides, (32, 16, 4))
1158: (4) assert_equal(i.operands[1].dtype, np.dtype('f4'))
1159: (0) def test_iter_allocate_output_opaxes():
1160: (4)     a = arange(24, dtype='i4').reshape(2, 3, 4)
1161: (4)     i = nditer([None, a], [], [['writeonly', 'allocate'], ['readonly']],
1162: (24)                                     op_dtypes=[np.dtype('u4'), None],
1163: (24)                                     op_axes=[[1, 2, 0], None])
1164: (4) assert_equal(i.operands[0].shape, (4, 2, 3))
1165: (4) assert_equal(i.operands[0].strides, (4, 48, 16))
1166: (4) assert_equal(i.operands[0].dtype, np.dtype('u4'))
1167: (0) def test_iter_allocate_output_types_promotion():
1168: (4)     i = nditer([array([3], dtype='f4'), array([0], dtype='f8'), None], [],
1169: (20)                                     [['readonly']] * 2 + [['writeonly', 'allocate']])
1170: (4)     assert_equal(i.dtypes[2], np.dtype('f8'))
1171: (4)     i = nditer([array([3], dtype='i4'), array([0], dtype='f4'), None], [],
1172: (20)                                     [['readonly']] * 2 + [['writeonly', 'allocate']])
1173: (4)     assert_equal(i.dtypes[2], np.dtype('f8'))
1174: (4)     i = nditer([array([3], dtype='f4'), array(0, dtype='f8'), None], [],
1175: (20)                                     [['readonly']] * 2 + [['writeonly', 'allocate']])
1176: (4)     assert_equal(i.dtypes[2], np.dtype('f4'))
1177: (4)     i = nditer([array([3], dtype='u4'), array(0, dtype='i4'), None], [],
1178: (20)                                     [['readonly']] * 2 + [['writeonly', 'allocate']])
1179: (4)     assert_equal(i.dtypes[2], np.dtype('u4'))
1180: (4)     i = nditer([array([3], dtype='u4'), array(-12, dtype='i4'), None], [],
1181: (20)                                     [['readonly']] * 2 + [['writeonly', 'allocate']])
1182: (4)     assert_equal(i.dtypes[2], np.dtype('i8'))
1183: (0) def test_iter_allocate_output_types_byte_order():
1184: (4)     a = array([3], dtype='u4').newbyteorder()
1185: (4)     i = nditer([a, None], [],
1186: (20)                                     [['readonly']], ['writeonly', 'allocate']))
1187: (4)     assert_equal(i.dtypes[0], i.dtypes[1])
1188: (4)     i = nditer([a, a, None], [],
1189: (20)                                     [['readonly']], ['readonly'], ['writeonly', 'allocate']))
1190: (4)     assert_(i.dtypes[0] != i.dtypes[2])
1191: (4)     assert_equal(i.dtypes[0].newbyteorder('='), i.dtypes[2])
1192: (0) def test_iter_allocate_output_types_scalar():
1193: (4)     i = nditer([None, 1, 2.3, np.float32(12), np.complex128(3)], [],
1194: (16)                                     [['writeonly', 'allocate']] + [['readonly']] * 4)
1195: (4)     assert_equal(i.operands[0].dtype, np.dtype('complex128'))
1196: (4)     assert_equal(i.operands[0].ndim, 0)
1197: (0) def test_iter_allocate_output_subtype():
1198: (4)     class MyNDArray(np.ndarray):
1199: (8)         __array_priority__ = 15
1200: (4)         a = np.array([[1, 2], [3, 4]]).view(MyNDArray)
1201: (4)         b = np.arange(4).reshape(2, 2).T
1202: (4)         i = nditer([a, b, None], [],
1203: (15)                                     [['readonly']], ['readonly'], ['writeonly', 'allocate']))
1204: (4)         assert_equal(type(a), type(i.operands[2]))
1205: (4)         assert_(type(b) is not type(i.operands[2]))
1206: (4)         assert_equal(i.operands[2].shape, (2, 2))
1207: (4)         i = nditer([a, b, None], [],
1208: (15)                                     [['readonly']], ['readonly'],
1209: (16)                                     ['writeonly', 'allocate', 'no_subtype']))
1210: (4)         assert_equal(type(b), type(i.operands[2]))
1211: (4)         assert_(type(a) is not type(i.operands[2]))

```

```

1212: (4)           assert_equal(i.operands[2].shape, (2, 2))
1213: (0)           def test_iter_allocate_output_errors():
1214: (4)             a = arange(6)
1215: (4)             assert_raises(TypeError, nditer, [a, None], [],
1216: (24)                   [ ['writeonly'], ['writeonly', 'allocate'] ])
1217: (4)             assert_raises(ValueError, nditer, [a, None], [],
1218: (24)                   [ ['readonly'], ['allocate', 'readonly'] ])
1219: (4)             assert_raises(ValueError, nditer, [a, None], ['buffered'],
1220: (44)                   [ 'allocate', 'readwrite' ])
1221: (4)             assert_raises(TypeError, nditer, [None, None], [],
1222: (24)                   [ ['writeonly', 'allocate'],
1223: (25)                   [ 'writeonly', 'allocate']],
1224: (24)                   op_dtypes=[None, np.dtype('f4')])
1225: (4)             a = arange(24, dtype='i4').reshape(2, 3, 4)
1226: (4)             assert_raises(ValueError, nditer, [a, None], [],
1227: (24)                   [ ['readonly'], ['writeonly', 'allocate']],
1228: (24)                   op_dtypes=[None, np.dtype('f4')],
1229: (24)                   op_axes=[None, [0, np.newaxis, 1]])
1230: (4)             assert_raises(ValueError, nditer, [a, None], [],
1231: (24)                   [ ['readonly'], ['writeonly', 'allocate']],
1232: (24)                   op_dtypes=[None, np.dtype('f4')],
1233: (24)                   op_axes=[None, [0, 3, 1]])
1234: (4)             assert_raises(ValueError, nditer, [a, None], [],
1235: (24)                   [ ['readonly'], ['writeonly', 'allocate']],
1236: (24)                   op_dtypes=[None, np.dtype('f4')],
1237: (24)                   op_axes=[None, [0, 2, 1, 0]])
1238: (4)             a = arange(24, dtype='i4').reshape(2, 3, 4)
1239: (4)             assert_raises(ValueError, nditer, [a, None], ["reduce_ok"],
1240: (24)                   [ ['readonly'], ['readwrite', 'allocate']],
1241: (24)                   op_dtypes=[None, np.dtype('f4')],
1242: (24)                   op_axes=[None, [0, np.newaxis, 2]])
1243: (0)           def test_all_allocated():
1244: (4)             i = np.nditer([None], op_dtypes=["int64"])
1245: (4)             assert i.operands[0].shape == ()
1246: (4)             assert i.dtypes == (np.dtype("int64"),)
1247: (4)             i = np.nditer([None], op_dtypes=["int64"], itershape=(2, 3, 4))
1248: (4)             assert i.operands[0].shape == (2, 3, 4)
1249: (0)           def test_iter_remove_axis():
1250: (4)             a = arange(24).reshape(2, 3, 4)
1251: (4)             i = nditer(a, ['multi_index'])
1252: (4)             i.remove_axis(1)
1253: (4)             assert_equal([x for x in i], a[:, 0,:].ravel())
1254: (4)             a = a[:-1,:,:]
1255: (4)             i = nditer(a, ['multi_index'])
1256: (4)             i.remove_axis(0)
1257: (4)             assert_equal([x for x in i], a[0,:,:].ravel())
1258: (0)           def test_iter_remove_multi_index_inner_loop():
1259: (4)             a = arange(24).reshape(2, 3, 4)
1260: (4)             i = nditer(a, ['multi_index'])
1261: (4)             assert_equal(i.ndim, 3)
1262: (4)             assert_equal(i.shape, (2, 3, 4))
1263: (4)             assert_equal(i.itviews[0].shape, (2, 3, 4))
1264: (4)             before = [x for x in i]
1265: (4)             i.remove_multi_index()
1266: (4)             after = [x for x in i]
1267: (4)             assert_equal(before, after)
1268: (4)             assert_equal(i.ndim, 1)
1269: (4)             assert_raises(ValueError, lambda i:i.shape, i)
1270: (4)             assert_equal(i.itviews[0].shape, (24,))
1271: (4)             i.reset()
1272: (4)             assert_equal(i.itersize, 24)
1273: (4)             assert_equal(i[0].shape, tuple())
1274: (4)             i.enable_external_loop()
1275: (4)             assert_equal(i.itersize, 24)
1276: (4)             assert_equal(i[0].shape, (24,))
1277: (4)             assert_equal(i.value, arange(24))
1278: (0)           def test_iter_iterindex():
1279: (4)             buffersize = 5
1280: (4)             a = arange(24).reshape(4, 3, 2)

```

```

1281: (4)          for flags in ([], ['buffered']):
1282: (8)            i = nditer(a, flags, buffersize=buffersize)
1283: (8)            assert_equal(iter_iterindices(i), list(range(24)))
1284: (8)            i.iterindex = 2
1285: (8)            assert_equal(iter_iterindices(i), list(range(2, 24)))
1286: (8)            i = nditer(a, flags, order='F', buffersize=buffersize)
1287: (8)            assert_equal(iter_iterindices(i), list(range(24)))
1288: (8)            i.iterindex = 5
1289: (8)            assert_equal(iter_iterindices(i), list(range(5, 24)))
1290: (8)            i = nditer(a[::-1], flags, order='F', buffersize=buffersize)
1291: (8)            assert_equal(iter_iterindices(i), list(range(24)))
1292: (8)            i.iterindex = 9
1293: (8)            assert_equal(iter_iterindices(i), list(range(9, 24)))
1294: (8)            i = nditer(a[::-1, ::-1], flags, order='C', buffersize=buffersize)
1295: (8)            assert_equal(iter_iterindices(i), list(range(24)))
1296: (8)            i.iterindex = 13
1297: (8)            assert_equal(iter_iterindices(i), list(range(13, 24)))
1298: (8)            i = nditer(a[::-1, ::-1], flags, buffersize=buffersize)
1299: (8)            assert_equal(iter_iterindices(i), list(range(24)))
1300: (8)            i.iterindex = 23
1301: (8)            assert_equal(iter_iterindices(i), list(range(23, 24)))
1302: (8)            i.reset()
1303: (8)            i.iterindex = 2
1304: (8)            assert_equal(iter_iterindices(i), list(range(2, 24)))
1305: (0)          def test_iter_iterrange():
1306: (4)            buffersize = 5
1307: (4)            a = arange(24, dtype='i4').reshape(4, 3, 2)
1308: (4)            a_fort = a.ravel(order='F')
1309: (4)            i = nditer(a, ['ranged'], ['readonly'], order='F',
1310: (16)              buffersize=buffersize)
1311: (4)            assert_equal(i.iterrange, (0, 24))
1312: (4)            assert_equal([x[()] for x in i], a_fort)
1313: (4)            for r in [(0, 24), (1, 2), (3, 24), (5, 5), (0, 20), (23, 24)]:
1314: (8)              i.iterrange = r
1315: (8)              assert_equal(i.iterrange, r)
1316: (8)              assert_equal([x[()] for x in i], a_fort[r[0]:r[1]])
1317: (4)            i = nditer(a, ['ranged', 'buffered'], ['readonly'], order='F',
1318: (16)              op_dtypes='f8', buffersize=buffersize)
1319: (4)            assert_equal(i.iterrange, (0, 24))
1320: (4)            assert_equal([x[()] for x in i], a_fort)
1321: (4)            for r in [(0, 24), (1, 2), (3, 24), (5, 5), (0, 20), (23, 24)]:
1322: (8)              i.iterrange = r
1323: (8)              assert_equal(i.iterrange, r)
1324: (8)              assert_equal([x[()] for x in i], a_fort[r[0]:r[1]])
1325: (4)          def get_array(i):
1326: (8)            val = np.array([], dtype='f8')
1327: (8)            for x in i:
1328: (12)              val = np.concatenate((val, x))
1329: (8)            return val
1330: (4)          i = nditer(a, ['ranged', 'buffered', 'external_loop'],
1331: (16)            ['readonly'], order='F',
1332: (16)              op_dtypes='f8', buffersize=buffersize)
1333: (4)          assert_equal(i.iterrange, (0, 24))
1334: (4)          assert_equal(get_array(i), a_fort)
1335: (4)          for r in [(0, 24), (1, 2), (3, 24), (5, 5), (0, 20), (23, 24)]:
1336: (8)            i.iterrange = r
1337: (8)            assert_equal(i.iterrange, r)
1338: (8)            assert_equal(get_array(i), a_fort[r[0]:r[1]])
1339: (0)          def test_iter_buffering():
1340: (4)            arrays = []
1341: (4)            arrays.append(np.arange(24,
1342: (20)              dtype='c16').reshape(2, 3, 4).T.newbyteorder().byteswap())
1343: (4)            arrays.append(np.arange(10, dtype='f4'))
1344: (4)            a = np.zeros((4*16+1,), dtype='i1')[1:]
1345: (4)            a.dtype = 'i4'
1346: (4)            a[:] = np.arange(16, dtype='i4')
1347: (4)            arrays.append(a)
1348: (4)            arrays.append(np.arange(120, dtype='i4').reshape(5, 3, 2, 4).T)
1349: (4)            for a in arrays:

```

```

1350: (8)
1351: (12)
1352: (12)
1353: (27)
1354: (27)
1355: (27)
1356: (27)
1357: (12)
1358: (16)
1359: (16)
1360: (16)
1361: (12)
1362: (0)
1363: (4)
1364: (4)
1365: (19)
1366: (19)
1367: (19)
1368: (19)
1369: (4)
1370: (4)
1371: (8)
1372: (12)
1373: (12)
1374: (12)
1375: (4)
1376: (0)
1377: (4)
1378: (4)
1379: (4)
'reduce_ok'],
1380: (20)
1381: (20)
1382: (20)
1383: (4)
1384: (4)
1385: (4)
1386: (4)
1387: (4)
1388: (8)
1389: (4)
1390: (4)
1391: (4)
1392: (4)
1393: (4)
1394: (8)
1395: (8)
1396: (8)
1397: (8)
[[x[0][()], x[1][()]] for x in i], list(zip(range(6),
[1]*6)))
1398: (0)
1399: (4)
1400: (4)
1401: (19)
1402: (19)
1403: (19)
1404: (19)
1405: (4)
1406: (8)
1407: (12)
1408: (4)
1409: (0)
1410: (4)
1411: (4)
1412: (19)
1413: (19)
1414: (19)
1415: (19)
1416: (4)

        for buffersize in (1, 2, 3, 5, 8, 11, 16, 1024):
            vals = []
            i = nditer(a, ['buffered', 'external_loop'],
                       [['readonly', 'nbo', 'aligned']],
                       order='C',
                       casting='equiv',
                       buffersize=buffersize)
            while not i.finished:
                assert_(i[0].size <= buffersize)
                vals.append(i[0].copy())
                i.iternext()
            assert_equal(np.concatenate(vals), a.ravel(order='C'))
def test_iter_write_buffering():
    a = np.arange(24).reshape(2, 3, 4).T.newbyteorder().byteswap()
    i = nditer(a, ['buffered'],
               [['readwrite', 'nbo', 'aligned']],
               casting='equiv',
               order='C',
               buffersize=16)
    x = 0
    with i:
        while not i.finished:
            i[0] = x
            x += 1
            i.iternext()
    assert_equal(a.ravel(order='C'), np.arange(24))
def test_iter_buffering_delayed_alloc():
    a = np.arange(6)
    b = np.arange(1, dtype='f4')
    i = nditer([a, b], ['buffered', 'delay_bufalloc', 'multi_index',
                        'reduce_ok'],
               ['readwrite'],
               casting='unsafe',
               op_dtypes='f4')
    assert_(i.has_delayed_bufalloc)
    assert_raises(ValueError, lambda i:i.multi_index, i)
    assert_raises(ValueError, lambda i:i[0], i)
    assert_raises(ValueError, lambda i:i[0:2], i)
    def assign_iter(i):
        i[0] = 0
    assert_raises(ValueError, assign_iter, i)
    i.reset()
    assert_(not i.has_delayed_bufalloc)
    assert_equal(i.multi_index, (0,))
    with i:
        assert_equal(i[0], 0)
        i[1] = 1
        assert_equal(i[0:2], [0, 1])
        assert_equal([[x[0][()], x[1][()]] for x in i], list(zip(range(6),
[1]*6)))
def test_iter_buffered_cast_simple():
    a = np.arange(10, dtype='f4')
    i = nditer(a, ['buffered', 'external_loop'],
               [['readwrite', 'nbo', 'aligned']],
               casting='same_kind',
               op_dtypes=[np.dtype('f8')],
               buffersize=3)
    with i:
        for v in i:
            v[...] *= 2
    assert_equal(a, 2*np.arange(10, dtype='f4'))
def test_iter_buffered_cast_byteswapped():
    a = np.arange(10, dtype='f4').newbyteorder().byteswap()
    i = nditer(a, ['buffered', 'external_loop'],
               [['readwrite', 'nbo', 'aligned']],
               casting='same_kind',
               op_dtypes=[np.dtype('f8').newbyteorder()],
               buffersize=3)
    with i:

```

```

1417: (8)
1418: (12)
1419: (4)
1420: (4)
1421: (8)
1422: (8)
1423: (8)
1424: (23)
1425: (23)
1426: (23)
1427: (23)
1428: (8)
1429: (12)
1430: (16)
1431: (8)
1432: (0)
1433: (4)
1434: (4)
1435: (4)
1436: (19)
1437: (19)
1438: (19)
1439: (19)
1440: (4)
1441: (8)
1442: (12)
1443: (4)
1444: (4)
1445: (4)
1446: (4)
1447: (19)
1448: (19)
1449: (19)
1450: (19)
1451: (4)
1452: (8)
1453: (12)
1454: (4)
1455: (4)
1456: (4)
1457: (4)
1458: (19)
1459: (19)
1460: (19)
1461: (19)
1462: (4)
1463: (8)
1464: (12)
1465: (4)
1466: (4)
1467: (4)
1468: (19)
1469: (19)
1470: (19)
1471: (19)
1472: (4)
1473: (8)
1474: (12)
1475: (4)
1476: (0)
1477: (4)
1478: (4)
1479: (4)
1480: (20)
1481: (20)
1482: (4)
1483: (4)
1484: (4)
1485: (4)

        for v in i:
            v[...] *= 2
    assert_equal(a, 2*np.arange(10, dtype='f4'))
    with suppress_warnings() as sup:
        sup.filter(np.ComplexWarning)
        a = np.arange(10, dtype='f8').newbyteorder().byteswap()
        i = nditer(a, ['buffered', 'external_loop'],
                   [['readwrite', 'nbo', 'aligned']],
                   casting='unsafe',
                   op_dtypes=[np.dtype('c8').newbyteorder()],
                   buffersize=3)
        with i:
            for v in i:
                v[...] *= 2
    assert_equal(a, 2*np.arange(10, dtype='f8'))
def test_iter_buffered_cast_byteswapped_complex():
    a = np.arange(10, dtype='c8').newbyteorder().byteswap()
    a += 2j
    i = nditer(a, ['buffered', 'external_loop'],
               [['readwrite', 'nbo', 'aligned']],
               casting='same_kind',
               op_dtypes=[np.dtype('c16')],
               buffersize=3)
    with i:
        for v in i:
            v[...] *= 2
    assert_equal(a, 2*np.arange(10, dtype='c8') + 4j)
    a = np.arange(10, dtype='c8')
    a += 2j
    i = nditer(a, ['buffered', 'external_loop'],
               [['readwrite', 'nbo', 'aligned']],
               casting='same_kind',
               op_dtypes=[np.dtype('c16').newbyteorder()],
               buffersize=3)
    with i:
        for v in i:
            v[...] *= 2
    assert_equal(a, 2*np.arange(10, dtype='c8') + 4j)
    a = np.arange(10, dtype=np.clongdouble).newbyteorder().byteswap()
    a += 2j
    i = nditer(a, ['buffered', 'external_loop'],
               [['readwrite', 'nbo', 'aligned']],
               casting='same_kind',
               op_dtypes=[np.dtype('c16')],
               buffersize=3)
    with i:
        for v in i:
            v[...] *= 2
    assert_equal(a, 2*np.arange(10, dtype=np.clongdouble) + 4j)
    a = np.arange(10, dtype=np.longdouble).newbyteorder().byteswap()
    i = nditer(a, ['buffered', 'external_loop'],
               [['readwrite', 'nbo', 'aligned']],
               casting='same_kind',
               op_dtypes=[np.dtype('f4')],
               buffersize=7)
    with i:
        for v in i:
            v[...] *= 2
    assert_equal(a, 2*np.arange(10, dtype=np.longdouble))
def test_iter_buffered_cast_structured_type():
    sdt = [('a', 'f4'), ('b', 'i8'), ('c', 'c8', (2, 3)), ('d', 'O')]
    a = np.arange(3, dtype='f4') + 0.5
    i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
               casting='unsafe',
               op_dtypes=sdt)
    vals = [np.array(x) for x in i]
    assert_equal(vals[0]['a'], 0.5)
    assert_equal(vals[0]['b'], 0)
    assert_equal(vals[0]['c'], [(0.5)]*3)*2

```

```

1486: (4) assert_equal(vals[0]['d'], 0.5)
1487: (4) assert_equal(vals[1]['a'], 1.5)
1488: (4) assert_equal(vals[1]['b'], 1)
1489: (4) assert_equal(vals[1]['c'], [[[1.5]]*3]*2)
1490: (4) assert_equal(vals[1]['d'], 1.5)
1491: (4) assert_equal(vals[0].dtype, np.dtype(sdt))
1492: (4) sdt = [('a', 'f4'), ('b', 'i8'), ('c', 'c8', (2, 3)), ('d', 'O')]
1493: (4) a = np.zeros((3,), dtype='O')
1494: (4) a[0] = (0.5, 0.5, [0.5, 0.5, 0.5], [0.5, 0.5, 0.5]), 0.5)
1495: (4) a[1] = (1.5, 1.5, [[1.5, 1.5, 1.5], [1.5, 1.5, 1.5]], 1.5)
1496: (4) a[2] = (2.5, 2.5, [[2.5, 2.5, 2.5], [2.5, 2.5, 2.5]], 2.5)
1497: (4) if HAS_REFCOUNT:
1498: (8)     rc = sys.getrefcount(a[0])
1499: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1500: (20)             casting='unsafe',
1501: (20)             op_dtypes=sdt)
1502: (4) vals = [x.copy() for x in i]
1503: (4) assert_equal(vals[0]['a'], 0.5)
1504: (4) assert_equal(vals[0]['b'], 0)
1505: (4) assert_equal(vals[0]['c'], [[[0.5]]*3]*2)
1506: (4) assert_equal(vals[0]['d'], 0.5)
1507: (4) assert_equal(vals[1]['a'], 1.5)
1508: (4) assert_equal(vals[1]['b'], 1)
1509: (4) assert_equal(vals[1]['c'], [[[1.5]]*3]*2)
1510: (4) assert_equal(vals[1]['d'], 1.5)
1511: (4) assert_equal(vals[0].dtype, np.dtype(sdt))
1512: (4) vals, i, x = [None]*3
1513: (4) if HAS_REFCOUNT:
1514: (8)     assert_equal(sys.getrefcount(a[0]), rc)
1515: (4) sdt = [('a', 'f4')]
1516: (4) a = np.array([(5.5,), (8,)], dtype=sdt)
1517: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1518: (20)             casting='unsafe',
1519: (20)             op_dtypes='i4')
1520: (4) assert_equal([x_[()] for x_ in i], [5, 8])
1521: (4) sdt = [('a', 'f4'), ('b', 'i8'), ('d', 'O')]
1522: (4) a = np.array([(5.5, 7, 'test'), (8, 10, 11)], dtype=sdt)
1523: (4) assert_raises(TypeError, lambda: (
1524: (8)     nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1525: (15)         casting='unsafe',
1526: (15)         op_dtypes='i4')))
1527: (4) sdt1 = [('a', 'f4'), ('b', 'i8'), ('d', 'O')]
1528: (4) sdt2 = [('d', 'u2'), ('a', 'O'), ('b', 'f8')]
1529: (4) a = np.array([(1, 2, 3), (4, 5, 6)], dtype=sdt1)
1530: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1531: (20)             casting='unsafe',
1532: (20)             op_dtypes=sdt2)
1533: (4) assert_equal(i[0].dtype, np.dtype(sdt2))
1534: (4) assert_equal([np.array(x_) for x_ in i],
1535: (17)             [np.array((1, 2, 3), dtype=sdt2),
1536: (18)               np.array((4, 5, 6), dtype=sdt2)])
1537: (0) def test_iter_buffered_cast_structured_type_failure_with_cleanup():
1538: (4)     sdt1 = [('a', 'f4'), ('b', 'i8'), ('d', 'O')]
1539: (4)     sdt2 = [('b', 'O'), ('a', 'f8')]
1540: (4)     a = np.array([(1, 2, 3), (4, 5, 6)], dtype=sdt1)
1541: (4)     for intent in ["readwrite", "readonly", "writeonly"]:
1542: (8)         simple_arr = np.array([1, 2], dtype="i,i") # requires clean up
1543: (8)         with pytest.raises(TypeError):
1544: (12)             nditer((simple_arr, a), ['buffered', 'refs_ok'], [intent, intent],
1545: (19)                 casting='unsafe', op_dtypes=['f,f', sdt2])
1546: (0) def test_buffered_cast_error_paths():
1547: (4)     with pytest.raises(ValueError):
1548: (8)         np.nditer((np.array("a", dtype="S1"),), op_dtypes=["i"],
1549: (18)             casting="unsafe", flags=["buffered"])
1550: (4)         it = np.nditer((np.array(1, dtype="i"),), op_dtypes=["S1"],
1551: (19)             op_flags=["writeonly"], casting="unsafe", flags=
1552: ("buffered"))
1553: (4)         with pytest.raises(ValueError):
1553: (8)             with it:

```

```

1554: (12)           buf = next(it)
1555: (12)           buf[...] = "a" # cannot be converted to int.
1556: (0)            @pytest.mark.skipif(IS_WASM, reason="Cannot start subprocess")
1557: (0)            @pytest.mark.skipif(not HAS_REFCOUNT, reason="PyPy seems to not hit this.")
1558: (0)            def test_buffered_cast_error_paths_unraisable():
1559: (4)              code = textwrap.dedent("""
1560: (8)                  import numpy as np
1561: (8)                  it = np.nditer((np.array(1, dtype="i"),), op_dtypes=["S1"],
1562: (23)                                op_flags=["writeonly"], casting="unsafe", flags=
1563: ["buffered"])
1564: (8)                  buf = next(it)
1565: (8)                  buf[...] = "a"
1566: (8)                  del buf, it # Flushing only happens during deallocate right now.
1567: (4)                  """
1568: (34)                  res = subprocess.check_output([sys.executable, "-c", code],
1569:   stderr=subprocess.STDOUT, text=True)
1570: (0)                  assert "ValueError" in res
1571: (4)            def test_iter_buffered_cast_subarray():
1572: (4)              sdt1 = [('a', 'f4')]
1573: (4)              sdt2 = [('a', 'f8', (3, 2, 2))]
1574: (4)              a = np.zeros((6,), dtype=sdt1)
1575: (4)              a['a'] = np.arange(6)
1576: (20)             i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1577:                               casting='unsafe',
1578:                               op_dtypes=sdt2)
1579: (4)             assert_equal(i[0].dtype, np.dtype(sdt2))
1580: (8)             for x, count in zip(i, list(range(6))):
1581: (4)                 assert_(np.all(x['a'] == count))
1582: (4)             sdt1 = [('a', 'O', (1, 1))]
1583: (4)             sdt2 = [('a', 'O', (3, 2, 2))]
1584: (4)             a = np.zeros((6,), dtype=sdt1)
1585: (4)             a['a'][:, 0, 0] = np.arange(6)
1586: (20)             i = nditer(a, ['buffered', 'refs_ok'], ['readwrite'],
1587:                               casting='unsafe',
1588:                               op_dtypes=sdt2)
1589: (4)             with i:
1590: (8)                 assert_equal(i[0].dtype, np.dtype(sdt2))
1591: (8)                 count = 0
1592: (12)                for x in i:
1593: (12)                    assert_(np.all(x['a'] == count))
1594: (12)                    x['a'][0] += 2
1595: (4)                    count += 1
1596: (4)                    assert_equal(a['a'], np.arange(6).reshape(6, 1, 1)+2)
1597: (4)                    sdt1 = [('a', 'O', (3, 2, 2))]
1598: (4)                    sdt2 = [('a', 'O', (1,))]
1599: (4)                    a = np.zeros((6,), dtype=sdt1)
1600: (4)                    a['a'][:, 0, 0, 0] = np.arange(6)
1601: (20)                   i = nditer(a, ['buffered', 'refs_ok'], ['readwrite'],
1602:                                     casting='unsafe',
1603:                                     op_dtypes=sdt2)
1604: (4)                   with i:
1605: (8)                     assert_equal(i[0].dtype, np.dtype(sdt2))
1606: (8)                     count = 0
1607: (12)                    for x in i:
1608: (12)                        assert_equal(x['a'], count)
1609: (12)                        x['a'] += 2
1610: (4)                        count += 1
1611: (4)                        assert_equal(a['a'], np.arange(6).reshape(6, 1, 1, 1)*np.ones((1, 3, 2,
1612: (2)))+2)
1613: (4)                        sdt1 = [('a', 'f8', (3, 2, 2))]
1614: (4)                        sdt2 = [('a', 'O', (1,))]
1615: (4)                        a = np.zeros((6,), dtype=sdt1)
1616: (20)                         a['a'][:, 0, 0, 0] = np.arange(6)
1617: (20)                         i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1618:                               casting='unsafe',
1619:                               op_dtypes=sdt2)
1620: (4)                         assert_equal(i[0].dtype, np.dtype(sdt2))
1621: (4)                         count = 0
1622: (4)                         for x in i:

```

```

1621: (8) assert_equal(x['a'], count)
1622: (8) count += 1
1623: (4) sdt1 = [('a', 'O', (3, 2, 2))]
1624: (4) sdt2 = [('a', 'f4', (1,))]
1625: (4) a = np.zeros((6,), dtype=sdt1)
1626: (4) a['a'][ :, 0, 0, 0] = np.arange(6)
1627: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1628: (20) casting='unsafe',
1629: (20) op_dtypes=sdt2)
1630: (4) assert_equal(i[0].dtype, np.dtype(sdt2))
1631: (4) count = 0
1632: (4) for x in i:
1633: (8)     assert_equal(x['a'], count)
1634: (8)     count += 1
1635: (4) sdt1 = [('a', 'O', (3, 2, 2))]
1636: (4) sdt2 = [('a', 'f4', (3, 2, 2))]
1637: (4) a = np.zeros((6,), dtype=sdt1)
1638: (4) a['a'] = np.arange(6*3*2*2).reshape(6, 3, 2, 2)
1639: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1640: (20) casting='unsafe',
1641: (20) op_dtypes=sdt2)
1642: (4) assert_equal(i[0].dtype, np.dtype(sdt2))
1643: (4) count = 0
1644: (4) for x in i:
1645: (8)     assert_equal(x['a'], a[count]['a'])
1646: (8)     count += 1
1647: (4) sdt1 = [('a', 'f8', (6,))]
1648: (4) sdt2 = [('a', 'f4', (2,))]
1649: (4) a = np.zeros((6,), dtype=sdt1)
1650: (4) a['a'] = np.arange(6*6).reshape(6, 6)
1651: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1652: (20) casting='unsafe',
1653: (20) op_dtypes=sdt2)
1654: (4) assert_equal(i[0].dtype, np.dtype(sdt2))
1655: (4) count = 0
1656: (4) for x in i:
1657: (8)     assert_equal(x['a'], a[count]['a'][:2])
1658: (8)     count += 1
1659: (4) sdt1 = [('a', 'f8', (2,))]
1660: (4) sdt2 = [('a', 'f4', (6,))]
1661: (4) a = np.zeros((6,), dtype=sdt1)
1662: (4) a['a'] = np.arange(6*2).reshape(6, 2)
1663: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1664: (20) casting='unsafe',
1665: (20) op_dtypes=sdt2)
1666: (4) assert_equal(i[0].dtype, np.dtype(sdt2))
1667: (4) count = 0
1668: (4) for x in i:
1669: (8)     assert_equal(x['a'][:2], a[count]['a'])
1670: (8)     assert_equal(x['a'][2:], [0, 0, 0, 0])
1671: (8)     count += 1
1672: (4) sdt1 = [('a', 'f8', (2,))]
1673: (4) sdt2 = [('a', 'f4', (2, 2))]
1674: (4) a = np.zeros((6,), dtype=sdt1)
1675: (4) a['a'] = np.arange(6*2).reshape(6, 2)
1676: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1677: (20) casting='unsafe',
1678: (20) op_dtypes=sdt2)
1679: (4) assert_equal(i[0].dtype, np.dtype(sdt2))
1680: (4) count = 0
1681: (4) for x in i:
1682: (8)     assert_equal(x['a'][0], a[count]['a'])
1683: (8)     assert_equal(x['a'][1], a[count]['a'])
1684: (8)     count += 1
1685: (4) sdt1 = [('a', 'f8', (2, 1))]
1686: (4) sdt2 = [('a', 'f4', (3, 2))]
1687: (4) a = np.zeros((6,), dtype=sdt1)
1688: (4) a['a'] = np.arange(6*2).reshape(6, 2, 1)
1689: (4) i = nditer(a, ['buffered', 'refs_ok'], ['readonly']),

```

```

1690: (20)                         casting='unsafe',
1691: (20)                         op_dtypes=sdt2)
1692: (4)                          assert_equal(i[0].dtype, np.dtype(sdt2))
1693: (4)                          count = 0
1694: (4)                          for x in i:
1695: (8)                            assert_equal(x['a'][::2, 0], a[count]['a'][::, 0])
1696: (8)                            assert_equal(x['a'][::2, 1], a[count]['a'][::, 1])
1697: (8)                            assert_equal(x['a'][2,:], [0, 0])
1698: (8)                            count += 1
1699: (4)                            sdt1 = [('a', 'f8', (2, 3))]
1700: (4)                            sdt2 = [('a', 'f4', (3, 2))]
1701: (4)                            a = np.zeros((6,), dtype=sdt1)
1702: (4)                            a['a'] = np.arange(6*2*3).reshape(6, 2, 3)
1703: (4)                            i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
1704: (20)                                casting='unsafe',
1705: (20)                                op_dtypes=sdt2)
1706: (4)                          assert_equal(i[0].dtype, np.dtype(sdt2))
1707: (4)                          count = 0
1708: (4)                          for x in i:
1709: (8)                            assert_equal(x['a'][::2, 0], a[count]['a'][::, 0])
1710: (8)                            assert_equal(x['a'][::2, 1], a[count]['a'][::, 1])
1711: (8)                            assert_equal(x['a'][2,:], [0, 0])
1712: (8)                            count += 1
1713: (0)  def test_iter_buffering_badwriteback():
1714: (4)    a = np.arange(6).reshape(2, 3, 1)
1715: (4)    b = np.arange(12).reshape(2, 3, 2)
1716: (4)    assert_raises(ValueError, nditer, [a, b],
1717: (18)        ['buffered', 'external_loop'],
1718: (18)        [['readwrite'], ['writeonly']],
1719: (18)        order='C')
1720: (4)    nditer([a, b], ['buffered', 'external_loop'],
1721: (11)        [['readonly'], ['writeonly']],
1722: (11)        order='C')
1723: (4)    a = np.arange(1).reshape(1, 1, 1)
1724: (4)    nditer([a, b], ['buffered', 'external_loop', 'reduce_ok'],
1725: (11)        [['readwrite'], ['writeonly']],
1726: (11)        order='C')
1727: (4)    a = np.arange(6).reshape(1, 3, 2)
1728: (4)    assert_raises(ValueError, nditer, [a, b],
1729: (18)        ['buffered', 'external_loop'],
1730: (18)        [['readwrite'], ['writeonly']],
1731: (18)        order='C')
1732: (4)    a = np.arange(4).reshape(2, 1, 2)
1733: (4)    assert_raises(ValueError, nditer, [a, b],
1734: (18)        ['buffered', 'external_loop'],
1735: (18)        [['readwrite'], ['writeonly']],
1736: (18)        order='C')
1737: (0)  def test_iter_buffering_string():
1738: (4)    a = np.array(['abc', 'a', 'abcd'], dtype=np.bytes_)
1739: (4)    assert_equal(a.dtype, np.dtype('S4'))
1740: (4)    assert_raises(TypeError, nditer, a, ['buffered'], ['readonly'],
1741: (18)        op_dtypes='S2')
1742: (4)    i = nditer(a, ['buffered'], ['readonly'], op_dtypes='S6')
1743: (4)    assert_equal(i[0], b'abc')
1744: (4)    assert_equal(i[0].dtype, np.dtype('S6'))
1745: (4)    a = np.array(['abc', 'a', 'abcd'], dtype=np.str_)
1746: (4)    assert_equal(a.dtype, np.dtype('U4'))
1747: (4)    assert_raises(TypeError, nditer, a, ['buffered'], ['readonly'],
1748: (20)        op_dtypes='U2')
1749: (4)    i = nditer(a, ['buffered'], ['readonly'], op_dtypes='U6')
1750: (4)    assert_equal(i[0], 'abc')
1751: (4)    assert_equal(i[0].dtype, np.dtype('U6'))
1752: (0)  def test_iter_buffering_growinner():
1753: (4)    a = np.arange(30)
1754: (4)    i = nditer(a, ['buffered', 'growinner', 'external_loop'],
1755: (27)                                buffersize=5)
1756: (4)    assert_equal(i[0].size, a.size)
1757: (0)    @pytest.mark.slow
1758: (0)    def test_iter_buffered_reduce_reuse():

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1759: (4)
1760: (4)
'refs_ok']
1761: (4)
1762: (4)
1763: (4)
1764: (4)
1765: (8)
1766: (12)
1767: (16)
1768: (20)
1769: (20)
strides)
1770: (20)
1771: (24)
1772: (4)
1773: (8)
1774: (28)
1775: (28)
1776: (8)
1777: (12)
1778: (12)
1779: (12)
1780: (12)
1781: (16)
1782: (12)
1783: (8)
1784: (12)
1785: (32)
op_flags=op_flags,
1786: (32)
1787: (12)
1788: (16)
1789: (16)
1790: (16)
1791: (16)
1792: (20)
1793: (16)
1794: (12)
1795: (0)
def test_iter_no_broadcast():
1796: (4)
1797: (4)
1798: (4)
1799: (4)
1800: (11)
1801: (12)
1802: (4)
1803: (18)
1804: (4)
1805: (18)
1806: (0)
1807: (4)
1808: (8)
1809: (8)
1810: (8)
1811: (8)
1812: (8)
1813: (8)
1814: (8)
1815: (8)
1816: (8)
1817: (8)
1818: (4)
def test_basic(self):
1819: (8)
1820: (8)
1821: (8)
1822: (8)
1823: (8)
1824: (8)
    a = np.arange(2*3**5)[3**5:3**5+1]
    flags = ['buffered', 'delay_bufalloc', 'multi_index', 'reduce_ok',
              op_flags = [('readonly'), ('readwrite', 'allocate')]
              op_axes_list = [[(0, 1, 2), (0, 1, -1)], [(0, 1, 2), (0, -1, -1)]]
              op_dtypes = [float, a.dtype]
              def get_params():
                  for xs in range(-3**2, 3**2 + 1):
                      for ys in range(xs, 3**2 + 1):
                          for op_axes in op_axes_list:
                              strides = (xs * a.itemsize, ys * a.itemsize, a.itemsize)
                              arr = np.lib.stride_tricks.as_strided(a, (3, 3, 3),
   strides)
                              for skip in [0, 1]:
                                  yield arr, op_axes, skip
              for arr, op_axes, skip in get_params():
                  nditer2 = np.nditer([arr.copy(), None],
                                      op_axes=op_axes, flags=flags, op_flags=op_flags,
                                      op_dtypes=op_dtypes)
                  with nditer2:
                      nditer2.operands[-1][...] = 0
                      nditer2.reset()
                      nditer2.iterindex = skip
                      for (a2_in, b2_in) in nditer2:
                          b2_in += a2_in.astype(np.int_)
                      comp_res = nditer2.operands[-1]
                  for bufsize in range(0, 3**3):
                      nditer1 = np.nditer([arr, None],
  op_axes=op_axes, flags=flags,
  buffersize=bufsize, op_dtypes=op_dtypes)
                      with nditer1:
                          nditer1.operands[-1][...] = 0
                          nditer1.reset()
                          nditer1.iterindex = skip
                          for (a1_in, b1_in) in nditer1:
                              b1_in += a1_in.astype(np.int_)
                          res = nditer1.operands[-1]
                          assert_array_equal(res, comp_res)
      def test_iter_no_broadcast():
          a = np.arange(24).reshape(2, 3, 4)
          b = np.arange(6).reshape(2, 3, 1)
          c = np.arange(12).reshape(3, 4)
          nditer([a, b, c], [],
                 [['readonly', 'no_broadcast'],
                  ['readonly'], ['readonly']])
          assert_raises(ValueError, nditer, [a, b, c], [],
                        [['readonly'], ['readonly', 'no_broadcast'], ['readonly']])
          assert_raises(ValueError, nditer, [a, b, c], [],
                        [['readonly'], ['readonly'], ['readonly', 'no_broadcast']])
      class TestIterNested:
          def test_basic(self):
              a = arange(12).reshape(2, 3, 2)
              i, j = np.nested_iters(a, [[0], [1, 2]])
              vals = [list(j) for _ in i]
              assert_equal(vals, [[0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11]])
              i, j = np.nested_iters(a, [[0, 1], [2]])
              vals = [list(j) for _ in i]
              assert_equal(vals, [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10, 11]])
              i, j = np.nested_iters(a, [[0, 2], [1]])
              vals = [list(j) for _ in i]
              assert_equal(vals, [[0, 2, 4], [1, 3, 5], [6, 8, 10], [7, 9, 11]])
          def test_reorder(self):
              a = arange(12).reshape(2, 3, 2)
              i, j = np.nested_iters(a, [[0], [2, 1]])
              vals = [list(j) for _ in i]
              assert_equal(vals, [[0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11]])
              i, j = np.nested_iters(a, [[1, 0], [2]])
              vals = [list(j) for _ in i]

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1825: (8) assert_equal(vals, [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10, 11]])
1826: (8) i, j = np.nested_iters(a, [[2, 0], [1]])
1827: (8) vals = [list(j) for _ in i]
1828: (8) assert_equal(vals, [[0, 2, 4], [1, 3, 5], [6, 8, 10], [7, 9, 11]])
1829: (8) i, j = np.nested_iters(a, [[0], [2, 1]], order='C')
1830: (8) vals = [list(j) for _ in i]
1831: (8) assert_equal(vals, [[0, 2, 4, 1, 3, 5], [6, 8, 10, 7, 9, 11]])
1832: (8) i, j = np.nested_iters(a, [[1, 0], [2]], order='C')
1833: (8) vals = [list(j) for _ in i]
1834: (8) assert_equal(vals, [[0, 1], [6, 7], [2, 3], [8, 9], [4, 5], [10, 11]])
1835: (8) i, j = np.nested_iters(a, [[2, 0], [1]], order='C')
1836: (8) vals = [list(j) for _ in i]
1837: (8) assert_equal(vals, [[0, 2, 4], [6, 8, 10], [1, 3, 5], [7, 9, 11]])
1838: (4) def test_flip_axes(self):
1839: (8) a = arange(12).reshape(2, 3, 2)[::-1, ::-1, ::-1]
1840: (8) i, j = np.nested_iters(a, [[0], [1, 2]])
1841: (8) vals = [list(j) for _ in i]
1842: (8) assert_equal(vals, [[0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11]])
1843: (8) i, j = np.nested_iters(a, [[0, 1], [2]])
1844: (8) vals = [list(j) for _ in i]
1845: (8) assert_equal(vals, [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10, 11]])
1846: (8) i, j = np.nested_iters(a, [[0, 2], [1]])
1847: (8) vals = [list(j) for _ in i]
1848: (8) assert_equal(vals, [[0, 2, 4], [1, 3, 5], [6, 8, 10], [7, 9, 11]])
1849: (8) i, j = np.nested_iters(a, [[0], [1, 2]], order='C')
1850: (8) vals = [list(j) for _ in i]
1851: (8) assert_equal(vals, [[11, 10, 9, 8, 7, 6], [5, 4, 3, 2, 1, 0]])
1852: (8) i, j = np.nested_iters(a, [[0, 1], [2]], order='C')
1853: (8) vals = [list(j) for _ in i]
1854: (8) assert_equal(vals, [[11, 10], [9, 8], [7, 6], [5, 4], [3, 2], [1, 0]])
1855: (8) i, j = np.nested_iters(a, [[0, 2], [1]], order='C')
1856: (8) vals = [list(j) for _ in i]
1857: (8) assert_equal(vals, [[11, 9, 7], [10, 8, 6], [5, 3, 1], [4, 2, 0]])
1858: (4) def test_broadcast(self):
1859: (8) a = arange(2).reshape(2, 1)
1860: (8) b = arange(3).reshape(1, 3)
1861: (8) i, j = np.nested_iters([a, b], [[0], [1]])
1862: (8) vals = [list(j) for _ in i]
1863: (8) assert_equal(vals, [[[0, 0], [0, 1], [0, 2]], [[1, 0], [1, 1], [1, 2]]])
1864: (8) i, j = np.nested_iters([a, b], [[1], [0]])
1865: (8) vals = [list(j) for _ in i]
1866: (8) assert_equal(vals, [[[0, 0], [1, 0]], [[0, 1], [1, 1]], [[0, 2], [1, 2]]])
1867: (4) def test_dtype_copy(self):
1868: (8) a = arange(6, dtype='i4').reshape(2, 3)
1869: (8) i, j = np.nested_iters(a, [[0], [1]],
1870: (28) op_flags=['readonly', 'copy'],
1871: (28) op_dtypes='f8')
1872: (8) assert_equal(j[0].dtype, np.dtype('f8'))
1873: (8) vals = [list(j) for _ in i]
1874: (8) assert_equal(vals, [[0, 1, 2], [3, 4, 5]])
1875: (8) vals = None
1876: (8) a = arange(6, dtype='f4').reshape(2, 3)
1877: (8) i, j = np.nested_iters(a, [[0], [1]],
1878: (28) op_flags=['readwrite', 'updateifcopy'],
1879: (28) casting='same_kind',
1880: (28) op_dtypes='f8')
1881: (8) with i, j:
1882: (12)     assert_equal(j[0].dtype, np.dtype('f8'))
1883: (12)     for x in i:
1884: (16)         for y in j:
1885: (20)             y [...] += 1
1886: (12)             assert_equal(a, [[0, 1, 2], [3, 4, 5]])
1887: (8)             assert_equal(a, [[1, 2, 3], [4, 5, 6]])
1888: (8)             a = arange(6, dtype='f4').reshape(2, 3)
1889: (8)             i, j = np.nested_iters(a, [[0], [1]],
1890: (28)                 op_flags=['readwrite', 'updateifcopy'],
1891: (28)                 casting='same_kind',

```

```

1892: (28)                                op_dtypes='f8')
1893: (8)       assert_equal(j[0].dtype, np.dtype('f8'))
1894: (8)       for x in i:
1895: (12)         for y in j:
1896: (16)           y[...] += 1
1897: (8)           assert_equal(a, [[0, 1, 2], [3, 4, 5]])
1898: (8)           i.close()
1899: (8)           j.close()
1900: (8)           assert_equal(a, [[1, 2, 3], [4, 5, 6]])
1901: (4) def test_dtype_buffered(self):
1902: (8)     a = arange(6, dtype='f4').reshape(2, 3)
1903: (8)     i, j = np.nested_iters(a, [[0], [1]],
1904: (28)           flags=['buffered'],
1905: (28)           op_flags=['readwrite'],
1906: (28)           casting='same_kind',
1907: (28)           op_dtypes='f8')
1908: (8)     assert_equal(j[0].dtype, np.dtype('f8'))
1909: (8)     for x in i:
1910: (12)       for y in j:
1911: (16)         y[...] += 1
1912: (8)         assert_equal(a, [[1, 2, 3], [4, 5, 6]])
1913: (4) def test_0d(self):
1914: (8)     a = np.arange(12).reshape(2, 3, 2)
1915: (8)     i, j = np.nested_iters(a, [[], [1, 0, 2]])
1916: (8)     vals = [list(j) for _ in i]
1917: (8)     assert_equal(vals, [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])
1918: (8)     i, j = np.nested_iters(a, [[1, 0, 2], []])
1919: (8)     vals = [list(j) for _ in i]
1920: (8)     assert_equal(vals, [[0], [1], [2], [3], [4], [5], [6], [7], [8], [9],
1921: [10], [11]])
1922: (8)     i, j, k = np.nested_iters(a, [[2, 0], [], [1]])
1923: (8)     vals = []
1924: (12)    for x in i:
1925: (16)      for y in j:
1926: (8)        vals.append([z for z in k])
1927: (4) def test_iter_nested_iters_dtype_buffered(self):
1928: (8)     a = arange(6, dtype='f4').reshape(2, 3)
1929: (8)     i, j = np.nested_iters(a, [[0], [1]],
1930: (28)           flags=['buffered'],
1931: (28)           op_flags=['readwrite'],
1932: (28)           casting='same_kind',
1933: (28)           op_dtypes='f8')
1934: (8)     with i, j:
1935: (12)       assert_equal(j[0].dtype, np.dtype('f8'))
1936: (12)       for x in i:
1937: (16)         for y in j:
1938: (20)           y[...] += 1
1939: (8)           assert_equal(a, [[1, 2, 3], [4, 5, 6]])
1940: (0) def test_iter_reduction_error():
1941: (4)     a = np.arange(6)
1942: (4)     assert_raises(ValueError, nditer, [a, None], [],
1943: (20)           [['readonly']], ['readwrite', 'allocate']),
1944: (20)           op_axes=[[0], [-1]])
1945: (4)     a = np.arange(6).reshape(2, 3)
1946: (4)     assert_raises(ValueError, nditer, [a, None], ['external_loop'],
1947: (20)           [['readonly']], ['readwrite', 'allocate']),
1948: (20)           op_axes=[[0, 1], [-1, -1]])
1949: (0) def test_iter_reduction():
1950: (4)     a = np.arange(6)
1951: (4)     i = nditer([a, None], ['reduce_ok'],
1952: (20)           [['readonly']], ['readwrite', 'allocate']),
1953: (20)           op_axes=[[0], [-1]])
1954: (4)     with i:
1955: (8)       i.operands[1][...] = 0
1956: (8)       for x, y in i:
1957: (12)         y[...] += x
1958: (8)         assert_equal(i.operands[1].ndim, 0)
1959: (8)         assert_equal(i.operands[1], np.sum(a))

```

```

1960: (4)
1961: (4)
1962: (20)
1963: (20)
1964: (4)
1965: (8)
1966: (8)
1967: (8)
1968: (8)
1969: (12)
1970: (16)
1971: (8)
1972: (8)
1973: (4)
1974: (4)
1975: (20)
1976: (20)
1977: (4)
1978: (28)
1979: (20)
1980: (20)
1981: (4)
1982: (8)
1983: (8)
1984: (8)
1985: (8)
1986: (12)
1987: (8)
1988: (12)
1989: (8)
1990: (8)
1991: (0)
1992: (4)
1993: (4)
1994: (4)
1995: (20)
1996: (20)
1997: (4)
1998: (8)
1999: (8)
2000: (8)
2001: (12)
2002: (4)
2003: (4)
2004: (4)
2005: (4)
2006: (20)
2007: (20)
2008: (4)
2009: (8)
2010: (8)
2011: (8)
2012: (12)
2013: (16)
2014: (4)
2015: (4)
2016: (4)
2017: (12)
2018: (12)
2019: (12)
2020: (12)
2021: (4)
2022: (8)
2023: (8)
2024: (8)
2025: (4)
2026: (4)
2027: (4)
2028: (4)

    a = np.arange(6).reshape(2, 3)
    i = nditer([a, None], ['reduce_ok', 'external_loop'],
               [['readonly'], ['readwrite', 'allocate']],
               op_axes=[[0, 1], [-1, -1]])
    with i:
        i.operands[1][...] = 0
        assert_equal(i[1].shape, (6,))
        assert_equal(i[1].strides, (0,))
        for x, y in i:
            for j in range(len(y)):
                y[j] += x[j]
        assert_equal(i.operands[1].ndim, 0)
        assert_equal(i.operands[1], np.sum(a))
    a = np.ones((2, 3, 5))
    it1 = nditer([a, None], ['reduce_ok', 'external_loop'],
                 [['readonly'], ['readwrite', 'allocate']],
                 op_axes=[None, [0, -1, 1]])
    it2 = nditer([a, None], ['reduce_ok', 'external_loop',
                            'buffered', 'delay_bufalloc'],
                 [['readonly'], ['readwrite', 'allocate']],
                 op_axes=[None, [0, -1, 1]], buffersize=10)
    with it1, it2:
        it1.operands[1].fill(0)
        it2.operands[1].fill(0)
        it2.reset()
        for x in it1:
            x[1][...] += x[0]
        for x in it2:
            x[1][...] += x[0]
        assert_equal(it1.operands[1], it2.operands[1])
        assert_equal(it2.operands[1].sum(), a.size)

def test_iter_buffering_reduction():
    a = np.arange(6)
    b = np.array(0., dtype='f8').byteswap().newbyteorder()
    i = nditer([a, b], ['reduce_ok', 'buffered'],
               [['readonly'], ['readwrite', 'nbo']],
               op_axes=[[0], [-1]])
    with i:
        assert_equal(i[1].dtype, np.dtype('f8'))
        assert_(i[1].dtype != b.dtype)
        for x, y in i:
            y[...] += x
    assert_equal(b, np.sum(a))
    a = np.arange(6).reshape(2, 3)
    b = np.array([0, 0], dtype='f8').byteswap().newbyteorder()
    i = nditer([a, b], ['reduce_ok', 'external_loop', 'buffered'],
               [['readonly'], ['readwrite', 'nbo']],
               op_axes=[[0, 1], [0, -1]])
    with i:
        assert_equal(i[1].shape, (3,))
        assert_equal(i[1].strides, (0,))
        for x, y in i:
            for j in range(len(y)):
                y[j] += x[j]
    assert_equal(b, np.sum(a, axis=1))
    p = np.arange(2) + 1
    it = np.nditer([p, None],
                  ['delay_bufalloc', 'reduce_ok', 'buffered', 'external_loop'],
                  [['readonly'], ['readwrite', 'allocate']],
                  op_axes=[[-1, 0], [-1, -1]],
                  itershape=(2, 2))
    with it:
        it.operands[1].fill(0)
        it.reset()
        assert_equal(it[0], [1, 2, 1, 2])
    x = np.ones((7, 13, 8), np.int8)[4:6, 1:11:6, 1:5].transpose(1, 2, 0)
    x[...] = np.arange(x.size).reshape(x.shape)
    y_base = np.arange(4*4, dtype=np.int8).reshape(4, 4)
    y_base_copy = y_base.copy()

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

2029: (4)
2030: (4)
2031: (19)
2032: (19)
2033: (4)
2034: (8)
2035: (12)
2036: (4)
2037: (4)
2038: (0)
2039: (4)
2040: (4)
2041: (4)
2042: (20)
2043: (20)
2044: (4)
2045: (8)
2046: (4)
2047: (4)
2048: (0)
2049: (4)
2050: (4)
2051: (4)
2052: (4)
2053: (4)
2054: (4)
2055: (4)
2056: (4)
2057: (20)
2058: (4)
2059: (20)
2060: (4)
2061: (20)
2062: (4)
2063: (20)
2064: (21)
2065: (21)
2066: (4)
2067: (20)
2068: (4)
2069: (20)
2070: (21)
2071: (21)
2072: (4)
2073: (20)
2074: (21)
2075: (21)
2076: (4)
2077: (20)
2078: (21)
2079: (21)
2080: (4)
2081: (20)
2082: (21)
2083: (20)
2084: (20)
2085: (4)
2086: (20)
2087: (21)
2088: (20)
2089: (20)
2090: (4)
2091: (20)
2092: (21)
2093: (20)
2094: (20)
2095: (0)
2096: (4)
2097: (8)

    y = y_base[::-2,:,:None]
    it = np.nditer([y, x],
                  ['buffered', 'external_loop', 'reduce_ok'],
                  [['readonly'], ['readonly']])

    with it:
        for a, b in it:
            a.fill(2)
    assert_equal(y_base[1::2], y_base_copy[1::2])
    assert_equal(y_base[1::2], 2)

def test_iter_buffering_reduction_reuse_reduce_loops():
    a = np.zeros((2, 7))
    b = np.zeros((1, 7))
    it = np.nditer([a, b], flags=['reduce_ok', 'external_loop', 'buffered'],
                  op_flags=[['readonly'], ['readwrite']],
                  buffersize=5)

    with it:
        bufsizes = [x.shape[0] for x, y in it]
    assert_equal(bufsizes, [5, 2, 5, 2])
    assert_equal(sum(bufsizes), a.size)

def test_iter_writemasked_badininput():
    a = np.zeros((2, 3))
    b = np.zeros((3,))
    m = np.array([[True, True, False], [False, True, False]])
    m2 = np.array([True, True, False])
    m3 = np.array([0, 1, 1], dtype='u1')
    mbad1 = np.array([0, 1, 1], dtype='i1')
    mbad2 = np.array([0, 1, 1], dtype='f4')
    assert_raises(ValueError, nditer, [a, m], [],
                  [['readwrite', 'writemasked'], ['readonly']])

    assert_raises(ValueError, nditer, [a, m], [],
                  [['readonly', 'writemasked'], ['readonly', 'arraymask']])

    assert_raises(ValueError, nditer, [a, m], [],
                  [['readonly'], ['readwrite', 'arraymask', 'writemasked']])

    assert_raises(ValueError, nditer, [a, m, m2], [],
                  [['readwrite', 'writemasked'],
                   ['readonly', 'arraymask'],
                   ['readonly', 'arraymask']])

    assert_raises(ValueError, nditer, [a, m], [],
                  [['readwrite'], ['readonly', 'arraymask']])

    assert_raises(ValueError, nditer, [a, b, m], ['reduce_ok'],
                  [['readonly'],
                   ['readwrite', 'writemasked'],
                   ['readonly', 'arraymask']]))

    np.nditer([a, b, m2], ['reduce_ok'],
              [['readonly'],
               ['readwrite', 'writemasked'],
               ['readonly', 'arraymask']])

    assert_raises(ValueError, nditer, [a, b, m2], ['reduce_ok'],
                  [['readonly'],
                   ['readwrite', 'writemasked'],
                   ['readonly', 'arraymask']]))

    np.nditer([a, m3], ['buffered'],
              [['readwrite', 'writemasked'],
               ['readonly', 'arraymask']],
              op_dtypes=['f4', None],
              casting='same_kind')

    assert_raises(TypeError, np.nditer, [a, mbad1], ['buffered'],
                  [['readwrite', 'writemasked'],
                   ['readonly', 'arraymask']],
                   op_dtypes=['f4', None],
                   casting='same_kind'))

    assert_raises(TypeError, np.nditer, [a, mbad2], ['buffered'],
                  [['readwrite', 'writemasked'],
                   ['readonly', 'arraymask']],
                   op_dtypes=['f4', None],
                   casting='same_kind'))

def _is_buffered(iterator):
    try:
        iterator.itviews
    
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

2098: (4)
2099: (8)
2100: (4)
2101: (0)
2102: (8)
2103: (9)
2104: (9)
2105: (9)
2106: (9)
2107: (9)
2108: (0)
2109: (4)
2110: (4)
2111: (4)
2112: (4)
2113: (4)
2114: (16)
2115: (17)
2116: (4)
2117: (8)
2118: (12)
2119: (4)
2120: (4)
2121: (16)
2122: (17)
2123: (4)
2124: (4)
2125: (8)
2126: (12)
2127: (12)
2128: (16)
2129: (4)
2130: (8)
2131: (4)
2132: (8)
2133: (8)
2134: (4)
2135: (16)
2136: (17)
2137: (16)
2138: (16)
2139: (4)
2140: (8)
2141: (12)
2142: (4)
2143: (0)
2144: (8)
2145: (8)
2146: (8)
2147: (8)
2148: (8)
2149: (8)
2150: (0)
2151: (4)
2152: (4)
2153: (4)
2154: (4)
2155: (4)
2156: (8)
2157: (4)
2158: (8)
2159: (4)
2160: (8)
2161: (18)
2162: (0)
2163: (4)
2164: (4)
2165: (4)
2166: (4)

        except ValueError:
            return True
        return False
@pytest.mark.parametrize("a",
    [np.zeros((3,), dtype='f8'),
     np.zeros((9876, 3*5), dtype='f8')[::2, :],
     np.zeros((4, 312, 124, 3), dtype='f8')[::2, :, ::2, :],
     np.zeros((9,), dtype='f8')[::3],
     np.zeros((9876, 3*10), dtype='f8')[::2, ::5],
     np.zeros((4, 312, 124, 3), dtype='f8')[::2, :, ::2, ::-1]])
def test_iter_writemasked(a):
    shape = a.shape
    reps = shape[-1] // 3
    msk = np.empty(shape, dtype=bool)
    msk[...] = [True, True, False] * reps
    it = np.nditer([a, msk], [],
                  [['readwrite', 'writemasked'],
                   ['readonly', 'arraymask']])
    with it:
        for x, m in it:
            x[...] = 1
    assert_equal(a, np.broadcast_to([1, 1, 1] * reps, shape))
    it = np.nditer([a, msk], ['buffered'],
                  [['readwrite', 'writemasked'],
                   ['readonly', 'arraymask']])
    is_buffered = True
    with it:
        for x, m in it:
            x[...] = 2.5
            if np.may_share_memory(x, a):
                is_buffered = False
    if not is_buffered:
        assert_equal(a, np.broadcast_to([2.5, 2.5, 2.5] * reps, shape))
    else:
        assert_equal(a, np.broadcast_to([2.5, 2.5, 1] * reps, shape))
        a[...] = 2.5
    it = np.nditer([a, msk], ['buffered'],
                  [['readwrite', 'writemasked'],
                   ['readonly', 'arraymask']],
                  op_dtypes=['i8', None],
                  casting='unsafe')
    with it:
        for x, m in it:
            x[...] = 3
    assert_equal(a, np.broadcast_to([3, 3, 2.5] * reps, shape))
@pytest.mark.parametrize(["mask", "mask_axes"], [
    (None, [-1, 0]),
    (np.zeros((1, 4), dtype="bool"), [0, 1]),
    (np.zeros((1, 4), dtype="bool"), None),
    (np.zeros(4, dtype="bool"), [-1, 0]),
    (np.zeros((), dtype="bool"), [-1, -1]),
    (np.zeros((), dtype="bool"), None)])
def test_iter_writemasked_broadcast_error(mask, mask_axes):
    arr = np.zeros((3, 4))
    itflags = ["reduce_ok"]
    mask_flags = ["arraymask", "readwrite", "allocate"]
    a_flags = ["writeonly", "writemasked"]
    if mask_axes is None:
        op_axes = None
    else:
        op_axes = [mask_axes, [0, 1]]
    with assert_raises(ValueError):
        np.nditer((mask, arr), flags=itflags, op_flags=[mask_flags, a_flags],
                  op_axes=op_axes)
def test_iter_writemasked_decref():
    arr = np.arange(10000).astype(">i,0")
    original = arr.copy()
    mask = np.random.randint(0, 2, size=10000).astype(bool)
    it = np.nditer([arr, mask], ['buffered', "refs_ok"],

```

```

2167: (19)                                [['readwrite', 'writemasked'],
2168: (20)                                ['readonly', 'arraymask']],
2169: (19)                                op_dtypes=["<i,0", "?"])
2170: (4)         singleton = object()
2171: (4)         if HAS_REFCOUNT:
2172: (8)             count = sys.getrefcount(singleton)
2173: (4)             for buf, mask_buf in it:
2174: (8)                 buf[...] = (3, singleton)
2175: (4)             del buf, mask_buf, it    # delete everything to ensure correct cleanup
2176: (4)         if HAS_REFCOUNT:
2177: (8)             assert sys.getrefcount(singleton) - count == np.count_nonzero(mask)
2178: (4)         assert_array_equal(arr[~mask], original[~mask])
2179: (4)         assert (arr[mask] == np.array((3, singleton), arr.dtype)).all()
2180: (4)         del arr
2181: (4)         if HAS_REFCOUNT:
2182: (8)             assert sys.getrefcount(singleton) == count
2183: (0)     def test_iter_non_writable_attribute_deletion():
2184: (4)         it = np.nditer(np.ones(2))
2185: (4)         attr = ["value", "shape", "operands", "itviews", "has_delayed_bufalloc",
2186: (12)             "iterationneedsapi", "has_multi_index", "has_index", "dtypes",
2187: (12)             "ndim", "nop", "itersize", "finished"]
2188: (4)         for s in attr:
2189: (8)             assert_raises(AttributeError, delattr, it, s)
2190: (0)     def test_iter_writable_attribute_deletion():
2191: (4)         it = np.nditer(np.ones(2))
2192: (4)         attr = [ "multi_index", "index", "iterrange", "iterindex"]
2193: (4)         for s in attr:
2194: (8)             assert_raises(AttributeError, delattr, it, s)
2195: (0)     def test_iter_element_deletion():
2196: (4)         it = np.nditer(np.ones(3))
2197: (4)         try:
2198: (8)             del it[1]
2199: (8)             del it[1:2]
2200: (4)         except TypeError:
2201: (8)             pass
2202: (4)         except Exception:
2203: (8)             raise AssertionError
2204: (0)     def test_iter_allocated_array_dtotypes():
2205: (4)         it = np.nditer(([1, 3, 20], None), op_dtotypes=[None, ('i4', (2,))])
2206: (4)         for a, b in it:
2207: (8)             b[0] = a - 1
2208: (8)             b[1] = a + 1
2209: (4)         assert_equal(it.operands[1], [[0, 2], [2, 4], [19, 21]])
2210: (4)         it = np.nditer(([1, 3, 20]), None, op_dtotypes=[None, ('i4', (2,))],
2211: (19)             flags=["reduce_ok"], op_axes=[None, (-1, 0)])
2212: (4)         for a, b in it:
2213: (8)             b[0] = a - 1
2214: (8)             b[1] = a + 1
2215: (4)         assert_equal(it.operands[1], [[0, 2], [2, 4], [19, 21]])
2216: (4)         it = np.nditer((10, 2, None), op_dtotypes=[None, None, ('i4', (2, 2))])
2217: (4)         for a, b, c in it:
2218: (8)             c[0, 0] = a - b
2219: (8)             c[0, 1] = a + b
2220: (8)             c[1, 0] = a * b
2221: (8)             c[1, 1] = a / b
2222: (4)         assert_equal(it.operands[2], [[8, 12], [20, 5]])
2223: (0)     def test_0d_iter():
2224: (4)         i = nditer([2, 3], ['multi_index'], [[['readonly']]])**2
2225: (4)         assert_equal(i.ndim, 0)
2226: (4)         assert_equal(next(i), (2, 3))
2227: (4)         assert_equal(i.multi_index, ())
2228: (4)         assert_equal(i.iterindex, 0)
2229: (4)         assert_raises(StopIteration, next, i)
2230: (4)         i.reset()
2231: (4)         assert_equal(next(i), (2, 3))
2232: (4)         assert_raises(StopIteration, next, i)
2233: (4)         i = nditer(np.arange(5), ['multi_index'], [[['readonly']]], op_axes=[()])
2234: (4)         assert_equal(i.ndim, 0)
2235: (4)         assert_equal(len(i), 1)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

2236: (4)
2237: (15)
2238: (4)
2239: (4)
2240: (4)
2241: (8)
2242: (4)
2243: (4)
2244: (4)
2245: (20)
2246: (4)
2247: (4)
2248: (4)
2249: (4)
2250: (4)
2251: (0)
2252: (4)
2253: (4)
2254: (4)
2255: (4)
2256: (4)
2257: (4)
2258: (8)
2259: (12)
2260: (4)
2261: (29)
2262: (0)
2263: (4)
2264: (4)
2265: (8)
2266: (0)
2267: (8)
2268: (8)
2269: (8)
2270: (0)
2271: (4)
2272: (4)
2273: (4)
2274: (0)
2275: (4)
2276: (4)
2277: (4)
2278: (4)
2279: (18)
2280: (0)
2281: (4)
2282: (4)
2283: (4)
2284: (8)
2285: (4)
2286: (4)
2287: (4)
2288: (8)
2289: (8)
2290: (8)
2291: (4)
2292: (4)
2293: (8)
2294: (12)
2295: (4)
2296: (4)
2297: (8)
2298: (12)
2299: (12)
2300: (16)
2301: (0)
2302: (4)
2303: (4)
2304: (4)

    i = nditer(np.arange(5), ['multi_index'], [['readonly']],
               op_axes=[()], itershape=())
    assert_equal(i.ndim, 0)
    assert_equal(len(i), 1)
    with assert_raises(ValueError):
        nditer(np.arange(5), ['multi_index'], itershape=())
    sdt = [('a', 'f4'), ('b', 'i8'), ('c', 'c8', (2, 3)), ('d', 'O')]
    a = np.array(0.5, dtype='f4')
    i = nditer(a, ['buffered', 'refs_ok'], ['readonly'],
               casting='unsafe', op_dtypes=sdt)
    vals = next(i)
    assert_equal(vals['a'], 0.5)
    assert_equal(vals['b'], 0)
    assert_equal(vals['c'], [[(0.5)]*3]*2)
    assert_equal(vals['d'], 0.5)
def test_object_iter_cleanup():
    assert_raises(TypeError, lambda: np.zeros((17000, 2), dtype='f4') * None)
    arr = np.arange(np.BUFSIZE * 10).reshape(10, -1).astype(str)
    oarr = arr.astype(object)
    oarr[:, -1] = None
    assert_raises(TypeError, lambda: np.add(oarr[:, ::-1], arr[:, ::-1]))
    class T:
        def __bool__(self):
            raise TypeError("Ambiguous")
    assert_raises(TypeError, np.logical_or.reduce,
                  np.array([T(), T()], dtype='O'))
def test_object_iter_cleanup_reduce():
    arr = np.array([[None, 1], [-1, -1], [None, 2], [-1, -1]])[:, ::2]
    with pytest.raises(TypeError):
        np.sum(arr)
    @pytest.mark.parametrize("arr", [
        np.ones((8000, 4, 2), dtype=object)[:, ::2, :],
        np.ones((8000, 4, 2), dtype=object, order="F")[:, ::2, :],
        np.ones((8000, 4, 2), dtype=object)[:, ::2, :].copy("F")])
def test_object_iter_cleanup_large_reduce(arr):
    out = np.ones(8000, dtype=np.intp)
    res = np.sum(arr, axis=(1, 2), dtype=object, out=out)
    assert_array_equal(res, np.full(8000, 4, dtype=object))
def test_iter_too_large():
    size = np.iinfo(np.intp).max // 1024
    arr = np.lib.stride_tricks.as_strided(np.zeros(1), (size,), (0,))
    assert_raises(ValueError, nditer, (arr, arr[:, None]))
    assert_raises(ValueError, nditer,
                  (arr, arr[:, None]), flags=['multi_index'])
def test_iter_too_large_with_multiindex():
    base_size = 2**10
    num = 1
    while base_size**num < np.iinfo(np.intp).max:
        num += 1
    shape_template = [1, 1] * num
    arrays = []
    for i in range(num):
        shape = shape_template[:]
        shape[i * 2] = 2**10
        arrays.append(np.empty(shape))
    arrays = tuple(arrays)
    for mode in range(6):
        with assert_raises(ValueError):
            _multiarray_tests.test_nditer_too_large(arrays, -1, mode)
    _multiarray_tests.test_nditer_too_large(arrays, -1, 7)
    for i in range(num):
        for mode in range(6):
            _multiarray_tests.test_nditer_too_large(arrays, i*2, mode)
            with assert_raises(ValueError):
                _multiarray_tests.test_nditer_too_large(arrays, i*2 + 1, mode)
def test_writebacks():
    a = np.arange(6, dtype='f4')
    au = a.byteswap().newbyteorder()
    assert_(a.dtype.byteorder != au.dtype.byteorder)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

2305: (4)
2306: (24)
2307: (4)
2308: (8)
2309: (4)
2310: (4)
2311: (24)
2312: (4)
2313: (8)
2314: (12)
2315: (12)
2316: (12)
2317: (4)
2318: (8)
2319: (4)
2320: (4)
2321: (4)
2322: (4)
2323: (24)
2324: (4)
2325: (8)
2326: (8)
2327: (8)
2328: (4)
2329: (4)
2330: (4)
2331: (4)
2332: (4)
2333: (17)
2334: (17)
2335: (4)
2336: (8)
2337: (12)
2338: (16)
2339: (4)
2340: (17)
2341: (17)
2342: (4)
2343: (8)
2344: (12)
2345: (16)
2346: (8)
2347: (4)
2348: (17)
2349: (17)
2350: (4)
2351: (4)
2352: (8)
2353: (12)
2354: (4)
2355: (4)
2356: (0)
2357: (4)
2358: (4)
2359: (4)
2360: (8)
2361: (8)
2362: (20)
2363: (8)
2364: (12)
2365: (8)
2366: (8)
2367: (8)
2368: (4)
2369: (8)
2370: (8)
2371: (20)
2372: (8)
2373: (12)

    it = nditer(au, [], [['readonly'], ['readonly'], ['writeonly','allocate']])
    casting='equiv', op_dtypes=[np.dtype('f4')])
    with it:
        it.operands[0][:] = 100
    assert_equal(au, 100)
    it = nditer(au, [], [['readonly'], ['readonly'], ['writeonly','allocate']])
    casting='equiv', op_dtypes=[np.dtype('f4')])
try:
    with it:
        assert_equal(au.flags.writeable, False)
        it.operands[0][:] = 0
        raise ValueError('exit context manager on exception')
except:
    pass
assert_equal(au, 0)
assert_equal(au.flags.writeable, True)
assert_raises(ValueError, getattr, it, 'operands')
it = nditer(au, [], [['readonly'], ['readonly'], ['writeonly','allocate']])
casting='equiv', op_dtypes=[np.dtype('f4')])
with it:
    x = it.operands[0]
    x[:] = 6
    assert_(x.flags.writebackifcopy)
assert_equal(au, 6)
assert_(not x.flags.writebackifcopy)
x[:] = 123 # x.data still valid
assert_equal(au, 6) # but not connected to au
it = nditer(au, [],
            [['readonly'], ['readonly'], ['writeonly','allocate']])
casting='equiv', op_dtypes=[np.dtype('f4')])
with it:
    with it:
        for x in it:
            x[...] = 123
it = nditer(au, [],
            [['readonly'], ['readonly'], ['writeonly','allocate']])
casting='equiv', op_dtypes=[np.dtype('f4')])
with it:
    with it:
        for x in it:
            x[...] = 123
        assert_raises(ValueError, getattr, it, 'operands')
it = nditer(au, [],
            [['readonly'], ['readonly'], ['writeonly','allocate']])
casting='equiv', op_dtypes=[np.dtype('f4')])
del au
with it:
    for x in it:
        x[...] = 123
enter = it.__enter__
assert_raises(RuntimeError, enter)
def test_close_equivalent():
    ''' using a context manager and using nditer.close are equivalent
    '''
    def add_close(x, y, out=None):
        addop = np.add
        it = np.nditer([x, y, out], [],
                      [['readonly'], ['readonly'], ['writeonly','allocate']])
        for (a, b, c) in it:
            addop(a, b, out=c)
        ret = it.operands[2]
        it.close()
        return ret
    def add_context(x, y, out=None):
        addop = np.add
        it = np.nditer([x, y, out], [],
                      [['readonly'], ['readonly'], ['writeonly','allocate']])
        with it:
            for (a, b, c) in it:

```

```

2374: (16)           addop(a, b, out=c)
2375: (12)           return it.operands[2]
2376: (4)            z = add_close(range(5), range(5))
2377: (4)            assert_equal(z, range(0, 10, 2))
2378: (4)            z = add_context(range(5), range(5))
2379: (4)            assert_equal(z, range(0, 10, 2))
2380: (0)             def test_close_raises():
2381: (4)               it = np.nditer(np.arange(3))
2382: (4)               assert_equal(next(it), 0)
2383: (4)               it.close()
2384: (4)               assert_raises(StopIteration, next, it)
2385: (4)               assert_raises(ValueError, getattr, it, 'operands')
2386: (0)             def test_close_parameters():
2387: (4)               it = np.nditer(np.arange(3))
2388: (4)               assert_raises(TypeError, it.close, 1)
2389: (0)             @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
2390: (0)             def test_warn_noclose():
2391: (4)               a = np.arange(6, dtype='f4')
2392: (4)               au = a.byteswap().newbyteorder()
2393: (4)               with suppress_warnings() as sup:
2394: (8)                 sup.record(RuntimeWarning)
2395: (8)                 it = np.nditer(au, [], [['readwrite', 'updateifcopy']],
2396: (24)                               casting='equiv', op_dtypes=[np.dtype('f4')])
2397: (8)               del it
2398: (8)               assert len(sup.log) == 1
2399: (0)             @pytest.mark.skipif(sys.version_info[:2] == (3, 9) and sys.platform ==
2400: "win32",
2401: (20)                           reason="Errors with Python 3.9 on Windows")
2402: (0)             @pytest.mark.parametrize(["in_dtype", "buf_dtype"],
2403: (8)               [("i", "O"), ("O", "i"), # most simple cases
2404: (9)               ("i,O", "O,O"), # structured partially only copying 0
2405: (9)               ("O,i", "i,O"), # structured casting to and from 0
2406: (9)               ])
2407: (0)             @pytest.mark.parametrize("steps", [1, 2, 3])
2408: (4)             def test_partial_iteration_cleanup(in_dtype, buf_dtype, steps):
2409: (4)               """
2410: (4)                 Checks for reference counting leaks during cleanup. Using explicit
2411: (4)                 reference counts lead to occasional false positives (at least in parallel
2412: (4)                 test setups). This test now should still test leaks correctly when
2413: (4)                 run e.g. with pytest-valgrind or pytest-leaks
2414: (4)               """
2415: (4)               value = 2**30 + 1 # just a random value that Python won't intern
2416: (4)               arr = np.full(int(np.BUFSIZE * 2.5), value).astype(in_dtype)
2417: (12)              it = np.nditer(arr, op_dtypes=[np.dtype(buf_dtype)],
2418: (4)                            flags=['buffered', 'external_loop', 'refs_ok'], casting="unsafe")
2419: (4)               for step in range(steps):
2420: (8)                 next(it)
2421: (4)               del it # not necessary, but we test the cleanup
2422: (4)               it = np.nditer(arr, op_dtypes=[np.dtype(buf_dtype)],
2423: (19)                             flags=['buffered', 'external_loop', 'refs_ok'],
2424: (4)                             casting="unsafe")
2425: (4)               for step in range(steps):
2426: (8)                 it.iteernext()
2427: (4)               del it # not necessary, but we test the cleanup
2428: (0)             @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
2429: (0)             @pytest.mark.parametrize(["in_dtype", "buf_dtype"],
2430: (9)               [("O", "i"), # most simple cases
2431: (10)               ("O,i", "i,O"), # structured casting to and from 0
2432: (10)               ])
2433: (0)             def test_partial_iteration_error(in_dtype, buf_dtype):
2434: (4)               value = 123 # relies on python cache (leak-check will still find it)
2435: (4)               arr = np.full(int(np.BUFSIZE * 2.5), value).astype(in_dtype)
2436: (4)               if in_dtype == "O":
2437: (8)                 arr[int(np.BUFSIZE * 1.5)] = None
2438: (4)               else:
2439: (8)                 arr[int(np.BUFSIZE * 1.5)]["f0"] = None
2440: (4)               count = sys.getrefcount(value)
2441: (4)               it = np.nditer(arr, op_dtypes=[np.dtype(buf_dtype)],
2442: (12)                             flags=['buffered', 'external_loop', 'refs_ok'], casting="unsafe")

```

```

2441: (4)           with pytest.raises(TypeError):
2442: (8)             next(it)
2443: (8)               next(it) # raises TypeError
2444: (4)             it.reset()
2445: (4)             with pytest.raises(TypeError):
2446: (8)               it.ite rn ext()
2447: (8)                 it.ite rn ext()
2448: (4)               assert count == sys.getrefcount(value)
2449: (0)             def test_debug_print(capfd):
2450: (4)               """
2451: (4)                   Matches the expected output of a debug print with the actual output.
2452: (4)                   Note that the iterator dump should not be considered stable API,
2453: (4)                   this test is mainly to ensure the print does not crash.
2454: (4)                   Currently uses a subprocess to avoid dealing with the C level `printf`s.
2455: (4)               """
2456: (4)               expected = """
2457: (4)               ----- BEGIN ITERATOR DUMP -----
2458: (4)               Iterator Address:
2459: (4)                 ItFlags: BUFFER REDUCE REUSE_REDUCE_LOOPS
2460: (4)                 NDim: 2
2461: (4)                 NOp: 2
2462: (4)                 IterSize: 50
2463: (4)                 IterStart: 0
2464: (4)                 IterEnd: 50
2465: (4)                 IterIndex: 0
2466: (4)                 Iterator SizeOf:
2467: (4)                 BufferData SizeOf:
2468: (4)                 AxisData SizeOf:
2469: (4)
2470: (4)                 Perm: 0 1
2471: (4)                 DTypes:
2472: (4)                   DTypes: dtype('float64') dtype('int32')
2473: (4)                 InitDataPtrs:
2474: (4)                   BaseOffsets: 0 0
2475: (4)                 Operands:
2476: (4)                   Operand DTypes: dtype('int64') dtype('float64')
2477: (4)                 OpItFlags:
2478: (4)                   Flags[0]: READ CAST ALIGNED
2479: (4)                   Flags[1]: READ WRITE CAST ALIGNED REDUCE
2480: (4)
2481: (4)                 BufferData:
2482: (4)                   BufferSize: 50
2483: (4)                   Size: 5
2484: (4)                   BufIterEnd: 5
2485: (4)                   REDUCE Pos: 0
2486: (4)                   REDUCE OuterSize: 10
2487: (4)                   REDUCE OuterDim: 1
2488: (4)                   Strides: 8 4
2489: (4)                 Ptrs:
2490: (4)                   REDUCE Outer Strides: 40 0
2491: (4)                   REDUCE Outer Ptrs:
2492: (4)                     ReadTransferFn:
2493: (4)                     ReadTransferData:
2494: (4)                     WriteTransferFn:
2495: (4)                     WriteTransferData:
2496: (4)                 Buffers:
2497: (4)
2498: (4)                   AxisData[0]:
2499: (4)                     Shape: 5
2500: (4)                     Index: 0
2501: (4)                     Strides: 16 8
2502: (4)                 Ptrs:
2503: (4)                   AxisData[1]:
2504: (4)                     Shape: 10
2505: (4)                     Index: 0
2506: (4)                     Strides: 80 0
2507: (4)                 Ptrs:
2508: (4)               ----- END ITERATOR DUMP -----
2509: (4)             """".strip().splitlines()

```

```

2510: (4) arr1 = np.arange(100, dtype=np.int64).reshape(10, 10)[:, ::2]
2511: (4) arr2 = np.arange(5.)
2512: (4) it = np.nditer((arr1, arr2), op_dtypes=["d", "i4"], casting="unsafe",
2513: (19)             flags=["reduce_ok", "buffered"],
2514: (19)             op_flags=[["readonly"], ["readwrite"]])
2515: (4) it.debug_print()
2516: (4) res = capfd.readouterr().out
2517: (4) res = res.strip().splitlines()
2518: (4) assert len(res) == len(expected)
2519: (4) for res_line, expected_line in zip(res, expected):
2520: (8)     assert res_line.startswith(expected_line.strip())
-----
```

## File 110 - test\_nep50\_promotions.py:

```

1: (0) """
2: (0) This file adds basic tests to test the NEP 50 style promotion compatibility
3: (0) mode. Most of these test are likely to be simply deleted again once NEP 50
4: (0) is adopted in the main test suite. A few may be moved elsewhere.
5: (0) """
6: (0) import operator
7: (0) import numpy as np
8: (0) import pytest
9: (0) from numpy.testing import IS_WASM
10: (0) @pytest.fixture(scope="module", autouse=True)
11: (0) def _weak_promotion_enabled():
12: (4)     state = np._get_promotion_state()
13: (4)     np._set_promotion_state("weak_and_warn")
14: (4)     yield
15: (4)     np._set_promotion_state(state)
16: (0) @pytest.mark.skipif(IS_WASM, reason="wasm doesn't have support for fp errors")
17: (0) def test_nep50_examples():
18: (4)     with pytest.warns(UserWarning, match="result dtype changed"):
19: (8)         res = np.uint8(1) + 2
20: (4)         assert res.dtype == np.uint8
21: (4)         with pytest.warns(UserWarning, match="result dtype changed"):
22: (8)             res = np.array([1], np.uint8) + np.int64(1)
23: (4)             assert res.dtype == np.int64
24: (4)             with pytest.warns(UserWarning, match="result dtype changed"):
25: (8)                 res = np.array([1], np.uint8) + np.array(1, dtype=np.int64)
26: (4)                 assert res.dtype == np.int64
27: (4)                 with pytest.warns(UserWarning, match="result dtype changed"):
28: (8)                     with np.errstate(over="raise"):
29: (12)                         res = np.uint8(100) + 200
30: (4)                         assert res.dtype == np.uint8
31: (4)                         with pytest.warns(Warning) as recwarn:
32: (8)                             res = np.float32(1) + 3e100
33: (4)                             warning = str(recwarn.pop(UserWarning).message)
34: (4)                             assert warning.startswith("result dtype changed")
35: (4)                             warning = str(recwarn.pop(RuntimeWarning).message)
36: (4)                             assert warning.startswith("overflow")
37: (4)                             assert len(recwarn) == 0 # no further warnings
38: (4)                             assert np.isinf(res)
39: (4)                             assert res.dtype == np.float32
40: (4)                             res = np.array([0.1], np.float32) == np.float64(0.1)
41: (4)                             assert res[0] == False
42: (4)                             with pytest.warns(UserWarning, match="result dtype changed"):
43: (8)                                 res = np.array([0.1], np.float32) + np.float64(0.1)
44: (4)                                 assert res.dtype == np.float64
45: (4)                                 with pytest.warns(UserWarning, match="result dtype changed"):
46: (8)                                     res = np.array([1.], np.float32) + np.int64(3)
47: (4)                                     assert res.dtype == np.float64
48: (0) @pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
49: (0) def test_nep50_weak_integers(dtype):
50: (4)     np._set_promotion_state("weak")
51: (4)     scalar_type = np.dtype(dtype).type
52: (4)     maxint = int(np.iinfo(dtype).max)
53: (4)     with np.errstate(over="warn"):
```

```

54: (8)           with pytest.warns(RuntimeWarning):
55: (12)             res = scalar_type(100) + maxint
56: (4)             assert res.dtype == dtype
57: (4)             res = np.array(100, dtype=dtype) + maxint
58: (4)             assert res.dtype == dtype
59: (0) @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
60: (0) def test_nep50_weak_integers_with_inexact(dtype):
61: (4)     np._set_promotion_state("weak")
62: (4)     scalar_type = np.dtype(dtype).type
63: (4)     too_big_int = int(np.finfo(dtype).max) * 2
64: (4)     if dtype in "dDG":
65: (8)         with pytest.raises(OverflowError):
66: (12)             scalar_type(1) + too_big_int
67: (8)             with pytest.raises(OverflowError):
68: (12)                 np.array(1, dtype=dtype) + too_big_int
69: (4)     else:
70: (8)         if dtype in "gG":
71: (12)             try:
72: (16)                 str(too_big_int)
73: (12)             except ValueError:
74: (16)                 pytest.skip("`huge_int -> string -> longdouble` failed")
75: (8)             with pytest.warns(RuntimeWarning):
76: (12)                 res = scalar_type(1) + too_big_int
77: (8)                 assert res.dtype == dtype
78: (8)                 assert res == np.inf
79: (8)             with pytest.warns(RuntimeWarning):
80: (12)                 res = np.add(np.array(1, dtype=dtype), too_big_int, dtype=dtype)
81: (8)                 assert res.dtype == dtype
82: (8)                 assert res == np.inf
83: (0) @pytest.mark.parametrize("op", [operator.add, operator.pow, operator.eq])
84: (0) def test_weak_promotion_scalar_path(op):
85: (4)     np._set_promotion_state("weak")
86: (4)     res = op(np.uint8(3), 5)
87: (4)     assert res == op(3, 5)
88: (4)     assert res.dtype == np.uint8 or res.dtype == bool
89: (4)     with pytest.raises(OverflowError):
90: (8)         op(np.uint8(3), 1000)
91: (4)         res = op(np.float32(3), 5.)
92: (4)         assert res == op(3., 5.)
93: (4)         assert res.dtype == np.float32 or res.dtype == bool
94: (0) def test_nep50_complex_promotion():
95: (4)     np._set_promotion_state("weak")
96: (4)     with pytest.warns(RuntimeWarning, match=".*overflow"):
97: (8)         res = np.complex64(3) + complex(2**300)
98: (4)         assert type(res) == np.complex64
99: (0) def test_nep50_integer_conversion_errors():
100: (4)     np._set_promotion_state("weak")
101: (4)     with pytest.raises(OverflowError, match=".*uint8"):
102: (8)         np.array([1], np.uint8) + 300
103: (4)     with pytest.raises(OverflowError, match=".*uint8"):
104: (8)         np.uint8(1) + 300
105: (4)     with pytest.raises(OverflowError,
106: (12)             match="Python integer -1 out of bounds for uint8"):
107: (8)         np.uint8(1) + -1
108: (0) def test_nep50_integer_regression():
109: (4)     np._set_promotion_state("legacy")
110: (4)     arr = np.array(1)
111: (4)     assert (arr + 2**63).dtype == np.float64
112: (4)     assert (arr[()] + 2**63).dtype == np.float64
113: (0) def test_nep50_with_axisconcatenator():
114: (4)     np._set_promotion_state("weak")
115: (4)     with pytest.raises(OverflowError):
116: (8)         np.r_[np.arange(5, dtype=np.int8), 255]
117: (0) @pytest.mark.parametrize("ufunc", [np.add, np.power])
118: (0) @pytest.mark.parametrize("state", ["weak", "weak_and_warn"])
119: (0) def test_nep50_huge_integers(ufunc, state):
120: (4)     np._set_promotion_state(state)
121: (4)     with pytest.raises(OverflowError):
122: (8)         ufunc(np.int64(0), 2**63) # 2**63 too large for int64

```

```

123: (4)             if state == "weak_and_warn":
124: (8)               with pytest.warns(UserWarning,
125: (16)                 match="result dtype changed.*float64.*uint64"):
126: (12)                   with pytest.raises(OverflowError):
127: (16)                     ufunc(np.uint64(0), 2**64)
128: (4)               else:
129: (8)                 with pytest.raises(OverflowError):
130: (12)                   ufunc(np.uint64(0), 2**64) # 2**64 cannot be represented by
131: (4)                     uint64
132: (8)
133: (16)                     if state == "weak_and_warn":
134: (12)                       with pytest.warns(UserWarning,
135: (4)                         match="result dtype changed.*float64.*uint64"):
136: (8)                           res = ufunc(np.uint64(1), 2**63)
137: (4)                         else:
138: (8)                           res = ufunc(np.uint64(1), 2**63)
139: (4)                         assert res.dtype == np.uint64
140: (8)                         assert res == ufunc(1, 2**63, dtype=object)
141: (4)                         with pytest.raises(OverflowError):
142: (8)                           ufunc(np.int64(1), 2**63) # np.array(2**63) would go to uint
143: (4)                         with pytest.raises(OverflowError):
144: (8)                           ufunc(np.int64(1), 2**100) # np.array(2**100) would go to object
145: (4)                           res = ufunc(1.0, 2**100)
146: (0)                           assert isinstance(res, np.float64)
147: (4) def test_nep50_in_concat_and_choose():
148: (4)     np._set_promotion_state("weak_and_warn")
149: (4)     with pytest.warns(UserWarning, match="result dtype changed"):
150: (8)       res = np.concatenate([np.float32(1), 1.], axis=None)
151: (4)       assert res.dtype == "float32"
152: (4)       with pytest.warns(UserWarning, match="result dtype changed"):
153: (8)         res = np.choose(1, [np.float32(1), 1.])
154: (4)         assert res.dtype == "float32"

```

---

## File 111 - test\_numeric.py:

```

1: (0)             import sys
2: (0)             import warnings
3: (0)             import itertools
4: (0)             import platform
5: (0)             import pytest
6: (0)             import math
7: (0)             from decimal import Decimal
8: (0)             import numpy as np
9: (0)             from numpy.core import umath
10: (0)            from numpy.random import rand, randint, randn
11: (0)            from numpy.testing import (
12: (4)              assert_, assert_equal, assert_raises, assert_raises_regex,
13: (4)              assert_array_equal, assert_almost_equal, assert_array_almost_equal,
14: (4)              assert_warnings, assert_array_max_ulp, HAS_REFCOUNT, IS_WASM
15: (4)            )
16: (0)            from numpy.core._rational_tests import rational
17: (0)            from hypothesis import given, strategies as st
18: (0)            from hypothesis.extra import numpy as hynp
19: (0)            class TestResize:
20: (4)              def test_copies(self):
21: (8)                A = np.array([[1, 2], [3, 4]])
22: (8)                Ar1 = np.array([[1, 2, 3, 4], [1, 2, 3, 4]])
23: (8)                assert_equal(np.resize(A, (2, 4)), Ar1)
24: (8)                Ar2 = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
25: (8)                assert_equal(np.resize(A, (4, 2)), Ar2)
26: (8)                Ar3 = np.array([[1, 2, 3], [4, 1, 2], [3, 4, 1], [2, 3, 4]])
27: (8)                assert_equal(np.resize(A, (4, 3)), Ar3)
28: (4)              def test_repeats(self):
29: (8)                A = np.array([1, 2, 3])
30: (8)                Ar1 = np.array([[1, 2, 3, 1], [2, 3, 1, 2]])
31: (8)                assert_equal(np.resize(A, (2, 4)), Ar1)
32: (8)                Ar2 = np.array([[1, 2], [3, 1], [2, 3], [1, 2]])
33: (8)                assert_equal(np.resize(A, (4, 2)), Ar2)

```

```

34: (8)             Ar3 = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]])
35: (8)             assert_equal(np.resize(A, (4, 3)), Ar3)
36: (4)             def test_zeroresize(self):
37: (8)                 A = np.array([[1, 2], [3, 4]])
38: (8)                 Ar = np.resize(A, (0,))
39: (8)                 assert_array_equal(Ar, np.array([]))
40: (8)                 assert_equal(A.dtype, Ar.dtype)
41: (8)                 Ar = np.resize(A, (0, 2))
42: (8)                 assert_equal(Ar.shape, (0, 2))
43: (8)                 Ar = np.resize(A, (2, 0))
44: (8)                 assert_equal(Ar.shape, (2, 0))
45: (4)             def test_reshape_from_zero(self):
46: (8)                 A = np.zeros(0, dtype=[('a', np.float32)])
47: (8)                 Ar = np.resize(A, (2, 1))
48: (8)                 assert_array_equal(Ar, np.zeros((2, 1), Ar.dtype))
49: (8)                 assert_equal(A.dtype, Ar.dtype)
50: (4)             def test_negative_resize(self):
51: (8)                 A = np.arange(0, 10, dtype=np.float32)
52: (8)                 new_shape = (-10, -1)
53: (8)                 with pytest.raises(ValueError, match=r"negative"):
54: (12)                     np.resize(A, new_shape=new_shape)
55: (4)             def test_subclass(self):
56: (8)                 class MyArray(np.ndarray):
57: (12)                     __array_priority__ = 1.
58: (8)                     my_arr = np.array([1]).view(MyArray)
59: (8)                     assert type(np.resize(my_arr, 5)) is MyArray
60: (8)                     assert type(np.resize(my_arr, 0)) is MyArray
61: (8)                     my_arr = np.array([]).view(MyArray)
62: (8)                     assert type(np.resize(my_arr, 5)) is MyArray
63: (0)             class TestNonarrayArgs:
64: (4)                 def test_choose(self):
65: (8)                     choices = [[0, 1, 2],
66: (19)                         [3, 4, 5],
67: (19)                         [5, 6, 7]]
68: (8)                     tgt = [5, 1, 5]
69: (8)                     a = [2, 0, 1]
70: (8)                     out = np.choose(a, choices)
71: (8)                     assert_equal(out, tgt)
72: (4)                 def test_clip(self):
73: (8)                     arr = [-1, 5, 2, 3, 10, -4, -9]
74: (8)                     out = np.clip(arr, 2, 7)
75: (8)                     tgt = [2, 5, 2, 3, 7, 2, 2]
76: (8)                     assert_equal(out, tgt)
77: (4)                 def test_compress(self):
78: (8)                     arr = [[0, 1, 2, 3, 4],
79: (15)                         [5, 6, 7, 8, 9]]
80: (8)                     tgt = [[5, 6, 7, 8, 9]]
81: (8)                     out = np.compress([0, 1], arr, axis=0)
82: (8)                     assert_equal(out, tgt)
83: (4)                 def test_count_nonzero(self):
84: (8)                     arr = [[0, 1, 7, 0, 0],
85: (15)                         [3, 0, 0, 2, 19]]
86: (8)                     tgt = np.array([2, 3])
87: (8)                     out = np.count_nonzero(arr, axis=1)
88: (8)                     assert_equal(out, tgt)
89: (4)                 def test_cumproduct(self):
90: (8)                     A = [[1, 2, 3], [4, 5, 6]]
91: (8)                     with assert_warns(DeprecationWarning):
92: (12)                         expected = np.array([1, 2, 6, 24, 120, 720])
93: (12)                         assert_(np.all(np.cumproduct(A) == expected))
94: (4)                 def test_diagonal(self):
95: (8)                     a = [[0, 1, 2, 3],
96: (13)                         [4, 5, 6, 7],
97: (13)                         [8, 9, 10, 11]]
98: (8)                     out = np.diagonal(a)
99: (8)                     tgt = [0, 5, 10]
100: (8)                     assert_equal(out, tgt)
101: (4)                 def test_mean(self):
102: (8)                     A = [[1, 2, 3], [4, 5, 6]]

```

```

103: (8) assert_(np.mean(A) == 3.5)
104: (8) assert_(np.all(np.mean(A, 0) == np.array([2.5, 3.5, 4.5])))
105: (8) assert_(np.all(np.mean(A, 1) == np.array([2., 5.])))
106: (8) with warnings.catch_warnings(record=True) as w:
107: (12)     warnings.filterwarnings('always', '', RuntimeWarning)
108: (12)     assert_(np.isnan(np.mean([])))
109: (12)     assert_(w[0].category is RuntimeWarning)
110: (4) def test_ptp(self):
111: (8)     a = [3, 4, 5, 10, -3, -5, 6.0]
112: (8)     assert_equal(np.ptp(a, axis=0), 15.0)
113: (4) def test_prod(self):
114: (8)     arr = [[1, 2, 3, 4],
115: (15)         [5, 6, 7, 9],
116: (15)         [10, 3, 4, 5]]
117: (8)     tgt = [24, 1890, 600]
118: (8)     assert_equal(np.prod(arr, axis=-1), tgt)
119: (4) def test_ravel(self):
120: (8)     a = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
121: (8)     tgt = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
122: (8)     assert_equal(np.ravel(a), tgt)
123: (4) def test_repeat(self):
124: (8)     a = [1, 2, 3]
125: (8)     tgt = [1, 1, 2, 2, 3, 3]
126: (8)     out = np.repeat(a, 2)
127: (8)     assert_equal(out, tgt)
128: (4) def test_reshape(self):
129: (8)     arr = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
130: (8)     tgt = [[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]]
131: (8)     assert_equal(np.reshape(arr, (2, 6)), tgt)
132: (4) def test_round(self):
133: (8)     arr = [1.56, 72.54, 6.35, 3.25]
134: (8)     tgt = [1.6, 72.5, 6.4, 3.2]
135: (8)     assert_equal(np.around(arr, decimals=1), tgt)
136: (8)     s = np.float64(1.)
137: (8)     assert_(isinstance(s.round(), np.float64))
138: (8)     assert_equal(s.round(), 1.)
139: (4) @pytest.mark.parametrize('dtype', [
140: (8)     np.int8, np.int16, np.int32, np.int64,
141: (8)     np.uint8, np.uint16, np.uint32, np.uint64,
142: (8)     np.float16, np.float32, np.float64,
143: (4) ])
144: (4) def test_dunder_round(self, dtype):
145: (8)     s = dtype(1)
146: (8)     assert_(isinstance(round(s), int))
147: (8)     assert_(isinstance(round(s, None), int))
148: (8)     assert_(isinstance(round(s, ndigits=None), int))
149: (8)     assert_equal(round(s), 1)
150: (8)     assert_equal(round(s, None), 1)
151: (8)     assert_equal(round(s, ndigits=None), 1)
152: (4) @pytest.mark.parametrize('val, ndigits', [
153: (8)     pytest.param(2**31 - 1, -1,
154: (12)         marks=pytest.mark.xfail(reason="Out of range of int32")
155: (8)     ),
156: (8)     (2**31 - 1, 1-math.ceil(math.log10(2**31 - 1))),
157: (8)     (2**31 - 1, -math.ceil(math.log10(2**31 - 1)))
158: (4) ])
159: (4) def test_dunder_round_edgecases(self, val, ndigits):
160: (8)     assert_equal(round(val, ndigits), round(np.int32(val), ndigits))
161: (4) def test_dunder_round_accuracy(self):
162: (8)     f = np.float64(5.1 * 10**73)
163: (8)     assert_(isinstance(round(f, -73), np.float64))
164: (8)     assert_array_max_ulp(round(f, -73), 5.0 * 10**73)
165: (8)     assert_(isinstance(round(f, ndigits=-73), np.float64))
166: (8)     assert_array_max_ulp(round(f, ndigits=-73), 5.0 * 10**73)
167: (8)     i = np.int64(501)
168: (8)     assert_(isinstance(round(i, -2), np.int64))
169: (8)     assert_array_max_ulp(round(i, -2), 500)
170: (8)     assert_(isinstance(round(i, ndigits=-2), np.int64))
171: (8)     assert_array_max_ulp(round(i, ndigits=-2), 500)

```

```

172: (4)          @pytest.mark.xfail(raises=AssertionError, reason="gh-15896")
173: (4)          def test_round_py_consistency(self):
174: (8)              f = 5.1 * 10**73
175: (8)              assert_equal(round(np.float64(f), -73), round(f, -73))
176: (4)          def test_searchsorted(self):
177: (8)              arr = [-8, -5, -1, 3, 6, 10]
178: (8)              out = np.searchsorted(arr, 0)
179: (8)              assert_equal(out, 3)
180: (4)          def test_size(self):
181: (8)              A = [[1, 2, 3], [4, 5, 6]]
182: (8)              assert_(np.size(A) == 6)
183: (8)              assert_(np.size(A, 0) == 2)
184: (8)              assert_(np.size(A, 1) == 3)
185: (4)          def test_squeeze(self):
186: (8)              A = [[[1, 1, 1], [2, 2, 2], [3, 3, 3]]]
187: (8)              assert_equal(np.squeeze(A).shape, (3, 3))
188: (8)              assert_equal(np.squeeze(np.zeros((1, 3, 1))).shape, (3,))
189: (8)              assert_equal(np.squeeze(np.zeros((1, 3, 1)), axis=0).shape, (3, 1))
190: (8)              assert_equal(np.squeeze(np.zeros((1, 3, 1)), axis=-1).shape, (1, 3))
191: (8)              assert_equal(np.squeeze(np.zeros((1, 3, 1)), axis=2).shape, (1, 3))
192: (8)              assert_equal(np.squeeze([np.zeros((3, 1))]).shape, (3,))
193: (8)              assert_equal(np.squeeze([np.zeros((3, 1))], axis=0).shape, (3, 1))
194: (8)              assert_equal(np.squeeze([np.zeros((3, 1))], axis=2).shape, (1, 3))
195: (8)              assert_equal(np.squeeze([np.zeros((3, 1))], axis=-1).shape, (1, 3))
196: (4)          def test_std(self):
197: (8)              A = [[1, 2, 3], [4, 5, 6]]
198: (8)              assert_almost_equal(np.std(A), 1.707825127659933)
199: (8)              assert_almost_equal(np.std(A, 0), np.array([1.5, 1.5, 1.5]))
200: (8)              assert_almost_equal(np.std(A, 1), np.array([0.81649658, 0.81649658]))
201: (8)              with warnings.catch_warnings(record=True) as w:
202: (12)                  warnings.filterwarnings('always', '', RuntimeWarning)
203: (12)                  assert_(np.isnan(np.std([])))
204: (12)                  assert_(w[0].category is RuntimeWarning)
205: (4)          def test_swapaxes(self):
206: (8)              tgt = [[[0, 4], [2, 6]], [[1, 5], [3, 7]]]
207: (8)              a = [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
208: (8)              out = np.swapaxes(a, 0, 2)
209: (8)              assert_equal(out, tgt)
210: (4)          def test_sum(self):
211: (8)              m = [[1, 2, 3],
212: (13)                  [4, 5, 6],
213: (13)                  [7, 8, 9]]
214: (8)              tgt = [[6], [15], [24]]
215: (8)              out = np.sum(m, axis=1, keepdims=True)
216: (8)              assert_equal(tgt, out)
217: (4)          def test_take(self):
218: (8)              tgt = [2, 3, 5]
219: (8)              indices = [1, 2, 4]
220: (8)              a = [1, 2, 3, 4, 5]
221: (8)              out = np.take(a, indices)
222: (8)              assert_equal(out, tgt)
223: (4)          def test_trace(self):
224: (8)              c = [[1, 2], [3, 4], [5, 6]]
225: (8)              assert_equal(np.trace(c), 5)
226: (4)          def test_transpose(self):
227: (8)              arr = [[1, 2], [3, 4], [5, 6]]
228: (8)              tgt = [[1, 3, 5], [2, 4, 6]]
229: (8)              assert_equal(np.transpose(arr, (1, 0)), tgt)
230: (4)          def test_var(self):
231: (8)              A = [[1, 2, 3], [4, 5, 6]]
232: (8)              assert_almost_equal(np.var(A), 2.9166666666666665)
233: (8)              assert_almost_equal(np.var(A, 0), np.array([2.25, 2.25, 2.25]))
234: (8)              assert_almost_equal(np.var(A, 1), np.array([0.66666667, 0.66666667]))
235: (8)              with warnings.catch_warnings(record=True) as w:
236: (12)                  warnings.filterwarnings('always', '', RuntimeWarning)
237: (12)                  assert_(np.isnan(np.var([])))
238: (12)                  assert_(w[0].category is RuntimeWarning)
239: (8)              B = np.array([None, 0])
240: (8)              B[0] = 1j

```

```

241: (8)           assert_almost_equal(np.var(B), 0.25)
242: (0)
243: (4)         class TestIscalar:
244: (8)             def test_isscalar(self):
245: (8)                 assert_(np.isscalar(3.1))
246: (8)                 assert_(np.isscalar(np.int16(12345)))
247: (8)                 assert_(np.isscalar(False))
248: (8)                 assert_(np.isscalar('numpy'))
249: (8)                 assert_(not np.isscalar([3.1]))
250: (8)                 assert_(not np.isscalar(None))
251: (8)                 from fractions import Fraction
252: (8)                 assert_(np.isscalar(Fraction(5, 17)))
253: (8)                 from numbers import Number
254: (8)                 assert_(np.isscalar(Number()))
254: (0)
255: (4)         class TestBoolScalar:
256: (8)             def test_logical(self):
257: (8)                 f = np.False_
258: (8)                 t = np.True_
259: (8)                 s = "xyz"
260: (8)                 assert_((t and s) is s)
261: (4)                 assert_((f and s) is f)
262: (8)             def test_bitwise_or(self):
263: (8)                 f = np.False_
264: (8)                 t = np.True_
265: (8)                 assert_((t | t) is t)
266: (8)                 assert_((f | t) is t)
267: (8)                 assert_((t | f) is t)
268: (4)                 assert_((f | f) is f)
268: (4)             def test_bitwise_and(self):
269: (8)                 f = np.False_
270: (8)                 t = np.True_
271: (8)                 assert_((t & t) is t)
272: (8)                 assert_((f & t) is f)
273: (8)                 assert_((t & f) is f)
274: (8)                 assert_((f & f) is f)
275: (4)             def test_bitwise_xor(self):
276: (8)                 f = np.False_
277: (8)                 t = np.True_
278: (8)                 assert_((t ^ t) is f)
279: (8)                 assert_((f ^ t) is t)
280: (8)                 assert_((t ^ f) is t)
281: (8)                 assert_((f ^ f) is f)
282: (0)
283: (4)         class TestBoolArray:
284: (8)             def setup_method(self):
285: (8)                 self.t = np.array([True] * 41, dtype=bool)[1::]
286: (8)                 self.f = np.array([False] * 41, dtype=bool)[1::]
287: (8)                 self.o = np.array([False] * 42, dtype=bool)[2::]
288: (8)                 self.nm = self.f.copy()
289: (8)                 self.im = self.t.copy()
290: (8)                 self.nm[3] = True
291: (8)                 self.nm[-2] = True
292: (8)                 self.im[3] = False
293: (4)                 self.im[-2] = False
294: (8)             def test_all_any(self):
295: (8)                 assert_(self.t.all())
296: (8)                 assert_(self.t.any())
297: (8)                 assert_(not self.f.all())
298: (8)                 assert_(not self.f.any())
299: (8)                 assert_(self.nm.any())
300: (8)                 assert_(self.im.any())
301: (8)                 assert_(not self.nm.all())
302: (8)                 assert_(not self.im.all())
302: (8)             for i in range(256 - 7):
303: (12)                 d = np.array([False] * 256, dtype=bool)[7::]
304: (12)                 d[i] = True
305: (12)                 assert_(np.any(d))
306: (12)                 e = np.array([True] * 256, dtype=bool)[7::]
307: (12)                 e[i] = False
308: (12)                 assert_(not np.all(e))
309: (12)                 assert_array_equal(e, ~d)

```

```

310: (8)
311: (12)
312: (12)
313: (12)
314: (12)
315: (12)
316: (12)
317: (4)
318: (8)
319: (8)
320: (8)
321: (8)
322: (8)
323: (8)
324: (8)
325: (8)
326: (8)
327: (8)
328: (8)
329: (4)
330: (8)
331: (8)
332: (8)
333: (8)
334: (8)
335: (8)
336: (8)
337: (8)
338: (8)
339: (8)
340: (8)
341: (8)
342: (8)
343: (8)
344: (8)
345: (8)
346: (8)
347: (8)
348: (8)
349: (8)
350: (8)
351: (8)
352: (8)
353: (8)
354: (8)
355: (8)
356: (8)
357: (8)
358: (8)
359: (8)
360: (0)
361: (4)
362: (8)
363: (8)
364: (8)
365: (8)
366: (8)
367: (8)
368: (12)
369: (12)
370: (12)
371: (8)
372: (8)
373: (12)
374: (12)
375: (12)
376: (8)
377: (8)
378: (8)

        for i in list(range(9, 6000, 507)) + [7764, 90021, -10]:
            d = np.array([False] * 100043, dtype=bool)
            d[i] = True
            assert_(np.any(d), msg="%r" % i)
            e = np.array([True] * 100043, dtype=bool)
            e[i] = False
            assert_(not np.all(e), msg="%r" % i)

    def test_logical_not_abs(self):
        assert_array_equal(~self.t, self.f)
        assert_array_equal(np.abs(~self.t), self.f)
        assert_array_equal(np.abs(~self.f), self.t)
        assert_array_equal(np.abs(self.f), self.f)
        assert_array_equal(~np.abs(self.f), self.t)
        assert_array_equal(~np.abs(self.t), self.f)
        assert_array_equal(np.abs(~self.nm), self.im)
        np.logical_not(self.t, out=self.o)
        assert_array_equal(self.o, self.f)
        np.abs(self.t, out=self.o)
        assert_array_equal(self.o, self.t)

    def test_logical_and_or_xor(self):
        assert_array_equal(self.t | self.t, self.t)
        assert_array_equal(self.f | self.f, self.f)
        assert_array_equal(self.t | self.f, self.t)
        assert_array_equal(self.f | self.t, self.t)
        np.logical_or(self.t, self.t, out=self.o)
        assert_array_equal(self.o, self.t)
        assert_array_equal(self.t & self.t, self.t)
        assert_array_equal(self.f & self.f, self.f)
        assert_array_equal(self.t & self.f, self.f)
        assert_array_equal(self.f & self.t, self.f)
        np.logical_and(self.t, self.t, out=self.o)
        assert_array_equal(self.o, self.t)
        assert_array_equal(self.t ^ self.t, self.f)
        assert_array_equal(self.f ^ self.f, self.f)
        assert_array_equal(self.t ^ self.f, self.t)
        assert_array_equal(self.f ^ self.t, self.t)
        np.logical_xor(self.t, self.t, out=self.o)
        assert_array_equal(self.o, self.f)
        assert_array_equal(self.nm & self.t, self.nm)
        assert_array_equal(self.im & self.f, False)
        assert_array_equal(self.nm & True, self.nm)
        assert_array_equal(self.im & False, self.f)
        assert_array_equal(self.nm | self.t, self.t)
        assert_array_equal(self.im | self.f, self.im)
        assert_array_equal(self.nm | True, self.t)
        assert_array_equal(self.im | False, self.im)
        assert_array_equal(self.nm ^ self.t, self.im)
        assert_array_equal(self.im ^ self.f, self.im)
        assert_array_equal(self.nm ^ True, self.im)
        assert_array_equal(self.im ^ False, self.im)

    class TestBoolCmp:
        def setup_method(self):
            self.f = np.ones(256, dtype=np.float32)
            self.ef = np.ones(self.f.size, dtype=bool)
            self.d = np.ones(128, dtype=np.float64)
            self.ed = np.ones(self.d.size, dtype=bool)
            s = 0
            for i in range(32):
                self.f[s:s+8] = [i & 2**x for x in range(8)]
                self.ef[s:s+8] = [(i & 2**x) != 0 for x in range(8)]
                s += 8
            s = 0
            for i in range(16):
                self.d[s:s+4] = [i & 2**x for x in range(4)]
                self.ed[s:s+4] = [(i & 2**x) != 0 for x in range(4)]
                s += 4
            self.nf = self.f.copy()
            self.nd = self.d.copy()
            self.nf[self.ef] = np.nan

```

```

379: (8)           self.nd[self.ed] = np.nan
380: (8)           self.inff = self.f.copy()
381: (8)           self.infd = self.d.copy()
382: (8)           self.inff[::3][self.ef[::3]] = np.inf
383: (8)           self.infd[::3][self.ed[::3]] = np.inf
384: (8)           self.inff[1::3][self.ef[1::3]] = -np.inf
385: (8)           self.infd[1::3][self.ed[1::3]] = -np.inf
386: (8)           self.inff[2::3][self.ef[2::3]] = np.nan
387: (8)           self.infd[2::3][self.ed[2::3]] = np.nan
388: (8)           self.efnonan = self.ef.copy()
389: (8)           self.efnonan[2::3] = False
390: (8)           self.ednonan = self.ed.copy()
391: (8)           self.ednonan[2::3] = False
392: (8)           self.signf = self.f.copy()
393: (8)           self.signd = self.d.copy()
394: (8)           self.signf[self.ef] *= -1.
395: (8)           self.signd[self.ed] *= -1.
396: (8)           self.signf[1::6][self.ef[1::6]] = -np.inf
397: (8)           self.signd[1::6][self.ed[1::6]] = -np.inf
398: (8)           if platform.processor() != 'riscv64':
399: (12)             self.signf[3::6][self.ef[3::6]] = -np.nan
400: (8)           self.signd[3::6][self.ed[3::6]] = -np.nan
401: (8)           self.signf[4::6][self.ef[4::6]] = -0.
402: (8)           self.signd[4::6][self.ed[4::6]] = -0.
403: (4)           def test_float(self):
404: (8)             for i in range(4):
405: (12)               assert_array_equal(self.f[i:] > 0, self.ef[i:])
406: (12)               assert_array_equal(self.f[i:] - 1 >= 0, self.ef[i:])
407: (12)               assert_array_equal(self.f[i:] == 0, ~self.ef[i:])
408: (12)               assert_array_equal(-self.f[i:] < 0, self.ef[i:])
409: (12)               assert_array_equal(-self.f[i:] + 1 <= 0, self.ef[i:])
410: (12)               r = self.f[i:] != 0
411: (12)               assert_array_equal(r, self.ef[i:])
412: (12)               r2 = self.f[i:] != np.zeros_like(self.f[i:])
413: (12)               r3 = 0 != self.f[i:]
414: (12)               assert_array_equal(r, r2)
415: (12)               assert_array_equal(r, r3)
416: (12)               assert_array_equal(r.view(np.int8), r.astype(np.int8))
417: (12)               assert_array_equal(r2.view(np.int8), r2.astype(np.int8))
418: (12)               assert_array_equal(r3.view(np.int8), r3.astype(np.int8))
419: (12)               assert_array_equal(np.isnan(self.nf[i:]), self.ef[i:])
420: (12)               assert_array_equal(np.isfinite(self.nf[i:]), ~self.ef[i:])
421: (12)               assert_array_equal(np.isfinite(self.inff[i:]), ~self.ef[i:])
422: (12)               assert_array_equal(np.isinf(self.inff[i:]), self.efnonan[i:])
423: (12)               assert_array_equal(np.signbit(self.signf[i:]), self.ef[i:])
424: (4)           def test_double(self):
425: (8)             for i in range(2):
426: (12)               assert_array_equal(self.d[i:] > 0, self.ed[i:])
427: (12)               assert_array_equal(self.d[i:] - 1 >= 0, self.ed[i:])
428: (12)               assert_array_equal(self.d[i:] == 0, ~self.ed[i:])
429: (12)               assert_array_equal(-self.d[i:] < 0, self.ed[i:])
430: (12)               assert_array_equal(-self.d[i:] + 1 <= 0, self.ed[i:])
431: (12)               r = self.d[i:] != 0
432: (12)               assert_array_equal(r, self.ed[i:])
433: (12)               r2 = self.d[i:] != np.zeros_like(self.d[i:])
434: (12)               r3 = 0 != self.d[i:]
435: (12)               assert_array_equal(r, r2)
436: (12)               assert_array_equal(r, r3)
437: (12)               assert_array_equal(r.view(np.int8), r.astype(np.int8))
438: (12)               assert_array_equal(r2.view(np.int8), r2.astype(np.int8))
439: (12)               assert_array_equal(r3.view(np.int8), r3.astype(np.int8))
440: (12)               assert_array_equal(np.isnan(self.nd[i:]), self.ed[i:])
441: (12)               assert_array_equal(np.isfinite(self.nd[i:]), ~self.ed[i:])
442: (12)               assert_array_equal(np.isfinite(self.infd[i:]), ~self.ed[i:])
443: (12)               assert_array_equal(np.isinf(self.infd[i:]), self.ednonan[i:])
444: (12)               assert_array_equal(np.signbit(self.signd[i:]), self.ed[i:])
445: (0)           class TestSeterr:
446: (4)             def test_default(self):
447: (8)               err = np.geterr()

```

```

448: (8)             assert_equal(err,
449: (21)             dict(divide='warn',
450: (26)                 invalid='warn',
451: (26)                 over='warn',
452: (26)                 under='ignore'))
453: (21)             )
454: (4)             def test_set(self):
455: (8)                 with np.errstate():
456: (12)                     err = np.seterr()
457: (12)                     old = np.seterr(divide='print')
458: (12)                     assert_(err == old)
459: (12)                     new = np.seterr()
460: (12)                     assert_(new['divide'] == 'print')
461: (12)                     np.seterr(over='raise')
462: (12)                     assert_(np.geterr()['over'] == 'raise')
463: (12)                     assert_(new['divide'] == 'print')
464: (12)                     np.seterr(**old)
465: (12)                     assert_(np.geterr() == old)
466: (4)             @pytest.mark.skipif(IS_WASM, reason="no wasm fp exception support")
467: (4)             @pytest.mark.skipif(platform.machine() == "armv5tel", reason="See gh-
413.")
468: (4)             def test_divide_err(self):
469: (8)                 with np.errstate(divide='raise'):
470: (12)                     with assert_raises(FloatingPointError):
471: (16)                         np.array([1.]) / np.array([0.])
472: (12)                         np.seterr(divide='ignore')
473: (12)                         np.array([1.]) / np.array([0.])
474: (4)             @pytest.mark.skipif(IS_WASM, reason="no wasm fp exception support")
475: (4)             def test_errobj(self):
476: (8)                 olderrobj = np.geterrobj()
477: (8)                 self.called = 0
478: (8)                 try:
479: (12)                     with warnings.catch_warnings(record=True) as w:
480: (16)                         warnings.simplefilter("always")
481: (16)                         with np.errstate(divide='warn'):
482: (20)                             np.seterrobj([20000, 1, None])
483: (20)                             np.array([1.]) / np.array([0.])
484: (20)                             assert_equal(len(w), 1)
485: (12)                 def log_err(*args):
486: (16)                     self.called += 1
487: (16)                     extobj_err = args
488: (16)                     assert_(len(extobj_err) == 2)
489: (16)                     assert_("divide" in extobj_err[0])
490: (12)                     with np.errstate(divide='ignore'):
491: (16)                         np.seterrobj([20000, 3, log_err])
492: (16)                         np.array([1.]) / np.array([0.])
493: (12)                         assert_equal(self.called, 1)
494: (12)                         np.seterrobj(olderrobj)
495: (12)                         with np.errstate(divide='ignore'):
496: (16)                             np.divide(1., 0., extobj=[20000, 3, log_err])
497: (12)                             assert_equal(self.called, 2)
498: (8)                         finally:
499: (12)                             np.seterrobj(olderrobj)
500: (12)                             del self.called
501: (4)             def test_errobj_noerrmask(self):
502: (8)                 olderrobj = np.geterrobj()
503: (8)                 try:
504: (12)                     np.seterrobj([umath.UFUNC_BUFSIZE_DEFAULT,
505: (25)                         umath.ERR_DEFAULT + 1, None])
506: (12)                     np.isnan(np.array([6]))
507: (12)                     for i in range(10000):
508: (16)                         np.seterrobj([umath.UFUNC_BUFSIZE_DEFAULT, umath.ERR_DEFAULT,
509: (29)                             None])
510: (12)                         np.isnan(np.array([6]))
511: (8)                         finally:
512: (12)                             np.seterrobj(olderrobj)
513: (0)             class TestFloatExceptions:
514: (4)                 def assert_raises_fpe(self, fpeerr, flop, x, y):
515: (8)                     ftype = type(x)

```

```

516: (8)
517: (12)
518: (12)
519: (20)
520: (8)
521: (12)
522: (20)
523: (4)
524: (8)
525: (8)
526: (8)
527: (8)
528: (4)
529: (4)
530: (4)
531: (8)
532: (12)
"
533: (31)
534: (31)
535: (8)
536: (12)
537: (12)
538: (16)
539: (16)
540: (16)
541: (16)
542: (16)
543: (16)
544: (12)
545: (16)
546: (16)
547: (16)
548: (16)
549: (16)
550: (16)
551: (16)
552: (12)
553: (12)
554: (12)
555: (16)
556: (36)
557: (16)
558: (36)
559: (12)
560: (35)
561: (12)
562: (35)
563: (12)
564: (35)
565: (12)
566: (35)
567: (12)
568: (35)
569: (12)
570: (35)
571: (12)
572: (16)
573: (12)
574: (12)
575: (35)
576: (12)
577: (16)
578: (12)
579: (12)
580: (16)
581: (12)
582: (12)
583: (35)

try:
    flop(x, y)
    assert_(False,
            "Type %s did not raise fpe error '%s'." % (ftype, fpeerr))
except FloatingPointError as exc:
    assert_(str(exc).find(fpeerr) >= 0,
            "Type %s raised wrong fpe error '%s'." % (ftype, exc))

def assert_op_raises_fpe(self, fpeerr, flop, sc1, sc2):
    self.assert_raises_fpe(fpeerr, flop, sc1, sc2)
    self.assert_raises_fpe(fpeerr, flop, sc1[()], sc2)
    self.assert_raises_fpe(fpeerr, flop, sc1, sc2[()])
    self.assert_raises_fpe(fpeerr, flop, sc1[()], sc2[()])

@pytest.mark.skipif(IS_WASM, reason="no wasm fp exception support")
@pytest.mark.parametrize("typecode", np.typecodes["AllFloat"])
def test_floating_exceptions(self, typecode):
    if 'bsd' in sys.platform and typecode in 'gG':
        pytest.skip(reason="Fallback impl for (c)longdouble may not raise
"
                           "FPE errors as expected on BSD OSes,
                           "see gh-24876, gh-23379")

    with np.errstate(all='raise'):
        ftype = np.obj2sctype(typecode)
        if np.dtype(ftype).kind == 'f':
            fi = np.finfo(ftype)
            ft_tiny = fi._machar.tiny
            ft_max = fi.max
            ft_eps = fi.eps
            underflow = 'underflow'
            divbyzero = 'divide by zero'
        else:
            rtype = type(ftype(0).real)
            fi = np.finfo(rtype)
            ft_tiny = ftype(fi._machar.tiny)
            ft_max = ftype(fi.max)
            ft_eps = ftype(fi.eps)
            underflow = ''
            divbyzero = ''
        overflow = 'overflow'
        invalid = 'invalid'
        if not np.isnan(ft_tiny):
            self.assert_raises_fpe(underflow,
                                  lambda a, b: a/b, ft_tiny, ft_max)
            self.assert_raises_fpe(underflow,
                                  lambda a, b: a*b, ft_tiny, ft_tiny)
        self.assert_raises_fpe(overflow,
                              lambda a, b: a*b, ft_max, ftype(2))
        self.assert_raises_fpe(overflow,
                              lambda a, b: a/b, ft_max, ftype(0.5))
        self.assert_raises_fpe(overflow,
                              lambda a, b: a+b, ft_max, ft_max*ft_eps)
        self.assert_raises_fpe(overflow,
                              lambda a, b: a-b, -ft_max, ft_max*ft_eps)
        self.assert_raises_fpe(overflow,
                              np.power, ftype(2), ftype(2**fi.nexp))
        self.assert_raises_fpe(divbyzero,
                              lambda a, b: a/b, ftype(1), ftype(0))
        self.assert_raises_fpe(
            invalid, lambda a, b: a/b, ftype(np.inf), ftype(np.inf))
        self.assert_raises_fpe(invalid,
                              lambda a, b: a/b, ftype(0), ftype(0))
        self.assert_raises_fpe(
            invalid, lambda a, b: a-b, ftype(np.inf), ftype(np.inf))
        self.assert_raises_fpe(
            invalid, lambda a, b: a+b, ftype(np.inf), ftype(-np.inf))
        self.assert_raises_fpe(invalid,
                              lambda a, b: a*b, ftype(0), ftype(np.inf))

```

```

584: (4) @pytest.mark.skipif(IS_WASM, reason="no wasm fp exception support")
585: (4) def test_warnings(self):
586: (8)     with warnings.catch_warnings(record=True) as w:
587: (12)         warnings.simplefilter("always")
588: (12)         with np.errstate(all="warn"):
589: (16)             np.divide(1, 0.)
590: (16)             assert_equal(len(w), 1)
591: (16)             assert_("divide by zero" in str(w[0].message))
592: (16)             np.array(1e300) * np.array(1e300)
593: (16)             assert_equal(len(w), 2)
594: (16)             assert_("overflow" in str(w[-1].message))
595: (16)             np.array(np.inf) - np.array(np.inf)
596: (16)             assert_equal(len(w), 3)
597: (16)             assert_("invalid value" in str(w[-1].message))
598: (16)             np.array(1e-300) * np.array(1e-300)
599: (16)             assert_equal(len(w), 4)
600: (16)             assert_("underflow" in str(w[-1].message))
601: (0) class TestTypes:
602: (4)     def check_promotion_cases(self, promote_func):
603: (8)         b = np.bool_(0)
604: (8)         i8, i16, i32, i64 = np.int8(0), np.int16(0), np.int32(0), np.int64(0)
605: (8)         u8, u16, u32, u64 = np.uint8(0), np.uint16(0), np.uint32(0),
np.uint64(0)
606: (8)         f32, f64, fld = np.float32(0), np.float64(0), np.longdouble(0)
607: (8)         c64, c128, cld = np.complex64(0), np.complex128(0), np.clongdouble(0)
608: (8)         assert_equal(promote_func(i8, i16), np.dtype(np.int16))
609: (8)         assert_equal(promote_func(i32, i8), np.dtype(np.int32))
610: (8)         assert_equal(promote_func(i16, i64), np.dtype(np.int64))
611: (8)         assert_equal(promote_func(u8, u32), np.dtype(np.uint32))
612: (8)         assert_equal(promote_func(f32, f64), np.dtype(np.float64))
613: (8)         assert_equal(promote_func(fld, f32), np.dtype(np.longdouble))
614: (8)         assert_equal(promote_func(f64, fld), np.dtype(np.longdouble))
615: (8)         assert_equal(promote_func(c128, c64), np.dtype(np.complex128))
616: (8)         assert_equal(promote_func(cld, c128), np.dtype(np.clongdouble))
617: (8)         assert_equal(promote_func(c64, fld), np.dtype(np.clongdouble))
618: (8)         assert_equal(promote_func(b, i32), np.dtype(np.int32))
619: (8)         assert_equal(promote_func(b, u8), np.dtype(np.uint8))
620: (8)         assert_equal(promote_func(i8, u8), np.dtype(np.int16))
621: (8)         assert_equal(promote_func(u8, i32), np.dtype(np.int32))
622: (8)         assert_equal(promote_func(i64, u32), np.dtype(np.int64))
623: (8)         assert_equal(promote_func(u64, i32), np.dtype(np.float64))
624: (8)         assert_equal(promote_func(i32, f32), np.dtype(np.float64))
625: (8)         assert_equal(promote_func(i64, f32), np.dtype(np.float64))
626: (8)         assert_equal(promote_func(f32, i16), np.dtype(np.float32))
627: (8)         assert_equal(promote_func(f32, u32), np.dtype(np.float64))
628: (8)         assert_equal(promote_func(f32, c64), np.dtype(np.complex64))
629: (8)         assert_equal(promote_func(c128, f32), np.dtype(np.complex128))
630: (8)         assert_equal(promote_func(cld, f64), np.dtype(np.clongdouble))
631: (8)         assert_equal(promote_func(np.array([b]), i8), np.dtype(np.int8))
632: (8)         assert_equal(promote_func(np.array([b]), u8), np.dtype(np.uint8))
633: (8)         assert_equal(promote_func(np.array([b]), i32), np.dtype(np.int32))
634: (8)         assert_equal(promote_func(np.array([b]), u32), np.dtype(np.uint32))
635: (8)         assert_equal(promote_func(np.array([i8]), i64), np.dtype(np.int8))
636: (8)         assert_equal(promote_func(u64, np.array([i32])), np.dtype(np.int32))
637: (8)         assert_equal(promote_func(i64, np.array([u32])), np.dtype(np.uint32))
638: (8)         assert_equal(promote_func(np.int32(-1), np.array([u64])),
639: (21)             np.dtype(np.float64))
640: (8)         assert_equal(promote_func(f64, np.array([f32])), np.dtype(np.float32))
641: (8)         assert_equal(promote_func(fld, np.array([f32])), np.dtype(np.float32))
642: (8)         assert_equal(promote_func(np.array([f64]), fld), np.dtype(np.float64))
643: (8)         assert_equal(promote_func(fld, np.array([c64])),
644: (21)             np.dtype(np.complex64))
645: (8)         assert_equal(promote_func(c64, np.array([f64])),
646: (21)             np.dtype(np.complex128))
647: (8)         assert_equal(promote_func(np.complex64(3j), np.array([f64])),
648: (21)             np.dtype(np.complex128))
649: (8)         assert_equal(promote_func(np.array([b]), f64), np.dtype(np.float64))
650: (8)         assert_equal(promote_func(np.array([b]), i64), np.dtype(np.int64))
651: (8)         assert_equal(promote_func(np.array([b]), u64), np.dtype(np.uint64))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

652: (8) assert_equal(promote_func(np.array([i8]), f64), np.dtype(np.float64))
653: (8) assert_equal(promote_func(np.array([u16]), f64), np.dtype(np.float64))
654: (8) assert_equal(promote_func(np.array([u16]), i32), np.dtype(np.uint16))
655: (8) assert_equal(promote_func(np.array([f32]), c128),
656: (21) np.dtype(np.complex64))
657: (4) def test_coercion(self):
658: (8)     def res_type(a, b):
659: (12)         return np.add(a, b).dtype
660: (8)     self.check_promotion_cases(res_type)
661: (8)     for a in [np.array([True, False]), np.array([-3, 12], dtype=np.int8)]:
662: (12)         b = 1.234 * a
663: (12)         assert_equal(b.dtype, np.dtype('f8'), "array type %s" % a.dtype)
664: (12)         b = np.longdouble(1.234) * a
665: (12)         assert_equal(b.dtype, np.dtype(np.longdouble),
666: (25)             "array type %s" % a.dtype)
667: (12)         b = np.float64(1.234) * a
668: (12)         assert_equal(b.dtype, np.dtype('f8'), "array type %s" % a.dtype)
669: (12)         b = np.float32(1.234) * a
670: (12)         assert_equal(b.dtype, np.dtype('f4'), "array type %s" % a.dtype)
671: (12)         b = np.float16(1.234) * a
672: (12)         assert_equal(b.dtype, np.dtype('f2'), "array type %s" % a.dtype)
673: (12)         b = 1.234j * a
674: (12)         assert_equal(b.dtype, np.dtype('c16'), "array type %s" % a.dtype)
675: (12)         b = np.clongdouble(1.234j) * a
676: (12)         assert_equal(b.dtype, np.dtype(np.clongdouble),
677: (25)             "array type %s" % a.dtype)
678: (12)         b = np.complex128(1.234j) * a
679: (12)         assert_equal(b.dtype, np.dtype('c16'), "array type %s" % a.dtype)
680: (12)         b = np.complex64(1.234j) * a
681: (12)         assert_equal(b.dtype, np.dtype('c8'), "array type %s" % a.dtype)
682: (4) def test_result_type(self):
683: (8)     self.check_promotion_cases(np.result_type)
684: (8)     assert_(np.result_type(None) == np.dtype(None))
685: (4) def test_promote_types_endian(self):
686: (8)     assert_equal(np.promote_types('<i8', '<i8'), np.dtype('i8'))
687: (8)     assert_equal(np.promote_types('>i8', '>i8'), np.dtype('i8'))
688: (8)     assert_equal(np.promote_types('>i8', '>U16'), np.dtype('U21'))
689: (8)     assert_equal(np.promote_types('<i8', '<U16'), np.dtype('U21'))
690: (8)     assert_equal(np.promote_types('>U16', '>i8'), np.dtype('U21'))
691: (8)     assert_equal(np.promote_types('<U16', '<i8'), np.dtype('U21'))
692: (8)     assert_equal(np.promote_types('<S5', '<U8'), np.dtype('U8'))
693: (8)     assert_equal(np.promote_types('>S5', '>U8'), np.dtype('U8'))
694: (8)     assert_equal(np.promote_types('<U8', '<S5'), np.dtype('U8'))
695: (8)     assert_equal(np.promote_types('>U8', '>S5'), np.dtype('U8'))
696: (8)     assert_equal(np.promote_types('<U5', '<U8'), np.dtype('U8'))
697: (8)     assert_equal(np.promote_types('>U8', '>U5'), np.dtype('U8'))
698: (8)     assert_equal(np.promote_types('<M8', '<M8'), np.dtype('M8'))
699: (8)     assert_equal(np.promote_types('>M8', '>M8'), np.dtype('M8'))
700: (8)     assert_equal(np.promote_types('<m8', '<m8'), np.dtype('m8'))
701: (8)     assert_equal(np.promote_types('>m8', '>m8'), np.dtype('m8'))
702: (4) def test_can_cast_and_promote_usertypes(self):
703: (8)     valid_types = ["int8", "int16", "int32", "int64", "bool"]
704: (8)     invalid_types = "BHILQP" + "FDG" + "mM" + "f" + "V"
705: (8)     rational_dt = np.dtype(rational)
706: (8)     for numpy_dtype in valid_types:
707: (12)         numpy_dtype = np.dtype(numpy_dtype)
708: (12)         assert np.can_cast(numpy_dtype, rational_dt)
709: (12)         assert np.promote_types(numpy_dtype, rational_dt) is rational_dt
710: (8)     for numpy_dtype in invalid_types:
711: (12)         numpy_dtype = np.dtype(numpy_dtype)
712: (12)         assert not np.can_cast(numpy_dtype, rational_dt)
713: (12)         with pytest.raises(TypeError):
714: (16)             np.promote_types(numpy_dtype, rational_dt)
715: (8)     double_dt = np.dtype("double")
716: (8)     assert np.can_cast(rational_dt, double_dt)
717: (8)     assert np.promote_types(double_dt, rational_dt) is double_dt
718: (4)     @pytest.mark.parametrize("swap", [ "", "swap"])
719: (4)     @pytest.mark.parametrize("string_dtype", [ "U", "S"])
720: (4)     def test_promote_types_strings(self, swap, string_dtype):

```

```

721: (8)
722: (12)
723: (8)
724: (12)
725: (8)
726: (8)
727: (8)
728: (8)
729: (8)
730: (8)
731: (8)
732: (8)
733: (8)
734: (8)
735: (8)
736: (8)
737: (8)
738: (8)
739: (8)
740: (8)
741: (8)
742: (8)
743: (8)
744: (8)
745: (8)
746: (8)
747: (8)
748: (8)
749: (4)
750: (12)
751: (13)
752: (12)
753: (4)
754: (8)
755: (12)
756: (4)
757: (12)
758: (13)
759: (14)
760: (13)
761: (13)
762: (12)
763: (4)
764: (8)
765: (4)
766: (12)
767: (12)
768: (4)
769: (8)
770: (8)
771: (8)
772: (8)
773: (8)
774: (8)
775: (12)
776: (8)
777: (8)
778: (12)
779: (8)
780: (12)
781: (8)
782: (4)
783: (4)
784: (4)
785: (12)
786: (16)
787: (16)
788: (16)
789: (4)

        if swap == "swap":
            promote_types = lambda a, b: np.promote_types(b, a)
        else:
            promote_types = np.promote_types
    S = string_dtype
    assert_equal(promote_types('bool', S), np.dtype(S+'5'))
    assert_equal(promote_types('b', S), np.dtype(S+'4'))
    assert_equal(promote_types('u1', S), np.dtype(S+'3'))
    assert_equal(promote_types('u2', S), np.dtype(S+'5'))
    assert_equal(promote_types('u4', S), np.dtype(S+'10'))
    assert_equal(promote_types('u8', S), np.dtype(S+'20'))
    assert_equal(promote_types('i1', S), np.dtype(S+'4'))
    assert_equal(promote_types('i2', S), np.dtype(S+'6'))
    assert_equal(promote_types('i4', S), np.dtype(S+'11'))
    assert_equal(promote_types('i8', S), np.dtype(S+'21'))
    assert_equal(promote_types('bool', S+'1'), np.dtype(S+'5'))
    assert_equal(promote_types('bool', S+'30'), np.dtype(S+'30'))
    assert_equal(promote_types('b', S+'1'), np.dtype(S+'4'))
    assert_equal(promote_types('b', S+'30'), np.dtype(S+'30'))
    assert_equal(promote_types('u1', S+'1'), np.dtype(S+'3'))
    assert_equal(promote_types('u1', S+'30'), np.dtype(S+'30'))
    assert_equal(promote_types('u2', S+'1'), np.dtype(S+'5'))
    assert_equal(promote_types('u2', S+'30'), np.dtype(S+'30'))
    assert_equal(promote_types('u4', S+'1'), np.dtype(S+'10'))
    assert_equal(promote_types('u4', S+'30'), np.dtype(S+'30'))
    assert_equal(promote_types('u8', S+'1'), np.dtype(S+'20'))
    assert_equal(promote_types('u8', S+'30'), np.dtype(S+'30'))
    assert_equal(promote_types('O', S+'30'), np.dtype('O'))

@pytest.mark.parametrize(["dtype1", "dtype2"],
    [[np.dtype("V6"), np.dtype("V10")], # mismatch shape
     [np.dtype([("name1", "i8")]), np.dtype([("name2", "i8")])],
    ])

def test_invalid_void_promotion(self, dtype1, dtype2):
    with pytest.raises(TypeError):
        np.promote_types(dtype1, dtype2)

@pytest.mark.parametrize(["dtype1", "dtype2"],
    [[np.dtype("V10"), np.dtype("V10")],
     [np.dtype([("name1", "i8")]),
      np.dtype([("name1", np.dtype("i8").newbyteorder())])],
     [np.dtype("i8,i8"), np.dtype("i8,>i8")],
     [np.dtype("i8,i8"), np.dtype("i4,i4")],
    ])

def test_valid_void_promotion(self, dtype1, dtype2):
    assert np.promote_types(dtype1, dtype2) == dtype1

@pytest.mark.parametrize("dtype",
    list(np.typecodes["All"]) +
    ["i,i", "10i", "S3", "S100", "U3", "U100", "rational"])

def test_promote_identical_types_metadata(self, dtype):
    metadata = {1: 1}
    dtype = np.dtype(dtype, metadata=metadata)
    res = np.promote_types(dtype, dtype)
    assert res.metadata == dtype.metadata
    dtype = dtype.newbyteorder()
    if dtype.isnative:
        return
    res = np.promote_types(dtype, dtype)
    if dtype.char != "U":
        assert res.metadata is None
    else:
        assert res.metadata == metadata
    assert res.isnative

@pytest.mark.slow
@pytest.mark.filterwarnings('ignore:Promotion of numbers:FutureWarning')
@pytest.mark.parametrize(["dtype1", "dtype2"],
    itertools.product(
        list(np.typecodes["All"]) +
        ["i,i", "S3", "S100", "U3", "U100", "rational"],
        repeat=2))

def test_promote_types_metadata(self, dtype1, dtype2):

```

```
790: (8)         """Metadata handling in promotion does not appear formalized
791: (8)         right now in NumPy. This test should thus be considered to
792: (8)         document behaviour, rather than test the correct definition of it.
793: (8)         This test is very ugly, it was useful for rewriting part of the
794: (8)         promotion, but probably should eventually be replaced/deleted
795: (8)         (i.e. when metadata handling in promotion is better defined).
796: (8)
797: (8)         metadata1 = {1: 1}
798: (8)         metadata2 = {2: 2}
799: (8)         dtype1 = np.dtype(dtype1, metadata=metadata1)
800: (8)         dtype2 = np.dtype(dtype2, metadata=metadata2)
801: (8)         try:
802: (12)             res = np.promote_types(dtype1, dtype2)
803: (8)         except TypeError:
804: (12)             return
805: (8)         if res.char not in "USV" or res.names is not None or res.shape != ():
806: (12)             assert res.metadata is None
807: (8)         elif res == dtype1:
808: (12)             assert res is dtype1
809: (8)         elif res == dtype2:
810: (12)             if np.promote_types(dtype1, dtype2.kind) == dtype2:
811: (16)                 res.metadata is None
812: (12)             else:
813: (16)                 res.metadata == metadata2
814: (8)         else:
815: (12)             assert res.metadata is None
816: (8)         dtype1 = dtype1.newbyteorder()
817: (8)         assert dtype1.metadata == metadata1
818: (8)         res_bs = np.promote_types(dtype1, dtype2)
819: (8)         assert res_bs == res
820: (8)         assert res_bs.metadata == res.metadata
821: (4)         def test_can_cast(self):
822: (8)             assert_(np.can_cast(np.int32, np.int64))
823: (8)             assert_(np.can_cast(np.float64, complex))
824: (8)             assert_(not np.can_cast(complex, float))
825: (8)             assert_(np.can_cast('i8', 'f8'))
826: (8)             assert_(not np.can_cast('i8', 'f4'))
827: (8)             assert_(np.can_cast('i4', 'S11'))
828: (8)             assert_(np.can_cast('i8', 'i8', 'no'))
829: (8)             assert_(not np.can_cast('<i8', '>i8', 'no'))
830: (8)             assert_(np.can_cast('<i8', '>i8', 'equiv'))
831: (8)             assert_(not np.can_cast('<i4', '>i8', 'equiv'))
832: (8)             assert_(np.can_cast('<i4', '>i8', 'safe'))
833: (8)             assert_(not np.can_cast('<i8', '>i4', 'safe'))
834: (8)             assert_(np.can_cast('<i8', '>i4', 'same_kind'))
835: (8)             assert_(not np.can_cast('<i8', '>u4', 'same_kind'))
836: (8)             assert_(np.can_cast('<i8', '>u4', 'unsafe'))
837: (8)             assert_(np.can_cast('bool', 'S5'))
838: (8)             assert_(not np.can_cast('bool', 'S4'))
839: (8)             assert_(np.can_cast('b', 'S4'))
840: (8)             assert_(not np.can_cast('b', 'S3'))
841: (8)             assert_(np.can_cast('u1', 'S3'))
842: (8)             assert_(not np.can_cast('u1', 'S2'))
843: (8)             assert_(np.can_cast('u2', 'S5'))
844: (8)             assert_(not np.can_cast('u2', 'S4'))
845: (8)             assert_(np.can_cast('u4', 'S10'))
846: (8)             assert_(not np.can_cast('u4', 'S9'))
847: (8)             assert_(np.can_cast('u8', 'S20'))
848: (8)             assert_(not np.can_cast('u8', 'S19'))
849: (8)             assert_(np.can_cast('i1', 'S4'))
850: (8)             assert_(not np.can_cast('i1', 'S3'))
851: (8)             assert_(np.can_cast('i2', 'S6'))
852: (8)             assert_(not np.can_cast('i2', 'S5'))
853: (8)             assert_(np.can_cast('i4', 'S11'))
854: (8)             assert_(not np.can_cast('i4', 'S10'))
855: (8)             assert_(np.can_cast('i8', 'S21'))
856: (8)             assert_(not np.can_cast('i8', 'S20'))
857: (8)             assert_(np.can_cast('bool', 'S5'))
858: (8)             assert_(not np.can_cast('bool', 'S4'))
```

```

859: (8) assert_(np.can_cast('b', 'U4'))
860: (8) assert_(not np.can_cast('b', 'U3'))
861: (8) assert_(np.can_cast('u1', 'U3'))
862: (8) assert_(not np.can_cast('u1', 'U2'))
863: (8) assert_(np.can_cast('u2', 'U5'))
864: (8) assert_(not np.can_cast('u2', 'U4'))
865: (8) assert_(np.can_cast('u4', 'U10'))
866: (8) assert_(not np.can_cast('u4', 'U9'))
867: (8) assert_(np.can_cast('u8', 'U20'))
868: (8) assert_(not np.can_cast('u8', 'U19'))
869: (8) assert_(np.can_cast('i1', 'U4'))
870: (8) assert_(not np.can_cast('i1', 'U3'))
871: (8) assert_(np.can_cast('i2', 'U6'))
872: (8) assert_(not np.can_cast('i2', 'U5'))
873: (8) assert_(np.can_cast('i4', 'U11'))
874: (8) assert_(not np.can_cast('i4', 'U10'))
875: (8) assert_(np.can_cast('i8', 'U21'))
876: (8) assert_(not np.can_cast('i8', 'U20'))
877: (8) assert_raises(TypeError, np.can_cast, 'i4', None)
878: (8) assert_raises(TypeError, np.can_cast, None, 'i4')
879: (8) assert_(np.can_cast(from_=np.int32, to=np.int64))
880: (4) def test_can_cast_simple_to_structured(self):
881: (8) assert_(not np.can_cast('i4', 'i4,i4'))
882: (8) assert_(not np.can_cast('i4', 'i4,i2'))
883: (8) assert_(np.can_cast('i4', 'i4,i4', casting='unsafe'))
884: (8) assert_(np.can_cast('i4', 'i4,i2', casting='unsafe'))
885: (8) assert_(not np.can_cast('i2', [('f1', 'i4')]))
886: (8) assert_(not np.can_cast('i2', [('f1', 'i4')], casting='same_kind'))
887: (8) assert_(np.can_cast('i2', [('f1', 'i4')], casting='unsafe'))
888: (8) assert_(not np.can_cast('i2', [('f1', 'i4,i4')]))
889: (8) assert_(np.can_cast('i2', [('f1', 'i4,i4')], casting='unsafe'))
890: (8) assert_(not np.can_cast('i2', [('f1', '(2,3)i4')]))
891: (8) assert_(np.can_cast('i2', [('f1', '(2,3)i4')], casting='unsafe'))
892: (4) def test_can_cast_structured_to_simple(self):
893: (8) assert_(not np.can_cast([('f1', 'i4')], 'i4'))
894: (8) assert_(np.can_cast([('f1', 'i4')], 'i4', casting='unsafe'))
895: (8) assert_(np.can_cast([('f1', 'i4')], 'i2', casting='unsafe'))
896: (8) assert_(not np.can_cast('i4,i4', 'i4', casting='unsafe'))
897: (8) assert_(not np.can_cast([('f1', [('x', 'i4')])], 'i4'))
898: (8) assert_(np.can_cast([('f1', [('x', 'i4')])], 'i4', casting='unsafe'))
899: (8) assert_(not np.can_cast([('f0', '(3,)i4')], 'i4'))
900: (8) assert_(np.can_cast([('f0', '(3,)i4')], 'i4', casting='unsafe'))
901: (8) assert_(not np.can_cast([('f0', ('i4,i4'), (2,))], 'i4',
902: (32) casting='unsafe'))
903: (4) def test_can_cast_values(self):
904: (8) for dt in np.sctypes['int'] + np.sctypes['uint']:
905: (12)     ii = np.iinfo(dt)
906: (12)     assert_(np.can_cast(ii.min, dt))
907: (12)     assert_(np.can_cast(ii.max, dt))
908: (12)     assert_(not np.can_cast(ii.min - 1, dt))
909: (12)     assert_(not np.can_cast(ii.max + 1, dt))
910: (8)     for dt in np.sctypes['float']:
911: (12)         fi = np.finfo(dt)
912: (12)         assert_(np.can_cast(fi.min, dt))
913: (12)         assert_(np.can_cast(fi.max, dt))
914: (0) class NIterError(Exception):
915: (4)     pass
916: (0) class TestFromiter:
917: (4)     def makegen(self):
918: (8)         return (x**2 for x in range(24))
919: (4)     def test_types(self):
920: (8)         ai32 = np.fromiter(self.makegen(), np.int32)
921: (8)         ai64 = np.fromiter(self.makegen(), np.int64)
922: (8)         af = np.fromiter(self.makegen(), float)
923: (8)         assert_(ai32.dtype == np.dtype(np.int32))
924: (8)         assert_(ai64.dtype == np.dtype(np.int64))
925: (8)         assert_(af.dtype == np.dtype(float))
926: (4)     def test_lengths(self):
927: (8)         expected = np.array(list(self.makegen()))

```

```

928: (8)             a = np.fromiter(self.makegen(), int)
929: (8)             a20 = np.fromiter(self.makegen(), int, 20)
930: (8)             assert_(len(a) == len(expected))
931: (8)             assert_(len(a20) == 20)
932: (8)             assert_raises(ValueError, np.fromiter,
933: (26)                         self.makegen(), int, len(expected) + 10)
934: (4)             def test_values(self):
935: (8)                 expected = np.array(list(self.makegen()))
936: (8)                 a = np.fromiter(self.makegen(), int)
937: (8)                 a20 = np.fromiter(self.makegen(), int, 20)
938: (8)                 assert_(np.all(a == expected, axis=0))
939: (8)                 assert_(np.all(a20 == expected[:20], axis=0))
940: (4)             def load_data(self, n, eindex):
941: (8)                 for e in range(n):
942: (12)                     if e == eindex:
943: (16)                         raise NIrror('error at index %s' % eindex)
944: (12)                     yield e
945: (4)             @pytest.mark.parametrize("dtype", [int, object])
946: (4)             @pytest.mark.parametrize(["count", "error_index"], [(10, 5), (10, 9)])
947: (4)             def test_2592(self, count, error_index, dtype):
948: (8)                 iterable = self.load_data(count, error_index)
949: (8)                 with pytest.raises(NIrror):
950: (12)                     np.fromiter(iterable, dtype=dtype, count=count)
951: (4)             @pytest.mark.parametrize("dtype", ["S", "S0", "V0", "U0"])
952: (4)             def test_empty_not_structured(self, dtype):
953: (8)                 with pytest.raises(ValueError, match="Must specify length"):
954: (12)                     np.fromiter([], dtype=dtype)
955: (4)             @pytest.mark.parametrize(["dtype", "data"],
956: (12)                         [("d", [1, 2, 3, 4, 5, 6, 7, 8, 9]),
957: (13)                             ("O", [1, 2, 3, 4, 5, 6, 7, 8, 9]),
958: (13)                             ("i,O", [(1, 2), (5, 4), (2, 3), (9, 8), (6, 7)]),
959: (13)                             ("2i", [(1, 2), (5, 4), (2, 3), (9, 8), (6, 7)]),
960: (13)                             (np.dtype("O", (2, 3))),
961: (14)                             [((1, 2, 3), (3, 4, 5)), ((3, 2, 1), (5, 4, 3))]))
962: (4)             @pytest.mark.parametrize("length_hint", [0, 1])
963: (4)             def test_growth_and_complicated_dtotypes(self, dtype, data, length_hint):
964: (8)                 dtype = np.dtype(dtype)
965: (8)                 data = data * 100 # make sure we realloc a bit
966: (8)                 class MyIter:
967: (12)                     def __length_hint__(self):
968: (16)                         return length_hint # 0 or 1
969: (12)                     def __iter__(self):
970: (16)                         return iter(data)
971: (8)                     res = np.fromiter(MyIter(), dtype=dtype)
972: (8)                     expected = np.array(data, dtype=dtype)
973: (8)                     assert_array_equal(res, expected)
974: (4)             def test_empty_result(self):
975: (8)                 class MyIter:
976: (12)                     def __length_hint__(self):
977: (16)                         return 10
978: (12)                     def __iter__(self):
979: (16)                         return iter([]) # actual iterator is empty.
980: (8)                     res = np.fromiter(MyIter(), dtype="d")
981: (8)                     assert res.shape == (0,)
982: (8)                     assert res.dtype == "d"
983: (4)             def test_too_few_items(self):
984: (8)                 msg = "iterator too short: Expected 10 but iterator had only 3 items."
985: (8)                 with pytest.raises(ValueError, match=msg):
986: (12)                     np.fromiter([1, 2, 3], count=10, dtype=int)
987: (4)             def test_failed_itemsetting(self):
988: (8)                 with pytest.raises(TypeError):
989: (12)                     np.fromiter([1, None, 3], dtype=int)
990: (8)                     iterable = ((2, 3, 4) for i in range(5))
991: (8)                     with pytest.raises(ValueError):
992: (12)                         np.fromiter(iterable, dtype=np.dtype((int, 2)))
993: (0)             class TestNonzero:
994: (4)                 def test_nonzero_trivial(self):
995: (8)                     assert_equal(np.count_nonzero(np.array([])), 0)
996: (8)                     assert_equal(np.count_nonzero(np.array[], dtype='?'), 0)

```

```

997: (8) assert_equal(np.nonzero(np.array([])), ([],))
998: (8) assert_equal(np.count_nonzero(np.array([0])), 0)
999: (8) assert_equal(np.count_nonzero(np.array([0], dtype='?')), 0)
1000: (8) assert_equal(np.nonzero(np.array([0])), ([],))
1001: (8) assert_equal(np.count_nonzero(np.array([1])), 1)
1002: (8) assert_equal(np.count_nonzero(np.array([1], dtype='?')), 1)
1003: (8) assert_equal(np.nonzero(np.array([1])), ([0],))
1004: (4) def test_nonzero_zerod(self):
1005: (8)     assert_equal(np.count_nonzero(np.array(0)), 0)
1006: (8)     assert_equal(np.count_nonzero(np.array(0, dtype='?')), 0)
1007: (8)     with assert_warns(DeprecationWarning):
1008: (12)         assert_equal(np.nonzero(np.array(0)), ([],))
1009: (8)     assert_equal(np.count_nonzero(np.array(1)), 1)
1010: (8)     assert_equal(np.count_nonzero(np.array(1, dtype='?')), 1)
1011: (8)     with assert_warns(DeprecationWarning):
1012: (12)         assert_equal(np.nonzero(np.array(1)), ([0],))
1013: (4) def test_nonzero_onedim(self):
1014: (8)     x = np.array([1, 0, 2, -1, 0, 0, 8])
1015: (8)     assert_equal(np.count_nonzero(x), 4)
1016: (8)     assert_equal(np.count_nonzero(x), 4)
1017: (8)     assert_equal(np.nonzero(x), ([0, 2, 3, 6],))
1018: (8)     x = np.array([(1, 2, -5, -3), (0, 0, 2, 7), (1, 1, 0, 1), (-1, 3, 1,
0), (0, 7, 0, 4)], dtype=[('a', 'i4'), ('b', 'i2'), ('c', 'i1'), ('d',
'i8')])])
1019: (21)
1020: (8)     assert_equal(np.count_nonzero(x['a']), 3)
1021: (8)     assert_equal(np.count_nonzero(x['b']), 4)
1022: (8)     assert_equal(np.count_nonzero(x['c']), 3)
1023: (8)     assert_equal(np.count_nonzero(x['d']), 4)
1024: (8)     assert_equal(np.nonzero(x['a']), ([0, 2, 3],))
1025: (8)     assert_equal(np.nonzero(x['b']), ([0, 2, 3, 4],))
1026: (4) def test_nonzero_twodim(self):
1027: (8)     x = np.array([[0, 1, 0], [2, 0, 3]])
1028: (8)     assert_equal(np.count_nonzero(x.astype('i1')), 3)
1029: (8)     assert_equal(np.count_nonzero(x.astype('i2')), 3)
1030: (8)     assert_equal(np.count_nonzero(x.astype('i4')), 3)
1031: (8)     assert_equal(np.count_nonzero(x.astype('i8')), 3)
1032: (8)     assert_equal(np.nonzero(x), ([0, 1, 1], [1, 0, 2]))
1033: (8)     x = np.eye(3)
1034: (8)     assert_equal(np.count_nonzero(x.astype('i1')), 3)
1035: (8)     assert_equal(np.count_nonzero(x.astype('i2')), 3)
1036: (8)     assert_equal(np.count_nonzero(x.astype('i4')), 3)
1037: (8)     assert_equal(np.count_nonzero(x.astype('i8')), 3)
1038: (8)     assert_equal(np.nonzero(x), ([0, 1, 2], [0, 1, 2]))
1039: (8)     x = np.array([(0, 1), (0, 0), (1, 11),
1040: (19)             [(1, 1), (1, 0), (0, 0)],
1041: (19)             [(0, 0), (1, 5), (0, 1)]], dtype=[('a', 'f4'), ('b',
'u1')]))
1042: (8)     assert_equal(np.count_nonzero(x['a']), 4)
1043: (8)     assert_equal(np.count_nonzero(x['b']), 5)
1044: (8)     assert_equal(np.nonzero(x['a']), ([0, 1, 1, 2], [2, 0, 1, 1]))
1045: (8)     assert_equal(np.nonzero(x['b']), ([0, 0, 1, 2, 2], [0, 2, 0, 1, 2]))
1046: (8)     assert_(not x['a'].T.flags.aligned)
1047: (8)     assert_equal(np.count_nonzero(x['a'].T), 4)
1048: (8)     assert_equal(np.count_nonzero(x['b'].T), 5)
1049: (8)     assert_equal(np.nonzero(x['a'].T), ([0, 1, 1, 2], [1, 1, 2, 0]))
1050: (8)     assert_equal(np.nonzero(x['b'].T), ([0, 0, 1, 2, 2], [0, 1, 2, 0, 2]))
1051: (4) def test_sparse(self):
1052: (8)     for i in range(20):
1053: (12)         c = np.zeros(200, dtype=bool)
1054: (12)         c[i::20] = True
1055: (12)         assert_equal(np.nonzero(c)[0], np.arange(i, 200 + i, 20))
1056: (12)         c = np.zeros(400, dtype=bool)
1057: (12)         c[10 + i:20 + i] = True
1058: (12)         c[20 + i*2] = True
1059: (12)         assert_equal(np.nonzero(c)[0],
1060: (25)             np.concatenate((np.arange(10 + i, 20 + i), [20 +
i*2])))
1061: (4)     def test_return_type(self):

```

```

1062: (8)
1063: (12)
1064: (8)
1065: (12)
1066: (16)
1067: (16)
1068: (16)
1069: (20)
1070: (24)
1071: (24)
1072: (4)
1073: (8)
1074: (8)
1075: (8)
1076: (8)
1077: (8)
1078: (8)
1079: (8)
1080: (8)
1081: (8)
1082: (22)
1083: (4)
1084: (8)
1085: (8)
1086: (12)
1087: (12)
1088: (8)
1089: (12)
1090: (12)
1091: (16)
1092: (20)
1093: (20)
1094: (20)
1095: (20)
1096: (16)
1097: (20)
1098: (20)
1099: (20)
1100: (20)
1101: (20)
1102: (16)
1103: (16)
1104: (34)
1105: (16)
1106: (16)
1107: (34)
1108: (16)
1109: (16)
1110: (29)
1111: (16)
1112: (29)
1113: (16)
1114: (29)
1115: (12)
1116: (16)
1117: (16)
1118: (16)
1119: (34)
1120: (16)
1121: (16)
1122: (34)
1123: (16)
1124: (16)
1125: (29)
1126: (16)
1127: (29)
1128: (16)
1129: (29)
1130: (4)

        class C(np.ndarray):
            pass
        for view in (C, np.ndarray):
            for nd in range(1, 4):
                shape = tuple(range(2, 2+nd))
                x = np.arange(np.prod(shape)).reshape(shape).view(view)
                for nzx in (np.nonzero(x), x.nonzero()):
                    for nzx_i in nzx:
                        assert_(type(nzx_i) is np.ndarray)
                        assert_(nzx_i.flags.writeable)
    def test_count_nonzero_axis(self):
        m = np.array([[0, 1, 7, 0, 0], [3, 0, 0, 2, 19]])
        expected = np.array([1, 1, 1, 1, 1])
        assert_equal(np.count_nonzero(m, axis=0), expected)
        expected = np.array([2, 3])
        assert_equal(np.count_nonzero(m, axis=1), expected)
        assert_raises(ValueError, np.count_nonzero, m, axis=(1, 1))
        assert_raises(TypeError, np.count_nonzero, m, axis='foo')
        assert_raises(np.AxisError, np.count_nonzero, m, axis=3)
        assert_raises(TypeError, np.count_nonzero,
                      m, axis=np.array([1, 2])))
    def test_count_nonzero_axis_all_dtypes(self):
        msg = "Mismatch for dtype: %s"
        def assert_equal_w_dt(a, b, err_msg):
            assert_equal(a.dtype, b.dtype, err_msg=err_msg)
            assert_equal(a, b, err_msg=err_msg)
        for dt in np.typecodes['All']:
            err_msg = msg % (np.dtype(dt).name,)
            if dt != 'V':
                if dt != 'M':
                    m = np.zeros((3, 3), dtype=dt)
                    n = np.ones(1, dtype=dt)
                    m[0, 0] = n[0]
                    m[1, 0] = n[0]
                else: # np.zeros doesn't work for np.datetime64
                    m = np.array(['1970-01-01'] * 9)
                    m = m.reshape((3, 3))
                    m[0, 0] = '1970-01-12'
                    m[1, 0] = '1970-01-12'
                    m = m.astype(dt)
            expected = np.array([2, 0, 0], dtype=np.intp)
            assert_equal_w_dt(np.count_nonzero(m, axis=0),
                             expected, err_msg=err_msg)
            expected = np.array([1, 1, 0], dtype=np.intp)
            assert_equal_w_dt(np.count_nonzero(m, axis=1),
                             expected, err_msg=err_msg)
            expected = np.array(2)
            assert_equal(np.count_nonzero(m, axis=(0, 1)),
                        expected, err_msg=err_msg)
            assert_equal(np.count_nonzero(m, axis=None),
                        expected, err_msg=err_msg)
            assert_equal(np.count_nonzero(m),
                        expected, err_msg=err_msg)
        if dt == 'V':
            m = np.array([np.void(1)] * 6).reshape((2, 3))
            expected = np.array([0, 0, 0], dtype=np.intp)
            assert_equal_w_dt(np.count_nonzero(m, axis=0),
                             expected, err_msg=err_msg)
            expected = np.array([0, 0], dtype=np.intp)
            assert_equal_w_dt(np.count_nonzero(m, axis=1),
                             expected, err_msg=err_msg)
            expected = np.array(0)
            assert_equal(np.count_nonzero(m, axis=(0, 1)),
                        expected, err_msg=err_msg)
            assert_equal(np.count_nonzero(m, axis=None),
                        expected, err_msg=err_msg)
            assert_equal(np.count_nonzero(m),
                        expected, err_msg=err_msg)
    def test_count_nonzero_axis_consistent(self):

```

```

1131: (8)             from itertools import combinations, permutations
1132: (8)             axis = (0, 1, 2, 3)
1133: (8)             size = (5, 5, 5, 5)
1134: (8)             msg = "Mismatch for axis: %s"
1135: (8)             rng = np.random.RandomState(1234)
1136: (8)             m = rng.randint(-100, 100, size=size)
1137: (8)             n = m.astype(object)
1138: (8)             for length in range(len(axis)):
1139: (12)                 for combo in combinations(axis, length):
1140: (16)                     for perm in permutations(combo):
1141: (20)                         assert_equal(
1142: (24)                             np.count_nonzero(m, axis=perm),
1143: (24)                             np.count_nonzero(n, axis=perm),
1144: (24)                             err_msg=msg % (perm,))
1145: (4)             def test_countnonzero_axis_empty(self):
1146: (8)                 a = np.array([[0, 0, 1], [1, 0, 1]])
1147: (8)                 assert_equal(np.count_nonzero(a, axis=()), a.astype(bool))
1148: (4)             def test_countnonzero_keepdims(self):
1149: (8)                 a = np.array([[0, 0, 1, 0],
1150: (22)                     [0, 3, 5, 0],
1151: (22)                     [7, 9, 2, 0]])
1152: (8)                 assert_equal(np.count_nonzero(a, axis=0, keepdims=True),
1153: (21)                     [[1, 2, 3, 0]])
1154: (8)                 assert_equal(np.count_nonzero(a, axis=1, keepdims=True),
1155: (21)                     [[1], [2], [3]])
1156: (8)                 assert_equal(np.count_nonzero(a, keepdims=True),
1157: (21)                     [[6]]])
1158: (4)             def test_array_method(self):
1159: (8)                 m = np.array([[1, 0, 0], [4, 0, 6]])
1160: (8)                 tgt = [[0, 1, 1], [0, 0, 2]]
1161: (8)                 assert_equal(m.nonzero(), tgt)
1162: (4)             def test_nonzero_invalid_object(self):
1163: (8)                 a = np.array([np.array([1, 2]), 3], dtype=object)
1164: (8)                 assert_raises(ValueError, np.nonzero, a)
1165: (8)                 class BoolErrors:
1166: (12)                     def __bool__(self):
1167: (16)                         raise ValueError("Not allowed")
1168: (8)                     assert_raises(ValueError, np.nonzero, np.array([BoolErrors()]))
1169: (4)             def test_nonzero_sideeffect_safety(self):
1170: (8)                 class FalseThenTrue:
1171: (12)                     _val = False
1172: (12)                     def __bool__(self):
1173: (16)                         try:
1174: (20)                             return self._val
1175: (16)                         finally:
1176: (20)                             self._val = True
1177: (8)                 class TrueThenFalse:
1178: (12)                     _val = True
1179: (12)                     def __bool__(self):
1180: (16)                         try:
1181: (20)                             return self._val
1182: (16)                         finally:
1183: (20)                             self._val = False
1184: (8)                 a = np.array([True, FalseThenTrue()])
1185: (8)                 assert_raises(RuntimeError, np.nonzero, a)
1186: (8)                 a = np.array([[True], [FalseThenTrue()]])
1187: (8)                 assert_raises(RuntimeError, np.nonzero, a)
1188: (8)                 a = np.array([False, TrueThenFalse()])
1189: (8)                 assert_raises(RuntimeError, np.nonzero, a)
1190: (8)                 a = np.array([[False], [TrueThenFalse()]])
1191: (8)                 assert_raises(RuntimeError, np.nonzero, a)
1192: (4)             def test_nonzero_sideeffects_structured_void(self):
1193: (8)                 arr = np.zeros(5, dtype="i1,i8,i8") # `ones` may short-circuit
1194: (8)                 assert arr.flags.aligned # structs are considered "aligned"
1195: (8)                 assert not arr["f2"].flags.aligned
1196: (8)                 np.nonzero(arr)
1197: (8)                 assert arr.flags.aligned
1198: (8)                 np.count_nonzero(arr)
1199: (8)                 assert arr.flags.aligned

```

```

1200: (4)           def test_nonzero_exception_safe(self):
1201: (8)             class ThrowsAfter:
1202: (12)               def __init__(self, iters):
1203: (16)                 self.iters_left = iters
1204: (12)               def __bool__(self):
1205: (16)                 if self.iters_left == 0:
1206: (20)                   raise ValueError("called `iters` times")
1207: (16)                 self.iters_left -= 1
1208: (16)               return True
1209: (8)
1210: (8)             """
1211: (8)             Test that a ValueError is raised instead of a SystemError
1212: (8)             If the __bool__ function is called after the error state is set,
1213: (8)             Python (cpython) will raise a SystemError.
1214: (8)
1215: (8)             """
1216: (8)             a = np.array([ThrowsAfter(5)]*10)
1217: (8)             assert_raises(ValueError, np.nonzero, a)
1218: (8)             a = np.array([ThrowsAfter(15)]*10)
1219: (8)             assert_raises(ValueError, np.nonzero, a)
1220: (8)             a = np.array([[ThrowsAfter(15)]]*10)
1221: (8)             assert_raises(ValueError, np.nonzero, a)
1222: (4)             @pytest.mark.skipif(IS_WASM, reason="wasm doesn't have threads")
1223: (4)             def test_structured_threadsafety(self):
1224: (8)               from concurrent.futures import ThreadPoolExecutor
1225: (8)               dt = np.dtype([("", "f8")])
1226: (8)               dt = np.dtype([("", dt)])
1227: (8)               dt = np.dtype([("", dt)] * 2)
1228: (8)               arr = np.random.uniform(size=(5000, 4)).view(dt)[:, 0]
1229: (8)               def func(arr):
1230: (8)                 arr.nonzero()
1231: (8)               tpe = ThreadPoolExecutor(max_workers=8)
1232: (8)               futures = [tpe.submit(func, arr) for _ in range(10)]
1233: (8)               for f in futures:
1234: (8)                 f.result()
1235: (8)               assert arr.dtype is dt
1236: (0)             class TestIndex:
1237: (4)               def test_boolean(self):
1238: (8)                 a = rand(3, 5, 8)
1239: (8)                 V = rand(5, 8)
1240: (8)                 g1 = randint(0, 5, size=15)
1241: (8)                 g2 = randint(0, 8, size=15)
1242: (8)                 V[g1, g2] = -V[g1, g2]
1243: (8)                 assert_((np.array([a[0][V > 0], a[1][V > 0], a[2][V > 0]])) == a[:, V > 0]).all()
1244: (4)               def test_boolean_edgecase(self):
1245: (8)                 a = np.array([], dtype='int32')
1246: (8)                 b = np.array([], dtype='bool')
1247: (8)                 c = a[b]
1248: (0)               assert_equal(c, [])
1249: (4)               assert_equal(c.dtype, np.dtype('int32'))
1250: (0)             class TestBinaryRepr:
1251: (4)               def test_zero(self):
1252: (8)                 assert_equal(np.binary_repr(0), '0')
1253: (4)               def test_positive(self):
1254: (8)                 assert_equal(np.binary_repr(10), '1010')
1255: (8)                 assert_equal(np.binary_repr(12522),
1256: (21)                               '11000011101010')
1257: (8)                 assert_equal(np.binary_repr(10736848),
1258: (21)                               '101000111101010011010000')
1259: (4)               def test_negative(self):
1260: (8)                 assert_equal(np.binary_repr(-1), '-1')
1261: (8)                 assert_equal(np.binary_repr(-10), '-1010')
1262: (8)                 assert_equal(np.binary_repr(-12522),
1263: (21)                               '-11000011101010')
1264: (4)               def test_sufficient_width(self):
1265: (8)                 assert_equal(np.binary_repr(0, width=5), '00000')
1266: (8)                 assert_equal(np.binary_repr(10, width=7), '0001010')
1267: (8)                 assert_equal(np.binary_repr(-5, width=7), '1111011')

```

```

1268: (4)           def test_neg_width_boundaries(self):
1269: (8)             assert_equal(np.binary_repr(-128, width=8), '10000000')
1270: (8)             for width in range(1, 11):
1271: (12)               num = -2**width - 1
1272: (12)               exp = '1' + (width - 1) * '0'
1273: (12)               assert_equal(np.binary_repr(num, width=width), exp)
1274: (4)             def test_large_neg_int64(self):
1275: (8)               assert_equal(np.binary_repr(np.int64(-2**62), width=64),
1276: (21)                 '11' + '0'*62)
1277: (0)             class TestBaseRepr:
1278: (4)               def test_base3(self):
1279: (8)                 assert_equal(np.base_repr(3**5, 3), '10000')
1280: (4)               def test_positive(self):
1281: (8)                 assert_equal(np.base_repr(12, 10), '12')
1282: (8)                 assert_equal(np.base_repr(12, 10, 4), '000012')
1283: (8)                 assert_equal(np.base_repr(12, 4), '30')
1284: (8)                 assert_equal(np.base_repr(3731624803700888, 36), '10QR0ROFCEW')
1285: (4)               def test_negative(self):
1286: (8)                 assert_equal(np.base_repr(-12, 10), '-12')
1287: (8)                 assert_equal(np.base_repr(-12, 10, 4), '-000012')
1288: (8)                 assert_equal(np.base_repr(-12, 4), '-30')
1289: (4)               def test_base_range(self):
1290: (8)                 with assert_raises(ValueError):
1291: (12)                   np.base_repr(1, 1)
1292: (8)                 with assert_raises(ValueError):
1293: (12)                   np.base_repr(1, 37)
1294: (0)             class TestArrayComparisons:
1295: (4)               def test_array_equal(self):
1296: (8)                 res = np.array_equal(np.array([1, 2]), np.array([1, 2]))
1297: (8)                 assert_(res)
1298: (8)                 assert_(type(res) is bool)
1299: (8)                 res = np.array_equal(np.array([1, 2]), np.array([1, 2, 3]))
1300: (8)                 assert_(not res)
1301: (8)                 assert_(type(res) is bool)
1302: (8)                 res = np.array_equal(np.array([1, 2]), np.array([3, 4]))
1303: (8)                 assert_(not res)
1304: (8)                 assert_(type(res) is bool)
1305: (8)                 res = np.array_equal(np.array([1, 2]), np.array([1, 3]))
1306: (8)                 assert_(not res)
1307: (8)                 assert_(type(res) is bool)
1308: (8)                 res = np.array_equal(np.array(['a']), np.array(['a']),
1309: (8)                   dtype='S1'))
1310: (8)               assert_(res)
1311: (8)               assert_(type(res) is bool)
1312: (29)               res = np.array_equal(np.array([('a', 1)], dtype='S1,u4'),
1313: (8)                             np.array([('a', 1)], dtype='S1,u4'))
1314: (8)               assert_(res)
1315: (4)               assert_(type(res) is bool)
1316: (8)             def test_array_equal_equal_nan(self):
1317: (8)               a1 = np.array([1, 2, np.nan])
1318: (8)               a2 = np.array([1, np.nan, 2])
1319: (8)               a3 = np.array([1, 2, np.inf])
1320: (8)               assert_(not np.array_equal(a1, a1))
1321: (8)               assert_(np.array_equal(a1, a1, equal_nan=True))
1322: (8)               assert_(not np.array_equal(a1, a2, equal_nan=True))
1323: (8)               assert_(not np.array_equal(a1, a3, equal_nan=True))
1324: (8)               a = np.array(np.nan)
1325: (8)               assert_(not np.array_equal(a, a))
1326: (8)               assert_(np.array_equal(a, a, equal_nan=True))
1327: (8)               a = np.array([1, 2, 3], dtype=int)
1328: (8)               assert_(np.array_equal(a, a))
1329: (8)               assert_(np.array_equal(a, a, equal_nan=True))
1330: (8)               a = np.array([[0, 1], [np.nan, 1]])
1331: (8)               assert_(not np.array_equal(a, a))
1332: (8)               assert_(np.array_equal(a, a, equal_nan=True))
1333: (8)               a, b = [np.array([1 + 1j])] * 2
1334: (8)               a.real, b.imag = np.nan, np.nan
1335: (8)               assert_(not np.array_equal(a, b, equal_nan=False))

```

```

1336: (4)
1337: (8)
1338: (8)
1339: (8)
1340: (8)
1341: (8)
1342: (8)
1343: (4)
1344: (8)
1345: (8)
1346: (8)
1347: (8)
1348: (8)
1349: (8)
1350: (8)
1351: (8)
1352: (8)
1353: (8)
1354: (8)
1355: (8)
1356: (8)
1357: (8)
1358: (8)
1359: (8)
1360: (8)
1361: (8)
1362: (8)
1363: (8)
1364: (8)
1365: (8)
1366: (8)
1367: (8)
1368: (8)
1369: (8)
1370: (8)
1371: (4)
1372: (4)
1373: (8)
1374: (8)
1375: (8)
1376: (8)
1377: (12)
1378: (8)
1379: (8)
1380: (8)
1381: (0)
1382: (4)
1383: (4)
1384: (12)
1385: (12)
1386: (8)
1387: (4)
1388: (8)
1389: (8)
1390: (8)
1391: (8)
1392: (8)
1393: (4)
1394: (0)
1395: (4)
1396: (8)
1397: (8)
1398: (4)
1399: (8)
1400: (4)
1401: (8)
1402: (8)
1403: (4)

        def test_none_comparisons_elementwise(self):
            a = np.array([None, 1, None], dtype=object)
            assert_equal(a == None, [True, False, True])
            assert_equal(a != None, [False, True, False])
            a = np.ones(3)
            assert_equal(a == None, [False, False, False])
            assert_equal(a != None, [True, True, True])

        def test_array_equiv(self):
            res = np.array_equiv(np.array([1, 2]), np.array([1, 2]))
            assert_(res)
            assert_(type(res) is bool)
            res = np.array_equiv(np.array([1, 2]), np.array([1, 2, 3]))
            assert_(not res)
            assert_(type(res) is bool)
            res = np.array_equiv(np.array([1, 2]), np.array([3, 4]))
            assert_(not res)
            assert_(type(res) is bool)
            res = np.array_equiv(np.array([1, 2]), np.array([1, 3]))
            assert_(not res)
            assert_(type(res) is bool)
            res = np.array_equiv(np.array([1, 1]), np.array([1]))
            assert_(res)
            assert_(type(res) is bool)
            res = np.array_equiv(np.array([1, 1]), np.array([[1], [1]]))
            assert_(res)
            assert_(type(res) is bool)
            res = np.array_equiv(np.array([1, 2]), np.array([2]))
            assert_(not res)
            assert_(type(res) is bool)
            res = np.array_equiv(np.array([1, 2]), np.array([[1], [2]]))
            assert_(not res)
            assert_(type(res) is bool)
            res = np.array_equiv(np.array([1, 2]), np.array([[1, 2, 3], [4, 5, 6],
                [7, 8, 9]]))
            assert_(not res)
            assert_(type(res) is bool)

    @pytest.mark.parametrize("dtype", ["V0", "V3", "V10"])
    def test_compare_unstructured_voids(self, dtype):
        zeros = np.zeros(3, dtype=dtype)
        assert_array_equal(zeros, zeros)
        assert not (zeros != zeros).any()
        if dtype == "V0":
            return
        nonzeros = np.array([b"1", b"2", b"3"], dtype=dtype)
        assert not (zeros == nonzeros).any()
        assert (zeros != nonzeros).all()

    def assert_array_strict_equal(x, y):
        assert_array_equal(x, y)
        if ((x.dtype.alignment <= 8 or
            np.intp().dtype.itemsize != 4) and
            sys.platform != 'win32'):
            assert_(x.flags == y.flags)
        else:
            assert_(x.flags.owndata == y.flags.owndata)
            assert_(x.flags.writeable == y.flags.writeable)
            assert_(x.flags.c_contiguous == y.flags.c_contiguous)
            assert_(x.flags.f_contiguous == y.flags.f_contiguous)
            assert_(x.flags.writebackifcopy == y.flags.writebackifcopy)
            assert_(x.dtype.isnative == y.dtype.isnative)

    class TestClip:
        def setup_method(self):
            self.nr = 5
            self.nc = 3

        def fastclip(self, a, m, M, out=None, **kwargs):
            return a.clip(m, M, out=out, **kwargs)

        def clip(self, a, m, M, out=None):
            selector = np.less(a, m) + 2*np.greater(a, M)
            return selector.choose((a, m, M), out=out)

        def _generate_data(self, n, m):

```

```

1404: (8)             return randn(n, m)
1405: (4)             def _generate_data_complex(self, n, m):
1406: (8)                 return randn(n, m) + 1.j * rand(n, m)
1407: (4)             def _generate_flt_data(self, n, m):
1408: (8)                 return (randn(n, m)).astype(np.float32)
1409: (4)             def _neg_byteorder(self, a):
1410: (8)                 a = np.asarray(a)
1411: (8)                 if sys.byteorder == 'little':
1412: (12)                     a = a.astype(a.dtype.newbyteorder('>'))
1413: (8)                 else:
1414: (12)                     a = a.astype(a.dtype.newbyteorder('<'))
1415: (8)             return a
1416: (4)             def _generate_non_native_data(self, n, m):
1417: (8)                 data = randn(n, m)
1418: (8)                 data = self._neg_byteorder(data)
1419: (8)                 assert_(not data.dtype.isnative)
1420: (8)             return data
1421: (4)             def _generate_int_data(self, n, m):
1422: (8)                 return (10 * rand(n, m)).astype(np.int64)
1423: (4)             def _generate_int32_data(self, n, m):
1424: (8)                 return (10 * rand(n, m)).astype(np.int32)
1425: (4) @pytest.mark.parametrize("dtype", '?bhilqpBHILQPefdgFDG0')
1426: (4)             def test_ones_pathological(self, dtype):
1427: (8)                 arr = np.ones(10, dtype=dtype)
1428: (8)                 expected = np.zeros(10, dtype=dtype)
1429: (8)                 actual = np.clip(arr, 1, 0)
1430: (8)                 if dtype == 'O':
1431: (12)                     assert actual.tolist() == expected.tolist()
1432: (8)                 else:
1433: (12)                     assert_equal(actual, expected)
1434: (4)             def test_simple_double(self):
1435: (8)                 a = self._generate_data(self.nr, self.nc)
1436: (8)                 m = 0.1
1437: (8)                 M = 0.6
1438: (8)                 ac = self.fastclip(a, m, M)
1439: (8)                 act = self.clip(a, m, M)
1440: (8)                 assert_array_strict_equal(ac, act)
1441: (4)             def test_simple_int(self):
1442: (8)                 a = self._generate_int_data(self.nr, self.nc)
1443: (8)                 a = a.astype(int)
1444: (8)                 m = -2
1445: (8)                 M = 4
1446: (8)                 ac = self.fastclip(a, m, M)
1447: (8)                 act = self.clip(a, m, M)
1448: (8)                 assert_array_strict_equal(ac, act)
1449: (4)             def test_array_double(self):
1450: (8)                 a = self._generate_data(self.nr, self.nc)
1451: (8)                 m = np.zeros(a.shape)
1452: (8)                 M = m + 0.5
1453: (8)                 ac = self.fastclip(a, m, M)
1454: (8)                 act = self.clip(a, m, M)
1455: (8)                 assert_array_strict_equal(ac, act)
1456: (4)             def test_simple_nonnaive(self):
1457: (8)                 a = self._generate_non_native_data(self.nr, self.nc)
1458: (8)                 m = -0.5
1459: (8)                 M = 0.6
1460: (8)                 ac = self.fastclip(a, m, M)
1461: (8)                 act = self.clip(a, m, M)
1462: (8)                 assert_array_equal(ac, act)
1463: (8)                 a = self._generate_data(self.nr, self.nc)
1464: (8)                 m = -0.5
1465: (8)                 M = self._neg_byteorder(0.6)
1466: (8)                 assert_(not M.dtype.isnative)
1467: (8)                 ac = self.fastclip(a, m, M)
1468: (8)                 act = self.clip(a, m, M)
1469: (8)                 assert_array_equal(ac, act)
1470: (4)             def test_simple_complex(self):
1471: (8)                 a = 3 * self._generate_data_complex(self.nr, self.nc)
1472: (8)                 m = -0.5

```

```

1473: (8)             M = 1.
1474: (8)             ac = self.fastclip(a, m, M)
1475: (8)             act = self.clip(a, m, M)
1476: (8)             assert_array_strict_equal(ac, act)
1477: (8)             a = 3 * self._generate_data(self.nr, self.nc)
1478: (8)             m = -0.5 + 1.j
1479: (8)             M = 1. + 2.j
1480: (8)             ac = self.fastclip(a, m, M)
1481: (8)             act = self.clip(a, m, M)
1482: (8)             assert_array_strict_equal(ac, act)
1483: (4)             def test_clip_complex(self):
1484: (8)                 a = np.ones(10, dtype=complex)
1485: (8)                 m = a.min()
1486: (8)                 M = a.max()
1487: (8)                 am = self.fastclip(a, m, None)
1488: (8)                 aM = self.fastclip(a, None, M)
1489: (8)                 assert_array_strict_equal(am, a)
1490: (8)                 assert_array_strict_equal(aM, a)
1491: (4)             def test_clip_non_contig(self):
1492: (8)                 a = self._generate_data(self.nr * 2, self.nc * 3)
1493: (8)                 a = a[::-2, ::3]
1494: (8)                 assert_(not a.flags['F_CONTIGUOUS'])
1495: (8)                 assert_(not a.flags['C_CONTIGUOUS'])
1496: (8)                 ac = self.fastclip(a, -1.6, 1.7)
1497: (8)                 act = self.clip(a, -1.6, 1.7)
1498: (8)                 assert_array_strict_equal(ac, act)
1499: (4)             def test_simple_out(self):
1500: (8)                 a = self._generate_data(self.nr, self.nc)
1501: (8)                 m = -0.5
1502: (8)                 M = 0.6
1503: (8)                 ac = np.zeros(a.shape)
1504: (8)                 act = np.zeros(a.shape)
1505: (8)                 self.fastclip(a, m, M, ac)
1506: (8)                 self.clip(a, m, M, act)
1507: (8)                 assert_array_strict_equal(ac, act)
1508: (4)             @pytest.mark.parametrize("casting", [None, "unsafe"])
1509: (4)             def test_simple_int32_inout(self, casting):
1510: (8)                 a = self._generate_int32_data(self.nr, self.nc)
1511: (8)                 m = np.float64(0)
1512: (8)                 M = np.float64(2)
1513: (8)                 ac = np.zeros(a.shape, dtype=np.int32)
1514: (8)                 act = ac.copy()
1515: (8)                 if casting is None:
1516: (12)                     with pytest.raises(TypeError):
1517: (16)                         self.fastclip(a, m, M, ac, casting=casting)
1518: (8)                 else:
1519: (12)                     self.fastclip(a, m, M, ac, casting=casting)
1520: (12)                     self.clip(a, m, M, act)
1521: (12)                     assert_array_strict_equal(ac, act)
1522: (4)             def test_simple_int64_out(self):
1523: (8)                 a = self._generate_int32_data(self.nr, self.nc)
1524: (8)                 m = np.int32(-1)
1525: (8)                 M = np.int32(1)
1526: (8)                 ac = np.zeros(a.shape, dtype=np.int64)
1527: (8)                 act = ac.copy()
1528: (8)                 self.fastclip(a, m, M, ac)
1529: (8)                 self.clip(a, m, M, act)
1530: (8)                 assert_array_strict_equal(ac, act)
1531: (4)             def test_simple_int64_inout(self):
1532: (8)                 a = self._generate_int32_data(self.nr, self.nc)
1533: (8)                 m = np.zeros(a.shape, np.float64)
1534: (8)                 M = np.float64(1)
1535: (8)                 ac = np.zeros(a.shape, dtype=np.int32)
1536: (8)                 act = ac.copy()
1537: (8)                 self.fastclip(a, m, M, out=ac, casting="unsafe")
1538: (8)                 self.clip(a, m, M, act)
1539: (8)                 assert_array_strict_equal(ac, act)
1540: (4)             def test_simple_int32_out(self):
1541: (8)                 a = self._generate_data(self.nr, self.nc)

```

```
1542: (8)          m = -1.0
1543: (8)          M = 2.0
1544: (8)          ac = np.zeros(a.shape, dtype=np.int32)
1545: (8)          act = ac.copy()
1546: (8)          self.fastclip(a, m, M, out=ac, casting="unsafe")
1547: (8)          self.clip(a, m, M, act)
1548: (8)          assert_array_strict_equal(ac, act)
1549: (4)          def test_simple_inplace_01(self):
1550: (8)              a = self._generate_data(self.nr, self.nc)
1551: (8)              ac = a.copy()
1552: (8)              m = np.zeros(a.shape)
1553: (8)              M = 1.0
1554: (8)              self.fastclip(a, m, M, a)
1555: (8)              self.clip(a, m, M, ac)
1556: (8)              assert_array_strict_equal(a, ac)
1557: (4)          def test_simple_inplace_02(self):
1558: (8)              a = self._generate_data(self.nr, self.nc)
1559: (8)              ac = a.copy()
1560: (8)              m = -0.5
1561: (8)              M = 0.6
1562: (8)              self.fastclip(a, m, M, a)
1563: (8)              self.clip(ac, m, M, ac)
1564: (8)              assert_array_strict_equal(a, ac)
1565: (4)          def test_noncontig_inplace(self):
1566: (8)              a = self._generate_data(self.nr * 2, self.nc * 3)
1567: (8)              a = a[::-2, ::3]
1568: (8)              assert_(not a.flags['F_CONTIGUOUS'])
1569: (8)              assert_(not a.flags['C_CONTIGUOUS'])
1570: (8)              ac = a.copy()
1571: (8)              m = -0.5
1572: (8)              M = 0.6
1573: (8)              self.fastclip(a, m, M, a)
1574: (8)              self.clip(ac, m, M, ac)
1575: (8)              assert_array_equal(a, ac)
1576: (4)          def test_type_cast_01(self):
1577: (8)              a = self._generate_data(self.nr, self.nc)
1578: (8)              m = -0.5
1579: (8)              M = 0.6
1580: (8)              ac = self.fastclip(a, m, M)
1581: (8)              act = self.clip(a, m, M)
1582: (8)              assert_array_strict_equal(ac, act)
1583: (4)          def test_type_cast_02(self):
1584: (8)              a = self._generate_int_data(self.nr, self.nc)
1585: (8)              a = a.astype(np.int32)
1586: (8)              m = -2
1587: (8)              M = 4
1588: (8)              ac = self.fastclip(a, m, M)
1589: (8)              act = self.clip(a, m, M)
1590: (8)              assert_array_strict_equal(ac, act)
1591: (4)          def test_type_cast_03(self):
1592: (8)              a = self._generate_int32_data(self.nr, self.nc)
1593: (8)              m = -2
1594: (8)              M = 4
1595: (8)              ac = self.fastclip(a, np.float64(m), np.float64(M))
1596: (8)              act = self.clip(a, np.float64(m), np.float64(M))
1597: (8)              assert_array_strict_equal(ac, act)
1598: (4)          def test_type_cast_04(self):
1599: (8)              a = self._generate_int32_data(self.nr, self.nc)
1600: (8)              m = np.float32(-2)
1601: (8)              M = np.float32(4)
1602: (8)              act = self.fastclip(a, m, M)
1603: (8)              ac = self.clip(a, m, M)
1604: (8)              assert_array_strict_equal(ac, act)
1605: (4)          def test_type_cast_05(self):
1606: (8)              a = self._generate_int_data(self.nr, self.nc)
1607: (8)              m = -0.5
1608: (8)              M = 1.
1609: (8)              ac = self.fastclip(a, m * np.zeros(a.shape), M)
1610: (8)              act = self.clip(a, m * np.zeros(a.shape), M)
```

```

1611: (8)             assert_array.strict_equal(ac, act)
1612: (4)             def test_type_cast_06(self):
1613: (8)                 a = self._generate_data(self.nr, self.nc)
1614: (8)                 m = 0.5
1615: (8)                 m_s = self._neg_byteorder(m)
1616: (8)                 M = 1.
1617: (8)                 act = self.clip(a, m_s, M)
1618: (8)                 ac = self.fastclip(a, m_s, M)
1619: (8)                 assert_array.strict_equal(ac, act)
1620: (4)             def test_type_cast_07(self):
1621: (8)                 a = self._generate_data(self.nr, self.nc)
1622: (8)                 m = -0.5 * np.ones(a.shape)
1623: (8)                 M = 1.
1624: (8)                 a_s = self._neg_byteorder(a)
1625: (8)                 assert_(not a_s.dtype.isnative)
1626: (8)                 act = a_s.clip(m, M)
1627: (8)                 ac = self.fastclip(a_s, m, M)
1628: (8)                 assert_array.strict_equal(ac, act)
1629: (4)             def test_type_cast_08(self):
1630: (8)                 a = self._generate_data(self.nr, self.nc)
1631: (8)                 m = -0.5
1632: (8)                 M = 1.
1633: (8)                 a_s = self._neg_byteorder(a)
1634: (8)                 assert_(not a_s.dtype.isnative)
1635: (8)                 ac = self.fastclip(a_s, m, M)
1636: (8)                 act = a_s.clip(m, M)
1637: (8)                 assert_array.strict_equal(ac, act)
1638: (4)             def test_type_cast_09(self):
1639: (8)                 a = self._generate_data(self.nr, self.nc)
1640: (8)                 m = -0.5 * np.ones(a.shape)
1641: (8)                 M = 1.
1642: (8)                 m_s = self._neg_byteorder(m)
1643: (8)                 assert_(not m_s.dtype.isnative)
1644: (8)                 ac = self.fastclip(a, m_s, M)
1645: (8)                 act = self.clip(a, m_s, M)
1646: (8)                 assert_array.strict_equal(ac, act)
1647: (4)             def test_type_cast_10(self):
1648: (8)                 a = self._generate_int_data(self.nr, self.nc)
1649: (8)                 b = np.zeros(a.shape, dtype=np.float32)
1650: (8)                 m = np.float32(-0.5)
1651: (8)                 M = np.float32(1)
1652: (8)                 act = self.clip(a, m, M, out=b)
1653: (8)                 ac = self.fastclip(a, m, M, out=b)
1654: (8)                 assert_array.strict_equal(ac, act)
1655: (4)             def test_type_cast_11(self):
1656: (8)                 a = self._generate_non_native_data(self.nr, self.nc)
1657: (8)                 b = a.copy()
1658: (8)                 b = b.astype(b.dtype.newbyteorder('>'))
1659: (8)                 bt = b.copy()
1660: (8)                 m = -0.5
1661: (8)                 M = 1.
1662: (8)                 self.fastclip(a, m, M, out=b)
1663: (8)                 self.clip(a, m, M, out=bt)
1664: (8)                 assert_array.strict_equal(b, bt)
1665: (4)             def test_type_cast_12(self):
1666: (8)                 a = self._generate_int_data(self.nr, self.nc)
1667: (8)                 b = np.zeros(a.shape, dtype=np.float32)
1668: (8)                 m = np.int32(0)
1669: (8)                 M = np.int32(1)
1670: (8)                 act = self.clip(a, m, M, out=b)
1671: (8)                 ac = self.fastclip(a, m, M, out=b)
1672: (8)                 assert_array.strict_equal(ac, act)
1673: (4)             def test_clip_with_out_simple(self):
1674: (8)                 a = self._generate_data(self.nr, self.nc)
1675: (8)                 m = -0.5
1676: (8)                 M = 0.6
1677: (8)                 ac = np.zeros(a.shape)
1678: (8)                 act = np.zeros(a.shape)
1679: (8)                 self.fastclip(a, m, M, ac)

```

```

1680: (8)             self.clip(a, m, M, act)
1681: (8)             assert_array_strict_equal(ac, act)
1682: (4)             def test_clip_with_out_simple2(self):
1683: (8)                 a = self._generate_int32_data(self.nr, self.nc)
1684: (8)                 m = np.float64(0)
1685: (8)                 M = np.float64(2)
1686: (8)                 ac = np.zeros(a.shape, dtype=np.int32)
1687: (8)                 act = ac.copy()
1688: (8)                 self.fastclip(a, m, M, out=ac, casting="unsafe")
1689: (8)                 self.clip(a, m, M, act)
1690: (8)                 assert_array_strict_equal(ac, act)
1691: (4)             def test_clip_with_out_simple_int32(self):
1692: (8)                 a = self._generate_int32_data(self.nr, self.nc)
1693: (8)                 m = np.int32(-1)
1694: (8)                 M = np.int32(1)
1695: (8)                 ac = np.zeros(a.shape, dtype=np.int64)
1696: (8)                 act = ac.copy()
1697: (8)                 self.fastclip(a, m, M, ac)
1698: (8)                 self.clip(a, m, M, act)
1699: (8)                 assert_array_strict_equal(ac, act)
1700: (4)             def test_clip_with_out_array_int32(self):
1701: (8)                 a = self._generate_int32_data(self.nr, self.nc)
1702: (8)                 m = np.zeros(a.shape, np.float64)
1703: (8)                 M = np.float64(1)
1704: (8)                 ac = np.zeros(a.shape, dtype=np.int32)
1705: (8)                 act = ac.copy()
1706: (8)                 self.fastclip(a, m, M, out=ac, casting="unsafe")
1707: (8)                 self.clip(a, m, M, act)
1708: (8)                 assert_array_strict_equal(ac, act)
1709: (4)             def test_clip_with_out_array_outint32(self):
1710: (8)                 a = self._generate_data(self.nr, self.nc)
1711: (8)                 m = -1.0
1712: (8)                 M = 2.0
1713: (8)                 ac = np.zeros(a.shape, dtype=np.int32)
1714: (8)                 act = ac.copy()
1715: (8)                 self.fastclip(a, m, M, out=ac, casting="unsafe")
1716: (8)                 self.clip(a, m, M, act)
1717: (8)                 assert_array_strict_equal(ac, act)
1718: (4)             def test_clip_with_out_transposed(self):
1719: (8)                 a = np.arange(16).reshape(4, 4)
1720: (8)                 out = np.empty_like(a).T
1721: (8)                 a.clip(4, 10, out=out)
1722: (8)                 expected = self.clip(a, 4, 10)
1723: (8)                 assert_array_equal(out, expected)
1724: (4)             def test_clip_with_out_memory_overlap(self):
1725: (8)                 a = np.arange(16).reshape(4, 4)
1726: (8)                 ac = a.copy()
1727: (8)                 a[:-1].clip(4, 10, out=a[1:])
1728: (8)                 expected = self.clip(ac[:-1], 4, 10)
1729: (8)                 assert_array_equal(a[1:], expected)
1730: (4)             def test_clip_inplace_array(self):
1731: (8)                 a = self._generate_data(self.nr, self.nc)
1732: (8)                 ac = a.copy()
1733: (8)                 m = np.zeros(a.shape)
1734: (8)                 M = 1.0
1735: (8)                 self.fastclip(a, m, M, a)
1736: (8)                 self.clip(a, m, M, ac)
1737: (8)                 assert_array_strict_equal(a, ac)
1738: (4)             def test_clip_inplace_simple(self):
1739: (8)                 a = self._generate_data(self.nr, self.nc)
1740: (8)                 ac = a.copy()
1741: (8)                 m = -0.5
1742: (8)                 M = 0.6
1743: (8)                 self.fastclip(a, m, M, a)
1744: (8)                 self.clip(a, m, M, ac)
1745: (8)                 assert_array_strict_equal(a, ac)
1746: (4)             def test_clip_func_takes_out(self):
1747: (8)                 a = self._generate_data(self.nr, self.nc)
1748: (8)                 ac = a.copy()

```

```

1749: (8)             m = -0.5
1750: (8)             M = 0.6
1751: (8)             a2 = np.clip(a, m, M, out=a)
1752: (8)             self.clip(a, m, M, ac)
1753: (8)             assert_array_almost_equal(a2, ac)
1754: (8)             assert_(a2 is a)
1755: (4)             def test_clip_nan(self):
1756: (8)                 d = np.arange(7.)
1757: (8)                 assert_equal(d.clip(min=np.nan), np.nan)
1758: (8)                 assert_equal(d.clip(max=np.nan), np.nan)
1759: (8)                 assert_equal(d.clip(min=np.nan, max=np.nan), np.nan)
1760: (8)                 assert_equal(d.clip(min=-2, max=np.nan), np.nan)
1761: (8)                 assert_equal(d.clip(min=np.nan, max=10), np.nan)
1762: (4)             def test_object_clip(self):
1763: (8)                 a = np.arange(10, dtype=object)
1764: (8)                 actual = np.clip(a, 1, 5)
1765: (8)                 expected = np.array([1, 1, 2, 3, 4, 5, 5, 5, 5, 5])
1766: (8)                 assert actual.tolist() == expected.tolist()
1767: (4)             def test_clip_all_none(self):
1768: (8)                 a = np.arange(10, dtype=object)
1769: (8)                 with assert_raises_regex(ValueError, 'max or min'):
1770: (12)                     np.clip(a, None, None)
1771: (4)             def test_clip_invalid_casting(self):
1772: (8)                 a = np.arange(10, dtype=object)
1773: (8)                 with assert_raises_regex(ValueError,
1774: (33)                     'casting must be one of'):
1775: (12)                     self.fastclip(a, 1, 8, casting="garbage")
1776: (4)             @pytest.mark.parametrize("amin, amax", [
1777: (8)                 (1, 0),
1778: (8)                 (1, np.zeros(10)),
1779: (8)                 (np.ones(10), np.zeros(10)),
1780: (8)                 ])
1781: (4)             def test_clip_value_min_max_flip(self, amin, amax):
1782: (8)                 a = np.arange(10, dtype=np.int64)
1783: (8)                 expected = np.minimum(np.maximum(a, amin), amax)
1784: (8)                 actual = np.clip(a, amin, amax)
1785: (8)                 assert_equal(actual, expected)
1786: (4)             @pytest.mark.parametrize("arr, amin, amax, exp", [
1787: (8)                 (np.zeros(10, dtype=np.int64),
1788: (9)                     0,
1789: (9)                     -2**64+1,
1790: (9)                     np.full(10, -2**64+1, dtype=object)),
1791: (8)                     (np.zeros(10, dtype='m8') - 1,
1792: (9)                     0,
1793: (9)                     0,
1794: (9)                     np.zeros(10, dtype='m8')),
1795: (4)                 ])
1796: (4)             def test_clip_problem_cases(self, arr, amin, amax, exp):
1797: (8)                 actual = np.clip(arr, amin, amax)
1798: (8)                 assert_equal(actual, exp)
1799: (4)             @pytest.mark.parametrize("arr, amin, amax", [
1800: (8)                 (np.zeros(10, dtype=np.int64),
1801: (9)                     np.array(np.nan),
1802: (9)                     np.zeros(10, dtype=np.int32)),
1803: (4)                 ])
1804: (4)             def test_clip_scalar_nan_propagation(self, arr, amin, amax):
1805: (8)                 expected = np.minimum(np.maximum(arr, amin), amax)
1806: (8)                 actual = np.clip(arr, amin, amax)
1807: (8)                 assert_equal(actual, expected)
1808: (4)             @pytest.mark.xfail(reason="propagation doesn't match spec")
1809: (4)             @pytest.mark.parametrize("arr, amin, amax", [
1810: (8)                 (np.array([1] * 10, dtype='m8'),
1811: (9)                     np.timedelta64('NaT'),
1812: (9)                     np.zeros(10, dtype=np.int32)),
1813: (4)                 ])
1814: (4)             @pytest.mark.filterwarnings("ignore::DeprecationWarning")
1815: (4)             def test_NaT_propagation(self, arr, amin, amax):
1816: (8)                 expected = np.minimum(np.maximum(arr, amin), amax)
1817: (8)                 actual = np.clip(arr, amin, amax)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1818: (8)             assert_equal(actual, expected)
1819: (4)             @given(
1820: (8)                 data=st.data(),
1821: (8)                 arr=hypn.arrays(
1822: (12)                   dtype=hypn.integer_dtypes() | hypn.floating_dtypes(),
1823: (12)                   shape=hypn.array_shapes()
1824: (8)             )
1825: (4)         )
1826: (4)     def test_clip_property(self, data, arr):
1827: (8)         """A property-based test using Hypothesis.
1828: (8)         This aims for maximum generality: it could in principle generate *any*
1829: (8)         valid inputs to np.clip, and in practice generates much more varied
1830: (8)         inputs than human testers come up with.
1831: (8)         Because many of the inputs have tricky dependencies - compatible
1832: (8)         dtypes
1833: (8)         and mutually-broadcastable shapes - we use `st.data()` strategy draw
1834: (8)         values *inside* the test function, from strategies we construct based
1835: (8)         on previous values. An alternative would be to define a custom
1836: (8)         strategy
1837: (8)         with `@st.composite`, but until we have duplicated code inline is
1838: (8)         fine.
1839: (8)         That accounts for most of the function; the actual test is just three
1840: (8)         lines to calculate and compare actual vs expected results!
1841: (8)         """
1842: (12)     numeric_dtypes = hypn.integer_dtypes() | hypn.floating_dtypes()
1843: (16)     in_shapes, result_shape = data.draw(
1844: (12)         hypn.mutually_broadcastable_shapes(
1845: (16)             num_shapes=2, base_shape=arr.shape
1846: (8)         )
1847: (8)         s = numeric_dtypes.flatmap(
1848: (12)             lambda x: hypn.from_dtype(x, allow_nan=False))
1849: (8)             amin = data.draw(s | hypn.arrays(dtype=numeric_dtypes,
1850: (12)                 shape=in_shapes[0], elements={"allow_nan": False}))
1851: (8)             amax = data.draw(s | hypn.arrays(dtype=numeric_dtypes,
1852: (12)                 shape=in_shapes[1], elements={"allow_nan": False}))
1853: (8)             result = np.clip(arr, amin, amax)
1854: (8)             t = np.result_type(arr, amin, amax)
1855: (8)             expected = np.minimum(amax, np.maximum(arr, amin, dtype=t), dtype=t)
1856: (0)             assert result.dtype == t
1857: (8)             assert_array_equal(result, expected)
1858: (0)         class TestAllclose:
1859: (4)             rtol = 1e-5
1860: (4)             atol = 1e-8
1861: (4)             def setup_method(self):
1862: (8)                 self.olderr = np.seterr(invalid='ignore')
1863: (4)             def teardown_method(self):
1864: (8)                 np.seterr(**self.olderr)
1865: (4)             def tst_allclose(self, x, y):
1866: (8)                 assert_(np.allclose(x, y), "%s and %s not close" % (x, y))
1867: (4)             def tst_not_allclose(self, x, y):
1868: (8)                 assert_(not np.allclose(x, y), "%s and %s shouldn't be close" % (x,
1869: (8)                     y))
1870: (4)             def test_ip_allclose(self):
1871: (8)                 arr = np.array([100, 1000])
1872: (8)                 aran = np.arange(125).reshape((5, 5, 5))
1873: (16)                 atol = self.atol
1874: (16)                 rtol = self.rtol
1875: (16)                 data = [([1, 0], [1, 0]),
1876: (16)                     ([atol], [0]),
1877: (16)                     ([1], [1+rtol+atol]),
1878: (16)                     (arr, arr + arr*rtol),
1879: (16)                     (arr, arr + arr*rtol + atol*2),
1880: (8)                     (aran, aran + aran*rtol),
1881: (12)                     (np.inf, np.inf),
1882: (4)                     (np.inf, [np.inf])]
1883: (8)                 for (x, y) in data:
1884: (8)                     self.tst_allclose(x, y)
1885: (4)             def test_ip_not_allclose(self):

```

```

1883: (8)         aran = np.arange(125).reshape((5, 5, 5))
1884: (8)         atol = self.atol
1885: (8)         rtol = self.rtol
1886: (8)         data = [([np.inf, 0], [1, np.inf]),
1887: (16)           ([np.inf, 0], [1, 0]),
1888: (16)           ([np.inf, np.inf], [1, np.inf]),
1889: (16)           ([np.inf, np.inf], [1, 0]),
1890: (16)           ([-np.inf, 0], [np.inf, 0]),
1891: (16)           ([np.nan, 0], [np.nan, 0]),
1892: (16)           ([[atol*2], [0]]),
1893: (16)           ([1], [1+rtol+atol*2]),
1894: (16)           (aran, aran + aran*atol + atol*2),
1895: (16)           (np.array([np.inf, 1]), np.array([0, np.inf])))
1896: (8)         for (x, y) in data:
1897: (12)           self.tst_not_allclose(x, y)
1898: (4)         def test_no_parameter_modification(self):
1899: (8)             x = np.array([np.inf, 1])
1900: (8)             y = np.array([0, np.inf])
1901: (8)             np.allclose(x, y)
1902: (8)             assert_array_equal(x, np.array([np.inf, 1]))
1903: (8)             assert_array_equal(y, np.array([0, np.inf]))
1904: (4)         def test_min_int(self):
1905: (8)             min_int = np.iinfo(np.int_).min
1906: (8)             a = np.array([min_int], dtype=np.int_)
1907: (8)             assert_(np.allclose(a, a))
1908: (4)         def test_equalnan(self):
1909: (8)             x = np.array([1.0, np.nan])
1910: (8)             assert_(np.allclose(x, x, equal_nan=True))
1911: (4)         def test_return_class_is_ndarray(self):
1912: (8)             class Foo(np.ndarray):
1913: (12)                 def __new__(cls, *args, **kwargs):
1914: (16)                     return np.array(*args, **kwargs).view(cls)
1915: (8)             a = Foo([1])
1916: (8)             assert_(type(np.allclose(a, a)) is bool)
1917: (0)         class TestIsclose:
1918: (4)             rtol = 1e-5
1919: (4)             atol = 1e-8
1920: (4)             def _setup(self):
1921: (8)                 atol = self.atol
1922: (8)                 rtol = self.rtol
1923: (8)                 arr = np.array([100, 1000])
1924: (8)                 aran = np.arange(125).reshape((5, 5, 5))
1925: (8)                 self.all_close_tests = [
1926: (16)                   ([1, 0], [1, 0]),
1927: (16)                   ([[atol]], [0]),
1928: (16)                   ([1], [1 + rtol + atol]),
1929: (16)                   (arr, arr + arr*rtol),
1930: (16)                   (arr, arr + arr*rtol + atol),
1931: (16)                   (aran, aran + aran*rtol),
1932: (16)                   (np.inf, np.inf),
1933: (16)                   (np.inf, [np.inf]),
1934: (16)                   ([np.inf, -np.inf], [np.inf, -np.inf]),
1935: (16)                   []
1936: (8)                 self.none_close_tests = [
1937: (16)                   ([np.inf, 0], [1, np.inf]),
1938: (16)                   ([np.inf, -np.inf], [1, 0]),
1939: (16)                   ([np.inf, np.inf], [1, -np.inf]),
1940: (16)                   ([np.inf, np.inf], [1, 0]),
1941: (16)                   ([np.nan, 0], [np.nan, -np.inf]),
1942: (16)                   ([[atol*2], [0]]),
1943: (16)                   ([1], [1 + rtol + atol*2]),
1944: (16)                   (aran, aran + rtol*1.1*aran + atol*1.1),
1945: (16)                   (np.array([np.inf, 1]), np.array([0, np.inf])),
1946: (16)                   []
1947: (8)                 self.some_close_tests = [
1948: (16)                   ([np.inf, 0], [np.inf, atol*2]),
1949: (16)                   ([[atol, 1, 1e6*(1 + 2*rtol) + atol], [0, np.nan, 1e6]]),
1950: (16)                   (np.arange(3), [0, 1, 2.1]),
1951: (16)                   ([np.nan, [np.nan, np.nan, np.nan]]),

```

```

1952: (16) ([0], [atol, np.inf, -np.inf, np.nan]),
1953: (16) (0, [atol, np.inf, -np.inf, np.nan]),
1954: (16) ]
1955: (8) self.some_close_results = [
1956: (16) [True, False],
1957: (16) [True, False, False],
1958: (16) [True, True, False],
1959: (16) [False, False, False],
1960: (16) [True, False, False, False],
1961: (16) [True, False, False, False],
1962: (16) ]
1963: (4) def test_ip_isclose(self):
1964: (8)     self._setup()
1965: (8)     tests = self.some_close_tests
1966: (8)     results = self.some_close_results
1967: (8)     for (x, y), result in zip(tests, results):
1968: (12)         assert_array_equal(np.isclose(x, y), result)
1969: (4) def tst_all_isclose(self, x, y):
1970: (8)     assert_(np.all(np.isclose(x, y)), "%s and %s not close" % (x, y))
1971: (4) def tst_none_isclose(self, x, y):
1972: (8)     msg = "%s and %s shouldn't be close"
1973: (8)     assert_(not np.any(np.isclose(x, y)), msg % (x, y))
1974: (4) def tst_isclose_allclose(self, x, y):
1975: (8)     msg = "isclose.all() and allclose aren't same for %s and %s"
1976: (8)     msg2 = "isclose and allclose aren't same for %s and %s"
1977: (8)     if np.isscalar(x) and np.isscalar(y):
1978: (12)         assert_(np.isclose(x, y) == np.allclose(x, y), msg=msg2 % (x, y))
1979: (8)     else:
1980: (12)         assert_array_equal(np.isclose(x, y).all(), np.allclose(x, y), msg
% (x, y))
1981: (4) def test_ip_all_isclose(self):
1982: (8)     self._setup()
1983: (8)     for (x, y) in self.all_close_tests:
1984: (12)         self.tst_all_isclose(x, y)
1985: (4) def test_ip_none_isclose(self):
1986: (8)     self._setup()
1987: (8)     for (x, y) in self.none_close_tests:
1988: (12)         self.tst_none_isclose(x, y)
1989: (4) def test_ip_isclose_allclose(self):
1990: (8)     self._setup()
1991: (8)     tests = (self.all_close_tests + self.none_close_tests +
1992: (17)             self.some_close_tests)
1993: (8)     for (x, y) in tests:
1994: (12)         self.tst_isclose_allclose(x, y)
1995: (4) def test_equal_nan(self):
1996: (8)     assert_array_equal(np.isclose(np.nan, np.nan, equal_nan=True), [True])
1997: (8)     arr = np.array([1.0, np.nan])
1998: (8)     assert_array_equal(np.isclose(arr, arr, equal_nan=True), [True, True])
1999: (4) def test_masked_arrays(self):
2000: (8)     x = np.ma.masked_where([True, True, False], np.arange(3))
2001: (8)     assert_(type(x) is type(np.isclose(2, x)))
2002: (8)     assert_(type(x) is type(np.isclose(x, 2)))
2003: (8)     x = np.ma.masked_where([True, True, False], [np.nan, np.inf, np.nan])
2004: (8)     assert_(type(x) is type(np.isclose(np.inf, x)))
2005: (8)     assert_(type(x) is type(np.isclose(x, np.inf)))
2006: (8)     x = np.ma.masked_where([True, True, False], [np.nan, np.nan, np.nan])
2007: (8)     y = np.isclose(np.nan, x, equal_nan=True)
2008: (8)     assert_(type(x) is type(y))
2009: (8)     assert_array_equal([True, True, False], y.mask)
2010: (8)     y = np.isclose(x, np.nan, equal_nan=True)
2011: (8)     assert_(type(x) is type(y))
2012: (8)     assert_array_equal([True, True, False], y.mask)
2013: (8)     x = np.ma.masked_where([True, True, False], [np.nan, np.nan, np.nan])
2014: (8)     y = np.isclose(x, x, equal_nan=True)
2015: (8)     assert_(type(x) is type(y))
2016: (8)     assert_array_equal([True, True, False], y.mask)
2017: (4) def test_scalar_return(self):
2018: (8)     assert_(np.isscalar(np.isclose(1, 1)))
2019: (4) def test_no_parameter_modification(self):

```

```

2020: (8)          x = np.array([np.inf, 1])
2021: (8)          y = np.array([0, np.inf])
2022: (8)          np.isclose(x, y)
2023: (8)          assert_array_equal(x, np.array([np.inf, 1]))
2024: (8)          assert_array_equal(y, np.array([0, np.inf]))
2025: (4)          def test_non_finite_scalar(self):
2026: (8)              assert_(np.isclose(np.inf, -np.inf) is np.False_)
2027: (8)              assert_(np.isclose(0, np.inf) is np.False_)
2028: (8)              assert_(type(np.isclose(0, np.inf)) is np.bool_)
2029: (4)          def test_timedelta(self):
2030: (8)              a = np.array([[1, 2, 3, "NaT"]], dtype="m8[ns]")
2031: (8)              assert np.isclose(a, a, atol=0, equal_nan=True).all()
2032: (8)              assert np.isclose(a, a, atol=np.timedelta64(1, "ns"),
equal_nan=True).all()
2033: (8)                  assert np.allclose(a, a, atol=0, equal_nan=True)
2034: (8)                  assert np.allclose(a, a, atol=np.timedelta64(1, "ns"), equal_nan=True)
2035: (0)          class TestStdVar:
2036: (4)              def setup_method(self):
2037: (8)                  self.A = np.array([1, -1, 1, -1])
2038: (8)                  self.real_var = 1
2039: (4)              def test_basic(self):
2040: (8)                  assert_almost_equal(np.var(self.A), self.real_var)
2041: (8)                  assert_almost_equal(np.std(self.A)**2, self.real_var)
2042: (4)              def test_scalars(self):
2043: (8)                  assert_equal(np.var(1), 0)
2044: (8)                  assert_equal(np.std(1), 0)
2045: (4)              def test_ddof1(self):
2046: (8)                  assert_almost_equal(np.var(self.A, ddof=1),
self.real_var * len(self.A) / (len(self.A) - 1))
2047: (28)                  assert_almost_equal(np.std(self.A, ddof=1)**2,
self.real_var*len(self.A) / (len(self.A) - 1))
2048: (8)
2049: (28)
2050: (4)              def test_ddof2(self):
2051: (8)                  assert_almost_equal(np.var(self.A, ddof=2),
self.real_var * len(self.A) / (len(self.A) - 2))
2052: (28)                  assert_almost_equal(np.std(self.A, ddof=2)**2,
self.real_var * len(self.A) / (len(self.A) - 2))
2053: (8)
2054: (28)
2055: (4)              def test_out_scalar(self):
2056: (8)                  d = np.arange(10)
2057: (8)                  out = np.array(0.)
2058: (8)                  r = np.std(d, out=out)
2059: (8)                  assert_(r is out)
2060: (8)                  assert_array_equal(r, out)
2061: (8)                  r = np.var(d, out=out)
2062: (8)                  assert_(r is out)
2063: (8)                  assert_array_equal(r, out)
2064: (8)                  r = np.mean(d, out=out)
2065: (8)                  assert_(r is out)
2066: (8)                  assert_array_equal(r, out)
2067: (0)          class TestStdVarComplex:
2068: (4)              def test_basic(self):
2069: (8)                  A = np.array([1, 1.j, -1, -1.j])
2070: (8)                  real_var = 1
2071: (8)                  assert_almost_equal(np.var(A), real_var)
2072: (8)                  assert_almost_equal(np.std(A)**2, real_var)
2073: (4)              def test_scalars(self):
2074: (8)                  assert_equal(np.var(1j), 0)
2075: (8)                  assert_equal(np.std(1j), 0)
2076: (0)          class TestCreationFuncs:
2077: (4)              def setup_method(self):
2078: (8)                  dtypes = {np.dtype(tp) for tp in
itertools.chain(*np.sctypes.values())}
2079: (8)                  variable_sized = {tp for tp in dtypes if tp.str.endswith('0')}
2080: (8)                  self.dtypes = sorted(dtypes - variable_sized |
{np.dtype(tp.str.replace("0", str(i))) |
for tp in variable_sized for i in range(1, 10)},
key=lambda dtype: dtype.str)
2081: (29)
2082: (30)
2083: (29)
2084: (8)                  self.orders = {'C': 'c_contiguous', 'F': 'f_contiguous'}
2085: (8)                  self.ndims = 10
2086: (4)              def check_function(self, func, fill_value=None):

```

```

2087: (8)           par = ((0, 1, 2),
2088: (15)          range(self.ndims),
2089: (15)          self.orders,
2090: (15)          self.dtypes)
2091: (8)          fill_kwarg = {}
2092: (8)          if fill_value is not None:
2093: (12)             fill_kwarg = {'fill_value': fill_value}
2094: (8)          for size, ndims, order, dtype in itertools.product(*par):
2095: (12)             shape = ndims * [size]
2096: (12)             if fill_kwarg and dtype.str.startswith('|V'):
2097: (16)               continue
2098: (12)             arr = func(shape, order=order, dtype=dtype,
2099: (23)                           **fill_kwarg)
2100: (12)             assert_equal(arr.dtype, dtype)
2101: (12)             assert_(getattr(arr.flags, self.orders[order]))
2102: (12)             if fill_value is not None:
2103: (16)               if dtype.str.startswith('|S'):
2104: (20)                 val = str(fill_value)
2105: (16)               else:
2106: (20)                 val = fill_value
2107: (16)             assert_equal(arr, dtype.type(val))
2108: (4)           def test_zeros(self):
2109: (8)             self.check_function(np.zeros)
2110: (4)           def test_ones(self):
2111: (8)             self.check_function(np.ones)
2112: (4)           def test_empty(self):
2113: (8)             self.check_function(np.empty)
2114: (4)           def test_full(self):
2115: (8)             self.check_function(np.full, 0)
2116: (8)             self.check_function(np.full, 1)
2117: (4) @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
2118: (4)           def test_for_reference_leak(self):
2119: (8)             dim = 1
2120: (8)             beg = sys.getrefcount(dim)
2121: (8)             np.zeros([dim]*10)
2122: (8)             assert_(sys.getrefcount(dim) == beg)
2123: (8)             np.ones([dim]*10)
2124: (8)             assert_(sys.getrefcount(dim) == beg)
2125: (8)             np.empty([dim]*10)
2126: (8)             assert_(sys.getrefcount(dim) == beg)
2127: (8)             np.full([dim]*10, 0)
2128: (8)             assert_(sys.getrefcount(dim) == beg)
2129: (0)           class TestLikeFuncs:
2130: (4)             '''Test ones_like, zeros_like, empty_like and full_like'''
2131: (4)             def setup_method(self):
2132: (8)               self.data = [
2133: (16)                 (np.array(3.), None),
2134: (16)                 (np.array(3), 'f8'),
2135: (16)                 (np.arange(6, dtype='f4'), None),
2136: (16)                 (np.arange(6), 'c16'),
2137: (16)                 (np.arange(6).reshape(2, 3), None),
2138: (16)                 (np.arange(6).reshape(3, 2), 'i1'),
2139: (16)                 (np.arange(6).reshape((2, 3), order='F'), None),
2140: (16)                 (np.arange(6).reshape((3, 2), order='F'), 'i1'),
2141: (16)                 (np.arange(24).reshape(2, 3, 4), None),
2142: (16)                 (np.arange(24).reshape(4, 3, 2), 'f4'),
2143: (16)                 (np.arange(24).reshape((2, 3, 4), order='F'), None),
2144: (16)                 (np.arange(24).reshape((4, 3, 2), order='F'), 'f4'),
2145: (16)                 (np.arange(24).reshape(2, 3, 4).swapaxes(0, 1), None),
2146: (16)                 (np.arange(24).reshape(4, 3, 2).swapaxes(0, 1), '?'),
2147: (21)               ]
2148: (8)               self.shapes = [(), (5,), (5,6,), (5,6,7,)]
2149: (4)           def compare_array_value(self, dz, value, fill_value):
2150: (8)             if value is not None:
2151: (12)               if fill_value:
2152: (16)                 z = np.array(value).astype(dz.dtype)
2153: (16)                 assert_(np.all(dz == z))
2154: (12)               else:
2155: (16)                 assert_(np.all(dz == value))

```

```

2156: (4)
2157: (8)
2158: (12)
2159: (8)
2160: (12)
2161: (8)
2162: (12)
2163: (12)
2164: (12)
2165: (25)
2166: (12)
2167: (12)
2168: (12)
2169: (16)
2170: (12)
2171: (16)
2172: (12)
2173: (12)
2174: (12)
2175: (12)
2176: (12)
2177: (16)
2178: (12)
2179: (16)
2180: (12)
2181: (12)
2182: (12)
2183: (12)
2184: (12)
2185: (16)
2186: (12)
2187: (16)
2188: (12)
2189: (12)
2190: (12)
2191: (12)
2192: (16)
2193: (12)
2194: (16)
2195: (12)
2196: (16)
2197: (12)
2198: (16)
2199: (12)
2200: (12)
2201: (16)
2202: (20)
2203: (39)
2204: (20)
2205: (20)
2206: (24)
2207: (20)
2208: (24)
2209: (20)
2210: (24)
2211: (20)
2212: (24)
2213: (20)
2214: (16)
2215: (20)
2216: (58)
2217: (58)
**fill_kwarg).strides),
2218: (33)
2219: (53)
2220: (16)
2221: (20)
2222: (58)
2223: (58)

def check_like_function(self, like_function, value, fill_value=False):
    if fill_value:
        fill_kwarg = {'fill_value': value}
    else:
        fill_kwarg = {}
    for d, dtype in self.data:
        dz = like_function(d, dtype=dtype, **fill_kwarg)
        assert_equal(dz.shape, d.shape)
        assert_equal(np.array(dz.strides)*d.dtype.itemsize,
                    np.array(d.strides)*dz.dtype.itemsize)
        assert_equal(d.flags.c_contiguous, dz.flags.c_contiguous)
        assert_equal(d.flags.f_contiguous, dz.flags.f_contiguous)
        if dtype is None:
            assert_equal(dz.dtype, d.dtype)
        else:
            assert_equal(dz.dtype, np.dtype(dtype))
        self.compare_array_value(dz, value, fill_value)
        dz = like_function(d, order='C', dtype=dtype, **fill_kwarg)
        assert_equal(dz.shape, d.shape)
        assert_(dz.flags.c_contiguous)
        if dtype is None:
            assert_equal(dz.dtype, d.dtype)
        else:
            assert_equal(dz.dtype, np.dtype(dtype))
        self.compare_array_value(dz, value, fill_value)
        dz = like_function(d, order='F', dtype=dtype, **fill_kwarg)
        assert_equal(dz.shape, d.shape)
        assert_(dz.flags.f_contiguous)
        if dtype is None:
            assert_equal(dz.dtype, d.dtype)
        else:
            assert_equal(dz.dtype, np.dtype(dtype))
        self.compare_array_value(dz, value, fill_value)
        dz = like_function(d, order='A', dtype=dtype, **fill_kwarg)
        assert_equal(dz.shape, d.shape)
        if d.flags.f_contiguous:
            assert_(dz.flags.f_contiguous)
        else:
            assert_(dz.flags.c_contiguous)
        if dtype is None:
            assert_equal(dz.dtype, d.dtype)
        else:
            assert_equal(dz.dtype, np.dtype(dtype))
        self.compare_array_value(dz, value, fill_value)
        for s in self.shapes:
            for o in 'CFA':
                sz = like_function(d, dtype=dtype, shape=s, order=o,
                                    **fill_kwarg)
                assert_equal(sz.shape, s)
                if dtype is None:
                    assert_equal(sz.dtype, d.dtype)
                else:
                    assert_equal(sz.dtype, np.dtype(dtype))
                if o == 'C' or (o == 'A' and d.flags.c_contiguous):
                    assert_(sz.flags.c_contiguous)
                elif o == 'F' or (o == 'A' and d.flags.f_contiguous):
                    assert_(sz.flags.f_contiguous)
                self.compare_array_value(sz, value, fill_value)
            if (d.ndim != len(s)):
                assert_equal(np.argsort(like_function(d, dtype=dtype,
  shape=s, order='K',
**fill_kwarg).strides),
                            np.argsort(np.empty(s, dtype=dtype,
  order='C').strides))
            else:
                assert_equal(np.argsort(like_function(d, dtype=dtype,
  shape=s, order='K',
**fill_kwarg).strides),
                            np.argsort(np.empty(s, dtype=dtype,
  order='C').strides))

```

```

**fill_kwarg).strides),
2224: (33)                                     np.argsort(d.strides))
2225: (8)          class MyNDArray(np.ndarray):
2226: (12)            pass
2227: (8)            a = np.array([[1, 2], [3, 4]]).view(MyNDArray)
2228: (8)            b = like_function(a, **fill_kwarg)
2229: (8)            assert_(type(b) is MyNDArray)
2230: (8)            b = like_function(a, subok=False, **fill_kwarg)
2231: (8)            assert_(type(b) is not MyNDArray)
2232: (4)          def test_ones_like(self):
2233: (8)            self.check_like_function(np.ones_like, 1)
2234: (4)          def test_zeros_like(self):
2235: (8)            self.check_like_function(np.zeros_like, 0)
2236: (4)          def test_empty_like(self):
2237: (8)            self.check_like_function(np.empty_like, None)
2238: (4)          def test_filled_like(self):
2239: (8)            self.check_like_function(np.full_like, 0, True)
2240: (8)            self.check_like_function(np.full_like, 1, True)
2241: (8)            self.check_like_function(np.full_like, 1000, True)
2242: (8)            self.check_like_function(np.full_like, 123.456, True)
2243: (8)            with np.errstate(invalid="ignore"):
2244: (12)              self.check_like_function(np.full_like, np.inf, True)
2245: (4)          @pytest.mark.parametrize('likefunc', [np.empty_like, np.full_like,
2246: (42)   np.zeros_like, np.ones_like])
2247: (4)          @pytest.mark.parametrize('dtype', [str, bytes])
2248: (4)          def test_dtype_str_bytes(self, likefunc, dtype):
2249: (8)            a = np.arange(16).reshape(2, 8)
2250: (8)            b = a[:, ::2] # Ensure b is not contiguous.
2251: (8)            kwargs = {'fill_value': ''} if likefunc == np.full_like else {}
2252: (8)            result = likefunc(b, dtype=dtype, **kwargs)
2253: (8)            if dtype == str:
2254: (12)              assert result.strides == (16, 4)
2255: (8)            else:
2256: (12)              assert result.strides == (4, 1)
2257: (0)          class TestCorrelate:
2258: (4)            def _setup(self, dt):
2259: (8)              self.x = np.array([1, 2, 3, 4, 5], dtype=dt)
2260: (8)              self.xs = np.arange(1, 20)[::3]
2261: (8)              self.y = np.array([-1, -2, -3], dtype=dt)
2262: (8)              self.z1 = np.array([-3., -8., -14., -20., -26., -14., -5.], dtype=dt)
2263: (8)              self.z1_4 = np.array([-2., -5., -8., -11., -14., -5.], dtype=dt)
2264: (8)              self.z1r = np.array([-15., -22., -22., -16., -10., -4., -1.],
2265: (8)                dtype=dt)
2266: (8)              self.z2 = np.array([-5., -14., -26., -20., -14., -8., -3.], dtype=dt)
2267: (8)              self.z2r = np.array([-1., -4., -10., -16., -22., -22., -15.],
2268: (27)                dtype=dt)
2269: (4)            def test_float(self):
2270: (8)              self._setup(float)
2271: (8)              z = np.correlate(self.x, self.y, 'full')
2272: (8)              assert_array_almost_equal(z, self.z1)
2273: (8)              z = np.correlate(self.x, self.y[:-1], 'full')
2274: (8)              assert_array_almost_equal(z, self.z1_4)
2275: (8)              z = np.correlate(self.y, self.x, 'full')
2276: (8)              assert_array_almost_equal(z, self.z2)
2277: (8)              z = np.correlate(self.x[::-1], self.y, 'full')
2278: (8)              assert_array_almost_equal(z, self.z1r)
2279: (8)              z = np.correlate(self.y, self.x[::-1], 'full')
2280: (8)              assert_array_almost_equal(z, self.z2r)
2281: (8)              z = np.correlate(self.xs, self.y, 'full')
2282: (8)              assert_array_almost_equal(z, self.zs)
2283: (4)            def test_object(self):
2284: (8)              self._setup(Decimal)
2285: (8)              z = np.correlate(self.x, self.y, 'full')
2286: (8)              assert_array_almost_equal(z, self.z1)
2287: (8)              z = np.correlate(self.y, self.x, 'full')
2288: (8)              assert_array_almost_equal(z, self.z2)
2289: (4)            def test_no_overwrite(self):

```

```

2290: (8)             d = np.ones(100)
2291: (8)             k = np.ones(3)
2292: (8)             np.correlate(d, k)
2293: (8)             assert_array_equal(d, np.ones(100))
2294: (8)             assert_array_equal(k, np.ones(3))
2295: (4)             def test_complex(self):
2296: (8)                 x = np.array([1, 2, 3, 4+1j], dtype=complex)
2297: (8)                 y = np.array([-1, -2j, 3+1j], dtype=complex)
2298: (8)                 r_z = np.array([-3-1j, 6, 8+1j, 11+5j, -5+8j, -4-1j], dtype=complex)
2299: (8)                 r_z = r_z[::-1].conjugate()
2300: (8)                 z = np.correlate(y, x, mode='full')
2301: (8)                 assert_array_almost_equal(z, r_z)
2302: (4)             def test_zero_size(self):
2303: (8)                 with pytest.raises(ValueError):
2304: (12)                     np.correlate(np.array([]), np.ones(1000), mode='full')
2305: (8)                 with pytest.raises(ValueError):
2306: (12)                     np.correlate(np.ones(1000), np.array([]), mode='full')
2307: (4)             def test_mode(self):
2308: (8)                 d = np.ones(100)
2309: (8)                 k = np.ones(3)
2310: (8)                 default_mode = np.correlate(d, k, mode='valid')
2311: (8)                 with assert_warns(DeprecationWarning):
2312: (12)                     valid_mode = np.correlate(d, k, mode='v')
2313: (8)                 assert_array_equal(valid_mode, default_mode)
2314: (8)                 with assert_raises(ValueError):
2315: (12)                     np.correlate(d, k, mode=-1)
2316: (8)                 assert_array_equal(np.correlate(d, k, mode=0), valid_mode)
2317: (8)                 with assert_raises(TypeError):
2318: (12)                     np.correlate(d, k, mode=None)
2319: (0)             class TestConvolve:
2320: (4)                 def test_object(self):
2321: (8)                     d = [1.] * 100
2322: (8)                     k = [1.] * 3
2323: (8)                     assert_array_almost_equal(np.convolve(d, k)[2:-2], np.full(98, 3))
2324: (4)                 def test_no_overwrite(self):
2325: (8)                     d = np.ones(100)
2326: (8)                     k = np.ones(3)
2327: (8)                     np.convolve(d, k)
2328: (8)                     assert_array_equal(d, np.ones(100))
2329: (8)                     assert_array_equal(k, np.ones(3))
2330: (4)                 def test_mode(self):
2331: (8)                     d = np.ones(100)
2332: (8)                     k = np.ones(3)
2333: (8)                     default_mode = np.convolve(d, k, mode='full')
2334: (8)                     with assert_warns(DeprecationWarning):
2335: (12)                         full_mode = np.convolve(d, k, mode='f')
2336: (8)                     assert_array_equal(full_mode, default_mode)
2337: (8)                     with assert_raises(ValueError):
2338: (12)                         np.convolve(d, k, mode=-1)
2339: (8)                     assert_array_equal(np.convolve(d, k, mode=2), full_mode)
2340: (8)                     with assert_raises(TypeError):
2341: (12)                         np.convolve(d, k, mode=None)
2342: (0)             class TestArgwhere:
2343: (4)                 @pytest.mark.parametrize('nd', [0, 1, 2])
2344: (4)                 def test_nd(self, nd):
2345: (8)                     x = np.empty((2,)*nd, bool)
2346: (8)                     x[...] = False
2347: (8)                     assert_equal(np.argwhere(x).shape, (0, nd))
2348: (8)                     x[...] = True
2349: (8)                     x.flat[0] = False
2350: (8)                     assert_equal(np.argwhere(x).shape, (1, nd))
2351: (8)                     x[...] = True
2352: (8)                     x.flat[0] = False
2353: (8)                     assert_equal(np.argwhere(x).shape, (x.size - 1, nd))
2354: (8)                     x[...] = True
2355: (8)                     assert_equal(np.argwhere(x).shape, (x.size, nd))
2356: (4)                 def test_2D(self):
2357: (8)                     x = np.arange(6).reshape((2, 3))
2358: (8)                     assert_array_equal(np.argwhere(x > 1),

```

```

2359: (27)
2360: (28)
2361: (28)
2362: (28)
2363: (4)           [[0, 2],
2364: (8)             [1, 0],
2365: (0)             [1, 1],
2366: (4)             [1, 2]])
2367: (8)         def test_list(self):
2368: (8)             assert_equal(np.argwhere([4, 0, 2, 1, 3]), [[0], [2], [3], [4]])
2369: (8)         class TestStringFunction:
2370: (8)             def test_set_string_function(self):
2371: (8)                 a = np.array([1])
2372: (8)                 np.set_string_function(lambda x: "FOO", repr=True)
2373: (8)                 assert_equal(repr(a), "FOO")
2374: (8)                 np.set_string_function(None, repr=True)
2375: (8)                 assert_equal(repr(a), "array([1])")
2376: (0)                 np.set_string_function(lambda x: "FOO", repr=False)
2377: (4)                 assert_equal(str(a), "FOO")
2378: (8)                 np.set_string_function(None, repr=False)
2379: (8)                 assert_equal(str(a), "[1]")
2380: (8)         class TestRoll:
2381: (4)             def test_roll1d(self):
2382: (8)                 x = np.arange(10)
2383: (8)                 xr = np.roll(x, 2)
2384: (8)                 assert_equal(xr, np.array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7]))
2385: (8)             def test_roll2d(self):
2386: (8)                 x2 = np.reshape(np.arange(10), (2, 5))
2387: (8)                 x2r = np.roll(x2, 1)
2388: (8)                 assert_equal(x2r, np.array([[9, 0, 1, 2, 3], [4, 5, 6, 7, 8]]))
2389: (8)                 x2r = np.roll(x2, 1, axis=0)
2390: (8)                 assert_equal(x2r, np.array([[5, 6, 7, 8, 9], [0, 1, 2, 3, 4]]))
2391: (8)                 x2r = np.roll(x2, 1, axis=1)
2392: (8)                 assert_equal(x2r, np.array([[4, 0, 1, 2, 3], [9, 5, 6, 7, 8]]))
2393: (8)                 x2r = np.roll(x2, 1, axis=(0, 1))
2394: (8)                 assert_equal(x2r, np.array([[9, 5, 6, 7, 8], [4, 0, 1, 2, 3]]))
2395: (8)                 x2r = np.roll(x2, (1, 0), axis=(0, 1))
2396: (8)                 assert_equal(x2r, np.array([[5, 6, 7, 8, 9], [0, 1, 2, 3, 4]]))
2397: (8)                 x2r = np.roll(x2, (-1, 0), axis=(0, 1))
2398: (8)                 assert_equal(x2r, np.array([[5, 6, 7, 8, 9], [0, 1, 2, 3, 4]]))
2399: (8)                 x2r = np.roll(x2, (0, 1), axis=(0, 1))
2400: (8)                 assert_equal(x2r, np.array([[4, 0, 1, 2, 3], [9, 5, 6, 7, 8]]))
2401: (8)                 x2r = np.roll(x2, (0, -1), axis=(0, 1))
2402: (8)                 assert_equal(x2r, np.array([[1, 2, 3, 4, 0], [6, 7, 8, 9, 5]]))
2403: (8)                 x2r = np.roll(x2, (1, 1), axis=(0, 1))
2404: (8)                 assert_equal(x2r, np.array([[9, 5, 6, 7, 8], [4, 0, 1, 2, 3]]))
2405: (8)                 x2r = np.roll(x2, 1, axis=(0, 0))
2406: (8)                 assert_equal(x2r, np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]))
2407: (8)                 x2r = np.roll(x2, 1, axis=(1, 1))
2408: (8)                 assert_equal(x2r, np.array([[3, 4, 0, 1, 2], [8, 9, 5, 6, 7]]))
2409: (8)                 x2r = np.roll(x2, 6, axis=1)
2410: (8)                 assert_equal(x2r, np.array([[4, 0, 1, 2, 3], [9, 5, 6, 7, 8]]))
2411: (4)             def test_roll_empty(self):
2412: (8)                 x = np.array([])
2413: (8)                 assert_equal(np.roll(x, 1), np.array([]))
2414: (0)         class TestRollaxis:
2415: (4)             tgtshape = {(0, 0): (1, 2, 3, 4), (0, 1): (1, 2, 3, 4),
2416: (16)               (0, 2): (2, 1, 3, 4), (0, 3): (2, 3, 1, 4),
2417: (16)               (0, 4): (2, 3, 4, 1),
2418: (16)               (1, 0): (2, 1, 3, 4), (1, 1): (1, 2, 3, 4),
2419: (16)               (1, 2): (1, 2, 3, 4), (1, 3): (1, 3, 2, 4),
2420: (16)               (1, 4): (1, 3, 4, 2),
2421: (16)               (2, 0): (3, 1, 2, 4), (2, 1): (1, 3, 2, 4),
2422: (16)               (2, 2): (1, 2, 3, 4), (2, 3): (1, 2, 3, 4),
2423: (16)               (2, 4): (1, 2, 4, 3),
2424: (16)               (3, 0): (4, 1, 2, 3), (3, 1): (1, 4, 2, 3),
2425: (16)               (3, 2): (1, 2, 4, 3), (3, 3): (1, 2, 3, 4),
2426: (16)               (3, 4): (1, 2, 3, 4)}
2427: (4)         def test_exceptions(self):

```

```

2428: (8) a = np.arange(1*2*3*4).reshape(1, 2, 3, 4)
2429: (8) assert_raises(np.AxisError, np.rollaxis, a, -5, 0)
2430: (8) assert_raises(np.AxisError, np.rollaxis, a, 0, -5)
2431: (8) assert_raises(np.AxisError, np.rollaxis, a, 4, 0)
2432: (8) assert_raises(np.AxisError, np.rollaxis, a, 0, 5)
2433: (4) def test_results(self):
2434: (8)     a = np.arange(1*2*3*4).reshape(1, 2, 3, 4).copy()
2435: (8)     aind = np.indices(a.shape)
2436: (8)     assert_(a.flags['OWNDATA'])
2437: (8)     for (i, j) in self.tgtshape:
2438: (12)         res = np.rollaxis(a, axis=i, start=j)
2439: (12)         i0, i1, i2, i3 = aind[np.array(res.shape) - 1]
2440: (12)         assert_(np.all(res[i0, i1, i2, i3] == a))
2441: (12)         assert_(res.shape == self.tgtshape[(i, j)], str((i,j)))
2442: (12)         assert_(not res.flags['OWNDATA'])
2443: (12)         ip = i + 1
2444: (12)         res = np.rollaxis(a, axis=-ip, start=j)
2445: (12)         i0, i1, i2, i3 = aind[np.array(res.shape) - 1]
2446: (12)         assert_(np.all(res[i0, i1, i2, i3] == a))
2447: (12)         assert_(res.shape == self.tgtshape[(4 - ip, j)])
2448: (12)         assert_(not res.flags['OWNDATA'])
2449: (12)         jp = j + 1 if j < 4 else j
2450: (12)         res = np.rollaxis(a, axis=i, start=-jp)
2451: (12)         i0, i1, i2, i3 = aind[np.array(res.shape) - 1]
2452: (12)         assert_(np.all(res[i0, i1, i2, i3] == a))
2453: (12)         assert_(res.shape == self.tgtshape[(i, 4 - jp)])
2454: (12)         assert_(not res.flags['OWNDATA'])
2455: (12)         ip = i + 1
2456: (12)         jp = j + 1 if j < 4 else j
2457: (12)         res = np.rollaxis(a, axis=-ip, start=-jp)
2458: (12)         i0, i1, i2, i3 = aind[np.array(res.shape) - 1]
2459: (12)         assert_(np.all(res[i0, i1, i2, i3] == a))
2460: (12)         assert_(res.shape == self.tgtshape[(4 - ip, 4 - jp)])
2461: (12)         assert_(not res.flags['OWNDATA'])
2462: (0) class TestMoveaxis:
2463: (4)     def test_move_to_end(self):
2464: (8)         x = np.random.randn(5, 6, 7)
2465: (8)         for source, expected in [(0, (6, 7, 5)),
2466: (33)             (1, (5, 7, 6)),
2467: (33)             (2, (5, 6, 7)),
2468: (33)             (-1, (5, 6, 7))]:
2469: (12)             actual = np.moveaxis(x, source, -1).shape
2470: (12)             assert_(actual, expected)
2471: (4)     def test_move_new_position(self):
2472: (8)         x = np.random.randn(1, 2, 3, 4)
2473: (8)         for source, destination, expected in [
2474: (16)             (0, 1, (2, 1, 3, 4)),
2475: (16)             (1, 2, (1, 3, 2, 4)),
2476: (16)             (1, -1, (1, 3, 4, 2)),
2477: (16)             ]:
2478: (12)             actual = np.moveaxis(x, source, destination).shape
2479: (12)             assert_(actual, expected)
2480: (4)     def test_preserve_order(self):
2481: (8)         x = np.zeros((1, 2, 3, 4))
2482: (8)         for source, destination in [
2483: (16)             (0, 0),
2484: (16)             (3, -1),
2485: (16)             (-1, 3),
2486: (16)             ([0, -1], [0, -1]),
2487: (16)             ([2, 0], [2, 0]),
2488: (16)             (range(4), range(4)),
2489: (16)             ]:
2490: (12)             actual = np.moveaxis(x, source, destination).shape
2491: (12)             assert_(actual, (1, 2, 3, 4))
2492: (4)     def test_move_multiples(self):
2493: (8)         x = np.zeros((0, 1, 2, 3))
2494: (8)         for source, destination, expected in [
2495: (16)             ([0, 1], [2, 3], (2, 3, 0, 1)),
2496: (16)             ([2, 3], [0, 1], (2, 3, 0, 1)),

```

```

2497: (16) ([0, 1, 2], [2, 3, 0], (2, 3, 0, 1)),
2498: (16) ([3, 0], [1, 0], (0, 3, 1, 2)),
2499: (16) ([0, 3], [0, 1], (0, 3, 1, 2)),
2500: (16) ]:
2501: (12)     actual = np.moveaxis(x, source, destination).shape
2502: (12)     assert_(actual, expected)
2503: (4) def test_errors(self):
2504: (8)     x = np.random.randn(1, 2, 3)
2505: (8)     assert_raises_regex(np.AxisError, 'source.*out of bounds',
2506: (28)                 np.moveaxis, x, 3, 0)
2507: (8)     assert_raises_regex(np.AxisError, 'source.*out of bounds',
2508: (28)                 np.moveaxis, x, -4, 0)
2509: (8)     assert_raises_regex(np.AxisError, 'destination.*out of bounds',
2510: (28)                 np.moveaxis, x, 0, 5)
2511: (8)     assert_raises_regex(ValueError, 'repeated axis in `source`',
2512: (28)                 np.moveaxis, x, [0, 0], [0, 1])
2513: (8)     assert_raises_regex(ValueError, 'repeated axis in `destination`',
2514: (28)                 np.moveaxis, x, [0, 1], [1, 1])
2515: (8)     assert_raises_regex(ValueError, 'must have the same number',
2516: (28)                 np.moveaxis, x, 0, [0, 1])
2517: (8)     assert_raises_regex(ValueError, 'must have the same number',
2518: (28)                 np.moveaxis, x, [0, 1], [0])
2519: (4) def test_array_likes(self):
2520: (8)     x = np.ma.zeros((1, 2, 3))
2521: (8)     result = np.moveaxis(x, 0, 0)
2522: (8)     assert_(x.shape, result.shape)
2523: (8)     assert_(isinstance(result, np.ma.MaskedArray))
2524: (8)     x = [1, 2, 3]
2525: (8)     result = np.moveaxis(x, 0, 0)
2526: (8)     assert_(x, list(result))
2527: (8)     assert_(isinstance(result, np.ndarray))
2528: (0) class TestCross:
2529: (4)     def test_2x2(self):
2530: (8)         u = [1, 2]
2531: (8)         v = [3, 4]
2532: (8)         z = -2
2533: (8)         cp = np.cross(u, v)
2534: (8)         assert_equal(cp, z)
2535: (8)         cp = np.cross(v, u)
2536: (8)         assert_equal(cp, -z)
2537: (4)     def test_2x3(self):
2538: (8)         u = [1, 2]
2539: (8)         v = [3, 4, 5]
2540: (8)         z = np.array([10, -5, -2])
2541: (8)         cp = np.cross(u, v)
2542: (8)         assert_equal(cp, z)
2543: (8)         cp = np.cross(v, u)
2544: (8)         assert_equal(cp, -z)
2545: (4)     def test_3x3(self):
2546: (8)         u = [1, 2, 3]
2547: (8)         v = [4, 5, 6]
2548: (8)         z = np.array([-3, 6, -3])
2549: (8)         cp = np.cross(u, v)
2550: (8)         assert_equal(cp, z)
2551: (8)         cp = np.cross(v, u)
2552: (8)         assert_equal(cp, -z)
2553: (4)     def test_broadcasting(self):
2554: (8)         u = np.tile([1, 2], (11, 1))
2555: (8)         v = np.tile([3, 4], (11, 1))
2556: (8)         z = -2
2557: (8)         assert_equal(np.cross(u, v), z)
2558: (8)         assert_equal(np.cross(v, u), -z)
2559: (8)         assert_equal(np.cross(u, u), 0)
2560: (8)         u = np.tile([1, 2], (11, 1)).T
2561: (8)         v = np.tile([3, 4, 5], (11, 1))
2562: (8)         z = np.tile([10, -5, -2], (11, 1))
2563: (8)         assert_equal(np.cross(u, v, axis=0), z)
2564: (8)         assert_equal(np.cross(v, u.T), -z)
2565: (8)         assert_equal(np.cross(v, v), 0)

```

```

2566: (8)          u = np.tile([1, 2, 3], (11, 1)).T
2567: (8)          v = np.tile([3, 4], (11, 1)).T
2568: (8)          z = np.tile([-12, 9, -2], (11, 1))
2569: (8)          assert_equal(np.cross(u, v, axisa=0, axisb=0), z)
2570: (8)          assert_equal(np.cross(v.T, u.T), -z)
2571: (8)          assert_equal(np.cross(u.T, u.T), 0)
2572: (8)          u = np.tile([1, 2, 3], (5, 1))
2573: (8)          v = np.tile([4, 5, 6], (5, 1)).T
2574: (8)          z = np.tile([-3, 6, -3], (5, 1))
2575: (8)          assert_equal(np.cross(u, v, axisb=0), z)
2576: (8)          assert_equal(np.cross(v.T, u), -z)
2577: (8)          assert_equal(np.cross(u, u), 0)
2578: (4)          def test_broadcasting_shapes(self):
2579: (8)              u = np.ones((2, 1, 3))
2580: (8)              v = np.ones((5, 3))
2581: (8)              assert_equal(np.cross(u, v).shape, (2, 5, 3))
2582: (8)              u = np.ones((10, 3, 5))
2583: (8)              v = np.ones((2, 5))
2584: (8)              assert_equal(np.cross(u, v, axisa=1, axisb=0).shape, (10, 5, 3))
2585: (8)              assert_raises(np.AxisError, np.cross, u, v, axisa=1, axisb=2)
2586: (8)              assert_raises(np.AxisError, np.cross, u, v, axisa=3, axisb=0)
2587: (8)              u = np.ones((10, 3, 5, 7))
2588: (8)              v = np.ones((5, 7, 2))
2589: (8)              assert_equal(np.cross(u, v, axisa=1, axisc=2).shape, (10, 5, 3, 7))
2590: (8)              assert_raises(np.AxisError, np.cross, u, v, axisa=-5, axisb=2)
2591: (8)              assert_raises(np.AxisError, np.cross, u, v, axisa=1, axisb=-4)
2592: (8)              u = np.ones((3, 4, 2))
2593: (8)              for axisc in range(-2, 2):
2594: (12)                  assert_equal(np.cross(u, u, axisc=axisc).shape, (3, 4))
2595: (4)          def test_uint8_int32_mixed_dtypes(self):
2596: (8)              u = np.array([[195, 8, 9]], np.uint8)
2597: (8)              v = np.array([250, 166, 68], np.int32)
2598: (8)              z = np.array([[950, 11010, -30370]], dtype=np.int32)
2599: (8)              assert_equal(np.cross(v, u), z)
2600: (8)              assert_equal(np.cross(u, v), -z)
2601: (0)          def test_outer_out_param():
2602: (4)              arr1 = np.ones((5,))
2603: (4)              arr2 = np.ones((2,))
2604: (4)              arr3 = np.linspace(-2, 2, 5)
2605: (4)              out1 = np.ndarray(shape=(5,5))
2606: (4)              out2 = np.ndarray(shape=(2, 5))
2607: (4)              res1 = np.outer(arr1, arr3, out1)
2608: (4)              assert_equal(res1, out1)
2609: (4)              assert_equal(np.outer(arr2, arr3, out2), out2)
2610: (0)          class TestIndices:
2611: (4)              def test_simple(self):
2612: (8)                  [x, y] = np.indices((4, 3))
2613: (8)                  assert_array_equal(x, np.array([[0, 0, 0],
2614: (40)                                  [1, 1, 1],
2615: (40)                                  [2, 2, 2],
2616: (40)                                  [3, 3, 3]]))
2617: (8)                  assert_array_equal(y, np.array([[0, 1, 2],
2618: (40)                                  [0, 1, 2],
2619: (40)                                  [0, 1, 2],
2620: (40)                                  [0, 1, 2]]))
2621: (4)              def test_single_input(self):
2622: (8)                  [x] = np.indices((4,))
2623: (8)                  assert_array_equal(x, np.array([0, 1, 2, 3]))
2624: (8)                  [x] = np.indices((4,), sparse=True)
2625: (8)                  assert_array_equal(x, np.array([0, 1, 2, 3]))
2626: (4)              def test_scalar_input(self):
2627: (8)                  assert_array_equal([], np.indices(()))
2628: (8)                  assert_array_equal([], np.indices(()), sparse=True)
2629: (8)                  assert_array_equal([], np.indices((0,)))
2630: (8)                  assert_array_equal([], np.indices((0,), sparse=True))
2631: (4)              def test_sparse(self):
2632: (8)                  [x, y] = np.indices((4,3), sparse=True)
2633: (8)                  assert_array_equal(x, np.array([[0], [1], [2], [3]]))
2634: (8)                  assert_array_equal(y, np.array([[0, 1, 2]]))

```

```

2635: (4) @pytest.mark.parametrize("dtype", [np.int32, np.int64, np.float32,
2636: (4) np.float64])
2636: (4) @pytest.mark.parametrize("dims", [(), (0,), (4, 3)])
2637: (4) def test_return_type(self, dtype, dims):
2638: (8)     inds = np.indices(dims, dtype=dtype)
2639: (8)     assert_(inds.dtype == dtype)
2640: (8)     for arr in np.indices(dims, dtype=dtype, sparse=True):
2641: (12)         assert_(arr.dtype == dtype)
2642: (0) class TestRequire:
2643: (4)     flag_names = ['C', 'C_CONTIGUOUS', 'CONTIGUOUS',
2644: (8)             'F', 'F_CONTIGUOUS', 'FORTRAN',
2645: (8)             'A', 'ALIGNED',
2646: (8)             'W', 'WRITEABLE',
2647: (8)             'O', 'OWNDATA']
2648: (4)     def generate_all_false(self, dtype):
2649: (8)         arr = np.zeros((2, 2), [('junk', 'i1'), ('a', dtype)])
2650: (8)         arr.setflags(write=False)
2651: (8)         a = arr['a']
2652: (8)         assert_(not a.flags['C'])
2653: (8)         assert_(not a.flags['F'])
2654: (8)         assert_(not a.flags['O'])
2655: (8)         assert_(not a.flags['W'])
2656: (8)         assert_(not a.flags['A'])
2657: (8)         return a
2658: (4)     def set_and_check_flag(self, flag, dtype, arr):
2659: (8)         if dtype is None:
2660: (12)             dtype = arr.dtype
2661: (8)         b = np.require(arr, dtype, [flag])
2662: (8)         assert_(b.flags[flag])
2663: (8)         assert_(b.dtype == dtype)
2664: (8)         c = np.require(b, None, [flag])
2665: (8)         if flag[0] != 'O':
2666: (12)             assert_(c is b)
2667: (8)         else:
2668: (12)             assert_(c.flags[flag])
2669: (4)     def test_require_each(self):
2670: (8)         id = ['f8', 'i4']
2671: (8)         fd = [None, 'f8', 'c16']
2672: (8)         for idtype, fdtype, flag in itertools.product(id, fd,
self.flag_names):
2673: (12)             a = self.generate_all_false(idtype)
2674: (12)             self.set_and_check_flag(flag, fdtype, a)
2675: (4)     def test_unknown_requirement(self):
2676: (8)         a = self.generate_all_false('f8')
2677: (8)         assert_raises(KeyError, np.require, a, None, 'Q')
2678: (4)     def test_non_array_input(self):
2679: (8)         a = np.require([1, 2, 3, 4], 'i4', ['C', 'A', 'O'])
2680: (8)         assert_(a.flags['O'])
2681: (8)         assert_(a.flags['C'])
2682: (8)         assert_(a.flags['A'])
2683: (8)         assert_(a.dtype == 'i4')
2684: (8)         assert_equal(a, [1, 2, 3, 4])
2685: (4)     def test_C_and_F_simul(self):
2686: (8)         a = self.generate_all_false('f8')
2687: (8)         assert_raises(ValueError, np.require, a, None, ['C', 'F'])
2688: (4)     def test_ensure_array(self):
2689: (8)         class ArraySubclass(np.ndarray):
2690: (12)             pass
2691: (8)             a = ArraySubclass((2, 2))
2692: (8)             b = np.require(a, None, ['E'])
2693: (8)             assert_(type(b) is np.ndarray)
2694: (4)         def test_preserve_subtype(self):
2695: (8)             class ArraySubclass(np.ndarray):
2696: (12)                 pass
2697: (8)                 for flag in self.flag_names:
2698: (12)                     a = ArraySubclass((2, 2))
2699: (12)                     self.set_and_check_flag(flag, None, a)
2700: (0)         class TestBroadcast:
2701: (4)             def test_broadcast_in_args(self):

```

```

2702: (8)             arrs = [np.empty((6, 7)), np.empty((5, 6, 1)), np.empty((7,)),
2703: (16)             np.empty((5, 1, 7))]
2704: (8)             mits = [np.broadcast(*arrs),
2705: (16)                 np.broadcast(np.broadcast(*arrs[:0]),
np.broadcast(*arrs[0:])),,
2706: (16)                 np.broadcast(np.broadcast(*arrs[1:])),
2707: (16)                 np.broadcast(np.broadcast(*arrs[:2])),
2708: (16)                 np.broadcast(arrs[0], np.broadcast(*arrs[1:-1]), arrs[-1]))]
2709: (8)             for mit in mits:
2710: (12)                 assert_equal(mit.shape, (5, 6, 7))
2711: (12)                 assert_equal(mit.ndim, 3)
2712: (12)                 assert_equal(mit.nd, 3)
2713: (12)                 assert_equal(mit.numiter, 4)
2714: (12)                 for a, ia in zip(arrs, mit.iters):
2715: (16)                     assert_(a is ia.base)
2716: (4)             def test_broadcast_single_arg(self):
2717: (8)                 arrs = [np.empty((5, 6, 7))]
2718: (8)                 mit = np.broadcast(*arrs)
2719: (8)                 assert_equal(mit.shape, (5, 6, 7))
2720: (8)                 assert_equal(mit.ndim, 3)
2721: (8)                 assert_equal(mit.nd, 3)
2722: (8)                 assert_equal(mit.numiter, 1)
2723: (8)                 assert_(arrs[0] is mit.iters[0].base)
2724: (4)             def test_number_of_arguments(self):
2725: (8)                 arr = np.empty((5,))
2726: (8)                 for j in range(35):
2727: (12)                     arrs = [arr] * j
2728: (12)                     if j > 32:
2729: (16)                         assert_raises(ValueError, np.broadcast, *arrs)
2730: (12)                     else:
2731: (16)                         mit = np.broadcast(*arrs)
2732: (16)                         assert_equal(mit.numiter, j)
2733: (4)             def test_broadcast_error_kwarg(self):
2734: (8)                 arrs = [np.empty((5, 6, 7))]
2735: (8)                 mit = np.broadcast(*arrs)
2736: (8)                 mit2 = np.broadcast(*arrs, **{})
2737: (8)                 assert_equal(mit.shape, mit2.shape)
2738: (8)                 assert_equal(mit.ndim, mit2.ndim)
2739: (8)                 assert_equal(mit.nd, mit2.nd)
2740: (8)                 assert_equal(mit.numiter, mit2.numiter)
2741: (8)                 assert_(mit.iters[0].base is mit2.iters[0].base)
2742: (8)                 assert_raises(ValueError, np.broadcast, 1, **{'x': 1})
2743: (4)             def test_shape_mismatch_error_message(self):
2744: (8)                 with pytest.raises(ValueError, match=r"arg 0 with shape \((1, 3\)\) and "
2745: (45)                               r"arg 2 with shape \((2,\)\)":)
2746: (12)                     np.broadcast([[1, 2, 3]], [[4], [5]], [6, 7])
2747: (0)             class TestKeepdims:
2748: (4)                 class sub_array(np.ndarray):
2749: (8)                     def sum(self, axis=None, dtype=None, out=None):
2750: (12)                         return np.ndarray.sum(self, axis, dtype, out, keepdims=True)
2751: (4)             def test_raise(self):
2752: (8)                 sub_class = self.sub_array
2753: (8)                 x = np.arange(30).view(sub_class)
2754: (8)                 assert_raises(TypeError, np.sum, x, keepdims=True)
2755: (0)             class TestTensorDot:
2756: (4)                 def test_zero_dimension(self):
2757: (8)                     a = np.ndarray((3,0))
2758: (8)                     b = np.ndarray((0,4))
2759: (8)                     td = np.tensordot(a, b, (1, 0))
2760: (8)                     assert_array_equal(td, np.dot(a, b))
2761: (8)                     assert_array_equal(td, np.einsum('ij,jk', a, b))
2762: (4)                 def test_zero_dimensional(self):
2763: (8)                     arr_0d = np.array(1)
2764: (8)                     ret = np.tensordot(arr_0d, arr_0d, ([], [])) # contracting no axes is
well defined
2765: (8)                     assert_array_equal(ret, arr_0d)

```

## File 112 - test\_numericatypes.py:

```

1: (0)          import sys
2: (0)          import itertools
3: (0)          import pytest
4: (0)          import numpy as np
5: (0)          from numpy.testing import assert_, assert_equal, assert_raises, IS_PYPY
6: (0)          Pdescr = [
7: (4)              ('x', 'i4', (2,)),
8: (4)              ('y', 'f8', (2, 2)),
9: (4)              ('z', 'u1')]
10: (0)         PbufferT = [
11: (4)             ([3, 2], [[6., 4.], [6., 4.]], 8),
12: (4)             ([4, 3], [[7., 5.], [7., 5.]], 9),
13: (4)             ]
14: (0)         Ndescr = [
15: (4)             ('x', 'i4', (2,)),
16: (4)             ('Info', [
17: (8)                 ('value', 'c16'),
18: (8)                 ('y2', 'f8'),
19: (8)                 ('Info2', [
20: (12)                     ('name', 'S2'),
21: (12)                     ('value', 'c16', (2,)),
22: (12)                     ('y3', 'f8', (2,)),
23: (12)                     ('z3', 'u4', (2,))),
24: (8)                     ('name', 'S2'),
25: (8)                     ('z2', 'b1'))),
26: (4)             ('color', 'S2'),
27: (4)             ('info', [
28: (8)                 ('Name', 'U8'),
29: (8)                 ('Value', 'c16'))),
30: (4)             ('y', 'f8', (2, 2)),
31: (4)             ('z', 'u1')]
32: (0)         NbufferT = [
33: (4)             ([3, 2], (6j, 6., (b'nn', [6j, 4j], [6., 4.], [1, 2]), b'NN', True),
34: (5)             b'cc', ('NN', 6j), [[6., 4.], [6., 4.]], 8),
35: (4)             ([4, 3], (7j, 7., (b'oo', [7j, 5j], [7., 5.], [2, 1]), b'OO', False),
36: (5)             b'dd', ('OO', 7j), [[7., 5.], [7., 5.]], 9),
37: (4)             ]
38: (0)         byteorder = {'little': '<', 'big': '>'}[sys.byteorder]
39: (0)         def normalize_descr(descr):
40: (4)             "Normalize a description adding the platform byteorder."
41: (4)             out = []
42: (4)             for item in descr:
43: (8)                 dtype = item[1]
44: (8)                 if isinstance(dtype, str):
45: (12)                     if dtype[0] not in ['|', '<', '>']:
46: (16)                         onebyte = dtype[1:] == "1"
47: (16)                         if onebyte or dtype[0] in ['S', 'V', 'b']:
48: (20)                             dtype = "|" + dtype
49: (16)                         else:
50: (20)                             dtype = byteorder + dtype
51: (12)                     if len(item) > 2 and np.prod(item[2]) > 1:
52: (16)                         nitem = (item[0], dtype, item[2])
53: (12)                     else:
54: (16)                         nitem = (item[0], dtype)
55: (12)                     out.append(nitem)
56: (8)                 elif isinstance(dtype, list):
57: (12)                     l = normalize_descr(dtype)
58: (12)                     out.append((item[0], l))
59: (8)                 else:
60: (12)                     raise ValueError("Expected a str or list and got %s" %
61: (29)                                     (type(item))))
62: (4)             return out
63: (0)         class CreateZeros:
64: (4)             """Check the creation of heterogeneous arrays zero-valued"""
65: (4)             def test_zeros0D(self):

```

```

66: (8)         """Check creation of 0-dimensional objects"""
67: (8)         h = np.zeros((), dtype=self._descr)
68: (8)         assert_(normalize_descr(self._descr) == h.dtype.descr)
69: (8)         assert_(h.dtype.fields['x'][0].name[:4] == 'void')
70: (8)         assert_(h.dtype.fields['x'][0].char == 'V')
71: (8)         assert_(h.dtype.fields['x'][0].type == np void)
72: (8)         assert_equal(h['z'], np.zeros((), dtype='u1'))
73: (4)         def test_zerosSD(self):
74: (8)             """Check creation of single-dimensional objects"""
75: (8)             h = np.zeros((2,), dtype=self._descr)
76: (8)             assert_(normalize_descr(self._descr) == h.dtype.descr)
77: (8)             assert_(h.dtype['y'].name[:4] == 'void')
78: (8)             assert_(h.dtype['y'].char == 'V')
79: (8)             assert_(h.dtype['y'].type == np void)
80: (8)             assert_equal(h['z'], np.zeros((2,), dtype='u1'))
81: (4)         def test_zerosMD(self):
82: (8)             """Check creation of multi-dimensional objects"""
83: (8)             h = np.zeros((2, 3), dtype=self._descr)
84: (8)             assert_(normalize_descr(self._descr) == h.dtype.descr)
85: (8)             assert_(h.dtype['z'].name == 'uint8')
86: (8)             assert_(h.dtype['z'].char == 'B')
87: (8)             assert_(h.dtype['z'].type == np.uint8)
88: (8)             assert_equal(h['z'], np.zeros((2, 3), dtype='u1'))
89: (0)         class TestCreateZerosPlain(CreateZeros):
90: (4)             """Check the creation of heterogeneous arrays zero-valued (plain)"""
91: (4)             _descr = Pdescr
92: (0)         class TestCreateZerosNested(CreateZeros):
93: (4)             """Check the creation of heterogeneous arrays zero-valued (nested)"""
94: (4)             _descr = Ndescr
95: (0)         class CreateValues:
96: (4)             """Check the creation of heterogeneous arrays with values"""
97: (4)             def test_tuple(self):
98: (8)                 """Check creation from tuples"""
99: (8)                 h = np.array(self._buffer, dtype=self._descr)
100: (8)                assert_(normalize_descr(self._descr) == h.dtype.descr)
101: (8)                if self.multiple_rows:
102: (12)                  assert_(h.shape == (2,))
103: (8)                else:
104: (12)                  assert_(h.shape == ())
105: (4)                def test_list_of_tuple(self):
106: (8)                    """Check creation from list of tuples"""
107: (8)                    h = np.array([self._buffer], dtype=self._descr)
108: (8)                    assert_(normalize_descr(self._descr) == h.dtype.descr)
109: (8)                    if self.multiple_rows:
110: (12)                      assert_(h.shape == (1, 2))
111: (8)                    else:
112: (12)                      assert_(h.shape == (1,))
113: (4)                def test_list_of_list_of_tuple(self):
114: (8)                    """Check creation from list of list of tuples"""
115: (8)                    h = np.array([[self._buffer]], dtype=self._descr)
116: (8)                    assert_(normalize_descr(self._descr) == h.dtype.descr)
117: (8)                    if self.multiple_rows:
118: (12)                      assert_(h.shape == (1, 1, 2))
119: (8)                    else:
120: (12)                      assert_(h.shape == (1, 1))
121: (0)                class TestCreateValuesPlainSingle(CreateValues):
122: (4)                    """Check the creation of heterogeneous arrays (plain, single row)"""
123: (4)                    _descr = Pdescr
124: (4)                    multiple_rows = 0
125: (4)                    _buffer = PbufferT[0]
126: (0)                class TestCreateValuesPlainMultiple(CreateValues):
127: (4)                    """Check the creation of heterogeneous arrays (plain, multiple rows)"""
128: (4)                    _descr = Pdescr
129: (4)                    multiple_rows = 1
130: (4)                    _buffer = PbufferT
131: (0)                class TestCreateValuesNestedSingle(CreateValues):
132: (4)                    """Check the creation of heterogeneous arrays (nested, single row)"""
133: (4)                    _descr = Ndescr
134: (4)                    multiple_rows = 0

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

135: (4)
136: (0)
137: (4)
138: (4)
139: (4)
140: (4)
141: (0)
142: (4)
143: (4)
144: (8)
145: (8)
146: (12)
147: (12)
148: (12)
149: (12)
150: (8)
151: (12)
152: (12)
153: (45)
154: (12)
155: (45)
156: (12)
157: (45)
158: (0)
159: (4)
160: (4)
161: (4)
162: (4)
163: (0)
164: (4)
165: (4)
166: (4)
167: (4)
168: (0)
169: (4)
170: (4)
171: (8)
172: (8)
173: (8)
174: (12)
175: (12)
176: (12)
177: (12)
178: (8)
179: (12)
180: (12)
181: (43)
182: (12)
183: (43)
184: (12)
185: (43)
186: (4)
187: (8)
188: (8)
189: (8)
190: (12)
191: (25)
192: (12)
193: (25)
194: (12)
195: (25)
196: (12)
197: (25)
198: (8)
199: (12)
200: (25)
201: (32)
202: (32)
203: (12)

    _buffer = NbufferT[0]
class TestCreateValuesNestedMultiple(CreateValues):
    """Check the creation of heterogeneous arrays (nested, multiple rows)"""
    _descr = Ndescr
    multiple_rows = 1
    _buffer = NbufferT
class ReadValuesPlain:
    """Check the reading of values in heterogeneous arrays (plain)"""
    def test_access_fields(self):
        h = np.array(self._buffer, dtype=self._descr)
        if not self.multiple_rows:
            assert_(h.shape == ())
            assert_equal(h['x'], np.array(self._buffer[0], dtype='i4'))
            assert_equal(h['y'], np.array(self._buffer[1], dtype='f8'))
            assert_equal(h['z'], np.array(self._buffer[2], dtype='u1'))
        else:
            assert_(len(h) == 2)
            assert_equal(h['x'], np.array([self._buffer[0][0],
   self._buffer[1][0]]), dtype='i4')
            assert_equal(h['y'], np.array([self._buffer[0][1],
   self._buffer[1][1]]), dtype='f8')
            assert_equal(h['z'], np.array([self._buffer[0][2],
   self._buffer[1][2]]), dtype='u1'))
class TestReadValuesPlainSingle(ReadValuesPlain):
    """Check the creation of heterogeneous arrays (plain, single row)"""
    _descr = Pdescr
    multiple_rows = 0
    _buffer = PbufferT[0]
class TestReadValuesPlainMultiple(ReadValuesPlain):
    """Check the values of heterogeneous arrays (plain, multiple rows)"""
    _descr = Pdescr
    multiple_rows = 1
    _buffer = PbufferT
class ReadValuesNested:
    """Check the reading of values in heterogeneous arrays (nested)"""
    def test_access_top_fields(self):
        """Check reading the top fields of a nested array"""
        h = np.array(self._buffer, dtype=self._descr)
        if not self.multiple_rows:
            assert_(h.shape == ())
            assert_equal(h['x'], np.array(self._buffer[0], dtype='i4'))
            assert_equal(h['y'], np.array(self._buffer[4], dtype='f8'))
            assert_equal(h['z'], np.array(self._buffer[5], dtype='u1'))
        else:
            assert_(len(h) == 2)
            assert_equal(h['x'], np.array([self._buffer[0][0],
   self._buffer[1][0]]), dtype='i4')
            assert_equal(h['y'], np.array([self._buffer[0][4],
   self._buffer[1][4]]), dtype='f8')
            assert_equal(h['z'], np.array([self._buffer[0][5],
   self._buffer[1][5]]), dtype='u1'))
    def test_nested1_acessors(self):
        """Check reading the nested fields of a nested array (1st level)"""
        h = np.array(self._buffer, dtype=self._descr)
        if not self.multiple_rows:
            assert_equal(h['Info']['value'],
                         np.array(self._buffer[1][0], dtype='c16'))
            assert_equal(h['Info']['y2'],
                         np.array(self._buffer[1][1], dtype='f8'))
            assert_equal(h['info']['Name'],
                         np.array(self._buffer[3][0], dtype='U2'))
            assert_equal(h['info']['Value'],
                         np.array(self._buffer[3][1], dtype='c16'))
        else:
            assert_equal(h['Info']['value'],
                         np.array([self._buffer[0][1][0],
                                   self._buffer[1][1][0]]),
                         dtype='c16'))
            assert_equal(h['Info']['y2'],
                         np.array([self._buffer[0][1][1],
                                   self._buffer[1][1][1]]),
                         dtype='c16'))

```

```

204: (25) np.array([self._buffer[0][1][1],
205: (32)           self._buffer[1][1][1]],
206: (32)           dtype='f8'))
207: (12) assert_equal(h['info']['Name'],
208: (25)           np.array([self._buffer[0][3][0],
209: (32)             self._buffer[1][3][0]],
210: (31)             dtype='U2'))
211: (12) assert_equal(h['info']['Value'],
212: (25)           np.array([self._buffer[0][3][1],
213: (32)             self._buffer[1][3][1]],
214: (31)             dtype='c16'))
215: (4) def test_nested2_acessors(self):
216: (8)     """Check reading the nested fields of a nested array (2nd level)"""
217: (8)     h = np.array(self._buffer, dtype=self._descr)
218: (8)     if not self.multiple_rows:
219: (12)         assert_equal(h['Info']['Info2']['value'],
220: (25)             np.array(self._buffer[1][2][1], dtype='c16'))
221: (12)         assert_equal(h['Info']['Info2']['z3'],
222: (25)             np.array(self._buffer[1][2][3], dtype='u4'))
223: (8)     else:
224: (12)         assert_equal(h['Info']['Info2']['value'],
225: (25)             np.array([self._buffer[0][1][2][1],
226: (32)               self._buffer[1][1][2][1]],
227: (31)               dtype='c16'))
228: (12)         assert_equal(h['Info']['Info2']['z3'],
229: (25)             np.array([self._buffer[0][1][2][3],
230: (32)               self._buffer[1][1][2][3]],
231: (31)               dtype='u4'))
232: (4) def test_nested1_descriptor(self):
233: (8)     """Check access nested descriptors of a nested array (1st level)"""
234: (8)     h = np.array(self._buffer, dtype=self._descr)
235: (8)     assert_(h.dtype['Info']['value'].name == 'complex128')
236: (8)     assert_(h.dtype['Info']['y2'].name == 'float64')
237: (8)     assert_(h.dtype['info']['Name'].name == 'str256')
238: (8)     assert_(h.dtype['info']['Value'].name == 'complex128')
239: (4) def test_nested2_descriptor(self):
240: (8)     """Check access nested descriptors of a nested array (2nd level)"""
241: (8)     h = np.array(self._buffer, dtype=self._descr)
242: (8)     assert_(h.dtype['Info']['Info2']['value'].name == 'void256')
243: (8)     assert_(h.dtype['Info']['Info2']['z3'].name == 'void64')
244: (0) class TestReadValuesNestedSingle(ReadValuesNested):
245: (4)     """Check the values of heterogeneous arrays (nested, single row)"""
246: (4)     _descr = Ndescr
247: (4)     multiple_rows = False
248: (4)     _buffer = NbufferT[0]
249: (0) class TestReadValuesNestedMultiple(ReadValuesNested):
250: (4)     """Check the values of heterogeneous arrays (nested, multiple rows)"""
251: (4)     _descr = Ndescr
252: (4)     multiple_rows = True
253: (4)     _buffer = NbufferT
254: (0) class TestEmptyField:
255: (4)     def test_assign(self):
256: (8)         a = np.arange(10, dtype=np.float32)
257: (8)         a.dtype = [("int", "<0i4"), ("float", "<2f4")]
258: (8)         assert_(a['int'].shape == (5, 0))
259: (8)         assert_(a['float'].shape == (5, 2))
260: (0) class TestCommonType:
261: (4)     def test_scalar_loses1(self):
262: (8)         with pytest.warns(DeprecationWarning, match="np.find_common_type"):
263: (12)             res = np.find_common_type(['f4', 'f4', 'i2'], ['f8'])
264: (8)             assert_(res == 'f4')
265: (4)     def test_scalar_loses2(self):
266: (8)         with pytest.warns(DeprecationWarning, match="np.find_common_type"):
267: (12)             res = np.find_common_type(['f4', 'f4'], ['i8'])
268: (8)             assert_(res == 'f4')
269: (4)     def test_scalar_wins(self):
270: (8)         with pytest.warns(DeprecationWarning, match="np.find_common_type"):
271: (12)             res = np.find_common_type(['f4', 'f4', 'i2'], ['c8'])
272: (8)             assert_(res == 'c8')

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

273: (4)
274: (8)
275: (12)
276: (8)
277: (4)
278: (8)
279: (12)
280: (8)
281: (0)
282: (4)
283: (8)
284: (4)
285: (8)
286: (4)
287: (8)
288: (4)
289: (8)
290: (8)
291: (0)
292: (4)
293: (4)
294: (8)
295: (8)
296: (4)
297: (8)
298: (12)
299: (16)
300: (4)
301: (8)
302: (12)
303: (12)
304: (4)
305: (8)
306: (12)
307: (12)
308: (4)
309: (8)
310: (12)
311: (12)
312: (4)
313: (8)
314: (8)
315: (8)
316: (8)
317: (8)
318: (8)
319: (8)
320: (8)
321: (8)
322: (8)
323: (8)
324: (8)
325: (8)
326: (8)
327: (8)
328: (8)
329: (8)
330: (8)
331: (8)
332: (8)
333: (0)
334: (4)
335: (8)
336: (8)
337: (4)
338: (8)
339: (8)
340: (12)
341: (0)

    def test_scalar_wins2(self):
        with pytest.warns(DeprecationWarning, match="np.find_common_type"):
            res = np.find_common_type(['u4', 'i4', 'i4'], ['f4'])
        assert_(res == 'f8')
    def test_scalar_wins3(self): # doesn't go up to 'f16' on purpose
        with pytest.warns(DeprecationWarning, match="np.find_common_type"):
            res = np.find_common_type(['u8', 'i8', 'i8'], ['f8'])
        assert_(res == 'f8')

    class TestMultipleFields:
        def setup_method(self):
            self.ary = np.array([(1, 2, 3, 4), (5, 6, 7, 8)], dtype='i4,f4,i2,c8')
        def _bad_call(self):
            return self.ary['f0', 'f1']
        def test_no_tuple(self):
            assert_raises(IndexError, self._bad_call)
        def test_return(self):
            res = self.ary[['f0', 'f2']].tolist()
            assert_(res == [(1, 3), (5, 7)])

    class TestIsSubDType:
        wrappers = [np.dtype, lambda x: x]
        def test_both_abstract(self):
            assert_(np.issubdtype(np.floating, np.inexact))
            assert_(not np.issubdtype(np.inexact, np.floating))
        def test_same(self):
            for cls in (np.float32, np.int32):
                for w1, w2 in itertools.product(self.wrappers, repeat=2):
                    assert_(np.issubdtype(w1(cls), w2(cls)))
        def test_subclass(self):
            for w in self.wrappers:
                assert_(np.issubdtype(w(np.float32), np.floating))
                assert_(np.issubdtype(w(np.float64), np.floating))
        def test_subclass_backwards(self):
            for w in self.wrappers:
                assert_(not np.issubdtype(np.floating, w(np.float32)))
                assert_(not np.issubdtype(np.floating, w(np.float64)))
        def test_sibling_class(self):
            for w1, w2 in itertools.product(self.wrappers, repeat=2):
                assert_(not np.issubdtype(w1(np.float32), w2(np.float64)))
                assert_(not np.issubdtype(w1(np.float64), w2(np.float32)))
        def test_nondtype_nonscalartype(self):
            assert not np.issubdtype(np.float32, 'float64')
            assert not np.issubdtype(np.float32, 'f8')
            assert not np.issubdtype(np.int32, str)
            assert not np.issubdtype(np.int32, 'int64')
            assert not np.issubdtype(np.str_, 'void')
            assert not np.issubdtype(np.int8, int) # np.int8 is never np.int_
            assert not np.issubdtype(np.float32, float)
            assert not np.issubdtype(np.complex64, complex)
            assert not np.issubdtype(np.float32, "float")
            assert not np.issubdtype(np.float64, "f")
            assert np.issubdtype(np.float64, 'float64')
            assert np.issubdtype(np.float64, 'f8')
            assert np.issubdtype(np.str_, str)
            assert np.issubdtype(np.int64, 'int64')
            assert np.issubdtype(np void, 'void')
            assert np.issubdtype(np.int8, np.integer)
            assert np.issubdtype(np.float32, np.floating)
            assert np.issubdtype(np.complex64, np.complexfloating)
            assert np.issubdtype(np.float64, "float")
            assert np.issubdtype(np.float32, "f")

        class TestSctypeDict:
            def test_longdouble(self):
                assert_(np.sctypeDict['f8'] is not np.longdouble)
                assert_(np.sctypeDict['c16'] is not np.clongdouble)
            def test_ulong(self):
                assert_(np.sctypeDict['ulong'] is np.uint)
                with pytest.warns(FutureWarning):
                    assert not hasattr(np, 'ulong')

    class TestBitName:

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

342: (4)
343: (8)
344: (0)
345: (4)
np.longlong])
346: (4)
347: (8)
348: (4)
np.ulonglong])
349: (4)
350: (8)
351: (4)
np.longdouble])
352: (4)
353: (8)
354: (4)
355: (4)
356: (8)
357: (4)
358: (35)
359: (4)
360: (8)
361: (0)
362: (4)
363: (8)
364: (8)
365: (8)
366: (8)
367: (4)
368: (8)
369: (8)
370: (8)
371: (4)
372: (8)
373: (8)
374: (8)
375: (4)
376: (8)
377: (4)
378: (8)
379: (4)
380: (8)
381: (0)
382: (4)
383: (4)
384: (4)
385: (4)
386: (4)
387: (4)
388: (4)
389: (4)
390: (0)
391: (4)
392: (4)
393: (0)
394: (20)
PYTHONOPTIMIZE/Py_OptimizeFlag > 1")
395: (0)
396: (19)
397: (0)
398: (4)
399: (8)
400: (12)
401: (8)
402: (12)
403: (0)
404: (4)
405: (8)
406: (8)

def test_abstract(self):
    assert_raises(ValueError, np.core.numerictypes.bitname, np.floating)
class TestMaximumSctype:
    @pytest.mark.parametrize('t', [np.byte, np.short, np.intc, np.int_, np.longlong])
    def test_int(self, t):
        assert_equal(np.maximum_sctype(t), np.sctypes['int'][-1])
    @pytest.mark.parametrize('t', [np.ubyte, np ushort, np.uintc, np.uint, np.ulonglong])
    def test_uint(self, t):
        assert_equal(np.maximum_sctype(t), np.sctypes['uint'][-1])
    @pytest.mark.parametrize('t', [np.half, np.single, np.double, np.longdouble])
    def test_float(self, t):
        assert_equal(np.maximum_sctype(t), np.sctypes['float'][-1])
    @pytest.mark.parametrize('t', [np.csingle, np.cdouble, np.clongdouble])
    def test_complex(self, t):
        assert_equal(np.maximum_sctype(t), np.sctypes['complex'][-1])
    @pytest.mark.parametrize('t', [np.bool_, np.object_, np.str_, np.bytes_, np void])
    def test_other(self, t):
        assert_equal(np.maximum_sctype(t), t)
class Test_sctype2char:
    def test_scalar_type(self):
        assert_equal(np.sctype2char(np.double), 'd')
        assert_equal(np.sctype2char(np.int_), 'l')
        assert_equal(np.sctype2char(np.str_), 'U')
        assert_equal(np.sctype2char(np.bytes_), 'S')
    def test_other_type(self):
        assert_equal(np.sctype2char(float), 'd')
        assert_equal(np.sctype2char(list), 'O')
        assert_equal(np.sctype2char(np.ndarray), 'O')
    def test_third_party_scalar_type(self):
        from numpy.core._rational_tests import rational
        assert_raises(KeyError, np.sctype2char, rational)
        assert_raises(KeyError, np.sctype2char, rational(1))
    def test_array_instance(self):
        assert_equal(np.sctype2char(np.array([1.0, 2.0])), 'd')
    def test_abstract_type(self):
        assert_raises(KeyError, np.sctype2char, np.floating)
    def test_non_type(self):
        assert_raises(ValueError, np.sctype2char, 1)
    @pytest.mark.parametrize("rep, expected", [
        (np.int32, True),
        (list, False),
        (1.1, False),
        (str, True),
        (np.dtype(np.float64), True),
        (np.dtype((np.int16, (3, 4))), True),
        (np.dtype([('a', np.int8)]), True),
        ])
    def test_issctype(rep, expected):
        actual = np.issctype(rep)
        assert_equal(actual, expected)
    @pytest.mark.skipif(sys.flags.optimize > 1,
                       reason="no docstrings present to inspect when
PYTHONOPTIMIZE/Py_OptimizeFlag > 1")
    @pytest.mark.xfail(IS_PYPY,
                       reason="PyPy cannot modify tp_doc after PyType_Ready")
class TestDocStrings:
    def test_platform_dependent_aliases(self):
        if np.int64 is np.int_:
            assert_('int64' in np.int_.__doc__)
        elif np.int64 is np.longlong:
            assert_('int64' in np.longlong.__doc__)
class TestScalarTypeNames:
    numeric_types = [
        np.byte, np.short, np.intc, np.int_, np.longlong,
        np.ubyte, np ushort, np.uintc, np.uint, np.ulonglong,
```

```

407: (8)             np.half, np.single, np.double, np.longdouble,
408: (8)             np.csingle, np.cdouble, np.clongdouble,
409: (4)
410: (4)         def test_names_are_unique(self):
411: (8)             assert len(set(self.numeric_types)) == len(self.numeric_types)
412: (8)             names = [t.__name__ for t in self.numeric_types]
413: (8)             assert len(set(names)) == len(names)
414: (4)         @pytest.mark.parametrize('t', numeric_types)
415: (4)         def test_names_reflect_attributes(self, t):
416: (8)             """ Test that names correspond to where the type is under ``np.`` """
417: (8)             assert getattr(np, t.__name__) is t
418: (4)         @pytest.mark.parametrize('t', numeric_types)
419: (4)         def test_names_are_understood_by_dtype(self, t):
420: (8)             """ Test the dtype constructor maps names back to the type """
421: (8)             assert np.dtype(t.__name__).type is t

```

-----

File 113 - test\_numpy\_2\_0\_compat.py:

```

1: (0)             from os import path
2: (0)             import pickle
3: (0)             import numpy as np
4: (0)             class TestNumPy2Compatibility:
5: (4)                 data_dir = path.join(path.dirname(__file__), "data")
6: (4)                 filename = path.join(data_dir, "numpy_2_0_array.pkl")
7: (4)                 def test_importable_core_stubs(self):
8: (8)                     """
9: (8)                         Checks if stubs for `numpy._core` are importable.
10: (8)                     """
11: (8)                     from numpy._core.multiarray import _reconstruct
12: (8)                     from numpy._core.umath import cos
13: (8)                     from numpy._core._multiarray_umath import exp
14: (8)                     from numpy._core._internal import ndarray
15: (8)                     from numpy._core._dtype import _construction_repr
16: (8)                     from numpy._core._dtype_ctypes import dtype_from_ctypes_type
17: (4)                 def test_unpickle_numpy_2_0_file(self):
18: (8)                     """
19: (8)                         Checks that NumPy 1.26 and pickle is able to load pickles
20: (8)                         created with NumPy 2.0 without errors/warnings.
21: (8)                     """
22: (8)                     with open(self.filename, mode="rb") as file:
23: (12)                         content = file.read()
24: (8)                         assert b"numpy._core.multiarray" in content
25: (8)                         arr = pickle.loads(content, encoding="latin1")
26: (8)                         assert isinstance(arr, np.ndarray)
27: (8)                         assert arr.shape == (73,) and arr.dtype == np.float64
28: (4)                 def test_numpy_load_numpy_2_0_file(self):
29: (8)                     """
30: (8)                         Checks that `numpy.load` for NumPy 1.26 is able to load pickles
31: (8)                         created with NumPy 2.0 without errors/warnings.
32: (8)                     """
33: (8)                     arr = np.load(self.filename, encoding="latin1", allow_pickle=True)
34: (8)                     assert isinstance(arr, np.ndarray)
35: (8)                     assert arr.shape == (73,) and arr.dtype == np.float64

```

-----

File 114 - test\_print.py:

```

1: (0)             import sys
2: (0)             import pytest
3: (0)             import numpy as np
4: (0)             from numpy.testing import assert_, assert_equal, IS_MSL
5: (0)             from numpy.core.tests._locales import CommaDecimalPointLocale
6: (0)             from io import StringIO
7: (0)             _REF = {np.inf: 'inf', -np.inf: '-inf', np.nan: 'nan'}
8: (0)             @pytest.mark.parametrize('tp', [np.float32, np.double, np.longdouble])
9: (0)             def test_float_types(tp):

```

```

10: (4)      """ Check formatting.
11: (8)      This is only for the str function, and only for simple types.
12: (8)      The precision of np.float32 and np.longdouble aren't the same as the
13: (8)      python float precision.
14: (4)
15: (4)      """
16: (8)      for x in [0, 1, -1, 1e20]:
17: (21)          assert_equal(str(tp(x)), str(float(x)),
18: (4)                  err_msg='Failed str formatting for type %s' % tp)
19: (8)          if tp(1e16).itemsize > 4:
20: (21)              assert_equal(str(tp(1e16)), str(float('1e16')),
21: (4)                  err_msg='Failed str formatting for type %s' % tp)
22: (8)          else:
23: (8)              ref = '1e+16'
24: (21)              assert_equal(str(tp(1e16)), ref,
25: (0)                  err_msg='Failed str formatting for type %s' % tp)
26: (0)      @pytest.mark.parametrize('tp', [np.float32, np.double, np.longdouble])
27: (0)      def test_nan_inf_float(tp):
28: (4)          """ Check formatting of nan & inf.
29: (8)          This is only for the str function, and only for simple types.
30: (8)          The precision of np.float32 and np.longdouble aren't the same as the
31: (8)          python float precision.
32: (4)
33: (8)          for x in [np.inf, -np.inf, np.nan]:
34: (21)              assert_equal(str(tp(x)), _REF[x],
35: (0)                  err_msg='Failed str formatting for type %s' % tp)
36: (0)      @pytest.mark.parametrize('tp', [np.complex64, np.cdouble, np.clongdouble])
37: (0)      def test_complex_types(tp):
38: (4)          """Check formatting of complex types.
39: (8)          This is only for the str function, and only for simple types.
40: (8)          The precision of np.float32 and np.longdouble aren't the same as the
41: (8)          python float precision.
42: (4)
43: (8)          for x in [0, 1, -1, 1e20]:
44: (21)              assert_equal(str(tp(x)), str(complex(x)),
45: (8)                  err_msg='Failed str formatting for type %s' % tp)
46: (21)              assert_equal(str(tp(x*1j)), str(complex(x*1j)),
47: (8)                  err_msg='Failed str formatting for type %s' % tp)
48: (21)              assert_equal(str(tp(x + x*1j)), str(complex(x + x*1j)),
49: (8)                  err_msg='Failed str formatting for type %s' % tp)
50: (4)              if tp(1e16).itemsize > 8:
51: (8)                  assert_equal(str(tp(1e16)), str(complex(1e16)),
52: (21)                      err_msg='Failed str formatting for type %s' % tp)
53: (4)              else:
54: (8)                  ref = '(1e+16+0j)'
55: (21)                  assert_equal(str(tp(1e16)), ref,
56: (0)                      err_msg='Failed str formatting for type %s' % tp)
57: (0)      @pytest.mark.parametrize('dtype', [np.complex64, np.cdouble, np.clongdouble])
58: (0)      def test_complex_inf_nan(dtype):
59: (4)          """Check inf/nan formatting of complex types."""
60: (8)          TESTS = {
61: (8)              complex(np.inf, 0): "(inf+0j)",
62: (8)              complex(0, np.inf): "infj",
63: (8)              complex(-np.inf, 0): "(-inf+0j)",
64: (8)              complex(0, -np.inf): "-infj",
65: (8)              complex(np.inf, 1): "(inf+1j)",
66: (8)              complex(1, np.inf): "(1+infj)",
67: (8)              complex(-np.inf, 1): "(-inf+1j)",
68: (8)              complex(1, -np.inf): "(1-infj)",
69: (8)              complex(np.nan, 0): "(nan+0j)",
70: (8)              complex(0, np.nan): "nanj",
71: (8)              complex(-np.nan, 0): "(nan+0j)",
72: (8)              complex(0, -np.nan): "nanj",
73: (8)              complex(np.nan, 1): "(nan+1j)",
74: (8)              complex(1, np.nan): "(1+nanj)",
75: (8)              complex(-np.nan, 1): "(nan+1j)",
76: (4)              complex(1, -np.nan): "(1+nanj)",
77: (4)          }
78: (8)          for c, s in TESTS.items():
79: (8)              assert_equal(str(dtype(c)), s)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

79: (0)
80: (4)
81: (4)
82: (4)
83: (4)
84: (8)
85: (8)
86: (8)
87: (8)
88: (12)
89: (8)
90: (12)
91: (4)
92: (8)
93: (4)
94: (17)
95: (0)
96: (0)
97: (4)
98: (4)
99: (8)
100: (4)
101: (8)
102: (4)
103: (8)
104: (4)
105: (8)
106: (8)
107: (0)
108: (0)
109: (4)
110: (4)
111: (8)
112: (4)
113: (8)
114: (4)
115: (8)
116: (8)
117: (4)
118: (4)
119: (4)
120: (0)
121: (4)
122: (4)
123: (12)
124: (12)
125: (12)
126: (12)
127: (12)
128: (12)
129: (12)
130: (12)
131: (12)
132: (12)
133: (12)
134: (12)
135: (12)
136: (12)
137: (12)
138: (12)
139: (4)
140: (8)
141: (12)
142: (20)
143: (8)
144: (12)
145: (15)
%
146: (28)

def _test_redirected_print(x, tp, ref=None):
    file = StringIO()
    file_tp = StringIO()
    stdout = sys.stdout
    try:
        sys.stdout = file_tp
        print(tp(x))
        sys.stdout = file
        if ref:
            print(ref)
        else:
            print(x)
    finally:
        sys.stdout = stdout
    assert_equal(file.getvalue(), file_tp.getvalue(),
                err_msg='print failed for type%s' % tp)
@pytest.mark.parametrize('tp', [np.float32, np.double, np.longdouble])
def test_float_type_print(tp):
    """Check formatting when using print """
    for x in [0, 1, -1, 1e20]:
        _test_redirected_print(float(x), tp)
    for x in [np.inf, -np.inf, np.nan]:
        _test_redirected_print(float(x), tp, _REF[x])
    if tp(1e16).itemsize > 4:
        _test_redirected_print(float(1e16), tp)
    else:
        ref = '1e+16'
        _test_redirected_print(float(1e16), tp, ref)
@pytest.mark.parametrize('tp', [np.complex64, np.cdouble, np.clongdouble])
def test_complex_type_print(tp):
    """Check formatting when using print """
    for x in [0, 1, -1, 1e20]:
        _test_redirected_print(complex(x), tp)
    if tp(1e16).itemsize > 8:
        _test_redirected_print(complex(1e16), tp)
    else:
        ref = '(1e+16+0j)'
        _test_redirected_print(complex(1e16), tp, ref)
        _test_redirected_print(complex(np.inf, 1), tp, '(inf+1j)')
        _test_redirected_print(complex(-np.inf, 1), tp, '(-inf+1j)')
        _test_redirected_print(complex(-np.nan, 1), tp, '(nan+1j)')
def test_scalar_format():
    """Test the str.format method with NumPy scalar types"""
    tests = [('{0}', True, np.bool_),
             ('{0}', False, np.bool_),
             ('{0:d}', 130, np.uint8),
             ('{0:d}', 50000, np.uint16),
             ('{0:d}', 3000000000, np.uint32),
             ('{0:d}', 15000000000000000000, np.uint64),
             ('{0:d}', -120, np.int8),
             ('{0:d}', -30000, np.int16),
             ('{0:d}', -2000000000, np.int32),
             ('{0:d}', -7000000000000000000, np.int64),
             ('{0:g}', 1.5, np.float16),
             ('{0:g}', 1.5, np.float32),
             ('{0:g}', 1.5, np.float64),
             ('{0:g}', 1.5, np.longdouble),
             ('{0:g}', 1.5+0.5j, np.complex64),
             ('{0:g}', 1.5+0.5j, np.complex128),
             ('{0:g}', 1.5+0.5j, np.clongdouble)]
    for (fmat, val, valtype) in tests:
        try:
            assert_equal(fmat.format(val), fmat.format(valtype(val)),
                        "failed with val %s, type %s" % (val, valtype))
        except ValueError as e:
            assert_(False,
                    "format raised exception (fmt='%', val=%s, type=%s, exc='%'"
                    "(fmat, repr(val), repr(valtype), str(e)))"

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

147: (0)         class TestCommaDecimalPointLocale(CommaDecimalPointLocale):
148: (4)             def test_locale_single(self):
149: (8)                 assert_equal(str(np.float32(1.2)), str(float(1.2)))
150: (4)             def test_locale_double(self):
151: (8)                 assert_equal(str(np.double(1.2)), str(float(1.2)))
152: (4)             @pytest.mark.skipif(IS_MUSL,
153: (24)                             reason="test flaky on musllinux")
154: (4)             def test_locale_longdouble(self):
155: (8)                 assert_equal(str(np.longdouble('1.2')), str(float(1.2)))

```

---

## File 115 - test\_overrides.py:

```

1: (0)         import inspect
2: (0)         import sys
3: (0)         import os
4: (0)         import tempfile
5: (0)         from io import StringIO
6: (0)         from unittest import mock
7: (0)         import numpy as np
8: (0)         from numpy.testing import (
9: (4)             assert_, assert_equal, assert_raises, assert_raises_regex)
10: (0)        from numpy.core.overrides import (
11: (4)            _get_implementing_args, array_function_dispatch,
12: (4)            verify_matching_signatures)
13: (0)        from numpy.compat import pickle
14: (0)        import pytest
15: (0)        def _return_not_implemented(self, *args, **kwargs):
16: (4)            return NotImplemented
17: (0)        @array_function_dispatch(lambda array: (array,))
18: (0)        def dispatched_one_arg(array):
19: (4)            """Docstring."""
20: (4)            return 'original'
21: (0)        @array_function_dispatch(lambda array1, array2: (array1, array2))
22: (0)        def dispatched_two_arg(array1, array2):
23: (4)            """Docstring."""
24: (4)            return 'original'
25: (0)        class TestGetImplementingArgs:
26: (4)            def test_ndarray(self):
27: (8)                array = np.array(1)
28: (8)                args = _get_implementing_args([array])
29: (8)                assert_equal(list(args), [array])
30: (8)                args = _get_implementing_args([array, array])
31: (8)                assert_equal(list(args), [array])
32: (8)                args = _get_implementing_args([array, 1])
33: (8)                assert_equal(list(args), [array])
34: (8)                args = _get_implementing_args([1, array])
35: (8)                assert_equal(list(args), [array])
36: (4)            def test_ndarray_subclasses(self):
37: (8)                class OverrideSub(np.ndarray):
38: (12)                    __array_function__ = _return_not_implemented
39: (8)                class NoOverrideSub(np.ndarray):
40: (12)                    pass
41: (8)                array = np.array(1).view(np.ndarray)
42: (8)                override_sub = np.array(1).view(OverrideSub)
43: (8)                no_override_sub = np.array(1).view(NoOverrideSub)
44: (8)                args = _get_implementing_args([array, override_sub])
45: (8)                assert_equal(list(args), [override_sub, array])
46: (8)                args = _get_implementing_args([array, no_override_sub])
47: (8)                assert_equal(list(args), [no_override_sub, array])
48: (8)                args = _get_implementing_args(
49: (12)                    [override_sub, no_override_sub])
50: (8)                assert_equal(list(args), [override_sub, no_override_sub])
51: (4)            def test_ndarray_and_duck_array(self):
52: (8)                class Other:
53: (12)                    __array_function__ = _return_not_implemented
54: (8)                array = np.array(1)
55: (8)                other = Other()

```

```

56: (8)                                args = _get_implementing_args([other, array])
57: (8)                                assert_equal(list(args), [other, array])
58: (8)                                args = _get_implementing_args([array, other])
59: (8)                                assert_equal(list(args), [array, other])
60: (4) def test_ndarray_subclass_and_duck_array(self):
61: (8) class OverrideSub(np.ndarray):
62: (12)     __array_function__ = _return_not_implemented
63: (8) class Other:
64: (12)     __array_function__ = _return_not_implemented
65: (8) array = np.array(1)
66: (8) subarray = np.array(1).view(OverrideSub)
67: (8) other = Other()
68: (8) assert_equal(_get_implementing_args([array, subarray, other]),
69: (21)                 [subarray, array, other])
70: (8) assert_equal(_get_implementing_args([array, other, subarray]),
71: (21)                 [subarray, array, other])
72: (4) def test_many_duck_arrays(self):
73: (8) class A:
74: (12)     __array_function__ = _return_not_implemented
75: (8) class B(A):
76: (12)     __array_function__ = _return_not_implemented
77: (8) class C(A):
78: (12)     __array_function__ = _return_not_implemented
79: (8) class D:
80: (12)     __array_function__ = _return_not_implemented
81: (8) a = A()
82: (8) b = B()
83: (8) c = C()
84: (8) d = D()
85: (8) assert_equal(_get_implementing_args([1]), [])
86: (8) assert_equal(_get_implementing_args([a]), [a])
87: (8) assert_equal(_get_implementing_args([a, 1]), [a])
88: (8) assert_equal(_get_implementing_args([a, a, a]), [a])
89: (8) assert_equal(_get_implementing_args([a, d, a]), [a, d])
90: (8) assert_equal(_get_implementing_args([a, b]), [b, a])
91: (8) assert_equal(_get_implementing_args([b, a]), [b, a])
92: (8) assert_equal(_get_implementing_args([a, b, c]), [b, c, a])
93: (8) assert_equal(_get_implementing_args([a, c, b]), [c, b, a])
94: (4) def test_too_many_duck_arrays(self):
95: (8)     namespace = dict(__array_function__=_return_not_implemented)
96: (8)     types = [type('A' + str(i)), (object,), namespace) for i in range(33)]
97: (8)     relevant_args = [t() for t in types]
98: (8)     actual = _get_implementing_args(relevant_args[:32])
99: (8)     assert_equal(actual, relevant_args[:32])
100: (8)     with assert_raises_regex(TypeError, 'distinct argument types'):
101: (12)         _get_implementing_args(relevant_args)
102: (0) class TestNDArrayFunction:
103: (4)     def test_method(self):
104: (8)         class Other:
105: (12)             __array_function__ = _return_not_implemented
106: (8)         class NoOverrideSub(np.ndarray):
107: (12)             pass
108: (8)         class OverrideSub(np.ndarray):
109: (12)             __array_function__ = _return_not_implemented
110: (8)         array = np.array([1])
111: (8)         other = Other()
112: (8)         no_override_sub = array.view(NoOverrideSub)
113: (8)         override_sub = array.view(OverrideSub)
114: (8)         result = array.__array_function__(func=dispatched_two_arg,
115: (42)                           types=(np.ndarray,),
116: (42)                           args=(array, 1.), kwargs={})
117: (8)         assert_equal(result, 'original')
118: (8)         result = array.__array_function__(func=dispatched_two_arg,
119: (42)                           types=(np.ndarray, Other),
120: (42)                           args=(array, other), kwargs={})
121: (8)         assert_(result is NotImplemented)
122: (8)         result = array.__array_function__(func=dispatched_two_arg,
123: (42)                           types=(np.ndarray, NoOverrideSub),
124: (42)                           args=(array, no_override_sub)),

```

```

125: (42)                                     kwargs={}
126: (8)          assert_equal(result, 'original')
127: (8)          result = array.__array_function__(func=dispatched_two_arg,
128: (42)   types=(np.ndarray, OverrideSub),
129: (42)   args=(array, override_sub),
130: (42)   kwargs={})
131: (8)          assert_equal(result, 'original')
132: (8)          with assert_raises_regex(TypeError, 'no implementation found'):
133: (12)              np.concatenate((array, other))
134: (8)          expected = np.concatenate((array, array))
135: (8)          result = np.concatenate((array, no_override_sub))
136: (8)          assert_equal(result, expected.view(NoOverrideSub))
137: (8)          result = np.concatenate((array, override_sub))
138: (8)          assert_equal(result, expected.view(OverrideSub))
139: (4)          def test_no_wrapper(self):
140: (8)              array = np.array(1)
141: (8)              func = lambda x: x
142: (8)              with assert_raises_regex(AttributeError, '_implementation'):
143: (12)                  array.__array_function__(func=func, types=(np.ndarray,),
144: (37)                      args=(array,), kwargs={})
145: (0)          class TestArrayFunctionDispatch:
146: (4)              def test_pickle(self):
147: (8)                  for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
148: (12)                      roundtripped = pickle.loads(
149: (20)                          pickle.dumps(dispatched_one_arg, protocol=proto))
150: (12)                      assert_(roundtripped is dispatched_one_arg)
151: (4)          def test_name_and_docstring(self):
152: (8)              assert_equal(dispatched_one_arg.__name__, 'dispatched_one_arg')
153: (8)              if sys.flags.optimize < 2:
154: (12)                  assert_equal(dispatched_one_arg.__doc__, 'Docstring.')
155: (4)          def test_interface(self):
156: (8)              class MyArray:
157: (12)                  def __array_function__(self, func, types, args, kwargs):
158: (16)                      return (self, func, types, args, kwargs)
159: (8)              original = MyArray()
160: (8)              (obj, func, types, args, kwargs) = dispatched_one_arg(original)
161: (8)              assert_(obj is original)
162: (8)              assert_(func is dispatched_one_arg)
163: (8)              assert_equal(set(types), {MyArray})
164: (8)              assert_(args == (original,))
165: (8)              assert_equal(kwargs, {})
166: (4)          def test_not_implemented(self):
167: (8)              class MyArray:
168: (12)                  def __array_function__(self, func, types, args, kwargs):
169: (16)                      return NotImplemented
170: (8)              array = MyArray()
171: (8)              with assert_raises_regex(TypeError, 'no implementation found'):
172: (12)                  dispatched_one_arg(array)
173: (4)          def test_where_dispatch(self):
174: (8)              class DuckArray:
175: (12)                  def __array_function__(self, ufunc, method, *inputs, **kwargs):
176: (16)                      return "overridden"
177: (8)              array = np.array(1)
178: (8)              duck_array = DuckArray()
179: (8)              result = np.std(array, where=duck_array)
180: (8)              assert_equal(result, "overridden")
181: (0)          class TestVerifyMatchingSignatures:
182: (4)              def test_verify_matching_signatures(self):
183: (8)                  verify_matching_signatures(lambda x: 0, lambda x: 0)
184: (8)                  verify_matching_signatures(lambda x=None: 0, lambda x=None: 0)
185: (8)                  verify_matching_signatures(lambda x=1: 0, lambda x=None: 0)
186: (8)                  with assert_raises(RuntimeError):
187: (12)                      verify_matching_signatures(lambda a: 0, lambda b: 0)
188: (8)                  with assert_raises(RuntimeError):
189: (12)                      verify_matching_signatures(lambda x: 0, lambda x=None: 0)
190: (8)                  with assert_raises(RuntimeError):
191: (12)                      verify_matching_signatures(lambda x=None: 0, lambda y=None: 0)
192: (8)                  with assert_raises(RuntimeError):
193: (12)                      verify_matching_signatures(lambda x=1: 0, lambda y=1: 0)

```

```

194: (4) def test_array_function_dispatch(self):
195: (8)     with assert_raises(RuntimeError):
196: (12)         @array_function_dispatch(lambda x: (x,))
197: (12)             def f(y):
198: (16)                 pass
199: (8)             @array_function_dispatch(lambda x: (x,), verify=False)
200: (8)                 def f(y):
201: (12)                     pass
202: (0)     def _new_duck_type_and_implements():
203: (4)         """Create a duck array type and implements functions."""
204: (4)         HANDLED_FUNCTIONS = {}
205: (4)         class MyArray:
206: (8)             def __array_function__(self, func, types, args, kwargs):
207: (12)                 if func not in HANDLED_FUNCTIONS:
208: (16)                     return NotImplemented
209: (12)                 if not all(issubclass(t, MyArray) for t in types):
210: (16)                     return NotImplemented
211: (12)                 return HANDLED_FUNCTIONS[func](*args, **kwargs)
212: (4)         def implements(numpy_function):
213: (8)             """Register an __array_function__ implementations."""
214: (8)             def decorator(func):
215: (12)                 HANDLED_FUNCTIONS[numpy_function] = func
216: (12)                 return func
217: (8)             return decorator
218: (4)             return (MyArray, implements)
219: (0)         class TestArrayFunctionImplementation:
220: (4)             def test_one_arg(self):
221: (8)                 MyArray, implements = _new_duck_type_and_implements()
222: (8)                 @implements(dispatched_one_arg)
223: (8)                 def _(array):
224: (12)                     return 'myarray'
225: (8)                     assert_equal(dispatched_one_arg(1), 'original')
226: (8)                     assert_equal(dispatched_one_arg(MyArray()), 'myarray')
227: (4)             def test_optional_args(self):
228: (8)                 MyArray, implements = _new_duck_type_and_implements()
229: (8)                 @array_function_dispatch(lambda array, option=None: (array,))
230: (8)                 def func_with_option(array, option='default'):
231: (12)                     return option
232: (8)                     @implements(func_with_option)
233: (8)                     def my_array_func_with_option(array, new_option='myarray'):
234: (12)                         return new_option
235: (8)                         assert_equal(func_with_option(1), 'default')
236: (8)                         assert_equal(func_with_option(1, option='extra'), 'extra')
237: (8)                         assert_equal(func_with_option(MyArray()), 'myarray')
238: (8)                         with assert_raises(TypeError):
239: (12)                             func_with_option(MyArray(), option='extra')
240: (8)                             result = my_array_func_with_option(MyArray(), new_option='yes')
241: (8)                             assert_equal(result, 'yes')
242: (8)                             with assert_raises(TypeError):
243: (12)                                 func_with_option(MyArray(), new_option='no')
244: (4)             def test_not_implemented(self):
245: (8)                 MyArray, implements = _new_duck_type_and_implements()
246: (8)                 @array_function_dispatch(lambda array: (array,), module='my')
247: (8)                 def func(array):
248: (12)                     return array
249: (8)                     array = np.array(1)
250: (8)                     assert_(func(array) is array)
251: (8)                     assert_equal(func.__module__, 'my')
252: (8)                     with assert_raises_regex(
253: (16)                         TypeError, "no implementation found for 'my.func'"):
254: (12)                         func(MyArray())
255: (4)                     @pytest.mark.parametrize("name", ["concatenate", "mean", "asarray"])
256: (4)                     def test_signature_error_message_simple(self, name):
257: (8)                         func = getattr(np, name)
258: (8)                         try:
259: (12)                             func()
260: (8)                         except TypeError as e:
261: (12)                             exc = e
262: (8)                             assert exc.args[0].startswith(f"{name}()")
```

```

263: (4)
264: (8)
265: (12)
266: (8)
267: (8)
268: (12)
269: (8)
270: (12)
271: (8)
272: (12)
273: (8)
274: (12)
275: (12)
276: (8)
277: (12)
278: (16)
279: (28)
280: (12)
281: (4)
282: (4)
283: (8)
284: (8)
285: (12)
286: (8)
287: (8)
288: (12)
289: (8)
290: (12)
291: (12)
292: (8)
293: (12)
294: (4)
295: (8)
296: (8)
297: (8)
298: (8)
299: (8)
300: (4)
301: (12)
302: (12)
303: (12)
304: (8)
305: (4)
306: (8)
307: (12)
308: (4)
309: (8)
310: (12)
311: (8)
312: (8)
313: (12)
314: (8)
315: (12)
316: (4)
317: (8)
318: (8)
319: (12)
320: (16)
321: (20)
322: (12)
323: (8)
324: (12)
325: (8)
326: (8)
327: (12)
328: (8)
329: (12)
330: (0)
331: (4)

def test_signature_error_message(self):
    def _dispatcher():
        return ()
    @array_function_dispatch(_dispatcher)
    def func():
        pass
    try:
        func._implementation(bad_arg=3)
    except TypeError as e:
        expected_exception = e
    try:
        func(bad_arg=3)
        raise AssertionError("must fail")
    except TypeError as exc:
        if exc.args[0].startswith("_dispatcher"):
            pytest.skip("Python version is not using __qualname__ for "
                        "TypeError formatting.")
        assert exc.args == expected_exception.args
@pytest.mark.parametrize("value", [234, "this func is not replaced"])
def test_dispatcher_error(self, value):
    error = TypeError(value)
    def dispatcher():
        raise error
    @array_function_dispatch(dispatcher)
    def func():
        return 3
    try:
        func()
        raise AssertionError("must fail")
    except TypeError as exc:
        assert exc is error # unmodified exception
def test_properties(self):
    func = dispatched_two_arg
    assert str(func) == str(func._implementation)
    repr_no_id = repr(func).split("at ")[0]
    repr_no_id_impl = repr(func._implementation).split("at ")[0]
    assert repr_no_id == repr_no_id_impl
    @pytest.mark.parametrize("func", [
        lambda x, y: 0, # no like argument
        lambda like=None: 0, # not keyword only
        lambda *, like=None, a=3: 0, # not last (not that it matters)
    ])
def test_bad_like_sig(self, func):
    with pytest.raises(RuntimeError):
        array_function_dispatch()(func)
def test_bad_like_passing(self):
    def func(*, like=None):
        pass
    func_with_like = array_function_dispatch()(func)
    with pytest.raises(TypeError):
        func_with_like()
    with pytest.raises(TypeError):
        func_with_like(like=234)
def test_too_many_args(self):
    objs = []
    for i in range(40):
        class MyArr:
            def __array_function__(self, *args, **kwargs):
                return NotImplemented
        objs.append(MyArr())
    def _dispatch(*args):
        return args
    @array_function_dispatch(_dispatch)
    def func(*args):
        pass
        with pytest.raises(TypeError, match="maximum number"):
            func(*objs)
class TestNDArrayMethods:
    def test_repr(self):

```

```

332: (8)
333: (12)
334: (16)
335: (8)
336: (8)
337: (8)
338: (0)
339: (4)
340: (8)
341: (8)
342: (8)
343: (8)
344: (4)
345: (8)
346: (8)
347: (4)
348: (8)
349: (8)
350: (8)
351: (12)
352: (8)
353: (4)
354: (8)
355: (12)
356: (16)
357: (12)
358: (16)
359: (12)
360: (16)
361: (8)
362: (8)
363: (8)
364: (8)
365: (8)
366: (12)
367: (8)
368: (4)
369: (8)
370: (12)
371: (16)
372: (12)
373: (16)
374: (8)
375: (8)
376: (0)
377: (4)
378: (8)
379: (12)
380: (16)
381: (12)
382: (16)
383: (16)
384: (20)
385: (16)
386: (20)
387: (16)
388: (8)
389: (8)
390: (12)
391: (16)
392: (8)
393: (4)
394: (8)
395: (12)
396: (12)
397: (16)
398: (12)
399: (8)
400: (4)

        class MyArray(np.ndarray):
            def __array_function__(*args, **kwargs):
                return NotImplemented
            array = np.array(1).view(MyArray)
            assert_equal(repr(array), 'MyArray(1)')
            assert_equal(str(array), '1')
        class TestNumPyFunctions:
            def test_set_module(self):
                assert_equal(np.sum.__module__, 'numpy')
                assert_equal(np.char.equal.__module__, 'numpy.char')
                assert_equal(np.fft.fft.__module__, 'numpy.fft')
                assert_equal(np.linalg.solve.__module__, 'numpy.linalg')
            def test_inspect_sum(self):
                signature = inspect.signature(np.sum)
                assert_('axis' in signature.parameters)
            def test_override_sum(self):
                MyArray, implements = _new_duck_type_and_implements()
                @implements(np.sum)
                def _(array):
                    return 'yes'
                assert_equal(np.sum(MyArray()), 'yes')
            def test_sum_on_mock_array(self):
                class ArrayProxy:
                    def __init__(self, value):
                        self.value = value
                    def __array_function__(self, *args, **kwargs):
                        return self.value.__array_function__(*args, **kwargs)
                    def __array__(self, *args, **kwargs):
                        return self.value.__array__(*args, **kwargs)
                proxy = ArrayProxy(mock.Mock(spec=ArrayProxy))
                proxy.value.__array_function__.return_value = 1
                result = np.sum(proxy)
                assert_equal(result, 1)
                proxy.value.__array_function__.assert_called_once_with(
                    np.sum, (ArrayProxy,), (proxy,), {})
                proxy.value.__array__.assert_not_called()
            def test_sum_forwardingImplementation(self):
                class MyArray(np.ndarray):
                    def sum(self, axis, out):
                        return 'summed'
                    def __array_function__(self, func, types, args, kwargs):
                        return super().__array_function__(func, types, args, kwargs)
                array = np.array(1).view(MyArray)
                assert_equal(np.sum(array), 'summed')
        class TestArrayLike:
            def setup_method(self):
                class MyArray():
                    def __init__(self, function=None):
                        self.function = function
                    def __array_function__(self, func, types, args, kwargs):
                        assert func is getattr(np, func.__name__)
                        try:
                            my_func = getattr(self, func.__name__)
                        except AttributeError:
                            return NotImplemented
                        return my_func(*args, **kwargs)
                self.MyArray = MyArray
                class MyNoArrayFunctionArray():
                    def __init__(self, function=None):
                        self.function = function
                    self.MyNoArrayFunctionArray = MyNoArrayFunctionArray
            def add_method(self, name, arr_class, enable_value_error=False):
                def _definition(*args, **kwargs):
                    assert 'like' not in kwargs
                    if enable_value_error and 'value_error' in kwargs:
                        raise ValueError
                    return arr_class(getattr(arr_class, name))
                setattr(arr_class, name, _definition)
            def func_args(*args, **kwargs):

```

```

401: (8)             return args, kwargs
402: (4)         def test_array_like_not_implemented(self):
403: (8)             self.add_method('array', self.MyArray)
404: (8)             ref = self.MyArray.array()
405: (8)             with assert_raises_regex(TypeError, 'no implementation found'):
406: (12)                 array_like = np.asarray(1, like=ref)
407: (4)         _array_tests = [
408: (8)             ('array', *func_args((1,))),
409: (8)             ('asarray', *func_args((1,))),
410: (8)             ('asanyarray', *func_args((1,))),
411: (8)             ('ascontiguousarray', *func_args((2, 3))),
412: (8)             ('asfortranarray', *func_args((2, 3))),
413: (8)             ('require', *func_args((np.arange(6).reshape(2, 3),),
414: (31)                             requirements=['A', 'F'])),
415: (8)             ('empty', *func_args((1,))),
416: (8)             ('full', *func_args((1,), 2)),
417: (8)             ('ones', *func_args((1,))),
418: (8)             ('zeros', *func_args((1,))),
419: (8)             ('arange', *func_args(3)),
420: (8)             ('frombuffer', *func_args(b'\x00' * 8, dtype=int)),
421: (8)             ('fromiter', *func_args(range(3), dtype=int)),
422: (8)             ('fromstring', *func_args('1,2', dtype=int, sep=',', '')),
423: (8)             ('loadtxt', *func_args(lambda: StringIO('0 1\n2 3'))),
424: (8)             ('genfromtxt', *func_args(lambda: StringIO('1,2.1'),
425: (34)                             dtype=[('int', 'i8'), ('float', 'f8')]),
426: (34)                             delimiter=',', )),
427: (4)         ]
428: (4) @pytest.mark.parametrize('function, args, kwargs', _array_tests)
429: (4) @pytest.mark.parametrize('numpy_ref', [True, False])
430: (4) def test_array_like(self, function, args, kwargs, numpy_ref):
431: (8)     self.add_method('array', self.MyArray)
432: (8)     self.add_method(function, self.MyArray)
433: (8)     np_func = getattr(np, function)
434: (8)     my_func = getattr(self.MyArray, function)
435: (8)     if numpy_ref is True:
436: (12)         ref = np.array(1)
437: (8)     else:
438: (12)         ref = self.MyArray.array()
439: (8)     like_args = tuple(a() if callable(a) else a for a in args)
440: (8)     array_like = np_func(*like_args, **kwargs, like=ref)
441: (8)     if numpy_ref is True:
442: (12)         assert type(array_like) is np.ndarray
443: (12)         np_args = tuple(a() if callable(a) else a for a in args)
444: (12)         np_arr = np_func(*np_args, **kwargs)
445: (12)         if function == "empty":
446: (16)             np_arr.fill(1)
447: (16)             array_like.fill(1)
448: (12)             assert_equal(array_like, np_arr)
449: (8)         else:
450: (12)             assert type(array_like) is self.MyArray
451: (12)             assert array_like.function is my_func
452: (4) @pytest.mark.parametrize('function, args, kwargs', _array_tests)
453: (4) @pytest.mark.parametrize('ref', [1, [1], "MyNoArrayFunctionArray"])
454: (4) def test_no_array_function_like(self, function, args, kwargs, ref):
455: (8)     self.add_method('array', self.MyNoArrayFunctionArray)
456: (8)     self.add_method(function, self.MyNoArrayFunctionArray)
457: (8)     np_func = getattr(np, function)
458: (8)     if ref == "MyNoArrayFunctionArray":
459: (12)         ref = self.MyNoArrayFunctionArray.array()
460: (8)     like_args = tuple(a() if callable(a) else a for a in args)
461: (8)     with assert_raises_regex(TypeError,
462: (16)                     'The `like` argument must be an array-like that implements'):
463: (12)         np_func(*like_args, **kwargs, like=ref)
464: (4) @pytest.mark.parametrize('numpy_ref', [True, False])
465: (4) def test_array_like_fromfile(self, numpy_ref):
466: (8)     self.add_method('array', self.MyArray)
467: (8)     self.add_method("fromfile", self.MyArray)
468: (8)     if numpy_ref is True:
469: (12)         ref = np.array(1)

```

```

470: (8)
471: (12)
472: (8)
473: (8)
474: (12)
475: (12)
476: (12)
477: (12)
478: (16)
479: (16)
480: (16)
481: (16)
482: (12)
483: (16)
484: (16)
485: (4)
486: (8)
487: (8)
488: (8)
489: (12)
490: (4)
491: (4)
492: (8)
493: (8)
494: (8)
495: (8)
496: (8)
497: (8)
498: (8)
499: (8)
500: (12)
501: (12)
502: (8)
503: (0)
504: (4)
505: (4)
506: (8)
507: (12)
508: (8)
509: (8)
510: (8)
511: (4)
512: (4)
513: (4)
514: (4)
515: (8)
516: (4)
517: (4)
518: (4)
519: (4)
520: (4)
521: (4)
522: (8)
523: (0)
524: (4)
525: (4)
526: (4)
527: (8)
528: (28)
529: (8)
530: (8)
531: (8)
532: (4)
533: (4)

else:
    ref = self.MyArray.array()
data = np.random.random(5)
with tempfile.TemporaryDirectory() as tmpdir:
    fname = os.path.join(tmpdir, "testfile")
    data.tofile(fname)
    array_like = np.fromfile(fname, like=ref)
if numpy_ref is True:
    assert type(array_like) is np.ndarray
    np_res = np.fromfile(fname, like=ref)
    assert_equal(np_res, data)
    assert_equal(array_like, np_res)
else:
    assert type(array_like) is self.MyArray
    assert array_like.function is self.MyArray.fromfile
def test_exception_handling(self):
    self.add_method('array', self.MyArray, enable_value_error=True)
    ref = self.MyArray.array()
    with assert_raises(TypeError):
        np.array(1, value_error=True, like=ref)
@ pytest.mark.parametrize('function, args, kwargs', _array_tests)
def test_like_as_none(self, function, args, kwargs):
    self.add_method('array', self.MyArray)
    self.add_method(function, self.MyArray)
    np_func = getattr(np, function)
    like_args = tuple(a()) if callable(a) else a for a in args)
    like_args_exp = tuple(a() if callable(a) else a for a in args)
    array_like = np_func(*like_args, **kwargs, like=None)
    expected = np_func(*like_args_exp, **kwargs)
    if function == "empty":
        array_like.fill(1)
        expected.fill(1)
    assert_equal(array_like, expected)
def test_function_like():
    assert type(np.mean) is np.core._multiarray_umath._ArrayFunctionDispatcher
    class MyClass:
        def __array__(self):
            return np.arange(3)
        func1 = staticmethod(np.mean)
        func2 = np.mean
        func3 = classmethod(np.mean)
    m = MyClass()
    assert m.func1([10]) == 10
    assert m.func2() == 1 # mean of the arange
    with pytest.raises(TypeError, match="unsupported operand type"):
        m.func3()
    bound = np.mean.__get__(m, MyClass)
    assert bound() == 1
    bound = np.mean.__get__(None, MyClass) # unbound actually
    assert bound([10]) == 10
    bound = np.mean.__get__(MyClass) # classmethod
    with pytest.raises(TypeError, match="unsupported operand type"):
        bound()
def test_scipy_trapz_support_shim():
    import types
    import functools
    def _copy_func(f):
        g = types.FunctionType(f.__code__, f.__globals__, name=f.__name__,
                              argdefs=f.__defaults__, closure=f.__closure__)
        g = functools.update_wrapper(g, f)
        g.__kwdefaults__ = f.__kwdefaults__
        return g
    trapezoid = _copy_func(np.trapz)
    assert np.trapz([1, 2]) == trapezoid([1, 2])

```

-----  
File 116 - test\_records.py:

```

1: (0) import collections.abc
2: (0) import textwrap
3: (0) from io import BytesIO
4: (0) from os import path
5: (0) from pathlib import Path
6: (0) import pytest
7: (0) import numpy as np
8: (0) from numpy.testing import (
9: (4)     assert_, assert_equal, assert_array_equal, assert_array_almost_equal,
10: (4)     assert_raises, temppath,
11: (4) )
12: (0) from numpy.compat import pickle
13: (0) class TestFromrecords:
14: (4)     def test_fromrecords(self):
15: (8)         r = np.rec.fromrecords([[456, 'dbe', 1.2], [2, 'de', 1.3]],
16: (28)             names='col1,col2,col3')
17: (8)         assert_equal(r[0].item(), (456, 'dbe', 1.2))
18: (8)         assert_equal(r['col1'].dtype.kind, 'i')
19: (8)         assert_equal(r['col2'].dtype.kind, 'U')
20: (8)         assert_equal(r['col2'].dtype.itemsize, 12)
21: (8)         assert_equal(r['col3'].dtype.kind, 'f')
22: (4)     def test_fromrecords_0len(self):
23: (8)         """ Verify fromrecords works with a 0-length input """
24: (8)         dtype = [('a', float), ('b', float)]
25: (8)         r = np.rec.fromrecords([], dtype=dtype)
26: (8)         assert_equal(r.shape, (0,))
27: (4)     def test_fromrecords_2d(self):
28: (8)         data = [
29: (12)             [(1, 2), (3, 4), (5, 6)],
30: (12)             [(6, 5), (4, 3), (2, 1)]
31: (8)         ]
32: (8)         expected_a = [[1, 3, 5], [6, 4, 2]]
33: (8)         expected_b = [[2, 4, 6], [5, 3, 1]]
34: (8)         r1 = np.rec.fromrecords(data, dtype=[('a', int), ('b', int)])
35: (8)         assert_equal(r1['a'], expected_a)
36: (8)         assert_equal(r1['b'], expected_b)
37: (8)         r2 = np.rec.fromrecords(data, names=['a', 'b'])
38: (8)         assert_equal(r2['a'], expected_a)
39: (8)         assert_equal(r2['b'], expected_b)
40: (8)         assert_equal(r1, r2)
41: (4)     def test_method_array(self):
42: (8)         r = np.rec.array(b'abcdefg' * 100, formats='i2,a3,i4', shape=3,
byteorder='big')
43: (8)
44: (4)     def test_method_array2(self):
45: (8)         r = np.rec.array([(1, 11, 'a'), (2, 22, 'b'), (3, 33, 'c'), (4, 44,
'd'), (5, 55, 'ex'),
46: (21)             (6, 66, 'f'), (7, 77, 'g')], formats='u1,f4,a1')
47: (8)
48: (4)     def test_recarray_slices(self):
49: (8)         r = np.rec.array([(1, 11, 'a'), (2, 22, 'b'), (3, 33, 'c'), (4, 44,
'd'), (5, 55, 'ex'),
50: (21)             (6, 66, 'f'), (7, 77, 'g')], formats='u1,f4,a1')
51: (8)
52: (4)     def test_recarray_fromarrays(self):
53: (8)         x1 = np.array([1, 2, 3, 4])
54: (8)         x2 = np.array(['a', 'dd', 'xyz', '12'])
55: (8)         x3 = np.array([1.1, 2, 3, 4])
56: (8)         r = np.rec.fromarrays([x1, x2, x3], names='a,b,c')
57: (8)         assert_equal(r[1].item(), (2, 'dd', 2.0))
58: (8)         x1[1] = 34
59: (8)         assert_equal(r.a, np.array([1, 2, 3, 4]))
60: (4)     def test_recarray_fromfile(self):
61: (8)         data_dir = path.join(path.dirname(__file__), 'data')
62: (8)         filename = path.join(data_dir, 'recarray_from_file.fits')
63: (8)         fd = open(filename, 'rb')
64: (8)         fd.seek(2880 * 2)
65: (8)         r1 = np.rec.fromfile(fd, formats='f8,i4,a5', shape=3, byteorder='big')
66: (8)         fd.seek(2880 * 2)

```

```

67: (8)          r2 = np.rec.array(fd, formats='f8,i4,a5', shape=3, byteorder='big')
68: (8)          fd.seek(2880 * 2)
69: (8)          bytes_array = BytesIO()
70: (8)          bytes_array.write(fd.read())
71: (8)          bytes_array.seek(0)
72: (8)          r3 = np.rec.fromfile(bytes_array, formats='f8,i4,a5', shape=3,
byteorder='big')
73: (8)          fd.close()
74: (8)          assert_equal(r1, r2)
75: (8)          assert_equal(r2, r3)
76: (4)          def test_recarray_from_obj(self):
77: (8)          count = 10
78: (8)          a = np.zeros(count, dtype='O')
79: (8)          b = np.zeros(count, dtype='f8')
80: (8)          c = np.zeros(count, dtype='f8')
81: (8)          for i in range(len(a)):
82: (12)          a[i] = list(range(1, 10))
83: (8)          mine = np.rec.fromarrays([a, b, c], names='date,data1,data2')
84: (8)          for i in range(len(a)):
85: (12)          assert_(mine.date[i] == list(range(1, 10)))
86: (12)          assert_(mine.data1[i] == 0.0)
87: (12)          assert_(mine.data2[i] == 0.0)
88: (4)          def test_recarray_repr(self):
89: (8)          a = np.array([(1, 0.1), (2, 0.2)],
dtype=[('foo', '<i4'), ('bar', '<f8')])
90: (21)          a = np.rec.array(a)
91: (8)          assert_equal(
92: (8)          repr(a),
93: (12)          textwrap.dedent("""\
94: (12)          rec.array([(1, 0.1), (2, 0.2)],
dtype=[('foo', '<i4'), ('bar', '<f8')])"""))
95: (12)
96: (22)
97: (8)
98: (8)
99: (8)
100: (8)
101: (8)
102: (8)
103: (8)
104: (4)
105: (8)
106: (8)
107: (12)
108: (22)
109: (8)
110: (8)
111: (8)
112: (12)
113: (12)
114: (8)
115: (12)
116: (4)
117: (8)
118: (21)
119: (8)
120: (8)
121: (8)
122: (8)
123: (8)
124: (8)
125: (8)
126: (8)
127: (8)
128: (8)
129: (8)
130: (8)
131: (8)
132: (8)
133: (8)

record = arr_0d[()]
assert_equal(repr(record), "(1, 2., '2003')")
try:
    np.set_printoptions(legacy='1.13')
    assert_equal(repr(record), '(1, 2.0, datetime.date(2003, 1, 1))')
finally:
    np.set_printoptions(legacy=False)
def test_recarray_from_repr(self):
    a = np.array([(1,'ABC'), (2, "DEF")],
dtype=[('foo', int), ('bar', 'S4')])
    recordarr = np.rec.array(a)
    recarr = a.view(np.recarray)
    recordview = a.view(np.dtype((np.record, a.dtype)))
    recordarr_r = eval("numpy." + repr(recordarr), {'numpy': np})
    recarr_r = eval("numpy." + repr(recarr), {'numpy': np})
    recordview_r = eval("numpy." + repr(recordview), {'numpy': np})
    assert_equal(type(recordarr_r), np.recarray)
    assert_equal(recordarr_r.dtype.type, np.record)
    assert_equal(recordarr, recordarr_r)
    assert_equal(type(recarr_r), np.recarray)
    assert_equal(recarr_r.dtype.type, np.record)
    assert_equal(recarr, recarr_r)
    assert_equal(type(recordview_r), np.ndarray)
    assert_equal(recordview.dtype.type, np.record)
    assert_equal(recordview, recordview_r)

```

```

134: (4)
135: (8)
136: (21)
137: (8)
138: (8)
139: (8)
140: (8)
141: (8)
142: (8)
143: (8)
144: (8)
145: (8)
146: (8)
147: (8)
148: (8)
149: (8)
150: (8)
151: (43)
152: (8)
153: (8)
154: (8)
155: (12)
156: (8)
157: (8)
158: (8)
159: (22)
160: (8)
161: (26)
162: (8)
163: (8)
164: (8)
165: (8)
166: (8)
167: (8)
168: (8)
169: (8)
170: (8)
171: (8)
172: (8)
173: (12)
174: (12)
175: (12)
176: (12)
177: (4)
178: (8)
179: (12)
180: (12)
181: (12)
182: (23)
183: (8)
184: (12)
185: (12)
186: (12)
187: (23)
188: (8)
189: (8)
190: (8)
191: (12)
192: (4)
193: (8)
194: (24)
195: (23)
196: (8)
197: (8)
198: (8)
199: (8)
200: (8)
201: (8)
202: (8)

        def test_recarray_views(self):
            a = np.array([(1, 'ABC'), (2, "DEF")],
                         dtype=[('foo', int), ('bar', 'S4')])
            b = np.array([1,2,3,4,5], dtype=np.int64)
            assert_equal(np.rec.array(a).dtype.type, np.record)
            assert_equal(type(np.rec.array(a)), np.recarray)
            assert_equal(np.rec.array(b).dtype.type, np.int64)
            assert_equal(type(np.rec.array(b)), np.recarray)
            assert_equal(a.view(np.recarray).dtype.type, np.record)
            assert_equal(type(a.view(np.recarray)), np.recarray)
            assert_equal(b.view(np.recarray).dtype.type, np.int64)
            assert_equal(type(b.view(np.recarray)), np.recarray)
            r = np.rec.array(np.ones(4, dtype="f4,i4"))
            rv = r.view('f8').view('f4,i4')
            assert_equal(type(rv), np.recarray)
            assert_equal(rv.dtype.type, np.record)
            r = np.rec.array(np.ones(4, dtype=[('a', 'i4'), ('b', 'i4'),
  ('c', 'i4,i4')]))
            assert_equal(r['c'].dtype.type, np.record)
            assert_equal(type(r['c']), np.recarray)
            class C(np.recarray):
                pass
            c = r.view(C)
            assert_equal(type(c['c']), C)
            test_dtype = [('a', 'f4,f4'), ('b', 'V8'), ('c', ('f4',2)),
                          ('d', ('i8', 'i4,i4'))]
            r = np.rec.array([(1,1), b'11111111', [1,1], 1),
                            ((1,1), b'11111111', [1,1], 1)], dtype=test_dtype)
            assert_equal(r.a.dtype.type, np.record)
            assert_equal(r.b.dtype.type, np.void)
            assert_equal(r.c.dtype.type, np.float32)
            assert_equal(r.d.dtype.type, np.int64)
            r = np.rec.array(np.ones(4, dtype='i4,i4'))
            assert_equal(r.view('f4,f4').dtype.type, np.record)
            assert_equal(r.view(('i4',2)).dtype.type, np.int32)
            assert_equal(r.view('V8').dtype.type, np.void)
            assert_equal(r.view(('i8', 'i4,i4')).dtype.type, np.int64)
            arrs = [np.ones(4, dtype='f4,i4'), np.ones(4, dtype='f8')]
            for arr in arrs:
                rec = np.rec.array(arr)
                arr2 = rec.view(rec.dtype.fields or rec.dtype, np.ndarray)
                assert_equal(arr2.dtype.type, arr.dtype.type)
                assert_equal(type(arr2), type(arr))
        def test_recarray_from_names(self):
            ra = np.rec.array([
                (1, 'abc', 3.7000002861022949, 0),
                (2, 'xy', 6.6999998092651367, 1),
                (0, ' ', 0.40000000596046448, 0)],
                names='c1, c2, c3, c4')
            pa = np.rec.fromrecords([
                (1, 'abc', 3.7000002861022949, 0),
                (2, 'xy', 6.6999998092651367, 1),
                (0, ' ', 0.40000000596046448, 0)],
                names='c1, c2, c3, c4')
            assert_(ra.dtype == pa.dtype)
            assert_(ra.shape == pa.shape)
            for k in range(len(ra)):
                assert_(ra[k].item() == pa[k].item())
        def test_recarray_conflict_fields(self):
            ra = np.rec.array([(1, 'abc', 2.3), (2, 'xyz', 4.2),
                              (3, 'wrs', 1.3)],
                           names='field, shape, mean')
            ra.mean = [1.1, 2.2, 3.3]
            assert_array_almost_equal(ra['mean'], [1.1, 2.2, 3.3])
            assert_(type(ra.mean) is type(ra.var))
            ra.shape = (1, 3)
            assert_(ra.shape == (1, 3))
            ra.shape = ['A', 'B', 'C']
            assert_array_equal(ra['shape'], [['A', 'B', 'C']])

```

```

203: (8)           ra.field = 5
204: (8)           assert_array_equal(ra['field'], [[5, 5, 5]])
205: (8)           assert_(isinstance(ra.field, collections.abc.Callable))
206: (4)           def test_fromrecords_with_explicit_dtype(self):
207: (8)             a = np.rec.fromrecords([(1, 'a'), (2, 'bbb')], 
208: (32)                         dtype=[('a', int), ('b', object)])
209: (8)             assert_equal(a.a, [1, 2])
210: (8)             assert_equal(a[0].a, 1)
211: (8)             assert_equal(a.b, ['a', 'bbb'])
212: (8)             assert_equal(a[-1].b, 'bbb')
213: (8)             ndtype = np.dtype([('a', int), ('b', object)])
214: (8)             a = np.rec.fromrecords([(1, 'a'), (2, 'bbb')], dtype=ndtype)
215: (8)             assert_equal(a.a, [1, 2])
216: (8)             assert_equal(a[0].a, 1)
217: (8)             assert_equal(a.b, ['a', 'bbb'])
218: (8)             assert_equal(a[-1].b, 'bbb')
219: (4)           def test_recarray_stringtypes(self):
220: (8)             a = np.array([('abc ', 1), ('abc', 2)], 
221: (21)                         dtype=[('foo', 'S4'), ('bar', int)])
222: (8)             a = a.view(np.recarray)
223: (8)             assert_equal(a.foo[0] == a.foo[1], False)
224: (4)           def test_recarray_returntypes(self):
225: (8)             qux_fields = {'C': (np.dtype('S5'), 0), 'D': (np.dtype('S5'), 6)}
226: (8)             a = np.rec.array([('abc ', (1,1), 1, ('abcde', 'fgehi')), 
227: (26)                     ('abc', (2,3), 1, ('abcde', 'jklnm'))], 
228: (25)                     dtype=[('foo', 'S4'), 
229: (32)                     ('bar', [('A', int), ('B', int)]), 
230: (32)                     ('baz', int), ('qux', qux_fields)])
231: (8)             assert_equal(type(a.foo), np.ndarray)
232: (8)             assert_equal(type(a['foo']), np.ndarray)
233: (8)             assert_equal(type(a.bar), np.recarray)
234: (8)             assert_equal(type(a['bar']), np.recarray)
235: (8)             assert_equal(a.bar.dtype.type, np.record)
236: (8)             assert_equal(type(a['qux']), np.recarray)
237: (8)             assert_equal(a.qux.dtype.type, np.record)
238: (8)             assert_equal(dict(a.qux.dtype.fields), qux_fields)
239: (8)             assert_equal(type(a.baz), np.ndarray)
240: (8)             assert_equal(type(a['baz']), np.ndarray)
241: (8)             assert_equal(type(a[0].bar), np.record)
242: (8)             assert_equal(type(a[0]['bar']), np.record)
243: (8)             assert_equal(a[0].bar.A, 1)
244: (8)             assert_equal(a[0].bar['A'], 1)
245: (8)             assert_equal(a[0]['bar'].A, 1)
246: (8)             assert_equal(a[0]['bar']['A'], 1)
247: (8)             assert_equal(a[0].qux.D, b'fgehi')
248: (8)             assert_equal(a[0].qux['D'], b'fgehi')
249: (8)             assert_equal(a[0]['qux'].D, b'fgehi')
250: (8)             assert_equal(a[0]['qux']['D'], b'fgehi')
251: (4)           def test_zero_width_strings(self):
252: (8)             cols = [['test'] * 3, [''] * 3]
253: (8)             rec = np.rec.fromarrays(cols)
254: (8)             assert_equal(rec['f0'], ['test', 'test', 'test'])
255: (8)             assert_equal(rec['f1'], ['', '', ''])
256: (8)             dt = np.dtype([('f0', '|S4'), ('f1', '|S')])
257: (8)             rec = np.rec.fromarrays(cols, dtype=dt)
258: (8)             assert_equal(rec.itemsize, 4)
259: (8)             assert_equal(rec['f0'], [b'test', b'test', b'test'])
260: (8)             assert_equal(rec['f1'], [b'', b'', b''])
261: (0)           class TestPathUsage:
262: (4)             def test_tofile_fromfile(self):
263: (8)               with temppath(suffix='.bin') as path:
264: (12)                 path = Path(path)
265: (12)                 np.random.seed(123)
266: (12)                 a = np.random.rand(10).astype('f8,i4,a5')
267: (12)                 a[5] = (0.5, 10, 'abcde')
268: (12)                 with path.open("wb") as fd:
269: (16)                   a.tofile(fd)
270: (12)                 x = np.core.records.fromfile(path,
271: (41)                               formats='f8,i4,a5',

```

```

272: (41)
273: (12)           assert_array_equal(x, a)
274: (0)
275: (4)
276: (8)
277: (28)
278: (35)
279: (35)
280: (4)
281: (8)
282: (8)
283: (8)
284: (8)
285: (4)
286: (8)
287: (8)
288: (8)
289: (8)
290: (4)
291: (8)
292: (8)
293: (12)
294: (8)
295: (4)
296: (8)
297: (8)
298: (8)
299: (12)
300: (8)
301: (12)
302: (4)
303: (8)
304: (8)
305: (8)
306: (21)
307: (8)
308: (8)
309: (8)
310: (4)
311: (8)
312: (8)
313: (12)
314: (12)
315: (57)
316: (4)
317: (8)
318: (8)
319: (12)
320: (12)
321: (57)
322: (4)
323: (8)
324: (8)
325: (12)
326: (12)
327: (12)
328: (12)
329: (12)
330: (4)
331: (8)
332: (8)
333: (8)
334: (8)
335: (8)
336: (8)
337: (8)
338: (8)
339: (8)
340: (8)

    shape=10)
class TestRecord:
    def setup_method(self):
        self.data = np.rec.fromrecords([(1, 2, 3), (4, 5, 6)],
                                       dtype=[("col1", "<i4"),
  ("col2", "<i4"),
  ("col3", "<i4")])
    def test_assignment1(self):
        a = self.data
        assert_equal(a.col1[0], 1)
        a[0].col1 = 0
        assert_equal(a.col1[0], 0)
    def test_assignment2(self):
        a = self.data
        assert_equal(a.col1[0], 1)
        a.col1[0] = 0
        assert_equal(a.col1[0], 0)
    def test_invalid_assignment(self):
        a = self.data
        def assign_invalid_column(x):
            x[0].col5 = 1
        assert_raises(AttributeError, assign_invalid_column, a)
    def test_nonwriteable_setfield(self):
        r = np.rec.array([(0,), (1,)], dtype=[('f', 'i4')])
        r.flags.writeable = False
        with assert_raises(ValueError):
            r.f = [2, 3]
        with assert_raises(ValueError):
            r.setfield([2,3], *r.dtype.fields['f'])
    def test_out_of_order_fields(self):
        x = self.data[['col1', 'col2']]
        assert_equal(x.dtype.names, ('col1', 'col2'))
        assert_equal(x.dtype.descr,
                     [('col1', '<i4'), ('col2', '<i4'), ('', '|V4')])
        y = self.data[['col2', 'col1']]
        assert_equal(y.dtype.names, ('col2', 'col1'))
        assert_raises(ValueError, lambda: y.dtype.descr)
    def test_pickle_1(self):
        a = np.array([(1, [])], dtype=[('a', np.int32), ('b', np.int32, 0)])
        for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
            assert_equal(a, pickle.loads(pickle.dumps(a, protocol=proto)))
            assert_equal(a[0], pickle.loads(pickle.dumps(a[0],
   protocol=proto)))
    def test_pickle_2(self):
        a = self.data
        for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
            assert_equal(a, pickle.loads(pickle.dumps(a, protocol=proto)))
            assert_equal(a[0], pickle.loads(pickle.dumps(a[0],
   protocol=proto)))
    def test_pickle_3(self):
        a = self.data
        for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
            pa = pickle.loads(pickle.dumps(a[0], protocol=proto))
            assert_(pa.flags.c_contiguous)
            assert_(pa.flags.f_contiguous)
            assert_(pa.flags.writeable)
            assert_(pa.flags.aligned)
    def test_pickle_void(self):
        dt = np.dtype([('obj', 'O'), ('int', 'i')])
        a = np.empty(1, dtype=dt)
        data = (bytearray(b'eman'),)
        a['obj'] = data
        a['int'] = 42
        ctor, args = a[0].__reduce__()
        assert ctor is np.core.multiarray.scalar
        dtype, obj = args
        assert not isinstance(obj, bytes)
        assert_raises(RuntimeError, ctor, dtype, 13)

```

```

341: (8)           dump = pickle.dumps(a[0])
342: (8)           unpickled = pickle.loads(dump)
343: (8)           assert a[0] == unpickled
344: (8)           with pytest.warns(DeprecationWarning):
345: (12)             assert ctor(np.dtype("O"), data) is data
346: (4) def test_objview_record(self):
347: (8)     dt = np.dtype([('foo', 'i8'), ('bar', 'O')])
348: (8)     r = np.zeros((1,3), dtype=dt).view(np.recarray)
349: (8)     r.foo = np.array([1, 2, 3]) # TypeError?
350: (8)     ra = np.recarray((2,), dtype=[('x', object), ('y', float), ('z', int)])
351: (8)     ra[['x', 'y']] # TypeError?
352: (4) def test_record_scalar_setitem(self):
353: (8)     rec = np.recarray(1, dtype=[('x', float, 5)])
354: (8)     rec[0].x = 1
355: (8)     assert_equal(rec[0].x, np.ones(5))
356: (4) def test_missing_field(self):
357: (8)     arr = np.zeros((3,), dtype=[('x', int), ('y', int)])
358: (8)     assert_raises(KeyError, lambda: arr[['nofield']])
359: (4) def test_fromarrays_nested_structured_arrays(self):
360: (8)     arrays = [
361: (12)       np.arange(10),
362: (12)       np.ones(10, dtype=[('a', '<u2'), ('b', '<f4')]),
363: (8)   ]
364: (8)     arr = np.rec.fromarrays(arrays) # ValueError?
365: (4) @pytest.mark.parametrize('nfields', [0, 1, 2])
366: (4) def test_assign_dtype_attribute(self, nfields):
367: (8)     dt = np.dtype([('a', np.uint8), ('b', np.uint8), ('c', np.uint8)])
368: (8)     data = np.zeros(3, dt).view(np.recarray)
369: (8)     assert data.dtype.type == np.record
370: (8)     assert dt.type != np.record
371: (8)     data.dtype = dt
372: (8)     assert data.dtype.type == np.record
373: (4) @pytest.mark.parametrize('nfields', [0, 1, 2])
374: (4) def test_nested_fields_are_records(self, nfields):
375: (8)     """ Test that nested structured types are treated as records too """
376: (8)     dt = np.dtype([('a', np.uint8), ('b', np.uint8), ('c', np.uint8)])
377: (8)     dt_outer = np.dtype([('inner', dt)])
378: (8)     data = np.zeros(3, dt_outer).view(np.recarray)
379: (8)     assert isinstance(data, np.recarray)
380: (8)     assert isinstance(data['inner'], np.recarray)
381: (8)     data0 = data[0]
382: (8)     assert isinstance(data0, np.record)
383: (8)     assert isinstance(data0['inner'], np.record)
384: (4) def test_nested_dtype_padding(self):
385: (8)     """ test that trailing padding is preserved """
386: (8)     dt = np.dtype([('a', np.uint8), ('b', np.uint8), ('c', np.uint8)])
387: (8)     dt_padded_end = dt[['a', 'b']]
388: (8)     assert dt_padded_end.itemsize == dt.itemsize
389: (8)     dt_outer = np.dtype([('inner', dt_padded_end)])
390: (8)     data = np.zeros(3, dt_outer).view(np.recarray)
391: (8)     assert_equal(data['inner'].dtype, dt_padded_end)
392: (8)     data0 = data[0]
393: (8)     assert_equal(data0['inner'].dtype, dt_padded_end)
394: (0) def test_find_duplicate():
395: (4)     l1 = [1, 2, 3, 4, 5, 6]
396: (4)     assert_(np.rec.find_duplicate(l1) == [])
397: (4)     l2 = [1, 2, 1, 4, 5, 6]
398: (4)     assert_(np.rec.find_duplicate(l2) == [1])
399: (4)     l3 = [1, 2, 1, 4, 1, 6, 2, 3]
400: (4)     assert_(np.rec.find_duplicate(l3) == [1, 2])
401: (4)     l3 = [2, 2, 1, 4, 1, 6, 2, 3]
402: (4)     assert_(np.rec.find_duplicate(l3) == [2, 1])

```

```

1: (0)          import pytest
2: (0)          import warnings
3: (0)          import numpy as np
4: (0)          @pytest.mark.filterwarnings("error")
5: (0)          def test_getattr_warning():
6: (4)            class Wrapper:
7: (8)              def __init__(self, array):
8: (12)                self.array = array
9: (8)              def __len__(self):
10: (12)                return len(self.array)
11: (8)              def __getitem__(self, item):
12: (12)                return type(self)(self.array[item])
13: (8)              def __getattr__(self, name):
14: (12)                if name.startswith("__array__"):
15: (16)                  warnings.warn("object got converted", UserWarning,
stacklevel=1)
16: (12)                  return getattr(self.array, name)
17: (8)              def __repr__(self):
18: (12)                return "<Wrapper({self.array})>".format(self=self)
19: (4)                array = Wrapper(np.arange(10))
20: (4)                with pytest.raises(UserWarning, match="object got converted"):
21: (8)                  np.asarray(array)
22: (0)          def test_array_called():
23: (4)            class Wrapper:
24: (8)              val = '0' * 100
25: (8)              def __array__(self, result=None):
26: (12)                return np.array([self.val], dtype=object)
27: (4)                wrapped = Wrapper()
28: (4)                arr = np.array(wrapped, dtype=str)
29: (4)                assert arr.dtype == 'U100'
30: (4)                assert arr[0] == Wrapper.val

```

---

## File 118 - test\_regression.py:

```

1: (0)          import copy
2: (0)          import sys
3: (0)          import gc
4: (0)          import tempfile
5: (0)          import pytest
6: (0)          from os import path
7: (0)          from io import BytesIO
8: (0)          from itertools import chain
9: (0)          import numpy as np
10: (0)          from numpy.testing import (
11: (8)            assert_, assert_equal, IS_PYPY, assert_almost_equal,
12: (8)            assert_array_equal, assert_array_almost_equal, assert_raises,
13: (8)            assert_raises_regex, assert_warns, suppress_warnings,
14: (8)            _assert_valid_refcount, HAS_REFCOUNT, IS_PYTHON, IS_WASM
15: (8)          )
16: (0)          from numpy.testing._private.utils import _no_tracing, requires_memory
17: (0)          from numpy.compat import asbytes, asunicode, pickle
18: (0)          class TestRegression:
19: (4)            def test_invalid_round(self):
20: (8)              v = 4.759999999999998
21: (8)              assert_array_equal(np.array([v]), np.array(v))
22: (4)            def test_mem_empty(self):
23: (8)              np.empty((1,), dtype=[('x', np.int64)])
24: (4)            def test_pickle_transposed(self):
25: (8)              a = np.transpose(np.array([[2, 9], [7, 0], [3, 8]]))
26: (8)              for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
27: (12)                with BytesIO() as f:
28: (16)                  pickle.dump(a, f, protocol=proto)
29: (16)                  f.seek(0)
30: (16)                  b = pickle.load(f)
31: (12)                  assert_array_equal(a, b)
32: (4)            def test_dtype_names(self):

```

```

33: (8)           np.dtype([([('name', 'label'), np.int32, 3])])
34: (4)           def test_reduce(self):
35: (8)             assert_almost_equal(np.add.reduce([1., .5], dtype=None), 1.5)
36: (4)           def test_zeros_order(self):
37: (8)             np.zeros([3], int, 'C')
38: (8)             np.zeros([3], order='C')
39: (8)             np.zeros([3], int, order='C')
40: (4)           def test_asarray_with_order(self):
41: (8)             a = np.ones(2)
42: (8)             assert_(a is np.asarray(a, order='F'))
43: (4)           def test_ravel_with_order(self):
44: (8)             a = np.ones(2)
45: (8)             assert_(not a.ravel('F').flags.owndata)
46: (4)           def test_sort_bigEndian(self):
47: (8)             a = np.linspace(0, 10, 11)
48: (8)             c = a.astype(np.dtype('<f8'))
49: (8)             c.sort()
50: (8)             assert_array_almost_equal(c, a)
51: (4)           def test_negative_nd_indexing(self):
52: (8)             c = np.arange(125).reshape((5, 5, 5))
53: (8)             origidx = np.array([-1, 0, 1])
54: (8)             idx = np.array(origidx)
55: (8)             c[idx]
56: (8)             assert_array_equal(idx, origidx)
57: (4)           def test_char_dump(self):
58: (8)             ca = np.char.array(np.arange(1000, 1010), itemsize=4)
59: (8)             for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
60: (12)               with BytesIO() as f:
61: (16)                 pickle.dump(ca, f, protocol=proto)
62: (16)                 f.seek(0)
63: (16)                 ca = np.load(f, allow_pickle=True)
64: (4)           def test_noncontiguous_fill(self):
65: (8)             a = np.zeros((5, 3))
66: (8)             b = a[:, :2,]
67: (8)             def rs():
68: (12)               b.shape = (10,)
69: (8)               assert_raises(AttributeError, rs)
70: (4)           def test_bool(self):
71: (8)             np.bool_(1) # Should succeed
72: (4)           def test_indexing1(self):
73: (8)             descr = [(['x', [(['y', [(['z', 'c16', (2,)]), ]), ]], ],
74: (8)             buffer = ((([6j, 4j], ), ), )
75: (8)             h = np.array(buffer, dtype=descr)
76: (8)             h['x']['y']['z']
77: (4)           def test_indexing2(self):
78: (8)             descr = [(['x', 'i4', (2, )]]
79: (8)             buffer = ([3, 2], )
80: (8)             h = np.array(buffer, dtype=descr)
81: (8)             h['x']
82: (4)           def test_round(self):
83: (8)             x = np.array([1+2j])
84: (8)             assert_almost_equal(x**(-1), [1/(1+2j)])
85: (4)           def test_scalar_compare(self):
86: (8)             a = np.array(['test', 'auto'])
87: (8)             assert_array_equal(a == 'auto', np.array([False, True]))
88: (8)             assert_(a[1] == 'auto')
89: (8)             assert_(a[0] != 'auto')
90: (8)             b = np.linspace(0, 10, 11)
91: (8)             assert_array_equal(b != 'auto', np.ones(11, dtype=bool))
92: (8)             assert_(b[0] != 'auto')
93: (4)           def test_unicode_swapping(self):
94: (8)             ulen = 1
95: (8)             ucs_value = '\U0010FFFF'
96: (8)             ua = np.array([[ucs_value*ulen]*2]*3)*4, dtype='U%s' % ulen)
97: (8)             ua.newbyteorder() # Should succeed.
98: (4)           def test_object_array_fill(self):
99: (8)             x = np.zeros(1, 'O')
100: (8)             x.fill([])
101: (4)           def test_mem_dtype_align(self):

```

```

102: (8) assert_raises(TypeError, np.dtype,
103: (30)                                     {'names': ['a'], 'formats': ['foo']}, align=1)
104: (4) def test_endian_bool_indexing(self):
105: (8)     a = np.arange(10., dtype='>f8')
106: (8)     b = np.arange(10., dtype='<f8')
107: (8)     xa = np.where((a > 2) & (a < 6))
108: (8)     xb = np.where((b > 2) & (b < 6))
109: (8)     ya = ((a > 2) & (a < 6))
110: (8)     yb = ((b > 2) & (b < 6))
111: (8)     assert_array_almost_equal(xa, ya.nonzero())
112: (8)     assert_array_almost_equal(xb, yb.nonzero())
113: (8)     assert_(np.all(a[ya] > 0.5))
114: (8)     assert_(np.all(b[yb] > 0.5))
115: (4) def test_endian_where(self):
116: (8)     net = np.zeros(3, dtype='>f4')
117: (8)     net[1] = 0.00458849
118: (8)     net[2] = 0.605202
119: (8)     max_net = net.max()
120: (8)     test = np.where(net <= 0., max_net, net)
121: (8)     correct = np.array([0.60520202, 0.00458849, 0.60520202])
122: (8)     assert_array_almost_equal(test, correct)
123: (4) def test_endian_recarray(self):
124: (8)     dt = np.dtype([
125: (15)         ('head', '>u4'),
126: (15)         ('data', '>u4', 2),
127: (12)     ])
128: (8)     buf = np.recarray(1, dtype=dt)
129: (8)     buf[0]['head'] = 1
130: (8)     buf[0]['data'][:] = [1, 1]
131: (8)     h = buf[0]['head']
132: (8)     d = buf[0]['data'][0]
133: (8)     buf[0]['head'] = h
134: (8)     buf[0]['data'][0] = d
135: (8)     assert_(buf[0]['head'] == 1)
136: (4) def test_mem_dot(self):
137: (8)     x = np.random.randn(0, 1)
138: (8)     y = np.random.randn(10, 1)
139: (8)     _z = np.ones(10)
140: (8)     _dummy = np.empty((0, 10))
141: (8)     z = np.lib.stride_tricks.as_strided(_z, _dummy.shape, _dummy.strides)
142: (8)     np.dot(x, np.transpose(y), out=z)
143: (8)     assert_equal(_z, np.ones(10))
144: (8)     np.core.multiarray.dot(x, np.transpose(y), out=z)
145: (8)     assert_equal(_z, np.ones(10))
146: (4) def test_arange_endian(self):
147: (8)     ref = np.arange(10)
148: (8)     x = np.arange(10, dtype='<f8')
149: (8)     assert_array_equal(ref, x)
150: (8)     x = np.arange(10, dtype='>f8')
151: (8)     assert_array_equal(ref, x)
152: (4) def test_arange_inf_step(self):
153: (8)     ref = np.arange(0, 1, 10)
154: (8)     x = np.arange(0, 1, np.inf)
155: (8)     assert_array_equal(ref, x)
156: (8)     ref = np.arange(0, 1, -10)
157: (8)     x = np.arange(0, 1, -np.inf)
158: (8)     assert_array_equal(ref, x)
159: (8)     ref = np.arange(0, -1, -10)
160: (8)     x = np.arange(0, -1, -np.inf)
161: (8)     assert_array_equal(ref, x)
162: (8)     ref = np.arange(0, -1, 10)
163: (8)     x = np.arange(0, -1, np.inf)
164: (8)     assert_array_equal(ref, x)
165: (4) def test_arange_underflow_stop_and_step(self):
166: (8)     finfo = np.finfo(np.float64)
167: (8)     ref = np.arange(0, finfo.eps, 2 * finfo.eps)
168: (8)     x = np.arange(0, finfo.eps, finfo.max)
169: (8)     assert_array_equal(ref, x)
170: (8)     ref = np.arange(0, finfo.eps, -2 * finfo.eps)

```

```

171: (8)           x = np.arange(0, finfo.eps, -finfo.max)
172: (8)           assert_array_equal(ref, x)
173: (8)           ref = np.arange(0, -finfo.eps, -2 * finfo.eps)
174: (8)           x = np.arange(0, -finfo.eps, -finfo.max)
175: (8)           assert_array_equal(ref, x)
176: (8)           ref = np.arange(0, -finfo.eps, 2 * finfo.eps)
177: (8)           x = np.arange(0, -finfo.eps, finfo.max)
178: (8)           assert_array_equal(ref, x)
179: (4)           def test_argmax(self):
180: (8)             a = np.random.normal(0, 1, (4, 5, 6, 7, 8))
181: (8)             for i in range(a.ndim):
182: (12)               a.argmax(i) # Should succeed
183: (4)           def test_mem_divmod(self):
184: (8)             for i in range(10):
185: (12)               divmod(np.array([i])[0], 10)
186: (4)           def test_hstack_invalid_dims(self):
187: (8)             x = np.arange(9).reshape((3, 3))
188: (8)             y = np.array([0, 0, 0])
189: (8)             assert_raises(ValueError, np.hstack, (x, y))
190: (4)           def test_squeeze_type(self):
191: (8)             a = np.array([3])
192: (8)             b = np.array(3)
193: (8)             assert_(type(a.squeeze()) is np.ndarray)
194: (8)             assert_(type(b.squeeze()) is np.ndarray)
195: (4)           def test_add_identity(self):
196: (8)             assert_equal(0, np.add.identity)
197: (4)           def test_numpy_float_python_long_addition(self):
198: (8)             a = np.float_(23.) + 2**135
199: (8)             assert_equal(a, 23. + 2**135)
200: (4)           def test_binary_repr_0(self):
201: (8)             assert_equal('0', np.binary_repr(0))
202: (4)           def test_rec_iterate(self):
203: (8)             descr = np.dtype([('i', int), ('f', float), ('s', '|S3')])
204: (8)             x = np.rec.array([(1, 1.1, '1.0'),
205: (25)                           (2, 2.2, '2.0')], dtype=descr)
206: (8)             x[0].tolist()
207: (8)             [i for i in x[0]]
208: (4)           def test_unicode_string_comparison(self):
209: (8)             a = np.array('hello', np.str_)
210: (8)             b = np.array('world')
211: (8)             a == b
212: (4)           def test_tobytes_FORTRANORDER_discontiguous(self):
213: (8)             x = np.array(np.random.rand(3, 3), order='F')[ :, :2]
214: (8)             assert_array_almost_equal(x.ravel(), np.frombuffer(x.tobytes()))
215: (4)           def test_flat_assignment(self):
216: (8)             x = np.empty((3, 1))
217: (8)             x.flat = np.arange(3)
218: (8)             assert_array_almost_equal(x, [[0], [1], [2]])
219: (8)             x.flat = np.arange(3, dtype=float)
220: (8)             assert_array_almost_equal(x, [[0], [1], [2]])
221: (4)           def test_broadcast_flat_assignment(self):
222: (8)             x = np.empty((3, 1))
223: (8)             def bfa():
224: (12)               x[:] = np.arange(3)
225: (8)             def bfb():
226: (12)               x[:] = np.arange(3, dtype=float)
227: (8)             assert_raises(ValueError, bfa)
228: (8)             assert_raises(ValueError, bfb)
229: (4)             @pytest.mark.xfail(IS_WASM, reason="not sure why")
230: (4)             @pytest.mark.parametrize("index",
231: (12)               [np.ones(10, dtype=bool), np.arange(10)],
232: (12)               ids=["boolean-arr-index", "integer-arr-index"])
233: (4)           def test_nonarray_assignment(self, index):
234: (8)             a = np.arange(10)
235: (8)             with pytest.raises(ValueError):
236: (12)               a[index] = np.nan
237: (8)             with np.errstate(invalid="warn"):
238: (12)               with pytest.warns(RuntimeWarning, match="invalid value"):
239: (16)                 a[index] = np.array(np.nan) # Only warns

```

```

240: (4)     def test_unpickle_dtype_with_object(self):
241: (8)         dt = np.dtype([('x', int), ('y', np.object_), ('z', 'O')])
242: (8)         for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
243: (12)             with BytesIO() as f:
244: (16)                 pickle.dump(dt, f, protocol=proto)
245: (16)                 f.seek(0)
246: (16)                 dt_ = pickle.load(f)
247: (12)                 assert_equal(dt, dt_)
248: (4)     def test_mem_array_creation_invalid_specification(self):
249: (8)         dt = np.dtype([('x', int), ('y', np.object_)])
250: (8)         assert_raises(ValueError, np.array, [1, 'object'], dt)
251: (8)         np.array([(1, 'object')], dt)
252: (4)     def test_recarray_single_element(self):
253: (8)         a = np.array([1, 2, 3], dtype=np.int32)
254: (8)         b = a.copy()
255: (8)         r = np.rec.array(a, shape=1, formats=['3i4'], names=['d'])
256: (8)         assert_array_equal(a, b)
257: (8)         assert_equal(a, r[0][0])
258: (4)     def test_zero_sized_array_indexing(self):
259: (8)         tmp = np.array([])
260: (8)         def index_tmp():
261: (12)             tmp[np.array(10)]
262: (8)             assert_raises(IndexError, index_tmp)
263: (4)     def test_chararray_rstrip(self):
264: (8)         x = np.chararray((1,), 5)
265: (8)         x[0] = b'a '
266: (8)         x = x.rstrip()
267: (8)         assert_equal(x[0], b'a')
268: (4)     def test_object_array_shape(self):
269: (8)         assert_equal(np.array([[1, 2], 3, 4], dtype=object).shape, (3,))
270: (8)         assert_equal(np.array([[1, 2], [3, 4]], dtype=object).shape, (2, 2))
271: (8)         assert_equal(np.array([(1, 2), (3, 4)], dtype=object).shape, (2, 2))
272: (8)         assert_equal(np.array([], dtype=object).shape, (0,))
273: (8)         assert_equal(np.array([[], [], []], dtype=object).shape, (3, 0))
274: (8)         assert_equal(np.array([[3, 4], [5, 6], None], dtype=object).shape,
275: (3,)))
276: (4)     def test_mem_around(self):
277: (8)         x = np.zeros((1,))
278: (8)         y = [0]
279: (8)         decimal = 6
280: (8)         np.around(abs(x-y), decimal) <= 10.0**(-decimal)
281: (4)     def test_character_array_strip(self):
282: (8)         x = np.char.array("x", "x ", "x ")
283: (8)         for c in x:
284: (12)             assert_equal(c, "x")
285: (4)     def test_lexsort(self):
286: (8)         v = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
287: (8)         assert_equal(np.lexsort(v), 0)
288: (4)     def test_lexsort_invalid_sequence(self):
289: (8)         class BuggySequence:
290: (12)             def __len__(self):
291: (16)                 return 4
292: (12)             def __getitem__(self, key):
293: (16)                 raise KeyError
294: (8)                 assert_raises(KeyError, np.lexsort, BuggySequence())
295: (4)     def test_lexsort_zerolen_custom_strides(self):
296: (8)         xs = np.array([], dtype='i8')
297: (8)         assert np.lexsort((xs,)).shape[0] == 0 # Works
298: (8)         xs.strides = (16,)
299: (8)         assert np.lexsort((xs,)).shape[0] == 0 # Was: MemoryError
300: (4)     def test_lexsort_zerolen_custom_strides_2d(self):
301: (8)         xs = np.array([], dtype='i8')
302: (8)         xs.shape = (0, 2)
303: (8)         xs.strides = (16, 16)
304: (8)         assert np.lexsort((xs,), axis=0).shape[0] == 0
305: (8)         xs.shape = (2, 0)
306: (8)         xs.strides = (16, 16)
307: (8)         assert np.lexsort((xs,), axis=0).shape[0] == 2
308: (4)     def test_lexsort_invalid_axis(self):

```

```

308: (8) assert_raises(np.AxisError, np.lexsort, (np.arange(1),), axis=2)
309: (8) assert_raises(np.AxisError, np.lexsort, (np.array([]),), axis=1)
310: (8) assert_raises(np.AxisError, np.lexsort, (np.array(1),), axis=10)
311: (4) def test_lexsort_zerolen_element(self):
312: (8)     dt = np.dtype([]) # a void dtype with no fields
313: (8)     xs = np.empty(4, dt)
314: (8)     assert np.lexsort((xs,)).shape[0] == xs.shape[0]
315: (4) def test_pickle_py2_bytes_encoding(self):
316: (8)     test_data = [
317: (12)         (np.str_('\u06f2c'),
318: (13)             b"cnumpy.core.multiarray\nscalar\n(numpy\\ndtype\\np1\\n"
319: (13)             b"(S'U1'\\np2\\nI0\\nI1\\ntp3\\nRp4\\n(I3\\nS'<'\\np5\\nNNI4\\nI4\\n"
320: (13)             b"I0\\ntp6\\nbS',o\\x00\\x00'\\np7\\ntp8\\nRp9\\n."),
321: (12)             (np.array([9e123], dtype=np.float64),
322: (13)                 b"cnumpy.core.multiarray\\n_reconstruct\\np0\\n(numpy\\nndarray\\n"
323: (13)
b"p1\\n(I0\\ntp2\\nS'b'\\np3\\ntp4\\nRp5\\n(I1\\n(I1\\ntp6\\ncnumpy\\ndtype\\n"
324: (13)             b"p7\\n(S'f8'\\np8\\nI0\\nI1\\ntp9\\nRp10\\n(I3\\nS'<'\\np11\\nNNI-1\\nI-
1\\n"
325: (13)             b"I0\\ntp12\\nbI00\\nS'0\\x81\\xb7Z\\xaa:\\xabY'\\np13\\ntp14\\nb."),
326: (12)             (np.array([(9e123,)], dtype=[('name', float)])),
327: (13)
b"cnumpy.core.multiarray\\n_reconstruct\\np0\\n(numpy\\nndarray\\np1\\n"
328: (13)             b"
(I0\\ntp2\\nS'b'\\np3\\ntp4\\nRp5\\n(I1\\n(I1\\ntp6\\ncnumpy\\ndtype\\np7\\n"
329: (13)             b"
(S'V8'\\np8\\nI0\\nI1\\ntp9\\nRp10\\n(I3\\nS' | '\\np11\\nN(S'name'\\np12\\ntp13\\n"
330: (13)             b"
(dp14\\ng12\\n(g7\\n(S'f8'\\np15\\nI0\\nI1\\ntp16\\nRp17\\n(I3\\nS'<'\\np18\\nNNI-1\\n"
331: (13)             b"I-1\\nI0\\ntp19\\nbI0\\ntp20\\nsI8\\nI1\\nI0\\ntp21\\n"
332: (13)             b"bI00\\nS'0\\x81\\xb7Z\\xaa:\\xabY'\\np22\\ntp23\\nb."),
333: (8)         ]
334: (8)         for original, data in test_data:
335: (12)             result = pickle.loads(data, encoding='bytes')
336: (12)             assert_equal(result, original)
337: (12)             if isinstance(result, np.ndarray) and result.dtype.names is not
None:
338: (16)                 for name in result.dtype.names:
339: (20)                     assert_(isinstance(name, str))
340: (4) def test_pickle_dtype(self):
341: (8)     for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
342: (12)         pickle.dumps(float, protocol=proto)
343: (4) def test_swap_real(self):
344: (8)     assert_equal(np.arange(4, dtype='>c8').imag.max(), 0.0)
345: (8)     assert_equal(np.arange(4, dtype='<c8').imag.max(), 0.0)
346: (8)     assert_equal(np.arange(4, dtype='>c8').real.max(), 3.0)
347: (8)     assert_equal(np.arange(4, dtype='<c8').real.max(), 3.0)
348: (4) def test_object_array_from_list(self):
349: (8)     assert_(np.array([1, None, 'A']).shape == (3,))
350: (4) def test_multiple_assign(self):
351: (8)     a = np.zeros((3, 1), int)
352: (8)     a[[1, 2]] = 1
353: (4) def test_empty_array_type(self):
354: (8)     assert_equal(np.array([]).dtype, np.zeros(0).dtype)
355: (4) def test_void_copyswap(self):
356: (8)     dt = np.dtype([('one', '<i4'), ('two', '<i4')])
357: (8)     x = np.array((1, 2), dtype=dt)
358: (8)     x = x.byteswap()
359: (8)     assert_(x['one'] > 1 and x['two'] > 2)
360: (4) def test_method_args(self):
361: (8)     funcs1 = ['argmax', 'argmin', 'sum', 'any', 'all', 'cumsum',
362: (18)             'ptp', 'cumprod', 'prod', 'std', 'var', 'mean',
363: (18)             'round', 'min', 'max', 'argsort', 'sort']
364: (8)     funcs2 = ['compress', 'take', 'repeat']
365: (8)     for func in funcs1:
366: (12)         arr = np.random.rand(8, 7)
367: (12)         arr2 = arr.copy()
368: (12)         res1 = getattr(arr, func)()
369: (12)         res2 = getattr(np, func)(arr2)

```

```

370: (12)             if res1 is None:
371: (16)                 res1 = arr
372: (12)             if res1.dtype.kind in 'uib':
373: (16)                 assert_(res1 == res2).all(), func)
374: (12)             else:
375: (16)                 assert_(abs(res1-res2).max() < 1e-8, func)
376: (8)             for func in funcs2:
377: (12)                 arr1 = np.random.rand(8, 7)
378: (12)                 arr2 = np.random.rand(8, 7)
379: (12)                 res1 = None
380: (12)                 if func == 'compress':
381: (16)                     arr1 = arr1.ravel()
382: (16)                     res1 = getattr(arr2, func)(arr1)
383: (12)                 else:
384: (16)                     arr2 = (15*arr2).astype(int).ravel()
385: (12)                 if res1 is None:
386: (16)                     res1 = getattr(arr1, func)(arr2)
387: (12)                 res2 = getattr(np, func)(arr1, arr2)
388: (12)                 assert_(abs(res1-res2).max() < 1e-8, func)
389: (4)             def test_mem_lexsort_strings(self):
390: (8)                 lst = ['abc', 'cde', 'fgh']
391: (8)                 np.lexsort((lst,))
392: (4)             def test_fancy_index(self):
393: (8)                 x = np.array([1, 2])[np.array([0])]
394: (8)                 assert_equal(x.shape, (1,))
395: (4)             def test_reccarray_copy(self):
396: (8)                 dt = [('x', np.int16), ('y', np.float64)]
397: (8)                 ra = np.array([(1, 2.3)], dtype=dt)
398: (8)                 rb = np.rec.array(ra, dtype=dt)
399: (8)                 rb['x'] = 2.
400: (8)                 assert_(ra['x'] != rb['x'])
401: (4)             def test_rec_fromarray(self):
402: (8)                 x1 = np.array([[1, 2], [3, 4], [5, 6]])
403: (8)                 x2 = np.array(['a', 'dd', 'xyz'])
404: (8)                 x3 = np.array([1.1, 2, 3])
405: (8)                 np.rec.fromarrays([x1, x2, x3], formats="(2,)i4,a3,f8")
406: (4)             def test_object_array_assign(self):
407: (8)                 x = np.empty((2, 2), object)
408: (8)                 x.flat[2] = (1, 2, 3)
409: (8)                 assert_equal(x.flat[2], (1, 2, 3))
410: (4)             def test_ndmin_float64(self):
411: (8)                 x = np.array([1, 2, 3], dtype=np.float64)
412: (8)                 assert_equal(np.array(x, dtype=np.float32, ndmin=2).ndim, 2)
413: (8)                 assert_equal(np.array(x, dtype=np.float64, ndmin=2).ndim, 2)
414: (4)             def test_ndmin_order(self):
415: (8)                 assert_(np.array([1, 2], order='C', ndmin=3).flags.c_contiguous)
416: (8)                 assert_(np.array([1, 2], order='F', ndmin=3).flags.f_contiguous)
417: (8)                 assert_(np.array(np.ones((2, 2), order='F'),
418: (8)                               ndmin=3).flags.f_contiguous)
419: (4)                 assert_(np.array(np.ones((2, 2), order='C'),
420: (8)                               ndmin=3).flags.c_contiguous)
421: (8)             def test_mem_axis_minimization(self):
422: (8)                 data = np.arange(5)
423: (8)                 data = np.add.outer(data, data)
424: (4)             def test_mem_float_imag(self):
425: (8)                 np.float64(1.0).imag
426: (4)             def test_dtype_tuple(self):
427: (8)                 assert_(np.dtype('i4') == np.dtype(('i4', ())))
428: (4)             def test_numeric_carray_compare(self):
429: (8)                 assert_equal(np.array(['X'], 'c'), b'X')
430: (4)             def test_string_array_size(self):
431: (8)                 assert_raises(ValueError,
432: (30)                               np.array, [[['X']], ['X', 'X', 'X']], '|S1')
433: (4)             def test_dtype_repr(self):
434: (8)                 dt1 = np.dtype(('uint32', 2))
435: (8)                 dt2 = np.dtype(('uint32', (2,)))
436: (8)                 assert_equal(dt1.__repr__(), dt2.__repr__())

```

```

437: (4) def test_reshape_order(self):
438: (8)     a = np.arange(6).reshape(2, 3, order='F')
439: (8)     assert_equal(a, [[0, 2, 4], [1, 3, 5]])
440: (8)     a = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
441: (8)     b = a[:, 1]
442: (8)     assert_equal(b.reshape(2, 2, order='F'), [[2, 6], [4, 8]])
443: (4) def test_reshape_zero_strides(self):
444: (8)     a = np.ones(1)
445: (8)     a = np.lib.stride_tricks.as_strided(a, shape=(5,), strides=(0,))
446: (8)     assert_(a.reshape(5, 1).strides[0] == 0)
447: (4) def test_reshape_zero_size(self):
448: (8)     a = np.ones((0, 2))
449: (8)     a.shape = (-1, 2)
450: (4) @pytest.mark.skipif(np.ones(1).strides[0] == np.iinfo(np.intp).max,
451: (24)             reason="Using relaxed stride debug")
452: (4) def test_reshape_trailing_ones_strides(self):
453: (8)     a = np.zeros(12, dtype=np.int32)[::2] # not contiguous
454: (8)     strides_c = (16, 8, 8, 8)
455: (8)     strides_f = (8, 24, 48, 48)
456: (8)     assert_equal(a.reshape(3, 2, 1, 1).strides, strides_c)
457: (8)     assert_equal(a.reshape(3, 2, 1, 1, order='F').strides, strides_f)
458: (8)     assert_equal(np.array(0, dtype=np.int32).reshape(1, 1).strides, (4,
4))
459: (4) def test_repeat_discont(self):
460: (8)     a = np.arange(12).reshape(4, 3)[:, 2]
461: (8)     assert_equal(a.repeat(3), [2, 2, 2, 5, 5, 5, 8, 8, 8, 11, 11, 11])
462: (4) def test_array_index(self):
463: (8)     a = np.array([1, 2, 3])
464: (8)     a2 = np.array([[1, 2, 3]])
465: (8)     assert_equal(a[np.where(a == 3)], a2[np.where(a2 == 3)])
466: (4) def test_object_argmax(self):
467: (8)     a = np.array([1, 2, 3], dtype=object)
468: (8)     assert_(a.argmax() == 2)
469: (4) def test_recarray_fields(self):
470: (8)     dt0 = np.dtype([('f0', 'i4'), ('f1', 'i4')])
471: (8)     dt1 = np.dtype([('f0', 'i8'), ('f1', 'i8')])
472: (8)     for a in [np.array([(1, 2), (3, 4)], "i4,i4"),
473: (18)         np.rec.array([(1, 2), (3, 4)], "i4,i4"),
474: (18)         np.rec.array([(1, 2), (3, 4)]),
475: (18)         np.rec.fromarrays([(1, 2), (3, 4)], "i4,i4"),
476: (18)         np.rec.fromarrays([(1, 2), (3, 4)])]:
477: (12)         assert_(a.dtype in [dt0, dt1])
478: (4) def test_random_shuffle(self):
479: (8)     a = np.arange(5).reshape((5, 1))
480: (8)     b = a.copy()
481: (8)     np.random.shuffle(b)
482: (8)     assert_equal(np.sort(b, axis=0), a)
483: (4) def test_refcount_vdot(self):
484: (8)     _assert_valid_refcount(np.vdot)
485: (4) def test_startswith(self):
486: (8)     ca = np.char.array(['Hi', 'There'])
487: (8)     assert_equal(ca.startswith('H'), [True, False])
488: (4) def test_noncommutative_reduce_accumulate(self):
489: (8)     tosubtract = np.arange(5)
490: (8)     todive = np.array([2.0, 0.5, 0.25])
491: (8)     assert_equal(np.subtract.reduce(tosubtract), -10)
492: (8)     assert_equal(np.divide.reduce(todive), 16.0)
493: (8)     assert_array_equal(np.subtract.accumulate(tosubtract),
494: (12)         np.array([0, -1, -3, -6, -10]))
495: (8)     assert_array_equal(np.divide.accumulate(todive),
496: (12)         np.array([2., 4., 16.]))
497: (4) def test_convolve_empty(self):
498: (8)     assert_raises(ValueError, np.convolve, [], [1])
499: (8)     assert_raises(ValueError, np.convolve, [1], [])
500: (4) def test_multidim_byteswap(self):
501: (8)     r = np.array([(1, (0, 1, 2))], dtype="i2,3i2")
502: (8)     assert_array_equal(r.byteswap(),
503: (27)         np.array([(256, (0, 256, 512))], r.dtype))
504: (4) def test_string_NULL(self):

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

505: (8)
506: (21)
507: (4)
508: (8)
509: (8)
510: (4)
511: (8)
512: (8)
513: (8)
514: (8)
515: (8)
516: (4)
517: (8)
518: (8)
519: (8)
520: (12)
521: (8)
522: (12)
523: (8)
524: (12)
525: (8)
526: (12)
527: (4)
528: (8)
529: (8)
530: (12)
531: (4)
532: (8)
533: (8)
534: (8)
535: (8)
536: (8)
537: (28)
538: (4)
539: (8)
540: (8)
541: (8)
542: (4)
543: (8)
544: (8)
545: (4)
546: (8)
547: (8)
548: (4)
549: (8)
550: (8)
551: (4)
552: (8)
553: (12)
554: (4)
555: (8)
556: (8)
557: (8)
558: (4)
559: (8)
560: (8)
561: (4)
562: (8)
563: (8)
564: (8)
565: (8)
566: (8)
567: (4)
568: (8)
569: (12)
570: (12)
571: (12)
572: (8)
573: (4)

                        assert_equal(np.array("a\x00\x0b\x0c\x00").item(),
                                    'a\x00\x0b\x0c')
        def test_junk_in_string_fields_of_recarray(self):
            r = np.array([[b'abc']], dtype=[('var1', '|S20')])
            assert_(asbytes(r['var1'][0][0]) == b'abc')
        def test_take_output(self):
            x = np.arange(12).reshape((3, 4))
            a = np.take(x, [0, 2], axis=1)
            b = np.zeros_like(a)
            np.take(x, [0, 2], axis=1, out=b)
            assert_array_equal(a, b)
        def test_take_object_fail(self):
            d = 123.
            a = np.array([d, 1], dtype=object)
            if HAS_REFCOUNT:
                ref_d = sys.getrefcount(d)
            try:
                a.take([0, 100])
            except IndexError:
                pass
            if HAS_REFCOUNT:
                assert_(ref_d == sys.getrefcount(d))
        def test_array_str_64bit(self):
            s = np.array([1, np.nan], dtype=np.float64)
            with np.errstate(all='raise'):
                np.array_str(s) # Should succeed
        def test_frompyfunc_endian(self):
            from math import radians
            uradians = np.frompyfunc(radians, 1, 1)
            big_endian = np.array([83.4, 83.5], dtype='>f8')
            little_endian = np.array([83.4, 83.5], dtype='<f8')
            assert_almost_equal(uradians(big_endian).astype(float),
                                uradians(little_endian).astype(float))
        def test_mem_string_arr(self):
            s = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
            t = []
            np.hstack((t, s))
        def test_arr_transpose(self):
            x = np.random.rand(*2,)*16
            x.transpose(list(range(16))) # Should succeed
        def test_string_mergesort(self):
            x = np.array(['a'] * 32)
            assert_array_equal(x.argsort(kind='m'), np.arange(32))
        def test_argmax_byteorder(self):
            a = np.arange(3, dtype='>f')
            assert_(a[a.argmax()] == a.max())
        def test_rand_seed(self):
            for l in np.arange(4):
                np.random.seed(l)
        def test_mem_deallocation_leak(self):
            a = np.zeros(5, dtype=float)
            b = np.array(a, dtype=float)
            del a, b
        def test_mem_on_invalid_dtype(self):
            "Ticket #583"
            assert_raises(ValueError, np.fromiter, [['12', ''], ['13', '']], str)
        def test_dot_negative_stride(self):
            x = np.array([[1, 5, 25, 125., 625.]])
            y = np.array([[20.], [160.], [640.], [1280.], [1024.]])
            z = y[::-1].copy()
            y2 = y[::-1]
            assert_equal(np.dot(x, z), np.dot(x, y2))
        def test_object_casting(self):
            def rs():
                x = np.ones([484, 286])
                y = np.zeros([484, 286])
                x |= y
                assert_raises(TypeError, rs)
            def test_unicode_scalar(self):

```

```

574: (8)           x = np.array(["DROND", "DROND1"], dtype="U6")
575: (8)           el = x[1]
576: (8)           for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
577: (12)             new = pickle.loads(pickle.dumps(el, protocol=proto))
578: (12)             assert_equal(new, el)
579: (4)           def test_arange_non_native_dtype(self):
580: (8)             for T in ('>f4', '<f4'):
581: (12)               dt = np.dtype(T)
582: (12)               assert_equal(np.arange(0, dtype=dt).dtype, dt)
583: (12)               assert_equal(np.arange(0.5, dtype=dt).dtype, dt)
584: (12)               assert_equal(np.arange(5, dtype=dt).dtype, dt)
585: (4)           def test_bool_flat_indexing_invalid_nr_elements(self):
586: (8)             s = np.ones(10, dtype=float)
587: (8)             x = np.array((15,), dtype=float)
588: (8)             def ia(x, s, v):
589: (12)               x[(s > 0)] = v
590: (8)               assert_raises(IndexError, ia, x, s, np.zeros(9, dtype=float))
591: (8)               assert_raises(IndexError, ia, x, s, np.zeros(11, dtype=float))
592: (8)               assert_raises(ValueError, ia, x.flat, s, np.zeros(9, dtype=float))
593: (8)               assert_raises(ValueError, ia, x.flat, s, np.zeros(11, dtype=float))
594: (4)           def test_mem_scalar_indexing(self):
595: (8)             x = np.array([0], dtype=float)
596: (8)             index = np.array(0, dtype=np.int32)
597: (8)             x[index]
598: (4)           def test_binary_repr_0_width(self):
599: (8)             assert_equal(np.binary_repr(0, width=3), '000')
600: (4)           def test_fromstring(self):
601: (8)             assert_equal(np.fromstring("12:09:09", dtype=int, sep=":"), [12, 9, 9])
602: (21)           def test_searchsorted_variable_length(self):
603: (4)             x = np.array(['a', 'aa', 'b'])
604: (8)             y = np.array(['d', 'e'])
605: (8)             assert_equal(x.searchsorted(y), [3, 3])
606: (8)           def test_string_argsort_with_zeros(self):
607: (4)             x = np.frombuffer(b"\x00\x02\x00\x01", dtype="|S2")
608: (8)             assert_array_equal(x.argsort(kind='m'), np.array([1, 0]))
609: (8)             assert_array_equal(x.argsort(kind='q'), np.array([1, 0]))
610: (8)           def test_string_sort_with_zeros(self):
611: (4)             x = np.frombuffer(b"\x00\x02\x00\x01", dtype="|S2")
612: (8)             y = np.frombuffer(b"\x00\x01\x00\x02", dtype="|S2")
613: (8)             assert_array_equal(np.sort(x, kind="q"), y)
614: (8)           def test_copy_detection_zero_dim(self):
615: (4)             np.indices((0, 3, 4)).T.reshape(-1, 3)
616: (8)           def test_flat_byteorder(self):
617: (4)             x = np.arange(10)
618: (8)             assert_array_equal(x.astype('>i4'), x.astype('<i4').flat[:])
619: (8)             assert_array_equal(x.astype('>i4').flat[:], x.astype('<i4'))
620: (8)           def test_sign_bit(self):
621: (4)             x = np.array([0, -0.0, 0])
622: (8)             assert_equal(str(np.abs(x)), '[0. 0. 0.]')
623: (8)           def test_flat_index_byteswap(self):
624: (4)             for dt in (np.dtype('<i4'), np.dtype('>i4')):
625: (8)               x = np.array([-1, 0, 1], dtype=dt)
626: (12)               assert_equal(x.flat[0].dtype, x[0].dtype)
627: (12)           def test_copy_detection_corner_case(self):
628: (4)             np.indices((0, 3, 4)).T.reshape(-1, 3)
629: (8)           @pytest.mark.skipif(np.ones(1).strides[0] == np.iinfo(np.intp).max,
630: (4)                           reason="Using relaxed stride debug")
631: (24)           def test_copy_detection_corner_case2(self):
632: (4)             b = np.indices((0, 3, 4)).T.reshape(-1, 3)
633: (8)             assert_equal(b.strides, (3 * b.itemsize, b.itemsize))
634: (8)           def test_object_array_refcounting(self):
635: (4)             if not hasattr(sys, 'getrefcount'):
636: (8)               return
637: (12)             cnt = sys.getrefcount
638: (8)             a = object()
639: (8)             b = object()
640: (8)             c = object()
641: (8)             cnt0_a = cnt(a)

```

```

643: (8)             cnt0_b = cnt(b)
644: (8)             cnt0_c = cnt(c)
645: (8)             arr = np.zeros(5, dtype=np.object_)
646: (8)             arr[:] = a
647: (8)             assert_equal(cnt(a), cnt0_a + 5)
648: (8)             arr[:] = b
649: (8)             assert_equal(cnt(a), cnt0_a)
650: (8)             assert_equal(cnt(b), cnt0_b + 5)
651: (8)             arr[2:] = c
652: (8)             assert_equal(cnt(b), cnt0_b + 3)
653: (8)             assert_equal(cnt(c), cnt0_c + 2)
654: (8)             del arr
655: (8)             arr = np.zeros((5, 2), dtype=np.object_)
656: (8)             arr0 = np.zeros(2, dtype=np.object_)
657: (8)             arr0[0] = a
658: (8)             assert_(cnt(a) == cnt0_a + 1)
659: (8)             arr0[1] = b
660: (8)             assert_(cnt(b) == cnt0_b + 1)
661: (8)             arr[:, :] = arr0
662: (8)             assert_(cnt(a) == cnt0_a + 6)
663: (8)             assert_(cnt(b) == cnt0_b + 6)
664: (8)             arr[:, 0] = None
665: (8)             assert_(cnt(a) == cnt0_a + 1)
666: (8)             del arr, arr0
667: (8)             arr = np.zeros((5, 2), dtype=np.object_)
668: (8)             arr[:, 0] = a
669: (8)             arr[:, 1] = b
670: (8)             assert_(cnt(a) == cnt0_a + 5)
671: (8)             assert_(cnt(b) == cnt0_b + 5)
672: (8)             arr2 = arr.copy()
673: (8)             assert_(cnt(a) == cnt0_a + 10)
674: (8)             assert_(cnt(b) == cnt0_b + 10)
675: (8)             arr2 = arr[:, 0].copy()
676: (8)             assert_(cnt(a) == cnt0_a + 10)
677: (8)             assert_(cnt(b) == cnt0_b + 5)
678: (8)             arr2 = arr.flatten()
679: (8)             assert_(cnt(a) == cnt0_a + 10)
680: (8)             assert_(cnt(b) == cnt0_b + 10)
681: (8)             del arr, arr2
682: (8)             arr1 = np.zeros((5, 1), dtype=np.object_)
683: (8)             arr2 = np.zeros((5, 1), dtype=np.object_)
684: (8)             arr1[...] = a
685: (8)             arr2[...] = b
686: (8)             assert_(cnt(a) == cnt0_a + 5)
687: (8)             assert_(cnt(b) == cnt0_b + 5)
688: (8)             tmp = np.concatenate((arr1, arr2))
689: (8)             assert_(cnt(a) == cnt0_a + 5 + 5)
690: (8)             assert_(cnt(b) == cnt0_b + 5 + 5)
691: (8)             tmp = arr1.repeat(3, axis=0)
692: (8)             assert_(cnt(a) == cnt0_a + 5 + 3*5)
693: (8)             tmp = arr1.take([1, 2, 3], axis=0)
694: (8)             assert_(cnt(a) == cnt0_a + 5 + 3)
695: (8)             x = np.array([[0], [1], [0], [1], [1]], int)
696: (8)             tmp = x.choose(arr1, arr2)
697: (8)             assert_(cnt(a) == cnt0_a + 5 + 2)
698: (8)             assert_(cnt(b) == cnt0_b + 5 + 3)
699: (8)             del tmp # Avoid pyflakes unused variable warning
700: (4)             def test_mem_custom_float_to_array(self):
701: (8)                 class MyFloat:
702: (12)                     def __float__(self):
703: (16)                         return 1.0
704: (8)                         tmp = np.atleast_1d([MyFloat()])
705: (8)                         tmp.astype(float) # Should succeed
706: (4)             def test_object_array_refcount_self_assign(self):
707: (8)                 class VictimObject:
708: (12)                     deleted = False
709: (12)                     def __del__(self):
710: (16)                         self.deleted = True
711: (8)                         d = VictimObject()

```

```

712: (8)             arr = np.zeros(5, dtype=np.object_)
713: (8)             arr[:] = d
714: (8)             del d
715: (8)             arr[:] = arr # refcount of 'd' might hit zero here
716: (8)             assert_(not arr[0].deleted)
717: (8)             arr[:] = arr # trying to induce a segfault by doing it again...
718: (8)             assert_(not arr[0].deleted)
719: (4)             def test_mem_fromiter_invalid_dtype_string(self):
720: (8)                 x = [1, 2, 3]
721: (8)                 assert_raises(ValueError,
722: (30)                         np.fromiter, [xi for xi in x], dtype='S')
723: (4)             def test_reduce_big_object_array(self):
724: (8)                 oldsize = np.setbufsize(10*16)
725: (8)                 a = np.array([None]*161, object)
726: (8)                 assert_(not np.any(a))
727: (8)                 np.setbufsize(oldsize)
728: (4)             def test_mem_0d_array_index(self):
729: (8)                 np.zeros(10)[np.array(0)]
730: (4)             def test_nonnaive_endian_fill(self):
731: (8)                 if sys.byteorder == 'little':
732: (12)                     dtype = np.dtype('>i4')
733: (8)
734: (12)                     else:
735: (8)                         dtype = np.dtype('<i4')
736: (8)                     x = np.empty([1], dtype=dtype)
737: (8)                     x.fill(1)
738: (4)                     assert_equal(x, np.array([1], dtype=dtype))
739: (8)             def test_dot_alignment_sse2(self):
740: (8)                 x = np.zeros((30, 40))
741: (12)                 for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
742: (12)                     y = pickle.loads(pickle.dumps(x, protocol=proto))
743: (12)                     z = np.ones((1, y.shape[0]))
744: (4)                     np.dot(z, y)
745: (8)             def test_astype_copy(self):
746: (8)                 data_dir = path.join(path.dirname(__file__), 'data')
747: (8)                 filename = path.join(data_dir, "astype_copy.pkl")
748: (12)                 with open(filename, 'rb') as f:
749: (8)                     xp = pickle.load(f, encoding='latin1')
750: (8)                     xpd = xp.astype(np.float64)
751: (16)                     assert_((xp.__array_interface__['data'][0] !=
752: (4)                         xpd.__array_interface__['data'][0]))
753: (8)             def test_compress_small_type(self):
754: (8)                 import numpy as np
755: (8)                 a = np.array([[1, 2], [3, 4]])
756: (8)                 b = np.zeros((2, 1), dtype=np.single)
757: (12)                 try:
758: (12)                     a.compress([True, False], axis=1, out=b)
759: (33)                     raise AssertionError("compress with an out which cannot be "
760: (33)                             "safely casted should not return "
761: (8)                             "successfully")
762: (12)                 except TypeError:
763: (4)                     pass
764: (8)             def test_attributes(self):
765: (12)                 class TestArray(np.ndarray):
766: (16)                     def __new__(cls, data, info):
767: (16)                         result = np.array(data)
768: (16)                         result = result.view(cls)
769: (16)                         result.info = info
770: (12)                         return result
771: (16)                     def __array_finalize__(self, obj):
772: (8)                         self.info = getattr(obj, 'info', '')
773: (8)                     dat = TestArray([[1, 2, 3, 4], [5, 6, 7, 8]], 'jubba')
774: (8)                     assert_(dat.info == 'jubba')
775: (8)                     dat.resize((4, 2))
776: (8)                     assert_(dat.info == 'jubba')
777: (8)                     dat.sort()
778: (8)                     assert_(dat.info == 'jubba')
779: (8)                     dat.fill(2)
780: (8)                     assert_(dat.info == 'jubba')
780: (8)                     dat.put([2, 3, 4], [6, 3, 4])

```

```

781: (8) assert_(dat.info == 'jubba')
782: (8) dat.setfield(4, np.int32, 0)
783: (8) assert_(dat.info == 'jubba')
784: (8) dat.setflags()
785: (8) assert_(dat.info == 'jubba')
786: (8) assert_(dat.all(1).info == 'jubba')
787: (8) assert_(dat.any(1).info == 'jubba')
788: (8) assert_(dat.argmax(1).info == 'jubba')
789: (8) assert_(dat.argmin(1).info == 'jubba')
790: (8) assert_(dat.argsort(1).info == 'jubba')
791: (8) assert_(dat.astype(TestArray).info == 'jubba')
792: (8) assert_(dat.byteswap().info == 'jubba')
793: (8) assert_(dat.clip(2, 7).info == 'jubba')
794: (8) assert_(dat.compress([0, 1, 1]).info == 'jubba')
795: (8) assert_(dat.conj().info == 'jubba')
796: (8) assert_(dat.conjugate().info == 'jubba')
797: (8) assert_(dat.copy().info == 'jubba')
798: (8) dat2 = TestArray([2, 3, 1, 0], 'jubba')
799: (8) choices = [[0, 1, 2, 3], [10, 11, 12, 13],
800: (19) [20, 21, 22, 23], [30, 31, 32, 33]]
801: (8) assert_(dat2.choose(choices).info == 'jubba')
802: (8) assert_(dat.cumprod(1).info == 'jubba')
803: (8) assert_(dat.cumsum(1).info == 'jubba')
804: (8) assert_(dat.diagonal().info == 'jubba')
805: (8) assert_(dat.flatten().info == 'jubba')
806: (8) assert_(dat.getfield(np.int32, 0).info == 'jubba')
807: (8) assert_(dat.imag.info == 'jubba')
808: (8) assert_(dat.max(1).info == 'jubba')
809: (8) assert_(dat.mean(1).info == 'jubba')
810: (8) assert_(dat.min(1).info == 'jubba')
811: (8) assert_(dat.newbyteorder().info == 'jubba')
812: (8) assert_(dat.prod(1).info == 'jubba')
813: (8) assert_(dat.ptp(1).info == 'jubba')
814: (8) assert_(dat.ravel().info == 'jubba')
815: (8) assert_(dat.real.info == 'jubba')
816: (8) assert_(dat.repeat(2).info == 'jubba')
817: (8) assert_(dat.reshape((2, 4)).info == 'jubba')
818: (8) assert_(dat.round().info == 'jubba')
819: (8) assert_(dat.squeeze().info == 'jubba')
820: (8) assert_(dat.std(1).info == 'jubba')
821: (8) assert_(dat.sum(1).info == 'jubba')
822: (8) assert_(dat.swapaxes(0, 1).info == 'jubba')
823: (8) assert_(dat.take([2, 3, 5]).info == 'jubba')
824: (8) assert_(dat.transpose().info == 'jubba')
825: (8) assert_(dat.T.info == 'jubba')
826: (8) assert_(dat.var(1).info == 'jubba')
827: (8) assert_(dat.view(TestArray).info == 'jubba')
828: (8) assert_(type(dat.nonzero()[0]) is np.ndarray)
829: (8) assert_(type(dat.nonzero()[1]) is np.ndarray)
830: (4) def test_recarray_tolist(self):
831: (8) buf = np.zeros(40, dtype=np.int8)
832: (8) a = np.recarray(2, formats="i4,f8,f8", names="id,x,y", buf=buf)
833: (8) b = a.tolist()
834: (8) assert_(a[0].tolist() == b[0])
835: (8) assert_(a[1].tolist() == b[1])
836: (4) def test_nonscalar_item_method(self):
837: (8) a = np.arange(5)
838: (8) assert_raises(ValueError, a.item)
839: (4) def test_char_array_creation(self):
840: (8) a = np.array('123', dtype='c')
841: (8) b = np.array([b'1', b'2', b'3'])
842: (8) assert_equal(a, b)
843: (4) def test_unaligned_unicode_access(self):
844: (8) for i in range(1, 9):
845: (12)     msg = 'unicode offset: %d chars' % i
846: (12)     t = np.dtype([('a', 'S%d' % i), ('b', 'U2')])
847: (12)     x = np.array([(b'a', 'b')], dtype=t)
848: (12)     assert_equal(str(x), "[b'a', 'b']", err_msg=msg)
849: (4) def test_sign_for_complex_nan(self):

```

```

850: (8)
851: (12)
852: (12)
853: (12)
854: (12)
855: (4)
856: (8)
857: (8)
858: (8)
859: (8)
860: (8)
861: (8)
862: (8)
863: (4)
864: (8)
865: (8)
866: (8)
867: (8)
868: (8)
869: (8)
870: (8)
871: (8)
872: (8)
873: (8)
874: (8)
875: (4)
876: (8)
877: (8)
878: (4)
879: (8)
880: (8)
881: (8)
882: (4)
883: (8)
884: (8)
885: (4)
886: (8)
887: (8)
888: (8)
889: (8)
890: (8)
891: (4)
892: (8)
893: (8)
894: (12)
test_string.tobytes())
895: (8)
test_string)
896: (8)
897: (8)
898: (12)
899: (8)
900: (8)
901: (12)
902: (16)
903: (12)
904: (16)
905: (4)
906: (4)
907: (8)
908: (12)
909: (16)
910: (20)
911: (20)
912: (24)
913: (20)
914: (24)
915: (20)
916: (20)

        with np.errstate(invalid='ignore'):
            C = np.array([-np.inf, -2+1j, 0, 2-1j, np.inf, np.nan])
            have = np.sign(C)
            want = np.array([-1+0j, -1+0j, 0+0j, 1+0j, 1+0j, np.nan])
            assert_equal(have, want)

    def test_for_equal_names(self):
        dt = np.dtype([('foo', float), ('bar', float)])
        a = np.zeros(10, dt)
        b = list(a.dtype.names)
        b[0] = "notfoo"
        a.dtype.names = b
        assert_(a.dtype.names[0] == "notfoo")
        assert_(a.dtype.names[1] == "bar")

    def test_for_object_scalar_creation(self):
        a = np.object_()
        b = np.object_(3)
        b2 = np.object_(3.0)
        c = np.object_([4, 5])
        d = np.object_([None, {}, []])
        assert_(a is None)
        assert_(type(b) is int)
        assert_(type(b2) is float)
        assert_(type(c) is np.ndarray)
        assert_(c.dtype == object)
        assert_(d.dtype == object)

    def test_array_resize_method_system_error(self):
        x = np.array([[0, 1], [2, 3]])
        assert_raises(TypeError, x.resize, (2, 2), order='C')

    def test_for_zero_length_in_choose(self):
        "Ticket #882"
        a = np.array(1)
        assert_raises(ValueError, lambda x: x.choose([]), a)

    def test_array_ndmin_overflow(self):
        "Ticket #947."
        assert_raises(ValueError, lambda: np.array([1], ndmin=33))

    def test_void_scalar_with_titles(self):
        data = [('john', 4), ('mary', 5)]
        dtype1 =([('source:yy', 'name'), 'O'), ('source:xx', 'id'), int]
        arr = np.array(data, dtype=dtype1)
        assert_(arr[0][0] == 'john')
        assert_(arr[0][1] == 4)

    def test_void_scalar_constructor(self):
        test_string = np.array("test")
        test_string_void_scalar = np.core.multiarray.scalar(
            np.dtype("V", test_string.dtype.itemsize)),
        assert_(test_string_void_scalar.view(test_string.dtype) ==

        test_record = np.ones((), "i,i")
        test_record_void_scalar = np.core.multiarray.scalar(
            test_record.dtype, test_record.tobytes())
        assert_(test_record_void_scalar == test_record)
        for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
            assert_(pickle.loads(
                pickle.dumps(test_string, protocol=proto)) == test_string)
            assert_(pickle.loads(
                pickle.dumps(test_record, protocol=proto)) == test_record)

    @no_tracing
    def test_blasdot_uninitialized_memory(self):
        for m in [0, 1, 2]:
            for n in [0, 1, 2]:
                for k in range(3):
                    x = np.array([123456789e199], dtype=np.float64)
                    if IS_PYPY:
                        x.resize((m, 0), refcheck=False)
                    else:
                        x.resize((m, 0))
                    y = np.array([123456789e199], dtype=np.float64)
                    if IS_PYPY:

```

```

917: (24)           y.resize((0, n), refcheck=False)
918: (20)           else:
919: (24)             y.resize((0, n))
920: (20)             z = np.dot(x, y)
921: (20)             assert_(np.all(z == 0))
922: (20)             assert_(z.shape == (m, n))
923: (4)             def test_zeros(self):
924: (8)               sz = 2 ** 64
925: (8)               with assert_raises_regex(ValueError,
926: (33)                 'Maximum allowed dimension exceeded'):
927: (12)                 np.empty(sz)
928: (4)               def test_huge_arange(self):
929: (8)                 sz = 2 ** 64
930: (8)                 with assert_raises_regex(ValueError,
931: (33)                   'Maximum allowed size exceeded'):
932: (12)                     np.arange(sz)
933: (12)                     assert_(np.size == sz)
934: (4)               def test_fromiter_bytes(self):
935: (8)                 a = np.fromiter(list(range(10)), dtype='b')
936: (8)                 b = np.fromiter(list(range(10)), dtype='B')
937: (8)                 assert_(np.all(a == np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])))
938: (8)                 assert_(np.all(b == np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])))
939: (4)               def test_array_from_sequence_scalar_array(self):
940: (8)                 a = np.array((np.ones(2), np.array(2)), dtype=object)
941: (8)                 assert_equal(a.shape, (2,))
942: (8)                 assert_equal(a.dtype, np.dtype(object))
943: (8)                 assert_equal(a[0], np.ones(2))
944: (8)                 assert_equal(a[1], np.array(2))
945: (8)                 a = np.array(((1,), np.array(1)), dtype=object)
946: (8)                 assert_equal(a.shape, (2,))
947: (8)                 assert_equal(a.dtype, np.dtype(object))
948: (8)                 assert_equal(a[0], (1,))
949: (8)                 assert_equal(a[1], np.array(1))
950: (4)               def test_array_from_sequence_scalar_array2(self):
951: (8)                 t = np.array([np.array([]), np.array(0, object)], dtype=object)
952: (8)                 assert_equal(t.shape, (2,))
953: (8)                 assert_equal(t.dtype, np.dtype(object))
954: (4)               def test_array_too_big(self):
955: (8)                 assert_raises(ValueError, np.zeros, [975]*7, np.int8)
956: (8)                 assert_raises(ValueError, np.zeros, [26244]*5, np.int8)
957: (4)               def test_dtype_keyerrors_(self):
958: (8)                 dt = np.dtype([('f1', np.uint)])
959: (8)                 assert_raises(KeyError, dt.__getitem__, "f2")
960: (8)                 assert_raises(IndexError, dt.__getitem__, 1)
961: (8)                 assert_raises(TypeError, dt.__getitem__, 0.0)
962: (4)               def test_lexsort_buffer_length(self):
963: (8)                 a = np.ones(100, dtype=np.int8)
964: (8)                 b = np.ones(100, dtype=np.int32)
965: (8)                 i = np.lexsort((a[::-1], b))
966: (8)                 assert_equal(i, np.arange(100, dtype=int))
967: (4)               def test_object_array_to_fixed_string(self):
968: (8)                 a = np.array(['abcdefghijklmnopqrstuvwxyz'], dtype=np.object_)
969: (8)                 b = np.array(a, dtype=(np.str_, 8))
970: (8)                 assert_equal(a, b)
971: (8)                 c = np.array(a, dtype=(np.str_, 5))
972: (8)                 assert_equal(c, np.array(['abcde', 'ijklm']))
973: (8)                 d = np.array(a, dtype=(np.str_, 12))
974: (8)                 assert_equal(a, d)
975: (8)                 e = np.empty((2, ), dtype=(np.str_, 8))
976: (8)                 e[:] = a[:]
977: (8)                 assert_equal(a, e)
978: (4)               def test_unicode_to_string_cast(self):
979: (8)                 a = np.array([('abc', '\u03a3'), ('asdf', 'erw')], dtype='U')
980: (22)
981: (21)
982: (8)                 assert_raises(UnicodeEncodeError, np.array, a, 'S4')
983: (4)               def test_unicode_to_string_cast_error(self):
984: (8)                 a = np.array(['\x80'] * 129, dtype='U3')
985: (8)                 assert_raises(UnicodeEncodeError, np.array, a, 'S')

```

```

986: (8)          b = a.reshape(3, 43)[-1, :-1]
987: (8)          assert_raises(UnicodeEncodeError, np.array, b, 'S')
988: (4)          def test_mixed_string_byte_array_creation(self):
989: (8)              a = np.array(['1234', b'123'])
990: (8)              assert_(a.itemsize == 16)
991: (8)              a = np.array([b'123', '1234'])
992: (8)              assert_(a.itemsize == 16)
993: (8)              a = np.array(['1234', b'123', '12345'])
994: (8)              assert_(a.itemsize == 20)
995: (8)              a = np.array([b'123', '1234', b'12345'])
996: (8)              assert_(a.itemsize == 20)
997: (8)              a = np.array([b'123', '1234', b'1234'])
998: (8)              assert_(a.itemsize == 16)
999: (4)          def test_misaligned_objects_segfault(self):
1000: (8)             a1 = np.zeros((10,), dtype='O,c')
1001: (8)             a2 = np.array(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'],
1002: (8)               'S10')
1003: (8)             a1['f0'] = a2
1004: (8)             repr(a1)
1005: (8)             np.argmax(a1['f0'])
1006: (8)             a1['f0'][1] = "FOO"
1007: (8)             a1['f0'] = "FOO"
1008: (8)             np.array(a1['f0'], dtype='S')
1009: (8)             np.nonzero(a1['f0'])
1010: (8)             a1.sort()
1011: (4)          copy.deepcopy(a1)
1012: (8)          def test_misaligned_scalars_segfault(self):
1013: (8)              s1 = np.array(('a', 'Foo'), dtype='c,O')
1014: (8)              s2 = np.array(('b', 'Bar'), dtype='c,O')
1015: (8)              s1['f1'] = s2['f1']
1016: (4)          def test_misaligned_dot_product_objects(self):
1017: (8)              a = np.array([(1, 'a'), (0, 'a')], [(0, 'a'), (1, 'a')]),
1018: (8)              b = np.array([(4, 'a'), (1, 'a')], [(2, 'a'), (2, 'a')]),
1019: (8)              np.dot(a['f0'], b['f0'])
1020: (4)          def test_byteswap_complex_scalar(self):
1021: (8)              for dtype in [np.dtype('<+t') for t in np.typecodes['Complex']]:
1022: (12)                  z = np.array([2.2-1.1j], dtype)
1023: (12)                  x = z[0] # always native-endian
1024: (12)                  y = x.byteswap()
1025: (12)                  if x.dtype.byteorder == z.dtype.byteorder:
1026: (16)                      assert_equal(x, np.frombuffer(y.tobytes(),
1027: (12)                        dtype=dtype.newbyteorder()))
1028: (16)                  else:
1029: (12)                      assert_equal(x, np.frombuffer(y.tobytes(), dtype=dtype))
1030: (12)                      assert_equal(x.real, y.real.byteswap())
1031: (12)                      assert_equal(x.imag, y.imag.byteswap())
1032: (4)          def test_structured_arrays_with_objects1(self):
1033: (8)              stra = 'aaaa'
1034: (8)              strb = 'bbbb'
1035: (8)              x = np.array([(0, stra), (1, strb)]], 'i8,O')
1036: (8)              x[x.nonzero()] = x.ravel()[:1]
1037: (4)              assert_(x[0, 1] == x[0, 0])
1038: (8)              @pytest.mark.skipif(
1039: (8)                  sys.version_info >= (3, 12),
1040: (8)                  reason="Python 3.12 has immortal refcounts, this test no longer
1041: (4)                  works.")
1042: (4)          @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
1043: (4)          def test_structured_arrays_with_objects2(self):
1044: (8)              stra = 'aaaa'
1045: (8)              strb = 'bbbb'
1046: (8)              numb = sys.getrefcount(strb)
1047: (8)              numa = sys.getrefcount(stra)
1048: (8)              x = np.array([(0, stra), (1, strb)]], 'i8,O')
1049: (8)              x[x.nonzero()] = x.ravel()[:1]
1050: (8)              assert_(sys.getrefcount(strb) == numb)

```

```

1050: (8)             assert_(sys.getrefcount(stra) == numa + 2)
1051: (4)             def test_duplicate_title_and_name(self):
1052: (8)                 dtspec = [((‘a’, ‘a’), ‘i’), (‘b’, ‘i’)]
1053: (8)                 assert_raises(ValueError, np.dtype, dtspec)
1054: (4)             def test_signed_integer_division_overflow(self):
1055: (8)                 def test_type(t):
1056: (12)                     min = np.array([np.iinfo(t).min])
1057: (12)                     min //=-1
1058: (8)                     with np.errstate(over=“ignore”):
1059: (12)                         for t in (np.int8, np.int16, np.int32, np.int64, int):
1060: (16)                             test_type(t)
1061: (4)             def test_buffer_hashlib(self):
1062: (8)                 from hashlib import sha256
1063: (8)                 x = np.array([1, 2, 3], dtype=np.dtype(‘<i4’))
1064: (8)                 assert_equal(sha256(x).hexdigest(),
1065: (4) ‘4636993d3e1da4e9d6b8f87b79e8f7c6d018580d52661950eabc3845c5897a4d’)
1066: (4)             def test_0d_string_scalar(self):
1067: (8)                 np.asarray(‘x’, ‘>c’)
1068: (4)             def test_log1p_compiler_shenanigans(self):
1069: (8)                 assert_(np.isfinite(np.log1p(np.exp2(-53))))
1070: (4)             def test_fromiter_comparison(self):
1071: (8)                 a = np.fromiter(list(range(10)), dtype=‘b’)
1072: (8)                 b = np.fromiter(list(range(10)), dtype=‘B’)
1073: (8)                 assert_(np.all(a == np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])))
1074: (8)                 assert_(np.all(b == np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])))
1075: (4)             def test_fromstring_crash(self):
1076: (8)                 with assert_warnings(DeprecationWarning):
1077: (12)                     np.fromstring(b‘aa, aa, 1.0’, sep=‘,’)
1078: (4)             def test_ticket_1539(self):
1079: (8)                 dtypes = [x for x in np.sctypeDict.values()
1080: (18)                     if (issubclass(x, np.number)
1081: (22)                         and not issubclass(x, np.timedelta64))]
1082: (8)                 a = np.array([], np.bool_) # not x[0] because it is unordered
1083: (8)                 failures = []
1084: (8)                 for x in dtypes:
1085: (12)                     b = a.astype(x)
1086: (16)                     for y in dtypes:
1087: (16)                         c = a.astype(y)
1088: (20)                         try:
1089: (16)                             d = np.dot(b, c)
1090: (20)                         except TypeError:
1091: (16)                             failures.append((x, y))
1092: (20)                         else:
1093: (24)                             if d != 0:
1094: (8)                                 failures.append((x, y))
1095: (12)             if failures:
1096: (4)                 raise AssertionError(“Failures: %r” % failures)
1097: (4)             def test_ticket_1538(self):
1098: (8)                 x = np.finfo(np.float32)
1099: (8)                 for name in ‘eps epsneg max min resolution tiny’.split():
1100: (25)                     assert_equal(type(getattr(x, name)), np.float32,
1101: (4)                         err_msg=name)
1102: (8)             def test_ticket_1434(self):
1103: (8)                 data = np.array(((1, 2, 3), (4, 5, 6), (7, 8, 9)))
1104: (8)                 out = np.zeros((3,))
1105: (8)                 ret = data.var(axis=1, out=out)
1106: (8)                 assert_(ret is out)
1107: (8)                 assert_array_equal(ret, data.var(axis=1))
1108: (8)                 ret = data.std(axis=1, out=out)
1109: (8)                 assert_(ret is out)
1110: (8)                 assert_array_equal(ret, data.std(axis=1))
1111: (4)             def test_complex_nan_maximum(self):
1112: (8)                 cnan = complex(0, np.nan)
1113: (8)                 assert_equal(np.maximum(1, cnan), cnan)
1114: (4)             def test_subclass_int_tuple_assignment(self):
1115: (8)                 class Subclass(np.ndarray):
1116: (12)                     def __new__(cls, i):
1117: (16)                         return np.ones((i,)).view(cls)
1118: (8)                 x = Subclass(5)

```

```

1118: (8)           x[(0,)] = 2 # shouldn't raise an exception
1119: (8)           assert_equal(x[0], 2)
1120: (4)           def test_ufunc_no_unnecessary_views(self):
1121: (8)               class Subclass(np.ndarray):
1122: (12)                   pass
1123: (8)                   x = np.array([1, 2, 3]).view(Subclass)
1124: (8)                   y = np.add(x, x, x)
1125: (8)                   assert_equal(id(x), id(y))
1126: (4)           @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
1127: (4)           def test_take_refcount(self):
1128: (8)               a = np.arange(16, dtype=float)
1129: (8)               a.shape = (4, 4)
1130: (8)               lut = np.ones((5 + 3, 4), float)
1131: (8)               rgba = np.empty(shape=a.shape + (4,), dtype=lut.dtype)
1132: (8)               c1 = sys.getrefcount(rgba)
1133: (8)               try:
1134: (12)                   lut.take(a, axis=0, mode='clip', out=rgba)
1135: (8)               except TypeError:
1136: (12)                   pass
1137: (8)               c2 = sys.getrefcount(rgba)
1138: (8)               assert_equal(c1, c2)
1139: (4)           def test_fromfile_tofile_seeks(self):
1140: (8)               f0 = tempfile.NamedTemporaryFile()
1141: (8)               f = f0.file
1142: (8)               f.write(np.arange(255, dtype='u1').tobytes())
1143: (8)               f.seek(20)
1144: (8)               ret = np.fromfile(f, count=4, dtype='u1')
1145: (8)               assert_equal(ret, np.array([20, 21, 22, 23], dtype='u1'))
1146: (8)               assert_equal(f.tell(), 24)
1147: (8)               f.seek(40)
1148: (8)               np.array([1, 2, 3], dtype='u1').tofile(f)
1149: (8)               assert_equal(f.tell(), 43)
1150: (8)               f.seek(40)
1151: (8)               data = f.read(3)
1152: (8)               assert_equal(data, b"\x01\x02\x03")
1153: (8)               f.seek(80)
1154: (8)               f.read(4)
1155: (8)               data = np.fromfile(f, dtype='u1', count=4)
1156: (8)               assert_equal(data, np.array([84, 85, 86, 87], dtype='u1'))
1157: (8)               f.close()
1158: (4)           def test_complex_scalar_warning(self):
1159: (8)               for tp in [np.csingle, np.cdouble, np.clongdouble]:
1160: (12)                   x = tp(1+2j)
1161: (12)                   assert_warns(np.ComplexWarning, float, x)
1162: (12)                   with suppress_warnings() as sup:
1163: (16)                       sup.filter(np.ComplexWarning)
1164: (16)                       assert_equal(float(x), float(x.real))
1165: (4)           def test_complex_scalar_complex_cast(self):
1166: (8)               for tp in [np.csingle, np.cdouble, np.clongdouble]:
1167: (12)                   x = tp(1+2j)
1168: (12)                   assert_equal(complex(x), 1+2j)
1169: (4)           def test_complex_boolean_cast(self):
1170: (8)               for tp in [np.csingle, np.cdouble, np.clongdouble]:
1171: (12)                   x = np.array([0, 0+0.5j, 0.5+0j], dtype=tp)
1172: (12)                   assert_equal(x.astype(bool), np.array([0, 1, 1], dtype=bool))
1173: (12)                   assert_(np.any(x))
1174: (12)                   assert_(np.all(x[1:])))
1175: (4)           def test_uint_int_conversion(self):
1176: (8)               x = 2**64 - 1
1177: (8)               assert_equal(int(np.uint64(x)), x)
1178: (4)           def test_duplicate_field_names_assign(self):
1179: (8)               ra = np.fromiter(((i*3, i*2) for i in range(10)), dtype='i8,f8')
1180: (8)               ra.dtype.names = ('f1', 'f2')
1181: (8)               repr(ra) # should not cause a segmentation fault
1182: (8)               assert_raises(ValueError, setattr, ra.dtype, 'names', ('f1', 'f1'))
1183: (4)           def test_eq_string_and_object_array(self):
1184: (8)               a1 = np.array(['a', 'b'], dtype=object)
1185: (8)               a2 = np.array(['a', 'c'])
1186: (8)               assert_array_equal(a1 == a2, [True, False])

```

```

1187: (8)             assert_array_equal(a2 == a1, [True, False])
1188: (4)             def test_nonzero_byteswap(self):
1189: (8)                 a = np.array([0x80000000, 0x00000080, 0], dtype=np.uint32)
1190: (8)                 a.dtype = np.float32
1191: (8)                 assert_equal(a.nonzero()[0], [1])
1192: (8)                 a = a.byteswap().newbyteorder()
1193: (8)                 assert_equal(a.nonzero()[0], [1]) # [0] if nonzero() ignores swap
1194: (4)             def test_find_common_type_boolean(self):
1195: (8)                 with pytest.warns(DeprecationWarning, match="np.find_common_type"):
1196: (12)                     res = np.find_common_type([], ['?', '?'])
1197: (8)                     assert res == '?'
1198: (4)             def test_empty_mul(self):
1199: (8)                 a = np.array([1.])
1200: (8)                 a[1:1] *= 2
1201: (8)                 assert_equal(a, [1.])
1202: (4)             def test_array_side_effect(self):
1203: (8)                 assert_equal(np.dtype('S10').itemsize, 10)
1204: (8)                 np.array(['abc', 2], ['long ', '0123456789']), dtype=np.bytes_)
1205: (8)                 assert_equal(np.dtype('S10').itemsize, 10)
1206: (4)             def test_any_float(self):
1207: (8)                 a = np.array([0.1, 0.9])
1208: (8)                 assert_(np.any(a))
1209: (8)                 assert_(np.all(a))
1210: (4)             def test_large_float_sum(self):
1211: (8)                 a = np.arange(10000, dtype='f')
1212: (8)                 assert_equal(a.sum(dtype='d'), a.astype('d').sum())
1213: (4)             def test_ufunc_casting_out(self):
1214: (8)                 a = np.array(1.0, dtype=np.float32)
1215: (8)                 b = np.array(1.0, dtype=np.float64)
1216: (8)                 c = np.array(1.0, dtype=np.float32)
1217: (8)                 np.add(a, b, out=c)
1218: (8)                 assert_equal(c, 2.0)
1219: (4)             def test_array_scalar_contiguous(self):
1220: (8)                 assert_(np.array(1.0).flags.c_contiguous)
1221: (8)                 assert_(np.array(1.0).flags.f_contiguous)
1222: (8)                 assert_(np.array(np.float32(1.0)).flags.c_contiguous)
1223: (8)                 assert_(np.array(np.float32(1.0)).flags.f_contiguous)
1224: (4)             def test_squeeze_contiguous(self):
1225: (8)                 a = np.zeros((1, 2)).squeeze()
1226: (8)                 b = np.zeros((2, 2, 2), order='F')[ :, :, ::2].squeeze()
1227: (8)                 assert_(a.flags.c_contiguous)
1228: (8)                 assert_(a.flags.f_contiguous)
1229: (8)                 assert_(b.flags.f_contiguous)
1230: (4)             def test_squeeze_axis_handling(self):
1231: (8)                 class OldSqueeze(np.ndarray):
1232: (12)                     def __new__(cls,
1233: (24)                         input_array):
1234: (16)                         obj = np.asarray(input_array).view(cls)
1235: (16)                         return obj
1236: (12)                     def squeeze(self):
1237: (16)                         return super().squeeze()
1238: (8)                     oldsqueeze = OldSqueeze(np.array([[1],[2],[3]]))
1239: (8)                     assert_equal(np.squeeze(oldsqueeze),
1240: (21)                         np.array([1,2,3]))
1241: (8)                     assert_equal(np.squeeze(oldsqueeze, axis=None),
1242: (21)                         np.array([1,2,3]))
1243: (8)                     with assert_raises(TypeError):
1244: (12)                         np.squeeze(oldsqueeze, axis=1)
1245: (8)                     with assert_raises(TypeError):
1246: (12)                         np.squeeze(oldsqueeze, axis=0)
1247: (8)                     with assert_raises(ValueError):
1248: (12)                         np.squeeze(np.array([[1],[2],[3]]), axis=0)
1249: (4)             def test_reduce_contiguous(self):
1250: (8)                 a = np.add.reduce(np.zeros((2, 1, 2)), (0, 1))
1251: (8)                 b = np.add.reduce(np.zeros((2, 1, 2)), 1)
1252: (8)                 assert_(a.flags.c_contiguous)
1253: (8)                 assert_(a.flags.f_contiguous)
1254: (8)                 assert_(b.flags.c_contiguous)
1255: (4)                 @pytest.mark.skipif(IS_PYSTON, reason="Pyston disables recursion"

```

```

    checking")
1256: (4)         def test_object_array_self_reference(self):
1257: (8)             a = np.array(0, dtype=object)
1258: (8)             a[()] = a
1259: (8)             assert_raises(RecursionError, int, a)
1260: (8)             assert_raises(RecursionError, float, a)
1261: (8)             a[()] = None
1262: (4)             @pytest.mark.skipif(IS_PYSTON, reason="Pyston disables recursion
    checking")
1263: (4)         def test_object_array_circular_reference(self):
1264: (8)             a = np.array(0, dtype=object)
1265: (8)             b = np.array(0, dtype=object)
1266: (8)             a[()] = b
1267: (8)             b[()] = a
1268: (8)             assert_raises(RecursionError, int, a)
1269: (8)             a[()] = None
1270: (8)             a = np.array(0, dtype=object)
1271: (8)             a[...] += 1
1272: (8)             assert_equal(a, 1)
1273: (4)         def test_object_array_nested(self):
1274: (8)             a = np.array(0, dtype=object)
1275: (8)             b = np.array(0, dtype=object)
1276: (8)             a[()] = b
1277: (8)             assert_equal(int(a), int(0))
1278: (8)             assert_equal(float(a), float(0))
1279: (4)         def test_object_array_self_copy(self):
1280: (8)             a = np.array(object(), dtype=object)
1281: (8)             np.copyto(a, a)
1282: (8)             if HAS_REFCOUNT:
1283: (12)                 assert_(sys.getrefcount(a[()]) == 2)
1284: (8)             a[()].__class__ # will segfault if object was deleted
1285: (4)         def test_zerosize_accumulate(self):
1286: (8)             "Ticket #1733"
1287: (8)             x = np.array([[42, 0]], dtype=np.uint32)
1288: (8)             assert_equal(np.add.accumulate(x[:-1, 0]), [])
1289: (4)         def test_objectarray_setfield(self):
1290: (8)             x = np.array([1, 2, 3], dtype=object)
1291: (8)             assert_raises(TypeError, x.setfield, 4, np.int32, 0)
1292: (4)         def test_setting_rank0_string(self):
1293: (8)             "Ticket #1736"
1294: (8)             s1 = b"hello1"
1295: (8)             s2 = b"hello2"
1296: (8)             a = np.zeros((), dtype="S10")
1297: (8)             a[()] = s1
1298: (8)             assert_equal(a, np.array(s1))
1299: (8)             a[()] = np.array(s2)
1300: (8)             assert_equal(a, np.array(s2))
1301: (8)             a = np.zeros((), dtype='f4')
1302: (8)             a[()] = 3
1303: (8)             assert_equal(a, np.array(3))
1304: (8)             a[()] = np.array(4)
1305: (8)             assert_equal(a, np.array(4))
1306: (4)         def test_string_astype(self):
1307: (8)             "Ticket #1748"
1308: (8)             s1 = b'black'
1309: (8)             s2 = b'white'
1310: (8)             s3 = b'other'
1311: (8)             a = np.array([[s1], [s2], [s3]])
1312: (8)             assert_equal(a.dtype, np.dtype('S5'))
1313: (8)             b = a.astype(np.dtype('S0'))
1314: (8)             assert_equal(b.dtype, np.dtype('S5'))
1315: (4)         def test_ticket_1756(self):
1316: (8)             s = b'0123456789abcdef'
1317: (8)             a = np.array([s]*5)
1318: (8)             for i in range(1, 17):
1319: (12)                 a1 = np.array(a, "|%d" % i)
1320: (12)                 a2 = np.array([s[:i]]*5)
1321: (12)                 assert_equal(a1, a2)
1322: (4)             def test_fields_strides(self):

```

```

1323: (8)           "gh-2355"
1324: (8)           r = np.frombuffer(b'abcdefghijklmnp'*4*3, dtype='i4,(2,3)u2')
1325: (8)           assert_equal(r[0:3:2]['f1'], r['f1'][0:3:2])
1326: (8)           assert_equal(r[0:3:2]['f1'][0], r[0:3:2][0]['f1'])
1327: (8)           assert_equal(r[0:3:2]['f1'][0][()], r[0:3:2][0]['f1'][()])
1328: (8)           assert_equal(r[0:3:2]['f1'][0].strides, r[0:3:2][0]['f1'].strides)
1329: (4)           def test_alignment_update(self):
1330: (8)             a = np.arange(10)
1331: (8)             assert_(a.flags.aligned)
1332: (8)             a.strides = 3
1333: (8)             assert_(not a.flags.aligned)
1334: (4)           def test_ticket_1770(self):
1335: (8)             "Should not segfault on python 3k"
1336: (8)             import numpy as np
1337: (8)             try:
1338: (12)               a = np.zeros((1,), dtype=[('f1', 'f')])
1339: (12)               a['f1'] = 1
1340: (12)               a['f2'] = 1
1341: (8)             except ValueError:
1342: (12)               pass
1343: (8)             except Exception:
1344: (12)               raise AssertionError
1345: (4)           def test_ticket_1608(self):
1346: (8)             "x.flat shouldn't modify data"
1347: (8)             x = np.array([[1, 2], [3, 4]]).T
1348: (8)             np.array(x.flat)
1349: (8)             assert_equal(x, [[1, 3], [2, 4]])
1350: (4)           def test_pickle_string_overwrite(self):
1351: (8)             import re
1352: (8)             data = np.array([1], dtype='b')
1353: (8)             blob = pickle.dumps(data, protocol=1)
1354: (8)             data = pickle.loads(blob)
1355: (8)             s = re.sub("a(.)", "\x01\1", "a_")
1356: (8)             assert_equal(s[0], "\x01")
1357: (8)             data[0] = 0x6a
1358: (8)             s = re.sub("a(.)", "\x01\1", "a_")
1359: (8)             assert_equal(s[0], "\x01")
1360: (4)           def test_pickle_bytes_overwrite(self):
1361: (8)             for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
1362: (12)               data = np.array([1], dtype='b')
1363: (12)               data = pickle.loads(pickle.dumps(data, protocol=proto))
1364: (12)               data[0] = 0x7d
1365: (12)               bytestring = "\x01 ".encode('ascii')
1366: (12)               assert_equal(bytestring[0:1], '\x01'.encode('ascii'))
1367: (4)           def test_pickle_py2_array_latin1_hack(self):
1368: (8)             data =
(b"numpy.core.multiarray\n_reconstruct\np0\n(cnumpy\nndarray\np1\n(I0\n"
1369: (16)
b"tp2\nS'b'\nnp3\nntp4\nRp5\n(I1\n(I1\nntp6\nncnumpy\nndtype\np7\n(S'i1'\nnp8\n"
1370: (16)           b"I0\nI1\nntp9\nRp10\n(I3\nS'|'\nnp11\nNNNI-1\nI-
1\nI0\nntp12\nbI00\nS'\\x81'\n"
1371: (16)           b"p13\nntp14\nb."
1372: (8)             result = pickle.loads(data, encoding='latin1')
1373: (8)             assert_array_equal(result, np.array([129]).astype('b'))
1374: (8)             assert_raises(Exception, pickle.loads, data, encoding='koi8-r')
1375: (4)           def test_pickle_py2_scalar_latin1_hack(self):
1376: (8)             datas = [
1377: (12)               (np.str_('u6bd2'),
1378: (13)                 (b"numpy.core.multiarray\nscalar\np0\n(cnumpy\nndtype\np1\n"
1379: (14)                   b"(S'U1'\nnp2\nI0\nI1\nntp3\nRp4\n(I3\nS'<'\nnp5\nNNNI4\nI4\nI0\n"
1380: (14)                   b"tp6\nbS'\\xd2k\\x00\\x00'\nnp7\nntp8\nRp9\n.",),
1381: (13)                   'invalid'),
1382: (12)                   (np.float64(9e123),
1383: (13)
(b"numpy.core.multiarray\nscalar\np0\n(cnumpy\nndtype\np1\n(S'f8'\n"
1384: (14)           b"p2\nI0\nI1\nntp3\nRp4\n(I3\nS'<'\nnp5\nNNNI-1\nI-1\nI0\nntp6\n"
1385: (14)           b"bS'0'\\x81\\xb7Z\\xaa:\\xabY'\nnp7\nntp8\nRp9\n.",),
1386: (13)           'invalid'),
1387: (12)           (np.bytes_(b'\\x9c'), # different 8-bit code point in KOI8-R vs

```

```

latin1
1388: (13)
(b"numpy.core.multiarray\nscalar\np0\n(cnumpy\nndtype\\np1\\n(S'S1'\\np2\\n"
1389: (14)
b"I0\\nI1\\ntp3\\nRp4\\n(I3\\nS' | '\\np5\\nNNNI1\\nI1\\nI0\\ntp6\\nbs'\\x9c'\\np7\\n"
1390: (14)
b"tp8\\nRp9\\n."),
1391: (13)
'different'),
1392: (8)
1393: (8)
for original, data, koi8r_validity in datas:
1394: (12)
    result = pickle.loads(data, encoding='latin1')
1395: (12)
    assert_equal(result, original)
1396: (12)
    if koi8r_validity == 'different':
1397: (16)
        result = pickle.loads(data, encoding='koi8-r')
1398: (16)
        assert_(result != original)
1399: (12)
    elif koi8r_validity == 'invalid':
1400: (16)
        assert_raises(ValueError, pickle.loads, data, encoding='koi8-
r')
1401: (12)
    else:
        raise ValueError(koi8r_validity)
1402: (16)
1403: (4)
def test_structured_type_to_object(self):
1404: (8)
a_rec = np.array([(0, 1), (3, 2)], dtype='i4,i8')
1405: (8)
a_obj = np.empty((2,), dtype=object)
1406: (8)
a_obj[0] = (0, 1)
1407: (8)
a_obj[1] = (3, 2)
1408: (8)
assert_equal(a_rec.astype(object), a_obj)
1409: (8)
b = np.empty_like(a_obj)
1410: (8)
b[...] = a_rec
1411: (8)
assert_equal(b, a_obj)
1412: (8)
b = np.empty_like(a_rec)
1413: (8)
b[...] = a_obj
1414: (8)
assert_equal(b, a_rec)
1415: (4)
def test_assign_obj_listoflists(self):
1416: (8)
a = np.zeros(4, dtype=object)
1417: (8)
b = a.copy()
1418: (8)
a[0] = [1]
1419: (8)
a[1] = [2]
1420: (8)
a[2] = [3]
1421: (8)
a[3] = [4]
1422: (8)
b[...] = [[1], [2], [3], [4]]
1423: (8)
assert_equal(a, b)
1424: (8)
a = np.zeros((2, 2), dtype=object)
1425: (8)
a[...] = [[1, 2]]
1426: (8)
assert_equal(a, [[1, 2], [1, 2]])
1427: (4)
@pytest.mark.slow_pypy
1428: (4)
def test_memoryleak(self):
1429: (8)
    for i in range(1000):
1430: (12)
        a = np.empty((100000000,), dtype='i1')
1431: (12)
        del a
1432: (4)
@pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
1433: (4)
def test_ufunc_reduce_memoryleak(self):
1434: (8)
a = np.arange(6)
1435: (8)
acnt = sys.getrefcount(a)
1436: (8)
np.add.reduce(a)
1437: (8)
assert_equal(sys.getrefcount(a), acnt)
1438: (4)
def test_search_sorted_invalid_arguments(self):
1439: (8)
x = np.arange(0, 4, dtype='datetime64[D]')
1440: (8)
assert_raises(TypeError, x.searchsorted, 1)
1441: (4)
def test_string_truncation(self):
1442: (8)
    for val in [True, 1234, 123.4, complex(1, 234)]:
1443: (12)
        for tostr, dtype in [(asunicode, "U"), (asbytes, "S")]:
1444: (16)
            b = np.array([val, tostr('xx')], dtype=dtype)
1445: (16)
            assert_equal(tostr(b[0]), tostr(val))
1446: (16)
            b = np.array([tostr('xx'), val], dtype=dtype)
1447: (16)
            assert_equal(tostr(b[1]), tostr(val))
1448: (16)
            b = np.array([val, tostr('xxxxxxxxxx')], dtype=dtype)
1449: (16)
            assert_equal(tostr(b[0]), tostr(val))
1450: (16)
            b = np.array([tostr('xxxxxxxxxx'), val], dtype=dtype)
1451: (16)
            assert_equal(tostr(b[1]), tostr(val))
1452: (4)
def test_string_truncation_ucs2(self):

```

```

1453: (8)             a = np.array(['abcd'])
1454: (8)             assert_equal(a.dtype.itemsize, 16)
1455: (4)             def test_unique_stable(self):
1456: (8)                 v = np.array(([0]*5 + [1]*6 + [2]*6)*4)
1457: (8)                 res = np.unique(v, return_index=True)
1458: (8)                 tgt = (np.array([0, 1, 2]), np.array([ 0,  5, 11]))
1459: (8)                 assert_equal(res, tgt)
1460: (4)             def test_unicode_alloc_dealloc_match(self):
1461: (8)                 a = np.array(['abc'], dtype=np.str_)[0]
1462: (8)                 del a
1463: (4)             def test_refcount_error_in_clip(self):
1464: (8)                 a = np.zeros((2,), dtype='>i2').clip(min=0)
1465: (8)                 x = a + a
1466: (8)                 y = str(x)
1467: (8)                 assert_(y == "[0 0]")
1468: (4)             def test_searchsorted_wrong_dtype(self):
1469: (8)                 a = np.array([('a', 1)], dtype='S1, int')
1470: (8)                 assert_raises(TypeError, np.searchsorted, a, 1.2)
1471: (8)                 dtype = np.format_parser(['i4', 'i4'], [], [])
1472: (8)                 a = np.recarray((2,), dtype)
1473: (8)                 a[...] = [(1, 2), (3, 4)]
1474: (8)                 assert_raises(TypeError, np.searchsorted, a, 1)
1475: (4)             def test_complex64_alignment(self):
1476: (8)                 dtt = np.complex64
1477: (8)                 arr = np.arange(10, dtype=dtt)
1478: (8)                 arr2 = np.reshape(arr, (2, 5))
1479: (8)                 data_str = arr2.tobytes('F')
1480: (8)                 data_back = np.ndarray(arr2.shape,
1481: (30)                               arr2.dtype,
1482: (30)                               buffer=data_str,
1483: (30)                               order='F')
1484: (8)                 assert_array_equal(arr2, data_back)
1485: (4)             def test_structured_count_nonzero(self):
1486: (8)                 arr = np.array([0, 1]).astype('i4, (2)i4')[:1]
1487: (8)                 count = np.count_nonzero(arr)
1488: (8)                 assert_equal(count, 0)
1489: (4)             def test_copymodule_preserves_f_contiguity(self):
1490: (8)                 a = np.empty((2, 2), order='F')
1491: (8)                 b = copy.copy(a)
1492: (8)                 c = copy.deepcopy(a)
1493: (8)                 assert_(b.flags.fortran)
1494: (8)                 assert_(b.flags.f_contiguous)
1495: (8)                 assert_(c.flags.fortran)
1496: (8)                 assert_(c.flags.f_contiguous)
1497: (4)             def test_fortran_order_buffer(self):
1498: (8)                 import numpy as np
1499: (8)                 a = np.array([['Hello', 'Foob']], dtype='U5', order='F')
1500: (8)                 arr = np.ndarray(shape=[1, 2, 5], dtype='U1', buffer=a)
1501: (8)                 arr2 = np.array([[[['H', 'e', 'l', 'l', 'o'],
1502: (26)                               ['F', 'o', 'o', 'b', '']]],
1503: (8)                               assert_array_equal(arr, arr2)
1504: (4)             def test_assign_from_sequence_error(self):
1505: (8)                 arr = np.array([1, 2, 3])
1506: (8)                 assert_raises(ValueError, arr.__setitem__, slice(None), [9, 9])
1507: (8)                 arr.__setitem__(slice(None), [9])
1508: (8)                 assert_equal(arr, [9, 9, 9])
1509: (4)             def test_format_on_flex_array_element(self):
1510: (8)                 dt = np.dtype([('date', '<M8[D)'), ('val', '<f8')])
1511: (8)                 arr = np.array([('2000-01-01', 1)], dt)
1512: (8)                 formatted = '{0}'.format(arr[0])
1513: (8)                 assert_equal(formatted, str(arr[0]))
1514: (4)             def test_deepcopy_on_0d_array(self):
1515: (8)                 arr = np.array(3)
1516: (8)                 arr_cp = copy.deepcopy(arr)
1517: (8)                 assert_equal(arr, arr_cp)
1518: (8)                 assert_equal(arr.shape, arr_cp.shape)
1519: (8)                 assert_equal(int(arr), int(arr_cp))
1520: (8)                 assert_(arr is not arr_cp)
1521: (8)                 assert_(isinstance(arr_cp, type(arr)))

```

```

1522: (4) def test_deepcopy_F_order_object_array(self):
1523: (8)     a = {'a': 1}
1524: (8)     b = {'b': 2}
1525: (8)     arr = np.array([[a, b], [a, b]], order='F')
1526: (8)     arr_cp = copy.deepcopy(arr)
1527: (8)     assert_equal(arr, arr_cp)
1528: (8)     assert_(arr is not arr_cp)
1529: (8)     assert_(arr[0, 1] is not arr_cp[1, 1])
1530: (8)     assert_(arr[0, 1] is arr[1, 1])
1531: (8)     assert_(arr_cp[0, 1] is arr_cp[1, 1])
1532: (4) def test_deepcopy_empty_object_array(self):
1533: (8)     a = np.array([], dtype=object)
1534: (8)     b = copy.deepcopy(a)
1535: (8)     assert_(a.shape == b.shape)
1536: (4) def test_bool_subscript_crash(self):
1537: (8)     c = np.rec.array([(1, 2, 3), (4, 5, 6)])
1538: (8)     masked = c[np.array([True, False])]
1539: (8)     base = masked.base
1540: (8)     del masked, c
1541: (8)     base.dtype
1542: (4) def test_richcompare_crash(self):
1543: (8)     import operator as op
1544: (8)     class Foo:
1545: (12)         __array_priority__ = 1002
1546: (12)         def __array__(self, *args, **kwargs):
1547: (16)             raise Exception()
1548: (8)     rhs = Foo()
1549: (8)     lhs = np.array(1)
1550: (8)     for f in [op.lt, op.le, op.gt, op.ge]:
1551: (12)         assert_raises(TypeError, f, lhs, rhs)
1552: (8)     assert_(not op.eq(lhs, rhs))
1553: (8)     assert_(op.ne(lhs, rhs))
1554: (4) def test_richcompare_scalar_and_subclass(self):
1555: (8)     class Foo(np.ndarray):
1556: (12)         def __eq__(self, other):
1557: (16)             return "OK"
1558: (8)     x = np.array([1, 2, 3]).view(Foo)
1559: (8)     assert_equal(10 == x, "OK")
1560: (8)     assert_equal(np.int32(10) == x, "OK")
1561: (8)     assert_equal(np.array([10]) == x, "OK")
1562: (4) def test_pickle_empty_string(self):
1563: (8)     for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
1564: (12)         test_string = np.bytes_()
1565: (12)         assert_equal(pickle.loads(
1566: (16)             pickle.dumps(test_string, protocol=proto)), test_string)
1567: (4) def test_frompyfunc_many_args(self):
1568: (8)     def passer(*args):
1569: (12)         pass
1570: (8)     assert_raises(ValueError, np.frompyfunc, passer, 32, 1)
1571: (4) def test_repeat_broadcasting(self):
1572: (8)     a = np.arange(60).reshape(3, 4, 5)
1573: (8)     for axis in chain(range(-a.ndim, a.ndim), [None]):
1574: (12)         assert_equal(a.repeat(2, axis=axis), a.repeat([2], axis=axis))
1575: (4) def test_frompyfunc_nout_0(self):
1576: (8)     def f(x):
1577: (12)         x[0], x[-1] = x[-1], x[0]
1578: (8)         uf = np.frompyfunc(f, 1, 0)
1579: (8)         a = np.array([[1, 2, 3], [4, 5], [6, 7, 8, 9]], dtype=object)
1580: (8)         assert_equal(uf(a), ())
1581: (8)         expected = np.array([[3, 2, 1], [5, 4], [9, 7, 8, 6]], dtype=object)
1582: (8)         assert_array_equal(a, expected)
1583: (4) @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
1584: (4) def test_leak_in_structured_dtype_comparison(self):
1585: (8)     recordtype = np.dtype([('a', np.float64),
1586: (31)                     ('b', np.int32),
1587: (31)                     ('d', (str, 5))])
1588: (8)     a = np.zeros(2, dtype=recordtype)
1589: (8)     for i in range(100):
1590: (12)         a == a

```

```

1591: (8)             assert_(sys.getrefcount(a) < 10)
1592: (8)             before = sys.getrefcount(a)
1593: (8)             u, v = a[0], a[1]
1594: (8)             u == v
1595: (8)             del u, v
1596: (8)             gc.collect()
1597: (8)             after = sys.getrefcount(a)
1598: (8)             assert_equal(before, after)
1599: (4)             def test_empty_percentile(self):
1600: (8)                 assert_array_equal(np.percentile(np.arange(10), []), np.array([]))
1601: (4)             def test_void_COMPARE_segfault(self):
1602: (8)                 a = np.ones(3, dtype=[('object', 'O'), ('int', '<i2')]))
1603: (8)                 a.sort()
1604: (4)             def test_reshape_size_overflow(self):
1605: (8)                 a = np.ones(20)[::2]
1606: (8)                 if np.dtype(np.intp).itemsize == 8:
1607: (12)                     new_shape = (2, 13, 419, 691, 823, 2977518503)
1608: (8)                 else:
1609: (12)                     new_shape = (2, 7, 7, 43826197)
1610: (8)             assert_raises(ValueError, a.reshape, new_shape)
1611: (4)             @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
1612: (12)                         reason="PyPy bug in error formatting")
1613: (4)             def test_invalid_structured_dtypes(self):
1614: (8)                 assert_raises(ValueError, np.dtype, ('O', [('name', 'i8')]))
1615: (8)                 assert_raises(ValueError, np.dtype, ('i8', [('name', 'O')]))
1616: (8)                 assert_raises(ValueError, np.dtype,
1617: (22)                     ('i8', [('name', [('name', 'O')])]))
1618: (8)                 assert_raises(ValueError, np.dtype, [(['a', 'i4'], ('b', 'i4')], 'O'))
1619: (8)                 assert_raises(ValueError, np.dtype, ('i8', 'O'))
1620: (8)                 assert_raises(ValueError, np.dtype,
1621: (22)                     ('i', {'name': ('i', 0, 'title', 'oops')}))
1622: (8)                 assert_raises(ValueError, np.dtype,
1623: (22)                     ('i', {'name': ('i', 'wrongtype', 'title')}))
1624: (8)                 assert_raises(ValueError, np.dtype,
1625: (22)                     [(['a', 'O'], ('b', 'O')], [(['c', 'O'], ('d', 'O')])])
1626: (8)             a = np.ones(1, dtype=('O', [('name', 'O')]))
1627: (8)             assert_equal(a[0], 1)
1628: (8)             assert a[0] is a.item()
1629: (8)             assert type(a[0]) is int
1630: (4)             def test_correct_hash_dict(self):
1631: (8)                 all_types = set(np.sctypeDict.values()) - {np.void}
1632: (8)                 for t in all_types:
1633: (12)                     val = t()
1634: (12)                     try:
1635: (16)                         hash(val)
1636: (12)                     except TypeError as e:
1637: (16)                         assert_equal(t.__hash__, None)
1638: (12)                     else:
1639: (16)                         assert_(t.__hash__ != None)
1640: (4)             def test_scalar_copy(self):
1641: (8)                 scalar_types = set(np.sctypeDict.values())
1642: (8)                 values = {
1643: (12)                     np.void: b"a",
1644: (12)                     np.bytes_: b"a",
1645: (12)                     np.str_: "a",
1646: (12)                     np.datetime64: "2017-08-25",
1647: (8)                 }
1648: (8)                 for sctype in scalar_types:
1649: (12)                     item = sctype(values.get(sctype, 1))
1650: (12)                     item2 = copy.copy(item)
1651: (12)                     assert_equal(item, item2)
1652: (4)             def test_void_item_memview(self):
1653: (8)                 va = np.zeros(10, 'V4')
1654: (8)                 x = va[:1].item()
1655: (8)                 va[0] = b'\xFF\xFF\xFF\xFF'
1656: (8)                 del va
1657: (8)                 assert_equal(x, b'\x00\x00\x00\x00')
1658: (4)             def test_void_getitem(self):
1659: (8)                 assert_(np.array([b'a'], 'V1').astype('O') == b'a')

```

```

1660: (8) assert_(np.array([b'ab'], 'V2').astype('O') == b'ab')
1661: (8) assert_(np.array([b'abc'], 'V3').astype('O') == b'abc')
1662: (8) assert_(np.array([b'abcd'], 'V4').astype('O') == b'abcd')
1663: (4) def test_structarray_title(self):
1664: (8)     for j in range(5):
1665: (12)         structure = np.array([1], dtype=[(('x', 'X'), np.object_)])
1666: (12)         structure[0]['x'] = np.array([2])
1667: (12)         gc.collect()
1668: (4) def test_dtype_scalar_squeeze(self):
1669: (8)     values = {
1670: (12)         'S': b"a",
1671: (12)         'M': "2018-06-20",
1672: (8)
1673: (8)     for ch in np.typecodes['All']:
1674: (12)         if ch in 'O':
1675: (16)             continue
1676: (12)         sctype = np.dtype(ch).type
1677: (12)         scvalue = sctype(values.get(ch, 3))
1678: (12)         for axis in [None, ()]:
1679: (16)             squeezed = scvalue.squeeze(axis=axis)
1680: (16)             assert_equal(squeezed, scvalue)
1681: (16)             assert_equal(type(squeezed), type(scvalue))
1682: (4) def test_field_access_by_title(self):
1683: (8)     s = 'Some long field name'
1684: (8)     if HAS_REFCOUNT:
1685: (12)         base = sys.getrefcount(s)
1686: (8)         t = np.dtype([(s, 'f1'), np.float64])
1687: (8)         data = np.zeros(10, t)
1688: (8)         for i in range(10):
1689: (12)             str(data[['f1']])
1690: (12)             if HAS_REFCOUNT:
1691: (16)                 assert_(base <= sys.getrefcount(s))
1692: (4) @pytest.mark.parametrize('val', [
1693: (8)     np.ones((10, 10), dtype='int32'),
1694: (8)     np.uint64(10),
1695: (8)
1696: (4)     @pytest.mark.parametrize('protocol',
1697: (8)         range(2, pickle.HIGHEST_PROTOCOL + 1)
1698: (8)
1699: (4)     def test_pickle_module(self, protocol, val):
1700: (8)         s = pickle.dumps(val, protocol)
1701: (8)         assert b'_multiarray_umath' not in s
1702: (8)         if protocol == 5 and len(val.shape) > 0:
1703: (12)             assert b'numpy.core.numeric' in s
1704: (8)         else:
1705: (12)             assert b'numpy.core.multiarray' in s
1706: (4)     def test_object_casting_errors(self):
1707: (8)         arr = np.array(['AAAAA', 18465886.0, 18465886.0], dtype=object)
1708: (8)         assert_raises(ValueError, arr.astype, 'c8')
1709: (4)     def test_eff1d_casting(self):
1710: (8)         x = np.array([1, 2, 4, 7, 0], dtype=np.int16)
1711: (8)         res = np.ediff1d(x, to_begin=-99, to_end=np.array([88, 99]))
1712: (8)         assert_equal(res, [-99, 1, 2, 3, -7, 88, 99])
1713: (8)         res = np.ediff1d(x, to_begin=(1<<20), to_end=(1<<20))
1714: (8)         assert_equal(res, [0, 1, 2, 3, -7, 0])
1715: (4)     def test_pickle_datetime64_array(self):
1716: (8)         d = np.datetime64('2015-07-04 12:59:59.50', 'ns')
1717: (8)         arr = np.array([d])
1718: (8)         for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
1719: (12)             dumped = pickle.dumps(arr, protocol=proto)
1720: (12)             assert_equal(pickle.loads(dumped), arr)
1721: (4)     def test_bad_array_interface(self):
1722: (8)         class T:
1723: (12)             __array_interface__ = {}
1724: (8)             with assert_raises(ValueError):
1725: (12)                 np.array([T()])
1726: (4)     def test_2d_array_shape(self):
1727: (8)         class T:
1728: (12)             def __array__(self):

```

```

1729: (16)             return np.ndarray(shape=(0,0))
1730: (12)             def __iter__(self):
1731: (16)                 return iter([])
1732: (12)             def __getitem__(self, idx):
1733: (16)                 raise AssertionError("__getitem__ was called")
1734: (12)             def __len__(self):
1735: (16)                 return 0
1736: (8)             t = T()
1737: (8)             arr = np.array([t])
1738: (8)             assert arr.shape == (1, 0, 0)
1739: (4) @pytest.mark.skipif(sys.maxsize < 2 ** 31 + 1, reason='overflows 32-bit
python')
1740: (4)             def test_to_ctypes(self):
1741: (8)                 arr = np.zeros((2 ** 31 + 1), 'b')
1742: (8)                 assert arr.size * arr.itemsize > 2 ** 31
1743: (8)                 c_arr = np.ctypeslib.as_ctypes(arr)
1744: (8)                 assert_equal(c_arr._length_, arr.size)
1745: (4)             def test_complex_conversion_error(self):
1746: (8)                 with pytest.raises(TypeError, match=r"Unable to convert dtype.*"):
1747: (12)                     complex(np.array("now", np.datetime64))
1748: (4)             def test_array_interface_descr(self):
1749: (8)                 dt = np.dtype(dict(names=['a', 'b'],
1750: (27)                     offsets=[0, 0],
1751: (27)                     formats=[np.int64, np.int64]))
1752: (8)                 descr = np.array((1, 1), dtype=dt).__array_interface__['descr']
1753: (8)                 assert descr == [(b'', '|V8')] # instead of [(b'', '|V8')]
1754: (4) @pytest.mark.skipif(sys.maxsize < 2 ** 31 + 1, reason='overflows 32-bit
python')
1755: (4)             @requires_memory(free_bytes=9e9)
1756: (4)             def test_dot_big_stride(self):
1757: (8)                 int32_max = np.iinfo(np.int32).max
1758: (8)                 n = int32_max + 3
1759: (8)                 a = np.empty([n], dtype=np.float32)
1760: (8)                 b = a[::-n-1]
1761: (8)                 b[...] = 1
1762: (8)                 assert b.strides[0] > int32_max * b.dtype.itemsize
1763: (8)                 assert np.dot(b, b) == 2.0
1764: (4)             def test_frompyfunc_name(self):
1765: (8)                 def cass@0(x):
1766: (12)                     return x
1767: (8)                 f = np.frompyfunc(cass@0, 1, 1)
1768: (8)                 assert str(f) == "<ufunc 'cass@0 (vectorized)'>"
1769: (4)             @pytest.mark.parametrize("operation", [
1770: (8)                 'add', 'subtract', 'multiply', 'floor_divide',
1771: (8)                 'conjugate', 'fmod', 'square', 'reciprocal',
1772: (8)                 'power', 'absolute', 'negative', 'positive',
1773: (8)                 'greater', 'greater_equal', 'less',
1774: (8)                 'less_equal', 'equal', 'not_equal', 'logical_and',
1775: (8)                 'logical_not', 'logical_or', 'bitwise_and', 'bitwise_or',
1776: (8)                 'bitwise_xor', 'invert', 'left_shift', 'right_shift',
1777: (8)                 'gcd', 'lcm'
1778: (8)             ])
1779: (4)
1780: (4)             @pytest.mark.parametrize("order", [
1781: (8)                 ('b->', 'B->'),
1782: (8)                 ('h->', 'H->'),
1783: (8)                 ('i->', 'I->'),
1784: (8)                 ('l->', 'L->'),
1785: (8)                 ('q->', 'Q->'),
1786: (8)             ])
1787: (4)
1788: (4)             def test_ufunc_order(self, operation, order):
1789: (8)                 def get_idx(string, str_lst):
1790: (12)                     for i, s in enumerate(str_lst):
1791: (16)                         if string in s:
1792: (20)                             return i
1793: (12)                         raise ValueError(f"{string} not in list")
1794: (8)                 types = getattr(np, operation).types
1795: (8)                 assert get_idx(order[0], types) < get_idx(order[1], types), (

```

```

1796: (16)             f"Unexpected types order of ufunc in {operation}"
1797: (16)             f"for {order}. Possible fix: Use signed before unsigned"
1798: (16)             "in generate_umath.py")
1799: (4)              def test_nonbool_logical(self):
1800: (8)                size = 100
1801: (8)                a = np.frombuffer(b'\x01' * size, dtype=np.bool_)
1802: (8)                b = np.frombuffer(b'\x80' * size, dtype=np.bool_)
1803: (8)                expected = np.ones(size, dtype=np.bool_)
1804: (8)                assert_array_equal(np.logical_and(a, b), expected)
-----
```

## File 119 - test\_scalarbuffer.py:

```

1: (0)             """
2: (0)             Test scalar buffer interface adheres to PEP 3118
3: (0)             """
4: (0)             import numpy as np
5: (0)             from numpy.core._rational_tests import rational
6: (0)             from numpy.core._multiarray_tests import get_buffer_info
7: (0)             import pytest
8: (0)             from numpy.testing import assert_, assert_equal, assert_raises
9: (0)             scalars_and_codes = [
10: (4)               (np.bool_, '?'),
11: (4)               (np.byte, 'b'),
12: (4)               (np.short, 'h'),
13: (4)               (np.intc, 'i'),
14: (4)               (np.int_, 'l'),
15: (4)               (np.longlong, 'q'),
16: (4)               (np.ubyte, 'B'),
17: (4)               (np ushort, 'H'),
18: (4)               (np.uintc, 'I'),
19: (4)               (np.uint, 'L'),
20: (4)               (np.ulonglong, 'Q'),
21: (4)               (np.half, 'e'),
22: (4)               (np.single, 'f'),
23: (4)               (np.double, 'd'),
24: (4)               (np.longdouble, 'g'),
25: (4)               (np.csingle, 'Zf'),
26: (4)               (np.cdouble, 'Zd'),
27: (4)               (np.clongdouble, 'Zg'),
28: (0)
29: (0)             ]
30: (0)             scalars_only, codes_only = zip(*scalars_and_codes)
31: (4)             class TestScalarPEP3118:
32: (4)               @pytest.mark.parametrize('scalar', scalars_only, ids=codes_only)
33: (8)               def test_scalar_match_array(self, scalar):
34: (8)                 x = scalar()
35: (8)                 a = np.array([], dtype=np.dtype(scalar))
36: (8)                 mv_x = memoryview(x)
37: (8)                 mv_a = memoryview(a)
38: (4)                 assert_equal(mv_x.format, mv_a.format)
39: (4)               @pytest.mark.parametrize('scalar', scalars_only, ids=codes_only)
40: (4)               def test_scalar_dim(self, scalar):
41: (8)                 x = scalar()
42: (8)                 mv_x = memoryview(x)
43: (8)                 assert_equal(mv_x.itemsize, np.dtype(scalar).itemsize)
44: (8)                 assert_equal(mv_x.ndim, 0)
45: (8)                 assert_equal(mv_x.shape, ())
46: (8)                 assert_equal(mv_x.strides, ())
47: (8)                 assert_equal(mv_x.suboffsets, ())
48: (4)               @pytest.mark.parametrize('scalar, code', scalars_and_codes,
49: (8)                 ids=codes_only)
50: (4)               def test_scalar_code_and_properties(self, scalar, code):
51: (8)                 x = scalar()
52: (8)                 expected = dict(strides=(), itemsize=x.dtype.itemsize, ndim=0,
53: (24)                               shape=(), format=code, readonly=True)
54: (8)                 mv_x = memoryview(x)
55: (8)                 assert self._as_dict(mv_x) == expected
56: (4)               @pytest.mark.parametrize('scalar', scalars_only, ids=codes_only)
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

55: (4)          def test_scalar_buffers_readonly(self, scalar):
56: (8)            x = scalar()
57: (8)            with pytest.raises(BufferError, match="scalar buffer is readonly"):
58: (12)              get_buffer_info(x, ["WRITABLE"])
59: (4)          def test_void_scalar_structured_data(self):
60: (8)            dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
61: (8)            x = np.array(('ndarray_scalar', (1.2, 3.0)), dtype=dt)[()]
62: (8)            assert_(isinstance(x, np.void))
63: (8)            mv_x = memoryview(x)
64: (8)            expected_size = 16 * np.dtype((np.str_, 1)).itemsize
65: (8)            expected_size += 2 * np.dtype(np.float64).itemsize
66: (8)            assert_equal(mv_x.itemsize, expected_size)
67: (8)            assert_equal(mv_x.ndim, 0)
68: (8)            assert_equal(mv_x.shape, ())
69: (8)            assert_equal(mv_x.strides, ())
70: (8)            assert_equal(mv_x.suboffsets, ())
71: (8)            a = np.array([('Sarah', (8.0, 7.0)), ('John', (6.0, 7.0))], dtype=dt)
72: (8)            assert_(isinstance(a, np.ndarray))
73: (8)            mv_a = memoryview(a)
74: (8)            assert_equal(mv_x.itemsize, mv_a.itemsize)
75: (8)            assert_equal(mv_x.format, mv_a.format)
76: (8)            with pytest.raises(BufferError, match="scalar buffer is readonly"):
77: (12)              get_buffer_info(x, ["WRITABLE"])
78: (4)          def _as_dict(self, m):
79: (8)            return dict(strides=m.strides, shape=m.shape, itemsize=m.itemsize,
80: (20)                  ndim=m.ndim, format=m.format, readonly=m.readonly)
81: (4)          def test_datetime_memoryview(self):
82: (8)            dt1 = np.datetime64('2016-01-01')
83: (8)            dt2 = np.datetime64('2017-01-01')
84: (8)            expected = dict(strides=(1,), itemsize=1, ndim=1, shape=(8,),
85: (24)                  format='B', readonly=True)
86: (8)            v = memoryview(dt1)
87: (8)            assert self._as_dict(v) == expected
88: (8)            v = memoryview(dt2 - dt1)
89: (8)            assert self._as_dict(v) == expected
90: (8)            dt = np.dtype([('a', 'uint16'), ('b', 'M8[s]')])
91: (8)            a = np.empty(1, dt)
92: (8)            assert_raises((ValueError, BufferError), memoryview, a[0])
93: (8)            with pytest.raises(BufferError, match="scalar buffer is readonly"):
94: (12)              get_buffer_info(dt1, ["WRITABLE"])
95: (4)          @pytest.mark.parametrize('s', [
96: (8)            pytest.param("\x32\x32", id="ascii"),
97: (8)            pytest.param("\uFE0F\uFE0F", id="basic multilingual"),
98: (8)            pytest.param("\U0001f4bb\U0001f4bb", id="non-BMP"),
99: (4)          ])
100: (4)         def test_str_ucs4(self, s):
101: (8)           s = np.str_(s) # only our subclass implements the buffer protocol
102: (8)           expected = dict(strides=(), itemsize=8, ndim=0, shape=(), format='2w',
103: (24)                 readonly=True)
104: (8)           v = memoryview(s)
105: (8)           assert self._as_dict(v) == expected
106: (8)           code_points = np.frombuffer(v, dtype='i4')
107: (8)           assert_equal(code_points, [ord(c) for c in s])
108: (8)           with pytest.raises(BufferError, match="scalar buffer is readonly"):
109: (12)             get_buffer_info(s, ["WRITABLE"])
110: (4)         def test_user_scalar_fails_buffer(self):
111: (8)           r = rational(1)
112: (8)           with assert_raises(TypeError):
113: (12)             memoryview(r)
114: (8)           with pytest.raises(BufferError, match="scalar buffer is readonly"):
115: (12)             get_buffer_info(r, ["WRITABLE"])

```

---

File 120 - test\_scalarinherit.py:

```

1: (0)          """ Test printing of scalar types.
2: (0)          """
3: (0)          import pytest

```

```
4: (0) import numpy as np
5: (0) from numpy.testing import assert_, assert_raises
6: (0) class A:
7: (4)     pass
8: (0) class B(A, np.float64):
9: (4)     pass
10: (0) class C(B):
11: (4)     pass
12: (0) class D(C, B):
13: (4)     pass
14: (0) class B0(np.float64, A):
15: (4)     pass
16: (0) class C0(B0):
17: (4)     pass
18: (0) class HasNew:
19: (4)     def __new__(cls, *args, **kwargs):
20: (8)         return cls, args, kwargs
21: (0) class B1(np.float64, HasNew):
22: (4)     pass
23: (0) class TestInherit:
24: (4)     def test_init(self):
25: (8)         x = B(1.0)
26: (8)         assert_(str(x) == '1.0')
27: (8)         y = C(2.0)
28: (8)         assert_(str(y) == '2.0')
29: (8)         z = D(3.0)
30: (8)         assert_(str(z) == '3.0')
31: (4)     def test_init2(self):
32: (8)         x = B0(1.0)
33: (8)         assert_(str(x) == '1.0')
34: (8)         y = C0(2.0)
35: (8)         assert_(str(y) == '2.0')
36: (4)     def test_gh_15395(self):
37: (8)         x = B1(1.0)
38: (8)         assert_(str(x) == '1.0')
39: (8)         with pytest.raises(TypeError):
40: (12)             B1(1.0, 2.0)
41: (0) class TestCharacter:
42: (4)     def test_char_radd(self):
43: (8)         np_s = np.bytes_('abc')
44: (8)         np_u = np.str_('abc')
45: (8)         s = b'def'
46: (8)         u = 'def'
47: (8)         assert_(np_s.__radd__(np_s) is NotImplemented)
48: (8)         assert_(np_s.__radd__(np_u) is NotImplemented)
49: (8)         assert_(np_s.__radd__(s) is NotImplemented)
50: (8)         assert_(np_s.__radd__(u) is NotImplemented)
51: (8)         assert_(np_u.__radd__(np_s) is NotImplemented)
52: (8)         assert_(np_u.__radd__(np_u) is NotImplemented)
53: (8)         assert_(np_u.__radd__(s) is NotImplemented)
54: (8)         assert_(np_u.__radd__(u) is NotImplemented)
55: (8)         assert_(s + np_s == b'defabc')
56: (8)         assert_(u + np_u == 'defabc')
57: (8)         class MyStr(str, np.generic):
58: (12)             pass
59: (8)         with assert_raises(TypeError):
60: (12)             ret = s + MyStr('abc')
61: (8)         class MyBytes(bytes, np.generic):
62: (12)             pass
63: (8)             ret = s + MyBytes(b'abc')
64: (8)             assert(type(ret) is type(s))
65: (8)             assert ret == b"defabc"
66: (4)         def test_char_repeat(self):
67: (8)             np_s = np.bytes_('abc')
68: (8)             np_u = np.str_('abc')
69: (8)             res_s = b'abc' * 5
70: (8)             res_u = 'abc' * 5
71: (8)             assert_(np_s * 5 == res_s)
72: (8)             assert_(np_u * 5 == res_u)
```

-----  
File 121 - test\_scalarmath.py:

```

1: (0)         import contextlib
2: (0)         import sys
3: (0)         import warnings
4: (0)         import itertools
5: (0)         import operator
6: (0)         import platform
7: (0)         from numpy._utils import _pep440
8: (0)         import pytest
9: (0)         from hypothesis import given, settings
10: (0)        from hypothesis.strategies import sampled_from
11: (0)        from hypothesis.extra import numpy as hynp
12: (0)        import numpy as np
13: (0)        from numpy.testing import (
14: (4)            assert_, assert_equal, assert_raises, assert_almost_equal,
15: (4)            assert_array_equal, IS_PYPY, suppress_warnings, _gen_alignment_data,
16: (4)            assert_warns, _SUPPORTS_SVE,
17: (4)        )
18: (0)        try:
19: (4)            COMPILERS = np.show_config(mode="dicts")["Compilers"]
20: (4)            USING_CLANG_CL = COMPILERS["c"]["name"] == "clang-cl"
21: (0)        except TypeError:
22: (4)            USING_CLANG_CL = False
23: (0)        types = [np.bool_, np.byte, np.ubyte, np.short, np ushort, np.intc, np.uintc,
24: (9)            np.int_, np.uint, np.longlong, np ulonglong,
25: (9)            np.single, np.double, np.longdouble, np.csingle,
26: (9)            np.cdouble, np.clongdouble]
27: (0)        floating_types = np.floating.__subclasses__()
28: (0)        complex_floating_types = np.complexfloating.__subclasses__()
29: (0)        objecty_things = [object(), None]
30: (0)        reasonable_operators_for_scalars = [
31: (4)            operator.lt, operator.le, operator.eq, operator.ne, operator.ge,
32: (4)            operator.gt, operator.add, operator.floordiv, operator.mod,
33: (4)            operator.mul, operator.pow, operator.sub, operator.truediv,
34: (0)        ]
35: (0)        class TestTypes:
36: (4)            def test_types(self):
37: (8)                for atype in types:
38: (12)                    a = atype(1)
39: (12)                    assert_(a == 1, "error with %r: got %r" % (atype, a))
40: (4)            def test_type_add(self):
41: (8)                for k, atype in enumerate(types):
42: (12)                    a_scalar = atype(3)
43: (12)                    a_array = np.array([3], dtype=atype)
44: (12)                    for l, btype in enumerate(types):
45: (16)                        b_scalar = btype(1)
46: (16)                        b_array = np.array([1], dtype=btype)
47: (16)                        c_scalar = a_scalar + b_scalar
48: (16)                        c_array = a_array + b_array
49: (16)                        assert_equal(c_scalar.dtype, c_array.dtype,
50: (27)                            "error with types (%d/'%c' + %d/'%c')" %
51: (28)                                (k, np.dtype(atype).char, l,
np.dtype(btype).char))
52: (4)            def test_type_create(self):
53: (8)                for k, atype in enumerate(types):
54: (12)                    a = np.array([1, 2, 3], atype)
55: (12)                    b = atype([1, 2, 3])
56: (12)                    assert_equal(a, b)
57: (4)            def test_leak(self):
58: (8)                for i in range(200000):
59: (12)                    np.add(1, 1)
60: (0)            def check_ufunc_scalar_equivalence(op, arr1, arr2):
61: (4)                scalar1 = arr1[()]
62: (4)                scalar2 = arr2[()]
63: (4)                assert isinstance(scalar1, np.generic)

```

```

64: (4)             assert isinstance(scalar2, np.generic)
65: (4)             if arr1.dtype.kind == "c" or arr2.dtype.kind == "c":
66: (8)                 comp_ops = {operator.ge, operator.gt, operator.le, operator.lt}
67: (8)                 if op in comp_ops and (np.isnan(scalar1) or np.isnan(scalar2)):
68: (12)                     pytest.xfail("complex comp ufuncs use sort-order, scalars do
not.")
69: (4)             if op == operator.pow and arr2.item() in [-1, 0, 0.5, 1, 2]:
70: (8)                 pytest.skip("array**2 can have incorrect/weird result dtype")
71: (4)             with warnings.catch_warnings(), np.errstate(all="ignore"):
72: (8)                 warnings.simplefilter("error", DeprecationWarning)
73: (8)             try:
74: (12)                 res = op(arr1, arr2)
75: (8)             except Exception as e:
76: (12)                 with pytest.raises(type(e)):
77: (16)                     op(scalar1, scalar2)
78: (8)             else:
79: (12)                 scalar_res = op(scalar1, scalar2)
80: (12)                 assert_array_equal(scalar_res, res, strict=True)
81: (0)             @pytest.mark.slow
82: (0)             @settings(max_examples=10000, deadline=2000)
83: (0)             @given(sampled_from(reasonable_operators_for_scalars),
84: (7)                 hyhp.arrays(dtype=hyhp.scalar_dtypes(), shape=()),
85: (7)                 hyhp.arrays(dtype=hyhp.scalar_dtypes(), shape=()))
86: (0)             def test_array_scalar_ufunc_equivalence(op, arr1, arr2):
87: (4)                 """
88: (4)                 This is a thorough test attempting to cover important promotion paths
89: (4)                 and ensuring that arrays and scalars stay as aligned as possible.
90: (4)                 However, if it creates troubles, it should maybe just be removed.
91: (4)                 """
92: (4)                 check_ufunc_scalar_equivalence(op, arr1, arr2)
93: (0)             @pytest.mark.slow
94: (0)             @given(sampled_from(reasonable_operators_for_scalars),
95: (7)                 hyhp.scalar_dtypes(), hyhp.scalar_dtypes())
96: (0)             def test_array_scalar_ufunc_dtypes(op, dt1, dt2):
97: (4)                 arr1 = np.array(2, dtype=dt1)
98: (4)                 arr2 = np.array(3, dtype=dt2) # some power do weird things.
99: (4)                 check_ufunc_scalar_equivalence(op, arr1, arr2)
100: (0)            @pytest.mark.parametrize("fscalar", [np.float16, np.float32])
101: (0)            def test_int_float_promotion_truediv(fscalar):
102: (4)                i = np.int8(1)
103: (4)                f = fscalar(1)
104: (4)                expected = np.result_type(i, f)
105: (4)                assert (i / f).dtype == expected
106: (4)                assert (f / i).dtype == expected
107: (4)                assert (i / i).dtype == np.dtype("float64")
108: (4)                assert (np.int16(1) / f).dtype == np.dtype("float32")
109: (0)            class TestBaseMath:
110: (4)                @pytest.mark.xfail(_SUPPORTS_SVE, reason="gh-22982")
111: (4)                def test_blocked(self):
112: (8)                    for dt, sz in [(np.float32, 11), (np.float64, 7), (np.int32, 11)]:
113: (12)                        for out, inp1, inp2, msg in _gen_alignment_data(dtype=dt,
114: (60)   type='binary',
115: (60)   max_size=sz):
116: (16)                            exp1 = np.ones_like(inp1)
117: (16)                            inp1[...] = np.ones_like(inp1)
118: (16)                            inp2[...] = np.zeros_like(inp2)
119: (16)                            assert_almost_equal(np.add(inp1, inp2), exp1, err_msg=msg)
120: (16)                            assert_almost_equal(np.add(inp1, 2), exp1 + 2, err_msg=msg)
121: (16)                            assert_almost_equal(np.add(1, inp2), exp1, err_msg=msg)
122: (16)                            np.add(inp1, inp2, out=out)
123: (16)                            assert_almost_equal(out, exp1, err_msg=msg)
124: (16)                            inp2[...] += np.arange(inp2.size, dtype=dt) + 1
125: (16)                            assert_almost_equal(np.square(inp2),
126: (36)   np.multiply(inp2, inp2), err_msg=msg)
127: (16)                            if dt != np.int32:
128: (20)                                assert_almost_equal(np.reciprocal(inp2),
129: (40)   np.divide(1, inp2), err_msg=msg)
130: (16)                                inp1[...] = np.ones_like(inp1)
131: (16)                                np.add(inp1, 2, out=out)

```

```

132: (16) assert_almost_equal(out, exp1 + 2, err_msg=msg)
133: (16) inp2[...] = np.ones_like(inp2)
134: (16) np.add(2, inp2, out=out)
135: (16) assert_almost_equal(out, exp1 + 2, err_msg=msg)
136: (4) def test_lower_align(self):
137: (8)     d = np.zeros(23 * 8, dtype=np.int8)[4:-4].view(np.float64)
138: (8)     o = np.zeros(23 * 8, dtype=np.int8)[4:-4].view(np.float64)
139: (8)     assert_almost_equal(d + d, d * 2)
140: (8)     np.add(d, d, out=o)
141: (8)     np.add(np.ones_like(d), d, out=o)
142: (8)     np.add(d, np.ones_like(d), out=o)
143: (8)     np.add(np.ones_like(d), d)
144: (8)     np.add(d, np.ones_like(d))
145: (0) class TestPower:
146: (4)     def test_small_types(self):
147: (8)         for t in [np.int8, np.int16, np.float16]:
148: (12)             a = t(3)
149: (12)             b = a ** 4
150: (12)             assert_(b == 81, "error with %r: got %r" % (t, b))
151: (4)     def test_large_types(self):
152: (8)         for t in [np.int32, np.int64, np.float32, np.float64, np.longdouble]:
153: (12)             a = t(51)
154: (12)             b = a ** 4
155: (12)             msg = "error with %r: got %r" % (t, b)
156: (12)             if np.issubdtype(t, np.integer):
157: (16)                 assert_(b == 6765201, msg)
158: (12)             else:
159: (16)                 assert_almost_equal(b, 6765201, err_msg=msg)
160: (4)     def test_integers_to_negative_integer_power(self):
161: (8)         exp = [np.array(-1, dt)[()] for dt in 'bhilq']
162: (8)         base = [np.array(1, dt)[()] for dt in 'bhilqBHILQ']
163: (8)         for i1, i2 in itertools.product(base, exp):
164: (12)             if i1.dtype != np.uint64:
165: (16)                 assert_raises(ValueError, operator.pow, i1, i2)
166: (12)             else:
167: (16)                 res = operator.pow(i1, i2)
168: (16)                 assert_(res.dtype.type is np.float64)
169: (16)                 assert_almost_equal(res, 1.)
170: (8)         base = [np.array(-1, dt)[()] for dt in 'bhilq']
171: (8)         for i1, i2 in itertools.product(base, exp):
172: (12)             if i1.dtype != np.uint64:
173: (16)                 assert_raises(ValueError, operator.pow, i1, i2)
174: (12)             else:
175: (16)                 res = operator.pow(i1, i2)
176: (16)                 assert_(res.dtype.type is np.float64)
177: (16)                 assert_almost_equal(res, -1.)
178: (8)         base = [np.array(2, dt)[()] for dt in 'bhilqBHILQ']
179: (8)         for i1, i2 in itertools.product(base, exp):
180: (12)             if i1.dtype != np.uint64:
181: (16)                 assert_raises(ValueError, operator.pow, i1, i2)
182: (12)             else:
183: (16)                 res = operator.pow(i1, i2)
184: (16)                 assert_(res.dtype.type is np.float64)
185: (16)                 assert_almost_equal(res, .5)
186: (4)     def test_mixed_types(self):
187: (8)         typelist = [np.int8, np.int16, np.float16,
188: (20)                     np.float32, np.float64, np.int8,
189: (20)                     np.int16, np.int32, np.int64]
190: (8)         for t1 in typelist:
191: (12)             for t2 in typelist:
192: (16)                 a = t1(3)
193: (16)                 b = t2(2)
194: (16)                 result = a**b
195: (16)                 msg = ("error with %r and %r:"
196: (23)                     "got %r, expected %r") % (t1, t2, result, 9)
197: (16)                 if np.issubdtype(np.dtype(result), np.integer):
198: (20)                     assert_(result == 9, msg)
199: (16)                 else:
200: (20)                     assert_almost_equal(result, 9, err_msg=msg)

```

```

201: (4)          def test_modular_power(self):
202: (8)            a = 5
203: (8)            b = 4
204: (8)            c = 10
205: (8)            expected = pow(a, b, c) # noqa: F841
206: (8)            for t in (np.int32, np.float32, np.complex64):
207: (12)              assert_raises(TypeError, operator.pow, t(a), b, c)
208: (12)              assert_raises(TypeError, operator.pow, np.array(t(a)), b, c)
209: (0)          def floordiv_and_mod(x, y):
210: (4)            return (x // y, x % y)
211: (0)          def _signs(dt):
212: (4)            if dt in np.typecodes['UnsignedInteger']:
213: (8)              return (+1,)
214: (4)            else:
215: (8)              return (+1, -1)
216: (0)          class TestModulus:
217: (4)            def test_modulus_basic(self):
218: (8)              dt = np.typecodes['AllInteger'] + np.typecodes['Float']
219: (8)              for op in [floordiv_and_mod, divmod]:
220: (12)                for dt1, dt2 in itertools.product(dt, dt):
221: (16)                  for sg1, sg2 in itertools.product(_signs(dt1), _signs(dt2)):
222: (20)                    fmt = 'op: %s, dt1: %s, dt2: %s, sg1: %s, sg2: %s'
223: (20)                    msg = fmt % (op.__name__, dt1, dt2, sg1, sg2)
224: (20)                    a = np.array(sg1*71, dtype=dt1)[()]
225: (20)                    b = np.array(sg2*19, dtype=dt2)[()]
226: (20)                    div, rem = op(a, b)
227: (20)                    assert_equal(div*b + rem, a, err_msg=msg)
228: (20)                    if sg2 == -1:
229: (24)                      assert_(b < rem <= 0, msg)
230: (20)                    else:
231: (24)                      assert_(b > rem >= 0, msg)
232: (4)            def test_float_modulus_exact(self):
233: (8)              nlst = list(range(-127, 0))
234: (8)              plst = list(range(1, 128))
235: (8)              dividend = nlst + [0] + plst
236: (8)              divisor = nlst + plst
237: (8)              arg = list(itertools.product(dividend, divisor))
238: (8)              tgt = list(divmod(*t) for t in arg)
239: (8)              a, b = np.array(arg, dtype=int).T
240: (8)              tgtdiv, tgtrem = np.array(tgt, dtype=float).T
241: (8)              tgtdiv = np.where((tgtdiv == 0.0) & ((b < 0) ^ (a < 0)), -0.0, tgtdiv)
242: (8)              tgtrem = np.where((tgtrem == 0.0) & (b < 0), -0.0, tgtrem)
243: (8)              for op in [floordiv_and_mod, divmod]:
244: (12)                for dt in np.typecodes['Float']:
245: (16)                  msg = 'op: %s, dtype: %s' % (op.__name__, dt)
246: (16)                  fa = a.astype(dt)
247: (16)                  fb = b.astype(dt)
248: (16)                  div, rem = zip(*[op(a_, b_) for a_, b_ in zip(fa, fb)])
249: (16)                  assert_equal(div, tgtdiv, err_msg=msg)
250: (16)                  assert_equal(rem, tgtrem, err_msg=msg)
251: (4)            def test_float_modulus_roundoff(self):
252: (8)              dt = np.typecodes['Float']
253: (8)              for op in [floordiv_and_mod, divmod]:
254: (12)                for dt1, dt2 in itertools.product(dt, dt):
255: (16)                  for sg1, sg2 in itertools.product((+1, -1), (+1, -1)):
256: (20)                    fmt = 'op: %s, dt1: %s, dt2: %s, sg1: %s, sg2: %s'
257: (20)                    msg = fmt % (op.__name__, dt1, dt2, sg1, sg2)
258: (20)                    a = np.array(sg1*78*6e-8, dtype=dt1)[()]
259: (20)                    b = np.array(sg2*6e-8, dtype=dt2)[()]
260: (20)                    div, rem = op(a, b)
261: (20)                    assert_equal(div*b + rem, a, err_msg=msg)
262: (20)                    if sg2 == -1:
263: (24)                      assert_(b < rem <= 0, msg)
264: (20)                    else:
265: (24)                      assert_(b > rem >= 0, msg)
266: (4)            def test_float_modulus_corner_cases(self):
267: (8)              for dt in np.typecodes['Float']:
268: (12)                b = np.array(1.0, dtype=dt)
269: (12)                a = np.nextafter(np.array(0.0, dtype=dt), -b)

```

```

270: (12)           rem = operator.mod(a, b)
271: (12)           assert_(rem <= b, 'dt: %s' % dt)
272: (12)           rem = operator.mod(-a, -b)
273: (12)           assert_(rem >= -b, 'dt: %s' % dt)
274: (8)            with suppress_warnings() as sup:
275: (12)              sup.filter(RuntimeWarning, "invalid value encountered in
276: (12)              sup.filter(RuntimeWarning, "divide by zero encountered in
277: (12)              sup.filter(RuntimeWarning, "divide by zero encountered in
278: (12)              sup.filter(RuntimeWarning, "divide by zero encountered in divmod")
279: (12)              sup.filter(RuntimeWarning, "invalid value encountered in divmod")
280: (12)            for dt in np.typecodes['Float']:
281: (16)              fone = np.array(1.0, dtype=dt)
282: (16)              fzer = np.array(0.0, dtype=dt)
283: (16)              finf = np.array(np.inf, dtype=dt)
284: (16)              fnan = np.array(np.nan, dtype=dt)
285: (16)              rem = operator.mod(fone, fzer)
286: (16)              assert_(np.isnan(rem), 'dt: %s' % dt)
287: (16)              rem = operator.mod(fone, fnan)
288: (16)              assert_(np.isnan(rem), 'dt: %s' % dt)
289: (16)              rem = operator.mod(finf, fone)
290: (16)              assert_(np.isnan(rem), 'dt: %s' % dt)
291: (16)              for op in [floordiv_and_mod, divmod]:
292: (20)                  div, mod = op(fone, fzer)
293: (20)                  assert_(np.isinf(div)) and assert_(np.isnan(mod))
294: (4)               def test_inplace_floordiv_handling(self):
295: (8)                 a = np.array([1, 2], np.int64)
296: (8)                 b = np.array([1, 2], np.uint64)
297: (8)                 with pytest.raises(TypeError,
298: (16)                     match=r"Cannot cast ufunc 'floor_divide' output from"):
299: (12)                     a // b
300: (0)          class TestComplexDivision:
301: (4)            def test_zero_division(self):
302: (8)              with np.errstate(all="ignore"):
303: (12)                for t in [np.complex64, np.complex128]:
304: (16)                  a = t(0.0)
305: (16)                  b = t(1.0)
306: (16)                  assert_(np.isinf(b/a))
307: (16)                  b = t(complex(np.inf, np.inf))
308: (16)                  assert_(np.isinf(b/a))
309: (16)                  b = t(complex(np.inf, np.nan))
310: (16)                  assert_(np.isinf(b/a))
311: (16)                  b = t(complex(np.nan, np.inf))
312: (16)                  assert_(np.isinf(b/a))
313: (16)                  b = t(complex(np.nan, np.nan))
314: (16)                  assert_(np.isnan(b/a))
315: (16)                  b = t(0.)
316: (16)                  assert_(np.isnan(b/a))
317: (4)               def test_signed_zeros(self):
318: (8)                 with np.errstate(all="ignore"):
319: (12)                   for t in [np.complex64, np.complex128]:
320: (16)                     data = (
321: (20)                       (( 0.0,-1.0), ( 0.0, 1.0), (-1.0,-0.0)),
322: (20)                       (( 0.0,-1.0), ( 0.0,-1.0), ( 1.0,-0.0)),
323: (20)                       (( 0.0,-1.0), (-0.0,-1.0), ( 1.0, 0.0)),
324: (20)                       (( 0.0,-1.0), (-0.0, 1.0), (-1.0, 0.0)),
325: (20)                       (( 0.0, 1.0), ( 0.0,-1.0), (-1.0, 0.0)),
326: (20)                       (( 0.0,-1.0), ( 0.0,-1.0), ( 1.0,-0.0)),
327: (20)                       ((-0.0,-1.0), ( 0.0,-1.0), ( 1.0,-0.0)),
328: (20)                       ((-0.0, 1.0), ( 0.0,-1.0), (-1.0,-0.0)))
329: (16)                     )
330: (16)                     for cases in data:
331: (20)                         n = cases[0]
332: (20)                         d = cases[1]
333: (20)                         ex = cases[2]
334: (20)                         result = t(complex(n[0], n[1])) / t(complex(d[0], d[1]))
335: (20)                         assert_equal(result.real, ex[0])

```

```

336: (20)                                assert_equal(result.imag, ex[1])
337: (4)       def test_branches(self):
338: (8)           with np.errstate(all="ignore"):
339: (12)               for t in [np.complex64, np.complex128]:
340: (16)                   data = list()
341: (16)                   data.append((( 2.0, 1.0), ( 2.0, 1.0), (1.0, 0.0)))
342: (16)                   data.append((( 1.0, 2.0), ( 1.0, 2.0), (1.0, 0.0)))
343: (16)                   for cases in data:
344: (20)                       n = cases[0]
345: (20)                       d = cases[1]
346: (20)                       ex = cases[2]
347: (20)                       result = t(complex(n[0], n[1])) / t(complex(d[0], d[1]))
348: (20)                       assert_equal(result.real, ex[0])
349: (20)                       assert_equal(result.imag, ex[1])
350: (0)       class TestConversion:
351: (4)           def test_int_from_long(self):
352: (8)               l = [1e6, 1e12, 1e18, -1e6, -1e12, -1e18]
353: (8)               li = [10**6, 10**12, 10**18, -10**6, -10**12, -10**18]
354: (8)               for T in [None, np.float64, np.int64]:
355: (12)                   a = np.array(l, dtype=T)
356: (12)                   assert_equal([int(_m) for _m in a], li)
357: (8)               a = np.array(l[:3], dtype=np.uint64)
358: (8)               assert_equal([int(_m) for _m in a], li[:3])
359: (4)       def test_iinfo_long_values(self):
360: (8)           for code in 'bBhH':
361: (12)               with pytest.warns(DeprecationWarning):
362: (16)                   res = np.array(np.iinfo(code).max + 1, dtype=code)
363: (12)                   tgt = np.iinfo(code).min
364: (12)                   assert_(res == tgt)
365: (8)               for code in np.typecodes['AllInteger']:
366: (12)                   res = np.array(np.iinfo(code).max, dtype=code)
367: (12)                   tgt = np.iinfo(code).max
368: (12)                   assert_(res == tgt)
369: (8)               for code in np.typecodes['AllInteger']:
370: (12)                   res = np.dtype(code).type(np.iinfo(code).max)
371: (12)                   tgt = np.iinfo(code).max
372: (12)                   assert_(res == tgt)
373: (4)       def test_int_raise_behaviour(self):
374: (8)           def overflow_error_func(dtype):
375: (12)               dtype(np.iinfo(dtype).max + 1)
376: (8)               for code in [np.int_, np.uint, np.longlong, np.ulonglong]:
377: (12)                   assert_raises(OverflowError, overflow_error_func, code)
378: (4)       def test_int_from_infinite_longdouble(self):
379: (8)           x = np.longdouble(np.inf)
380: (8)           assert_raises(OverflowError, int, x)
381: (8)           with suppress_warnings() as sup:
382: (12)               sup.record(np.ComplexWarning)
383: (12)               x = np.clongdouble(np.inf)
384: (12)               assert_raises(OverflowError, int, x)
385: (12)               assert_equal(len(sup.log), 1)
386: (4) @pytest.mark.skipif(not IS_PYPY, reason="Test is PyPy only (gh-9972)")
387: (4)       def test_int_from_infinite_longdouble__int__(self):
388: (8)           x = np.longdouble(np.inf)
389: (8)           assert_raises(OverflowError, x.__int__)
390: (8)           with suppress_warnings() as sup:
391: (12)               sup.record(np.ComplexWarning)
392: (12)               x = np.clongdouble(np.inf)
393: (12)               assert_raises(OverflowError, x.__int__)
394: (12)               assert_equal(len(sup.log), 1)
395: (4) @pytest.mark.skipif(np.finfo(np.double) == np.finfo(np.longdouble),
396: (24)                                     reason="long double is same as double")
397: (4) @pytest.mark.skipif(platform.machine().startswith("ppc"),
398: (24)                                     reason="IBM double double")
399: (4)       def test_int_from_huge_longdouble(self):
400: (8)           exp = np.finfo(np.double).maxexp - 1
401: (8)           huge_ld = 2 * 1234 * np.longdouble(2) ** exp
402: (8)           huge_i = 2 * 1234 * 2 ** exp
403: (8)           assert_(huge_ld != np.inf)
404: (8)           assert_equal(int(huge_ld), huge_i)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

405: (4)
406: (8)
407: (8)
408: (8)
409: (8)
410: (4)
411: (8)
412: (12)
413: (12)
(dt1,))
414: (12)
415: (16)
[()],
416: (24)
417: (16)
dtype=dt2)[()], 
418: (24)
419: (8)
420: (12)
(dt1,))
421: (12)
(dt1,))
422: (12)
(dt1,))
423: (12)
424: (16)
[()],
425: (24)
426: (16)
dtype=dt2)[()],
427: (24)
428: (16)
[()],
429: (24)
430: (8)
431: (12)
(dt1,))
432: (12)
(dt1,))
433: (12)
(dt1,))
434: (12)
435: (16)
[()],
436: (24)
437: (16)
dtype=dt2)[()],
438: (24)
439: (16)
[()],
440: (24)
441: (4)
442: (8)
443: (12)
444: (12)
445: (12)
446: (12)
447: (12)
448: (12)
449: (12)
450: (8)
451: (8)
452: (0)
453: (4)
454: (8)
455: (8)
456: (8)
457: (8)
458: (8)

        def test_int_from_longdouble(self):
            x = np.longdouble(1.5)
            assert_equal(int(x), 1)
            x = np.longdouble(-10.5)
            assert_equal(int(x), -10)
        def test_numpy_scalar_relational_operators(self):
            for dt1 in np.typecodes['AllInteger']:
                assert_(1 > np.array(0, dtype=dt1)[()], "type %s failed" % (dt1,))
                assert_(not 1 < np.array(0, dtype=dt1)[()], "type %s failed" %
(dt1,))
                for dt2 in np.typecodes['AllInteger']:
                    assert_(np.array(1, dtype=dt1)[()] > np.array(0, dtype=dt2)
[()],
                     "type %s and %s failed" % (dt1, dt2))
                    assert_(not np.array(1, dtype=dt1)[()] < np.array(0,
                     "type %s and %s failed" % (dt1, dt2)))
            for dt1 in 'BHILQP':
                assert_(-1 < np.array(1, dtype=dt1)[()], "type %s failed" %
(dt1,))
                assert_(not -1 > np.array(1, dtype=dt1)[()], "type %s failed" %
(dt1,))
                assert_(-1 != np.array(1, dtype=dt1)[()], "type %s failed" %
(dt1,))
                for dt2 in 'bhilqp':
                    assert_(np.array(1, dtype=dt1)[()] > np.array(-1, dtype=dt2)
[()],
                     "type %s and %s failed" % (dt1, dt2))
                    assert_(not np.array(1, dtype=dt1)[()] < np.array(-1,
                     "type %s and %s failed" % (dt1, dt2)))
            assert_(np.array(1, dtype=dt1)[()] != np.array(-1, dtype=dt2)
[()],
                     "type %s and %s failed" % (dt1, dt2))
            for dt1 in 'bhilqp' + np.typecodes['Float']:
                assert_(1 > np.array(-1, dtype=dt1)[()], "type %s failed" %
(dt1,))
                assert_(not 1 < np.array(-1, dtype=dt1)[()], "type %s failed" %
(dt1,))
                assert_(-1 == np.array(-1, dtype=dt1)[()], "type %s failed" %
(dt1,))
                for dt2 in 'bhilqp' + np.typecodes['Float']:
                    assert_(np.array(1, dtype=dt1)[()] > np.array(-1, dtype=dt2)
[()],
                     "type %s and %s failed" % (dt1, dt2))
                    assert_(not np.array(1, dtype=dt1)[()] < np.array(-1,
                     "type %s and %s failed" % (dt1, dt2)))
            assert_(np.array(-1, dtype=dt1)[()] == np.array(-1, dtype=dt2)
[()],
                     "type %s and %s failed" % (dt1, dt2))
        def test_scalar_comparison_to_none(self):
            with warnings.catch_warnings(record=True) as w:
                warnings.filterwarnings('always', '', FutureWarning)
                assert_(not np.float32(1) == None)
                assert_(not np.str_('test') == None)
                assert_(not np.datetime64('NaT') == None)
                assert_(np.float32(1) != None)
                assert_(np.str_('test') != None)
                assert_(np.datetime64('NaT') != None)
            assert_(len(w) == 0)
            assert_(np.equal(np.datetime64('NaT'), None))
    class TestRepr:
        def _test_type_repr(self, t):
            finfo = np.finfo(t)
            last_fraction_bit_idx = finfo.nexp + finfo.nmant
            last_exponent_bit_idx = finfo.nexp
            storage_bytes = np.dtype(t).itemsize*8
            for which in ['small denorm', 'small norm']:

```

```

459: (12)           constr = np.array([0x00]*storage_bytes, dtype=np.uint8)
460: (12)           if which == 'small denorm':
461: (16)             byte = last_fraction_bit_idx // 8
462: (16)             bytebit = 7-(last_fraction_bit_idx % 8)
463: (16)             constr[byte] = 1 << bytebit
464: (12)           elif which == 'small norm':
465: (16)             byte = last_exponent_bit_idx // 8
466: (16)             bytebit = 7-(last_exponent_bit_idx % 8)
467: (16)             constr[byte] = 1 << bytebit
468: (12)           else:
469: (16)             raise ValueError('hmm')
470: (12)           val = constr.view(t)[0]
471: (12)           val_repr = repr(val)
472: (12)           val2 = t(eval(val_repr))
473: (12)           if not (val2 == 0 and val < 1e-100):
474: (16)             assert_equal(val, val2)
475: (4)             def test_float_repr(self):
476: (8)               for t in [np.float32, np.float64]:
477: (12)                 self._test_type_repr(t)
478: (0)             if not IS_PYPY:
479: (4)               class TestSizeOf:
480: (8)                 def test_equal_nbytes(self):
481: (12)                   for type in types:
482: (16)                     x = type(0)
483: (16)                     assert_(sys.getsizeof(x) > x.nbytes)
484: (8)             def test_error(self):
485: (12)               d = np.float32()
486: (12)               assert_raises(TypeError, d.__sizeof__, "a")
487: (0)             class TestMultiply:
488: (4)               def test_seq_repeat(self):
489: (8)                 accepted_types = set(np.typecodes["AllInteger"])
490: (8)                 deprecated_types = {'?'}
491: (8)                 forbidden_types = (
492: (12)                   set(np.typecodes["All"]) - accepted_types - deprecated_types)
493: (8)                 forbidden_types -= {'V'} # can't default-construct void scalars
494: (8)                 for seq_type in (list, tuple):
495: (12)                   seq = seq_type([1, 2, 3])
496: (12)                   for numpy_type in accepted_types:
497: (16)                     i = np.dtype(numpy_type).type(2)
498: (16)                     assert_equal(seq * i, seq * int(i))
499: (16)                     assert_equal(i * seq, int(i) * seq)
500: (12)                   for numpy_type in deprecated_types:
501: (16)                     i = np.dtype(numpy_type).type()
502: (16)                     assert_equal(
503: (20)                       assert_warns(DeprecationWarning, operator.mul, seq, i),
504: (20)                       seq * int(i))
505: (16)                     assert_equal(
506: (20)                       assert_warns(DeprecationWarning, operator.mul, i, seq),
507: (20)                       int(i) * seq)
508: (12)                   for numpy_type in forbidden_types:
509: (16)                     i = np.dtype(numpy_type).type()
510: (16)                     assert_raises(TypeError, operator.mul, seq, i)
511: (16)                     assert_raises(TypeError, operator.mul, i, seq)
512: (4)             def test_no_seq_repeat_basic_array_like(self):
513: (8)               class ArrayLike:
514: (12)                 def __init__(self, arr):
515: (16)                   self.arr = arr
516: (12)                 def __array__(self):
517: (16)                   return self.arr
518: (8)                 for arr_like in (ArrayLike(np.ones(3)), memoryview(np.ones(3))):
519: (12)                   assert_array_equal(arr_like * np.float32(3.), np.full(3, 3.))
520: (12)                   assert_array_equal(np.float32(3.) * arr_like, np.full(3, 3.))
521: (12)                   assert_array_equal(arr_like * np.int_(3), np.full(3, 3))
522: (12)                   assert_array_equal(np.int_(3) * arr_like, np.full(3, 3))
523: (0)             class TestNegative:
524: (4)               def test_exceptions(self):
525: (8)                 a = np.ones((), dtype=np.bool_)[()]
526: (8)                 assert_raises(TypeError, operator.neg, a)
527: (4)               def test_result(self):

```

```

528: (8)             types = np.typecodes['AllInteger'] + np.typecodes['AllFloat']
529: (8)             with suppress_warnings() as sup:
530: (12)             sup.filter(RuntimeWarning)
531: (12)             for dt in types:
532: (16)                 a = np.ones((), dtype=dt)[()]
533: (16)                 if dt in np.typecodes['UnsignedInteger']:
534: (20)                     st = np.dtype(dt).type
535: (20)                     max = st(np.iinfo(dt).max)
536: (20)                     assert_equal(operator.neg(a), max)
537: (16)                 else:
538: (20)                     assert_equal(operator.neg(a) + a, 0)
539: (0)             class TestSubtract:
540: (4)                 def test_exceptions(self):
541: (8)                     a = np.ones((), dtype=np.bool_)[()]
542: (8)                     assert_raises(TypeError, operator.sub, a, a)
543: (4)                 def test_result(self):
544: (8)                     types = np.typecodes['AllInteger'] + np.typecodes['AllFloat']
545: (8)                     with suppress_warnings() as sup:
546: (12)                         sup.filter(RuntimeWarning)
547: (12)                         for dt in types:
548: (16)                             a = np.ones((), dtype=dt)[()]
549: (16)                             assert_equal(operator.sub(a, a), 0)
550: (0)             class TestAbs:
551: (4)                 def _test_abs_func(self, absfunc, test_dtype):
552: (8)                     x = test_dtype(-1.5)
553: (8)                     assert_equal(absfunc(x), 1.5)
554: (8)                     x = test_dtype(0.0)
555: (8)                     res = absfunc(x)
556: (8)                     assert_equal(res, 0.0)
557: (8)                     x = test_dtype(-0.0)
558: (8)                     res = absfunc(x)
559: (8)                     assert_equal(res, 0.0)
560: (8)                     x = test_dtype(np.finfo(test_dtype).max)
561: (8)                     assert_equal(absfunc(x), x.real)
562: (8)                     with suppress_warnings() as sup:
563: (12)                         sup.filter(UserWarning)
564: (12)                         x = test_dtype(np.finfo(test_dtype).tiny)
565: (12)                         assert_equal(absfunc(x), x.real)
566: (8)                         x = test_dtype(np.finfo(test_dtype).min)
567: (8)                         assert_equal(absfunc(x), -x.real)
568: (4)             @pytest.mark.parametrize("dtype", floating_types + complex_floating_types)
569: (4)             def test_builtin_abs(self, dtype):
570: (8)                 if (
571: (16)                     sys.platform == "cygwin" and dtype == np.clongdouble and
572: (16)                     (
573: (20)                         _pep440.parse(platform.release().split("-")[0])
574: (20)                         < _pep440.Version("3.3.0")
575: (16)                     )
576: (8)                 ):
577: (12)                     pytest.xfail(
578: (16)                         reason="absl is computed in double precision on cygwin < 3.3"
579: (12)                     )
580: (8)                     self._test_abs_func(abs, dtype)
581: (4)             @pytest.mark.parametrize("dtype", floating_types + complex_floating_types)
582: (4)             def test_numpy_abs(self, dtype):
583: (8)                 if (
584: (16)                     sys.platform == "cygwin" and dtype == np.clongdouble and
585: (16)                     (
586: (20)                         _pep440.parse(platform.release().split("-")[0])
587: (20)                         < _pep440.Version("3.3.0")
588: (16)                     )
589: (8)                 ):
590: (12)                     pytest.xfail(
591: (16)                         reason="absl is computed in double precision on cygwin < 3.3"
592: (12)                     )
593: (8)                     self._test_abs_func(np.abs, dtype)
594: (0)             class TestBitShifts:
595: (4)                 @pytest.mark.parametrize('type_code', np.typecodes['AllInteger'])
596: (4)                 @pytest.mark.parametrize('op',

```

```

597: (8)           [operator.rshift, operator.lshift], ids=['>>', '<<'])
598: (4)           def test_shift_all_bits(self, type_code, op):
599: (8)             """Shifts where the shift amount is the width of the type or wider """
600: (8)             if (
601: (16)               USING_CLANG_CL and
602: (16)               type_code in ("l", "L") and
603: (16)               op is operator.lshift
604: (8)             ):
605: (12)               pytest.xfail("Failing on clang-cl builds")
606: (8)             dt = np.dtype(type_code)
607: (8)             nbits = dt.itemsize * 8
608: (8)             for val in [5, -5]:
609: (12)               for shift in [nbits, nbits + 4]:
610: (16)                 val_scl = np.array(val).astype(dt)[()]
611: (16)                 shift_scl = dt.type(shift)
612: (16)                 res_scl = op(val_scl, shift_scl)
613: (16)                 if val_scl < 0 and op is operator.rshift:
614: (20)                   assert_equal(res_scl, -1)
615: (16)                 else:
616: (20)                   assert_equal(res_scl, 0)
617: (16)             val_arr = np.array([val_scl]*32, dtype=dt)
618: (16)             shift_arr = np.array([shift]*32, dtype=dt)
619: (16)             res_arr = op(val_arr, shift_arr)
620: (16)             assert_equal(res_arr, res_scl)
621: (0)           class TestHash:
622: (4)             @pytest.mark.parametrize("type_code", np.typecodes['AllInteger'])
623: (4)             def test_integer_hashes(self, type_code):
624: (8)               scalar = np.dtype(type_code).type
625: (8)               for i in range(128):
626: (12)                 assert hash(i) == hash(scalar(i))
627: (4)             @pytest.mark.parametrize("type_code", np.typecodes['AllFloat'])
628: (4)             def test_float_and_complex_hashes(self, type_code):
629: (8)               scalar = np.dtype(type_code).type
630: (8)               for val in [np.pi, np.inf, 3, 6.]:
631: (12)                 numpy_val = scalar(val)
632: (12)                 if numpy_val.dtype.kind == 'c':
633: (16)                   val = complex(numpy_val)
634: (12)                 else:
635: (16)                   val = float(numpy_val)
636: (12)                 assert val == numpy_val
637: (12)                 assert hash(val) == hash(numpy_val)
638: (8)                 if hash(float(np.nan)) != hash(float(np.nan)):
639: (12)                   assert hash(scalar(np.nan)) != hash(scalar(np.nan))
640: (4)             @pytest.mark.parametrize("type_code", np.typecodes['Complex'])
641: (4)             def test_complex_hashes(self, type_code):
642: (8)               scalar = np.dtype(type_code).type
643: (8)               for val in [np.pi+1j, np.inf-3j, 3j, 6.+1j]:
644: (12)                 numpy_val = scalar(val)
645: (12)                 assert hash(complex(numpy_val)) == hash(numpy_val)
646: (0)           @contextlib.contextmanager
647: (0)           def recursionlimit(n):
648: (4)             o = sys.getrecursionlimit()
649: (4)             try:
650: (8)               sys.setrecursionlimit(n)
651: (8)               yield
652: (4)             finally:
653: (8)               sys.setrecursionlimit(o)
654: (0)             @given(sampled_from(objecty_things),
655: (7)               sampled_from(reasonable_operators_for_scalars),
656: (7)               sampled_from(types))
657: (0)             def test_operator_object_left(o, op, type_):
658: (4)               try:
659: (8)                 with recursionlimit(200):
660: (12)                   op(o, type_(1))
661: (4)                 except TypeError:
662: (8)                   pass
663: (0)               @given(sampled_from(objecty_things),
664: (7)                 sampled_from(reasonable_operators_for_scalars),
665: (7)                 sampled_from(types))

```

```

666: (0)
667: (4)
668: (8)
669: (12)
670: (4)
671: (8)
672: (0)
673: (7)
674: (7)
675: (0)
676: (4)
677: (8)
678: (4)
679: (8)
680: (0)
681: (0)
682: (0)
683: (4)
684: (8)
685: (4)
686: (8)
687: (4)
688: (8)
689: (4)
690: (8)
691: (0)
692: (0)
693: (0)
694: (4)
695: (8)
696: (4)
697: (8)
698: (4)
699: (8)
700: (4)
701: (8)
702: (0)
703: (0)
704: (8)
705: (8)
706: (8)
707: (0)
708: (4)
709: (4)
710: (4)
711: (4)
712: (8)
713: (0)
714: (0)
715: (8)
716: (8)
717: (8)
718: (8)
719: (12)
720: (8)
721: (0)
722: (4)
723: (4)
724: (4)
725: (4)
726: (8)
727: (0)
728: (0)
729: (4)
730: (4)
731: (8)
732: (4)
733: (4)
734: (0)

def test_operator_object_right(o, op, type_):
    try:
        with recursionlimit(200):
            op(type_(1), o)
    except TypeError:
        pass
@given(sampled_from(reasonable_operators_for_scalars),
       sampled_from(types),
       sampled_from(types))
def test_operator_scalars(op, type1, type2):
    try:
        op(type1(1), type2(1))
    except TypeError:
        pass
@ pytest.mark.parametrize("op", reasonable_operators_for_scalars)
@ pytest.mark.parametrize("val", [None, 2**64])
def test_longdouble_inf_loop(op, val):
    try:
        op(np.longdouble(3), val)
    except TypeError:
        pass
    try:
        op(val, np.longdouble(3))
    except TypeError:
        pass
@ pytest.mark.parametrize("op", reasonable_operators_for_scalars)
@ pytest.mark.parametrize("val", [None, 2**64])
def test_clongdouble_inf_loop(op, val):
    try:
        op(np.clongdouble(3), val)
    except TypeError:
        pass
    try:
        op(val, np.clongdouble(3))
    except TypeError:
        pass
@ pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
@ pytest.mark.parametrize("operation", [
    lambda min, max: max + max,
    lambda min, max: min - max,
    lambda min, max: max * max], ids=["+", "-", "*"])
def test_scalar_integer_operation_overflow(dtype, operation):
    st = np.dtype(dtype).type
    min = st(np.iinfo(dtype).min)
    max = st(np.iinfo(dtype).max)
    with pytest.warns(RuntimeWarning, match="overflow encountered"):
        operation(min, max)
@ pytest.mark.parametrize("dtype", np.typecodes["Integer"])
@ pytest.mark.parametrize("operation", [
    lambda min, neg_1: -min,
    lambda min, neg_1: abs(min),
    lambda min, neg_1: min * neg_1,
    pytest.param(lambda min, neg_1: min // neg_1,
                marks=pytest.mark.skip(reason="broken on some platforms")),
    lambda min, neg_1: min / neg_1], ids=["neg", "abs", "*", "//"])
def test_scalar_signed_integer_overflow(dtype, operation):
    st = np.dtype(dtype).type
    min = st(np.iinfo(dtype).min)
    neg_1 = st(-1)
    with pytest.warns(RuntimeWarning, match="overflow encountered"):
        operation(min, neg_1)
@ pytest.mark.parametrize("dtype", np.typecodes["UnsignedInteger"])
def test_scalar_unsigned_integer_overflow(dtype):
    val = np.dtype(dtype).type(8)
    with pytest.warns(RuntimeWarning, match="overflow encountered"):
        -val
    zero = np.dtype(dtype).type(0)
    -zero # does not warn
@ pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

735: (0)
736: (8)
737: (8)
738: (0)
739: (4)
740: (4)
741: (4)
742: (4)
743: (8)
744: (0)
745: (4)
746: (4)
747: (4)
748: (4)
749: (4)
750: (4)
751: (4)
752: (4)
753: (4)
754: (4)
755: (4)
756: (4)
757: (4)
758: (0)
759: (0)
760: (0)
761: (0)
762: (4)
763: (4)
764: (4)
765: (4)
766: (4)
767: (4)
768: (4)
769: (4)
770: (4)
771: (4)
772: (8)
773: (4)
774: (8)
775: (4)
776: (8)
777: (4)
778: (8)
779: (4)
780: (4)
781: (4)
782: (4)
783: (4)
784: (4)
785: (0)
786: (4)
787: (4)
788: (4)
789: (0)
790: (0)
791: (0)
792: (0)
793: (4)
794: (8)
795: (4)
796: (8)
797: (4)
798: (15)
799: (4)
800: (4)
801: (4)
802: (8)
803: (4)

    @pytest.mark.parametrize("operation", [
        lambda val, zero: val // zero,
        lambda val, zero: val % zero, ],
        ids=["//", "%"])
def test_scalar_integer_operation_divbyzero(dtype, operation):
    st = np.dtype(dtype).type
    val = st(100)
    zero = st(0)
    with pytest.warns(RuntimeWarning, match="divide by zero"):
        operation(val, zero)

    ops_with_names = [
        ("__lt__", "__gt__", operator.lt, True),
        ("__le__", "__ge__", operator.le, True),
        ("__eq__", "__eq__", operator.eq, True),
        ("__ne__", "__ne__", operator.ne, True),
        ("__gt__", "__lt__", operator.gt, True),
        ("__ge__", "__le__", operator.ge, True),
        ("__floordiv__", "__rfloordiv__", operator.floordiv, False),
        ("__truediv__", "__rtruediv__", operator.truediv, False),
        ("__add__", "__radd__", operator.add, False),
        ("__mod__", "__rmod__", operator.mod, False),
        ("__mul__", "__rmul__", operator.mul, False),
        ("__pow__", "__rpow__", operator.pow, False),
        ("__sub__", "__rsub__", operator.sub, False),
    ]
    @pytest.mark.parametrize(["__op__", "__rop__", "op", "cmp"], ops_with_names)
    @pytest.mark.parametrize("sctype", [np.float32, np.float64, np.longdouble])
    def test_subclass_deferral(sctype, __op__, __rop__, op, cmp):
        """
        This test covers scalar subclass deferral. Note that this is exceedingly
        complicated, especially since it tends to fall back to the array paths and
        these additionally add the "array priority" mechanism.
        The behaviour was modified subtly in 1.22 (to make it closer to how Python
        scalars work). Due to its complexity and the fact that subclassing NumPy
        scalars is probably a bad idea to begin with. There is probably room
        for adjustments here.
        """
        class myf_simple1(sctype):
            pass
        class myf_simple2(sctype):
            pass
        def op_func(self, other):
            return __op__
        def rop_func(self, other):
            return __rop__
        myf_op = type("myf_op", (sctype,), {__op__: op_func, __rop__: rop_func})
        res = op(myf_simple1(1), myf_simple2(2))
        assert type(res) == sctype or type(res) == np.bool_
        assert op(myf_simple1(1), myf_simple2(2)) == op(1, 2) # inherited
        assert op(myf_op(1), myf_simple1(2)) == __op__
        assert op(myf_simple1(1), myf_op(2)) == op(1, 2) # inherited
        def test_longdouble_complex():
            x = np.longdouble(1)
            assert x + 1j == 1+1j
            assert 1j + x == 1+1j
        @pytest.mark.parametrize(["__op__", "__rop__", "op", "cmp"], ops_with_names)
        @pytest.mark.parametrize("subtype", [float, int, complex, np.float16])
        @np._no_nep50_warning()
        def test_pyscalar_subclasses(subtype, __op__, __rop__, op, cmp):
            def op_func(self, other):
                return __op__
            def rop_func(self, other):
                return __rop__
            myt = type("myt", (subtype,), {
                __op__: op_func, __rop__: rop_func, "__array_ufunc__": None})
            assert op(myt(1), np.float64(2)) == __op__
            assert op(np.float64(1), myt(2)) == __rop__
            if op in {operator.mod, operator.floordiv} and subtype == complex:
                return # module is not support for complex. Do not test.
            if __rop__ == __op__:

```

```

804: (8)         return
805: (4)     myt = type("myt", (subtype,), {__rop__: rop_func})
806: (4)     res = op(myt(1), np.float16(2))
807: (4)     expected = op(subtype(1), np.float16(2))
808: (4)     assert res == expected
809: (4)     assert type(res) == type(expected)
810: (4)     res = op(np.float32(2), myt(1))
811: (4)     expected = op(np.float32(2), subtype(1))
812: (4)     assert res == expected
813: (4)     assert type(res) == type(expected)
814: (4)     res = op(myt(1), np.longdouble(2))
815: (4)     expected = op(subtype(1), np.longdouble(2))
816: (4)     assert res == expected
817: (4)     assert type(res) == type(expected)
818: (4)     res = op(np.float32(2), myt(1))
819: (4)     expected = op(np.longdouble(2), subtype(1))
820: (4)     assert res == expected
-----
```

## File 122 - test\_scalarprint.py:

```

1: (0)         """ Test printing of scalar types.
2: (0)         """
3: (0)         import code
4: (0)         import platform
5: (0)         import pytest
6: (0)         import sys
7: (0)         from tempfile import TemporaryFile
8: (0)         import numpy as np
9: (0)         from numpy.testing import assert_, assert_equal, assert_raises, IS_MUSL
10: (0)        class TestRealScalars:
11: (4)            def test_str(self):
12: (8)                svals = [0.0, -0.0, 1, -1, np.inf, -np.inf, np.nan]
13: (8)                styps = [np.float16, np.float32, np.float64, np.longdouble]
14: (8)                wanted = [
15: (13)                    ['0.0', '0.0', '0.0', '0.0'],
16: (13)                    ['-0.0', '-0.0', '-0.0', '-0.0'],
17: (13)                    ['1.0', '1.0', '1.0', '1.0'],
18: (13)                    ['-1.0', '-1.0', '-1.0', '-1.0'],
19: (13)                    ['inf', 'inf', 'inf', 'inf'],
20: (13)                    ['-inf', '-inf', '-inf', '-inf'],
21: (13)                    ['nan', 'nan', 'nan', 'nan']]
22: (8)                for wants, val in zip(wanted, svals):
23: (12)                    for want, styp in zip(wants, styps):
24: (16)                        msg = 'for str({}({}))'.format(np.dtype(styp).name, repr(val))
25: (16)                        assert_equal(str(styp(val)), want, err_msg=msg)
26: (4)            def test_scalar_cutoffs(self):
27: (8)                def check(v):
28: (12)                    assert_equal(str(np.float64(v)), str(v))
29: (12)                    assert_equal(str(np.float64(v)), repr(v))
30: (12)                    assert_equal(repr(np.float64(v)), repr(v))
31: (12)                    assert_equal(repr(np.float64(v)), str(v))
32: (8)                check(1.12345678901234567890)
33: (8)                check(0.0112345678901234567890)
34: (8)                check(1e-5)
35: (8)                check(1e-4)
36: (8)                check(1e15)
37: (8)                check(1e16)
38: (4)            def test_py2_float_print(self):
39: (8)                x = np.double(0.199999999999)
40: (8)                with TemporaryFile('r+t') as f:
41: (12)                    print(x, file=f)
42: (12)                    f.seek(0)
43: (12)                    output = f.read()
44: (8)                    assert_equal(output, str(x) + '\n')
45: (8)                    def userinput():
46: (12)                        yield 'np.sqrt(2)'
47: (12)                        raise EOFError

```







```

224: (25)           "1.2" if tp != np.float16 else "1.2002")
225: (12)           assert_equal(fpos(tp('1.'), trim='-' ), "1")
226: (12)           assert_equal(fpos(tp('1.001'), precision=1, trim='-' ), "1")
227: (4)            @pytest.mark.skipif(not platform.machine().startswith("ppc64"),
228: (24)                           reason="only applies to ppc float128 values")
229: (4)            def test_ppc64_ibm_double_double128(self):
230: (8)              x = np.float128('2.123123123123123123123123e-286')
231: (8)              got = [str(x/np.float128('2e' + str(i))) for i in range(0,40)]
232: (8)              expected = [
233: (12)                "1.06156156156156156156156156156156156156157e-286",
234: (12)                "1.06156156156156156156156156156156158e-287",
235: (12)                "1.06156156156156156156156156156156159e-288",
236: (12)                "1.0615615615615615615615615615615615615616e-289",
237: (12)                "1.06156156156156156156156156156156156157e-290",
238: (12)                "1.06156156156156156156156156156156156156e-291",
239: (12)                "1.0615615615615615615615615615615615615616e-292",
240: (12)                "1.0615615615615615615615615615615615615615e-293",
241: (12)                "1.061561561561561561561561561561561562e-294",
242: (12)                "1.06156156156156156156156156156156155e-295",
243: (12)                "1.0615615615615615615615615615615616e-296",
244: (12)                "1.06156156156156156156156156156156e-297",
245: (12)                "1.06156156156156156156156156157e-298",
246: (12)                "1.0615615615615615615615615616e-299",
247: (12)                "1.06156156156156156156156156156e-300",
248: (12)                "1.06156156156156156156156155e-301",
249: (12)                "1.0615615615615615615615616e-302",
250: (12)                "1.061561561561561561562e-303",
251: (12)                "1.06156156156156156156156e-304",
252: (12)                "1.0615615615615615615618e-305",
253: (12)                "1.06156156156156156156e-306",
254: (12)                "1.06156156156156157e-307",
255: (12)                "1.06156156156156156156e-308",
256: (12)                "1.06156156156156e-309",
257: (12)                "1.06156156156157e-310",
258: (12)                "1.0615615615616e-311",
259: (12)                "1.06156156156e-312",
260: (12)                "1.06156156154e-313",
261: (12)                "1.0615615616e-314",
262: (12)                "1.06156156e-315",
263: (12)                "1.06156155e-316",
264: (12)                "1.061562e-317",
265: (12)                "1.06156e-318",
266: (12)                "1.06155e-319",
267: (12)                "1.0617e-320",
268: (12)                "1.06e-321",
269: (12)                "1.04e-322",
270: (12)                "1e-323",
271: (12)                "0.0",
272: (12)                "0.0"]
273: (8)            assert_equal(got, expected)
274: (8)            a = np.float128('2')/np.float128('3')
275: (8)            b = np.float128(str(a))
276: (8)            assert_equal(str(a), str(b))
277: (8)            assert_(a != b)
278: (4)            def float32_roundtrip(self):
279: (8)              x = np.float32(1024 - 2**-14)
280: (8)              y = np.float32(1024 - 2**-13)
281: (8)              assert_(repr(x) != repr(y))
282: (8)              assert_equal(np.float32(repr(x)), x)
283: (8)              assert_equal(np.float32(repr(y)), y)
284: (4)            def float64_vs_python(self):
285: (8)              assert_equal(repr(np.float64(0.1)), repr(0.1))
286: (8)              assert_(repr(np.float64(0.2000000000000004)) != repr(0.2))

```

-----

File 123 - test\_scalar\_ctors.py:

1: (0) """

```

2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (4)
8: (4)
9: (0)
10: (4)
11: (8)
12: (8)
13: (8)
14: (8)
15: (8)
16: (8)
17: (4)
18: (8)
19: (8)
20: (8)
21: (8)
22: (8)
23: (8)
24: (8)
25: (8)
26: (8)
27: (8)
28: (8)
29: (8)
30: (8)
31: (8)
32: (8)
33: (8)
34: (8)
35: (0)
36: (4)
37: (8)
38: (8)
39: (8)
40: (8)
41: (8)
42: (12)
43: (8)
44: (12)
45: (8)
46: (4)
47: (8)
48: (8)
49: (8)
50: (12)
51: (4)
52: (8)
53: (12)
54: (4)
55: (8)
56: (12)
57: (0)
58: (4)
59: (8)
60: (4)
61: (8)
62: (12)
63: (0)
64: (0)
65: (0)
66: (0)
67: (0)
68: (4)
69: (4)
70: (8)

Test the scalar constructors, which also do type-coercion
"""
import pytest
import numpy as np
from numpy.testing import (
    assert_equal, assert_almost_equal, assert_warns,
)
class TestFromString:
    def test_floating(self):
        fsingle = np.single('1.234')
        fdouble = np.double('1.234')
        flongdouble = np.longdouble('1.234')
        assert_almost_equal(fsingle, 1.234)
        assert_almost_equal(fdouble, 1.234)
        assert_almost_equal(flongdouble, 1.234)
    def test_floating_overflow(self):
        """ Strings containing an unrepresentable float overflow """
        fhalf = np.half('1e10000')
        assert_equal(fhalf, np.inf)
        fsingle = np.single('1e10000')
        assert_equal(fsingle, np.inf)
        fdouble = np.double('1e10000')
        assert_equal(fdouble, np.inf)
        flongdouble = assert_warns(RuntimeWarning, np.longdouble, '1e10000')
        assert_equal(flongdouble, np.inf)
        fhalf = np.half('-1e10000')
        assert_equal(fhalf, -np.inf)
        fsingle = np.single('-1e10000')
        assert_equal(fsingle, -np.inf)
        fdouble = np.double('-1e10000')
        assert_equal(fdouble, -np.inf)
        flongdouble = assert_warns(RuntimeWarning, np.longdouble, '-1e10000')
        assert_equal(flongdouble, -np.inf)
class TestExtraArgs:
    def test_superclass(self):
        s = np.str_(b'\x61', encoding='unicode-escape')
        assert s == 'a'
        s = np.str_(b'\x61', 'unicode-escape')
        assert s == 'a'
        with pytest.raises(UnicodeDecodeError):
            np.str_(b'\xx', encoding='unicode-escape')
        with pytest.raises(UnicodeDecodeError):
            np.str_(b'\xx', 'unicode-escape')
        assert np.bytes_(-2) == b'-2'
    def test_datetime(self):
        dt = np.datetime64('2000-01', ('M', 2))
        assert np.datetime_data(dt) == ('M', 2)
        with pytest.raises(TypeError):
            np.datetime64('2000', garbage=True)
    def test_bool(self):
        with pytest.raises(TypeError):
            np.bool_(False, garbage=True)
    def test_void(self):
        with pytest.raises(TypeError):
            np.void(b'test', garbage=True)
class TestFromInt:
    def test_intp(self):
        assert_equal(1024, np.intp(1024))
    def test_uint64_from_negative(self):
        with pytest.warns(DeprecationWarning):
            assert_equal(np.uint64(-2), np.uint64(18446744073709551614))
int_types = [np.byte, np.short, np.intc, np.int_, np.longlong]
uint_types = [np.ubyte, np ushort, np.uintc, np.uint, np ulonglong]
float_types = [np.half, np.single, np.double, np.longdouble]
cfloat_types = [np.csingle, np.cdouble, np.clongdouble]
class TestArrayFromScalar:
    """ gh-15467 """
    def _do_test(self, t1, t2):
        x = t1(2)

```

```

71: (8)             arr = np.array(x, dtype=t2)
72: (8)             if t2 is None:
73: (12)                 assert arr.dtype.type is t1
74: (8)             else:
75: (12)                 assert arr.dtype.type is t2
76: (4)             @pytest.mark.parametrize('t1', int_types + uint_types)
77: (4)             @pytest.mark.parametrize('t2', int_types + uint_types + [None])
78: (4)             def test_integers(self, t1, t2):
79: (8)                 return self._do_test(t1, t2)
80: (4)             @pytest.mark.parametrize('t1', float_types)
81: (4)             @pytest.mark.parametrize('t2', float_types + [None])
82: (4)             def test_reals(self, t1, t2):
83: (8)                 return self._do_test(t1, t2)
84: (4)             @pytest.mark.parametrize('t1', cfloat_types)
85: (4)             @pytest.mark.parametrize('t2', cfloat_types + [None])
86: (4)             def test_complex(self, t1, t2):
87: (8)                 return self._do_test(t1, t2)
88: (0)             @pytest.mark.parametrize("length",
89: (8)                 [5, np.int8(5), np.array(5, dtype=np.uint16)])
90: (0)             def test_void_via_length(length):
91: (4)                 res = np.void(length)
92: (4)                 assert type(res) is np.void
93: (4)                 assert res.item() == b"\0" * 5
94: (4)                 assert res.dtype == "V5"
95: (0)             @pytest.mark.parametrize("bytes_",
96: (8)                 [b"spam", np.array(567.)])
97: (0)             def test_void_from_byteslike(bytes_):
98: (4)                 res = np.void(bytes_)
99: (4)                 expected = bytes(bytes_)
100: (4)                 assert type(res) is np.void
101: (4)                 assert res.item() == expected
102: (4)                 res = np.void(bytes_, dtype="V100")
103: (4)                 assert type(res) is np.void
104: (4)                 assert res.item()[:len(expected)] == expected
105: (4)                 assert res.item()[len(expected):] == b"\0" * (res nbytes - len(expected))
106: (4)                 res = np.void(bytes_, dtype="V4")
107: (4)                 assert type(res) is np.void
108: (4)                 assert res.item() == expected[:4]
109: (0)             def test_void_arraylike_trumps_byteslike():
110: (4)                 m = memoryview(b"just one mintleaf?")
111: (4)                 res = np.void(m)
112: (4)                 assert type(res) is np.ndarray
113: (4)                 assert res.dtype == "V1"
114: (4)                 assert res.shape == (18,)
115: (0)             def test_void_dtype_arg():
116: (4)                 res = np.void((1, 2), dtype="i,i")
117: (4)                 assert res.item() == (1, 2)
118: (4)                 res = np.void((2, 3), "i,i")
119: (4)                 assert res.item() == (2, 3)
120: (0)             @pytest.mark.parametrize("data",
121: (8)                 [5, np.int8(5), np.array(5, dtype=np.uint16)])
122: (0)             def test_void_from_integer_with_dtype(data):
123: (4)                 res = np.void(data, dtype="i,i")
124: (4)                 assert type(res) is np.void
125: (4)                 assert res.dtype == "i,i"
126: (4)                 assert res["f0"] == 5 and res["f1"] == 5
127: (0)             def test_void_from_structure():
128: (4)                 dtype = np.dtype([('s', [(('f', 'f8'), ('u', 'U1'))]), ('i', 'i2')])
129: (4)                 data = np.array(((1., 'a'), 2), dtype=dtype)
130: (4)                 res = np.void(data[()], dtype=dtype)
131: (4)                 assert type(res) is np.void
132: (4)                 assert res.dtype == dtype
133: (4)                 assert res == data[()]
134: (0)             def test_void_bad_dtype():
135: (4)                 with pytest.raises(TypeError,
136: (12)                     match="void: descr must be a `void.*int64`"):
137: (8)                     np.void(4, dtype="i8")
138: (4)                 with pytest.raises(TypeError,
139: (12)                     match=r"void: descr must be a `void.*\(\d,\)`"):
```

140: (8) np.void(4, dtype="4i")

-----  
File 124 - test\_scalar\_methods.py:

```

1: (0) """
2: (0)     Test the scalar constructors, which also do type-coercion
3: (0) """
4: (0)     import fractions
5: (0)     import platform
6: (0)     import types
7: (0)     from typing import Any, Type
8: (0)     import pytest
9: (0)     import numpy as np
10: (0)    from numpy.testing import assert_equal, assert_raises, IS_MSL
11: (0)    class TestAsIntegerRatio:
12: (4)        @pytest.mark.parametrize("ftype", [
13: (8)            np.half, np.single, np.double, np.longdouble])
14: (4)        @pytest.mark.parametrize("f, ratio", [
15: (8)            (0.875, (7, 8)),
16: (8)            (-0.875, (-7, 8)),
17: (8)            (0.0, (0, 1)),
18: (8)            (11.5, (23, 2)),
19: (8)            ])
20: (4)        def test_small(self, ftype, f, ratio):
21: (8)            assert_equal(ftype(f).as_integer_ratio(), ratio)
22: (4)        @pytest.mark.parametrize("ftype", [
23: (8)            np.half, np.single, np.double, np.longdouble])
24: (4)        def test_simple_fractions(self, ftype):
25: (8)            R = fractions.Fraction
26: (8)            assert_equal(R(0, 1),
27: (21)                R(*ftype(0.0).as_integer_ratio()))
28: (8)            assert_equal(R(5, 2),
29: (21)                R(*ftype(2.5).as_integer_ratio()))
30: (8)            assert_equal(R(1, 2),
31: (21)                R(*ftype(0.5).as_integer_ratio()))
32: (8)            assert_equal(R(-2100, 1),
33: (21)                R(*ftype(-2100.0).as_integer_ratio()))
34: (4)        @pytest.mark.parametrize("ftype", [
35: (8)            np.half, np.single, np.double, np.longdouble])
36: (4)        def test_errors(self, ftype):
37: (8)            assert_raises(OverflowError, ftype('inf').as_integer_ratio)
38: (8)            assert_raises(OverflowError, ftype('-inf').as_integer_ratio)
39: (8)            assert_raises(ValueError, ftype('nan').as_integer_ratio)
40: (4)        def test_against_known_values(self):
41: (8)            R = fractions.Fraction
42: (8)            assert_equal(R(1075, 512),
43: (21)                R(*np.half(2.1).as_integer_ratio()))
44: (8)            assert_equal(R(-1075, 512),
45: (21)                R(*np.half(-2.1).as_integer_ratio()))
46: (8)            assert_equal(R(4404019, 2097152),
47: (21)                R(*np.single(2.1).as_integer_ratio()))
48: (8)            assert_equal(R(-4404019, 2097152),
49: (21)                R(*np.single(-2.1).as_integer_ratio()))
50: (8)            assert_equal(R(4728779608739021, 2251799813685248),
51: (21)                R(*np.double(2.1).as_integer_ratio()))
52: (8)            assert_equal(R(-4728779608739021, 2251799813685248),
53: (21)                R(*np.double(-2.1).as_integer_ratio()))
54: (4)        @pytest.mark.parametrize("ftype, frac_vals, exp_vals", [
55: (8)            (np.half, [0.0, 0.01154830649280303, 0.31082276347447274,
56: (19)                0.527350517124794, 0.8308562335072596],
57: (18)                [0, 1, 0, -8, 12]),
58: (8)            (np.single, [0.0, 0.09248576989263226, 0.8160498218131407,
59: (21)                0.17389442853722373, 0.7956044195067877],
60: (20)                [0, 12, 10, 17, -26]),
61: (8)            (np.double, [0.0, 0.031066908499895136, 0.5214135908877832,
62: (21)                0.45780736035689296, 0.5906586745934036],
63: (20)                [0, -801, 51, 194, -653]),
```

```

64: (8)           pytest.param(
65: (12)         np.longdouble,
66: (12)         [0.0, 0.20492557202724854, 0.4277180662199366, 0.9888085019891495,
67: (13)         0.9620175814461964],
68: (12)         [0, -7400, 14266, -7822, -8721],
69: (12)         marks=[
70: (16)           pytest.mark.skipif(
71: (20)             np.finfo(np.double) == np.finfo(np.longdouble),
72: (20)             reason="long double is same as double"),
73: (16)           pytest.mark.skipif(
74: (20)             platform.machine().startswith("ppc"),
75: (20)             reason="IBM double double"),
76: (12)         ]
77: (8)       )
78: (4)
79: (4)     def test_roundtrip(self, ftype, frac_vals, exp_vals):
80: (8)       for frac, exp in zip(frac_vals, exp_vals):
81: (12)         f = np.ldexp(ftype(frac), exp)
82: (12)         assert f.dtype == ftype
83: (12)         n, d = f.as_integer_ratio()
84: (12)         try:
85: (16)           nf = np.longdouble(n)
86: (16)           df = np.longdouble(d)
87: (16)           if not np.isfinite(df):
88: (20)             raise OverflowError
89: (12)         except (OverflowError, RuntimeWarning):
90: (16)           pytest.skip("longdouble too small on this platform")
91: (12)           assert_equal(nf / df, f, "{} / {}".format(n, d))
92: (0)     class TestIsInteger:
93: (4)       @pytest.mark.parametrize("str_value", ["inf", "nan"])
94: (4)       @pytest.mark.parametrize("code", np.typecodes["Float"])
95: (4)       def test_special(self, code: str, str_value: str) -> None:
96: (8)         cls = np.dtype(code).type
97: (8)         value = cls(str_value)
98: (8)         assert not value.is_integer()
99: (4)       @pytest.mark.parametrize(
100: (8)         "code", np.typecodes["Float"] + np.typecodes["AllInteger"]
101: (4)     )
102: (4)     def test_true(self, code: str) -> None:
103: (8)       float_array = np.arange(-5, 5).astype(code)
104: (8)       for value in float_array:
105: (12)         assert value.is_integer()
106: (4)       @pytest.mark.parametrize("code", np.typecodes["Float"])
107: (4)       def test_false(self, code: str) -> None:
108: (8)         float_array = np.arange(-5, 5).astype(code)
109: (8)         float_array *= 1.1
110: (8)         for value in float_array:
111: (12)           if value == 0:
112: (16)             continue
113: (12)           assert not value.is_integer()
114: (0)     class TestClassGetItem:
115: (4)       @pytest.mark.parametrize("cls", [
116: (8)         np.number,
117: (8)         np.integer,
118: (8)         np.inexact,
119: (8)         np.unsignedinteger,
120: (8)         np.signedinteger,
121: (8)         np.floating,
122: (4)     ])
123: (4)     def test_abc(self, cls: Type[np.number]) -> None:
124: (8)       alias = cls[Any]
125: (8)       assert isinstance(alias, types.GenericAlias)
126: (8)       assert alias.__origin__ is cls
127: (4)     def test_abc_complexfloating(self) -> None:
128: (8)       alias = np.complexfloating[Any, Any]
129: (8)       assert isinstance(alias, types.GenericAlias)
130: (8)       assert alias.__origin__ is np.complexfloating
131: (4)     @pytest.mark.parametrize("arg_len", range(4))
132: (4)     def test_abc_complexfloating_subscript_tuple(self, arg_len: int) -> None:

```

```

133: (8)             arg_tup = (Any,) * arg_len
134: (8)             if arg_len in (1, 2):
135: (12)               assert np.complexfloating[arg_tup]
136: (8)
137: (12)               match = f"Too {'few' if arg_len == 0 else 'many'} arguments"
138: (12)               with pytest.raises(TypeError, match=match):
139: (16)                 np.complexfloating[arg_tup]
140: (4) @pytest.mark.parametrize("cls", [np.generic, np.flexible, np.character])
141: (4) def test_abc_non_numeric(self, cls: Type[np.generic]) -> None:
142: (8)     with pytest.raises(TypeError):
143: (12)       cls[Any]
144: (4) @pytest.mark.parametrize("code", np.typecodes["All"])
145: (4) def test_concrete(self, code: str) -> None:
146: (8)     cls = np.dtype(code).type
147: (8)     with pytest.raises(TypeError):
148: (12)       cls[Any]
149: (4) @pytest.mark.parametrize("arg_len", range(4))
150: (4) def test_subscript_tuple(self, arg_len: int) -> None:
151: (8)     arg_tup = (Any,) * arg_len
152: (8)     if arg_len == 1:
153: (12)       assert np.number[arg_tup]
154: (8)     else:
155: (12)       with pytest.raises(TypeError):
156: (16)         np.number[arg_tup]
157: (4) def test_subscript_scalar(self) -> None:
158: (8)     assert np.number[Any]
159: (0) class TestBitCount:
160: (4) @pytest.mark.parametrize("itype", np.sctypes['int']+np.sctypes['uint'])
161: (4) def test_small(self, itype):
162: (8)     for a in range(max(np.iinfo(itype).min, 0), 128):
163: (12)       msg = f"Smoke test for {itype}({{a}}).bit_count()"
164: (12)       assert itype(a).bit_count() == bin(a).count("1"), msg
165: (4) def test_bit_count(self):
166: (8)     for exp in [10, 17, 63]:
167: (12)       a = 2**exp
168: (12)       assert np.uint64(a).bit_count() == 1
169: (12)       assert np.uint64(a - 1).bit_count() == exp
170: (12)       assert np.uint64(a ^ 63).bit_count() == 7
171: (12)       assert np.uint64((a - 1) ^ 510).bit_count() == exp - 8

```

---

## File 125 - test\_shape\_base.py:

```

1: (0)             import pytest
2: (0)             import numpy as np
3: (0)             from numpy.core import (
4: (4)               array, arange, atleast_1d, atleast_2d, atleast_3d, block, vstack, hstack,
5: (4)               newaxis, concatenate, stack
6: (4)             )
7: (0)             from numpy.core.shape_base import (_block_dispatcher, _block_setup,
8: (35)                           _block_concatenate, _block_slicing)
9: (0)             from numpy.testing import (
10: (4)               assert_, assert_raises, assert_array_equal, assert_equal,
11: (4)               assert_raises_regex, assert_warns, IS_PYPY
12: (4)             )
13: (0) class TestAtleast1d:
14: (4)     def test_0D_array(self):
15: (8)       a = array(1)
16: (8)       b = array(2)
17: (8)       res = [atleast_1d(a), atleast_1d(b)]
18: (8)       desired = [array([1]), array([2])]
19: (8)       assert_array_equal(res, desired)
20: (4)     def test_1D_array(self):
21: (8)       a = array([1, 2])
22: (8)       b = array([2, 3])
23: (8)       res = [atleast_1d(a), atleast_1d(b)]
24: (8)       desired = [array([1, 2]), array([2, 3])]
25: (8)       assert_array_equal(res, desired)

```

```

26: (4)          def test_2D_array(self):
27: (8)            a = array([[1, 2], [1, 2]])
28: (8)            b = array([[2, 3], [2, 3]])
29: (8)            res = [atleast_1d(a), atleast_1d(b)]
30: (8)            desired = [a, b]
31: (8)            assert_array_equal(res, desired)
32: (4)          def test_3D_array(self):
33: (8)            a = array([[1, 2], [1, 2]])
34: (8)            b = array([[2, 3], [2, 3]])
35: (8)            a = array([a, a])
36: (8)            b = array([b, b])
37: (8)            res = [atleast_1d(a), atleast_1d(b)]
38: (8)            desired = [a, b]
39: (8)            assert_array_equal(res, desired)
40: (4)          def test_r1array(self):
41: (8)            """ Test to make sure equivalent Travis O's r1array function
42: (8)
43: (8)            assert_(atleast_1d(3).shape == (1,))
44: (8)            assert_(atleast_1d(3j).shape == (1,))
45: (8)            assert_(atleast_1d(3.0).shape == (1,))
46: (8)            assert_(atleast_1d([[2, 3], [4, 5]]).shape == (2, 2))
47: (0)          class TestAtleast2d:
48: (4)            def test_0D_array(self):
49: (8)              a = array(1)
50: (8)              b = array(2)
51: (8)              res = [atleast_2d(a), atleast_2d(b)]
52: (8)              desired = [array([[1]]), array([[2]])]
53: (8)              assert_array_equal(res, desired)
54: (4)            def test_1D_array(self):
55: (8)              a = array([1, 2])
56: (8)              b = array([2, 3])
57: (8)              res = [atleast_2d(a), atleast_2d(b)]
58: (8)              desired = [array([[1, 2]]), array([[2, 3]])]
59: (8)              assert_array_equal(res, desired)
60: (4)            def test_2D_array(self):
61: (8)              a = array([[1, 2], [1, 2]])
62: (8)              b = array([[2, 3], [2, 3]])
63: (8)              res = [atleast_2d(a), atleast_2d(b)]
64: (8)              desired = [a, b]
65: (8)              assert_array_equal(res, desired)
66: (4)            def test_3D_array(self):
67: (8)              a = array([[1, 2], [1, 2]])
68: (8)              b = array([[2, 3], [2, 3]])
69: (8)              a = array([a, a])
70: (8)              b = array([b, b])
71: (8)              res = [atleast_2d(a), atleast_2d(b)]
72: (8)              desired = [a, b]
73: (8)              assert_array_equal(res, desired)
74: (4)            def test_r2array(self):
75: (8)              """ Test to make sure equivalent Travis O's r2array function
76: (8)
77: (8)              assert_(atleast_2d(3).shape == (1, 1))
78: (8)              assert_(atleast_2d([3j, 1]).shape == (1, 2))
79: (8)              assert_(atleast_2d([[3, 1], [4, 5]], [[3, 5], [1, 2]]].shape == (2,
2, 2))
80: (0)          class TestAtleast3d:
81: (4)            def test_0D_array(self):
82: (8)              a = array(1)
83: (8)              b = array(2)
84: (8)              res = [atleast_3d(a), atleast_3d(b)]
85: (8)              desired = [array([[1]]), array([[2]])]
86: (8)              assert_array_equal(res, desired)
87: (4)            def test_1D_array(self):
88: (8)              a = array([1, 2])
89: (8)              b = array([2, 3])
90: (8)              res = [atleast_3d(a), atleast_3d(b)]
91: (8)              desired = [array([[1, 2]]), array([[2, 3]])]
92: (8)              assert_array_equal(res, desired)
93: (4)            def test_2D_array(self):

```

```

94: (8)             a = array([[1, 2], [1, 2]])
95: (8)             b = array([[2, 3], [2, 3]])
96: (8)             res = [atleast_3d(a), atleast_3d(b)]
97: (8)             desired = [a[:, :, newaxis], b[:, :, newaxis]]
98: (8)             assert_array_equal(res, desired)
99: (4)             def test_3D_array(self):
100: (8)                 a = array([[1, 2], [1, 2]])
101: (8)                 b = array([[2, 3], [2, 3]])
102: (8)                 a = array([a, a])
103: (8)                 b = array([b, b])
104: (8)                 res = [atleast_3d(a), atleast_3d(b)]
105: (8)                 desired = [a, b]
106: (8)                 assert_array_equal(res, desired)
107: (0)             class TestHstack:
108: (4)                 def test_non_iterable(self):
109: (8)                     assert_raises(TypeError, hstack, 1)
110: (4)                 def test_empty_input(self):
111: (8)                     assert_raises(ValueError, hstack, ())
112: (4)                 def test_0D_array(self):
113: (8)                     a = array(1)
114: (8)                     b = array(2)
115: (8)                     res = hstack([a, b])
116: (8)                     desired = array([1, 2])
117: (8)                     assert_array_equal(res, desired)
118: (4)                 def test_1D_array(self):
119: (8)                     a = array([1])
120: (8)                     b = array([2])
121: (8)                     res = hstack([a, b])
122: (8)                     desired = array([1, 2])
123: (8)                     assert_array_equal(res, desired)
124: (4)                 def test_2D_array(self):
125: (8)                     a = array([[1], [2]])
126: (8)                     b = array([[1], [2]])
127: (8)                     res = hstack([a, b])
128: (8)                     desired = array([[1, 1], [2, 2]])
129: (8)                     assert_array_equal(res, desired)
130: (4)                 def test_generator(self):
131: (8)                     with pytest.raises(TypeError, match="arrays to stack must be"):
132: (12)                         hstack((np.arange(3) for _ in range(2)))
133: (8)                     with pytest.raises(TypeError, match="arrays to stack must be"):
134: (12)                         hstack(map(lambda x: x, np.ones((3, 2))))
135: (4)                 def test_casting_and_dtype(self):
136: (8)                     a = np.array([1, 2, 3])
137: (8)                     b = np.array([2.5, 3.5, 4.5])
138: (8)                     res = np.hstack((a, b), casting="unsafe", dtype=np.int64)
139: (8)                     expected_res = np.array([1, 2, 3, 2, 3, 4])
140: (8)                     assert_array_equal(res, expected_res)
141: (4)                 def test_casting_and_dtype_type_error(self):
142: (8)                     a = np.array([1, 2, 3])
143: (8)                     b = np.array([2.5, 3.5, 4.5])
144: (8)                     with pytest.raises(TypeError):
145: (12)                         hstack((a, b), casting="safe", dtype=np.int64)
146: (0)             class TestVstack:
147: (4)                 def test_non_iterable(self):
148: (8)                     assert_raises(TypeError, vstack, 1)
149: (4)                 def test_empty_input(self):
150: (8)                     assert_raises(ValueError, vstack, ())
151: (4)                 def test_0D_array(self):
152: (8)                     a = array(1)
153: (8)                     b = array(2)
154: (8)                     res = vstack([a, b])
155: (8)                     desired = array([[1], [2]])
156: (8)                     assert_array_equal(res, desired)
157: (4)                 def test_1D_array(self):
158: (8)                     a = array([1])
159: (8)                     b = array([2])
160: (8)                     res = vstack([a, b])
161: (8)                     desired = array([[1], [2]])
162: (8)                     assert_array_equal(res, desired)

```

```

163: (4)           def test_2D_array(self):
164: (8)             a = array([[1], [2]])
165: (8)             b = array([[1], [2]])
166: (8)             res = vstack([a, b])
167: (8)             desired = array([[1], [2], [1], [2]])
168: (8)             assert_array_equal(res, desired)
169: (4)           def test_2D_array2(self):
170: (8)             a = array([1, 2])
171: (8)             b = array([1, 2])
172: (8)             res = vstack([a, b])
173: (8)             desired = array([[1, 2], [1, 2]])
174: (8)             assert_array_equal(res, desired)
175: (4)           def test_generator(self):
176: (8)             with pytest.raises(TypeError, match="arrays to stack must be"):
177: (12)               vstack((np.arange(3) for _ in range(2)))
178: (4)           def test_casting_and_dtype(self):
179: (8)             a = np.array([1, 2, 3])
180: (8)             b = np.array([2.5, 3.5, 4.5])
181: (8)             res = np.vstack((a, b), casting="unsafe", dtype=np.int64)
182: (8)             expected_res = np.array([[1, 2, 3], [2, 3, 4]])
183: (8)             assert_array_equal(res, expected_res)
184: (4)           def test_casting_and_dtype_type_error(self):
185: (8)             a = np.array([1, 2, 3])
186: (8)             b = np.array([2.5, 3.5, 4.5])
187: (8)             with pytest.raises(TypeError):
188: (12)               vstack((a, b), casting="safe", dtype=np.int64)
189: (0)           class TestConcatenate:
190: (4)             def test_returns_copy(self):
191: (8)               a = np.eye(3)
192: (8)               b = np.concatenate([a])
193: (8)               b[0, 0] = 2
194: (8)               assert b[0, 0] != a[0, 0]
195: (4)             def test_exceptions(self):
196: (8)               for ndim in [1, 2, 3]:
197: (12)                 a = np.ones((1,)*ndim)
198: (12)                 np.concatenate((a, a), axis=0) # OK
199: (12)                 assert_raises(np.AxisError, np.concatenate, (a, a), axis=ndim)
200: (12)                 assert_raises(np.AxisError, np.concatenate, (a, a), axis=-(ndim +
1))
201: (8)                 assert_raises(ValueError, concatenate, (0,))
202: (8)                 assert_raises(ValueError, concatenate, (np.array(0),))
203: (8)                 assert_raises_regex(
204: (12)                   ValueError,
205: (12)                   r"all the input arrays must have same number of dimensions, but "
206: (12)                   r"the array at index 0 has 1 dimension\s\ and the array at "
207: (12)                   r"index 1 has 2 dimension\s\",
208: (12)                   np.concatenate, (np.zeros(1), np.zeros((1, 1))))
209: (8)                 a = np.ones((1, 2, 3))
210: (8)                 b = np.ones((2, 2, 3))
211: (8)                 axis = list(range(3))
212: (8)                 for i in range(3):
213: (12)                   np.concatenate((a, b), axis=axis[i]) # OK
214: (12)                   assert_raises_regex(
215: (16)                     ValueError,
216: (16)                     "all the input array dimensions except for the concatenation
axis "
217: (16)                     "must match exactly, but along dimension {}, the array at "
218: (16)                     "index 0 has size 1 and the array at index 1 has size 2"
219: (16)                     .format(i),
220: (16)                     np.concatenate, (a, b), axis=axis[i])
221: (12)                     assert_raises(ValueError, np.concatenate, (a, b), axis=axis[2])
222: (12)                     a = np.moveaxis(a, -1, 0)
223: (12)                     b = np.moveaxis(b, -1, 0)
224: (12)                     axis.append(axis.pop(0))
225: (8)                     assert_raises(ValueError, concatenate, ())
226: (4)           def test_concatenate_axis_None(self):
227: (8)             a = np.arange(4, dtype=np.float64).reshape((2, 2))
228: (8)             b = list(range(3))
229: (8)             c = ['x']

```

```

230: (8)             r = np.concatenate((a, a), axis=None)
231: (8)             assert_equal(r.dtype, a.dtype)
232: (8)             assert_equal(r.ndim, 1)
233: (8)             r = np.concatenate((a, b), axis=None)
234: (8)             assert_equal(r.size, a.size + len(b))
235: (8)             assert_equal(r.dtype, a.dtype)
236: (8)             r = np.concatenate((a, b, c), axis=None, dtype="U")
237: (8)             d = array(['0.0', '1.0', '2.0', '3.0',
238: (19)               '0', '1', '2', 'x'])
239: (8)             assert_array_equal(r, d)
240: (8)             out = np.zeros(a.size + len(b))
241: (8)             r = np.concatenate((a, b), axis=None)
242: (8)             rout = np.concatenate((a, b), axis=None, out=out)
243: (8)             assert_(out is rout)
244: (8)             assert_equal(r, rout)
245: (4)             def test_large_concatenate_axis_None(self):
246: (8)                 x = np.arange(1, 100)
247: (8)                 r = np.concatenate(x, None)
248: (8)                 assert_array_equal(x, r)
249: (8)                 r = np.concatenate(x, 100) # axis is >= MAXDIMS
250: (8)                 assert_array_equal(x, r)
251: (4)             def test_concatenate(self):
252: (8)                 r4 = list(range(4))
253: (8)                 assert_array_equal(concatenate((r4,)), r4)
254: (8)                 assert_array_equal(concatenate((tuple(r4),)), r4)
255: (8)                 assert_array_equal(concatenate((array(r4),)), r4)
256: (8)                 r3 = list(range(3))
257: (8)                 assert_array_equal(concatenate((r4, r3)), r4 + r3)
258: (8)                 assert_array_equal(concatenate((tuple(r4), r3)), r4 + r3)
259: (8)                 assert_array_equal(concatenate((array(r4), r3)), r4 + r3)
260: (8)                 assert_array_equal(concatenate((r4, r3), 0), r4 + r3)
261: (8)                 assert_array_equal(concatenate((r4, r3), -1), r4 + r3)
262: (8)                 a23 = array([[10, 11, 12], [13, 14, 15]])
263: (8)                 a13 = array([[0, 1, 2]])
264: (8)                 res = array([[10, 11, 12], [13, 14, 15], [0, 1, 2]])
265: (8)                 assert_array_equal(concatenate((a23, a13)), res)
266: (8)                 assert_array_equal(concatenate((a23, a13), 0), res)
267: (8)                 assert_array_equal(concatenate((a23.T, a13.T), 1), res.T)
268: (8)                 assert_array_equal(concatenate((a23.T, a13.T), -1), res.T)
269: (8)                 assert_raises(ValueError, concatenate, (a23.T, a13.T), 0)
270: (8)                 res = arange(2 * 3 * 7).reshape((2, 3, 7))
271: (8)                 a0 = res[..., :4]
272: (8)                 a1 = res[..., 4:6]
273: (8)                 a2 = res[..., 6:]
274: (8)                 assert_array_equal(concatenate((a0, a1, a2), 2), res)
275: (8)                 assert_array_equal(concatenate((a0, a1, a2), -1), res)
276: (8)                 assert_array_equal(concatenate((a0.T, a1.T, a2.T), 0), res.T)
277: (8)                 out = res.copy()
278: (8)                 rout = concatenate((a0, a1, a2), 2, out=out)
279: (8)                 assert_(out is rout)
280: (8)                 assert_equal(res, rout)
281: (4)             @pytest.mark.skipif(IS_PYPY, reason="PYPY handles sq_concat, nb_add
differently than cpython")
282: (4)             def test_operator_concat(self):
283: (8)                 import operator
284: (8)                 a = array([1, 2])
285: (8)                 b = array([3, 4])
286: (8)                 n = [1, 2]
287: (8)                 res = array([1, 2, 3, 4])
288: (8)                 assert_raises(TypeError, operator.concat, a, b)
289: (8)                 assert_raises(TypeError, operator.concat, a, n)
290: (8)                 assert_raises(TypeError, operator.concat, n, a)
291: (8)                 assert_raises(TypeError, operator.concat, a, 1)
292: (8)                 assert_raises(TypeError, operator.concat, 1, a)
293: (4)             def test_bad_out_shape(self):
294: (8)                 a = array([1, 2])
295: (8)                 b = array([3, 4])
296: (8)                 assert_raises(ValueError, concatenate, (a, b), out=np.empty(5))
297: (8)                 assert_raises(ValueError, concatenate, (a, b), out=np.empty((4,1)))

```

```

298: (8) assert_raises(ValueError, concatenate, (a, b), out=np.empty((1,4)))
299: (8) concatenate((a, b), out=np.empty(4))
300: (4) @pytest.mark.parametrize("axis", [None, 0])
301: (4) @pytest.mark.parametrize("out_dtype", ["c8", "f4", "f8", ">f8", "i8",
    "S4"])
302: (4) @pytest.mark.parametrize("casting",
303: (12)     ['no', 'equiv', 'safe', 'same_kind', 'unsafe'])
304: (4) def test_out_and_dtype(self, axis, out_dtype, casting):
305: (8)     out = np.empty(4, dtype=out_dtype)
306: (8)     to_concat = (array([1.1, 2.2]), array([3.3, 4.4]))
307: (8)     if not np.can_cast(to_concat[0], out_dtype, casting=casting):
308: (12)         with assert_raises(TypeError):
309: (16)             concatenate(to_concat, out=out, axis=axis, casting=casting)
310: (12)         with assert_raises(TypeError):
311: (16)             concatenate(to_concat, dtype=out.dtype,
312: (28)                 axis=axis, casting=casting)
313: (8)     else:
314: (12)         res_out = concatenate(to_concat, out=out,
315: (34)             axis=axis, casting=casting)
316: (12)         res_dtype = concatenate(to_concat, dtype=out.dtype,
317: (36)             axis=axis, casting=casting)
318: (12)         assert res_out is out
319: (12)         assert_array_equal(out, res_dtype)
320: (12)         assert res_dtype.dtype == out_dtype
321: (8)         with assert_raises(TypeError):
322: (12)             concatenate(to_concat, out=out, dtype=out_dtype, axis=axis)
323: (4) @pytest.mark.parametrize("axis", [None, 0])
324: (4) @pytest.mark.parametrize("string_dt", ["S", "U", "S0", "U0"])
325: (4) @pytest.mark.parametrize("arrs",
326: (12)     [[[0.],), ([0.], [1]), ([0], ["string"], [1.])])
327: (4) def test_dtype_with_promotion(self, arrs, string_dt, axis):
328: (8)     res = np.concatenate(arrs, axis=axis, dtype=string_dt,
casting="unsafe")
329: (8)     assert res.dtype == np.array(1.).astype(string_dt).dtype
330: (4) @pytest.mark.parametrize("axis", [None, 0])
331: (4) def test_string_dtype_does_not_inspect(self, axis):
332: (8)     with pytest.raises(TypeError):
333: (12)         np.concatenate(([None], [1]), dtype="S", axis=axis)
334: (8)     with pytest.raises(TypeError):
335: (12)         np.concatenate(([None], [1]), dtype="U", axis=axis)
336: (4) @pytest.mark.parametrize("axis", [None, 0])
337: (4) def test_subarray_error(self, axis):
338: (8)     with pytest.raises(TypeError, match=".*subarray dtype"):
339: (12)         np.concatenate(([1], [1]), dtype="(2,)i", axis=axis)
340: (0) def test_stack():
341: (4)     assert_raises(TypeError, stack, 1)
342: (4)     for input_ in [(1, 2, 3),
343: (19)         [np.int32(1), np.int32(2), np.int32(3)],
344: (19)         [np.array(1), np.array(2), np.array(3)]]:
345: (8)         assert_array_equal(stack(input_), [1, 2, 3])
346: (4)     a = np.array([1, 2, 3])
347: (4)     b = np.array([4, 5, 6])
348: (4)     r1 = array([[1, 2, 3], [4, 5, 6]])
349: (4)     assert_array_equal(np.stack((a, b)), r1)
350: (4)     assert_array_equal(np.stack((a, b), axis=1), r1.T)
351: (4)     assert_array_equal(np.stack(list([a, b])), r1)
352: (4)     assert_array_equal(np.stack(array([a, b])), r1)
353: (4)     arrays = [np.random.randn(3) for _ in range(10)]
354: (4)     axes = [0, 1, -1, -2]
355: (4)     expected_shapes = [(10, 3), (3, 10), (3, 10), (10, 3)]
356: (4)     for axis, expected_shape in zip(axes, expected_shapes):
357: (8)         assert_equal(np.stack(arrays, axis).shape, expected_shape)
358: (4)     assert_raises_regex(np.AxisError, 'out of bounds', stack, arrays, axis=2)
359: (4)     assert_raises_regex(np.AxisError, 'out of bounds', stack, arrays, axis=-3)
360: (4)     arrays = [np.random.randn(3, 4) for _ in range(10)]
361: (4)     axes = [0, 1, 2, -1, -2, -3]
362: (4)     expected_shapes = [(10, 3, 4), (3, 10, 4), (3, 4, 10),
363: (23)         (3, 4, 10), (3, 10, 4), (10, 3, 4)]
364: (4)     for axis, expected_shape in zip(axes, expected_shapes):

```

```

365: (8) assert_equal(np.stack(arrays, axis).shape, expected_shape)
366: (4) assert_(stack([], [], []).shape == (3, 0))
367: (4) assert_(stack([], [], [], axis=1).shape == (0, 3))
368: (4) out = np.zeros_like(r1)
369: (4) np.stack((a, b), out=out)
370: (4) assert_array_equal(out, r1)
371: (4) assert_raises_regex(ValueError, 'need at least one array', stack, [])
372: (4) assert_raises_regex(ValueError, 'must have the same shape',
373: (24) stack, [1, np.arange(3)])
374: (4) assert_raises_regex(ValueError, 'must have the same shape',
375: (24) stack, [np.arange(3), 1])
376: (4) assert_raises_regex(ValueError, 'must have the same shape',
377: (24) stack, [np.arange(3), 1], axis=1)
378: (4) assert_raises_regex(ValueError, 'must have the same shape',
379: (24) stack, [np.zeros((3, 3)), np.zeros(3)], axis=1)
380: (4) assert_raises_regex(ValueError, 'must have the same shape',
381: (24) stack, [np.arange(2), np.arange(3)])
382: (4) with pytest.raises(TypeError, match="arrays to stack must be"):
383: (8)     stack((x for x in range(3)))
384: (4) a = np.array([1, 2, 3])
385: (4) b = np.array([2.5, 3.5, 4.5])
386: (4) res = np.stack((a, b), axis=1, casting="unsafe", dtype=np.int64)
387: (4) expected_res = np.array([[1, 2], [2, 3], [3, 4]])
388: (4) assert_array_equal(res, expected_res)
389: (4) with assert_raises(TypeError):
390: (8)     stack((a, b), dtype=np.int64, axis=1, casting="safe")
391: (0) @pytest.mark.parametrize("axis", [0])
392: (0) @pytest.mark.parametrize("out_dtype", ["c8", "f4", "f8", ">f8", "i8"])
393: (0) @pytest.mark.parametrize("casting",
394: (25)         ['no', 'equiv', 'safe', 'same_kind', 'unsafe'])
395: (0) def test_stack_out_and_dtype(axis, out_dtype, casting):
396: (4)     to_concat = (array([1, 2]), array([3, 4]))
397: (4)     res = array([[1, 2], [3, 4]])
398: (4)     out = np.zeros_like(res)
399: (4)     if not np.can_cast(to_concat[0], out_dtype, casting=casting):
400: (8)         with assert_raises(TypeError):
401: (12)             stack(to_concat, dtype=out_dtype,
402: (18)                 axis=axis, casting=casting)
403: (4)     else:
404: (8)         res_out = stack(to_concat, out=out,
405: (24)             axis=axis, casting=casting)
406: (8)         res_dtype = stack(to_concat, dtype=out_dtype,
407: (26)             axis=axis, casting=casting)
408: (8)         assert res_out is out
409: (8)         assert_array_equal(out, res_dtype)
410: (8)         assert res_dtype.dtype == out_dtype
411: (4)         with assert_raises(TypeError):
412: (8)             stack(to_concat, out=out, dtype=out_dtype, axis=axis)
413: (0) class TestBlock:
414: (4)     @pytest.fixture(params=['block', 'force_concatenate', 'force_slicing'])
415: (4)     def block(self, request):
416: (8)         def _block_force_concatenate(arrays):
417: (12)             arrays, list_ndim, result_ndim, _ = _block_setup(arrays)
418: (12)             return _block_concatenate(arrays, list_ndim, result_ndim)
419: (8)         def _block_force_slicing(arrays):
420: (12)             arrays, list_ndim, result_ndim, _ = _block_setup(arrays)
421: (12)             return _block_slicing(arrays, list_ndim, result_ndim)
422: (8)         if request.param == 'force_concatenate':
423: (12)             return _block_force_concatenate
424: (8)         elif request.param == 'force_slicing':
425: (12)             return _block_force_slicing
426: (8)         elif request.param == 'block':
427: (12)             return block
428: (8)         else:
429: (12)             raise ValueError('Unknown blocking request. There is a typo in the
tests.')
430: (4)     def test_returns_copy(self, block):
431: (8)         a = np.eye(3)
432: (8)         b = block(a)

```

```

433: (8)             b[0, 0] = 2
434: (8)             assert b[0, 0] != a[0, 0]
435: (4)             def test_block_total_size_estimate(self, block):
436: (8)                 _, _, _, total_size = _block_setup([1])
437: (8)                 assert total_size == 1
438: (8)                 _, _, _, total_size = _block_setup([[1]])
439: (8)                 assert total_size == 1
440: (8)                 _, _, _, total_size = _block_setup([[1, 1]])
441: (8)                 assert total_size == 2
442: (8)                 _, _, _, total_size = _block_setup([[1], [1]])
443: (8)                 assert total_size == 2
444: (8)                 _, _, _, total_size = _block_setup([[1, 2], [3, 4]])
445: (8)                 assert total_size == 4
446: (4)             def test_block_simple_row_wise(self, block):
447: (8)                 a_2d = np.ones((2, 2))
448: (8)                 b_2d = 2 * a_2d
449: (8)                 desired = np.array([[1, 1, 2, 2],
450: (28)                           [1, 1, 2, 2]])
451: (8)                 result = block([a_2d, b_2d])
452: (8)                 assert_equal(desired, result)
453: (4)             def test_block_simple_column_wise(self, block):
454: (8)                 a_2d = np.ones((2, 2))
455: (8)                 b_2d = 2 * a_2d
456: (8)                 expected = np.array([[1, 1],
457: (29)                           [1, 1],
458: (29)                           [2, 2],
459: (29)                           [2, 2]])
460: (8)                 result = block([[a_2d], [b_2d]])
461: (8)                 assert_equal(expected, result)
462: (4)             def test_block_with_1d_arrays_row_wise(self, block):
463: (8)                 a = np.array([1, 2, 3])
464: (8)                 b = np.array([2, 3, 4])
465: (8)                 expected = np.array([1, 2, 3, 2, 3, 4])
466: (8)                 result = block([a, b])
467: (8)                 assert_equal(expected, result)
468: (4)             def test_block_with_1d_arrays_multiple_rows(self, block):
469: (8)                 a = np.array([1, 2, 3])
470: (8)                 b = np.array([2, 3, 4])
471: (8)                 expected = np.array([[1, 2, 3, 2, 3, 4],
472: (29)                           [1, 2, 3, 2, 3, 4]])
473: (8)                 result = block([[a, b], [a, b]])
474: (8)                 assert_equal(expected, result)
475: (4)             def test_block_with_1d_arrays_column_wise(self, block):
476: (8)                 a_1d = np.array([1, 2, 3])
477: (8)                 b_1d = np.array([2, 3, 4])
478: (8)                 expected = np.array([[1, 2, 3],
479: (29)                           [2, 3, 4]])
480: (8)                 result = block([[a_1d], [b_1d]])
481: (8)                 assert_equal(expected, result)
482: (4)             def test_block_mixed_1d_and_2d(self, block):
483: (8)                 a_2d = np.ones((2, 2))
484: (8)                 b_1d = np.array([2, 2])
485: (8)                 result = block([[a_2d], [b_1d]])
486: (8)                 expected = np.array([[1, 1],
487: (29)                           [1, 1],
488: (29)                           [2, 2]])
489: (8)                 assert_equal(expected, result)
490: (4)             def test_block_complicated(self, block):
491: (8)                 one_2d = np.array([[1, 1, 1]])
492: (8)                 two_2d = np.array([[2, 2, 2]])
493: (8)                 three_2d = np.array([[3, 3, 3, 3, 3, 3]])
494: (8)                 four_1d = np.array([4, 4, 4, 4, 4, 4])
495: (8)                 five_0d = np.array(5)
496: (8)                 six_1d = np.array([6, 6, 6, 6, 6])
497: (8)                 zero_2d = np.zeros((2, 6))
498: (8)                 expected = np.array([[1, 1, 1, 2, 2, 2],
499: (29)                           [3, 3, 3, 3, 3, 3],
500: (29)                           [4, 4, 4, 4, 4, 4],
501: (29)                           [5, 6, 6, 6, 6, 6],

```

```

502: (29) [0, 0, 0, 0, 0, 0],
503: (29) [0, 0, 0, 0, 0, 0])
504: (8) result = block([[one_2d, two_2d],
505: (24) [three_2d],
506: (24) [four_1d],
507: (24) [five_0d, six_1d],
508: (24) [zero_2d]])
509: (8) assert_equal(result, expected)
510: (4) def test_nested(self, block):
511: (8) one = np.array([1, 1, 1])
512: (8) two = np.array([[2, 2, 2], [2, 2, 2], [2, 2, 2]])
513: (8) three = np.array([3, 3, 3])
514: (8) four = np.array([4, 4, 4])
515: (8) five = np.array(5)
516: (8) six = np.array([6, 6, 6, 6])
517: (8) zero = np.zeros((2, 6))
518: (8) result = block([
519: (12) [
520: (16) block([
521: (19) [one],
522: (19) [three],
523: (19) [four]
524: (16) ]),
525: (16) two
526: (12) ],
527: (12) [five, six],
528: (12) [zero]
529: (8) ])
530: (8) expected = np.array([[1, 1, 1, 2, 2, 2],
531: (29) [3, 3, 3, 2, 2, 2],
532: (29) [4, 4, 4, 2, 2, 2],
533: (29) [5, 6, 6, 6, 6],
534: (29) [0, 0, 0, 0, 0, 0],
535: (29) [0, 0, 0, 0, 0, 0]])
536: (8) assert_equal(result, expected)
537: (4) def test_3d(self, block):
538: (8) a000 = np.ones((2, 2, 2), int) * 1
539: (8) a100 = np.ones((3, 2, 2), int) * 2
540: (8) a010 = np.ones((2, 3, 2), int) * 3
541: (8) a001 = np.ones((2, 2, 3), int) * 4
542: (8) a011 = np.ones((2, 3, 3), int) * 5
543: (8) a101 = np.ones((3, 2, 3), int) * 6
544: (8) a110 = np.ones((3, 3, 2), int) * 7
545: (8) a111 = np.ones((3, 3, 3), int) * 8
546: (8) result = block([
547: (12) [
548: (16) [a000, a001],
549: (16) [a010, a011],
550: (12) ],
551: (12) [
552: (16) [a100, a101],
553: (16) [a110, a111],
554: (12) ]
555: (8) ])
556: (8) expected = array([[[1, 1, 4, 4, 4],
557: (27) [1, 1, 4, 4, 4],
558: (27) [3, 3, 5, 5, 5],
559: (27) [3, 3, 5, 5, 5],
560: (27) [3, 3, 5, 5, 5]],
561: (26) [[[1, 1, 4, 4, 4],
562: (27) [1, 1, 4, 4, 4],
563: (27) [3, 3, 5, 5, 5],
564: (27) [3, 3, 5, 5, 5],
565: (27) [3, 3, 5, 5, 5]],
566: (26) [[[2, 2, 6, 6, 6],
567: (27) [2, 2, 6, 6, 6],
568: (27) [7, 7, 8, 8, 8],
569: (27) [7, 7, 8, 8, 8],
570: (27) [7, 7, 8, 8, 8]],
```

```

571: (26) [[2, 2, 6, 6, 6],
572: (27) [2, 2, 6, 6, 6],
573: (27) [7, 7, 8, 8, 8],
574: (27) [7, 7, 8, 8, 8],
575: (27) [7, 7, 8, 8, 8]],
576: (26) [[2, 2, 6, 6, 6],
577: (27) [2, 2, 6, 6, 6],
578: (27) [7, 7, 8, 8, 8],
579: (27) [7, 7, 8, 8, 8],
580: (27) [7, 7, 8, 8, 8]])
581: (8) assert_array_equal(result, expected)
582: (4) def test_block_with_mismatched_shape(self, block):
583: (8)     a = np.array([0, 0])
584: (8)     b = np.eye(2)
585: (8)     assert_raises(ValueError, block, [a, b])
586: (8)     assert_raises(ValueError, block, [b, a])
587: (8)     to_block = [[np.ones((2,3)), np.ones((2,2))],
588: (20)                 [np.ones((2,2)), np.ones((2,2))]]
589: (8)     assert_raises(ValueError, block, to_block)
590: (4) def test_no_lists(self, block):
591: (8)     assert_equal(block(1), np.array(1))
592: (8)     assert_equal(block(np.eye(3)), np.eye(3))
593: (4) def test_invalid_nesting(self, block):
594: (8)     msg = 'depths are mismatched'
595: (8)     assert_raises_regex(ValueError, msg, block, [1, [2]])
596: (8)     assert_raises_regex(ValueError, msg, block, [1, []])
597: (8)     assert_raises_regex(ValueError, msg, block, [[1], 2])
598: (8)     assert_raises_regex(ValueError, msg, block, [[], 2])
599: (8)     assert_raises_regex(ValueError, msg, block, [
600: (12)         [[1], [2]],
601: (12)         [[3, 4]],
602: (12)         [5] # missing brackets
603: (8)     ])
604: (4) def test_empty_lists(self, block):
605: (8)     assert_raises_regex(ValueError, 'empty', block, [])
606: (8)     assert_raises_regex(ValueError, 'empty', block, [[]])
607: (8)     assert_raises_regex(ValueError, 'empty', block, [[1], []])
608: (4) def test_tuple(self, block):
609: (8)     assert_raises_regex(TypeError, 'tuple', block, ([1, 2], [3, 4]))
610: (8)     assert_raises_regex(TypeError, 'tuple', block, [(1, 2), (3, 4)])
611: (4) def test_different_ndims(self, block):
612: (8)     a = 1.
613: (8)     b = 2 * np.ones((1, 2))
614: (8)     c = 3 * np.ones((1, 1, 3))
615: (8)     result = block([a, b, c])
616: (8)     expected = np.array([[1., 2., 2., 3., 3., 3.]])
617: (8)     assert_equal(result, expected)
618: (4) def test_different_ndims_depths(self, block):
619: (8)     a = 1.
620: (8)     b = 2 * np.ones((1, 2))
621: (8)     c = 3 * np.ones((1, 2, 3))
622: (8)     result = block([[a, b], [c]])
623: (8)     expected = np.array([[1., 2., 2.],
624: (30)                     [3., 3., 3.],
625: (30)                     [3., 3., 3.]]])
626: (8)     assert_equal(result, expected)
627: (4) def test_block_memory_order(self, block):
628: (8)     arr_c = np.zeros((3)*3, order='C')
629: (8)     arr_f = np.zeros((3)*3, order='F')
630: (8)     b_c = [[[arr_c, arr_c],
631: (16)                 [arr_c, arr_c]],
632: (15)                 [[arr_c, arr_c],
633: (16)                               [arr_c, arr_c]]]
634: (8)     b_f = [[[arr_f, arr_f],
635: (16)                 [arr_f, arr_f]],
636: (15)                 [[arr_f, arr_f],
637: (16)                               [arr_f, arr_f]]]
638: (8)     assert block(b_c).flags['C_CONTIGUOUS']
639: (8)     assert block(b_f).flags['F_CONTIGUOUS']

```

```

640: (8)         arr_c = np.zeros((3, 3), order='C')
641: (8)         arr_f = np.zeros((3, 3), order='F')
642: (8)         b_c = [[arr_c, arr_c],
643: (15)           [arr_c, arr_c]]
644: (8)         b_f = [[arr_f, arr_f],
645: (15)           [arr_f, arr_f]]
646: (8)         assert block(b_c).flags['C_CONTIGUOUS']
647: (8)         assert block(b_f).flags['F_CONTIGUOUS']
648: (0)         def test_block_dispatcher():
649: (4)             class ArrayLike:
650: (8)                 pass
651: (4)                 a = ArrayLike()
652: (4)                 b = ArrayLike()
653: (4)                 c = ArrayLike()
654: (4)                 assert_equal(list(_block_dispatcher(a)), [a])
655: (4)                 assert_equal(list(_block_dispatcher([a])), [a])
656: (4)                 assert_equal(list(_block_dispatcher([a, b])), [a, b])
657: (4)                 assert_equal(list(_block_dispatcher([[a], [b, [c]]])), [a, b, c])
658: (4)                 assert_equal(list(_block_dispatcher((a, b))), [(a, b)])

```

---

File 126 - test\_simd.py:

```

1: (0)         import pytest, math, re
2: (0)         import itertools
3: (0)         import operator
4: (0)         from numpy.core._simd import targets, clear_floatstatus, get_floatstatus
5: (0)         from numpy.core._multiarray_umath import __cpu_baseline__
6: (0)         def check_floatstatus(divbyzero=False, overflow=False,
7: (22)               underflow=False, invalid=False,
8: (22)               all=False):
9: (4)             err = get_floatstatus()
10: (4)            ret = (all or divbyzero) and (err & 1) != 0
11: (4)            ret |= (all or overflow) and (err & 2) != 0
12: (4)            ret |= (all or underflow) and (err & 4) != 0
13: (4)            ret |= (all or invalid) and (err & 8) != 0
14: (4)            return ret
15: (0)         class _Test_Utility:
16: (4)             npyv = None
17: (4)             sfx = None
18: (4)             target_name = None
19: (4)             def __getattr__(self, attr):
20: (8)                 """
21: (8)                     To call NPV intrinsics without the attribute 'npyv' and
22: (8)                     auto suffixing intrinsics according to class attribute 'sfx'
23: (8)                     """
24: (8)                 return getattr(self.npyv, attr + "_" + self.sfx)
25: (4)             def _x2(self, intrin_name):
26: (8)                 return getattr(self.npyv, f"{intrin_name}_{self.sfx}x2")
27: (4)             def _data(self, start=None, count=None, reverse=False):
28: (8)                 """
29: (8)                     Create list of consecutive numbers according to number of vector's
30: (8)                     lanes.
31: (8)                     """
32: (12)                     if start is None:
33: (8)                         start = 1
34: (12)                     if count is None:
35: (8)                         count = self.nlanes
36: (8)                     rng = range(start, start + count)
37: (12)                     if reverse:
38: (8)                         rng = reversed(rng)
39: (12)                     if self._is_fp():
40: (8)                         return [x / 1.0 for x in rng]
41: (4)                     return list(rng)
42: (8)             def _is_unsigned(self):
43: (8)                 return self.sfx[0] == 'u'
44: (4)             def _is_signed(self):
44: (8)                 return self.sfx[0] == 's'

```

```

45: (4)           def _is_fp(self):
46: (8)             return self.sfx[0] == 'f'
47: (4)           def _scalar_size(self):
48: (8)             return int(self.sfx[1:])
49: (4)           def _int_clip(self, seq):
50: (8)             if self._is_fp():
51: (12)               return seq
52: (8)             max_int = self._int_max()
53: (8)             min_int = self._int_min()
54: (8)             return [min(max(v, min_int), max_int) for v in seq]
55: (4)           def _int_max(self):
56: (8)             if self._is_fp():
57: (12)               return None
58: (8)             max_u = self._to_unsigned(self.setall(-1))[0]
59: (8)             if self._is_signed():
60: (12)               return max_u // 2
61: (8)             return max_u
62: (4)           def _int_min(self):
63: (8)             if self._is_fp():
64: (12)               return None
65: (8)             if self._is_unsigned():
66: (12)               return 0
67: (8)             return -(self._int_max() + 1)
68: (4)           def _true_mask(self):
69: (8)             max_unsig = getattr(self.npyv, "setall_u" + self.sfx[1:])(-1)
70: (8)             return max_unsig[0]
71: (4)           def _to_unsigned(self, vector):
72: (8)             if isinstance(vector, (list, tuple)):
73: (12)               return getattr(self.npyv, "load_u" + self.sfx[1:])(vector)
74: (8)             else:
75: (12)               sfx = vector.__name__.replace("npvv_", "")
76: (12)               if sfx[0] == "b":
77: (16)                 cvt_intrinsic = "cvt_u{0}_b{0}"
78: (12)               else:
79: (16)                 cvt_intrinsic = "reinterpret_u{0}_{1}"
80: (12)               return getattr(self.npyv, cvt_intrinsic.format(sfx[1:], sfx))(vector)
81: (4)           def _pinfinity(self):
82: (8)             return float("inf")
83: (4)           def _neginfinity(self):
84: (8)             return -float("inf")
85: (4)           def _nan(self):
86: (8)             return float("nan")
87: (4)           def _cpu_features(self):
88: (8)             target = self.target_name
89: (8)             if target == "baseline":
90: (12)               target = __cpu_baseline__
91: (8)             else:
92: (12)               target = target.split('__') # multi-target separator
93: (8)             return ' '.join(target)
94: (0)           class SIMD_BOOL(_Test_Utility):
95: (4)             """
96: (4)               To test all boolean vector types at once
97: (4)             """
98: (4)           def _nlanes(self):
99: (8)             return getattr(self.npyv, "nlanes_u" + self.sfx[1:])
100: (4)           def _data(self, start=None, count=None, reverse=False):
101: (8)             true_mask = self._true_mask()
102: (8)             rng = range(self._nlanes())
103: (8)             if reverse:
104: (12)               rng = reversed(rng)
105: (8)             return [true_mask if x % 2 else 0 for x in rng]
106: (4)           def _load_b(self, data):
107: (8)             len_str = self.sfx[1:]
108: (8)             load = getattr(self.npyv, "load_u" + len_str)
109: (8)             cvt = getattr(self.npyv, f"cvt_b{len_str}_u{len_str}")
110: (8)             return cvt(load(data))
111: (4)           def test_operators_logical(self):
112: (8)             """
113: (8)               Logical operations for boolean types.

```

```

114: (8)           Test intrinsics:
115: (12)          npyv_xor_##SFX, npyv_and_##SFX, npyv_or_##SFX, npyv_not_##SFX,
116: (12)          npyv_andc_b8, npyv_orc_b8, npyv_xnor_b8
117: (8)
118: (8)          """
119: (8)          data_a = self._data()
120: (8)          data_b = self._data(reverse=True)
121: (8)          vdata_a = self._load_b(data_a)
122: (8)          vdata_b = self._load_b(data_b)
123: (8)          data_and = [a & b for a, b in zip(data_a, data_b)]
124: (8)          vand = getattr(self, "and")(vdata_a, vdata_b)
125: (8)          assert vand == data_and
126: (8)          data_or = [a | b for a, b in zip(data_a, data_b)]
127: (8)          vor = getattr(self, "or")(vdata_a, vdata_b)
128: (8)          assert vor == data_or
129: (8)          data_xor = [a ^ b for a, b in zip(data_a, data_b)]
130: (8)          vxor = getattr(self, "xor")(vdata_a, vdata_b)
131: (8)          assert vxor == data_xor
132: (8)          vnot = getattr(self, "not")(vdata_a)
133: (8)          assert vnot == data_b
134: (12)         if self.sfx not in ("b8"):
135: (8)             return
136: (8)         data_andc = [(a & ~b) & 0xFF for a, b in zip(data_a, data_b)]
137: (8)         vandc = getattr(self, "andc")(vdata_a, vdata_b)
138: (8)         assert data_andc == vandc
139: (8)         data_orc = [(a | ~b) & 0xFF for a, b in zip(data_a, data_b)]
140: (8)         vorc = getattr(self, "orc")(vdata_a, vdata_b)
141: (8)         assert data_orc == vorc
142: (8)         data_xnor = [~(a ^ b) & 0xFF for a, b in zip(data_a, data_b)]
143: (8)         vxnor = getattr(self, "xnor")(vdata_a, vdata_b)
144: (4)          assert data_xnor == vxnor
145: (8)          def test_tobits(self):
146: (8)              data2bits = lambda data: sum([int(x != 0) << i for i, x in
147: (12)                  for data in (self._data(), self._data(reverse=True))]):
148: (12)                  vdata = self._load_b(data)
149: (12)                  data_bits = data2bits(data)
150: (12)                  tobits = self.tobits(vdata)
151: (12)                  bin_tobits = bin(tobits)
152: (4)                    assert bin_tobits == bin(data_bits)
153: (8)
154: (8)          def test_pack(self):
155: (8)              """
156: (12)                  Pack multiple vectors into one
157: (12)                  Test intrinsics:
158: (12)                      npyv_pack_b8_b16
159: (8)                      npyv_pack_b8_b32
160: (8)                      npyv_pack_b8_b64
161: (12)                      """
162: (8)
163: (8)          if self.sfx not in ("b16", "b32", "b64"):
164: (12)              return
165: (8)          data = self._data()
166: (8)          rdata = self._data(reverse=True)
167: (8)          vdata = self._load_b(data)
168: (12)          vrdata = self._load_b(rdata)
169: (12)          pack_simd = getattr(self.npyv, f"pack_b8_{self.sfx}")
170: (8)
171: (12)          if self.sfx == "b16":
172: (12)              spack = [(i & 0xFF) for i in (list(rdata) + list(data))]
173: (8)              vpack = pack_simd(vrdata, vdata)
174: (12)          elif self.sfx == "b32":
175: (12)              spack = [(i & 0xFF) for i in (2*list(rdata) + 2*list(data))]
176: (31)              vpack = pack_simd(vrdata, vrdata, vdata, vdata,
177: (8)                                vdata, vdata, vdata, vdata)
178: (4)                assert vpack == spack
179: (4)                @pytest.mark.parametrize("intrin", ["any", "all"])
180: (8)                @pytest.mark.parametrize("data", (
181: (8)                  [-1, 0],

```

```

182: (8)           [-1],
183: (8)           [0]
184: (4)
185: (4)           def test_operators_crosstest(self, intrin, data):
186: (8)           """
187: (8)             Test intrinsics:
188: (12)             npyv_any_##SFX
189: (12)             npyv_all_##SFX
190: (8)
191: (8)             """
192: (8)             data_a = self._load_b(data * self._nlanes())
193: (8)             func = eval(intrin)
194: (8)             intrin = getattr(self, intrin)
195: (8)             desired = func(data_a)
196: (8)             simd = intrin(data_a)
197: (8)             assert not not simd == desired
198: (0)           class _SIMD_INT(_TestUtility):
199: (4)           """
200: (4)             To test all integer vector types at once
201: (4)           """
202: (4)           def test_operators_shift(self):
203: (8)           if self.sfx in ("u8", "s8"):
204: (12)             return
205: (8)           data_a = self._data(self._int_max() - self._nlanes)
206: (8)           data_b = self._data(self._int_min(), reverse=True)
207: (8)           vdata_a, vdata_b = self.load(data_a), self.load(data_b)
208: (12)           for count in range(self._scalar_size()):
209: (12)             data_shl_a = self.load([a << count for a in data_a])
210: (12)             shl = self.shl(vdata_a, count)
211: (12)             assert shl == data_shl_a
212: (12)             data_shr_a = self.load([a >> count for a in data_a])
213: (12)             shr = self.shr(vdata_a, count)
214: (12)             assert shr == data_shr_a
215: (12)           for count in range(1, self._scalar_size()):
216: (12)             data_shl_a = self.load([a << count for a in data_a])
217: (12)             shli = self.shli(vdata_a, count)
218: (12)             assert shli == data_shl_a
219: (12)             data_shr_a = self.load([a >> count for a in data_a])
220: (12)             shri = self.shri(vdata_a, count)
221: (12)             assert shri == data_shr_a
222: (4)           def test_arithmetic_subadd_saturated(self):
223: (8)           if self.sfx in ("u32", "s32", "u64", "s64"):
224: (12)             return
225: (8)           data_a = self._data(self._int_max() - self._nlanes)
226: (8)           data_b = self._data(self._int_min(), reverse=True)
227: (8)           vdata_a, vdata_b = self.load(data_a), self.load(data_b)
228: (8)           data_adds = self._int_clip([a + b for a, b in zip(data_a, data_b)])
229: (8)           adds = self.adds(vdata_a, vdata_b)
230: (8)           assert adds == data_adds
231: (8)           data_subs = self._int_clip([a - b for a, b in zip(data_a, data_b)])
232: (8)           subs = self.subs(vdata_a, vdata_b)
233: (8)           assert subs == data_subs
234: (4)           def test_math_max_min(self):
235: (8)           data_a = self._data()
236: (8)           data_b = self._data(self._nlanes)
237: (8)           vdata_a, vdata_b = self.load(data_a), self.load(data_b)
238: (8)           data_max = [max(a, b) for a, b in zip(data_a, data_b)]
239: (8)           simd_max = self.max(vdata_a, vdata_b)
240: (8)           assert simd_max == data_max
241: (8)           data_min = [min(a, b) for a, b in zip(data_a, data_b)]
242: (8)           simd_min = self.min(vdata_a, vdata_b)
243: (8)           assert simd_min == data_min
244: (4)           @pytest.mark.parametrize("start", [-100, -10000, 0, 100, 10000])
245: (4)           def test_reduce_max_min(self, start):
246: (8)           """
247: (12)             Test intrinsics:
248: (12)             npyv_reduce_max_##sfx
249: (12)             npyv_reduce_min_##sfx
250: (8)           vdata_a = self.load(self._data(start))

```

```

251: (8)             assert self.reduce_max(vdata_a) == max(vdata_a)
252: (8)             assert self.reduce_min(vdata_a) == min(vdata_a)
253: (0) class _SIMD_FP32(_Test_Utility):
254: (4)             """
255: (4)             To only test single precision
256: (4)             """
257: (4)             def test_conversions(self):
258: (8)                 """
259: (8)                 Round to nearest even integer, assume CPU control register is set to
rounding.
260: (8)             Test intrinsics:
261: (12)             npyv_round_s32_##SFX
262: (8)             """
263: (8)             features = self._cpu_features()
264: (8)             if not self.npyv.simd_f64 and re.match(r".*(NEON|ASIMD)", features):
265: (12)                 _round = lambda v: int(v + (0.5 if v >= 0 else -0.5))
266: (8)             else:
267: (12)                 _round = round
268: (8)             vdata_a = self.load(self._data())
269: (8)             vdata_a = self.sub(vdata_a, self.setall(0.5))
270: (8)             data_round = [_round(x) for x in vdata_a]
271: (8)             vround = self.round_s32(vdata_a)
272: (8)             assert vround == data_round
273: (0) class _SIMD_FP64(_Test_Utility):
274: (4)             """
275: (4)             To only test double precision
276: (4)             """
277: (4)             def test_conversions(self):
278: (8)                 """
279: (8)                 Round to nearest even integer, assume CPU control register is set to
rounding.
280: (8)             Test intrinsics:
281: (12)             npyv_round_s32_##SFX
282: (8)             """
283: (8)             vdata_a = self.load(self._data())
284: (8)             vdata_a = self.sub(vdata_a, self.setall(0.5))
285: (8)             vdata_b = self.mul(vdata_a, self.setall(-1.5))
286: (8)             data_round = [round(x) for x in list(vdata_a) + list(vdata_b)]
287: (8)             vround = self.round_s32(vdata_a, vdata_b)
288: (8)             assert vround == data_round
289: (0) class _SIMD_FP(_Test_Utility):
290: (4)             """
291: (4)             To test all float vector types at once
292: (4)             """
293: (4)             def test_arithmetic_fused(self):
294: (8)                 vdata_a, vdata_b, vdata_c = [self.load(self._data())]*3
295: (8)                 vdata(cx2 = self.add(vdata_c, vdata_c)
296: (8)                 data_fma = self.load([a * b + c for a, b, c in zip(vdata_a, vdata_b,
vdata_c)])
297: (8)                 fma = self.muladd(vdata_a, vdata_b, vdata_c)
298: (8)                 assert fma == data_fma
299: (8)                 fms = self.mulsub(vdata_a, vdata_b, vdata_c)
300: (8)                 data_fms = self.sub(data_fma, vdata(cx2)
301: (8)                 assert fms == data_fms
302: (8)                 nfma = self.nmuladd(vdata_a, vdata_b, vdata_c)
303: (8)                 data_nfma = self.sub(vdata(cx2, data_fma)
304: (8)                 assert nfma == data_nfma
305: (8)                 nfms = self.nmulsub(vdata_a, vdata_b, vdata_c)
306: (8)                 data_nfms = self.mul(data_fma, self.setall(-1))
307: (8)                 assert nfms == data_nfms
308: (8)                 fmas = list(self.muladdsub(vdata_a, vdata_b, vdata_c))
309: (8)                 assert fmas[0::2] == list(data_fms)[0::2]
310: (8)                 assert fmas[1::2] == list(data_fma)[1::2]
311: (4)             def test_abs(self):
312: (8)                 pinf, ninf, nan = self._pinfinity(), self._ninfinity(), self._nan()
313: (8)                 data = self._data()
314: (8)                 vdata = self.load(self._data())
315: (8)                 abs_cases = ((-0, 0), (ninf, pinf), (pinf, pinf), (nan, nan))
316: (8)                 for case, desired in abs_cases:

```

```

317: (12)
318: (12)
319: (12)
320: (8)
321: (8)
322: (4)
323: (8)
324: (8)
325: (8)
326: (8)
327: (8)
328: (12)
329: (12)
330: (12)
331: (8)
precision
332: (8)
333: (8)
334: (4)
335: (8)
336: (8)
337: (8)
338: (8)
339: (8)
340: (12)
341: (12)
342: (12)
343: (8)
344: (8)
345: (8)
346: (4)
347: (4)
348: (4)
349: (8)
350: (8)
351: (12)
352: (12)
353: (12)
354: (12)
355: (8)
356: (8)
357: (8)
358: (8)
359: (8)
360: (8)
361: (12)
362: (12)
363: (12)
364: (8)
365: (12)
366: (16)
367: (16)
368: (16)
369: (16)
370: (8)
371: (12)
372: (12)
373: (8)
374: (12)
375: (12)
376: (12)
377: (12)
378: (8)
379: (12)
380: (8)
381: (12)
382: (8)
383: (12)

            data_abs = [desired]*self.nlanes
            vabs = self.abs(self.setall(case))
            assert vabs == pytest.approx(data_abs, nan_ok=True)
            vabs = self.abs(self.mul(vdata, self.setall(-1)))
            assert vabs == data
        def test_sqrt(self):
            pinf, ninf, nan = self._pinfinity(), self._ninfinity(), self._nan()
            data = self._data()
            vdata = self.load(self._data())
            sqrt_cases = ((-0.0, -0.0), (0.0, 0.0), (-1.0, nan), (ninf, nan),
                           (pinf, pinf))
            for case, desired in sqrt_cases:
                data_sqrt = [desired]*self.nlanes
                sqrt = self.sqrt(self.setall(case))
                assert sqrt == pytest.approx(data_sqrt, nan_ok=True)
            data_sqrt = self.load([math.sqrt(x) for x in data]) # load to truncate
            sqrt = self.sqrt(vdata)
            assert sqrt == data_sqrt
        def test_square(self):
            pinf, ninf, nan = self._pinfinity(), self._ninfinity(), self._nan()
            data = self._data()
            vdata = self.load(self._data())
            square_cases = ((nan, nan), (pinf, pinf), (ninf, pinf))
            for case, desired in square_cases:
                data_square = [desired]*self.nlanes
                square = self.square(self.setall(case))
                assert square == pytest.approx(data_square, nan_ok=True)
            data_square = [x*x for x in data]
            square = self.square(vdata)
            assert square == data_square
    @pytest.mark.parametrize("intrin, func", [("ceil", math.ceil),
  ("trunc", math.trunc), ("floor", math.floor), ("rint", round)])
    def test_rounding(self, intrin, func):
        """
        Test intrinsics:
            npyv_rint##SFX
            npyv_ceil##SFX
            npyv_trunc##SFX
            npyv_floor##SFX
        """
        intrin_name = intrin
        intrin = getattr(self, intrin)
        pinf, ninf, nan = self._pinfinity(), self._ninfinity(), self._nan()
        round_cases = ((nan, nan), (pinf, pinf), (ninf, ninf))
        for case, desired in round_cases:
            data_round = [desired]*self.nlanes
            _round = intrin(self.setall(case))
            assert _round == pytest.approx(data_round, nan_ok=True)
        for x in range(0, 2**20, 256**2):
            for w in (-1.05, -1.10, -1.15, 1.05, 1.10, 1.15):
                data = self.load([(x+a)*w for a in range(self.nlanes)])
                data_round = [func(x) for x in data]
                _round = intrin(data)
                assert _round == data_round
        for i in (
            1.1529215045988576e+18, 4.6116860183954304e+18,
            5.902958103546122e+20, 2.3611832414184488e+21
        ):
            x = self.setall(i)
            y = intrin(x)
            data_round = [func(n) for n in x]
            assert y == data_round
        if intrin_name == "floor":
            data_szero = (-0.0,)
        else:
            data_szero = (-0.0, -0.25, -0.30, -0.45, -0.5)
        for w in data_szero:
            _round = self._to_unsigned(intrin(self.setall(w)))

```

```

384: (12)             data_round = self._to_unsigned(self.setall(-0.0))
385: (12)             assert _round == data_round
386: (4)              @pytest.mark.parametrize("intrin", [
387: (8)                "max", "maxp", "maxn", "min", "minp", "minn"
388: (4)            ])
389: (4)            def test_max_min(self, intrin):
390: (8)            """
391: (8)            Test intrinsics:
392: (12)            npyv_max_##sfx
393: (12)            npyv_maxp_##sfx
394: (12)            npyv_maxn_##sfx
395: (12)            npyv_min_##sfx
396: (12)            npyv_minp_##sfx
397: (12)            npyv_minn_##sfx
398: (12)            npyv_reduce_max_##sfx
399: (12)            npyv_reduce_maxp_##sfx
400: (12)            npyv_reduce_maxn_##sfx
401: (12)            npyv_reduce_min_##sfx
402: (12)            npyv_reduce_minp_##sfx
403: (12)            npyv_reduce_minn_##sfx
404: (8)            """
405: (8)            pinf, ninf, nan = self._pinfinity(), self._ninfinity(), self._nan()
406: (8)            chk_nan = {"xp": 1, "np": 1, "nn": 2, "xn": 2}.get(intrin[-2:], 0)
407: (8)            func = eval(intrin[:3])
408: (8)            reduce_intrinsic = getattr(self, "reduce_" + intrin)
409: (8)            intrinsic = getattr(self, intrinsic)
410: (8)            hf_nlanes = self.nlanes//2
411: (8)            cases = (
412: (12)                ([0.0, -0.0], [-0.0, 0.0]),
413: (12)                ([10, -10], [10, -10]),
414: (12)                ([pinf, 10], [10, ninf]),
415: (12)                ([10, pinf], [ninf, 10]),
416: (12)                ([10, -10], [10, -10]),
417: (12)                ([-10, 10], [-10, 10]))
418: (8)        )
419: (8)        for op1, op2 in cases:
420: (12)            vdata_a = self.load(op1*hf_nlanes)
421: (12)            vdata_b = self.load(op2*hf_nlanes)
422: (12)            data = func(vdata_a, vdata_b)
423: (12)            simd = intrinsic(vdata_a, vdata_b)
424: (12)            assert simd == data
425: (12)            data = func(vdata_a)
426: (12)            simd = reduce_intrinsic(vdata_a)
427: (12)            assert simd == data
428: (8)        if not chk_nan:
429: (12)            return
430: (8)        if chk_nan == 1:
431: (12)            test_nan = lambda a, b: (
432: (16)                b if math.isnan(a) else a if math.isnan(b) else b
433: (12)            )
434: (8)        else:
435: (12)            test_nan = lambda a, b: (
436: (16)                nan if math.isnan(a) or math.isnan(b) else b
437: (12)            )
438: (8)        cases = (
439: (12)            (nan, 10),
440: (12)            (10, nan),
441: (12)            (nan, pinf),
442: (12)            (pinf, nan),
443: (12)            (nan, nan)
444: (8)        )
445: (8)        for op1, op2 in cases:
446: (12)            vdata_ab = self.load([op1, op2]*hf_nlanes)
447: (12)            data = test_nan(op1, op2)
448: (12)            simd = reduce_intrinsic(vdata_ab)
449: (12)            assert simd == pytest.approx(data, nan_ok=True)
450: (12)            vdata_a = self.setall(op1)
451: (12)            vdata_b = self.setall(op2)
452: (12)            data = [data] * self.nlanes

```

```

453: (12)           simd = intrin(vdata_a, vdata_b)
454: (12)           assert simd == pytest.approx(data, nan_ok=True)
455: (4)            def test_reciprocal(self):
456: (8)             pinf, ninf, nan = self._pinfinity(), self._ninfinity(), self._nan()
457: (8)             data = self._data()
458: (8)             vdata = self.load(self._data())
459: (8)             recip_cases = ((nan, nan), (pinf, 0.0), (ninf, -0.0), (0.0, pinf),
(-0.0, ninf))
460: (8)
461: (12)           for case, desired in recip_cases:
462: (12)             data_recip = [desired]*self.nlanes
463: (12)             recip = self.recip(self.setall(case))
464: (8)             assert recip == pytest.approx(data_recip, nan_ok=True)
precision
465: (8)             data_recip = self.load([1/x for x in data]) # load to truncate
466: (8)             recip = self.recip(vdata)
467: (4)             assert recip == data_recip
468: (8)            def test_special_cases(self):
469: (8)              """
470: (12)                Compare Not NaN. Test intrinsics:
471: (8)                  npv_notnan_##SFX
472: (8)              """
473: (8)              nnan = self.notnan(self.setall(self._nan()))
474: (4)              assert nnan == [0]*self.nlanes
@ pytest.mark.parametrize("intrin_name", [
475: (8)                "rint", "trunc", "ceil", "floor"
])
476: (4)            def test_unary_invalid_fpexception(self, intrin_name):
477: (4)              intrin = getattr(self, intrin_name)
478: (8)              for d in [float("nan"), float("inf"), -float("inf")]:
479: (8)                v = self.setall(d)
480: (12)                clear_floatstatus()
481: (12)                intrin(v)
482: (12)                assert check_floatstatus(invalid=True) == False
483: (12)            @ pytest.mark.parametrize('py_comp,np_comp', [
484: (4)              (operator.lt, "cmplt"),
485: (8)              (operator.le, "cmple"),
486: (8)              (operator.gt, "cmpgt"),
487: (8)              (operator.ge, "cmpge"),
488: (8)              (operator.eq, "cmpeq"),
489: (8)              (operator.ne, "cmpneq")
])
490: (8)
491: (4)            def test_comparison_with_nan(self, py_comp, np_comp):
492: (4)              pinf, ninf, nan = self._pinfinity(), self._ninfinity(), self._nan()
493: (8)              mask_true = self._true_mask()
494: (8)              def to_bool(vector):
495: (8)                return [lane == mask_true for lane in vector]
496: (12)              intrin = getattr(self, np_comp)
497: (8)              cmp_cases = ((0, nan), (nan, 0), (nan, nan), (pinf, nan),
498: (8)                            (ninf, nan), (-0.0, +0.0))
499: (21)
500: (8)              for case_operand1, case_operand2 in cmp_cases:
501: (12)                data_a = [case_operand1]*self.nlanes
502: (12)                data_b = [case_operand2]*self.nlanes
503: (12)                vdata_a = self.setall(case_operand1)
504: (12)                vdata_b = self.setall(case_operand2)
505: (12)                vcmp = to_bool(intrin(vdata_a, vdata_b))
506: (12)                data_cmp = [py_comp(a, b) for a, b in zip(data_a, data_b)]
507: (12)                assert vcmp == data_cmp
508: (4)            @ pytest.mark.parametrize("intrin", ["any", "all"])
509: (4)            @ pytest.mark.parametrize("data", (
510: (8)              [float("nan"), 0],
511: (8)              [0, float("nan")],
512: (8)              [float("nan"), 1],
513: (8)              [1, float("nan")],
514: (8)              [float("nan"), float("nan")],
515: (8)              [0.0, -0.0],
516: (8)              [-0.0, 0.0],
517: (8)              [1.0, -0.0]
))
518: (4)            def test_operators_crosstest(self, intrin, data):

```

```

520: (8)           """
521: (8)             Test intrinsics:
522: (12)             npyv_any_##SFX
523: (12)             npyv_all_##SFX
524: (8)           """
525: (8)             data_a = self.load(data * self.nlanes)
526: (8)             func = eval(intrin)
527: (8)             intrin = getattr(self, intrin)
528: (8)             desired = func(data_a)
529: (8)             simd = intrin(data_a)
530: (8)             assert not not simd == desired
531: (0)         class _SIMD_ALL(_Test_Utility):
532: (4)           """
533: (4)             To test all vector types at once
534: (4)           """
535: (4)             def test_memory_load(self):
536: (8)               data = self._data()
537: (8)               load_data = self.load(data)
538: (8)               assert load_data == data
539: (8)               loada_data = self.loada(data)
540: (8)               assert loada_data == data
541: (8)               loads_data = self.loads(data)
542: (8)               assert loads_data == data
543: (8)               loadl = self.loadl(data)
544: (8)               loadl_half = list(loadl)[:self.nlanes//2]
545: (8)               data_half = data[:self.nlanes//2]
546: (8)               assert loadl_half == data_half
547: (8)               assert loadl != data # detect overflow
548: (4)             def test_memory_store(self):
549: (8)               data = self._data()
550: (8)               vdata = self.load(data)
551: (8)               store = [0] * self.nlanes
552: (8)               self.store(store, vdata)
553: (8)               assert store == data
554: (8)               store_a = [0] * self.nlanes
555: (8)               self.storea(store_a, vdata)
556: (8)               assert store_a == data
557: (8)               store_s = [0] * self.nlanes
558: (8)               self.stores(store_s, vdata)
559: (8)               assert store_s == data
560: (8)               store_l = [0] * self.nlanes
561: (8)               self.storel(store_l, vdata)
562: (8)               assert store_l[:self.nlanes//2] == data[:self.nlanes//2]
563: (8)               assert store_l != vdata # detect overflow
564: (8)               store_h = [0] * self.nlanes
565: (8)               self.storeh(store_h, vdata)
566: (8)               assert store_h[:self.nlanes//2] == data[self.nlanes//2:]
567: (8)               assert store_h != vdata # detect overflow
568: (4)             @pytest.mark.parametrize("intrin, elsizes, scale, fill", [
569: (8)               ("self.load_tillz, self.load_till", (32, 64), 1, [0xffff]),
570: (8)               ("self.load2_tillz, self.load2_till", (32, 64), 2, [0xffff, 0x7fff]),
571: (4)             ])
572: (4)             def test_memory_partial_load(self, intrin, elsizes, scale, fill):
573: (8)               if self._scalar_size() not in elsizes:
574: (12)                 return
575: (8)               npyv_load_tillz, npyv_load_till = eval(intrin)
576: (8)               data = self._data()
577: (8)               lanes = list(range(1, self.nlanes + 1))
578: (8)               lanes += [self.nlanes**2, self.nlanes**4] # test out of range
579: (8)               for n in lanes:
580: (12)                 load_till = npyv_load_till(data, n, *fill)
581: (12)                 load_tillz = npyv_load_tillz(data, n)
582: (12)                 n *= scale
583: (12)                 data_till = data[:n] + fill * ((self.nlanes-n) // scale)
584: (12)                 assert load_till == data_till
585: (12)                 data_tillz = data[:n] + [0] * (self.nlanes-n)
586: (12)                 assert load_tillz == data_tillz
587: (4)               @pytest.mark.parametrize("intrin, elsizes, scale", [
588: (8)                 ("self.store_till", (32, 64), 1),

```

```

589: (8)             ("self.store2_till", (32, 64), 2),
590: (4)         ])
591: (4)     def test_memory_partial_store(self, intrin, elsizes, scale):
592: (8)         if self._scalar_size() not in elsizes:
593: (12)             return
594: (8)         npyv_store_till = eval(intrin)
595: (8)         data = self._data()
596: (8)         data_rev = self._data(reverse=True)
597: (8)         vdata = self.load(data)
598: (8)         lanes = list(range(1, self.nlanes + 1))
599: (8)         lanes += [self.nlanes**2, self.nlanes**4]
600: (8)         for n in lanes:
601: (12)             data_till = data_rev.copy()
602: (12)             data_till[:n*scale] = data[:n*scale]
603: (12)             store_till = self._data(reverse=True)
604: (12)             npyv_store_till(store_till, n, vdata)
605: (12)             assert store_till == data_till
606: (4)     @pytest.mark.parametrize("intrin, elsizes, scale", [
607: (8)         ("self.loadn", (32, 64), 1),
608: (8)         ("self.loadn2", (32, 64), 2),
609: (4)     ])
610: (4)     def test_memory_noncont_load(self, intrin, elsizes, scale):
611: (8)         if self._scalar_size() not in elsizes:
612: (12)             return
613: (8)         npyv_loadn = eval(intrin)
614: (8)         for stride in range(-64, 64):
615: (12)             if stride < 0:
616: (16)                 data = self._data(stride, -stride*self.nlanes)
617: (16)                 data_stride = list(itertools.chain(
618: (20)                     *[zip(*[data[-i::stride] for i in range(scale, 0, -1)])]
619: (16)                 ))
620: (12)             elif stride == 0:
621: (16)                 data = self._data()
622: (16)                 data_stride = data[0:scale] * (self.nlanes//scale)
623: (12)             else:
624: (16)                 data = self._data(count=stride*self.nlanes)
625: (16)                 data_stride = list(itertools.chain(
626: (20)                     *[zip(*[data[i::stride] for i in range(scale)])]
627: (16)                 ))
628: (12)             data_stride = self.load(data_stride) # cast unsigned
629: (12)             loadn = npyv_loadn(data, stride)
630: (12)             assert loadn == data_stride
631: (4)     @pytest.mark.parametrize("intrin, elsizes, scale, fill", [
632: (8)         ("self.loadn_tillz, self.loadn_till", (32, 64), 1, [0xffff]),
633: (8)         ("self.loadn2_tillz, self.loadn2_till", (32, 64), 2, [0xffff,
634: (12)             0x7fff]),
635: (4)     ])
636: (4)     def test_memory_noncont_partial_load(self, intrin, elsizes, scale, fill):
637: (8)         if self._scalar_size() not in elsizes:
638: (12)             return
639: (8)         npyv_loadn_tillz, npyv_loadn_till = eval(intrin)
640: (8)         lanes = list(range(1, self.nlanes + 1))
641: (8)         lanes += [self.nlanes**2, self.nlanes**4]
642: (12)         for stride in range(-64, 64):
643: (16)             if stride < 0:
644: (16)                 data = self._data(stride, -stride*self.nlanes)
645: (20)                 data_stride = list(itertools.chain(
646: (16)                     *[zip(*[data[-i::stride] for i in range(scale, 0, -1)])]
647: (12)                 ))
648: (16)             elif stride == 0:
649: (16)                 data = self._data()
650: (12)                 data_stride = data[0:scale] * (self.nlanes//scale)
651: (16)             else:
652: (16)                 data = self._data(count=stride*self.nlanes)
653: (20)                 data_stride = list(itertools.chain(
654: (16)                     *[zip(*[data[i::stride] for i in range(scale)])]
655: (12)                 ))
656: (12)             data_stride = list(self.load(data_stride)) # cast unsigned
for n in lanes:

```

```

657: (16)             nscale = n * scale
658: (16)             llanes = self.nlanes - nscale
659: (16)             data_stride_till = (
660: (20)                 data_stride[:nscale] + fill * (llanes//scale)
661: (16)             )
662: (16)             loadn_till = npyv_loadn_till(data, stride, n, *fill)
663: (16)             assert loadn_till == data_stride_till
664: (16)             data_stride_tillz = data_stride[:nscale] + [0] * llanes
665: (16)             loadn_tillz = npyv_loadn_tillz(data, stride, n)
666: (16)             assert loadn_tillz == data_stride_tillz
667: (4) @pytest.mark.parametrize("intrin, elsizes, scale", [
668: (8)     ("self.storen", (32, 64), 1),
669: (8)     ("self.storen2", (32, 64), 2),
670: (4) ])
671: (4) def test_memory_noncont_store(self, intrin, elsizes, scale):
672: (8)     if self._scalar_size() not in elsizes:
673: (12)         return
674: (8)     npyv_storen = eval(intrin)
675: (8)     data = self._data()
676: (8)     vdata = self.load(data)
677: (8)     hlanes = self.nlanes // scale
678: (8)     for stride in range(1, 64):
679: (12)         data_storen = [0xff] * stride * self.nlanes
680: (12)         for s in range(0, hlanes*stride, stride):
681: (16)             i = (s//stride)*scale
682: (16)             data_storen[s:s+scale] = data[i:i+scale]
683: (12)         storen = [0xff] * stride * self.nlanes
684: (12)         storen += [0x7f]*64
685: (12)         npyv_storen(storen, stride, vdata)
686: (12)         assert storen[:-64] == data_storen
687: (12)         assert storen[-64:] == [0x7f]*64 # detect overflow
688: (8)     for stride in range(-64, 0):
689: (12)         data_storen = [0xff] * -stride * self.nlanes
690: (12)         for s in range(0, hlanes*stride, stride):
691: (16)             i = (s//stride)*scale
692: (16)             data_storen[s-scale:s or None] = data[i:i+scale]
693: (12)         storen = [0x7f]*64
694: (12)         storen += [0xff] * -stride * self.nlanes
695: (12)         npyv_storen(storen, stride, vdata)
696: (12)         assert storen[64:] == data_storen
697: (12)         assert storen[:64] == [0x7f]*64 # detect overflow
698: (8)     data_storen = [0x7f] * self.nlanes
699: (8)     storen = data_storen.copy()
700: (8)     data_storen[0:scale] = data[-scale:]
701: (8)     npyv_storen(storen, 0, vdata)
702: (8)     assert storen == data_storen
703: (4) @pytest.mark.parametrize("intrin, elsizes, scale", [
704: (8)     ("self.storen_till", (32, 64), 1),
705: (8)     ("self.storen2_till", (32, 64), 2),
706: (4) ])
707: (4) def test_memory_noncont_partial_store(self, intrin, elsizes, scale):
708: (8)     if self._scalar_size() not in elsizes:
709: (12)         return
710: (8)     npyv_storen_till = eval(intrin)
711: (8)     data = self._data()
712: (8)     vdata = self.load(data)
713: (8)     lanes = list(range(1, self.nlanes + 1))
714: (8)     lanes += [self.nlanes**2, self.nlanes**4]
715: (8)     hlanes = self.nlanes // scale
716: (8)     for stride in range(1, 64):
717: (12)         for n in lanes:
718: (16)             data_till = [0xff] * stride * self.nlanes
719: (16)             tdata = data[:n*scale] + [0xff] * (self.nlanes-n*scale)
720: (16)             for s in range(0, hlanes*stride, stride)[:n]:
721: (20)                 i = (s//stride)*scale
722: (20)                 data_till[s:s+scale] = tdata[i:i+scale]
723: (16)             storen_till = [0xff] * stride * self.nlanes
724: (16)             storen_till += [0x7f]*64
725: (16)             npyv_storen_till(storen_till, stride, n, vdata)

```

```

726: (16)                         assert storen_till[:-64] == data_till
727: (16)                         assert storen_till[-64:] == [0x7f]*64 # detect overflow
728: (8)
729: (12)
730: (16)
731: (16)
732: (16)
733: (20)
734: (20)
735: (16)
736: (16)
737: (16)
738: (16)
739: (16)
740: (8)
741: (12)
742: (12)
743: (12)
744: (12)
745: (12)
746: (4) @pytest.mark.parametrize("intrin, table_size, elsize", [
747: (8)     ("self.lut32", 32, 32),
748: (8)     ("self.lut16", 16, 64)
749: (4)])
750: (4)
751: (8)
752: (8)
753: (12)
754: (12)
755: (8)
756: (8)
757: (12)
758: (8)
759: (8)
760: (8)
761: (8)
762: (12)
763: (12)
764: (12)
765: (12)
766: (4)
767: (8)
768: (8)
769: (8)
770: (12)
771: (12)
772: (8)
773: (8)
774: (8)
775: (8)
776: (8)
777: (8)
778: (8)
779: (8)
780: (12)
781: (8)
782: (12)
783: (8)
784: (12)
785: (12)
786: (8)
vdata_b)
787: (8)
788: (8)
vdata_b)
789: (8)
790: (8)
791: (8)
792: (4)
def test_reorder(self):
    assert select_a == data_a
    select_b = self.select(self.cmpeq(self.zero(), self.zero()), vdata_a,
                          vdata_b)
    assert select_b == data_b
    assert self.extract0(vdata_b) == vdata_b[0]
    self.npyv.cleanup()

```

```

793: (8)             data_a, data_b = self._data(), self._data(reverse=True)
794: (8)             vdata_a, vdata_b = self.load(data_a), self.load(data_b)
795: (8)             data_a_lo = data_a[:self.nlanes//2]
796: (8)             data_b_lo = data_b[:self.nlanes//2]
797: (8)             data_a_hi = data_a[self.nlanes//2:]
798: (8)             data_b_hi = data_b[self.nlanes//2:]
799: (8)             combinel = self.combinel(vdata_a, vdata_b)
800: (8)             assert combinel == data_a_lo + data_b_lo
801: (8)             combineh = self.combineh(vdata_a, vdata_b)
802: (8)             assert combineh == data_a_hi + data_b_hi
803: (8)             combine = self.combine(vdata_a, vdata_b)
804: (8)             assert combine == (data_a_lo + data_b_lo, data_a_hi + data_b_hi)
805: (8)             data_zipl = self.load([
806: (12)                 v for p in zip(data_a_lo, data_b_lo) for v in p
807: (8)             ])
808: (8)             data_ziph = self.load([
809: (12)                 v for p in zip(data_a_hi, data_b_hi) for v in p
810: (8)             ])
811: (8)             vzip = self.zip(vdata_a, vdata_b)
812: (8)             assert vzip == (data_zipl, data_ziph)
813: (8)             vzip = [0]*self.nlanes*2
814: (8)             self._x2("store")(vzip, (vdata_a, vdata_b))
815: (8)             assert vzip == list(data_zipl) + list(data_ziph)
816: (8)             unzip = self.unzip(data_zipl, data_ziph)
817: (8)             assert unzip == (data_a, data_b)
818: (8)             unzip = self._x2("load")(list(data_zipl) + list(data_ziph))
819: (8)             assert unzip == (data_a, data_b)
820: (4)             def test_reordered_rev64(self):
821: (8)                 ssize = self._scalar_size()
822: (8)                 if ssize == 64:
823: (12)                     return
824: (8)                 data_rev64 = [
825: (12)                     y for x in range(0, self.nlanes, 64//ssize)
826: (14)                         for y in reversed(range(x, x + 64//ssize))
827: (8)                 ]
828: (8)                 rev64 = self.rev64(self.load(range(self.nlanes)))
829: (8)                 assert rev64 == data_rev64
830: (4)             def test_reordered_permi128(self):
831: (8)                 """
832: (8)                     Test permuting elements for each 128-bit lane.
833: (8)                     npyv_permi128_##sfx
834: (8)                 """
835: (8)                 ssize = self._scalar_size()
836: (8)                 if ssize < 32:
837: (12)                     return
838: (8)                 data = self.load(self._data())
839: (8)                 permn = 128//ssize
840: (8)                 permd = permn-1
841: (8)                 nlane128 = self.nlanes//permn
842: (8)                 shf1 = [0, 1] if ssize == 64 else [0, 2, 4, 6]
843: (8)                 for i in range(permn):
844: (12)                     indices = [(i >> shf) & permd for shf in shf1]
845: (12)                     vperm = self.permi128(data, *indices)
846: (12)                     data_vperm = [
847: (16)                         data[j + (e & -permn)]
848: (16)                         for e, j in enumerate(indices*nlane128)
849: (12)                     ]
850: (12)                     assert vperm == data_vperm
851: (4)             @pytest.mark.parametrize('func, intrin', [
852: (8)                 (operator.lt, "cmplt"),
853: (8)                 (operator.le, "cmple"),
854: (8)                 (operator.gt, "cmpgt"),
855: (8)                 (operator.ge, "cmpge"),
856: (8)                 (operator.eq, "cmpeq")
857: (4)             ])
858: (4)             def test_operators_comparison(self, func, intrin):
859: (8)                 if self._is_fp():
860: (12)                     data_a = self._data()
861: (8)                 else:

```

```

862: (12)             data_a = self._data(self._int_max() - self.nlanes)
863: (8)              data_b = self._data(self._int_min(), reverse=True)
864: (8)              vdata_a, vdata_b = self.load(data_a), self.load(data_b)
865: (8)              intrin = getattr(self, intrin)
866: (8)              mask_true = self._true_mask()
867: (8)              def to_bool(vector):
868: (12)                  return [lane == mask_true for lane in vector]
869: (8)              data_cmp = [func(a, b) for a, b in zip(data_a, data_b)]
870: (8)              cmp = to_bool(intrin(vdata_a, vdata_b))
871: (8)              assert cmp == data_cmp
872: (4)              def test_operators_logical(self):
873: (8)                  if self._is_fp():
874: (12)                      data_a = self._data()
875: (8)                  else:
876: (12)                      data_a = self._data(self._int_max() - self.nlanes)
877: (8)                      data_b = self._data(self._int_min(), reverse=True)
878: (8)                      vdata_a, vdata_b = self.load(data_a), self.load(data_b)
879: (8)                      if self._is_fp():
880: (12)                          data_cast_a = self._to_unsigned(vdata_a)
881: (12)                          data_cast_b = self._to_unsigned(vdata_b)
882: (12)                          cast, cast_data = self._to_unsigned, self._to_unsigned
883: (8)                      else:
884: (12)                          data_cast_a, data_cast_b = data_a, data_b
885: (12)                          cast, cast_data = lambda a: a, self.load
886: (8)                          data_xor = cast_data([a ^ b for a, b in zip(data_cast_a,
887: (8)                            data_cast_b)])
888: (8)                          vxor = cast(self.xor(vdata_a, vdata_b))
889: (8)                          assert vxor == data_xor
890: (8)                          data_or = cast_data([a | b for a, b in zip(data_cast_a,
891: (8)                            data_cast_b)])
892: (8)                          vor = cast(getattr(self, "or"))(vdata_a, vdata_b))
893: (8)                          assert vor == data_or
894: (8)                          data_and = cast_data([a & b for a, b in zip(data_cast_a,
895: (8)                            data_cast_b)])
896: (8)                          vand = cast(getattr(self, "and"))(vdata_a, vdata_b))
897: (8)                          assert vand == data_and
898: (8)                          data_not = cast_data([~a for a in data_cast_a])
899: (12)                          vnot = cast(getattr(self, "not"))(vdata_a))
900: (8)                          assert vnot == data_not
901: (8)                          if self.sfx not in ("u8"):
902: (8)                              return
903: (4)                          data_andc = [a & ~b for a, b in zip(data_cast_a, data_cast_b)]
904: (4)                          vandc = cast(getattr(self, "andc"))(vdata_a, vdata_b))
905: (8)                          assert vandc == data_andc
906: (8) @pytest.mark.parametrize("intrin", ["any", "all"])
907: (8) @pytest.mark.parametrize("data", (
908: (8)    [1, 2, 3, 4],
909: (8)    [-1, -2, -3, -4],
910: (8)    [0, 1, 2, 3, 4],
911: (8)    [0x7f, 0x7fff, 0x7fffffff, 0x7fffffffffffff],
912: (8)    [0, -1, -2, -3, 4],
913: (4)    [0],
914: (4)    [1],
915: (8)    [-1]
916: (4) ))
917: (4) def test_operators_crosstest(self, intrin, data):
918: (8) """
919: (8)     Test intrinsics:
920: (8)         npyv_any_##SFX
921: (8)         npyv_all_##SFX
922: (8) """
923: (8)         data_a = self.load(data * self.nlanes)
924: (8)         func = eval(intrin)
925: (8)         intrin = getattr(self, intrin)
926: (4)         desired = func(data_a)
927: (8)         simd = intrin(data_a)
928: (8)         assert not not simd == desired
929: (4)         def test_conversion_boolean(self):
930: (8)             bsfx = "b" + self.sfx[1:]
```

```

928: (8)          to_boolean = getattr(self.npyv, "cvt_%s_%s" % (bsfx, self.sfx))
929: (8)          from_boolean = getattr(self.npyv, "cvt_%s_%s" % (self.sfx, bsfx))
930: (8)          false_vb = to_boolean(self.setall(0))
931: (8)          true_vb = self.cmpeq(self.setall(0), self.setall(0))
932: (8)          assert false_vb != true_vb
933: (8)          false_vsfx = from_boolean(false_vb)
934: (8)          true_vsfx = from_boolean(true_vb)
935: (8)          assert false_vsfx != true_vsfx
936: (4)          def test_conversion_expand(self):
937: (8)          """
938: (8)              Test expand intrinsics:
939: (12)                  npyv_expand_u16_u8
940: (12)                  npyv_expand_u32_u16
941: (8)
942: (8)
943: (12)
944: (8)          if self.sfx not in ("u8", "u16"):
945: (8)              return
946: (8)          totype = self.sfx[0]+str(int(self.sfx[1:])*2)
947: (8)          expand = getattr(self.npyv, f"expand_{totype}_{self.sfx}")
948: (8)          data = self._data(self._int_max() - self.nlanes)
949: (8)          vdata = self.load(data)
950: (8)          edata = expand(vdata)
951: (8)          data_lo = data[:self.nlanes//2]
952: (4)          def test_arithmetic_subadd(self):
953: (8)              if self._is_fp():
954: (12)                  data_a = self._data()
955: (8)
956: (12)              else:
957: (8)                  data_a = self._data(self._int_max() - self.nlanes)
958: (8)                  data_b = self._data(self._int_min(), reverse=True)
959: (8)                  vdata_a, vdata_b = self.load(data_a), self.load(data_b)
960: (8)                  data_add = self.load([a + b for a, b in zip(data_a, data_b)]) # load
961: (8)                  to cast
962: (8)                  add = self.add(vdata_a, vdata_b)
963: (8)                  assert add == data_add
964: (8)                  data_sub = self.load([a - b for a, b in zip(data_a, data_b)])
965: (4)          def test_arithmetic_mul(self):
966: (8)              if self.sfx in ("u64", "s64"):
967: (12)                  return
968: (8)
969: (12)
970: (8)
971: (12)              if self._is_fp():
972: (8)                  data_a = self._data()
973: (8)
974: (8)                  data_b = self._data(self._int_min(), reverse=True)
975: (8)                  vdata_a, vdata_b = self.load(data_a), self.load(data_b)
976: (8)                  data_mul = self.load([a * b for a, b in zip(data_a, data_b)])
977: (4)          def test_arithmetic_div(self):
978: (8)
979: (12)
980: (8)          if not self._is_fp():
981: (8)              return
982: (8)          data_a, data_b = self._data(), self._data(reverse=True)
983: (8)          vdata_a, vdata_b = self.load(data_a), self.load(data_b)
984: (8)          data_div = self.load([a / b for a, b in zip(data_a, data_b)])
985: (4)          def test_arithmetic_intdiv(self):
986: (8)
987: (8)          """
988: (12)              Test integer division intrinsics:
989: (12)                  npyv_divisor_##sfx
990: (8)                  npyv_divc_##sfx
991: (8)
992: (12)
993: (8)
994: (8)
995: (12)

```

```

996: (12)           Divide towards zero works with large integers > 2^53,
997: (12)           and wrap around overflow similar to what C does.
998: (12)
999: (12)
1000: (16)
1001: (12)
1002: (12)
1003: (16)
1004: (12)
1005: (8)
1006: (8)
1007: (8)
1008: (8)
1009: (8)
1010: (12)
1011: (12)
1012: (8)
1013: (12)
1014: (12)
1015: (8)
1016: (12)
1017: (12)
1018: (8)
1019: (8)
1020: (12)
1021: (12)
1022: (16)
1023: (12)
1024: (12)
1025: (12)
1026: (12)
1027: (12)
1028: (4)
1029: (8)
1030: (8)
1031: (12)
1032: (8)
1033: (8)
1034: (12)
1035: (8)
1036: (8)
1037: (8)
1038: (8)
1039: (8)
1040: (4)
1041: (8)
1042: (8)
1043: (12)
1044: (8)
1045: (8)
1046: (12)
1047: (8)
1048: (8)
1049: (12)
1050: (12)
1051: (12)
1052: (12)
1053: (12)
1054: (4)
1055: (8)
1056: (8)
1057: (8)
1058: (12)
1059: (8)
1060: (8)
1061: (8)
1062: (8)
1063: (8)
1064: (8)

         if d == -1 and a == int_min:
             return a
         sign_a, sign_d = a < 0, d < 0
         if a == 0 or sign_a == sign_d:
             return a // d
         return (a + sign_d - sign_a) // d + 1
data = [1, -int_min] # to test overflow
data += range(0, 2**8, 2**5)
data += range(0, 2**8, 2**5-1)
bsize = self._scalar_size()
if bsize > 8:
    data += range(2**8, 2**16, 2**13)
    data += range(2**8, 2**16, 2**13-1)
if bsize > 16:
    data += range(2**16, 2**32, 2**29)
    data += range(2**16, 2**32, 2**29-1)
if bsize > 32:
    data += range(2**32, 2**64, 2**61)
    data += range(2**32, 2**64, 2**61-1)
data += [-x for x in data]
for dividend, divisor in itertools.product(data, data):
    divisor = self.setall(divisor)[0] # cast
    if divisor == 0:
        continue
    dividend = self.load(self._data(dividend))
    data_divc = [trunc_div(a, divisor) for a in dividend]
    divisor_parms = self.divisor(divisor)
    divc = self.divc(dividend, divisor_parms)
    assert divc == data_divc
def test_arithmetic_reduce_sum(self):
    """
    Test reduce sum intrinsics:
    npyv_sum_##sfx
    """
    if self.sfx not in ("u32", "u64", "f32", "f64"):
        return
    data = self._data()
    vdata = self.load(data)
    data_sum = sum(data)
    vsum = self.sum(vdata)
    assert vsum == data_sum
def test_arithmetic_reduce_sumup(self):
    """
    Test extend reduce sum intrinsics:
    npyv_sumup_##sfx
    """
    if self.sfx not in ("u8", "u16"):
        return
    rdata = (0, self.nlanes, self._int_min(), self._int_max()-self.nlanes)
    for r in rdata:
        data = self._data(r)
        vdata = self.load(data)
        data_sum = sum(data)
        vsum = self.sumup(vdata)
        assert vsum == data_sum
def test_mask_conditional(self):
    """
    Conditional addition and subtraction for all supported data types.
    Test intrinsics:
    npyv_ifadd_##SFX, npyv_ifsub_##SFX
    """
    vdata_a = self.load(self._data())
    vdata_b = self.load(self._data(reverse=True))
    true_mask = self.cmpeq(self.zero(), self.zero())
    false_mask = self.cmpneq(self.zero(), self.zero())
    data_sub = self.sub(vdata_b, vdata_a)

```

```

1065: (8)         ifsub = self.ifsub(true_mask, vdata_b, vdata_a, vdata_b)
1066: (8)         assert ifsub == data_sub
1067: (8)         ifsub = self.ifsub(false_mask, vdata_a, vdata_b, vdata_b)
1068: (8)         assert ifsub == vdata_b
1069: (8)         data_add = self.add(vdata_b, vdata_a)
1070: (8)         ifadd = self.ifadd(true_mask, vdata_b, vdata_a, vdata_b)
1071: (8)         assert ifadd == data_add
1072: (8)         ifadd = self.ifadd(false_mask, vdata_a, vdata_b, vdata_b)
1073: (8)         assert ifadd == vdata_b
1074: (8)         if not self._is_fp():
1075: (12)             return
1076: (8)         data_div = self.div(vdata_b, vdata_a)
1077: (8)         ifdiv = self.ifdiv(true_mask, vdata_b, vdata_a, vdata_b)
1078: (8)         assert ifdiv == data_div
1079: (8)         ifdivz = self.ifdivz(true_mask, vdata_b, vdata_a)
1080: (8)         assert ifdivz == data_div
1081: (8)         ifdiv = self.ifdiv(false_mask, vdata_a, vdata_b, vdata_b)
1082: (8)         assert ifdiv == vdata_b
1083: (8)         ifdivz = self.ifdivz(false_mask, vdata_a, vdata_b)
1084: (8)         assert ifdivz == self.zero()
1085: (0)         bool_sfx = ("b8", "b16", "b32", "b64")
1086: (0)         int_sfx = ("u8", "s8", "u16", "s16", "u32", "s32", "u64", "s64")
1087: (0)         fp_sfx = ("f32", "f64")
1088: (0)         all_sfx = int_sfx + fp_sfx
1089: (0)         tests_registry = {
1090: (4)             bool_sfx: _SIMD_BOOL,
1091: (4)             int_sfx : _SIMD_INT,
1092: (4)             fp_sfx : _SIMD_FP,
1093: (4)             ("f32",): _SIMD_FP32,
1094: (4)             ("f64",): _SIMD_FP64,
1095: (4)             all_sfx : _SIMD_ALL
1096: (0)         }
1097: (0)         for target_name, npyv in targets.items():
1098: (4)             simd_width = npyv.simd if npyv else ''
1099: (4)             pretty_name = target_name.split('__') # multi-target separator
1100: (4)             if len(pretty_name) > 1:
1101: (8)                 pretty_name = f"{' '.join(pretty_name)}"
1102: (4)             else:
1103: (8)                 pretty_name = pretty_name[0]
1104: (4)             skip = ""
1105: (4)             skip_sfx = dict()
1106: (4)             if not npyv:
1107: (8)                 skip = f"target '{pretty_name}' isn't supported by current machine"
1108: (4)             elif not npyv.simd:
1109: (8)                 skip = f"target '{pretty_name}' isn't supported by NPYV"
1110: (4)             else:
1111: (8)                 if not npyvsimd_f32:
1112: (12)                     skip_sfx["f32"] = f"target '{pretty_name}' \"\n                      doesn't support single-precision"
1113: (31)
1114: (8)                     if not npyvsimd_f64:
1115: (12)                         skip_sfx["f64"] = f"target '{pretty_name}' doesn't\"\n                           support double-precision"
1116: (31)
1117: (4)             for sfxes, cls in tests_registry.items():
1118: (8)                 for sfx in sfxes:
1119: (12)                     skip_m = skip_sfx.get(sfx, skip)
1120: (12)                     inhr = (cls,)
1121: (12)                     attr = dict(npyv=targets[target_name], sfx=sfx,
1122: (12)                               target_name=target_name)
1123: (12)                     tcls =
1124: (16)                     type(f"Test{cls.__name__}_{simd_width}_{target_name}_{sfx}", inhr, attr)
1125: (12)                     if skip_m:
1126: (16)                         pytest.mark.skip(reason=skip_m)(tcls)
1127: (12)                     globals()[tcls.__name__] = tcls

```

-----  
File 127 - test\_simd\_module.py:

```
1: (0)         import pytest
```

```

2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (0)
8: (0)
9: (0)
10: (0)
npyv_mod.simd]
11: (0)
12: (0)
13: (0)
14: (0)
15: (0)
16: (4)
17: (0)
18: (4)
19: (0)
20: (0)
21: (0)
NPYV support")
22: (0)
23: (4)
24: (4)
25: (8)
26: (8)
27: (8)
28: (4)
29: (4)
30: (8)
31: (8)
32: (4)
33: (8)
34: (8)
35: (12)
36: (12)
37: (12)
38: (12)
39: (12)
40: (12)
41: (12)
42: (12)
f'reinterpret_{sfx}_u32")(a)
43: (4)
        @pytest.mark.skipif(not npyv2, reason=
44: (8)
            "could not find a second SIMD extension with NPYV support"
45: (4)
)
46: (4)
47: (8)
48: (8)
49: (8)
50: (8)
51: (4)
52: (4)
53: (8)
54: (8)
55: (8)
56: (8)
57: (8)
58: (8)
59: (8)
60: (8)
61: (8)
62: (8)
63: (8)
64: (4)
65: (4)
66: (8)
67: (8)
from numpy.core._simd import targets
"""
This testing unit only for checking the sanity of common functionality,
therefore all we need is just to take one submodule that represents any
of enabled SIMD extensions to run the test on it and the second submodule
required to run only one check related to the possibility of mixing
the data types among each submodule.
"""
npyvs = [npyv_mod for npyv_mod in targets.values() if npyv_mod and
npyv, npyv2 = (npyvs + [None, None])[:2]
unsigned_sfx = ["u8", "u16", "u32", "u64"]
signed_sfx = ["s8", "s16", "s32", "s64"]
fp_sfx = []
if npyv and npyv.simd_f32:
    fp_sfx.append("f32")
if npyv and npyv.simd_f64:
    fp_sfx.append("f64")
int_sfx = unsigned_sfx + signed_sfx
all_sfx = unsigned_sfx + int_sfx
@ pytest.mark.skipif(not npyv, reason="could not find any SIMD extension with
NPYV support")
class Test SIMD_MODULE:
    @pytest.mark.parametrize('sfx', all_sfx)
    def test_num_lanes(self, sfx):
        nlanes = getattr(npyv, "nlanes_" + sfx)
        vector = getattr(npyv, "setall_" + sfx)(1)
        assert len(vector) == nlanes
    @pytest.mark.parametrize('sfx', all_sfx)
    def test_type_name(self, sfx):
        vector = getattr(npyv, "setall_" + sfx)(1)
        assert vector.__name__ == "npyv_" + sfx
    def test_raises(self):
        a, b = [npyv.setall_u32(1)]*2
        for sfx in all_sfx:
            vcb = lambda intrin: getattr(npyv, f"{intrin}_{sfx}")
            pytest.raises(TypeError, vcb("add"), a)
            pytest.raises(TypeError, vcb("add"), a, b, a)
            pytest.raises(TypeError, vcb("setall"))
            pytest.raises(TypeError, vcb("setall"), [1])
            pytest.raises(TypeError, vcb("load"), 1)
            pytest.raises(ValueError, vcb("load"), [1])
            pytest.raises(ValueError, vcb("store"), [1], getattr(npyv,
f'reinterpret_{sfx}_u32')(a))
    @pytest.mark.skipif(not npyv2, reason=
        "could not find a second SIMD extension with NPYV support"
    )
    def test_nomix(self):
        a = npyv.setall_u32(1)
        a2 = npyv2.setall_u32(1)
        pytest.raises(TypeError, npyv.add_u32, a2, a2)
        pytest.raises(TypeError, npyv2.add_u32, a, a)
    @pytest.mark.parametrize('sfx', unsigned_sfx)
    def test_unsigned_overflow(self, sfx):
        nlanes = getattr(npyv, "nlanes_" + sfx)
        maxu = (1 << int(sfx[1:])) - 1
        maxu_72 = (1 << 72) - 1
        lane = getattr(npyv, "setall_" + sfx)(maxu_72)[0]
        assert lane == maxu
        lanes = getattr(npyv, "load_" + sfx)([maxu_72] * nlanes)
        assert lanes == [maxu] * nlanes
        lane = getattr(npyv, "setall_" + sfx)(-1)[0]
        assert lane == maxu
        lanes = getattr(npyv, "load_" + sfx)([-1] * nlanes)
        assert lanes == [maxu] * nlanes
    @pytest.mark.parametrize('sfx', signed_sfx)
    def test_signed_overflow(self, sfx):
        nlanes = getattr(npyv, "nlanes_" + sfx)
        maxs_72 = (1 << 71) - 1

```

```

68: (8)             lane = getattr(npv, "setall_" + sfx)(maxs_72)[0]
69: (8)             assert lane == -1
70: (8)             lanes = getattr(npv, "load_" + sfx)([maxs_72] * nlanes)
71: (8)             assert lanes == [-1] * nlanes
72: (8)             mins_72 = -1 << 71
73: (8)             lane = getattr(npv, "setall_" + sfx)(mins_72)[0]
74: (8)             assert lane == 0
75: (8)             lanes = getattr(npv, "load_" + sfx)([mins_72] * nlanes)
76: (8)             assert lanes == [0] * nlanes
77: (4)             def test_truncate_f32(self):
78: (8)                 if not npvsimd_f32:
79: (12)                     pytest.skip("F32 isn't support by the SIMD extension")
80: (8)                     f32 = npv.setall_f32(0.1)[0]
81: (8)                     assert f32 != 0.1
82: (8)                     assert round(f32, 1) == 0.1
83: (4)             def test_compare(self):
84: (8)                 data_range = range(0, npv.nlanes_u32)
85: (8)                 vdata = npv.load_u32(data_range)
86: (8)                 assert vdata == list(data_range)
87: (8)                 assert vdata == tuple(data_range)
88: (8)                 for i in data_range:
89: (12)                     assert vdata[i] == data_range[i]

```

---

## File 128 - test\_strings.py:

```

1: (0)             import pytest
2: (0)             import operator
3: (0)             import numpy as np
4: (0)             from numpy.testing import assert_array_equal
5: (0)             COMPARISONS = [
6: (4)                 (operator.eq, np.equal, "=="),
7: (4)                 (operator.ne, np.not_equal, "!="),
8: (4)                 (operator.lt, np.less, "<"),
9: (4)                 (operator.le, np.less_equal, "<="),
10: (4)                (operator.gt, np.greater, ">"),
11: (4)                (operator.ge, np.greater_equal, ">="),
12: (0)            ]
13: (0)             @pytest.mark.parametrize(["op", "ufunc", "sym"], COMPARISONS)
14: (0)             def test_mixed_string_comparison_ufuncs_fail(op, ufunc, sym):
15: (4)                 arr_string = np.array(["a", "b"], dtype="S")
16: (4)                 arr_unicode = np.array(["a", "c"], dtype="U")
17: (4)                 with pytest.raises(TypeError, match="did not contain a loop"):
18: (8)                     ufunc(arr_string, arr_unicode)
19: (4)                 with pytest.raises(TypeError, match="did not contain a loop"):
20: (8)                     ufunc(arr_unicode, arr_string)
21: (0)             @pytest.mark.parametrize(["op", "ufunc", "sym"], COMPARISONS)
22: (0)             def test_mixed_string_comparisons_ufuncs_with_cast(op, ufunc, sym):
23: (4)                 arr_string = np.array(["a", "b"], dtype="S")
24: (4)                 arr_unicode = np.array(["a", "c"], dtype="U")
25: (4)                 res1 = ufunc(arr_string, arr_unicode, signature="UU->?", casting="unsafe")
26: (4)                 res2 = ufunc(arr_string, arr_unicode, signature="SS->?", casting="unsafe")
27: (4)                 expected = op(arr_string.astype('U'), arr_unicode)
28: (4)                 assert_array_equal(res1, expected)
29: (4)                 assert_array_equal(res2, expected)
30: (0)             @pytest.mark.parametrize(["op", "ufunc", "sym"], COMPARISONS)
31: (0)             @pytest.mark.parametrize("dtypes", [
32: (8)                 ("S2", "S2"), ("S2", "S10"),
33: (8)                 ("<U1", "<U1"), ("<U1", ">U1"), (">U1", ">U1"),
34: (8)                 ("<U1", "<U10"), ("<U1", ">U10")])
35: (0)             @pytest.mark.parametrize("aligned", [True, False])
36: (0)             def test_string_comparisons(op, ufunc, sym, dtypes, aligned):
37: (4)                 native_dt = np.dtype(dtypes[0]).newbyteorder("=")
38: (4)                 arr = np.arange(2**15).view(native_dt).astype(dtypes[0])
39: (4)                 if not aligned:
40: (8)                     new = np.zeros(arr nbytes + 1, dtype=np.uint8)[1:].view(dtypes[0])
41: (8)                     new[...] = arr
42: (8)                     arr = new

```

```

43: (4) arr2 = arr.astype(dtotypes[1], copy=True)
44: (4) np.random.shuffle(arr2)
45: (4) arr[0] = arr2[0] # make sure one matches
46: (4) expected = [op(d1, d2) for d1, d2 in zip(arr.tolist(), arr2.tolist())]
47: (4) assert_array_equal(op(arr, arr2), expected)
48: (4) assert_array_equal(ufunc(arr, arr2), expected)
49: (4) assert_array_equal(np.compare_chararrays(arr, arr2, sym, False), expected)
50: (4) expected = [op(d2, d1) for d1, d2 in zip(arr.tolist(), arr2.tolist())]
51: (4) assert_array_equal(op(arr2, arr), expected)
52: (4) assert_array_equal(ufunc(arr2, arr), expected)
53: (4) assert_array_equal(np.compare_chararrays(arr2, arr, sym, False), expected)
54: (0) @pytest.mark.parametrize(["op", "ufunc", "sym"], COMPARISONS)
55: (0) @pytest.mark.parametrize("dtypes", [
56: (8)     ("S2", "S2"), ("S2", "S10"), ("<U1", "<U1"), ("<U1", ">U10")]
57: (0) def test_string_comparisons_empty(op, ufunc, sym, dtypes):
58: (4)     arr = np.empty((1, 0, 1, 5), dtype=dtypes[0])
59: (4)     arr2 = np.empty((100, 1, 0, 1), dtype=dtypes[1])
60: (4)     expected = np.empty(np.broadcast_shapes(arr.shape, arr2.shape),
61: (4)         dtype=bool)
62: (4)     assert_array_equal(op(arr, arr2), expected)
63: (4)     assert_array_equal(ufunc(arr, arr2), expected)
64: (4)     assert_array_equal(np.compare_chararrays(arr, arr2, sym, False), expected)
65: (0) @pytest.mark.parametrize("str_dt", ["S", "U"])
66: (0) @pytest.mark.parametrize("float_dt", np.typecodes["AllFloat"])
67: (0) def test_float_to_string_cast(str_dt, float_dt):
68: (4)     float_dt = np.dtype(float_dt)
69: (4)     fi = np.finfo(float_dt)
70: (4)     arr = np.array([np.nan, np.inf, -np.inf, fi.max, fi.min], dtype=float_dt)
71: (4)     expected = ["nan", "inf", "-inf", repr(fi.max), repr(fi.min)]
72: (8)     if float_dt.kind == 'c':
73: (4)         expected = [f"({r}+0j)" for r in expected]
74: (4)     res = arr.astype(str_dt)
75: (4)     assert_array_equal(res, np.array(expected, dtype=str_dt))

```

---

## File 129 - test\_ufunc.py:

```

1: (0) import warnings
2: (0) import itertools
3: (0) import sys
4: (0) import ctypes as ct
5: (0) import pytest
6: (0) from pytest import param
7: (0) import numpy as np
8: (0) import numpy.core._umath_tests as umt
9: (0) import numpy.linalg._umath_linalg as uml
10: (0) import numpy.core._operand_flag_tests as opflag_tests
11: (0) import numpy.core._rational_tests as _rational_tests
12: (0) from numpy.testing import (
13: (4)     assert_, assert_equal, assert_raises, assert_array_equal,
14: (4)     assert_almost_equal, assert_array_almost_equal, assert_no_warnings,
15: (4)     assert_allclose, HAS_REFCOUNT, suppress_warnings, IS_WASM, IS_PYPY,
16: (4) )
17: (0) from numpy.testing._private.utils import requires_memory
18: (0) from numpy.compat import pickle
19: (0) UNARY_UFUNCS = [obj for obj in np.core.umath.__dict__.values()
20: (20)                 if isinstance(obj, np.ufunc)]
21: (0) UNARY_OBJECT_UFUNCS = [uf for uf in UNARY_UFUNCS if "0->0" in uf.types]
22: (0) class TestUfuncKwargs:
23: (4)     def test_kwarg_exact(self):
24: (8)         assert_raises(TypeError, np.add, 1, 2, castingx='safe')
25: (8)         assert_raises(TypeError, np.add, 1, 2, dtypex=int)
26: (8)         assert_raises(TypeError, np.add, 1, 2, extobjx=[4096])
27: (8)         assert_raises(TypeError, np.add, 1, 2, outx=None)
28: (8)         assert_raises(TypeError, np.add, 1, 2, sigx='ii->i')
29: (8)         assert_raises(TypeError, np.add, 1, 2, signaturex='ii->i')
30: (8)         assert_raises(TypeError, np.add, 1, 2, subokx=False)
31: (8)         assert_raises(TypeError, np.add, 1, 2, wherex=[True])

```

```

32: (4)           def test_sig_signature(self):
33: (8)             assert_raises(TypeError, np.add, 1, 2, sig='ii->i',
34: (22)                     signature='ii->i')
35: (4)           def test_sig_dtype(self):
36: (8)             assert_raises(TypeError, np.add, 1, 2, sig='ii->i',
37: (22)                     dtype=int)
38: (8)             assert_raises(TypeError, np.add, 1, 2, signature='ii->i',
39: (22)                     dtype=int)
40: (4)           def test_extobj_refcount(self):
41: (8)             assert_raises(TypeError, np.add, 1, 2, extobj=[4096], parrot=True)
42: (0)         class TestUfuncGenericLoops:
43: (4)           """Test generic loops.
44: (4)           The loops to be tested are:
45: (8)             PyUFunc_ff_f_As_dd_d
46: (8)             PyUFunc_ff_f
47: (8)             PyUFunc_dd_d
48: (8)             PyUFunc_gg_g
49: (8)             PyUFunc_FF_F_As_DD_D
50: (8)             PyUFunc_DD_D
51: (8)             PyUFunc_FF_F
52: (8)             PyUFunc_GG_G
53: (8)             PyUFunc_OO_O
54: (8)             PyUFunc_OO_O_method
55: (8)             PyUFunc_f_f_As_d_d
56: (8)             PyUFunc_d_d
57: (8)             PyUFunc_f_f
58: (8)             PyUFunc_g_g
59: (8)             PyUFunc_F_F_As_D_D
60: (8)             PyUFunc_F_F
61: (8)             PyUFunc_D_D
62: (8)             PyUFunc_G_G
63: (8)             PyUFunc_O_O
64: (8)             PyUFunc_O_O_method
65: (8)             PyUFunc_On_Om
66: (4)           Where:
67: (8)             f -- float
68: (8)             d -- double
69: (8)             g -- long double
70: (8)             F -- complex float
71: (8)             D -- complex double
72: (8)             G -- complex long double
73: (8)             O -- python object
74: (4)           It is difficult to assure that each of these loops is entered from the
75: (4)           Python level as the special cased loops are a moving target and the
76: (4)           corresponding types are architecture dependent. We probably need to
77: (4)           define C level testing ufuncs to get at them. For the time being, I've
78: (4)           just looked at the signatures registered in the build directory to find
79: (4)           relevant functions.
80: (4)           """
81: (4)           np_dtotypes = [
82: (8)             (np.single, np.single), (np.single, np.double),
83: (8)             (np.csingle, np.csingle), (np.csingle, np.cdouble),
84: (8)             (np.double, np.double), (np.longdouble, np.longdouble),
85: (8)             (np.cdouble, np.cdouble), (np.clongdouble, np.clongdouble)]
86: (4)           @pytest.mark.parametrize('input_dtype,output_dtype', np_dtotypes)
87: (4)           def test_unary_PyUFunc(self, input_dtype, output_dtype, f=np.exp, x=0,
y=1):
88: (8)             xs = np.full(10, input_dtype(x), dtype=output_dtype)
89: (8)             ys = f(xs)[:2]
90: (8)             assert_allclose(ys, y)
91: (8)             assert_equal(ys.dtype, output_dtype)
92: (4)           def f2(x, y):
93: (8)             return x**y
94: (4)           @pytest.mark.parametrize('input_dtype,output_dtype', np_dtotypes)
95: (4)           def test_binary_PyUFunc(self, input_dtype, output_dtype, f=f2, x=0, y=1):
96: (8)             xs = np.full(10, input_dtype(x), dtype=output_dtype)
97: (8)             ys = f(xs, xs)[:2]
98: (8)             assert_allclose(ys, y)
99: (8)             assert_equal(ys.dtype, output_dtype)

```

```

100: (4)
101: (8)
102: (12)
103: (8)
104: (12)
105: (4)
106: (8)
107: (8)
108: (4)
109: (8)
110: (8)
111: (4)
112: (8)
113: (8)
114: (4)
115: (8)
116: (8)
117: (4)
118: (8)
119: (8)
120: (4)
121: (8)
122: (8)
123: (8)
124: (8)
125: (8)
126: (4)
127: (4)
128: (8)
129: (8)
130: (8)
131: (12)
132: (16)
133: (20)
134: (16)
135: (20)
136: (8)
137: (8)
138: (8)
139: (12)
140: (16)
141: (12)
142: (16)
143: (20)
144: (12)
145: (16)
146: (16)
147: (0)
148: (4)
149: (0)
150: (4)
151: (8)
152: (12)
153: (46)
154: (12)
155: (44)
156: (12)
157: (4)
158: (8)
159: (19)
160: (8)
161: (4)
162: (4)
163: (8)
164: (8)
165: (8)
166: (4)
167: (8)
168: (8)

    class foo:
        def conjugate(self):
            return np.bool_(1)
        def logical_xor(self, obj):
            return np.bool_(1)
    def test_unary_PyUFunc_O_0(self):
        x = np.ones(10, dtype=object)
        assert_(np.all(np.abs(x) == 1))
    def test_unary_PyUFunc_O_0_method_simple(self, foo=foo):
        x = np.full(10, foo(), dtype=object)
        assert_(np.all(np.conjugate(x) == True))
    def test_binary_PyUFunc_OO_0(self):
        x = np.ones(10, dtype=object)
        assert_(np.all(np.add(x, x) == 2))
    def test_binary_PyUFunc_OO_0_method(self, foo=foo):
        x = np.full(10, foo(), dtype=object)
        assert_(np.all(np.logical_xor(x, x)))
    def test_binary_PyUFunc_On_Om_method(self, foo=foo):
        x = np.full((10, 2, 3), foo(), dtype=object)
        assert_(np.all(np.logical_xor(x, x)))
    def test_python_complex_conjugate(self):
        arr = np.array([1+2j, 3-4j], dtype="O")
        assert isinstance(arr[0], complex)
        res = np.conjugate(arr)
        assert res.dtype == np.dtype("O")
        assert_array_equal(res, np.array([1-2j, 3+4j], dtype="O"))
    @pytest.mark.parametrize("ufunc", UNARY_OBJECT_UFUNCS)
    def test_unary_PyUFunc_O_0_method_full(self, ufunc):
        """Compare the result of the object loop with non-object one"""
        val = np.float64(np.pi/4)
        class MyFloat(np.float64):
            def __getattr__(self, attr):
                try:
                    return super().__getattr__(attr)
                except AttributeError:
                    return lambda: getattr(np.core.umath, attr)(val)
        num_arr = np.array(val, dtype=np.float64)
        obj_arr = np.array(MyFloat(val), dtype="O")
        with np.errstate(all="raise"):
            try:
                res_num = ufunc(num_arr)
            except Exception as exc:
                with assert_raises(type(exc)):
                    ufunc(obj_arr)
            else:
                res_obj = ufunc(obj_arr)
                assert_array_almost_equal(res_num.astype("O"), res_obj)
    def _pickleable_module_global():
        pass
    class TestUfunc:
        def test_pickle(self):
            for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
                assert_(pickle.loads(pickle.dumps(np.sin,
  protocol=proto)) is np.sin)
            res = pickle.loads(pickle.dumps(_rational_tests.test_add,
  protocol=proto))
            assert_(res is _rational_tests.test_add)
        def test_pickle_withstring(self):
            astring = (b"numpy.core\n_ufunc_reconstruct\np0\n"
                       b"(S'numpy.core.umath'\nnp1\nS'cos'\nnp2\nntp3\nRp4\n."
            assert_(pickle.loads(astring) is np.cos)
    @pytest.mark.skipif(IS_PYPY, reason="'is' check does not work on PyPy")
    def test_pickle_name_is_qualname(self):
        _pickleable_module_global.ufunc = umt._pickleable_module_global_ufunc
        obj = pickle.loads(pickle.dumps(_pickleable_module_global.ufunc))
        assert obj is umt._pickleable_module_global_ufunc
    def test_reduceat_shifting_sum(self):
        L = 6
        x = np.arange(L)

```

```

169: (8)           idx = np.array(list(zip(np.arange(L - 2), np.arange(L - 2) +
2)).ravel())
170: (8)           assert_array_equal(np.add.reduceat(x, idx)[::2], [1, 3, 5, 7])
171: (4)           def test_all_ufunc(self):
172: (8)               """Try to check presence and results of all ufuncs.
173: (8)               The list of ufuncs comes from generate_umath.py and is as follows:
174: (8)               ===== ===== ===== ===== ===== =====
175: (8)               done   args   function      types      notes
176: (8)               =====   =====   =====      =====      =====
177: (8)               n      1      conjugate    nums + 0
178: (8)               n      1      absolute     nums + 0      complex -> real
179: (8)               n      1      negative    nums + 0
180: (8)               n      1      sign        nums + 0      -> int
181: (8)               n      1      invert       bool + ints + 0  flts raise an error
182: (8)               n      1      degrees     real + M      cmplx raise an error
183: (8)               n      1      radians    real + M      cmplx raise an error
184: (8)               n      1      arccos     flts + M
185: (8)               n      1      arccosh    flts + M
186: (8)               n      1      arcsin     flts + M
187: (8)               n      1      arcsinh    flts + M
188: (8)               n      1      arctan     flts + M
189: (8)               n      1      arctanh    flts + M
190: (8)               n      1      cos         flts + M
191: (8)               n      1      sin         flts + M
192: (8)               n      1      tan         flts + M
193: (8)               n      1      cosh        flts + M
194: (8)               n      1      sinh        flts + M
195: (8)               n      1      tanh        flts + M
196: (8)               n      1      exp         flts + M
197: (8)               n      1      expm1      flts + M
198: (8)               n      1      log         flts + M
199: (8)               n      1      log10      flts + M
200: (8)               n      1      log1p      flts + M
201: (8)               n      1      sqrt        flts + M      real x < 0 raises error
202: (8)               n      1      ceil        real + M
203: (8)               n      1      trunc      real + M
204: (8)               n      1      floor      real + M
205: (8)               n      1      fabs        real + M
206: (8)               n      1      rint        flts + M
207: (8)               n      1      isnan      flts          -> bool
208: (8)               n      1      isinf      flts          -> bool
209: (8)               n      1      isfinite   flts          -> bool
210: (8)               n      1      signbit    real          -> bool
211: (8)               n      1      modf       real          -> (frac, int)
212: (8)               n      1      logical_not bool + nums + M -> bool
213: (8)               n      2      left_shift ints + 0      flts raise an error
214: (8)               n      2      right_shift ints + 0      flts raise an error
215: (8)               n      2      add         bool + nums + 0  boolean + is ||
216: (8)               n      2      subtract   bool + nums + 0  boolean - is ^
217: (8)               n      2      multiply   bool + nums + 0  boolean * is &
218: (8)               n      2      divide     nums + 0
219: (8)               n      2      floor_divide nums + 0
220: (8)               n      2      true_divide nums + 0      bBhH -> f, iIlLqQ -> d
221: (8)               n      2      fmod       nums + M
222: (8)               n      2      power     nums + 0
223: (8)               n      2      greater    bool + nums + 0 -> bool
224: (8)               n      2      greater_equal bool + nums + 0 -> bool
225: (8)               n      2      less       bool + nums + 0 -> bool
226: (8)               n      2      less_equal  bool + nums + 0 -> bool
227: (8)               n      2      equal      bool + nums + 0 -> bool
228: (8)               n      2      not_equal  bool + nums + 0 -> bool
229: (8)               n      2      logical_and bool + nums + M -> bool
230: (8)               n      2      logical_or  bool + nums + M -> bool
231: (8)               n      2      logical_xor bool + nums + M -> bool
232: (8)               n      2      maximum    bool + nums + 0
233: (8)               n      2      minimum    bool + nums + 0
234: (8)               n      2      bitwise_and bool + ints + 0  flts raise an error
235: (8)               n      2      bitwise_or  bool + ints + 0  flts raise an error
236: (8)               n      2      bitwise_xor bool + ints + 0  flts raise an error

```

```

237: (8)           n      2      arctan2      real + M
238: (8)           n      2      remainder    ints + real + 0
239: (8)           n      2      hypot       real + M
240: (8)           =====  =====  =====  =====  =====
241: (8)           Types other than those listed will be accepted, but they are cast to
242: (8)           the smallest compatible type for which the function is defined. The
243: (8)           casting rules are:
244: (8)           bool -> int8 -> float32
245: (8)           ints -> double
246: (8)           """
247: (8)           pass
248: (4)           size_inferred = 2
249: (4)           can_ignore = 4
250: (4)           def test_signature0(self):
251: (8)             enabled, num_dims, ixs, flags, sizes = umt.test_signature(
252: (12)               2, 1, "(i),(i)->()")
253: (8)             assert_equal(enabled, 1)
254: (8)             assert_equal(num_dims, (1, 1, 0))
255: (8)             assert_equal(ixs, (0, 0))
256: (8)             assert_equal(flags, (self.size_inferred,))
257: (8)             assert_equal(sizes, (-1, ))
258: (4)           def test_signature1(self):
259: (8)             enabled, num_dims, ixs, flags, sizes = umt.test_signature(
260: (12)               2, 1, "(),()->()")
261: (8)             assert_equal(enabled, 0)
262: (8)             assert_equal(num_dims, (0, 0, 0))
263: (8)             assert_equal(ixs, ())
264: (8)             assert_equal(flags, ())
265: (8)             assert_equal(sizes, ())
266: (4)           def test_signature2(self):
267: (8)             enabled, num_dims, ixs, flags, sizes = umt.test_signature(
268: (12)               2, 1, "(i1,i2),(J_1)->(_KAB)")
269: (8)             assert_equal(enabled, 1)
270: (8)             assert_equal(num_dims, (2, 1, 1))
271: (8)             assert_equal(ixs, (0, 1, 2, 3))
272: (8)             assert_equal(flags, (self.size_inferred,)*4)
273: (8)             assert_equal(sizes, (-1, -1, -1, -1))
274: (4)           def test_signature3(self):
275: (8)             enabled, num_dims, ixs, flags, sizes = umt.test_signature(
276: (12)               2, 1, "(i1, i12), (J_1)->(i12, i2)")
277: (8)             assert_equal(enabled, 1)
278: (8)             assert_equal(num_dims, (2, 1, 2))
279: (8)             assert_equal(ixs, (0, 1, 2, 1, 3))
280: (8)             assert_equal(flags, (self.size_inferred,)*4)
281: (8)             assert_equal(sizes, (-1, -1, -1, -1))
282: (4)           def test_signature4(self):
283: (8)             enabled, num_dims, ixs, flags, sizes = umt.test_signature(
284: (12)               2, 1, "(n,k),(k,m)->(n,m)")
285: (8)             assert_equal(enabled, 1)
286: (8)             assert_equal(num_dims, (2, 2, 2))
287: (8)             assert_equal(ixs, (0, 1, 1, 2, 0, 2))
288: (8)             assert_equal(flags, (self.size_inferred,)*3)
289: (8)             assert_equal(sizes, (-1, -1, -1))
290: (4)           def test_signature5(self):
291: (8)             enabled, num_dims, ixs, flags, sizes = umt.test_signature(
292: (12)               2, 1, "(n?,k),(k,m?)->(n?,m?)")
293: (8)             assert_equal(enabled, 1)
294: (8)             assert_equal(num_dims, (2, 2, 2))
295: (8)             assert_equal(ixs, (0, 1, 1, 2, 0, 2))
296: (8)             assert_equal(flags, (self.size_inferred | self.can_ignore,
297: (29)                           self.size_inferred,
298: (29)                           self.size_inferred | self.can_ignore))
299: (8)             assert_equal(sizes, (-1, -1, -1))
300: (4)           def test_signature6(self):
301: (8)             enabled, num_dims, ixs, flags, sizes = umt.test_signature(
302: (12)               1, 1, "(3)->()")
303: (8)             assert_equal(enabled, 1)
304: (8)             assert_equal(num_dims, (1, 0))
305: (8)             assert_equal(ixs, (0,))
```

```

306: (8) assert_equal(flags, (0,))
307: (8) assert_equal(sizes, (3,))
308: (4) def test_signature7(self):
309: (8)     enabled, num_dims, ixs, flags, sizes = umt.test_signature(
310: (12)         3, 1, "(3),(03,3),(n)->(9)")
311: (8)     assert_equal(enabled, 1)
312: (8)     assert_equal(num_dims, (1, 2, 1, 1))
313: (8)     assert_equal(ixs, (0, 0, 0, 1, 2))
314: (8)     assert_equal(flags, (0, self.size_inferred, 0))
315: (8)     assert_equal(sizes, (3, -1, 9))
316: (4) def test_signature8(self):
317: (8)     enabled, num_dims, ixs, flags, sizes = umt.test_signature(
318: (12)         3, 1, "(3?),(3?,3?),(n)->(9)")
319: (8)     assert_equal(enabled, 1)
320: (8)     assert_equal(num_dims, (1, 2, 1, 1))
321: (8)     assert_equal(ixs, (0, 0, 0, 1, 2))
322: (8)     assert_equal(flags, (self.can_ignore, self.size_inferred, 0))
323: (8)     assert_equal(sizes, (3, -1, 9))
324: (4) def test_signature9(self):
325: (8)     enabled, num_dims, ixs, flags, sizes = umt.test_signature(
326: (12)         1, 1, "( 3 ) -> ( )")
327: (8)     assert_equal(enabled, 1)
328: (8)     assert_equal(num_dims, (1, 0))
329: (8)     assert_equal(ixs, (0,))
330: (8)     assert_equal(flags, (0,))
331: (8)     assert_equal(sizes, (3,))
332: (4) def test_signature10(self):
333: (8)     enabled, num_dims, ixs, flags, sizes = umt.test_signature(
334: (12)         3, 1, "( 3? ) , (3? , 3?) ,(n )-> ( 9)")
335: (8)     assert_equal(enabled, 1)
336: (8)     assert_equal(num_dims, (1, 2, 1, 1))
337: (8)     assert_equal(ixs, (0, 0, 0, 1, 2))
338: (8)     assert_equal(flags, (self.can_ignore, self.size_inferred, 0))
339: (8)     assert_equal(sizes, (3, -1, 9))
340: (4) def test_signature_failure_extra_parenthesis(self):
341: (8)     with assert_raises(ValueError):
342: (12)         umt.test_signature(2, 1, "((i)),(i)->()")
343: (4) def test_signature_failure_mismatching_parenthesis(self):
344: (8)     with assert_raises(ValueError):
345: (12)         umt.test_signature(2, 1, "(i),)i(->()")
346: (4) def test_signature_failure_signature_missing_input_arg(self):
347: (8)     with assert_raises(ValueError):
348: (12)         umt.test_signature(2, 1, "(i),->()")
349: (4) def test_signature_failure_signature_missing_output_arg(self):
350: (8)     with assert_raises(ValueError):
351: (12)         umt.test_signature(2, 2, "(i),(i)->()")
352: (4) def test_get_signature(self):
353: (8)     assert_equal(umt.inner1d.signature, "(i),(i)->()")
354: (4) def test_forced_sig(self):
355: (8)     a = 0.5*np.arange(3, dtype='f8')
356: (8)     assert_equal(np.add(a, 0.5), [0.5, 1, 1.5])
357: (8)     with pytest.warns(DeprecationWarning):
358: (12)         assert_equal(np.add(a, 0.5, sig='i', casting='unsafe'), [0, 0, 1])
359: (8)         assert_equal(np.add(a, 0.5, sig='ii->i', casting='unsafe'), [0, 0, 1])
360: (8)     with pytest.warns(DeprecationWarning):
361: (12)         assert_equal(np.add(a, 0.5, sig='i4',), casting='unsafe'),
362: (25)             [0, 0, 1])
363: (8)         assert_equal(np.add(a, 0.5, sig='i4', 'i4', 'i4'),
364: (44)   casting='unsafe'), [0, 0, 1])
365: (8)         b = np.zeros((3,), dtype='f8')
366: (8)         np.add(a, 0.5, out=b)
367: (8)         assert_equal(b, [0.5, 1, 1.5])
368: (8)         b[:] = 0
369: (8)         with pytest.warns(DeprecationWarning):
370: (12)             np.add(a, 0.5, sig='i', out=b, casting='unsafe')
371: (8)             assert_equal(b, [0, 0, 1])
372: (8)             b[:] = 0
373: (8)             np.add(a, 0.5, sig='ii->i', out=b, casting='unsafe')
374: (8)             assert_equal(b, [0, 0, 1])

```

```

375: (8)          b[:] = 0
376: (8)          with pytest.warns(DeprecationWarning):
377: (12)            np.add(a, 0.5, sig='i4',), out=b, casting='unsafe')
378: (8)          assert_equal(b, [0, 0, 1])
379: (8)          b[:] = 0
380: (8)          np.add(a, 0.5, sig='i4', 'i4', 'i4'), out=b, casting='unsafe')
381: (8)          assert_equal(b, [0, 0, 1])
382: (4)          def test_signature_all_None(self):
383: (8)            res1 = np.add([3], [4], sig=(None, None, None))
384: (8)            res2 = np.add([3], [4])
385: (8)            assert_array_equal(res1, res2)
386: (8)            res1 = np.maximum([3], [4], sig=(None, None, None))
387: (8)            res2 = np.maximum([3], [4])
388: (8)            assert_array_equal(res1, res2)
389: (8)            with pytest.raises(TypeError):
390: (12)              np.add(3, 4, signature=(None,))
391: (4)          def test_signature_dtype_type(self):
392: (8)            float_dtype = type(np.dtype(np.float64))
393: (8)            np.add(3, 4, signature=(float_dtype, float_dtype, None))
394: (4)          @pytest.mark.parametrize("get_kwarg", [
395: (12)            lambda dt: dict(dtype=x),
396: (12)            lambda dt: dict(signature=(x, None, None))])
397: (4)          def test_signature_dtype_instances_allowed(self, get_kwarg):
398: (8)            int64 = np.dtype("int64")
399: (8)            int64_2 = pickle.loads(pickle.dumps(int64))
400: (8)            assert int64 is not int64_2
401: (8)            assert np.add(1, 2, **get_kwarg(int64_2)).dtype == int64
402: (8)            td = np.timedelta(2, "s")
403: (8)            assert np.add(td, td, **get_kwarg("m8")).dtype == "m8[s]"
404: (4)          @pytest.mark.parametrize("get_kwarg", [
405: (12)            param(lambda x: dict(dtype=x), id="dtype"),
406: (12)            param(lambda x: dict(signature=(x, None, None)), id="signature")))
407: (4)          def test_signature_dtype_instances_allowed(self, get_kwarg):
408: (8)            msg = "The `dtype` and `signature` arguments to ufuncs"
409: (8)            with pytest.raises(TypeError, match=msg):
410: (12)              np.add(3, 5, **get_kwarg(np.dtype("int64").newbyteorder()))
411: (8)            with pytest.raises(TypeError, match=msg):
412: (12)              np.add(3, 5, **get_kwarg(np.dtype("m8[ns]")))
413: (8)            with pytest.raises(TypeError, match=msg):
414: (12)              np.add(3, 5, **get_kwarg("m8[ns]"))
415: (4)          @pytest.mark.parametrize("casting", ["unsafe", "same_kind", "safe"])
416: (4)          def test_partial_signature_mismatch(self, casting):
417: (8)            res = np.ldexp(np.float32(1.), np.int_(2), dtype="d")
418: (8)            assert res.dtype == "d"
419: (8)            res = np.ldexp(np.float32(1.), np.int_(2), signature=(None, None,
420: ("d")))
421: (8)            assert res.dtype == "d"
422: (12)          with pytest.raises(TypeError):
423: (8)            np.ldexp(1., np.uint64(3), dtype="d")
424: (12)          with pytest.raises(TypeError):
425: (4)            np.ldexp(1., np.uint64(3), signature=(None, None, "d"))
426: (8)          def test_partial_signature_mismatch_with_cache(self):
427: (12)            with pytest.raises(TypeError):
428: (8)              np.add(np.float16(1), np.uint64(2), sig=("e", "d", None))
429: (8)            np.add(np.float16(1), np.float64(2))
430: (8)            with pytest.raises(TypeError):
431: (12)              np.add(np.float16(1), np.uint64(2), sig=("e", "d", None))
432: (4)          def test_use_output_signature_for_all_arguments(self):
433: (8)            res = np.power(1.5, 2.8, dtype=np.intp, casting="unsafe")
434: (8)            assert res == 1 # the cast happens first.
435: (23)          res = np.power(1.5, 2.8, signature=(None, None, np.intp),
436: ("casting="unsafe"))
437: (8)          assert res == 1
438: (8)          with pytest.raises(TypeError):
439: (12)            np.power(1.5, 2.8, dtype=np.intp)
440: (4)          def test_signature_errors(self):
441: (8)            with pytest.raises(TypeError,
442: ("match="the signature object to ufunc must be a string
443: or")):
```

```

442: (12) np.add(3, 4, signature=123.) # neither a string nor a tuple
443: (8) with pytest.raises(ValueError):
444: (12)     np.add(3, 4, signature="%^->#")
445: (8) with pytest.raises(ValueError):
446: (12)     np.add(3, 4, signature=b"ii-i") # incomplete and byte string
447: (8) with pytest.raises(ValueError):
448: (12)     np.add(3, 4, signature="ii>i") # incomplete string
449: (8) with pytest.raises(ValueError):
450: (12)     np.add(3, 4, signature=(None, "f8")) # bad length
451: (8) with pytest.raises(UnicodeDecodeError):
452: (12)     np.add(3, 4, signature=b"\xff\xff->i")
453: (4) def test_forced_dtype_times(self):
454: (8)     a = np.array(['2010-01-02', '1999-03-14', '1833-03'], dtype='>M8[D]')
455: (8)     np.maximum(a, a, dtype="M")
456: (8)     np.maximum.reduce(a, dtype="M")
457: (8)     arr = np.arange(10, dtype="m8[s]")
458: (8)     np.add(arr, arr, dtype="m")
459: (8)     np.maximum(arr, arr, dtype="m")
460: (4) @pytest.mark.parametrize("ufunc", [np.add, np.sqrt])
461: (4) def test_cast_safety(self, ufunc):
462: (8)     """Basic test for the safest casts, because ufuncs inner loops can
463: (8)     indicate a cast-safety as well (which is normally always "no").
464: (8) """
465: (8)     def call_ufunc(arr, **kwargs):
466: (12)         return ufunc(*(arr,) * ufunc.nin, **kwargs)
467: (8)     arr = np.array([1., 2., 3.], dtype=np.float32)
468: (8)     arr_bs = arr.astype(arr.dtype.newbyteorder())
469: (8)     expected = call_ufunc(arr)
470: (8)     res = call_ufunc(arr, casting="no")
471: (8)     assert_array_equal(expected, res)
472: (8)     with pytest.raises(TypeError):
473: (12)         call_ufunc(arr_bs, casting="no")
474: (8)     res = call_ufunc(arr_bs, casting="equiv")
475: (8)     assert_array_equal(expected, res)
476: (8)     with pytest.raises(TypeError):
477: (12)         call_ufunc(arr_bs, dtype=np.float64, casting="equiv")
478: (8)     res = call_ufunc(arr_bs, dtype=np.float64, casting="safe")
479: (8)     expected = call_ufunc(arr.astype(np.float64)) # upcast
480: (8)     assert_array_equal(expected, res)
481: (4) def test_true_divide(self):
482: (8)     a = np.array(10)
483: (8)     b = np.array(20)
484: (8)     tgt = np.array(0.5)
485: (8)     for tc in 'bhilqBHILQefdgFDG':
486: (12)         dt = np.dtype(tc)
487: (12)         aa = a.astype(dt)
488: (12)         bb = b.astype(dt)
489: (12)         for x, y in itertools.product([aa, -aa], [bb, -bb]):
490: (16)             if tc in 'FDG':
491: (20)                 tgt = complex(x)/complex(y)
492: (16)             else:
493: (20)                 tgt = float(x)/float(y)
494: (16)             res = np.true_divide(x, y)
495: (16)             rtol = max(np.finfo(res).resolution, 1e-15)
496: (16)             assert_allclose(res, tgt, rtol=rtol)
497: (16)             if tc in 'bhilqBHILQ':
498: (20)                 assert_(res.dtype.name == 'float64')
499: (16)             else:
500: (20)                 assert_(res.dtype.name == dt.name )
501: (16)             for tcout in 'bhilqBHILQ':
502: (20)                 dtout = np.dtype(tcout)
503: (20)                 assert_raises(TypeError, np.true_divide, x, y,
504: (16)                     for tcout in 'efdg':
505: (20)                         dtout = np.dtype(tcout)
506: (20)                         if tc in 'FDG':
507: (24)                             assert_raises(TypeError, np.true_divide, x, y,
508: (20)                                 else:

```

```

509: (24)             tgt = float(x)/float(y)
510: (24)             rtol = max(np.finfo(dtout).resolution, 1e-15)
511: (24)             with suppress_warnings() as sup:
512: (28)                 sup.filter(UserWarning)
513: (28)                 if not np.isnan(np.finfo(dtout).tiny):
514: (32)                     atol = max(np.finfo(dtout).tiny, 3e-308)
515: (28)                 else:
516: (32)                     atol = 3e-308
517: (24)             with np.errstate(invalid='ignore', over='ignore'):
518: (28)                 res = np.true_divide(x, y, dtype=dtout)
519: (24)                 if not np.isfinite(res) and tcout == 'e':
520: (28)                     continue
521: (24)                 assert_allclose(res, tgt, rtol=rtol, atol=atol)
522: (24)                 assert_(res.dtype.name == dtout.name)
523: (16)             for tcout in 'FDG':
524: (20)                 dtout = np.dtype(tcout)
525: (20)                 tgt = complex(x)/complex(y)
526: (20)                 rtol = max(np.finfo(dtout).resolution, 1e-15)
527: (20)                 with suppress_warnings() as sup:
528: (24)                     sup.filter(UserWarning)
529: (24)                     if not np.isnan(np.finfo(dtout).tiny):
530: (28)                         atol = max(np.finfo(dtout).tiny, 3e-308)
531: (24)                     else:
532: (28)                         atol = 3e-308
533: (20)                 res = np.true_divide(x, y, dtype=dtout)
534: (20)                 if not np.isfinite(res):
535: (24)                     continue
536: (20)                 assert_allclose(res, tgt, rtol=rtol, atol=atol)
537: (20)                 assert_(res.dtype.name == dtout.name)
538: (8)             a = np.ones((), dtype=np.bool_)
539: (8)             res = np.true_divide(a, a)
540: (8)             assert_(res == 1.0)
541: (8)             assert_(res.dtype.name == 'float64')
542: (8)             res = np.true_divide(~a, a)
543: (8)             assert_(res == 0.0)
544: (8)             assert_(res.dtype.name == 'float64')
545: (4)         def test_sum_stability(self):
546: (8)             a = np.ones(500, dtype=np.float32)
547: (8)             assert_almost_equal((a / 10.).sum() - a.size / 10., 0, 4)
548: (8)             a = np.ones(500, dtype=np.float64)
549: (8)             assert_almost_equal((a / 10.).sum() - a.size / 10., 0, 13)
550: (4)         @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
551: (4)         def test_sum(self):
552: (8)             for dt in (int, np.float16, np.float32, np.float64, np.longdouble):
553: (12)                 for v in (0, 1, 2, 7, 8, 9, 15, 16, 19, 127,
554: (22)                         128, 1024, 1235):
555: (16)                     with warnings.catch_warnings(record=True) as w:
556: (20)                         warnings.simplefilter("always", RuntimeWarning)
557: (20)                         tgt = dt(v * (v + 1) / 2)
558: (20)                         overflow = not np.isfinite(tgt)
559: (20)                         assert_equal(len(w), 1 * overflow)
560: (20)                         d = np.arange(1, v + 1, dtype=dt)
561: (20)                         assert_almost_equal(np.sum(d), tgt)
562: (20)                         assert_equal(len(w), 2 * overflow)
563: (20)                         assert_almost_equal(np.sum(d[::-1]), tgt)
564: (20)                         assert_equal(len(w), 3 * overflow)
565: (12)                     d = np.ones(500, dtype=dt)
566: (12)                     assert_almost_equal(np.sum(d[::-2]), 250.)
567: (12)                     assert_almost_equal(np.sum(d[1:-2]), 250.)
568: (12)                     assert_almost_equal(np.sum(d[::-3]), 167.)
569: (12)                     assert_almost_equal(np.sum(d[1:-3]), 167.)
570: (12)                     assert_almost_equal(np.sum(d[::-2]), 250.)
571: (12)                     assert_almost_equal(np.sum(d[-1:-2]), 250.)
572: (12)                     assert_almost_equal(np.sum(d[::-3]), 167.)
573: (12)                     assert_almost_equal(np.sum(d[-1:-3]), 167.)
574: (12)                     d = np.ones((1,), dtype=dt)
575: (12)                     d += d
576: (12)                     assert_almost_equal(d, 2.)
577: (4)         def test_sum_complex(self):

```

```

578: (8)           for dt in (np.complex64, np.complex128, np.clongdouble):
579: (12)             for v in (0, 1, 2, 7, 8, 9, 15, 16, 19, 127,
580: (22)                   128, 1024, 1235):
581: (16)               tgt = dt(v * (v + 1) / 2) - dt((v * (v + 1) / 2) * 1j)
582: (16)               d = np.empty(v, dtype=dt)
583: (16)               d.real = np.arange(1, v + 1)
584: (16)               d.imag = -np.arange(1, v + 1)
585: (16)               assert_almost_equal(np.sum(d), tgt)
586: (16)               assert_almost_equal(np.sum(d[::-1]), tgt)
587: (12)             d = np.ones(500, dtype=dt) + 1j
588: (12)             assert_almost_equal(np.sum(d[:2]), 250. + 250j)
589: (12)             assert_almost_equal(np.sum(d[1:2]), 250. + 250j)
590: (12)             assert_almost_equal(np.sum(d[:3]), 167. + 167j)
591: (12)             assert_almost_equal(np.sum(d[1:3]), 167. + 167j)
592: (12)             assert_almost_equal(np.sum(d[:-2]), 250. + 250j)
593: (12)             assert_almost_equal(np.sum(d[-1::-2]), 250. + 250j)
594: (12)             assert_almost_equal(np.sum(d[:-3]), 167. + 167j)
595: (12)             assert_almost_equal(np.sum(d[-1::-3]), 167. + 167j)
596: (12)             d = np.ones((1,), dtype=dt) + 1j
597: (12)             d += d
598: (12)             assert_almost_equal(d, 2. + 2j)
599: (4)           def test_sum_initial(self):
600: (8)             assert_equal(np.sum([3], initial=2), 5)
601: (8)             assert_almost_equal(np.sum([0.2], initial=0.1), 0.3)
602: (8)             assert_equal(np.sum(np.ones((2, 3, 5), dtype=np.int64), axis=(0, 2),
initial=2),
603: (21)                     [12, 12, 12])
604: (4)           def test_sum_where(self):
605: (8)             assert_equal(np.sum([[1., 2.], [3., 4.]], where=[True, False]), 4.)
606: (8)             assert_equal(np.sum([[1., 2.], [3., 4.]], axis=0, initial=5.,
where=[True, False]), [9., 5.])
607: (28)
608: (4)           def test_inner1d(self):
609: (8)             a = np.arange(6).reshape((2, 3))
610: (8)             assert_array_equal(umt.inner1d(a, a), np.sum(a*a, axis=-1))
611: (8)             a = np.arange(6)
612: (8)             assert_array_equal(umt.inner1d(a, a), np.sum(a*a))
613: (4)           def test_broadcast(self):
614: (8)             msg = "broadcast"
615: (8)             a = np.arange(4).reshape((2, 1, 2))
616: (8)             b = np.arange(4).reshape((1, 2, 2))
617: (8)             assert_array_equal(umt.inner1d(a, b), np.sum(a*b, axis=-1),
err_msg=msg)
618: (8)             msg = "extend & broadcast loop dimensions"
619: (8)             b = np.arange(4).reshape((2, 2))
620: (8)             assert_array_equal(umt.inner1d(a, b), np.sum(a*b, axis=-1),
err_msg=msg)
621: (8)             a = np.arange(8).reshape((4, 2))
622: (8)             b = np.arange(4).reshape((4, 1))
623: (8)             assert_raises(ValueError, umt.inner1d, a, b)
624: (8)             a = np.arange(8).reshape((4, 2))
625: (8)             b = np.array(7)
626: (8)             assert_raises(ValueError, umt.inner1d, a, b)
627: (8)             a = np.arange(2).reshape((2, 1, 1))
628: (8)             b = np.arange(3).reshape((3, 1, 1))
629: (8)             assert_raises(ValueError, umt.inner1d, a, b)
630: (8)             a = np.arange(2)
631: (8)             b = np.arange(4).reshape((2, 2))
632: (8)             u, v = np.broadcast_arrays(a, b)
633: (8)             assert_equal(u.strides[0], 0)
634: (8)             x = u + v
635: (8)             with warnings.catch_warnings(record=True) as w:
636: (12)               warnings.simplefilter("always")
637: (12)               u += v
638: (12)               assert_equal(len(w), 1)
639: (12)               assert_(x[0, 0] != u[0, 0])
640: (8)               a = np.arange(6).reshape(3, 2)
641: (8)               b = np.ones(2)
642: (8)               out = np.empty(())
643: (8)               assert_raises(ValueError, umt.inner1d, a, b, out)

```

```

644: (8)          out2 = np.empty(3)
645: (8)          c = umt.inner1d(a, b, out2)
646: (8)          assert_(c is out2)
647: (4)          def test_out_broadcasts(self):
648: (8)              arr = np.arange(3).reshape(1, 3)
649: (8)              out = np.empty((5, 4, 3))
650: (8)              np.add(arr, arr, out=out)
651: (8)              assert (out == np.arange(3) * 2).all()
652: (8)              umt.inner1d(arr, arr, out=out)
653: (8)              assert (out == 5).all()
654: (4)          @pytest.mark.parametrize(["arr", "out"], [
655: (16)              ([2], np.empty(())),
656: (16)              ([1, 2], np.empty(1)),
657: (16)              (np.ones((4, 3)), np.empty((4, 1))]),
658: (12)              ids=["(1,)->()", "(2,)->(1,)", "(4, 3)->(4, 1)"])
659: (4)          def test_out_broadcast_errors(self, arr, out):
660: (8)              with pytest.raises(ValueError, match="non-broadcastable"):
661: (12)                  np.positive(arr, out=out)
662: (8)              with pytest.raises(ValueError, match="non-broadcastable"):
663: (12)                  np.add(np.ones(()), arr, out=out)
664: (4)          def test_type_cast(self):
665: (8)              msg = "type cast"
666: (8)              a = np.arange(6, dtype='short').reshape((2, 3))
667: (8)              assert_array_equal(umt.inner1d(a, a), np.sum(a*a, axis=-1),
668: (27)                  err_msg=msg)
669: (8)              msg = "type cast on one argument"
670: (8)              a = np.arange(6).reshape((2, 3))
671: (8)              b = a + 0.1
672: (8)              assert_array_almost_equal(umt.inner1d(a, b), np.sum(a*b, axis=-1),
673: (34)                  err_msg=msg)
674: (4)          def test_endian(self):
675: (8)              msg = "big endian"
676: (8)              a = np.arange(6, dtype='>i4').reshape((2, 3))
677: (8)              assert_array_equal(umt.inner1d(a, a), np.sum(a*a, axis=-1),
678: (27)                  err_msg=msg)
679: (8)              msg = "little endian"
680: (8)              a = np.arange(6, dtype='<i4').reshape((2, 3))
681: (8)              assert_array_equal(umt.inner1d(a, a), np.sum(a*a, axis=-1),
682: (27)                  err_msg=msg)
683: (8)              Ba = np.arange(1, dtype='>f8')
684: (8)              La = np.arange(1, dtype='<f8')
685: (8)              assert_equal((Ba+Ba).dtype, np.dtype('f8'))
686: (8)              assert_equal((Ba+La).dtype, np.dtype('f8'))
687: (8)              assert_equal((La+Ba).dtype, np.dtype('f8'))
688: (8)              assert_equal((La+La).dtype, np.dtype('f8'))
689: (8)              assert_equal(np.absolute(La).dtype, np.dtype('f8'))
690: (8)              assert_equal(np.absolute(Ba).dtype, np.dtype('f8'))
691: (8)              assert_equal(np.negative(La).dtype, np.dtype('f8'))
692: (8)              assert_equal(np.negative(Ba).dtype, np.dtype('f8'))
693: (4)          def test_incontiguous_array(self):
694: (8)              msg = "incontiguous memory layout of array"
695: (8)              x = np.arange(64).reshape((2, 2, 2, 2, 2, 2))
696: (8)              a = x[:, 0, :, 0, :, 0]
697: (8)              b = x[:, 1, :, 1, :, 1]
698: (8)              a[0, 0, 0] = -1
699: (8)              msg2 = "make sure it references to the original array"
700: (8)              assert_equal(x[0, 0, 0, 0, 0, 0], -1, err_msg=msg2)
701: (8)              assert_array_equal(umt.inner1d(a, b), np.sum(a*b, axis=-1),
702: (8)                  err_msg=msg)
703: (8)              x = np.arange(24).reshape(2, 3, 4)
704: (8)              a = x.T
705: (8)              b = x.T
706: (8)              a[0, 0, 0] = -1
707: (8)              assert_equal(x[0, 0, 0], -1, err_msg=msg2)
708: (4)          def test_output_argument(self):
709: (8)              msg = "output argument"
710: (8)              a = np.arange(12).reshape((2, 3, 2))

```

```

711: (8)             b = np.arange(4).reshape((2, 1, 2)) + 1
712: (8)             c = np.zeros((2, 3), dtype='int')
713: (8)             umt.inner1d(a, b, c)
714: (8)             assert_array_equal(c, np.sum(a*b, axis=-1), err_msg=msg)
715: (8)             c[:] = -1
716: (8)             umt.inner1d(a, b, out=c)
717: (8)             assert_array_equal(c, np.sum(a*b, axis=-1), err_msg=msg)
718: (8)             msg = "output argument with type cast"
719: (8)             c = np.zeros((2, 3), dtype='int16')
720: (8)             umt.inner1d(a, b, c)
721: (8)             assert_array_equal(c, np.sum(a*b, axis=-1), err_msg=msg)
722: (8)             c[:] = -1
723: (8)             umt.inner1d(a, b, out=c)
724: (8)             assert_array_equal(c, np.sum(a*b, axis=-1), err_msg=msg)
725: (8)             msg = "output argument with incontiguous layout"
726: (8)             c = np.zeros((2, 3, 4), dtype='int16')
727: (8)             umt.inner1d(a, b, c[..., 0])
728: (8)             assert_array_equal(c[..., 0], np.sum(a*b, axis=-1), err_msg=msg)
729: (8)             c[:] = -1
730: (8)             umt.inner1d(a, b, out=c[..., 0])
731: (8)             assert_array_equal(c[..., 0], np.sum(a*b, axis=-1), err_msg=msg)
732: (4)             def test_axes_argument(self):
733: (8)                 inner1d = umt.inner1d
734: (8)                 a = np.arange(27.).reshape((3, 3, 3))
735: (8)                 b = np.arange(10., 19.).reshape((3, 1, 3))
736: (8)                 c = inner1d(a, b)
737: (8)                 assert_array_equal(c, (a * b).sum(-1))
738: (8)                 c = inner1d(a, b, axes=[(-1,), (-1,), ()])
739: (8)                 assert_array_equal(c, (a * b).sum(-1))
740: (8)                 c = inner1d(a, b, axes=[-1, -1, ()])
741: (8)                 assert_array_equal(c, (a * b).sum(-1))
742: (8)                 c = inner1d(a, b, axes=[(-1,), -1, ()])
743: (8)                 assert_array_equal(c, (a * b).sum(-1))
744: (8)                 c = inner1d(a, b, axes=[-1, -1])
745: (8)                 assert_array_equal(c, (a * b).sum(-1))
746: (8)                 c = inner1d(a, b, axes=[np.int8(-1), np.array(-1, dtype=np.int32)])
747: (8)                 assert_array_equal(c, (a * b).sum(-1))
748: (8)                 c = inner1d(a, b, axes=[0, 0])
749: (8)                 assert_array_equal(c, (a * b).sum(0))
750: (8)                 c = inner1d(a, b, axes=[0, 2])
751: (8)                 assert_array_equal(c, (a.transpose(1, 2, 0) * b).sum(-1))
752: (8)                 assert_raises(TypeError, inner1d, a, b, axes=-1)
753: (8)                 assert_raises(ValueError, inner1d, a, b, axes=[-1])
754: (8)                 assert_raises(TypeError, inner1d, a, b, axes=[-1.0, -1.0])
755: (8)                 assert_raises(TypeError, inner1d, a, b, axes=[(-1.0,), -1])
756: (8)                 assert_raises(TypeError, inner1d, a, b, axes=[None, 1])
757: (8)                 assert_raises(np.AxisError, inner1d, a, b, axes=[-1, -1, -1])
758: (8)                 assert_raises(np.AxisError, inner1d, a, b, axes=[-1, -1, (-1,)])
759: (8)                 assert_raises(np.AxisError, inner1d, a, b, axes=[-1, (-2, -1), ()])
760: (8)                 assert_raises(ValueError, inner1d, a, b, axes=[0, 1])
761: (8)                 mm = umt.matrix_multiply
762: (8)                 a = np.arange(12).reshape((2, 3, 2))
763: (8)                 b = np.arange(8).reshape((2, 2, 2, 1)) + 1
764: (8)                 c = mm(a, b)
765: (8)                 assert_array_equal(c, np.matmul(a, b))
766: (8)                 c = mm(a, b, axes=[(-2, -1), (-2, -1), (-2, -1)])
767: (8)                 assert_array_equal(c, np.matmul(a, b))
768: (8)                 c = mm(a, b, axes=[(1, 2), (2, 3), (2, 3)])
769: (8)                 assert_array_equal(c, np.matmul(a, b))
770: (8)                 c = mm(a, b, axes=[(0, -1), (1, 2), (-2, -1)])
771: (8)                 assert_array_equal(c, np.matmul(a.transpose(1, 0, 2),
772: (40)                               b.transpose(0, 3, 1, 2)))
773: (8)                 c = np.empty((2, 2, 3, 1))
774: (8)                 d = mm(a, b, out=c, axes=[(1, 2), (2, 3), (2, 3)])
775: (8)                 assert_(c is d)
776: (8)                 assert_array_equal(c, np.matmul(a, b))
777: (8)                 c = np.empty((1, 2, 2, 3))
778: (8)                 d = mm(a, b, out=c, axes=[(-2, -1), (-2, -1), (3, 0)])
779: (8)                 assert_(c is d)

```

```

780: (8) assert_array_equal(c, np.matmul(a, b).transpose(3, 0, 1, 2))
781: (8) assert_raises(TypeError, mm, a, b, axis=1)
782: (8) assert_raises(TypeError, mm, a, b, axes=1)
783: (8) assert_raises(TypeError, mm, a, b, axes=(-2, -1), (-2, -1), (-2,
-1)))
784: (8) assert_raises(ValueError, mm, a, b, axes=[])
785: (8) assert_raises(ValueError, mm, a, b, axes=[(-2, -1)])
786: (8) assert_raises(TypeError, mm, a, b, axes=[None, None, None])
787: (8) assert_raises(TypeError,
    mm, a, b, axes=[[-2, -1], [-2, -1], [-2, -1]])
788: (22) assert_raises(TypeError,
    mm, a, b, axes=[(-2, -1), (-2, -1), [-2, -1]])
789: (8) assert_raises(TypeError,
    mm, a, b, axes=[(-2, -1), (-2, -1), None])
790: (22) assert_raises(np.AxisError, mm, a, b, axes=[-1, -1, -1])
791: (8) assert_raises(np.AxisError, mm, a, b, axes=[(-2, -1), (-2, -1), -1])
792: (8) assert_raises(ValueError, mm, a, b, axes=[(-2, -1), (-2, -1), -2])
793: (8) assert_raises(ValueError, mm, a, b, axes=[(-2, -1), (-2, -1), -2])
794: (8) assert_raises(ValueError, mm, a, b, axes=[(-2, -1), (-2, -1), (-2,
-2)])
795: (8) z = np.zeros((2, 2))
796: (8) assert_raises(ValueError, mm, z, z[0])
797: (8) assert_raises(ValueError, mm, z, z, out=z[:, 0])
798: (8) assert_raises(ValueError, mm, z[1], z, axes=[0, 1])
799: (8) assert_raises(ValueError, mm, z, z, out=z[0], axes=[0, 1])
800: (8) assert_raises(TypeError, np.add, 1., 1., axes=[0])
801: (8) assert_raises(TypeError, mm, z, z, axes=[0, 1], parrot=True)
802: (4) def test_axis_argument(self):
803: (8)     inner1d = umt.inner1d
804: (8)     a = np.arange(27.).reshape((3, 3, 3))
805: (8)     b = np.arange(10., 19.).reshape((3, 1, 3))
806: (8)     c = inner1d(a, b)
807: (8)     assert_array_equal(c, (a * b).sum(-1))
808: (8)     c = inner1d(a, b, axis=-1)
809: (8)     assert_array_equal(c, (a * b).sum(-1))
810: (8)     out = np.zeros_like(c)
811: (8)     d = inner1d(a, b, axis=-1, out=out)
812: (8)     assert_(d is out)
813: (8)     assert_array_equal(d, c)
814: (8)     c = inner1d(a, b, axis=0)
815: (8)     assert_array_equal(c, (a * b).sum(0))
816: (8)     a = np.arange(6).reshape((2, 3))
817: (8)     b = np.arange(10, 16).reshape((2, 3))
818: (8)     w = np.arange(20, 26).reshape((2, 3))
819: (8)     assert_array_equal(umt.innerwt(a, b, w, axis=0),
    np.sum(a * b * w, axis=0))
820: (27) assert_array_equal(umt.cumsum(a, axis=0), np.cumsum(a, axis=0))
821: (8) assert_array_equal(umt.cumsum(a, axis=-1), np.cumsum(a, axis=-1))
822: (8) out = np.empty_like(a)
823: (8) b = umt.cumsum(a, out=out, axis=0)
824: (8) assert_(out is b)
825: (8) assert_array_equal(b, np.cumsum(a, axis=0))
826: (8) b = umt.cumsum(a, out=out, axis=1)
827: (8) assert_(out is b)
828: (8) assert_array_equal(b, np.cumsum(a, axis=-1))
829: (8) assert_raises(TypeError, inner1d, a, b, axis=0, axes=[0, 0])
830: (8) assert_raises(TypeError, inner1d, a, b, axis=[0])
831: (8) mm = umt.matrix_multiply
832: (8) assert_raises(TypeError, mm, a, b, axis=1)
833: (8) out = np.empty((1, 2, 3), dtype=a.dtype)
834: (8) assert_raises(ValueError, umt.cumsum, a, out=out, axis=0)
835: (8) assert_raises(TypeError, np.add, 1., 1., axis=0)
836: (8) def test_kepdims_argument(self):
837: (4)     inner1d = umt.inner1d
838: (8)     a = np.arange(27.).reshape((3, 3, 3))
839: (8)     b = np.arange(10., 19.).reshape((3, 1, 3))
840: (8)     c = inner1d(a, b)
841: (8)     assert_array_equal(c, (a * b).sum(-1))
842: (8)     c = inner1d(a, b, keepdims=False)
843: (8)     assert_array_equal(c, (a * b).sum(-1))
844: (8)     c = inner1d(a, b, keepdims=True)
845: (8)     assert_array_equal(c, (a * b).sum(-1, keepdims=True))
846: (8)

```

```

847: (8)             out = np.zeros_like(c)
848: (8)             d = inner1d(a, b, keepdims=True, out=out)
849: (8)             assert_(d is out)
850: (8)             assert_array_equal(d, c)
851: (8)             c = inner1d(a, b, axis=-1, keepdims=False)
852: (8)             assert_array_equal(c, (a * b).sum(-1, keepdims=False))
853: (8)             c = inner1d(a, b, axis=-1, keepdims=True)
854: (8)             assert_array_equal(c, (a * b).sum(-1, keepdims=True))
855: (8)             c = inner1d(a, b, axis=0, keepdims=False)
856: (8)             assert_array_equal(c, (a * b).sum(0, keepdims=False))
857: (8)             c = inner1d(a, b, axis=0, keepdims=True)
858: (8)             assert_array_equal(c, (a * b).sum(0, keepdims=True))
859: (8)             c = inner1d(a, b, axes=[(-1), (-1, ), ()], keepdims=False)
860: (8)             assert_array_equal(c, (a * b).sum(-1))
861: (8)             c = inner1d(a, b, axes=[(-1), (-1, ), (-1, )], keepdims=True)
862: (8)             assert_array_equal(c, (a * b).sum(-1, keepdims=True))
863: (8)             c = inner1d(a, b, axes=[0, 0], keepdims=False)
864: (8)             assert_array_equal(c, (a * b).sum(0))
865: (8)             c = inner1d(a, b, axes=[0, 0, 0], keepdims=True)
866: (8)             assert_array_equal(c, (a * b).sum(0, keepdims=True))
867: (8)             c = inner1d(a, b, axes=[0, 2], keepdims=False)
868: (8)             assert_array_equal(c, (a.transpose(1, 2, 0) * b).sum(-1))
869: (8)             c = inner1d(a, b, axes=[0, 2], keepdims=True)
870: (8)             assert_array_equal(c, (a.transpose(1, 2, 0) * b).sum(-1,
871: (61)                                     keepdims=True))
872: (8)             c = inner1d(a, b, axes=[0, 2, 2], keepdims=True)
873: (8)             assert_array_equal(c, (a.transpose(1, 2, 0) * b).sum(-1,
874: (61)                                     keepdims=True))
875: (8)             c = inner1d(a, b, axes=[0, 2, 0], keepdims=True)
876: (8)             assert_array_equal(c, (a * b.transpose(2, 0, 1)).sum(0,
877: (8)                                     keepdims=True))
878: (8)             c = inner1d(a, b, axes=[0, 2, 1], keepdims=True)
879: (27)             assert_array_equal(c, (a.transpose(1, 0, 2) * b.transpose(0, 2, 1))
880: (8)                                     .sum(1, keepdims=True))
881: (8)             a = np.eye(3) * np.arange(4.)[:, np.newaxis, np.newaxis]
882: (8)             expected = uml.det(a)
883: (8)             c = uml.det(a, keepdims=False)
884: (8)             assert_array_equal(c, expected)
885: (8)             c = uml.det(a, keepdims=True)
886: (8)             assert_array_equal(c, expected[:, np.newaxis, np.newaxis])
887: (8)             a = np.eye(3) * np.arange(4.)[:, np.newaxis, np.newaxis]
888: (8)             expected_s, expected_l = uml.slogdet(a)
889: (8)             cs, cl = uml.slogdet(a, keepdims=False)
890: (8)             assert_array_equal(cs, expected_s)
891: (8)             assert_array_equal(cl, expected_l)
892: (8)             cs, cl = uml.slogdet(a, keepdims=True)
893: (8)             assert_array_equal(cs, expected_s[:, np.newaxis, np.newaxis])
894: (8)             assert_array_equal(cl, expected_l[:, np.newaxis, np.newaxis])
895: (8)             a = np.arange(6).reshape((2, 3))
896: (8)             b = np.arange(10, 16).reshape((2, 3))
897: (8)             w = np.arange(20, 26).reshape((2, 3))
898: (27)             assert_array_equal(umt.innerwt(a, b, w, keepdims=True),
899: (8)                                     np.sum(a * b * w, axis=-1, keepdims=True))
900: (27)             assert_array_equal(umt.innerwt(a, b, w, axis=0, keepdims=True),
901: (8)                                     np.sum(a * b * w, axis=0, keepdims=True))
902: (8)             assert_raises(TypeError, inner1d, a, b, keepdims='true')
903: (8)             mm = umt.matrix_multiply
904: (8)             assert_raises(TypeError, mm, a, b, keepdims=True)
905: (8)             assert_raises(TypeError, mm, a, b, keepdims=False)
906: (4)              assert_raises(TypeError, np.add, 1., 1., keepdims=False)
907: (8)              def test_innerwt(self):
908: (8)                  a = np.arange(6).reshape((2, 3))
909: (8)                  b = np.arange(10, 16).reshape((2, 3))
910: (8)                  w = np.arange(20, 26).reshape((2, 3))
911: (8)                  assert_array_equal(umt.innerwt(a, b, w), np.sum(a*b*w, axis=-1))
912: (8)                  a = np.arange(100, 124).reshape((2, 3, 4))
913: (8)                  b = np.arange(200, 224).reshape((2, 3, 4))
914: (8)                  w = np.arange(300, 324).reshape((2, 3, 4))
915: (8)                  assert_array_equal(umt.innerwt(a, b, w), np.sum(a*b*w, axis=-1))

```

```

915: (4)
916: (8)
917: (8)
918: (8)
919: (8)
920: (8)
921: (4)
922: (8)
923: (8)
924: (8)
925: (8)
926: (8)
927: (8)
928: (8)
929: (8)
930: (8)
931: (8)
932: (8)
933: (4)
934: (8)
935: (8)
936: (8)
937: (8)
938: (8)
939: (8)
940: (8)
941: (8)
942: (8)
943: (8)
944: (8)
945: (8)
946: (8)
947: (8)
948: (8)
949: (8)
950: (8)
951: (8)
952: (8)
953: (8)
954: (8)
955: (8)
956: (8)
957: (8)
958: (8)
959: (8)
960: (8)
961: (8)
962: (8)
963: (8)
964: (8)
965: (8)
966: (8)
967: (8)
968: (8)
969: (8)
970: (8)
971: (8)
972: (8)
973: (8)
974: (8)
975: (8)
976: (8)
977: (8)
978: (22)
979: (8)
980: (8)
981: (4)
982: (8)

        def test_innerwt_empty(self):
            """Test generalized ufunc with zero-sized operands"""
            a = np.array([], dtype='f8')
            b = np.array([], dtype='f8')
            w = np.array([], dtype='f8')
            assert_array_equal(umt.innerwt(a, b, w), np.sum(a*b*w, axis=-1))

        def test_cross1d(self):
            """Test with fixed-sized signature."""
            a = np.eye(3)
            assert_array_equal(umt.cross1d(a, a), np.zeros((3, 3)))
            out = np.zeros((3, 3))
            result = umt.cross1d(a[0], a, out)
            assert_(result is out)
            assert_array_equal(result, np.vstack((np.zeros(3), a[2], -a[1])))
            assert_raises(ValueError, umt.cross1d, np.eye(4), np.eye(4))
            assert_raises(ValueError, umt.cross1d, a, np.arange(4.))
            assert_raises(ValueError, umt.cross1d, a, np.arange(3.), np.zeros((3, 4)))
            assert_raises(ValueError, umt.cross1d, a, np.arange(3.), np.zeros(3))

        def test_can_ignore_signature(self):
            mat = np.arange(12).reshape((2, 3, 2))
            single_vec = np.arange(2)
            col_vec = single_vec[:, np.newaxis]
            col_vec_array = np.arange(8).reshape((2, 2, 2, 1)) + 1
            mm_col_vec = umt.matrix_multiply(mat, col_vec)
            matmul_col_vec = umt.matmul(mat, col_vec)
            assert_array_equal(matmul_col_vec, mm_col_vec)
            assert_raises(ValueError, umt.matrix_multiply, mat, single_vec)
            matmul_col = umt.matmul(mat, single_vec)
            assert_array_equal(matmul_col, mm_col_vec.squeeze())
            mm_col_vec = umt.matrix_multiply(mat, col_vec_array)
            matmul_col_vec = umt.matmul(mat, col_vec_array)
            assert_array_equal(matmul_col_vec, mm_col_vec)
            single_vec = np.arange(3)
            row_vec = single_vec[np.newaxis, :]
            row_vec_array = np.arange(24).reshape((4, 2, 1, 1, 3)) + 1
            mm_row_vec = umt.matrix_multiply(row_vec, mat)
            matmul_row_vec = umt.matmul(row_vec, mat)
            assert_array_equal(matmul_row_vec, mm_row_vec)
            assert_raises(ValueError, umt.matrix_multiply, single_vec, mat)
            matmul_row = umt.matmul(single_vec, mat)
            assert_array_equal(matmul_row, mm_row_vec.squeeze())
            mm_row_vec = umt.matrix_multiply(row_vec_array, mat)
            matmul_row_vec = umt.matmul(row_vec_array, mat)
            assert_array_equal(matmul_row_vec, mm_row_vec)
            col_vec = row_vec.T
            col_vec_array = row_vec_array.swapaxes(-2, -1)
            mm_row_col_vec = umt.matrix_multiply(row_vec, col_vec)
            matmul_row_col_vec = umt.matmul(row_vec, col_vec)
            assert_array_equal(matmul_row_col_vec, mm_row_col_vec)
            assert_raises(ValueError, umt.matrix_multiply, single_vec, single_vec)
            matmul_row_col = umt.matmul(single_vec, single_vec)
            assert_array_equal(matmul_row_col, mm_row_col_vec.squeeze())
            mm_row_col_array = umt.matrix_multiply(row_vec_array, col_vec_array)
            matmul_row_col_array = umt.matmul(row_vec_array, col_vec_array)
            assert_array_equal(matmul_row_col_array, mm_row_col_array)
            out = np.zeros_like(mm_row_col_array)
            out = umt.matrix_multiply(row_vec_array, col_vec_array, out=out)
            assert_array_equal(out, mm_row_col_array)
            out[:] = 0
            out = umt.matmul(row_vec_array, col_vec_array, out=out)
            assert_array_equal(out, mm_row_col_array)
            out = np.zeros_like(mm_row_col_vec)
            assert_raises(ValueError, umt.matrix_multiply, single_vec, single_vec,
                         out)
            out = umt.matmul(single_vec, single_vec, out)
            assert_array_equal(out, mm_row_col_vec.squeeze())

        def test_matrix_multiply(self):
            self.compare_matrix_multiply_results(np.int64)

```

```

983: (8)             self.compare_matrix_multiply_results(np.double)
984: (4)             def test_matrix_multiply_umath_empty(self):
985: (8)                 res = umt.matrix_multiply(np.ones((0, 10)), np.ones((10, 0)))
986: (8)                 assert_array_equal(res, np.zeros((0, 0)))
987: (8)                 res = umt.matrix_multiply(np.ones((10, 0)), np.ones((0, 10)))
988: (8)                 assert_array_equal(res, np.zeros((10, 10)))
989: (4)             def compare_matrix_multiply_results(self, tp):
990: (8)                 d1 = np.array(np.random.rand(2, 3, 4), dtype=tp)
991: (8)                 d2 = np.array(np.random.rand(2, 3, 4), dtype=tp)
992: (8)                 msg = "matrix multiply on type %s" % d1.dtype.name
993: (8)             def permute_n(n):
994: (12)                 if n == 1:
995: (16)                     return ([0],)
996: (12)                 ret = ()
997: (12)                 base = permute_n(n-1)
998: (12)                 for perm in base:
999: (16)                     for i in range(n):
1000: (20)                         new = perm + [n-1]
1001: (20)                         new[n-1] = new[i]
1002: (20)                         new[i] = n-1
1003: (20)                         ret += (new,)
1004: (12)                 return ret
1005: (8)             def slice_n(n):
1006: (12)                 if n == 0:
1007: (16)                     return (((),))
1008: (12)                 ret = ()
1009: (12)                 base = slice_n(n-1)
1010: (12)                 for sl in base:
1011: (16)                     ret += (sl+(slice(None),),)
1012: (16)                     ret += (sl+(slice(0, 1),),)
1013: (12)                 return ret
1014: (8)             def broadcastable(s1, s2):
1015: (12)                 return s1 == s2 or s1 == 1 or s2 == 1
1016: (8)             permute_3 = permute_n(3)
1017: (8)             slice_3 = slice_n(3) + ((slice(None, None, -1),)*3,)
1018: (8)             ref = True
1019: (8)             for p1 in permute_3:
1020: (12)                 for p2 in permute_3:
1021: (16)                     for s1 in slice_3:
1022: (20)                         for s2 in slice_3:
1023: (24)                             a1 = d1.transpose(p1)[s1]
1024: (24)                             a2 = d2.transpose(p2)[s2]
1025: (24)                             ref = ref and a1.base is not None
1026: (24)                             ref = ref and a2.base is not None
1027: (24)                             if (a1.shape[-1] == a2.shape[-2] and
1028: (32)                                 broadcastable(a1.shape[0], a2.shape[0])):
1029: (28)                                 assert_array_almost_equal(
1030: (32)                                     umt.matrix_multiply(a1, a2),
1031: (32)                                     np.sum(a2[..., np.newaxis].swapaxes(-3, -1) *
1032: (39)   a1[..., np.newaxis, :], axis=-1),
1033: (32)                                     err_msg=msg + ' %s %s' % (str(a1.shape),
1034: (58)   str(a2.shape)))
1035: (8)                                 assert_equal(ref, True, err_msg="reference check")
1036: (4)             def test_euclidean_pdist(self):
1037: (8)                 a = np.arange(12, dtype=float).reshape(4, 3)
1038: (8)                 out = np.empty((a.shape[0] * (a.shape[0] - 1) // 2,), dtype=a.dtype)
1039: (8)                 umt.euclidean_pdist(a, out)
1040: (8)                 b = np.sqrt(np.sum((a[:, None] - a)**2, axis=-1))
1041: (8)                 b = b[~np.tri(a.shape[0], dtype=bool)]
1042: (8)                 assert_almost_equal(out, b)
1043: (8)                 assert_raises(ValueError, umt.euclidean_pdist, a)
1044: (4)             def test_cumsum(self):
1045: (8)                 a = np.arange(10)
1046: (8)                 result = umt.cumsum(a)
1047: (8)                 assert_array_equal(result, a.cumsum())
1048: (4)             def test_object_logical(self):
1049: (8)                 a = np.array([3, None, True, False, "test", ""], dtype=object)
1050: (8)                 assert_equal(np.logical_or(a, None),
1051: (24)                               np.array([x or None for x in a], dtype=object))

```

```

1052: (8) assert_equal(np.logical_or(a, True),
1053: (24)         np.array([x or True for x in a], dtype=object))
1054: (8) assert_equal(np.logical_or(a, 12),
1055: (24)         np.array([x or 12 for x in a], dtype=object))
1056: (8) assert_equal(np.logical_or(a, "blah"),
1057: (24)         np.array([x or "blah" for x in a], dtype=object))
1058: (8) assert_equal(np.logical_and(a, None),
1059: (24)         np.array([x and None for x in a], dtype=object))
1060: (8) assert_equal(np.logical_and(a, True),
1061: (24)         np.array([x and True for x in a], dtype=object))
1062: (8) assert_equal(np.logical_and(a, 12),
1063: (24)         np.array([x and 12 for x in a], dtype=object))
1064: (8) assert_equal(np.logical_and(a, "blah"),
1065: (24)         np.array([x and "blah" for x in a], dtype=object))
1066: (8) assert_equal(np.logical_not(a),
1067: (24)         np.array([not x for x in a], dtype=object))
1068: (8) assert_equal(np.logical_or.reduce(a), 3)
1069: (8) assert_equal(np.logical_and.reduce(a), None)
1070: (4) def test_object_comparison(self):
1071: (8)     class HasComparisons:
1072: (12)         def __eq__(self, other):
1073: (16)             return '=='
```

behavior is a cast

```

1074: (8) arr0d = np.array(HasComparisons())
1075: (8) assert_equal(arr0d == arr0d, True)
1076: (8) assert_equal(np.equal(arr0d, arr0d), True) # normal behavior is a
cast
1077: (8) arr1d = np.array([HasComparisons()])
1078: (8) assert_equal(arr1d == arr1d, np.array([True]))
1079: (8) assert_equal(np.equal(arr1d, arr1d), np.array([True])) # normal
```

```

1080: (8) assert_equal(np.equal(arr1d, arr1d, dtype=object), np.array(['==']))
1081: (4) def test_object_array_reduction(self):
1082: (8)     a = np.array(['a', 'b', 'c'], dtype=object)
1083: (8)     assert_equal(np.sum(a), 'abc')
1084: (8)     assert_equal(np.max(a), 'c')
1085: (8)     assert_equal(np.min(a), 'a')
1086: (8)     a = np.array([True, False, True], dtype=object)
1087: (8)     assert_equal(np.sum(a), 2)
1088: (8)     assert_equal(np.prod(a), 0)
1089: (8)     assert_equal(np.any(a), True)
1090: (8)     assert_equal(np.all(a), False)
1091: (8)     assert_equal(np.max(a), True)
1092: (8)     assert_equal(np.min(a), False)
1093: (8)     assert_equal(np.array([[1]]), dtype=object).sum(), 1)
1094: (8)     assert_equal(np.array([[[1, 2]]], dtype=object).sum((0, 1)), [1, 2])
1095: (8)     assert_equal(np.array([1], dtype=object).sum(initial=1), 2)
1096: (8)     assert_equal(np.array([[1, 2, 3]], dtype=object)
1097: (21)         .sum(initial=[0], where=[False, True]), [0, 2, 3])
1098: (4) def test_object_array_accumulate_inplace(self):
1099: (8)     arr = np.ones(4, dtype=object)
1100: (8)     arr[:] = [[1] for i in range(4)]
1101: (8)     np.add.accumulate(arr, out=arr)
1102: (8)     np.add.accumulate(arr, out=arr)
1103: (8)     assert_array_equal(arr,
1104: (27)         np.array([[1]*i for i in [1, 3, 6, 10]]),
1105: (26)         )
1106: (8)     arr = np.ones((2, 4), dtype=object)
1107: (8)     arr[0, :] = [[2] for i in range(4)]
1108: (8)     np.add.accumulate(arr, out=arr, axis=-1)
1109: (8)     np.add.accumulate(arr, out=arr, axis=-1)
1110: (8)     assert_array_equal(arr[0, :],
1111: (27)         np.array([[2]*i for i in [1, 3, 6, 10]]),
1112: (26)         )
1113: (4) def test_object_array_accumulate_failure(self):
1114: (8)     res = np.add.accumulate(np.array([1, 0, 2], dtype=object))
1115: (8)     assert_array_equal(res, np.array([1, 1, 3], dtype=object))
1116: (8)     with pytest.raises(TypeError):
```

```

1117: (12)           np.add.accumulate([1, None, 2])
1118: (4)            def test_object_array_reduceat_inplace(self):
1119: (8)              arr = np.empty(4, dtype=object)
1120: (8)              arr[:] = [[1] for i in range(4)]
1121: (8)              out = np.empty(4, dtype=object)
1122: (8)              out[:] = [[1] for i in range(4)]
1123: (8)              np.add.reduceat(arr, np.arange(4), out=arr)
1124: (8)              np.add.reduceat(arr, np.arange(4), out=arr)
1125: (8)              assert_array_equal(arr, out)
1126: (8)              arr = np.ones((2, 4), dtype=object)
1127: (8)              arr[0, :] = [[2] for i in range(4)]
1128: (8)              out = np.ones((2, 4), dtype=object)
1129: (8)              out[0, :] = [[2] for i in range(4)]
1130: (8)              np.add.reduceat(arr, np.arange(4), out=arr, axis=-1)
1131: (8)              np.add.reduceat(arr, np.arange(4), out=arr, axis=-1)
1132: (8)              assert_array_equal(arr, out)
1133: (4)            def test_object_array_reduceat_failure(self):
1134: (8)              res = np.add.reduceat(np.array([1, None, 2], dtype=object), [1, 2])
1135: (8)              assert_array_equal(res, np.array([None, 2], dtype=object))
1136: (8)              with pytest.raises(TypeError):
1137: (12)                  np.add.reduceat([1, None, 2], [0, 2])
1138: (4)            def test_zerosize_reduction(self):
1139: (8)              for a in [[], np.array([], dtype=object)]:
1140: (12)                  assert_equal(np.sum(a), 0)
1141: (12)                  assert_equal(np.prod(a), 1)
1142: (12)                  assert_equal(np.any(a), False)
1143: (12)                  assert_equal(np.all(a), True)
1144: (12)                  assert_raises(ValueError, np.max, a)
1145: (12)                  assert_raises(ValueError, np.min, a)
1146: (4)            def test_axis_out_of_bounds(self):
1147: (8)              a = np.array([False, False])
1148: (8)              assert_raises(np.AxisError, a.all, axis=1)
1149: (8)              a = np.array([False, False])
1150: (8)              assert_raises(np.AxisError, a.all, axis=-2)
1151: (8)              a = np.array([False, False])
1152: (8)              assert_raises(np.AxisError, a.any, axis=1)
1153: (8)              a = np.array([False, False])
1154: (8)              assert_raises(np.AxisError, a.any, axis=-2)
1155: (4)            def test_scalar_reduction(self):
1156: (8)              assert_equal(np.sum(3, axis=0), 3)
1157: (8)              assert_equal(np.prod(3.5, axis=0), 3.5)
1158: (8)              assert_equal(np.any(True, axis=0), True)
1159: (8)              assert_equal(np.all(False, axis=0), False)
1160: (8)              assert_equal(np.max(3, axis=0), 3)
1161: (8)              assert_equal(np.min(2.5, axis=0), 2.5)
1162: (8)              assert_equal(np.power.reduce(3), 3)
1163: (8)              assert_(type(np.prod(np.float32(2.5), axis=0)) is np.float32)
1164: (8)              assert_(type(np.sum(np.float32(2.5), axis=0)) is np.float32)
1165: (8)              assert_(type(np.max(np.float32(2.5), axis=0)) is np.float32)
1166: (8)              assert_(type(np.min(np.float32(2.5), axis=0)) is np.float32)
1167: (8)              assert_(type(np.any(0, axis=0)) is np.bool_)
1168: (8)              class MyArray(np.ndarray):
1169: (12)                  pass
1170: (8)                  a = np.array(1).view(MyArray)
1171: (8)                  assert_(type(np.any(a)) is MyArray)
1172: (4)            def test_casting_out_param(self):
1173: (8)              a = np.ones((200, 100), np.int64)
1174: (8)              b = np.ones((200, 100), np.int64)
1175: (8)              c = np.ones((200, 100), np.float64)
1176: (8)              np.add(a, b, out=c)
1177: (8)              assert_equal(c, 2)
1178: (8)              a = np.zeros(65536)
1179: (8)              b = np.zeros(65536, dtype=np.float32)
1180: (8)              np.subtract(a, 0, out=b)
1181: (8)              assert_equal(b, 0)
1182: (4)            def test_where_param(self):
1183: (8)              a = np.arange(7)
1184: (8)              b = np.ones(7)
1185: (8)              c = np.zeros(7)

```

```

1186: (8) np.add(a, b, out=c, where=(a % 2 == 1))
1187: (8) assert_equal(c, [0, 2, 0, 4, 0, 6, 0])
1188: (8) a = np.arange(4).reshape(2, 2) + 2
1189: (8) np.power(a, [2, 3], out=a, where=[[0, 1], [1, 0]])
1190: (8) assert_equal(a, [[2, 27], [16, 5]])
1191: (8) np.subtract(a, 2, out=a, where=[True, False])
1192: (8) assert_equal(a, [[0, 27], [14, 5]])
1193: (4) def test_where_param_buffer_output(self):
1194: (8) a = np.ones(10, np.int64)
1195: (8) b = np.ones(10, np.int64)
1196: (8) c = 1.5 * np.ones(10, np.float64)
1197: (8) np.add(a, b, out=c, where=[1, 0, 0, 1, 0, 0, 1, 1, 1, 0])
1198: (8) assert_equal(c, [2, 1.5, 1.5, 2, 1.5, 1.5, 2, 2, 2, 1.5])
1199: (4) def test_where_param_alloc(self):
1200: (8) a = np.array([1], dtype=np.int64)
1201: (8) m = np.array([True], dtype=bool)
1202: (8) assert_equal(np.sqrt(a, where=m), [1])
1203: (8) a = np.array([1], dtype=np.float64)
1204: (8) m = np.array([True], dtype=bool)
1205: (8) assert_equal(np.sqrt(a, where=m), [1])
1206: (4) def test_where_with_broadcasting(self):
1207: (8) a = np.random.random((5000, 4))
1208: (8) b = np.random.random((5000, 1))
1209: (8) where = a > 0.3
1210: (8) out = np.full_like(a, 0)
1211: (8) np.less(a, b, where=where, out=out)
1212: (8) b_where = np.broadcast_to(b, a.shape)[where]
1213: (8) assert_array_equal((a[where] < b_where), out[where].astype(bool))
1214: (8) assert not out[~where].any() # outside mask, out remains all 0
1215: (4) def check_identityless_reduction(self, a):
1216: (8) a[...] = 1
1217: (8) a[1, 0, 0] = 0
1218: (8) assert_equal(np.minimum.reduce(a, axis=None), 0)
1219: (8) assert_equal(np.minimum.reduce(a, axis=(0, 1)), [0, 1, 1, 1])
1220: (8) assert_equal(np.minimum.reduce(a, axis=(0, 2)), [0, 1, 1])
1221: (8) assert_equal(np.minimum.reduce(a, axis=(1, 2)), [1, 0])
1222: (8) assert_equal(np.minimum.reduce(a, axis=0),
1223: (36) [[0, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1,
1]])]
1224: (8) assert_equal(np.minimum.reduce(a, axis=1),
1225: (36) [[1, 1, 1, 1], [0, 1, 1, 1]])
1226: (8) assert_equal(np.minimum.reduce(a, axis=2),
1227: (36) [[1, 1, 1], [0, 1, 1]])
1228: (8) assert_equal(np.minimum.reduce(a, axis=()), a)
1229: (8) a[...] = 1
1230: (8) a[0, 1, 0] = 0
1231: (8) assert_equal(np.minimum.reduce(a, axis=None), 0)
1232: (8) assert_equal(np.minimum.reduce(a, axis=(0, 1)), [0, 1, 1, 1])
1233: (8) assert_equal(np.minimum.reduce(a, axis=(0, 2)), [1, 0, 1])
1234: (8) assert_equal(np.minimum.reduce(a, axis=(1, 2)), [0, 1])
1235: (8) assert_equal(np.minimum.reduce(a, axis=0),
1236: (36) [[1, 1, 1, 1], [0, 1, 1, 1], [1, 1, 1,
1]])]
1237: (8) assert_equal(np.minimum.reduce(a, axis=1),
1238: (36) [[0, 1, 1, 1], [1, 1, 1, 1]])
1239: (8) assert_equal(np.minimum.reduce(a, axis=2),
1240: (36) [[1, 0, 1], [1, 1, 1]])
1241: (8) assert_equal(np.minimum.reduce(a, axis=()), a)
1242: (8) a[...] = 1
1243: (8) a[0, 0, 1] = 0
1244: (8) assert_equal(np.minimum.reduce(a, axis=None), 0)
1245: (8) assert_equal(np.minimum.reduce(a, axis=(0, 1)), [1, 0, 1, 1])
1246: (8) assert_equal(np.minimum.reduce(a, axis=(0, 2)), [0, 1, 1])
1247: (8) assert_equal(np.minimum.reduce(a, axis=(1, 2)), [0, 1])
1248: (8) assert_equal(np.minimum.reduce(a, axis=0),
1249: (36) [[1, 0, 1, 1], [1, 1, 1, 1], [1, 1, 1,
1]])]
1250: (8) assert_equal(np.minimum.reduce(a, axis=1),
1251: (36) [[1, 0, 1, 1], [1, 1, 1, 1]])

```

```

1252: (8) assert_equal(np.minimum.reduce(a, axis=2),
1253: (36) [0, 1, 1], [1, 1, 1]])
1254: (8) assert_equal(np.minimum.reduce(a, axis=()), a)
1255: (4) @requires_memory(6 * 1024**3)
1256: (4) @pytest.mark.skipif(sys.maxsize < 2**32,
1257: (12) reason="test array too large for 32bit platform")
1258: (4) def test_identityless_reduction_huge_array(self):
1259: (8) arr = np.zeros((2, 2**31), 'uint8')
1260: (8) arr[:, 0] = [1, 3]
1261: (8) arr[:, -1] = [4, 1]
1262: (8) res = np.maximum.reduce(arr, axis=0)
1263: (8) del arr
1264: (8) assert res[0] == 3
1265: (8) assert res[-1] == 4
1266: (4) def test_identityless_reduction_corder(self):
1267: (8) a = np.empty((2, 3, 4), order='C')
1268: (8) self.check_identityless_reduction(a)
1269: (4) def test_identityless_reduction_forder(self):
1270: (8) a = np.empty((2, 3, 4), order='F')
1271: (8) self.check_identityless_reduction(a)
1272: (4) def test_identityless_reduction_otherorder(self):
1273: (8) a = np.empty((2, 4, 3), order='C').swapaxes(1, 2)
1274: (8) self.check_identityless_reduction(a)
1275: (4) def test_identityless_reduction_noncontig(self):
1276: (8) a = np.empty((3, 5, 4), order='C').swapaxes(1, 2)
1277: (8) a = a[1:, 1:, 1:]
1278: (8) self.check_identityless_reduction(a)
1279: (4) def test_identityless_reduction_noncontig_unaligned(self):
1280: (8) a = np.empty((3*4*5*8 + 1,), dtype='i1')
1281: (8) a = a[1:].view(dtype='f8')
1282: (8) a.shape = (3, 4, 5)
1283: (8) a = a[1:, 1:, 1:]
1284: (8) self.check_identityless_reduction(a)
1285: (4) def test_reduce_identity_depends_on_loop(self):
1286: (8) """
1287: (8) The type of the result should always depend on the selected loop, not
1288: (8) necessarily the output (only relevant for object arrays).
1289: (8)
1290: (8) assert type(np.add.reduce([], dtype=object)) is int
1291: (8) out = np.array(None, dtype=object)
1292: (8) np.add.reduce([], out=out, dtype=np.float64)
1293: (8) assert type(out[()]) is float
1294: (4) def test_initial_reduction(self):
1295: (8) assert_equal(np.maximum.reduce([], initial=0), 0)
1296: (8) assert_equal(np.minimum.reduce([], initial=np.inf), np.inf)
1297: (8) assert_equal(np.maximum.reduce([], initial=-np.inf), -np.inf)
1298: (8) assert_equal(np.minimum.reduce([5], initial=4), 4)
1299: (8) assert_equal(np.maximum.reduce([4], initial=5), 5)
1300: (8) assert_equal(np.maximum.reduce([5], initial=4), 5)
1301: (8) assert_equal(np.minimum.reduce([4], initial=5), 4)
1302: (8) assert_raises(ValueError, np.minimum.reduce, [], initial=None)
1303: (8) assert_raises(ValueError, np.add.reduce, [], initial=None)
1304: (8) with pytest.raises(ValueError):
1305: (12)     np.add.reduce([], initial=None, dtype=object)
1306: (8) assert_equal(np.add.reduce([], initial=np._NoValue), 0)
1307: (8) a = np.array([10], dtype=object)
1308: (8) res = np.add.reduce(a, initial=5)
1309: (8) assert_equal(res, 15)
1310: (4) def test_empty_reduction_and_idenity(self):
1311: (8) arr = np.zeros((0, 5))
1312: (8) assert np.true_divide.reduce(arr, axis=1).shape == (0,)
1313: (8) with pytest.raises(ValueError):
1314: (12)     np.true_divide.reduce(arr, axis=0)
1315: (8) arr = np.zeros((0, 0, 5))
1316: (8) with pytest.raises(ValueError):
1317: (12)     np.true_divide.reduce(arr, axis=1)
1318: (8) res = np.true_divide.reduce(arr, axis=1, initial=1)
1319: (8) assert_array_equal(res, np.ones((0, 5)))
1320: (4) @pytest.mark.parametrize('axis', (0, 1, None))

```

```

1321: (4) @pytest.mark.parametrize('where', (np.array([False, True, True]),
1322: (39) np.array([[True], [False], [True]]),
1323: (39) np.array([[True, False, False],
1324: (49) [False, True, False],
1325: (49) [False, True, True]])))
1326: (4) def test_reduction_with_where(self, axis, where):
1327: (8)     a = np.arange(9.).reshape(3, 3)
1328: (8)     a_copy = a.copy()
1329: (8)     a_check = np.zeros_like(a)
1330: (8)     np.positive(a, out=a_check, where=where)
1331: (8)     res = np.add.reduce(a, axis=axis, where=where)
1332: (8)     check = a_check.sum(axis)
1333: (8)     assert_equal(res, check)
1334: (8)     assert_array_equal(a, a_copy)
1335: (4) @pytest.mark.parametrize(('axis', 'where'),
1336: (29) ((0, np.array([True, False, True])),
1337: (30) (1, [True, True, False]),
1338: (30) (None, True)))
1339: (4) @pytest.mark.parametrize('initial', (-np.inf, 5.))
1340: (4) def test_reduction_with_where_and_initial(self, axis, where, initial):
1341: (8)     a = np.arange(9.).reshape(3, 3)
1342: (8)     a_copy = a.copy()
1343: (8)     a_check = np.full(a.shape, -np.inf)
1344: (8)     np.positive(a, out=a_check, where=where)
1345: (8)     res = np.maximum.reduce(a, axis=axis, where=where, initial=initial)
1346: (8)     check = a_check.max(axis, initial=initial)
1347: (8)     assert_equal(res, check)
1348: (4) def test_reduction_where_initial_needed(self):
1349: (8)     a = np.arange(9.).reshape(3, 3)
1350: (8)     m = [False, True, False]
1351: (8)     assert_raises(ValueError, np.maximum.reduce, a, where=m)
1352: (4) def test_identityless_reduction_nonreorderable(self):
1353: (8)     a = np.array([[8.0, 2.0, 2.0], [1.0, 0.5, 0.25]])
1354: (8)     res = np.divide.reduce(a, axis=0)
1355: (8)     assert_equal(res, [8.0, 4.0, 8.0])
1356: (8)     res = np.divide.reduce(a, axis=1)
1357: (8)     assert_equal(res, [2.0, 8.0])
1358: (8)     res = np.divide.reduce(a, axis=())
1359: (8)     assert_equal(res, a)
1360: (8)     assert_raises(ValueError, np.divide.reduce, a, axis=(0, 1))
1361: (4) def test_reduce_zero_axis(self):
1362: (8)     def ok(f, *args, **kwargs):
1363: (12)         f(*args, **kwargs)
1364: (8)     def err(f, *args, **kwargs):
1365: (12)         assert_raises(ValueError, f, *args, **kwargs)
1366: (8)     def t(expect, func, n, m):
1367: (12)         expect(func, np.zeros((n, m)), axis=1)
1368: (12)         expect(func, np.zeros((m, n)), axis=0)
1369: (12)         expect(func, np.zeros((n // 2, n // 2, m)), axis=2)
1370: (12)         expect(func, np.zeros((n // 2, m, n // 2)), axis=1)
1371: (12)         expect(func, np.zeros((n, m // 2, m // 2)), axis=(1, 2))
1372: (12)         expect(func, np.zeros((m // 2, n, m // 2)), axis=(0, 2))
1373: (12)         expect(func, np.zeros((m // 3, m // 3, m // 3,
1374: (34)             n // 2, n // 2)), axis=(0, 1, 2))
1375: (33)         expect(func, np.zeros((10, m, n)), axis=(0, 1))
1376: (12)         expect(func, np.zeros((10, n, m)), axis=(0, 2))
1377: (12)         expect(func, np.zeros((m, 10, n)), axis=0)
1378: (12)         expect(func, np.zeros((10, m, n)), axis=1)
1379: (12)         expect(func, np.zeros((10, n, m)), axis=2)
1380: (12)     assert_equal(np.maximum.identity, None)
1381: (8)     t(ok, np.maximum.reduce, 30, 30)
1382: (8)     t(ok, np.maximum.reduce, 0, 30)
1383: (8)     t(err, np.maximum.reduce, 30, 0)
1384: (8)     t(err, np.maximum.reduce, 0, 0)
1385: (8)     err(np.maximum.reduce, [])
1386: (8)     np.maximum.reduce(np.zeros((0, 0)), axis=())
1387: (8)     t(ok, np.add.reduce, 30, 30)
1388: (8)     t(ok, np.add.reduce, 0, 30)

```

```

1390: (8)          t(ok, np.add.reduce, 30, 0)
1391: (8)          t(ok, np.add.reduce, 0, 0)
1392: (8)          np.add.reduce(())
1393: (8)          np.add.reduce(np.zeros((0, 0)), axis=())
1394: (8)          for uf in (np.maximum, np.add):
1395: (12)            uf.accumulate(np.zeros((30, 0)), axis=0)
1396: (12)            uf.accumulate(np.zeros((0, 30)), axis=0)
1397: (12)            uf.accumulate(np.zeros((30, 30)), axis=0)
1398: (12)            uf.accumulate(np.zeros((0, 0)), axis=0)
1399: (4)          def test_safe_casting(self):
1400: (8)            a = np.array([1, 2, 3], dtype=int)
1401: (8)            assert_array_equal(assert_no_warnings(np.add, a, 1.1),
1402: (27)                          [2.1, 3.1, 4.1])
1403: (8)            assert_raises(TypeError, np.add, a, 1.1, out=a)
1404: (8)          def add_inplace(a, b):
1405: (12)            a += b
1406: (8)            assert_raises(TypeError, add_inplace, a, 1.1)
1407: (8)            assert_no_warnings(np.add, a, 1.1, out=a, casting="unsafe")
1408: (8)            assert_array_equal(a, [2, 3, 4])
1409: (4)          def test_ufunc_custom_out(self):
1410: (8)            a = np.array([0, 1, 2], dtype='i8')
1411: (8)            b = np.array([0, 1, 2], dtype='i8')
1412: (8)            c = np.empty(3, dtype=_rational_tests.rational)
1413: (8)            result = _rational_tests.test_add(a, b, c)
1414: (8)            target = np.array([0, 2, 4], dtype=_rational_tests.rational)
1415: (8)            assert_equal(result, target)
1416: (8)            result = _rational_tests.test_add(a, b)
1417: (8)            assert_equal(result, target)
1418: (8)            result = _rational_tests.test_add(a, b.astype(np.uint16), out=c)
1419: (8)            assert_equal(result, target)
1420: (8)            with assert_raises(TypeError):
1421: (12)              _rational_tests.test_add(a, np.uint16(2))
1422: (4)          def test_operand_flags(self):
1423: (8)            a = np.arange(16, dtype='l').reshape(4, 4)
1424: (8)            b = np.arange(9, dtype='l').reshape(3, 3)
1425: (8)            opflag_tests(inplace_add(a[:-1, :-1], b))
1426: (8)            assert_equal(a, np.array([[0, 2, 4, 3], [7, 9, 11, 7],
1427: (12)              [14, 16, 18, 11], [12, 13, 14, 15]], dtype='l'))
1428: (8)            a = np.array(0)
1429: (8)            opflag_tests(inplace_add(a, 3))
1430: (8)            assert_equal(a, 3)
1431: (8)            opflag_tests(inplace_add(a, [3, 4]))
1432: (8)            assert_equal(a, 10)
1433: (4)          def test_struct_ufunc(self):
1434: (8)            import numpy.core._struct_ufunc_tests as struct_ufunc
1435: (8)            a = np.array([(1, 2, 3)], dtype='u8,u8,u8')
1436: (8)            b = np.array([(1, 2, 3)], dtype='u8,u8,u8')
1437: (8)            result = struct_ufunc.add_triplet(a, b)
1438: (8)            assert_equal(result, np.array([(2, 4, 6)], dtype='u8,u8,u8'))
1439: (8)            assert_raises(RuntimeError, struct_ufunc.register_fail)
1440: (4)          def test_custom_ufunc(self):
1441: (8)            a = np.array(
1442: (12)              [_rational_tests.rational(1, 2),
1443: (13)                _rational_tests.rational(1, 3),
1444: (13)                _rational_tests.rational(1, 4)],
1445: (12)              dtype=_rational_tests.rational)
1446: (8)            b = np.array(
1447: (12)              [_rational_tests.rational(1, 2),
1448: (13)                _rational_tests.rational(1, 3),
1449: (13)                _rational_tests.rational(1, 4)],
1450: (12)              dtype=_rational_tests.rational)
1451: (8)            result = _rational_tests.test_add_rationals(a, b)
1452: (8)            expected = np.array(
1453: (12)              [_rational_tests.rational(1),
1454: (13)                _rational_tests.rational(2, 3),
1455: (13)                _rational_tests.rational(1, 2)],
1456: (12)              dtype=_rational_tests.rational)
1457: (8)            assert_equal(result, expected)
1458: (4)          def test_custom_ufunc_forced_sig(self):

```

```

1459: (8)           with assert_raises(TypeError):
1460: (12)             np.multiply(_rational_tests.rational(1), 1,
1461: (24)               signature=(_rational_tests.rational, int, None))
1462: (4)             def test_custom_array_like(self):
1463: (8)               class MyThing:
1464: (12)                 __array_priority__ = 1000
1465: (12)                 rmul_count = 0
1466: (12)                 getitem_count = 0
1467: (12)                 def __init__(self, shape):
1468: (16)                   self.shape = shape
1469: (12)                 def __len__(self):
1470: (16)                   return self.shape[0]
1471: (12)                 def __getitem__(self, i):
1472: (16)                     MyThing.getitem_count += 1
1473: (16)                     if not isinstance(i, tuple):
1474: (20)                       i = (i,)
1475: (16)                     if len(i) > self.ndim:
1476: (20)                         raise IndexError("boo")
1477: (16)                     return MyThing(self.shape[len(i):])
1478: (12)                 def __rmul__(self, other):
1479: (16)                     MyThing.rmul_count += 1
1480: (16)                     return self
1481: (8)                     np.float64(5)*MyThing((3, 3))
1482: (8)                     assert_(MyThing.rmul_count == 1, MyThing.rmul_count)
1483: (8)                     assert_(MyThing.getitem_count <= 2, MyThing.getitem_count)
1484: (4)             @pytest.mark.parametrize("a", (
1485: (29)                 np.arange(10, dtype=int),
1486: (29)                 np.arange(10, dtype=_rational_tests.rational),
1487: (29)             ))
1488: (4)             def test_ufunc_at_basic(self, a):
1489: (8)               aa = a.copy()
1490: (8)               np.add.at(aa, [2, 5, 2], 1)
1491: (8)               assert_equal(aa, [0, 1, 4, 3, 4, 6, 6, 7, 8, 9])
1492: (8)               with pytest.raises(ValueError):
1493: (12)                 np.add.at(aa, [2, 5, 3])
1494: (8)                 aa = a.copy()
1495: (8)                 np.negative.at(aa, [2, 5, 3])
1496: (8)                 assert_equal(aa, [0, 1, -2, -3, 4, -5, 6, 7, 8, 9])
1497: (8)                 aa = a.copy()
1498: (8)                 b = np.array([100, 100, 100])
1499: (8)                 np.add.at(aa, [2, 5, 2], b)
1500: (8)                 assert_equal(aa, [0, 1, 202, 3, 4, 105, 6, 7, 8, 9])
1501: (8)                 with pytest.raises(ValueError):
1502: (12)                   np.negative.at(a, [2, 5, 3], [1, 2, 3])
1503: (8)                   with pytest.raises(ValueError):
1504: (12)                     np.add.at(a, [2, 5, 3], [[1, 2], 1])
1505: (4)             indexed_ufuncs = [np.add, np.subtract, np.multiply, np.floor_divide,
1506: (22)                           np.maximum, np.minimum, np.fmax, np.fmin]
1507: (4)             @pytest.mark.parametrize(
1508: (16)               "typecode", np.typecodes['AllInteger'] +
np.typecodes['Float'])
1509: (4)             @pytest.mark.parametrize("ufunc", indexed_ufuncs)
1510: (4)             def test_ufunc_at_inner_loops(self, typecode, ufunc):
1511: (8)               if ufunc is np.divide and typecode in np.typecodes['AllInteger']:
1512: (12)                 a = np.ones(100, dtype=typecode)
1513: (12)                 indx = np.random.randint(100, size=30, dtype=np.intp)
1514: (12)                 vals = np.arange(1, 31, dtype=typecode)
1515: (8)               else:
1516: (12)                 a = np.ones(1000, dtype=typecode)
1517: (12)                 indx = np.random.randint(1000, size=3000, dtype=np.intp)
1518: (12)                 vals = np.arange(3000, dtype=typecode)
1519: (8)                 atag = a.copy()
1520: (8)                 with warnings.catch_warnings(record=True) as w_at:
1521: (12)                   warnings.simplefilter('always')
1522: (12)                   ufunc.at(a, indx, vals)
1523: (8)                 with warnings.catch_warnings(record=True) as w_loop:
1524: (12)                   warnings.simplefilter('always')
1525: (12)                   for i, v in zip(indx, vals):
1526: (16)                     ufunc(atag[i], v, out=atag[i:i+1], casting="unsafe"))

```

```

1527: (8)             assert_equal(atag, a)
1528: (8)             if len(w_loop) > 0:
1529: (12)                 assert len(w_at) > 0
1530: (12)                 assert w_at[0].category == w_loop[0].category
1531: (12)                 assert str(w_at[0].message)[:10] == str(w_loop[0].message)[:10]
1532: (4)             @pytest.mark.parametrize("typecode", np.typecodes['Complex'])
1533: (4)             @pytest.mark.parametrize("ufunc", [np.add, np.subtract, np.multiply])
1534: (4)             def test_ufunc_at_inner_loops_complex(self, typecode, ufunc):
1535: (8)                 a = np.ones(10, dtype=typecode)
1536: (8)                 indx = np.concatenate([np.ones(6, dtype=np.intp),
1537: (31)                               np.full(18, 4, dtype=np.intp)])
1538: (8)                 value = a.dtype.type(1j)
1539: (8)                 ufunc.at(a, indx, value)
1540: (8)                 expected = np.ones_like(a)
1541: (8)                 if ufunc is np.multiply:
1542: (12)                     expected[1] = expected[4] = -1
1543: (8)                 else:
1544: (12)                     expected[1] += 6 * (value if ufunc is np.add else -value)
1545: (12)                     expected[4] += 18 * (value if ufunc is np.add else -value)
1546: (8)                 assert_array_equal(a, expected)
1547: (4)             def test_ufunc_at_ellipsis(self):
1548: (8)                 arr = np.zeros(5)
1549: (8)                 np.add.at(arr, slice(None), np.ones(5))
1550: (8)                 assert_array_equal(arr, np.ones(5))
1551: (4)             def test_ufunc_at_negative(self):
1552: (8)                 arr = np.ones(5, dtype=np.int32)
1553: (8)                 indx = np.arange(5)
1554: (8)                 umt.indexed_negative.at(arr, indx)
1555: (8)                 assert np.all(arr == [-1, -1, -1, -200, -1])
1556: (4)             def test_ufunc_at_large(self):
1557: (8)                 indices = np.zeros(8195, dtype=np.int16)
1558: (8)                 b = np.zeros(8195, dtype=float)
1559: (8)                 b[0] = 10
1560: (8)                 b[1] = 5
1561: (8)                 b[8192:] = 100
1562: (8)                 a = np.zeros(1, dtype=float)
1563: (8)                 np.add.at(a, indices, b)
1564: (8)                 assert a[0] == b.sum()
1565: (4)             def test_cast_index_fastpath(self):
1566: (8)                 arr = np.zeros(10)
1567: (8)                 values = np.ones(100000)
1568: (8)                 index = np.zeros(len(values), dtype=np.uint8)
1569: (8)                 np.add.at(arr, index, values)
1570: (8)                 assert arr[0] == len(values)
1571: (4)             @pytest.mark.parametrize("value", [
1572: (8)                 np.ones(1), np.ones(()), np.float64(1.), 1.])
1573: (4)             def test_ufunc_at_scalar_value_fastpath(self, value):
1574: (8)                 arr = np.zeros(1000)
1575: (8)                 index = np.repeat(np.arange(1000), 2)
1576: (8)                 np.add.at(arr, index, value)
1577: (8)                 assert_array_equal(arr, np.full_like(arr, 2 * value))
1578: (4)             def test_ufunc_at_multiD(self):
1579: (8)                 a = np.arange(9).reshape(3, 3)
1580: (8)                 b = np.array([[100, 100, 100], [200, 200, 200], [300, 300, 300]])
1581: (8)                 np.add.at(a, (slice(None), [1, 2, 1]), b)
1582: (8)                 assert_equal(a, [[0, 201, 102], [3, 404, 205], [6, 607, 308]])
1583: (8)                 a = np.arange(27).reshape(3, 3, 3)
1584: (8)                 b = np.array([100, 200, 300])
1585: (8)                 np.add.at(a, (slice(None), slice(None), [1, 2, 1]), b)
1586: (8)                 assert_equal(a,
1587: (12)                     [[[0, 401, 202],
1588: (14)                         [3, 404, 205],
1589: (14)                         [6, 407, 208]],
1590: (13)                         [[9, 410, 211],
1591: (14)                             [12, 413, 214],
1592: (14)                             [15, 416, 217]],
1593: (13)                             [[18, 419, 220],
1594: (14)                               [21, 422, 223],
1595: (14)                               [24, 425, 226]]])

```

```

1596: (8)          a = np.arange(9).reshape(3, 3)
1597: (8)          b = np.array([[100, 100, 100], [200, 200, 200], [300, 300, 300]])
1598: (8)          np.add.at(a, ([1, 2, 1], slice(None)), b)
1599: (8)          assert_equal(a, [[0, 1, 2], [403, 404, 405], [206, 207, 208]])
1600: (8)          a = np.arange(27).reshape(3, 3, 3)
1601: (8)          b = np.array([100, 200, 300])
1602: (8)          np.add.at(a, (slice(None), [1, 2, 1], slice(None)), b)
1603: (8)          assert_equal(a,
1604: (12)             [[[0, 1, 2],
1605: (14)               [203, 404, 605],
1606: (14)               [106, 207, 308]],
1607: (13)               [[9, 10, 11],
1608: (14)                 [212, 413, 614],
1609: (14)                 [115, 216, 317]],
1610: (13)               [[18, 19, 20],
1611: (14)                 [221, 422, 623],
1612: (14)                 [124, 225, 326]]])
1613: (8)          a = np.arange(9).reshape(3, 3)
1614: (8)          b = np.array([100, 200, 300])
1615: (8)          np.add.at(a, (0, [1, 2, 1]), b)
1616: (8)          assert_equal(a, [[0, 401, 202], [3, 4, 5], [6, 7, 8]])
1617: (8)          a = np.arange(27).reshape(3, 3, 3)
1618: (8)          b = np.array([100, 200, 300])
1619: (8)          np.add.at(a, ([1, 2, 1], 0, slice(None)), b)
1620: (8)          assert_equal(a,
1621: (12)             [[[0, 1, 2],
1622: (14)               [3, 4, 5],
1623: (14)               [6, 7, 8]],
1624: (13)               [[209, 410, 611],
1625: (14)                 [12, 13, 14],
1626: (14)                 [15, 16, 17]],
1627: (13)               [[118, 219, 320],
1628: (14)                 [21, 22, 23],
1629: (14)                 [24, 25, 26]]])
1630: (8)          a = np.arange(27).reshape(3, 3, 3)
1631: (8)          b = np.array([100, 200, 300])
1632: (8)          np.add.at(a, (slice(None), slice(None), slice(None)), b)
1633: (8)          assert_equal(a,
1634: (12)             [[[100, 201, 302],
1635: (14)               [103, 204, 305],
1636: (14)               [106, 207, 308]],
1637: (13)               [[109, 210, 311],
1638: (14)                 [112, 213, 314],
1639: (14)                 [115, 216, 317]],
1640: (13)               [[118, 219, 320],
1641: (14)                 [121, 222, 323],
1642: (14)                 [124, 225, 326]]])
1643: (4)          def test_ufunc_at_0D(self):
1644: (8)            a = np.array(0)
1645: (8)            np.add.at(a, (), 1)
1646: (8)            assert_equal(a, 1)
1647: (8)            assert_raises(IndexError, np.add.at, a, 0, 1)
1648: (8)            assert_raises(IndexError, np.add.at, a, [], 1)
1649: (4)          def test_ufunc_at_dtotypes(self):
1650: (8)            a = np.arange(10)
1651: (8)            np.power.at(a, [1, 2, 3, 2], 3.5)
1652: (8)            assert_equal(a, np.array([0, 1, 4414, 46, 4, 5, 6, 7, 8, 9]))
1653: (4)          def test_ufunc_at_boolean(self):
1654: (8)            a = np.arange(10)
1655: (8)            index = a % 2 == 0
1656: (8)            np.equal.at(a, index, [0, 2, 4, 6, 8])
1657: (8)            assert_equal(a, [1, 1, 1, 3, 1, 5, 1, 7, 1, 9])
1658: (8)            a = np.arange(10, dtype='u4')
1659: (8)            np.invert.at(a, [2, 5, 2])
1660: (8)            assert_equal(a, [0, 1, 2, 3, 4, 5 ^ 0xffffffff, 6, 7, 8, 9])
1661: (4)          def test_ufunc_at_advanced(self):
1662: (8)            orig = np.arange(4)
1663: (8)            a = orig[:, None][:, 0:0]
1664: (8)            np.add.at(a, [0, 1], 3)

```

```

1665: (8) assert_array_equal(orig, np.arange(4))
1666: (8) index = np.array([1, 2, 1], np.dtype('i').newbyteorder())
1667: (8) values = np.array([1, 2, 3, 4], np.dtype('f').newbyteorder())
1668: (8) np.add.at(values, index, 3)
1669: (8) assert_array_equal(values, [1, 8, 6, 4])
1670: (8) values = np.array(['a', 1], dtype=object)
1671: (8) assert_raises(TypeError, np.add.at, values, [0, 1], 1)
1672: (8) assert_array_equal(values, np.array(['a', 1], dtype=object))
1673: (8) assert_raises(ValueError, np.modf.at, np.arange(10), [1])
1674: (8) a = np.array([1, 2, 3])
1675: (8) np.maximum.at(a, [0], 0)
1676: (8) assert_equal(a, np.array([1, 2, 3]))
1677: (4) @pytest.mark.parametrize("dtype",
1678: (12)     np.typecodes['AllInteger'] + np.typecodes['Float'])
1679: (4) @pytest.mark.parametrize("ufunc",
1680: (12)     [np.add, np.subtract, np.divide, np.minimum, np.maximum])
1681: (4) def test_at_negative_indexes(self, dtype, ufunc):
1682: (8)     a = np.arange(0, 10).astype(dtype)
1683: (8)     idxs = np.array([-1, 1, -1, 2]).astype(np.intp)
1684: (8)     vals = np.array([1, 5, 2, 10], dtype=a.dtype)
1685: (8)     expected = a.copy()
1686: (8)     for i, v in zip(idxs, vals):
1687: (12)         expected[i] = ufunc(expected[i], v)
1688: (8)     ufunc.at(a, idxs, vals)
1689: (8)     assert_array_equal(a, expected)
1690: (8)     assert np.all(idxs == [-1, 1, -1, 2])
1691: (4) def test_at_not_none_signature(self):
1692: (8)     a = np.ones((2, 2, 2))
1693: (8)     b = np.ones((1, 2, 2))
1694: (8)     assert_raises(TypeError, np.matmul.at, a, [0], b)
1695: (8)     a = np.array([[1, 2], [3, 4]])
1696: (8)     assert_raises(TypeError, np.linalg._umath_linalg.det.at, a, [0])
1697: (4) def test_at_no_loop_for_op(self):
1698: (8)     arr = np.ones(10, dtype=str)
1699: (8)     with pytest.raises(np.core.exceptions.UFuncNoLoopError):
1700: (12)         np.add.at(arr, [0, 1], [0, 1])
1701: (4) def test_at_output_casting(self):
1702: (8)     arr = np.array([-1])
1703: (8)     np.equal.at(arr, [0], [0])
1704: (8)     assert arr[0] == 0
1705: (4) def test_at_broadcast_failure(self):
1706: (8)     arr = np.arange(5)
1707: (8)     with pytest.raises(ValueError):
1708: (12)         np.add.at(arr, [0, 1], [1, 2, 3])
1709: (4) def test_reduce_arguments(self):
1710: (8)     f = np.add.reduce
1711: (8)     d = np.ones((5,2), dtype=int)
1712: (8)     o = np.ones((2,), dtype=d.dtype)
1713: (8)     r = o * 5
1714: (8)     assert_equal(f(d), r)
1715: (8)     assert_equal(f(d, axis=0), r)
1716: (8)     assert_equal(f(d, 0), r)
1717: (8)     assert_equal(f(d, 0, dtype=None), r)
1718: (8)     assert_equal(f(d, 0, dtype='i'), r)
1719: (8)     assert_equal(f(d, 0, 'i'), r)
1720: (8)     assert_equal(f(d, 0, None), r)
1721: (8)     assert_equal(f(d, 0, None, out=None), r)
1722: (8)     assert_equal(f(d, 0, None, out=o), r)
1723: (8)     assert_equal(f(d, 0, None, o), r)
1724: (8)     assert_equal(f(d, 0, None, None), r)
1725: (8)     assert_equal(f(d, 0, None, None, keepdims=False), r)
1726: (8)     assert_equal(f(d, 0, None, None, True), r.reshape((1,) + r.shape))
1727: (8)     assert_equal(f(d, 0, None, None, False, 0), r)
1728: (8)     assert_equal(f(d, 0, None, None, False, initial=0), r)
1729: (8)     assert_equal(f(d, 0, None, None, False, 0, True), r)
1730: (8)     assert_equal(f(d, 0, None, None, False, 0, where=True), r)
1731: (8)     assert_equal(f(d, axis=0, dtype=None, out=None, keepdims=False), r)
1732: (8)     assert_equal(f(d, 0, dtype=None, out=None, keepdims=False), r)
1733: (8)     assert_equal(f(d, 0, None, out=None, keepdims=False), r)

```

```

1734: (8)
1735: (23)
1736: (8)
1737: (8)
1738: (8)
1739: (8)
1740: (8)
1741: (22)
1742: (8)
1743: (8)
1744: (8)
1745: (8)
1746: (8)
1747: (8)
1748: (8)
1749: (21)
1750: (8)
1751: (8)
1752: (8)
1753: (22)
1754: (8)
1755: (22)
1756: (8)
1757: (22)
1758: (4)
1759: (8)
1760: (12)
1761: (16)
1762: (46)
1763: (8)
1764: (8)
1765: (8)
1766: (8)
1767: (8)
1768: (4)
1769: (8)
1770: (8)
1771: (8)
1772: (8)
1773: (8)
1774: (8)
1775: (4)
1776: (8)
1777: (12)
1778: (12)
1779: (12)
1780: (12)
1781: (12)
1782: (12)
1783: (12)
1784: (8)
1785: (8)
1786: (8)
1787: (8)
1788: (12)
1789: (12)
1790: (4)
1791: (13)
bad
1792: (4)
1793: (13)
1794: (14)
1795: (4)
1796: (8)
1797: (8)
1798: (8)
1799: (4)
1800: (12)
1801: (4)

        assert_equal(f(d, 0, None, out=None, keepdims=False, initial=0,
                         where=True), r)
        assert_raises(TypeError, f)
        assert_raises(TypeError, f, d, 0, None, None, False, 0, True, 1)
        assert_raises(TypeError, f, d, "invalid")
        assert_raises(TypeError, f, d, axis="invalid")
        assert_raises(TypeError, f, d, axis="invalid", dtype=None,
                         keepdims=True)
        assert_raises(TypeError, f, d, 0, "invalid")
        assert_raises(TypeError, f, d, dtype="invalid")
        assert_raises(TypeError, f, d, dtype="invalid", out=None)
        assert_raises(TypeError, f, d, 0, None, "invalid")
        assert_raises(TypeError, f, d, out="invalid")
        assert_raises(TypeError, f, d, out="invalid", dtype=None)
        assert_raises(TypeError, f, d, 0, keepdims="invalid", dtype="invalid",
                         out=None)
        assert_raises(TypeError, f, d, axis=0, dtype=None, invalid=0)
        assert_raises(TypeError, f, d, invalid=0)
        assert_raises(TypeError, f, d, 0, keepdims=True, invalid="invalid",
                         out=None)
        assert_raises(TypeError, f, d, axis=0, dtype=None, keepdims=True,
                         out=None, invalid=0)
        assert_raises(TypeError, f, d, axis=0, dtype=None, out=None,
                         invalid=0)

def test_structured_equal(self):
    class MyA(np.ndarray):
        def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
            return getattr(ufunc, method)(*(input.view(np.ndarray)
   for input in inputs), **kwargs)

    a = np.arange(12.).reshape(4,3)
    ra = a.view(dtype=('f8,f8,f8')).squeeze()
    mra = ra.view(MyA)
    target = np.array([ True, False, False, False], dtype=bool)
    assert_equal(np.all(target == (mra == ra[0])), True)

def test_scalar_equal(self):
    a = np.array(0.)
    b = np.array('a')
    assert_(a != b)
    assert_(b != a)
    assert_(not (a == b))
    assert_(not (b == a))

def test_NotImplemented_not_returned(self):
    binary_funcs = [
        np.power, np.add, np.subtract, np.multiply, np.divide,
        np.true_divide, np.floor_divide, np.bitwise_and, np.bitwise_or,
        np.bitwise_xor, np.left_shift, np.right_shift, np.fmax,
        np.fmin, np.fmod, np.hypot, np.logaddexp, np.logaddexp2,
        np.maximum, np.minimum, np.mod,
        np.greater, np.greater_equal, np.less, np.less_equal,
        np.equal, np.not_equal]
    a = np.array('1')
    b = 1
    c = np.array([1., 2.])
    for f in binary_funcs:
        assert_raises(TypeError, f, a, b)
        assert_raises(TypeError, f, c, a)

@pytest.mark.parametrize("ufunc",
                       [np.logical_and, np.logical_or]) # logical_xor object loop is

@ pytest.mark.parametrize("signature",
                        [(None, None, object), (object, None, None),
                         (None, object, None)])
def test_logical_ufuncs_object_signatures(self, ufunc, signature):
    a = np.array([True, None, False], dtype=object)
    res = ufunc(a, a, signature=signature)
    assert res.dtype == object

@pytest.mark.parametrize("ufunc",
                       [np.logical_and, np.logical_or, np.logical_xor])
@ pytest.mark.parametrize("signature",

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1802: (17)             [(bool, None, object), (object, None, bool),
1803: (18)                         (None, object, bool)])
1804: (4)          def test_logical_ufuncs_mixed_object_signatures(self, ufunc, signature):
1805: (8)              a = np.array([True, None, False])
1806: (8)              with pytest.raises(TypeError):
1807: (12)                  ufunc(a, a, signature=signature)
1808: (4)          @pytest.mark.parametrize("ufunc",
1809: (12)              [np.logical_and, np.logical_or, np.logical_xor])
1810: (4)          def test_logical_ufuncs_support_anything(self, ufunc):
1811: (8)              a = np.array(b'1', dtype="V3")
1812: (8)              c = np.array([1., 2.])
1813: (8)              assert_array_equal(ufunc(a, c), ufunc([True, True], True))
1814: (8)              assert ufunc.reduce(a) == True
1815: (8)              out = np.zeros(2, dtype=np.int32)
1816: (8)              expected = ufunc([True, True], True).astype(out.dtype)
1817: (8)              assert_array_equal(ufunc(a, c, out=out), expected)
1818: (8)              out = np.zeros(), dtype=np.int32)
1819: (8)              assert ufunc.reduce(a, out=out) == True
1820: (8)              a = np.array([3], dtype="i")
1821: (8)              out = np.zeros(), dtype=a.dtype)
1822: (8)              assert ufunc.reduce(a, out=out) == 1
1823: (4)          @pytest.mark.parametrize("ufunc",
1824: (12)              [np.logical_and, np.logical_or, np.logical_xor])
1825: (4)          def test_logical_ufuncs_reject_string(self, ufunc):
1826: (8)          """
1827: (8)              Logical ufuncs are normally well defined by working with the boolean
1828: (8)              equivalent, i.e. casting all inputs to bools should work.
1829: (8)              However, casting strings to bools is *currently* weird, because it
1830: (8)              actually uses `bool(int(str))`. Thus we explicitly reject strings.
1831: (8)              This test should succeed (and can probably just be removed) as soon as
1832: (8)              string to bool casts are well defined in NumPy.
1833: (8)          """
1834: (8)          with pytest.raises(TypeError, match="contain a loop with signature"):
1835: (12)              ufunc(["1"], ["3"])
1836: (8)          with pytest.raises(TypeError, match="contain a loop with signature"):
1837: (12)              ufunc.reduce(["1", "2", "0"])
1838: (4)          @pytest.mark.parametrize("ufunc",
1839: (13)              [np.logical_and, np.logical_or, np.logical_xor])
1840: (4)          def test_logical_ufuncs_out_cast_check(self, ufunc):
1841: (8)              a = np.array('1')
1842: (8)              c = np.array([1., 2.])
1843: (8)              out = a.copy()
1844: (8)              with pytest.raises(TypeError):
1845: (12)                  ufunc(a, c, out=out, casting="equiv")
1846: (4)          def test_reducelike_byteorder_resolution(self):
1847: (8)              arr_be = np.arange(10, dtype=">i8")
1848: (8)              arr_le = np.arange(10, dtype="<i8")
1849: (8)              assert np.add.reduce(arr_be) == np.add.reduce(arr_le)
1850: (8)              assert_array_equal(np.add.accumulate(arr_be),
np.add.accumulate(arr_le))
1851: (8)              assert_array_equal(
1852: (12)                  np.add.reduceat(arr_be, [1]), np.add.reduceat(arr_le, [1]))
1853: (4)          def test_reducelike_out_promotes(self):
1854: (8)              arr = np.ones(1000, dtype=np.uint8)
1855: (8)              out = np.zeros(), dtype=np.uint16)
1856: (8)              assert np.add.reduce(arr, out=out) == 1000
1857: (8)              arr[:10] = 2
1858: (8)              assert np.multiply.reduce(arr, out=out) == 2**10
1859: (8)              arr = np.full(5, 2**25-1, dtype=np.int64)
1860: (8)              res = np.zeros(), dtype=np.float32)
1861: (8)              single_res = np.zeros(), dtype=np.float32)
1862: (8)              np.multiply.reduce(arr, out=single_res, dtype=np.float32)
1863: (8)              assert single_res != res
1864: (4)          def test_reducelike_output_needs_identical_cast(self):
1865: (8)              arr = np.ones(20, dtype="f8")
1866: (8)              out = np.empty(), dtype=arr.dtype.newbyteorder())
1867: (8)              expected = np.add.reduce(arr)
1868: (8)              np.add.reduce(arr, out=out)
1869: (8)              assert_array_equal(expected, out)

```

```

1870: (8)          out = np.empty(2, dtype=arr.dtype.newbyteorder())
1871: (8)          expected = np.add.reduceat(arr, [0, 1])
1872: (8)          np.add.reduceat(arr, [0, 1], out=out)
1873: (8)          assert_array_equal(expected, out)
1874: (8)          out = np.empty(arr.shape, dtype=arr.dtype.newbyteorder())
1875: (8)          expected = np.add.accumulate(arr)
1876: (8)          np.add.accumulate(arr, out=out)
1877: (8)          assert_array_equal(expected, out)
1878: (4)          def test_reduce_noncontig_output(self):
1879: (8)          x = np.arange(7*13*8, dtype=np.int16).reshape(7, 13, 8)
1880: (8)          x = x[4:6,1:11:6,1:5].transpose(1, 2, 0)
1881: (8)          y_base = np.arange(4*4, dtype=np.int16).reshape(4, 4)
1882: (8)          y = y_base[:,::2,:]
1883: (8)          y_base_copy = y_base.copy()
1884: (8)          r0 = np.add.reduce(x, out=y.copy(), axis=2)
1885: (8)          r1 = np.add.reduce(x, out=y, axis=2)
1886: (8)          assert_equal(r0, r1)
1887: (8)          assert_equal(y_base[1,:], y_base_copy[1,:])
1888: (8)          assert_equal(y_base[3,:], y_base_copy[3,:])
1889: (4)          @pytest.mark.parametrize("with_cast", [True, False])
1890: (4)          def test_reduceat_and_accumulate_out_shape_mismatch(self, with_cast):
1891: (8)          arr = np.arange(5)
1892: (8)          out = np.arange(3) # definitely wrong shape
1893: (8)          if with_cast:
1894: (12)             out = out.astype(np.float64)
1895: (8)          with pytest.raises(ValueError, match="(shape|size)"):
1896: (12)             np.add.reduceat(arr, [0, 3], out=out)
1897: (8)          with pytest.raises(ValueError, match="(shape|size)"):
1898: (12)             np.add.accumulate(arr, out=out)
1899: (4)          @pytest.mark.parametrize('out_shape',
1900: (29)             [(), (1,), (3,), (1, 1), (1, 3), (4, 3)])
1901: (4)          @pytest.mark.parametrize('keepdims', [True, False])
1902: (4)          @pytest.mark.parametrize('f_reduce', [np.add.reduce, np.minimum.reduce])
1903: (4)          def test_reduce_wrong_dimension_output(self, f_reduce, keepdims,
out_shape):
1904: (8)          a = np.arange(12.).reshape(4, 3)
1905: (8)          out = np.empty(out_shape, a.dtype)
1906: (8)          correct_out = f_reduce(a, axis=0, keepdims=keepdims)
1907: (8)          if out_shape != correct_out.shape:
1908: (12)             with assert_raises(ValueError):
1909: (16)                 f_reduce(a, axis=0, out=out, keepdims=keepdims)
1910: (8)          else:
1911: (12)             check = f_reduce(a, axis=0, out=out, keepdims=keepdims)
1912: (12)             assert_(check is out)
1913: (12)             assert_array_equal(check, correct_out)
1914: (4)          def test_reduce_output_does_not_broadcast_input(self):
1915: (8)          a = np.ones((1, 10))
1916: (8)          out_correct = (np.empty((1, 1)))
1917: (8)          out_incorrect = np.empty((3, 1))
1918: (8)          np.add.reduce(a, axis=-1, out=out_correct, keepdims=True)
1919: (8)          np.add.reduce(a, axis=-1, out=out_correct[:, 0], keepdims=False)
1920: (8)          with assert_raises(ValueError):
1921: (12)             np.add.reduce(a, axis=-1, out=out_incorrect, keepdims=True)
1922: (8)          with assert_raises(ValueError):
1923: (12)             np.add.reduce(a, axis=-1, out=out_incorrect[:, 0], keepdims=False)
1924: (4)          def test_reduce_output_subclass_ok(self):
1925: (8)          class MyArr(np.ndarray):
1926: (12)             pass
1927: (8)          out = np.empty(())
1928: (8)          np.add.reduce(np.ones(5), out=out) # no subclass, all fine
1929: (8)          out = out.view(MyArr)
1930: (8)          assert np.add.reduce(np.ones(5), out=out) is out
1931: (8)          assert type(np.add.reduce(out)) is MyArr
1932: (4)          def test_no_doc_string(self):
1933: (8)             assert_('\n' not in umt.inner1d_no_doc.__doc__)
1934: (4)          def test_invalid_args(self):
1935: (8)             exc = pytest.raises(TypeError, np.sqrt, None)
1936: (8)             assert exc.match('loop of ufunc does not support')
1937: (4)          @pytest.mark.parametrize('nat', [np.datetime64('nat'),

```

```

np.timedelta64('nat')])
1938: (4)             def test_nat_is_not_finite(self, nat):
1939: (8)                 try:
1940: (12)                     assert not np.isfinite(nat)
1941: (8)                 except TypeError:
1942: (12)                     pass # ok, just not implemented
1943: (4)             @pytest.mark.parametrize('nat', [np.datetime64('nat'),
np.timedelta64('nat')])
1944: (4)             def test_nat_is_nan(self, nat):
1945: (8)                 try:
1946: (12)                     assert np.isnan(nat)
1947: (8)                 except TypeError:
1948: (12)                     pass # ok, just not implemented
1949: (4)             @pytest.mark.parametrize('nat', [np.datetime64('nat'),
np.timedelta64('nat')])
1950: (4)             def test_nat_is_not_inf(self, nat):
1951: (8)                 try:
1952: (12)                     assert not np.isinf(nat)
1953: (8)                 except TypeError:
1954: (12)                     pass # ok, just not implemented
1955: (0)             @pytest.mark.parametrize('ufunc', [getattr(np, x) for x in dir(np)
1956: (32)   if isinstance(getattr(np, x), np.ufunc)])
1957: (0)
1958: (4)             def test_ufunc_types(ufunc):
1959: (4)                 """
1960: (4)                     Check all ufuncs that the correct type is returned. Avoid
1961: (4)                     object and boolean types since many operations are not defined for
1962: (4)                     them.
1963: (4)                     Choose the shape so even dot and matmul will succeed
1964: (4)                 """
1965: (8)                 for typ in ufunc.types:
1966: (12)                     if 'O' in typ or '?' in typ:
1967: (8)                         continue
1968: (8)                     inp, out = typ.split('->')
1969: (8)                     args = [np.ones((3, 3), t) for t in inp]
1970: (12)                     with warnings.catch_warnings(record=True):
1971: (12)                         warnings.filterwarnings("always")
1972: (8)                         res = ufunc(*args)
1973: (12)                         if isinstance(res, tuple):
1974: (12)                             outs = tuple(out)
1975: (12)                             assert len(res) == len(outs)
1976: (16)                             for r, t in zip(res, outs):
1977: (8)                                 assert r.dtype == np.dtype(t)
1978: (12)                         else:
1979: (0)                             assert res.dtype == np.dtype(out)
1980: (32)             @pytest.mark.parametrize('ufunc', [getattr(np, x) for x in dir(np)
1981: (0)   if isinstance(getattr(np, x), np.ufunc)])
1982: (0)             @np._no_nep50_warning()
1983: (4)             def test_ufunc_noncontiguous(ufunc):
1984: (4)                 """
1985: (4)                     Check that contiguous and non-contiguous calls to ufuncs
1986: (4)                     have the same results for values in range(9)
1987: (4)                 """
1988: (8)                 for typ in ufunc.types:
1989: (12)                     if any(set('O?mM') & set(typ)):
1990: (8)                         continue
1991: (8)                     inp, out = typ.split('->')
1992: (8)                     args_c = [np.empty(6, t) for t in inp]
1993: (8)                     args_n = [np.empty(18, t)[::3] for t in inp]
1994: (12)                     for a in args_c:
1995: (8)                         a.flat = range(1,7)
1996: (12)                     for a in args_n:
1997: (8)                         a.flat = range(1,7)
1998: (12)                     with warnings.catch_warnings(record=True):
1999: (12)                         warnings.filterwarnings("always")
2000: (12)                         res_c = ufunc(*args_c)
2001: (8)                         res_n = ufunc(*args_n)
2002: (12)                         if len(out) == 1:
2003: (12)                             res_c = (res_c,)
2003: (12)                             res_n = (res_n,) 
```

```

2004: (8)           for c_ar, n_ar in zip(res_c, res_n):
2005: (12)          dt = c_ar.dtype
2006: (12)          if np.issubdtype(dt, np.floating):
2007: (16)            res_eps = np.finfo(dt).eps
2008: (16)            tol = 2*res_eps
2009: (16)            assert_allclose(res_c, res_n, atol=tol, rtol=tol)
2010: (12)          else:
2011: (16)            assert_equal(c_ar, n_ar)
2012: (0) @pytest.mark.parametrize('ufunc', [np.sign, np.equal])
2013: (0) def test_ufunc_warn_with_nan(ufunc):
2014: (4)   b = np.array([0x7ff00000000000001], 'i8').view('f8')
2015: (4)   assert np.isnan(b)
2016: (4)   if ufunc.nin == 1:
2017: (8)     ufunc(b)
2018: (4)   elif ufunc.nin == 2:
2019: (8)     ufunc(b, b.copy())
2020: (4)   else:
2021: (8)     raise ValueError('ufunc with more than 2 inputs')
2022: (0) @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
2023: (0) def test_ufunc_out_casterrors():
2024: (4)   value = 123 # relies on python cache (leak-check will still find it)
2025: (4)   arr = np.array([value] * int(np.BUFSIZE * 1.5) +
2026: (19)     ["string"] +
2027: (19)     [value] * int(1.5 * np.BUFSIZE), dtype=object)
2028: (4)   out = np.ones(len(arr), dtype=np.intp)
2029: (4)   count = sys.getrefcount(value)
2030: (4)   with pytest.raises(ValueError):
2031: (8)     np.add(arr, arr, out=out, casting="unsafe")
2032: (4)   assert count == sys.getrefcount(value)
2033: (4)   assert out[-1] == 1
2034: (4)   with pytest.raises(ValueError):
2035: (8)     np.add(arr, arr, out=out, dtype=np.intp, casting="unsafe")
2036: (4)   assert count == sys.getrefcount(value)
2037: (4)   assert out[-1] == 1
2038: (0) @pytest.mark.parametrize("bad_offset", [0, int(np.BUFSIZE * 1.5)])
2039: (0) def test_ufunc_input_casterrors(bad_offset):
2040: (4)   value = 123
2041: (4)   arr = np.array([value] * bad_offset +
2042: (19)     ["string"] +
2043: (19)     [value] * int(1.5 * np.BUFSIZE), dtype=object)
2044: (4)   with pytest.raises(ValueError):
2045: (8)     np.add(arr, arr, dtype=np.intp, casting="unsafe")
2046: (0) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
2047: (0) @pytest.mark.parametrize("bad_offset", [0, int(np.BUFSIZE * 1.5)])
2048: (0) def test_ufunc_input_floatingpoint_error(bad_offset):
2049: (4)   value = 123
2050: (4)   arr = np.array([value] * bad_offset +
2051: (19)     [np.nan] +
2052: (19)     [value] * int(1.5 * np.BUFSIZE))
2053: (4)   with np.errstate(invalid="raise"), pytest.raises(FloatingPointError):
2054: (8)     np.add(arr, arr, dtype=np.intp, casting="unsafe")
2055: (0) def test_trivial_loop_invalid_cast():
2056: (4)   with pytest.raises(TypeError,
2057: (12)     match="cast ufunc 'add' input 0"):
2058: (8)     np.add(np.array(1, "i,i"), 3, signature="dd->d")
2059: (0) @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
2060: (0) @pytest.mark.parametrize("offset",
2061: (8)   [0, np.BUFSIZE//2, int(1.5*np.BUFSIZE)])
2062: (0) def test_reduce_casterrors(offset):
2063: (4)   value = 123 # relies on python cache (leak-check will still find it)
2064: (4)   arr = np.array([value] * offset +
2065: (19)     ["string"] +
2066: (19)     [value] * int(1.5 * np.BUFSIZE), dtype=object)
2067: (4)   out = np.array(-1, dtype=np.intp)
2068: (4)   count = sys.getrefcount(value)
2069: (4)   with pytest.raises(ValueError, match="invalid literal"):
2070: (8)     np.add.reduce(arr, dtype=np.intp, out=out, initial=None)
2071: (4)   assert count == sys.getrefcount(value)
2072: (4)   assert out[()] < value * offset

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

2073: (0)
2074: (4)
2075: (8)
2076: (4)
2077: (8)
2078: (0)
2079: (0)
2080: (8)
2081: (9)
2082: (9)
2083: (0)
2084: (4)
2085: (4)
2086: (8)
2087: (12)
2088: (4)
2089: (4)
2090: (8)
2091: (12)
2092: (0)
2093: (4)
2094: (8)
2095: (4)
2096: (8)
2097: (4)
2098: (8)
2099: (4)
2100: (0)
2101: (0)
def test_object_reduce_cleanup_on_failure():
    with pytest.raises(TypeError):
        np.add.reduce([1, 2, None], initial=4)
    with pytest.raises(TypeError):
        np.add.reduce([1, 2, None])
@ pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
@ pytest.mark.parametrize("method",
                         [np.add.accumulate, np.add.reduce,
                          pytest.param(lambda x: np.add.reduceat(x, [0]), id="reduceat"),
                          pytest.param(lambda x: np.log.at(x, [2]), id="at")])
def test_ufunc_methods_floaterrors(method):
    arr = np.array([np.inf, 0, -np.inf])
    with np.errstate(all="warn"):
        with pytest.warns(RuntimeWarning, match="invalid value"):
            method(arr)
    arr = np.array([np.inf, 0, -np.inf])
    with np.errstate(all="raise"):
        with pytest.raises(FloatingPointError):
            method(arr)
def _check_neg_zero(value):
    if value != 0.0:
        return False
    if not np.signbit(value.real):
        return False
    if value.dtype.kind == "c":
        return np.signbit(value.imag)
    return True
@ pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
def test_addition_negative_zero(dtype):
    dtype = np.dtype(dtype)
    if dtype.kind == "c":
        neg_zero = dtype.type(complex(-0.0, -0.0))
    else:
        neg_zero = dtype.type(-0.0)
    arr = np.array(neg_zero)
    arr2 = np.array(neg_zero)
    assert _check_neg_zero(arr + arr2)
    arr += arr2
    assert _check_neg_zero(arr)
@ pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
@ pytest.mark.parametrize("use_initial", [True, False])
def test_addition_reduce_negative_zero(dtype, use_initial):
    dtype = np.dtype(dtype)
    if dtype.kind == "c":
        neg_zero = dtype.type(complex(-0.0, -0.0))
    else:
        neg_zero = dtype.type(-0.0)
    kwargs = {}
    if use_initial:
        kwargs["initial"] = neg_zero
    else:
        pytest.xfail("-0. propagation in sum currently requires initial")
    for i in range(0, 150):
        arr = np.array([neg_zero] * i, dtype=dtype)
        res = np.sum(arr, **kwargs)
        if i > 0 or use_initial:
            assert _check_neg_zero(res)
        else:
            assert not np.signbit(res.real)
            assert not np.signbit(res.imag)
class TestLowlevelAPIAccess:
    def test_resolve_dtypes_basic(self):
        i4 = np.dtype("i4")
        f4 = np.dtype("f4")
        f8 = np.dtype("f8")
        r = np.add.resolve_dtypes((i4, f4, None))
        assert r == (f8, f8, f8)
        r = np.add.resolve_dtypes(
            i4, i4, None), signature=(None, None, "f4"))

```

```

2142: (8)             assert r == (f4, f4, f4)
2143: (8)             r = np.add.resolve_dtypes((f4, int, None))
2144: (8)             assert r == (f4, f4, f4)
2145: (8)             with pytest.raises(TypeError):
2146: (12)                 np.add.resolve_dtypes((i4, f4, None), casting="no")
2147: (4)             def test_weird_dtypes(self):
2148: (8)                 S0 = np.dtype("S0")
2149: (8)                 r = np.equal.resolve_dtypes((S0, S0, None))
2150: (8)                 assert r == (S0, S0, np.dtype(bool))
2151: (8)                 dts = np.dtype("10i")
2152: (8)                 with pytest.raises(TypeError):
2153: (12)                     np.equal.resolve_dtypes((dts, dts, None))
2154: (4)             def test_resolve_dtypes_reduction(self):
2155: (8)                 i4 = np.dtype("i4")
2156: (8)                 with pytest.raises(NotImplementedError):
2157: (12)                     np.add.resolve_dtypes((i4, i4, i4), reduction=True)
2158: (4)             @pytest.mark.parametrize("dtypes", [
2159: (12)                 (np.dtype("i"), np.dtype("i")),
2160: (12)                 (None, np.dtype("i"), np.dtype("f")),
2161: (12)                 (np.dtype("i"), None, np.dtype("f")),
2162: (12)                 ("i4", "i4", None)])
2163: (4)             def test_resolve_dtypes_errors(self, dtypes):
2164: (8)                 with pytest.raises(TypeError):
2165: (12)                     np.add.resolve_dtypes(dtypes)
2166: (4)             def test_resolve_dtypes_reduction(self):
2167: (8)                 i2 = np.dtype("i2")
2168: (8)                 long_ = np.dtype("long")
2169: (8)                 res = np.add.resolve_dtypes((None, i2, None), reduction=True)
2170: (8)                 assert res == (long_, long_, long_)
2171: (4)             def test_resolve_dtypes_reduction_errors(self):
2172: (8)                 i2 = np.dtype("i2")
2173: (8)                 with pytest.raises(TypeError):
2174: (12)                     np.add.resolve_dtypes((None, i2, i2))
2175: (8)                 with pytest.raises(TypeError):
2176: (12)                     np.add.signature((None, None, "i4"))
2177: (4)             @pytest.mark.skipif(not hasattr(ct, "pythonapi"),
2178: (12)                         reason="`ctypes.pythonapi` required for capsule unpacking.")
2179: (4)             def test_loop_access(self):
2180: (8)                 data_t = ct.ARRAY(ct.c_char_p, 2)
2181: (8)                 dim_t = ct.ARRAY(ct.c_size_t, 1)
2182: (8)                 strides_t = ct.ARRAY(ct.c_size_t, 2)
2183: (8)                 strided_loop_t = ct.CFUNCTYPE(
2184: (16)                     ct.c_int, ct.c_void_p, data_t, dim_t, strides_t, ct.c_void_p)
2185: (8)             class call_info_t(ct.Structure):
2186: (12)                 _fields_ = [
2187: (16)                     ("strided_loop", strided_loop_t),
2188: (16)                     ("context", ct.c_void_p),
2189: (16)                     ("auxdata", ct.c_void_p),
2190: (16)                     ("requires_pyapi", ct.c_byte),
2191: (16)                     ("no_floatingpoint_errors", ct.c_byte),
2192: (12)                 ]
2193: (8)                 i4 = np.dtype("i4")
2194: (8)                 dt, call_info_obj = np.negative._resolve_dtypes_and_context((i4, i4))
2195: (8)                 assert dt == (i4, i4) # can be used without casting
2196: (8)                 np.negative._get_strided_loop(call_info_obj)
2197: (8)                 ct.pythonapi.PyCapsule_GetPointer.restype = ct.c_void_p
2198: (8)                 call_info = ct.pythonapi.PyCapsule_GetPointer(
2199: (16)                     ct.py_object(call_info_obj),
2200: (16)                     ct.c_char_p(b"numpy_1.24_ufunc_call_info"))
2201: (8)                 call_info = ct.cast(call_info, ct.POINTER(call_info_t)).contents
2202: (8)                 arr = np.arange(10, dtype=i4)
2203: (8)                 call_info.strided_loop(
2204: (16)                     call_info.context,
2205: (16)                     data_t(arr.ctypes.data, arr.ctypes.data),
2206: (16)                     arr.ctypes.shape, # is a C-array with 10 here
2207: (16)                     strides_t(arr.ctypes.strides[0], arr.ctypes.strides[0]),
2208: (16)                     call_info.auxdata)
2209: (8)                 assert_array_equal(arr, -np.arange(10, dtype=i4))
2210: (4)             @pytest.mark.parametrize("strides", [1, (1, 2, 3), (1, "2")])

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2211: (4)         def test__get_strided_loop_errors_bad_strides(self, strides):
2212: (8)             i4 = np.dtype("i4")
2213: (8)             dt, call_info = np.negative._resolve_dtypes_and_context((i4, i4))
2214: (8)             with pytest.raises(TypeError, match="fixed_strides.*tuple.*or None"):
2215: (12)                 np.negative._get_strided_loop(call_info, fixed_strides=strides)
2216: (4)         def test__get_strided_loop_errors_bad_call_info(self):
2217: (8)             i4 = np.dtype("i4")
2218: (8)             dt, call_info = np.negative._resolve_dtypes_and_context((i4, i4))
2219: (8)             with pytest.raises(ValueError, match="PyCapsule"):
2220: (12)                 np.negative._get_strided_loop("not the capsule!")
2221: (8)             with pytest.raises(TypeError, match=".incompatible context"):
2222: (12)                 np.add._get_strided_loop(call_info)
2223: (8)                 np.negative._get_strided_loop(call_info)
2224: (8)             with pytest.raises(TypeError):
2225: (12)                 np.negative._get_strided_loop(call_info)
2226: (4)         def test_long_arrays(self):
2227: (8)             t = np.zeros((1029, 917), dtype=np.single)
2228: (8)             t[0][0] = 1
2229: (8)             t[28][414] = 1
2230: (8)             tc = np.cos(t)
2231: (8)             assert_equal(tc[0][0], tc[28][414])

```

---

## File 130 - test\_umath.py:

```

1: (0)             import platform
2: (0)             import warnings
3: (0)             import fnmatch
4: (0)             import itertools
5: (0)             import pytest
6: (0)             import sys
7: (0)             import os
8: (0)             import operator
9: (0)             from fractions import Fraction
10: (0)            from functools import reduce
11: (0)            from collections import namedtuple
12: (0)            import numpy.core.umath as ncu
13: (0)            from numpy.core import _umath_tests as ncu_tests
14: (0)            import numpy as np
15: (0)            from numpy.testing import (
16: (4)                assert_, assert_equal, assert_raises, assert_raises_regex,
17: (4)                assert_array_equal, assert_almost_equal, assert_array_almost_equal,
18: (4)                assert_array_max_ulp, assert_allclose, assert_no_warnings,
suppress_warnings,
19: (4)                _gen_alignment_data, assert_array_almost_equal_nulp, IS_WASM, IS_MUSL,
20: (4)                IS_PYPY
21: (4)            )
22: (0)            from numpy.testing._private.utils import _glibc_older_than
23: (0)            UFUNCS = [obj for obj in np.core.umath.__dict__.values()
24: (9)                if isinstance(obj, np.ufunc)]
25: (0)            UFUNCS_UNARY = [
26: (4)                uf for uf in UFUNCS if uf.nin == 1
27: (0)            ]
28: (0)            UFUNCS_UNARY_FP = [
29: (4)                uf for uf in UFUNCS_UNARY if 'f->f' in uf.types
30: (0)            ]
31: (0)            UFUNCS_BINARY = [
32: (4)                uf for uf in UFUNCS if uf.nin == 2
33: (0)            ]
34: (0)            UFUNCS_BINARY_ACC = [
35: (4)                uf for uf in UFUNCS_BINARY if hasattr(uf, "accumulate") and uf.nout == 1
36: (0)            ]
37: (0)            def interesting_binop_operands(val1, val2, dtype):
38: (4)                """
39: (4)                    Helper to create "interesting" operands to cover common code paths:
40: (4)                    * scalar inputs
41: (4)                    * only first "values" is an array (e.g. scalar division fast-paths)
42: (4)                    * Longer array (SIMD) placing the value of interest at different positions

```

```

43: (4) * Oddly strided arrays which may not be SIMD compatible
44: (4) It does not attempt to cover unaligned access or mixed dtypes.
45: (4) These are normally handled by the casting/buffering machinery.
46: (4) This is not a fixture (currently), since I believe a fixture normally
47: (4) only yields once?
48: (4) """
49: (4)     fill_value = 1 # could be a parameter, but maybe not an optional one?
50: (4)     arr1 = np.full(10003, dtype=dtype, fill_value=fill_value)
51: (4)     arr2 = np.full(10003, dtype=dtype, fill_value=fill_value)
52: (4)     arr1[0] = val1
53: (4)     arr2[0] = val2
54: (4)     extractor = lambda res: res
55: (4)     yield arr1[0], arr2[0], extractor, "scalars"
56: (4)     extractor = lambda res: res
57: (4)     yield arr1[0, ...], arr2[0, ...], extractor, "scalar-arrays"
58: (4)     arr1[0] = fill_value
59: (4)     arr2[0] = fill_value
60: (4)     for pos in [0, 1, 2, 3, 4, 5, -1, -2, -3, -4]:
61: (8)         arr1[pos] = val1
62: (8)         arr2[pos] = val2
63: (8)         extractor = lambda res: res[pos]
64: (8)         yield arr1, arr2, extractor, f"off-{pos}"
65: (8)         yield arr1, arr2[pos], extractor, f"off-{pos}-with-scalar"
66: (8)         arr1[pos] = fill_value
67: (8)         arr2[pos] = fill_value
68: (4)     for stride in [-1, 113]:
69: (8)         op1 = arr1[::-stride]
70: (8)         op2 = arr2[::-stride]
71: (8)         op1[10] = val1
72: (8)         op2[10] = val2
73: (8)         extractor = lambda res: res[10]
74: (8)         yield op1, op2, extractor, f"stride-{stride}"
75: (8)         op1[10] = fill_value
76: (8)         op2[10] = fill_value
77: (0)     def on_powerpc():
78: (4)         """ True if we are running on a Power PC platform."""
79: (4)         return platform.processor() == 'powerpc' or \
80: (11)             platform.machine().startswith('ppc')
81: (0)     def bad_arcsinh():
82: (4)         """The blocklisted trig functions are not accurate on aarch64/PPC for
83: (4)         complex256. Rather than dig through the actual problem skip the
84: (4)         test. This should be fixed when we can move past glibc2.17
85: (4)         which is the version in manylinux2014
86: (4)         """
87: (4)         if platform.machine() == 'aarch64':
88: (8)             x = 1.78e-10
89: (4)         elif on_powerpc():
90: (8)             x = 2.16e-10
91: (4)         else:
92: (8)             return False
93: (4)         v1 = np.arcsinh(np.float128(x))
94: (4)         v2 = np.arcsinh(np.complex256(x)).real
95: (4)         return abs((v1 / v2) - 1.0) > 1e-23
96: (0)     class _FilterInvalids:
97: (4)         def setup_method(self):
98: (8)             self.olderr = np.seterr(invalid='ignore')
99: (4)         def teardown_method(self):
100: (8)             np.seterr(**self.olderr)
101: (0)     class TestConstants:
102: (4)         def test_pi(self):
103: (8)             assert_allclose(ncu.pi, 3.141592653589793, 1e-15)
104: (4)         def test_e(self):
105: (8)             assert_allclose(ncu.e, 2.718281828459045, 1e-15)
106: (4)         def test_euler_gamma(self):
107: (8)             assert_allclose(ncu.euler_gamma, 0.5772156649015329, 1e-15)
108: (0)     class TestOut:
109: (4)         def test_out_subok(self):
110: (8)             for subok in (True, False):
111: (12)                 a = np.array(0.5)

```

```

112: (12)          o = np.empty(())
113: (12)          r = np.add(a, 2, o, subok=subok)
114: (12)          assert_(r is o)
115: (12)          r = np.add(a, 2, out=o, subok=subok)
116: (12)          assert_(r is o)
117: (12)          r = np.add(a, 2, out=(o,), subok=subok)
118: (12)          assert_(r is o)
119: (12)          d = np.array(5.7)
120: (12)          o1 = np.empty(())
121: (12)          o2 = np.empty(), dtype=np.int32)
122: (12)          r1, r2 = np.frexp(d, o1, None, subok=subok)
123: (12)          assert_(r1 is o1)
124: (12)          r1, r2 = np.frexp(d, None, o2, subok=subok)
125: (12)          assert_(r2 is o2)
126: (12)          r1, r2 = np.frexp(d, o1, o2, subok=subok)
127: (12)          assert_(r1 is o1)
128: (12)          assert_(r2 is o2)
129: (12)          r1, r2 = np.frexp(d, out=(o1, None), subok=subok)
130: (12)          assert_(r1 is o1)
131: (12)          r1, r2 = np.frexp(d, out=(None, o2), subok=subok)
132: (12)          assert_(r2 is o2)
133: (12)          r1, r2 = np.frexp(d, out=(o1, o2), subok=subok)
134: (12)          assert_(r1 is o1)
135: (12)          assert_(r2 is o2)
136: (12)          with assert_raises(TypeError):
137: (16)            r1, r2 = np.frexp(d, out=o1, subok=subok)
138: (12)          assert_raises(TypeError, np.add, a, 2, o, o, subok=subok)
139: (12)          assert_raises(TypeError, np.add, a, 2, o, out=o, subok=subok)
140: (12)          assert_raises(TypeError, np.add, a, 2, None, out=o, subok=subok)
141: (12)          assert_raises(ValueError, np.add, a, 2, out=(o, o), subok=subok)
142: (12)          assert_raises(ValueError, np.add, a, 2, out=(), subok=subok)
143: (12)          assert_raises(TypeError, np.add, a, 2, [], subok=subok)
144: (12)          assert_raises(TypeError, np.add, a, 2, out=[], subok=subok)
145: (12)          assert_raises(TypeError, np.add, a, 2, out=([],), subok=subok)
146: (12)          o.flags.writeable = False
147: (12)          assert_raises(ValueError, np.add, a, 2, o, subok=subok)
148: (12)          assert_raises(ValueError, np.add, a, 2, out=o, subok=subok)
149: (12)          assert_raises(ValueError, np.add, a, 2, out=(o,), subok=subok)
150: (4)           def test_out_wrap_subok(self):
151: (8)             class ArrayWrap(np.ndarray):
152: (12)               __array_priority__ = 10
153: (12)               def __new__(cls, arr):
154: (16)                 return np.asarray(arr).view(cls).copy()
155: (12)               def __array_wrap__(self, arr, context):
156: (16)                 return arr.view(type(self))
157: (8)               for subok in (True, False):
158: (12)                 a = ArrayWrap([0.5])
159: (12)                 r = np.add(a, 2, subok=subok)
160: (12)                 if subok:
161: (16)                   assert_(isinstance(r, ArrayWrap))
162: (12)                 else:
163: (16)                   assert_(type(r) == np.ndarray)
164: (12)                 r = np.add(a, 2, None, subok=subok)
165: (12)                 if subok:
166: (16)                   assert_(isinstance(r, ArrayWrap))
167: (12)                 else:
168: (16)                   assert_(type(r) == np.ndarray)
169: (12)                 r = np.add(a, 2, out=None, subok=subok)
170: (12)                 if subok:
171: (16)                   assert_(isinstance(r, ArrayWrap))
172: (12)                 else:
173: (16)                   assert_(type(r) == np.ndarray)
174: (12)                 r = np.add(a, 2, out=(None,), subok=subok)
175: (12)                 if subok:
176: (16)                   assert_(isinstance(r, ArrayWrap))
177: (12)                 else:
178: (16)                   assert_(type(r) == np.ndarray)
179: (12)                 d = ArrayWrap([5.7])
180: (12)                 o1 = np.empty((1,))

```

```

181: (12)          o2 = np.empty((1,), dtype=np.int32)
182: (12)          r1, r2 = np.frexp(d, o1, subok=subok)
183: (12)          if subok:
184: (16)              assert_(isinstance(r2, ArrayWrap))
185: (12)          else:
186: (16)              assert_(type(r2) == np.ndarray)
187: (12)          r1, r2 = np.frexp(d, o1, None, subok=subok)
188: (12)          if subok:
189: (16)              assert_(isinstance(r2, ArrayWrap))
190: (12)          else:
191: (16)              assert_(type(r2) == np.ndarray)
192: (12)          r1, r2 = np.frexp(d, None, o2, subok=subok)
193: (12)          if subok:
194: (16)              assert_(isinstance(r1, ArrayWrap))
195: (12)          else:
196: (16)              assert_(type(r1) == np.ndarray)
197: (12)          r1, r2 = np.frexp(d, out=(o1, None), subok=subok)
198: (12)          if subok:
199: (16)              assert_(isinstance(r2, ArrayWrap))
200: (12)          else:
201: (16)              assert_(type(r2) == np.ndarray)
202: (12)          r1, r2 = np.frexp(d, out=(None, o2), subok=subok)
203: (12)          if subok:
204: (16)              assert_(isinstance(r1, ArrayWrap))
205: (12)          else:
206: (16)              assert_(type(r1) == np.ndarray)
207: (12)          with assert_raises(TypeError):
208: (16)              r1, r2 = np.frexp(d, out=o1, subok=subok)
209: (0)          class TestComparisons:
210: (4)              import operator
211: (4)              @pytest.mark.parametrize('dtype', np.sctypes['uint'] + np.sctypes['int'] +
212: (29)                                np.sctypes['float'] + [np.bool_])
213: (4)              @pytest.mark.parametrize('py_comp,np_comp', [
214: (8)                  (operator.lt, np.less),
215: (8)                  (operator.le, np.less_equal),
216: (8)                  (operator.gt, np.greater),
217: (8)                  (operator.ge, np.greater_equal),
218: (8)                  (operator.eq, np.equal),
219: (8)                  (operator.ne, np.not_equal)
220: (4)          ])
221: (4)          def test_comparison_functions(self, dtype, py_comp, np_comp):
222: (8)              if dtype == np.bool_:
223: (12)                  a = np.random.choice(a=[False, True], size=1000)
224: (12)                  b = np.random.choice(a=[False, True], size=1000)
225: (12)                  scalar = True
226: (8)              else:
227: (12)                  a = np.random.randint(low=1, high=10, size=1000).astype(dtype)
228: (12)                  b = np.random.randint(low=1, high=10, size=1000).astype(dtype)
229: (12)                  scalar = 5
230: (8)              np_scalar = np.dtype(dtype).type(scalar)
231: (8)              a_lst = a.tolist()
232: (8)              b_lst = b.tolist()
233: (8)              comp_b = np_comp(a, b).view(np.uint8)
234: (8)              comp_b_list = [int(py_comp(x, y)) for x, y in zip(a_lst, b_lst)]
235: (8)              comp_s1 = np_comp(np_scalar, b).view(np.uint8)
236: (8)              comp_s1_list = [int(py_comp(scalar, x)) for x in b_lst]
237: (8)              comp_s2 = np_comp(a, np_scalar).view(np.uint8)
238: (8)              comp_s2_list = [int(py_comp(x, scalar)) for x in a_lst]
239: (8)              assert_(comp_b.tolist() == comp_b_list,
240: (12)                  f"Failed comparison ({py_comp.__name__})")
241: (8)              assert_(comp_s1.tolist() == comp_s1_list,
242: (12)                  f"Failed comparison ({py_comp.__name__})")
243: (8)              assert_(comp_s2.tolist() == comp_s2_list,
244: (12)                  f"Failed comparison ({py_comp.__name__})")
245: (4)          def test_ignore_object_identity_in_equal(self):
246: (8)              a = np.array([np.array([1, 2, 3]), None], dtype=object)
247: (8)              assert_raises(ValueError, np.equal, a, a)
248: (8)              class FunkyType:
249: (12)                  def __eq__(self, other):

```

```

250: (16)                     raise TypeError("I won't compare")
251: (8)                      a = np.array([FunkyType()])
252: (8)                      assert_raises(TypeError, np.equal, a, a)
253: (8)                      a = np.array([np.nan], dtype=object)
254: (8)                      assert_equal(np.equal(a, a), [False])
255: (4)  def test_ignore_object_identity_in_not_equal(self):
256: (8)    a = np.array([np.array([1, 2, 3]), None], dtype=object)
257: (8)    assert_raises(ValueError, np.not_equal, a, a)
258: (8)  class FunkyType:
259: (12)    def __ne__(self, other):
260: (16)      raise TypeError("I won't compare")
261: (8)      a = np.array([FunkyType()])
262: (8)      assert_raises(TypeError, np.not_equal, a, a)
263: (8)      a = np.array([np.nan], dtype=object)
264: (8)      assert_equal(np.not_equal(a, a), [True])
265: (4)  def test_error_in_equal_reduce(self):
266: (8)    a = np.array([0, 0])
267: (8)    assert_equal(np.equal.reduce(a, dtype=bool), True)
268: (8)    assert_raises(TypeError, np.equal.reduce, a)
269: (4)  def test_object_dtype(self):
270: (8)    assert np.equal(1, [1], dtype=object).dtype == object
271: (8)    assert np.equal(1, [1], signature=(None, None, "O")).dtype == object
272: (4)  def test_object_nonbool_dtype_error(self):
273: (8)    assert np.equal(1, [1], dtype=bool).dtype == bool
274: (8)    with pytest.raises(TypeError, match="No loop matching"):
275: (12)      np.equal(1, 1, dtype=np.int64)
276: (8)    with pytest.raises(TypeError, match="No loop matching"):
277: (12)      np.equal(1, 1, sig=(None, None, "1"))
278: (4)  @pytest.mark.parametrize("dtypes", ["QQ", "Qq"])
279: (4)  @pytest.mark.parametrize('py_comp, np_comp', [
280: (8)    (operator.lt, np.less),
281: (8)    (operator.le, np.less_equal),
282: (8)    (operator.gt, np.greater),
283: (8)    (operator.ge, np.greater_equal),
284: (8)    (operator.eq, np.equal),
285: (8)    (operator.ne, np.not_equal)
286: (4)  ])
287: (4)  @pytest.mark.parametrize("vals", [(2**60, 2**60+1), (2**60+1, 2**60)])
288: (4)  def test_large_integer_direct_comparison(
289: (12)    self, dtypes, py_comp, np_comp, vals):
290: (8)    a1 = np.array([2**60], dtype=dtypes[0])
291: (8)    a2 = np.array([2**60 + 1], dtype=dtypes[1])
292: (8)    expected = py_comp(2**60, 2**60+1)
293: (8)    assert py_comp(a1, a2) == expected
294: (8)    assert np_comp(a1, a2) == expected
295: (8)    s1 = a1[0]
296: (8)    s2 = a2[0]
297: (8)    assert isinstance(s1, np.integer)
298: (8)    assert isinstance(s2, np.integer)
299: (8)    assert py_comp(s1, s2) == expected
300: (8)    assert np_comp(s1, s2) == expected
301: (4)  @pytest.mark.parametrize("dtype", np.typecodes['UnsignedInteger'])
302: (4)  @pytest.mark.parametrize('py_comp_func, np_comp_func', [
303: (8)    (operator.lt, np.less),
304: (8)    (operator.le, np.less_equal),
305: (8)    (operator.gt, np.greater),
306: (8)    (operator.ge, np.greater_equal),
307: (8)    (operator.eq, np.equal),
308: (8)    (operator.ne, np.not_equal)
309: (4)  ])
310: (4)  @pytest.mark.parametrize("flip", [True, False])
311: (4)  def test_unsigned_signed_direct_comparison(
312: (12)    self, dtype, py_comp_func, np_comp_func, flip):
313: (8)    if flip:
314: (12)      py_comp = lambda x, y: py_comp_func(y, x)
315: (12)      np_comp = lambda x, y: np_comp_func(y, x)
316: (8)    else:
317: (12)      py_comp = py_comp_func
318: (12)      np_comp = np_comp_func

```

```

319: (8) arr = np.array([np.iinfo(dtype).max], dtype=dtype)
320: (8) expected = py_comp(int(arr[0]), -1)
321: (8) assert py_comp(arr, -1) == expected
322: (8) assert np_comp(arr, -1) == expected
323: (8) scalar = arr[0]
324: (8) assert isinstance(scalar, np.integer)
325: (8) assert py_comp(scalar, -1) == expected
326: (8) assert np_comp(scalar, -1) == expected
327: (0) class TestAdd:
328: (4)     def test_reduce_alignment(self):
329: (8)         a = np.zeros(2, dtype=[('a', np.int32), ('b', np.float64)])
330: (8)         a['a'] = -1
331: (8)         assert_equal(a['b'].sum(), 0)
332: (0) class TestDivision:
333: (4)     def test_division_int(self):
334: (8)         x = np.array([5, 10, 90, 100, -5, -10, -90, -100, -120])
335: (8)         if 5 / 10 == 0.5:
336: (12)             assert_equal(x / 100, [0.05, 0.1, 0.9, 1,
337: (35)                             -0.05, -0.1, -0.9, -1, -1.2])
338: (8)         else:
339: (12)             assert_equal(x / 100, [0, 0, 0, 1, -1, -1, -1, -2])
340: (8)             assert_equal(x // 100, [0, 0, 0, 1, -1, -1, -1, -2])
341: (8)             assert_equal(x % 100, [5, 10, 90, 0, 95, 90, 10, 0, 80])
342: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
343: (4) @pytest.mark.parametrize("dtype,ex_val", itertools.product(
344: (8)     np.sctypes['int'] + np.sctypes['uint'], (
345: (12)         (
346: (16)             "np.array(range(fo.max-lsize, fo.max)).astype(dtype)",
347: (16)             "np.arange(lsize).astype(dtype)",
348: (16)             "range(15)"
349: (12)         ),
350: (12)         (
351: (16)             "np.arange(fo.min, fo.min+lsize).astype(dtype)",
352: (16)             "np.arange(lsize//2, lsize//2).astype(dtype)",
353: (16)             "range(fo.min, fo.min + 15)"
354: (12)         ),
355: (16)             "np.array(range(fo.max-lsize, fo.max)).astype(dtype)",
356: (16)             "np.arange(lsize).astype(dtype)",
357: (16)             "[1,3,9,13,neg, fo.min+1, fo.min//2, fo.max//3, fo.max//4]"
358: (12)         )
359: (8)     )
360: (4) )
361: (4)     def test_division_int_boundary(self, dtype, ex_val):
362: (8)         fo = np.iinfo(dtype)
363: (8)         neg = -1 if fo.min < 0 else 1
364: (8)         lsize = 512 + 7
365: (8)         a, b, divisors = eval(ex_val)
366: (8)         a_lst, b_lst = a.tolist(), b.tolist()
367: (8)         c_div = lambda n, d: (
368: (12)             0 if d == 0 else (
369: (16)                 fo.min if (n and n == fo.min and d == -1) else n//d
370: (12)             )
371: (8)         )
372: (8)         with np.errstate(divide='ignore'):
373: (12)             ac = a.copy()
374: (12)             ac //= b
375: (12)             div_ab = a // b
376: (8)             div_lst = [c_div(x, y) for x, y in zip(a_lst, b_lst)]
377: (8)             msg = "Integer arrays floor division check (//)"
378: (8)             assert all(div_ab == div_lst), msg
379: (8)             msg_eq = "Integer arrays floor division check (//=)"
380: (8)             assert all(ac == div_lst), msg_eq
381: (8)             for divisor in divisors:
382: (12)                 ac = a.copy()
383: (12)                 with np.errstate(divide='ignore', over='ignore'):
384: (16)                     div_a = a // divisor
385: (16)                     ac //÷ divisor
386: (12)                     div_lst = [c_div(i, divisor) for i in a_lst]
387: (12)                     assert all(div_a == div_lst), msg

```

```

388: (12)             assert all(ac == div_lst), msg_eq
389: (8)              with np.errstate(divide='raise', over='raise'):
390: (12)                if 0 in b:
391: (16)                  with pytest.raises(FloatingPointError,
392: (24)                      match="divide by zero encountered in floor_divide"):
393: (20)                      a // b
394: (12)                else:
395: (16)                  a // b
396: (12)                if fo.min and fo.min in a:
397: (16)                  with pytest.raises(FloatingPointError,
398: (24)                      match='overflow encountered in floor_divide'):
399: (20)                      a // -1
400: (12)                elif fo.min:
401: (16)                  a // -1
402: (12)                with pytest.raises(FloatingPointError,
403: (20)                      match="divide by zero encountered in floor_divide"):
404: (16)                      a // 0
405: (12)                with pytest.raises(FloatingPointError,
406: (20)                      match="divide by zero encountered in floor_divide"):
407: (16)                      ac = a.copy()
408: (16)                      ac //=_0
409: (12)                      np.array([], dtype=dtype) // 0
410: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
411: (4) @pytest.mark.parametrize("dtype,ex_val", itertools.product(
412: (8)    np.sctypes['int'] + np.sctypes['uint'], (
413: (12)      "np.array([fo.max, 1, 2, 1, 1, 2, 3], dtype=dtype)",
414: (12)      "np.array([fo.min, 1, -2, 1, 1, 2, -3]).astype(dtype)",
415: (12)      "np.arange(fo.min, fo.min+(100*10), 10, dtype=dtype)",
416: (12)      "np.array(range(fo.max-(100*7), fo.max, 7)).astype(dtype)",
417: (8)
418: (4)
419: (4)        ))
420: (8)      def test_division_int_reduce(self, dtype, ex_val):
421: (8)        fo = np.iinfo(dtype)
422: (8)        a = eval(ex_val)
423: (8)        lst = a.tolist()
424: (12)        c_div = lambda n, d: (
425: (8)          0 if d == 0 or (n and n == fo.min and d == -1) else n//d
426: (8)
427: (12)        with np.errstate(divide='ignore'):
428: (8)          div_a = np.floor_divide.reduce(a)
429: (8)          div_lst = reduce(c_div, lst)
430: (8)          msg = "Reduce floor integer division check"
431: (8)          assert div_a == div_lst, msg
432: (12)          with np.errstate(divide='raise', over='raise'):
433: (20)            with pytest.raises(FloatingPointError,
434: (16)                match="divide by zero encountered in reduce"):
435: (12)                np.floor_divide.reduce(np.arange(-100, 100).astype(dtype))
436: (16)            if fo.min:
437: (24)              with pytest.raises(FloatingPointError,
438: (20)                  match='overflow encountered in reduce'):
439: (24)                  np.floor_divide.reduce(
440: (20)                      np.array([fo.min, 1, -1], dtype=dtype)
441: (4)
442: (12)            @pytest.mark.parametrize(
443: (12)              "dividend,divisor,quotient",
444: (13)              [(np.timedelta64(2,'Y'), np.timedelta64(2,'M'), 12),
445: (13)                (np.timedelta64(2,'Y'), np.timedelta64(-2,'M'), -12),
446: (13)                (np.timedelta64(-2,'Y'), np.timedelta64(2,'M'), -12),
447: (13)                (np.timedelta64(-2,'Y'), np.timedelta64(-2,'M'), 12),
448: (13)                (np.timedelta64(2,'M'), np.timedelta64(-2,'Y'), -1),
449: (13)                (np.timedelta64(2,'Y'), np.timedelta64(0,'M'), 0),
450: (13)                (np.timedelta64(2,'Y'), 2, np.timedelta64(1,'Y')),
451: (13)                (np.timedelta64(2,'Y'), -2, np.timedelta64(-1,'Y')),
452: (13)                (np.timedelta64(-2,'Y'), 2, np.timedelta64(-1,'Y')),
453: (13)                (np.timedelta64(-2,'Y'), -2, np.timedelta64(1,'Y')),
454: (13)                (np.timedelta64(-2,'Y'), -3, np.timedelta64(0,'Y')),
455: (13)                (np.timedelta64(-2,'Y'), 0, np.timedelta64('Nat','Y'))),
456: (12)

```

```

457: (4) def test_division_int_timedelta(self, dividend, divisor, quotient):
458: (8)     if divisor and (isinstance(quotient, int) or not np.isnat(quotient)):
459: (12)         msg = "Timedelta floor division check"
460: (12)         assert dividend // divisor == quotient, msg
461: (12)         msg = "Timedelta arrays floor division check"
462: (12)         dividend_array = np.array([dividend]*5)
463: (12)         quotient_array = np.array([quotient]*5)
464: (12)         assert all(dividend_array // divisor == quotient_array), msg
465: (8)     else:
466: (12)         if IS_WASM:
467: (16)             pytest.skip("fp errors don't work in wasm")
468: (12)             with np.errstate(divide='raise', invalid='raise'):
469: (16)                 with pytest.raises(FloatingPointError):
470: (20)                     dividend // divisor
471: (4) def test_division_complex(self):
472: (8)     msg = "Complex division implementation check"
473: (8)     x = np.array([1. + 1.*1j, 1. + .5*1j, 1. + 2.*1j],
474: (8)         dtype=np.complex128)
475: (8)     assert_almost_equal(x**2/x, x, err_msg=msg)
476: (8)     msg = "Complex division overflow/underflow check"
477: (8)     x = np.array([1.e+110, 1.e-110], dtype=np.complex128)
478: (8)     y = x**2/x
479: (8)     assert_almost_equal(y/x, [1, 1], err_msg=msg)
480: (4) def test_zero_division_complex(self):
481: (8)     with np.errstate(invalid="ignore", divide="ignore"):
482: (12)         x = np.array([0.0], dtype=np.complex128)
483: (12)         y = 1.0/x
484: (12)         assert_(np.isinf(y)[0])
485: (12)         y = complex(np.inf, np.nan)/x
486: (12)         assert_(np.isinf(y)[0])
487: (12)         y = complex(np.nan, np.inf)/x
488: (12)         assert_(np.isinf(y)[0])
489: (12)         y = complex(np.inf, np.inf)/x
490: (12)         assert_(np.isinf(y)[0])
491: (12)         y = 0.0/x
492: (12)         assert_(np.isnan(y)[0])
493: (4) def test_floor_division_complex(self):
494: (8)     x = np.array([.9 + 1j, -.1 + 1j, .9 + .5*1j, .9 + 2.*1j],
495: (8)         dtype=np.complex128)
496: (8)     with pytest.raises(TypeError):
497: (12)         x // 7
498: (8)     with pytest.raises(TypeError):
499: (12)         np.divmod(x, 7)
500: (4)     with pytest.raises(TypeError):
501: (8)         np.remainder(x, 7)
502: (8) def test_floor_division_signed_zero(self):
503: (8)     x = np.zeros(10)
504: (8)     assert_equal(np.signbit(x//1), 0)
505: (12)     assert_equal(np.signbit((-x)//1), 1)
506: (4) @pytest.mark.skipif(hasattr(np.__config__, "blas_ss12_info"),
507: (12)     reason="gh-22982")
508: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
509: (4) @pytest.mark.parametrize('dtype', np.typecodes['Float'])
510: (4) def test_floor_division_errors(self, dtype):
511: (8)     fnan = np.array(np.nan, dtype=dtype)
512: (8)     fone = np.array(1.0, dtype=dtype)
513: (8)     fzzer = np.array(0.0, dtype=dtype)
514: (12)     finf = np.array(np.inf, dtype=dtype)
515: (8)     with np.errstate(divide='raise', invalid='ignore'):
516: (12)         assert_raises(FloatingPointError, np.floor_divide, fone, fzzer)
517: (8)     with np.errstate(divide='ignore', invalid='raise'):
518: (12)         np.floor_divide(fone, fzzer)
519: (8)     with np.errstate(all='raise'):
520: (12)         np.floor_divide(fnan, fone)
521: (12)         np.floor_divide(fone, fnan)
522: (12)         np.floor_divide(fnan, fzzer)
523: (12)         np.floor_divide(fzzer, fnan)
524: (4) @pytest.mark.parametrize('dtype', np.typecodes['Float'])
525: (4) def test_floor_division_corner_cases(self, dtype):

```

```

524: (8)           x = np.zeros(10, dtype=dtype)
525: (8)           y = np.ones(10, dtype=dtype)
526: (8)           fnan = np.array(np.nan, dtype=dtype)
527: (8)           fone = np.array(1.0, dtype=dtype)
528: (8)           fzr = np.array(0.0, dtype=dtype)
529: (8)           finf = np.array(np.inf, dtype=dtype)
530: (8)           with suppress_warnings() as sup:
531: (12)             sup.filter(RuntimeWarning, "invalid value encountered in
      floor_divide")
532: (12)             div = np.floor_divide(fnan, fone)
533: (12)             assert(np.isnan(div)), "dt: %s, div: %s" % (dt, div)
534: (12)             div = np.floor_divide(fone, fnan)
535: (12)             assert(np.isnan(div)), "dt: %s, div: %s" % (dt, div)
536: (12)             div = np.floor_divide(fnan, fzr)
537: (12)             assert(np.isnan(div)), "dt: %s, div: %s" % (dt, div)
538: (8)             with np.errstate(divide='ignore'):
539: (12)               z = np.floor_divide(y, x)
540: (12)               assert_(np.isinf(z).all())
541: (0)             def floor_divide_and_remainder(x, y):
542: (4)               return (np.floor_divide(x, y), np.remainder(x, y))
543: (0)             def _signs(dt):
544: (4)               if dt in np.typecodes['UnsignedInteger']:
545: (8)                 return (+1,)
546: (4)               else:
547: (8)                 return (+1, -1)
548: (0)             class TestRemainder:
549: (4)               def test_remainder_basic(self):
550: (8)                 dt = np.typecodes['AllInteger'] + np.typecodes['Float']
551: (8)                 for op in [floor_divide_and_remainder, np.divmod]:
552: (12)                   for dt1, dt2 in itertools.product(dt, dt):
553: (16)                     for sg1, sg2 in itertools.product(_signs(dt1), _signs(dt2)):
554: (20)                       fmt = 'op: %s, dt1: %s, dt2: %s, sg1: %s, sg2: %s'
555: (20)                       msg = fmt % (op.__name__, dt1, dt2, sg1, sg2)
556: (20)                       a = np.array(sg1*71, dtype=dt1)
557: (20)                       b = np.array(sg2*19, dtype=dt2)
558: (20)                       div, rem = op(a, b)
559: (20)                       assert_equal(div*b + rem, a, err_msg=msg)
560: (20)                       if sg2 == -1:
561: (24)                         assert_(b < rem <= 0, msg)
562: (20)                       else:
563: (24)                         assert_(b > rem >= 0, msg)
564: (4)             def test_float_remainder_exact(self):
565: (8)               nlst = list(range(-127, 0))
566: (8)               plst = list(range(1, 128))
567: (8)               dividend = nlst + [0] + plst
568: (8)               divisor = nlst + plst
569: (8)               arg = list(itertools.product(dividend, divisor))
570: (8)               tgt = list(divmod(*t) for t in arg)
571: (8)               a, b = np.array(arg, dtype=int).T
572: (8)               tgtdiv, tgtrem = np.array(tgt, dtype=float).T
573: (8)               tgtdiv = np.where((tgtdiv == 0.0) & ((b < 0) ^ (a < 0)), -0.0, tgtdiv)
574: (8)               tgtrem = np.where((tgtrem == 0.0) & (b < 0), -0.0, tgtrem)
575: (8)               for op in [floor_divide_and_remainder, np.divmod]:
576: (12)                 for dt in np.typecodes['Float']:
577: (16)                   msg = 'op: %s, dtype: %s' % (op.__name__, dt)
578: (16)                   fa = a.astype(dt)
579: (16)                   fb = b.astype(dt)
580: (16)                   div, rem = op(fa, fb)
581: (16)                   assert_equal(div, tgtdiv, err_msg=msg)
582: (16)                   assert_equal(rem, tgtrem, err_msg=msg)
583: (4)             def test_float_remainder_roundoff(self):
584: (8)               dt = np.typecodes['Float']
585: (8)               for op in [floor_divide_and_remainder, np.divmod]:
586: (12)                 for dt1, dt2 in itertools.product(dt, dt):
587: (16)                   for sg1, sg2 in itertools.product((-1, 1), (-1, 1)):
588: (20)                     fmt = 'op: %s, dt1: %s, dt2: %s, sg1: %s, sg2: %s'
589: (20)                     msg = fmt % (op.__name__, dt1, dt2, sg1, sg2)
590: (20)                     a = np.array(sg1*78*6e-8, dtype=dt1)
591: (20)                     b = np.array(sg2*6e-8, dtype=dt2)

```

```

592: (20)                                div, rem = op(a, b)
593: (20)                                assert_equal(div*b + rem, a, err_msg=msg)
594: (20)                                if sg2 == -1:
595: (24)                                    assert_(b < rem <= 0, msg)
596: (20)                                else:
597: (24)                                    assert_(b > rem >= 0, msg)
598: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
599: (4) @pytest.mark.xfail(sys.platform.startswith("darwin"),
600: (12)   reason="MacOS seems to not give the correct 'invalid' warning for
"
601: (19)   ``fmod``. Hopefully, others always do.")
602: (4) @pytest.mark.parametrize('dtype', np.typecodes['Float'])
603: (4) def test_float_divmod_errors(self, dtype):
604: (8)     fzero = np.array(0.0, dtype=dtype)
605: (8)     fone = np.array(1.0, dtype=dtype)
606: (8)     finf = np.array(np.inf, dtype=dtype)
607: (8)     fnan = np.array(np.nan, dtype=dtype)
608: (8)     with np.errstate(divide='raise', invalid='ignore'):
609: (12)         assert_raises(FloatingPointError, np.divmod, fone, fzero)
610: (8)     with np.errstate(divide='ignore', invalid='raise'):
611: (12)         assert_raises(FloatingPointError, np.divmod, fone, fzero)
612: (8)     with np.errstate(invalid='raise'):
613: (12)         assert_raises(FloatingPointError, np.divmod, fzero, fzero)
614: (8)     with np.errstate(invalid='raise'):
615: (12)         assert_raises(FloatingPointError, np.divmod, finf, finf)
616: (8)     with np.errstate(divide='ignore', invalid='raise'):
617: (12)         assert_raises(FloatingPointError, np.divmod, finf, fzero)
618: (8)     with np.errstate(divide='raise', invalid='ignore'):
619: (12)         np.divmod(finf, fzero)
620: (4) @pytest.mark.skipif(hasattr(np.__config__, "blas_ssl2_info"),
621: (12)   reason="gh-22982")
622: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
623: (4) @pytest.mark.xfail(sys.platform.startswith("darwin"),
624: (11)   reason="MacOS seems to not give the correct 'invalid' warning for "
"
625: (18)   ``fmod``. Hopefully, others always do.")
626: (4) @pytest.mark.parametrize('dtype', np.typecodes['Float'])
627: (4) @pytest.mark.parametrize('fn', [np.fmod, np.remainder])
628: (4) def test_float_remainder_errors(self, dtype, fn):
629: (8)     fzero = np.array(0.0, dtype=dtype)
630: (8)     fone = np.array(1.0, dtype=dtype)
631: (8)     finf = np.array(np.inf, dtype=dtype)
632: (8)     fnan = np.array(np.nan, dtype=dtype)
633: (8)     with np.errstate(all='raise'):
634: (12)         with pytest.raises(FloatingPointError,
635: (20)   match="invalid value"):
636: (16)             fn(fone, fzero)
637: (12)             fn(fnan, fzero)
638: (12)             fn(fzero, fnan)
639: (12)             fn(fone, fnan)
640: (12)             fn(fnan, fone)
641: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
642: (4) def test_float_remainder_overflow(self):
643: (8)     a = np.finfo(np.float64).tiny
644: (8)     with np.errstate(over='ignore', invalid='ignore'):
645: (12)         div, mod = np.divmod(4, a)
646: (12)         np.isinf(div)
647: (12)         assert_(mod == 0)
648: (8)     with np.errstate(over='raise', invalid='ignore'):
649: (12)         assert_raises(FloatingPointError, np.divmod, 4, a)
650: (8)     with np.errstate(invalid='raise', over='ignore'):
651: (12)         assert_raises(FloatingPointError, np.divmod, 4, a)
652: (4) def test_float_divmod_corner_cases(self):
653: (8)     for dt in np.typecodes['Float']:
654: (12)         fnan = np.array(np.nan, dtype=dt)
655: (12)         fone = np.array(1.0, dtype=dt)
656: (12)         fzer = np.array(0.0, dtype=dt)
657: (12)         finf = np.array(np.inf, dtype=dt)
658: (12)         with suppress_warnings() as sup:
659: (16)             sup.filter(RuntimeWarning, "invalid value encountered in

```

```

        divmod")
660: (16)                     sup.filter(RuntimeWarning, "divide by zero encountered in
        divmod")
661: (16)                     div, rem = np.divmod(fone, fzer)
662: (16)                     assert(np.isinf(div)), 'dt: %s, div: %s' % (dt, rem)
663: (16)                     assert(np.isnan(rem)), 'dt: %s, rem: %s' % (dt, rem)
664: (16)                     div, rem = np.divmod(fzer, fzer)
665: (16)                     assert(np.isnan(rem)), 'dt: %s, rem: %s' % (dt, rem)
666: (16)                     assert_(np.isnan(div)), 'dt: %s, rem: %s' % (dt, rem)
667: (16)                     div, rem = np.divmod(finf, finf)
668: (16)                     assert(np.isnan(div)), 'dt: %s, rem: %s' % (dt, rem)
669: (16)                     assert(np.isnan(rem)), 'dt: %s, rem: %s' % (dt, rem)
670: (16)                     div, rem = np.divmod(finf, fzer)
671: (16)                     assert(np.isinf(div)), 'dt: %s, rem: %s' % (dt, rem)
672: (16)                     assert(np.isnan(rem)), 'dt: %s, rem: %s' % (dt, rem)
673: (16)                     div, rem = np.divmod(fnan, fone)
674: (16)                     assert(np.isnan(rem)), "dt: %s, rem: %s" % (dt, rem)
675: (16)                     assert(np.isnan(div)), "dt: %s, rem: %s" % (dt, rem)
676: (16)                     div, rem = np.divmod(fone, fnan)
677: (16)                     assert(np.isnan(rem)), "dt: %s, rem: %s" % (dt, rem)
678: (16)                     assert(np.isnan(div)), "dt: %s, rem: %s" % (dt, rem)
679: (16)                     div, rem = np.divmod(fnan, fzer)
680: (16)                     assert(np.isnan(rem)), "dt: %s, rem: %s" % (dt, rem)
681: (16)                     assert(np.isnan(div)), "dt: %s, rem: %s" % (dt, rem)
682: (4)                      def test_float_remainder_corner_cases(self):
683: (8)                        for dt in np.typecodes['Float']:
684: (12)                          fone = np.array(1.0, dtype=dt)
685: (12)                          fzer = np.array(0.0, dtype=dt)
686: (12)                          fnan = np.array(np.nan, dtype=dt)
687: (12)                          b = np.array(1.0, dtype=dt)
688: (12)                          a = np.nextafter(np.array(0.0, dtype=dt), -b)
689: (12)                          rem = np.remainder(a, b)
690: (12)                          assert_(rem <= b, 'dt: %s' % dt)
691: (12)                          rem = np.remainder(-a, -b)
692: (12)                          assert_(rem >= -b, 'dt: %s' % dt)
693: (8)                        with suppress_warnings() as sup:
694: (12)                          sup.filter(RuntimeWarning, "invalid value encountered in
       remainder")
695: (12)                          sup.filter(RuntimeWarning, "invalid value encountered in fmod")
696: (12)                          for dt in np.typecodes['Float']:
697: (16)                            fone = np.array(1.0, dtype=dt)
698: (16)                            fzer = np.array(0.0, dtype=dt)
699: (16)                            finf = np.array(np.inf, dtype=dt)
700: (16)                            fnan = np.array(np.nan, dtype=dt)
701: (16)                            rem = np.remainder(fone, fzer)
702: (16)                            assert_(np.isnan(rem), 'dt: %s, rem: %s' % (dt, rem))
703: (16)                            rem = np.remainder(finf, fone)
704: (16)                            fmod = np.fmod(finf, fone)
705: (16)                            assert_(np.isnan(fmod), 'dt: %s, fmod: %s' % (dt, fmod))
706: (16)                            assert_(np.isnan(rem), 'dt: %s, rem: %s' % (dt, rem))
707: (16)                            rem = np.remainder(finf, finf)
708: (16)                            fmod = np.fmod(finf, fone)
709: (16)                            assert_(np.isnan(rem), 'dt: %s, rem: %s' % (dt, rem))
710: (16)                            assert_(np.isnan(fmod), 'dt: %s, fmod: %s' % (dt, fmod))
711: (16)                            rem = np.remainder(finf, fzer)
712: (16)                            fmod = np.fmod(finf, fzer)
713: (16)                            assert_(np.isnan(rem), 'dt: %s, rem: %s' % (dt, rem))
714: (16)                            assert_(np.isnan(fmod), 'dt: %s, fmod: %s' % (dt, fmod))
715: (16)                            rem = np.remainder(fone, fnan)
716: (16)                            fmod = np.fmod(fone, fnan)
717: (16)                            assert_(np.isnan(rem), 'dt: %s, rem: %s' % (dt, rem))
718: (16)                            assert_(np.isnan(fmod), 'dt: %s, fmod: %s' % (dt, fmod))
719: (16)                            rem = np.remainder(fnan, fzer)
720: (16)                            fmod = np.fmod(fnan, fzer)
721: (16)                            assert_(np.isnan(rem), 'dt: %s, rem: %s' % (dt, rem))
722: (16)                            assert_(np.isnan(fmod), 'dt: %s, fmod: %s' % (dt, fmod))
723: (16)                            rem = np.remainder(fnan, fone)
724: (16)                            fmod = np.fmod(fnan, fone)
725: (16)                            assert_(np.isnan(rem), 'dt: %s, rem: %s' % (dt, rem))

```

```

726: (16) assert_(np.isnan(fmod), 'dt: %s, fmod: %s' % (dt, rem))
727: (0) class TestDivisionIntegerOverflowsAndDivideByZero:
728: (4)     result_type = namedtuple('result_type',
729: (12)         ['nocast', 'casted'])
730: (4)     helper_lambdas = {
731: (8)         'zero': lambda dtype: 0,
732: (8)         'min': lambda dtype: np.iinfo(dtype).min,
733: (8)         'neg_min': lambda dtype: -np.iinfo(dtype).min,
734: (8)         'min-zero': lambda dtype: (np.iinfo(dtype).min, 0),
735: (8)         'neg_min-zero': lambda dtype: (-np.iinfo(dtype).min, 0),
736: (4)     }
737: (4)     overflow_results = {
738: (8)         np.remainder: result_type(
739: (12)             helper_lambdas['zero'], helper_lambdas['zero']),
740: (8)         np.fmod: result_type(
741: (12)             helper_lambdas['zero'], helper_lambdas['zero']),
742: (8)         operator.mod: result_type(
743: (12)             helper_lambdas['zero'], helper_lambdas['zero']),
744: (8)         operator.floordiv: result_type(
745: (12)             helper_lambdas['min'], helper_lambdas['neg_min']),
746: (8)         np.floor_divide: result_type(
747: (12)             helper_lambdas['min'], helper_lambdas['neg_min']),
748: (8)         np.divmod: result_type(
749: (12)             helper_lambdas['min-zero'], helper_lambdas['neg_min-zero'])
750: (4)     }
751: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
752: (4) @pytest.mark.parametrize("dtype", np.typecodes["Integer"])
753: (4) def test_signed_division_overflow(self, dtype):
754: (8)     to_check = interesting_binop_operands(np.iinfo(dtype).min, -1, dtype)
755: (8)     for op1, op2, extractor, operand_identifier in to_check:
756: (12)         with pytest.warns(RuntimeWarning, match="overflow encountered"):
757: (16)             res = op1 // op2
758: (12)             assert res.dtype == op1.dtype
759: (12)             assert extractor(res) == np.iinfo(op1.dtype).min
760: (12)             res = op1 % op2
761: (12)             assert res.dtype == op1.dtype
762: (12)             assert extractor(res) == 0
763: (12)             res = np.fmod(op1, op2)
764: (12)             assert extractor(res) == 0
765: (12)             with pytest.warns(RuntimeWarning, match="overflow encountered"):
766: (16)                 res1, res2 = np.divmod(op1, op2)
767: (12)                 assert res1.dtype == res2.dtype == op1.dtype
768: (12)                 assert extractor(res1) == np.iinfo(op1.dtype).min
769: (12)                 assert extractor(res2) == 0
770: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
771: (4) @pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
772: (4) def test_divide_by_zero(self, dtype):
773: (8)     to_check = interesting_binop_operands(1, 0, dtype)
774: (8)     for op1, op2, extractor, operand_identifier in to_check:
775: (12)         with pytest.warns(RuntimeWarning, match="divide by zero"):
776: (16)             res = op1 // op2
777: (12)             assert res.dtype == op1.dtype
778: (12)             assert extractor(res) == 0
779: (12)             with pytest.warns(RuntimeWarning, match="divide by zero"):
780: (16)                 res1, res2 = np.divmod(op1, op2)
781: (12)                 assert res1.dtype == res2.dtype == op1.dtype
782: (12)                 assert extractor(res1) == 0
783: (12)                 assert extractor(res2) == 0
784: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
785: (4) @pytest.mark.parametrize("dividend_dtype",
786: (12)         np.sctypes['int'])
787: (4) @pytest.mark.parametrize("divisor_dtype",
788: (12)         np.sctypes['int'])
789: (4) @pytest.mark.parametrize("operation",
790: (12)         [np.remainder, np.fmod, np.divmod, np.floor_divide,
791: (13)             operator.mod, operator.floordiv])
792: (4) @np.errstate(divide='warn', over='warn')
793: (4) def test_overflows(self, dividend_dtype, divisor_dtype, operation):
794: (8)     arrays = [np.array([np.iinfo(dividend_dtype).min]*i,
```

```

795: (27)                         dtype=dividend_dtype) for i in range(1, 129)]
796: (8)                          divisor = np.array([-1], dtype=divisor_dtype)
797: (8)                          if np.dtype(dividend_dtype).itemsize >= np.dtype(
798: (16)                            divisor_dtype).itemsize and operation in (
799: (24)                              np.divmod, np.floor_divide, operator.floordiv):
800: (12)                            with pytest.warns(
801: (20)                              RuntimeWarning,
802: (20)                              match="overflow encountered in"):
803: (16)                            result = operation(
804: (28)                              dividend_dtype(np.iinfo(dividend_dtype).min),
805: (28)                              divisor_dtype(-1)
806: (24)                            )
807: (16)                          assert result == self.overflow_results[operation].necast(
808: (24)                            dividend_dtype)
809: (12)                          for a in arrays:
810: (16)                            with pytest.warns(
811: (24)                              RuntimeWarning,
812: (24)                              match="overflow encountered in"):
813: (20)                            result = np.array(operation(a, divisor)).flatten('f')
814: (20)                            expected_array = np.array(
815: (28)                              [self.overflow_results[operation].necast(
816: (32)                                dividend_dtype)]*len(a)).flatten()
817: (20)                            assert_array_equal(result, expected_array)
818: (8)                          else:
819: (12)                            result = operation(
820: (24)                              dividend_dtype(np.iinfo(dividend_dtype).min),
821: (24)                              divisor_dtype(-1)
822: (20)                            )
823: (12)                          assert result == self.overflow_results[operation].casted(
824: (20)                            dividend_dtype)
825: (12)                          for a in arrays:
826: (16)                            result = np.array(operation(a, divisor)).flatten('f')
827: (16)                            expected_array = np.array(
828: (24)                              [self.overflow_results[operation].casted(
829: (28)                                dividend_dtype)]*len(a)).flatten()
830: (16)                            assert_array_equal(result, expected_array)
831: (0)                          class TestCbrt:
832: (4)                            def test_cbrt_scalar(self):
833: (8)                              assert_almost_equal(np.cbrt(np.float32(-2.5)**3)), -2.5)
834: (4)                            def test_cbrt(self):
835: (8)                              x = np.array([1., 2., -3., np.inf, -np.inf])
836: (8)                              assert_almost_equal(np.cbrt(x**3), x)
837: (8)                              assert_(np.isnan(np.cbrt(np.nan)))
838: (8)                              assert_equal(np.cbrt(np.inf), np.inf)
839: (8)                              assert_equal(np.cbrt(-np.inf), -np.inf)
840: (0)                          class TestPower:
841: (4)                            def test_power_float(self):
842: (8)                              x = np.array([1., 2., 3.])
843: (8)                              assert_equal(x**0, [1., 1., 1.])
844: (8)                              assert_equal(x**1, x)
845: (8)                              assert_equal(x**2, [1., 4., 9.])
846: (8)                              y = x.copy()
847: (8)                              y **= 2
848: (8)                              assert_equal(y, [1., 4., 9.])
849: (8)                              assert_almost_equal(x**(-1), [1., 0.5, 1./3])
850: (8)                              assert_almost_equal(x**(0.5), [1., ncu.sqrt(2), ncu.sqrt(3)])
851: (8)                              for out, inp, msg in _gen_alignment_data(dtype=np.float32,
852: (49)  type='unary',
853: (49)  max_size=11):
854: (12)                                exp = [ncu.sqrt(i) for i in inp]
855: (12)                                assert_almost_equal(inp***(0.5), exp, err_msg=msg)
856: (12)                                np.sqrt(inp, out=out)
857: (12)                                assert_equal(out, exp, err_msg=msg)
858: (8)                                for out, inp, msg in _gen_alignment_data(dtype=np.float64,
859: (49)  type='unary',
860: (49)  max_size=7):
861: (12)                                  exp = [ncu.sqrt(i) for i in inp]
862: (12)                                  assert_almost_equal(inp***(0.5), exp, err_msg=msg)
863: (12)                                  np.sqrt(inp, out=out)

```

```

864: (12) assert_equal(out, exp, err_msg=msg)
865: (4) def test_power_complex(self):
866: (8)     x = np.array([1+2j, 2+3j, 3+4j])
867: (8)     assert_equal(x**0, [1., 1., 1.])
868: (8)     assert_equal(x**1, x)
869: (8)     assert_almost_equal(x**2, [-3+4j, -5+12j, -7+24j])
870: (8)     assert_almost_equal(x**3, [(1+2j)**3, (2+3j)**3, (3+4j)**3])
871: (8)     assert_almost_equal(x**4, [(1+2j)**4, (2+3j)**4, (3+4j)**4])
872: (8)     assert_almost_equal(x**(-1), [1/(1+2j), 1/(2+3j), 1/(3+4j)])
873: (8)     assert_almost_equal(x**(-2), [1/(1+2j)**2, 1/(2+3j)**2, 1/(3+4j)**2])
874: (8)     assert_almost_equal(x**(-3), [(-11+2j)/125, (-46-9j)/2197,
875: (38)                                     (-117-44j)/15625])
876: (8)     assert_almost_equal(x**(0.5), [ncu.sqrt(1+2j), ncu.sqrt(2+3j),
877: (39)                                     ncu.sqrt(3+4j)])
878: (8)     norm = 1./((x**14)[0])
879: (8)     assert_almost_equal(x**14 * norm,
880: (16)         [i * norm for i in [-76443+16124j, 23161315+58317492j,
881: (36)             5583548873 + 2465133864j]])
882: (8)     def assert_complex_equal(x, y):
883: (12)         assert_array_equal(x.real, y.real)
884: (12)         assert_array_equal(x.imag, y.imag)
885: (8)     for z in [complex(0, np.inf), complex(1, np.inf)]:
886: (12)         z = np.array([z], dtype=np.complex_)
887: (12)         with np.errstate(invalid="ignore"):
888: (16)             assert_complex_equal(z**1, z)
889: (16)             assert_complex_equal(z**2, z*z)
890: (16)             assert_complex_equal(z**3, z*z*z)
891: (4)     def test_power_zero(self):
892: (8)         zero = np.array([0j])
893: (8)         one = np.array([1+0j])
894: (8)         cnan = np.array([complex(np.nan, np.nan)])
895: (8)     def assert_complex_equal(x, y):
896: (12)         x, y = np.asarray(x), np.asarray(y)
897: (12)         assert_array_equal(x.real, y.real)
898: (12)         assert_array_equal(x.imag, y.imag)
899: (8)     for p in [0.33, 0.5, 1, 1.5, 2, 3, 4, 5, 6.6]:
900: (12)         assert_complex_equal(np.power(zero, p), zero)
901: (8)     assert_complex_equal(np.power(zero, 0), one)
902: (8)     with np.errstate(invalid="ignore"):
903: (12)         assert_complex_equal(np.power(zero, 0+1j), cnan)
904: (12)         for p in [0.33, 0.5, 1, 1.5, 2, 3, 4, 5, 6.6]:
905: (16)             assert_complex_equal(np.power(zero, -p), cnan)
906: (12)             assert_complex_equal(np.power(zero, -1+0.2j), cnan)
907: (4)     @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
908: (4)     def test_zero_power_nonzero(self):
909: (8)         zero = np.array([0.0+0.0j])
910: (8)         cnan = np.array([complex(np.nan, np.nan)])
911: (8)     def assert_complex_equal(x, y):
912: (12)         assert_array_equal(x.real, y.real)
913: (12)         assert_array_equal(x.imag, y.imag)
914: (8)     assert_complex_equal(np.power(zero, 1+4j), zero)
915: (8)     assert_complex_equal(np.power(zero, 2-3j), zero)
916: (8)     assert_complex_equal(np.power(zero, 1+1j), zero)
917: (8)     assert_complex_equal(np.power(zero, 1+0j), zero)
918: (8)     assert_complex_equal(np.power(zero, 1-1j), zero)
919: (8)     with pytest.warns(expected_warning=RuntimeWarning) as r:
920: (12)         assert_complex_equal(np.power(zero, -1+1j), cnan)
921: (12)         assert_complex_equal(np.power(zero, -2-3j), cnan)
922: (12)         assert_complex_equal(np.power(zero, -7+0j), cnan)
923: (12)         assert_complex_equal(np.power(zero, 0+1j), cnan)
924: (12)         assert_complex_equal(np.power(zero, 0-1j), cnan)
925: (8)         assert len(r) == 5
926: (4)     def test_fast_power(self):
927: (8)         x = np.array([1, 2, 3], np.int16)
928: (8)         res = x**2.0
929: (8)         assert_(x**2.00001).dtype is res.dtype
930: (8)         assert_array_equal(res, [1, 4, 9])
931: (8)         assert_(not np.may_share_memory(res, x))
932: (8)         assert_array_equal(x, [1, 2, 3])

```

```

933: (8)             res = x ** np.array([[[2]]])
934: (8)             assert_equal(res.shape, (1, 1, 3))
935: (4)             def test_integer_power(self):
936: (8)                 a = np.array([15, 15], 'i8')
937: (8)                 b = np.power(a, a)
938: (8)                 assert_equal(b, [437893890380859375, 437893890380859375])
939: (4)             def test_integer_power_with_integer_zero_exponent(self):
940: (8)                 dtypes = np.typecodes['Integer']
941: (8)                 for dt in dtypes:
942: (12)                     arr = np.arange(-10, 10, dtype=dt)
943: (12)                     assert_equal(np.power(arr, 0), np.ones_like(arr))
944: (8)                 dtypes = np.typecodes['UnsignedInteger']
945: (8)                 for dt in dtypes:
946: (12)                     arr = np.arange(10, dtype=dt)
947: (12)                     assert_equal(np.power(arr, 0), np.ones_like(arr))
948: (4)             def test_integer_power_of_1(self):
949: (8)                 dtypes = np.typecodes['AllInteger']
950: (8)                 for dt in dtypes:
951: (12)                     arr = np.arange(10, dtype=dt)
952: (12)                     assert_equal(np.power(1, arr), np.ones_like(arr))
953: (4)             def test_integer_power_of_zero(self):
954: (8)                 dtypes = np.typecodes['AllInteger']
955: (8)                 for dt in dtypes:
956: (12)                     arr = np.arange(1, 10, dtype=dt)
957: (12)                     assert_equal(np.power(0, arr), np.zeros_like(arr))
958: (4)             def test_integer_to_negative_power(self):
959: (8)                 dtypes = np.typecodes['Integer']
960: (8)                 for dt in dtypes:
961: (12)                     a = np.array([0, 1, 2, 3], dtype=dt)
962: (12)                     b = np.array([0, 1, 2, -3], dtype=dt)
963: (12)                     one = np.array(1, dtype=dt)
964: (12)                     minusone = np.array(-1, dtype=dt)
965: (12)                     assert_raises(ValueError, np.power, a, b)
966: (12)                     assert_raises(ValueError, np.power, a, minusone)
967: (12)                     assert_raises(ValueError, np.power, one, b)
968: (12)                     assert_raises(ValueError, np.power, one, minusone)
969: (4)             def test_float_to_inf_power(self):
970: (8)                 for dt in [np.float32, np.float64]:
971: (12)                     a = np.array([1, 1, 2, 2, -2, -2, np.inf, -np.inf], dt)
972: (12)                     b = np.array([np.inf, -np.inf, np.inf, -np.inf,
973: (32)                                     np.inf, -np.inf, np.inf, -np.inf], dt)
974: (12)                     r = np.array([1, 1, np.inf, 0, np.inf, 0, np.inf, 0], dt)
975: (12)                     assert_equal(np.power(a, b), r)
976: (0)             class TestFloat_power:
977: (4)                 def test_type_conversion(self):
978: (8)                     arg_type = '?bhilBHILefdgFDG'
979: (8)                     res_type = 'dddddddddggDDG'
980: (8)                     for dtin, dtout in zip(arg_type, res_type):
981: (12)                         msg = "dtin: %s, dtout: %s" % (dtin, dtout)
982: (12)                         arg = np.ones(1, dtype=dtin)
983: (12)                         res = np.float_power(arg, arg)
984: (12)                         assert_(res.dtype.name == np.dtype(dtout).name, msg)
985: (0)             class TestLog2:
986: (4)                 @pytest.mark.parametrize('dt', ['f', 'd', 'g'])
987: (4)                 def test_log2_values(self, dt):
988: (8)                     x = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
989: (8)                     y = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
990: (8)                     xf = np.array(x, dtype=dt)
991: (8)                     yf = np.array(y, dtype=dt)
992: (8)                     assert_almost_equal(np.log2(xf), yf)
993: (4)                 @pytest.mark.parametrize("i", range(1, 65))
994: (4)                 def test_log2_ints(self, i):
995: (8)                     v = np.log2(2.**i)
996: (8)                     assert_equal(v, float(i), err_msg='at exponent %d' % i)
997: (4)                 @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
998: (4)                 def test_log2_special(self):
999: (8)                     assert_equal(np.log2(1.), 0.)
1000: (8)                    assert_equal(np.log2(np.inf), np.inf)
1001: (8)                    assert_(np.isnan(np.log2(np.nan)))

```

```

1002: (8)
1003: (12)
1004: (12)
1005: (12)
1006: (12)
1007: (12)
1008: (12)
1009: (12)
1010: (0)
1011: (4)
1012: (8)
1013: (8)
1014: (8)
1015: (12)
1016: (12)
1017: (12)
1018: (0)
1019: (4)
1020: (8)
1021: (8)
1022: (8)
1023: (8)
1024: (12)
1025: (12)
1026: (12)
1027: (12)
1028: (4)
1029: (8)
1030: (8)
1031: (8)
1032: (8)
1033: (12)
1034: (12)
1035: (12)
1036: (12)
1037: (4)
1038: (8)
1039: (8)
1040: (8)
1041: (8)
1042: (8)
1043: (12)
1044: (16)
1045: (16)
1046: (16)
1047: (16)
1048: (4)
1049: (8)
1050: (8)
1051: (8)
1052: (8)
1053: (8)
1054: (4)
1055: (8)
1056: (8)
1057: (8)
1058: (8)
1059: (0)
1060: (4)
1061: (8)
1062: (8)
1063: (8)
1064: (12)
1065: (12)
1066: (12)
1067: (12)
1068: (8)
1069: (8)
1070: (8)

        with warnings.catch_warnings(record=True) as w:
            warnings.filterwarnings('always', '', RuntimeWarning)
            assert_(np.isnan(np.log2(-1.)))
            assert_(np.isnan(np.log2(-np.inf)))
            assert_equal(np.log2(0.), -np.inf)
            assert_(w[0].category is RuntimeWarning)
            assert_(w[1].category is RuntimeWarning)
            assert_(w[2].category is RuntimeWarning)

    class TestExp2:
        def test_exp2_values(self):
            x = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
            y = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
            for dt in ['f', 'd', 'g']:
                xf = np.array(x, dtype=dt)
                yf = np.array(y, dtype=dt)
                assert_almost_equal(np.exp2(yf), xf)

    class TestLogAddExp2(_FilterInvalids):
        def test_logaddexp2_values(self):
            x = [1, 2, 3, 4, 5]
            y = [5, 4, 3, 2, 1]
            z = [6, 6, 6, 6, 6]
            for dt, dec_ in zip(['f', 'd', 'g'], [6, 15, 15]):
                xf = np.log2(np.array(x, dtype=dt))
                yf = np.log2(np.array(y, dtype=dt))
                zf = np.log2(np.array(z, dtype=dt))
                assert_almost_equal(np.logaddexp2(xf, yf), zf, decimal=dec_)

        def test_logaddexp2_range(self):
            x = [1000000, -1000000, 1000200, -1000200]
            y = [1000200, -1000200, 1000000, -1000000]
            z = [1000200, -1000000, 1000200, -1000000]
            for dt in ['f', 'd', 'g']:
                logxf = np.array(x, dtype=dt)
                logyf = np.array(y, dtype=dt)
                logzf = np.array(z, dtype=dt)
                assert_almost_equal(np.logaddexp2(logxf, logyf), logzf)

        def test_inf(self):
            inf = np.inf
            x = [inf, -inf, inf, -inf, inf, 1, -inf, 1]
            y = [inf, inf, -inf, -inf, 1, inf, 1, -inf]
            z = [inf, inf, inf, -inf, inf, inf, 1, 1]
            with np.errstate(invalid='raise'):
                for dt in ['f', 'd', 'g']:
                    logxf = np.array(x, dtype=dt)
                    logyf = np.array(y, dtype=dt)
                    logzf = np.array(z, dtype=dt)
                    assert_equal(np.logaddexp2(logxf, logyf), logzf)

        def test_nan(self):
            assert_(np.isnan(np.logaddexp2(np.nan, np.inf)))
            assert_(np.isnan(np.logaddexp2(np.inf, np.nan)))
            assert_(np.isnan(np.logaddexp2(np.nan, 0)))
            assert_(np.isnan(np.logaddexp2(0, np.nan)))
            assert_(np.isnan(np.logaddexp2(np.nan, np.nan)))

        def test_reduce(self):
            assert_equal(np.logaddexp2.identity, -np.inf)
            assert_equal(np.logaddexp2.reduce([]), -np.inf)
            assert_equal(np.logaddexp2.reduce([-np.inf]), -np.inf)
            assert_equal(np.logaddexp2.reduce([-np.inf, 0]), 0)

    class TestLog:
        def test_log_values(self):
            x = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
            y = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
            for dt in ['f', 'd', 'g']:
                log2_ = 0.69314718055994530943
                xf = np.array(x, dtype=dt)
                yf = np.array(y, dtype=dt)*log2_
                assert_almost_equal(np.log(xf), yf)
            x = np.array([2, 0.937500, 3, 0.947500, 1.054697])
            xf = np.log(x)
            assert_almost_equal(np.log(x, out=x), xf)

```

```

1071: (8)             for dt in ['f', 'd', 'g']:
1072: (12)             try:
1073: (16)                 with np.errstate(all='raise'):
1074: (20)                     x = np.finfo(dt).max
1075: (20)                     np.log(x)
1076: (12)             except FloatingPointError as exc:
1077: (16)                 if dt == 'g' and IS_MUSL:
1078: (20)                     pytest.skip(
1079: (24)                         "Overflow has occurred for"
1080: (24)                         " np.log(np.finfo(np.longdouble).max)")
1081: (20)                 )
1082: (16)             else:
1083: (20)                 raise exc
1084: (4)             def test_log_strides(self):
1085: (8)                 np.random.seed(42)
1086: (8)                 strides = np.array([-4,-3,-2,-1,1,2,3,4])
1087: (8)                 sizes = np.arange(2,100)
1088: (8)                 for ii in sizes:
1089: (12)                     x_f64 = np.float64(np.random.uniform(low=0.01,
high=100.0,size=ii))
1090: (12)                     x_special = x_f64.copy()
1091: (12)                     x_special[3:-1:4] = 1.0
1092: (12)                     y_true = np.log(x_f64)
1093: (12)                     y_special = np.log(x_special)
1094: (12)                     for jj in strides:
1095: (16)                         assert_array_almost_nulp(np.log(x_f64[:jj]),
y_true[:jj], nulp=2)
1096: (16)                         assert_array_almost_nulp(np.log(x_special[:jj]),
y_special[:jj], nulp=2)
1097: (0)             class TestExp:
1098: (4)                 def test_exp_values(self):
1099: (8)                     x = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
1100: (8)                     y = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
1101: (8)                     for dt in ['f', 'd', 'g']:
1102: (12)                         log2_ = 0.69314718055994530943
1103: (12)                         xf = np.array(x, dtype=dt)
1104: (12)                         yf = np.array(y, dtype=dt)*log2_
1105: (12)                         assert_almost_equal(np.exp(yf), xf)
1106: (4)                 def test_exp_strides(self):
1107: (8)                     np.random.seed(42)
1108: (8)                     strides = np.array([-4,-3,-2,-1,1,2,3,4])
1109: (8)                     sizes = np.arange(2,100)
1110: (8)                     for ii in sizes:
1111: (12)                         x_f64 = np.float64(np.random.uniform(low=0.01,
high=709.1,size=ii))
1112: (12)                         y_true = np.exp(x_f64)
1113: (12)                         for jj in strides:
1114: (16)                             assert_array_almost_nulp(np.exp(x_f64[:jj]),
y_true[:jj], nulp=2)
1115: (0)             class TestSpecialFloats:
1116: (4)                 def test_exp_values(self):
1117: (8)                     with np.errstate(under='raise', over='raise'):
1118: (12)                         x = [np.nan, np.nan, np.inf, 0.]
1119: (12)                         y = [np.nan, -np.nan, np.inf, -np.inf]
1120: (12)                         for dt in ['e', 'f', 'd', 'g']:
1121: (16)                             xf = np.array(x, dtype=dt)
1122: (16)                             yf = np.array(y, dtype=dt)
1123: (16)                             assert_equal(np.exp(yf), xf)
1124: (4)                 @pytest.mark.xfail(
1125: (8)                     _glibc_older_than("2.17"),
1126: (8)                     reason="Older glibc versions may not raise appropriate FP exceptions")
1127: (4)
1128: (4)                 def test_exp_exceptions(self):
1129: (8)                     with np.errstate(over='raise'):
1130: (12)                         assert_raises(FloatingPointError, np.exp, np.float16(11.0899))
1131: (12)                         assert_raises(FloatingPointError, np.exp, np.float32(100.))
1132: (12)                         assert_raises(FloatingPointError, np.exp, np.float32(1E19))
1133: (12)                         assert_raises(FloatingPointError, np.exp, np.float64(800.))
1134: (12)                         assert_raises(FloatingPointError, np.exp, np.float64(1E19))

```

```

1135: (8)
1136: (12)
1137: (12)
1138: (12)
1139: (12)
1140: (12)
1141: (4)
1142: (4)
1143: (8)
1144: (12)
1145: (12)
1146: (12)
1147: (12)
1148: (16)
1149: (16)
1150: (16)
1151: (16)
1152: (16)
1153: (16)
1154: (16)
1155: (8)
1156: (12)
1157: (16)
1158: (30)
1159: (16)
1160: (30)
1161: (16)
1162: (30)
1163: (16)
1164: (30)
1165: (8)
1166: (12)
1167: (16)
1168: (30)
1169: (16)
1170: (30)
1171: (16)
1172: (30)
1173: (16)
1174: (30)
1175: (16)
1176: (30)
1177: (16)
1178: (30)
1179: (16)
1180: (30)
1181: (16)
1182: (30)
1183: (8)
1184: (12)
1185: (12)
1186: (4)
1187: (4)
1188: (4)
1189: (8)
1190: (12)
1191: (12)
1192: (12)
1193: (12)
1194: (12)
1195: (12)
1196: (4)
1197: (4)
1198: (8)
1199: (8)
1200: (4)
1201: (4)
1202: (8)
1203: (12)

        with np.errstate(under='raise'):
            assert_raises(FloatingPointError, np.exp, np.float16(-17.5))
            assert_raises(FloatingPointError, np.exp, np.float32(-1000.))
            assert_raises(FloatingPointError, np.exp, np.float32(-1E19))
            assert_raises(FloatingPointError, np.exp, np.float64(-1000.))
            assert_raises(FloatingPointError, np.exp, np.float64(-1E19))

    @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
    def test_log_values(self):
        with np.errstate(all='ignore'):
            x = [np.nan, np.nan, np.inf, np.nan, -np.inf, np.nan]
            y = [np.nan, -np.nan, np.inf, -np.inf, 0.0, -1.0]
            y1p = [np.nan, -np.nan, np.inf, -np.inf, -1.0, -2.0]
            for dt in ['e', 'f', 'd', 'g']:
                xf = np.array(x, dtype=dt)
                yf = np.array(y, dtype=dt)
                yf1p = np.array(y1p, dtype=dt)
                assert_equal(np.log(yf), xf)
                assert_equal(np.log2(yf), xf)
                assert_equal(np.log10(yf), xf)
                assert_equal(np.log1p(yf1p), xf)

        with np.errstate(divide='raise'):
            for dt in ['e', 'f', 'd']:
                assert_raises(FloatingPointError, np.log,
                             np.array(0.0, dtype=dt))
                assert_raises(FloatingPointError, np.log2,
                             np.array(0.0, dtype=dt))
                assert_raises(FloatingPointError, np.log10,
                             np.array(0.0, dtype=dt))
                assert_raises(FloatingPointError, np.log1p,
                             np.array(-1.0, dtype=dt))

        with np.errstate(invalid='raise'):
            for dt in ['e', 'f', 'd']:
                assert_raises(FloatingPointError, np.log,
                             np.array(-np.inf, dtype=dt))
                assert_raises(FloatingPointError, np.log,
                             np.array(-1.0, dtype=dt))
                assert_raises(FloatingPointError, np.log2,
                             np.array(-np.inf, dtype=dt))
                assert_raises(FloatingPointError, np.log2,
                             np.array(-1.0, dtype=dt))
                assert_raises(FloatingPointError, np.log10,
                             np.array(-np.inf, dtype=dt))
                assert_raises(FloatingPointError, np.log10,
                             np.array(-1.0, dtype=dt))
                assert_raises(FloatingPointError, np.log1p,
                             np.array(-np.inf, dtype=dt))
                assert_raises(FloatingPointError, np.log1p,
                             np.array(-2.0, dtype=dt))

        with assert_no_warnings():
            a = np.array(1e9, dtype='float32')
            np.log(a)

    @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
    @pytest.mark.parametrize('dtype', ['e', 'f', 'd', 'g'])
    def test_sincos_values(self, dtype):
        with np.errstate(all='ignore'):
            x = [np.nan, np.nan, np.nan, np.nan]
            y = [np.nan, -np.nan, np.inf, -np.inf]
            xf = np.array(x, dtype=dtype)
            yf = np.array(y, dtype=dtype)
            assert_equal(np.sin(yf), xf)
            assert_equal(np.cos(yf), xf)

    @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
    @pytest.mark.xfail(
        sys.platform.startswith("darwin"),
        reason="underflow is triggered for scalar 'sin'"
    )
    def test_sincos_underflow(self):
        with np.errstate(under='raise'):
            underflow_trigger = np.array(

```

```

1204: (16)           float.fromhex("0x1.f37f47a03f82ap-511"),
1205: (16)           dtype=np.float64
1206: (12)           )
1207: (12)           np.sin(underflow_trigger)
1208: (12)           np.cos(underflow_trigger)
1209: (4)    @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1210: (4)    @pytest.mark.parametrize('callable', [np.sin, np.cos])
1211: (4)    @pytest.mark.parametrize('dtype', ['e', 'f', 'd'])
1212: (4)    @pytest.mark.parametrize('value', [np.inf, -np.inf])
1213: (4)    def test_sincos_errors(self, callable, dtype, value):
1214: (8)        with np.errstate(invalid='raise'):
1215: (12)            assert_raises(FloatingPointError, callable,
1216: (16)                np.array([value], dtype=dtype))
1217: (4)    @pytest.mark.parametrize('callable', [np.sin, np.cos])
1218: (4)    @pytest.mark.parametrize('dtype', ['f', 'd'])
1219: (4)    @pytest.mark.parametrize('stride', [-1, 1, 2, 4, 5])
1220: (4)    def test_sincos_overlaps(self, callable, dtype, stride):
1221: (8)        N = 100
1222: (8)        M = N // abs(stride)
1223: (8)        rng = np.random.default_rng(42)
1224: (8)        x = rng.standard_normal(N, dtype)
1225: (8)        y = callable(x[::-stride])
1226: (8)        callable(x[::-stride], out=x[:M])
1227: (8)        assert_equal(x[:M], y)
1228: (4)    @pytest.mark.parametrize('dt', ['e', 'f', 'd', 'g'])
1229: (4)    def test_sqrt_values(self, dt):
1230: (8)        with np.errstate(all='ignore'):
1231: (12)            x = [np.nan, np.nan, np.inf, np.nan, 0.]
1232: (12)            y = [np.nan, -np.nan, np.inf, -np.inf, 0.]
1233: (12)            xf = np.array(x, dtype=dt)
1234: (12)            yf = np.array(y, dtype=dt)
1235: (12)            assert_equal(np.sqrt(yf), xf)
1236: (4)    def test_abs_values(self):
1237: (8)        x = [np.nan, np.nan, np.inf, np.inf, 0., 0., 1.0, 1.0]
1238: (8)        y = [np.nan, -np.nan, np.inf, -np.inf, 0., -0., -1.0, 1.0]
1239: (8)        for dt in ['e', 'f', 'd', 'g']:
1240: (12)            xf = np.array(x, dtype=dt)
1241: (12)            yf = np.array(y, dtype=dt)
1242: (12)            assert_equal(np.abs(yf), xf)
1243: (4)    @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1244: (4)    def test_square_values(self):
1245: (8)        x = [np.nan, np.nan, np.inf, np.inf]
1246: (8)        y = [np.nan, -np.nan, np.inf, -np.inf]
1247: (8)        with np.errstate(all='ignore'):
1248: (12)            for dt in ['e', 'f', 'd', 'g']:
1249: (16)                xf = np.array(x, dtype=dt)
1250: (16)                yf = np.array(y, dtype=dt)
1251: (16)                assert_equal(np.square(yf), xf)
1252: (8)        with np.errstate(over='raise'):
1253: (12)            assert_raises(FloatingPointError, np.square,
1254: (26)                np.array(1E3, dtype='e'))
1255: (12)            assert_raises(FloatingPointError, np.square,
1256: (26)                np.array(1E32, dtype='f'))
1257: (12)            assert_raises(FloatingPointError, np.square,
1258: (26)                np.array(1E200, dtype='d'))
1259: (4)    @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1260: (4)    def test_reciprocal_values(self):
1261: (8)        with np.errstate(all='ignore'):
1262: (12)            x = [np.nan, np.nan, 0.0, -0.0, np.inf, -np.inf]
1263: (12)            y = [np.nan, -np.nan, np.inf, -np.inf, 0., -0.]
1264: (12)            for dt in ['e', 'f', 'd', 'g']:
1265: (16)                xf = np.array(x, dtype=dt)
1266: (16)                yf = np.array(y, dtype=dt)
1267: (16)                assert_equal(np.reciprocal(yf), xf)
1268: (8)        with np.errstate(divide='raise'):
1269: (12)            for dt in ['e', 'f', 'd', 'g']:
1270: (16)                assert_raises(FloatingPointError, np.reciprocal,
1271: (30)                    np.array(-0.0, dtype=dt))
1272: (4)    @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")

```

```

1273: (4)           def test_tan(self):
1274: (8)             with np.errstate(all='ignore'):
1275: (12)               in_ = [np.nan, -np.nan, 0.0, -0.0, np.inf, -np.inf]
1276: (12)               out = [np.nan, np.nan, 0.0, -0.0, np.nan, np.nan]
1277: (12)               for dt in ['e', 'f', 'd']:
1278: (16)                 in_arr = np.array(in_, dtype=dt)
1279: (16)                 out_arr = np.array(out, dtype=dt)
1280: (16)                 assert_equal(np.tan(in_arr), out_arr)
1281: (8)             with np.errstate(invalid='raise'):
1282: (12)               for dt in ['e', 'f', 'd']:
1283: (16)                 assert_raises(FloatingPointError, np.tan,
1284: (30)                               np.array(np.inf, dtype=dt))
1285: (16)                 assert_raises(FloatingPointError, np.tan,
1286: (30)                               np.array(-np.inf, dtype=dt))
1287: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1288: (4)           def test_arcsincos(self):
1289: (8)             with np.errstate(all='ignore'):
1290: (12)               in_ = [np.nan, -np.nan, np.inf, -np.inf]
1291: (12)               out = [np.nan, np.nan, np.nan, np.nan]
1292: (12)               for dt in ['e', 'f', 'd']:
1293: (16)                 in_arr = np.array(in_, dtype=dt)
1294: (16)                 out_arr = np.array(out, dtype=dt)
1295: (16)                 assert_equal(np.arcsin(in_arr), out_arr)
1296: (16)                 assert_equal(np.arccos(in_arr), out_arr)
1297: (8)             for callable in [np.arcsin, np.arccos]:
1298: (12)               for value in [np.inf, -np.inf, 2.0, -2.0]:
1299: (16)                 for dt in ['e', 'f', 'd']:
1300: (20)                   with np.errstate(invalid='raise'):
1301: (24)                     assert_raises(FloatingPointError, callable,
1302: (38)                           np.array(value, dtype=dt))
1303: (4)           def test_arctan(self):
1304: (8)             with np.errstate(all='ignore'):
1305: (12)               in_ = [np.nan, -np.nan]
1306: (12)               out = [np.nan, np.nan]
1307: (12)               for dt in ['e', 'f', 'd']:
1308: (16)                 in_arr = np.array(in_, dtype=dt)
1309: (16)                 out_arr = np.array(out, dtype=dt)
1310: (16)                 assert_equal(np.arctan(in_arr), out_arr)
1311: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1312: (4)           def test_sinh(self):
1313: (8)             in_ = [np.nan, -np.nan, np.inf, -np.inf]
1314: (8)             out = [np.nan, np.nan, np.inf, -np.inf]
1315: (8)             for dt in ['e', 'f', 'd']:
1316: (12)               in_arr = np.array(in_, dtype=dt)
1317: (12)               out_arr = np.array(out, dtype=dt)
1318: (12)               assert_equal(np.sinh(in_arr), out_arr)
1319: (8)             with np.errstate(over='raise'):
1320: (12)               assert_raises(FloatingPointError, np.sinh,
1321: (26)                             np.array(12.0, dtype='e'))
1322: (12)               assert_raises(FloatingPointError, np.sinh,
1323: (26)                             np.array(120.0, dtype='f'))
1324: (12)               assert_raises(FloatingPointError, np.sinh,
1325: (26)                             np.array(1200.0, dtype='d'))
1326: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1327: (4) @pytest.mark.skipif('bsd' in sys.platform,
1328: (12)                           reason="fallback implementation may not raise, see gh-2487")
1329: (4)           def test_cosh(self):
1330: (8)             in_ = [np.nan, -np.nan, np.inf, -np.inf]
1331: (8)             out = [np.nan, np.nan, np.inf, np.inf]
1332: (8)             for dt in ['e', 'f', 'd']:
1333: (12)               in_arr = np.array(in_, dtype=dt)
1334: (12)               out_arr = np.array(out, dtype=dt)
1335: (12)               assert_equal(np.cosh(in_arr), out_arr)
1336: (8)             with np.errstate(over='raise'):
1337: (12)               assert_raises(FloatingPointError, np.cosh,
1338: (26)                             np.array(12.0, dtype='e'))
1339: (12)               assert_raises(FloatingPointError, np.cosh,
1340: (26)                             np.array(120.0, dtype='f'))
1341: (12)               assert_raises(FloatingPointError, np.cosh,

```

```

1342: (26)                                     np.array(1200.0, dtype='d'))
1343: (4)          def test_tanh(self):
1344: (8)            in_ = [np.nan, -np.nan, np.inf, -np.inf]
1345: (8)            out = [np.nan, np.nan, 1.0, -1.0]
1346: (8)            for dt in ['e', 'f', 'd']:
1347: (12)              in_arr = np.array(in_, dtype=dt)
1348: (12)              out_arr = np.array(out, dtype=dt)
1349: (12)              assert_equal(np.tanh(in_arr), out_arr)
1350: (4)          def test_arcsinh(self):
1351: (8)            in_ = [np.nan, -np.nan, np.inf, -np.inf]
1352: (8)            out = [np.nan, np.nan, np.inf, -np.inf]
1353: (8)            for dt in ['e', 'f', 'd']:
1354: (12)              in_arr = np.array(in_, dtype=dt)
1355: (12)              out_arr = np.array(out, dtype=dt)
1356: (12)              assert_equal(np.arcsinh(in_arr), out_arr)
1357: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1358: (4)          def test_arccosh(self):
1359: (8)            with np.errstate(all='ignore'):
1360: (12)              in_ = [np.nan, -np.nan, np.inf, -np.inf, 1.0, 0.0]
1361: (12)              out = [np.nan, np.nan, np.inf, np.nan, 0.0, np.nan]
1362: (12)              for dt in ['e', 'f', 'd']:
1363: (16)                in_arr = np.array(in_, dtype=dt)
1364: (16)                out_arr = np.array(out, dtype=dt)
1365: (16)                assert_equal(np.arccosh(in_arr), out_arr)
1366: (8)            for value in [0.0, -np.inf]:
1367: (12)              with np.errstate(invalid='raise'):
1368: (16)                for dt in ['e', 'f', 'd']:
1369: (20)                  assert_raises(FloatingPointError, np.arccosh,
1370: (34)                                  np.array(value, dtype=dt))
1371: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1372: (4)          def test_arctanh(self):
1373: (8)            with np.errstate(all='ignore'):
1374: (12)              in_ = [np.nan, -np.nan, np.inf, -np.inf, 1.0, -1.0, 2.0]
1375: (12)              out = [np.nan, np.nan, np.nan, np.nan, np.inf, -np.inf, np.nan]
1376: (12)              for dt in ['e', 'f', 'd']:
1377: (16)                in_arr = np.array(in_, dtype=dt)
1378: (16)                out_arr = np.array(out, dtype=dt)
1379: (16)                assert_equal(np.arctanh(in_arr), out_arr)
1380: (8)            for value in [1.01, np.inf, -np.inf, 1.0, -1.0]:
1381: (12)              with np.errstate(invalid='raise', divide='raise'):
1382: (16)                for dt in ['e', 'f', 'd']:
1383: (20)                  assert_raises(FloatingPointError, np.arctanh,
1384: (34)                                  np.array(value, dtype=dt))
1385: (8)            assert np.signbit(np.arctanh(-1j).real)
1386: (4) @pytest.mark.xfail(
1387: (8)    _glibc_older_than("2.17"),
1388: (8)    reason="Older glibc versions may not raise appropriate FP exceptions"
1389: (4))
1390: (4)          def test_exp2(self):
1391: (8)            with np.errstate(all='ignore'):
1392: (12)              in_ = [np.nan, -np.nan, np.inf, -np.inf]
1393: (12)              out = [np.nan, np.nan, np.inf, 0.0]
1394: (12)              for dt in ['e', 'f', 'd']:
1395: (16)                in_arr = np.array(in_, dtype=dt)
1396: (16)                out_arr = np.array(out, dtype=dt)
1397: (16)                assert_equal(np.exp2(in_arr), out_arr)
1398: (8)            for value in [2000.0, -2000.0]:
1399: (12)              with np.errstate(over='raise', under='raise'):
1400: (16)                for dt in ['e', 'f', 'd']:
1401: (20)                  assert_raises(FloatingPointError, np.exp2,
1402: (34)                                  np.array(value, dtype=dt))
1403: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1404: (4)          def test_expm1(self):
1405: (8)            with np.errstate(all='ignore'):
1406: (12)              in_ = [np.nan, -np.nan, np.inf, -np.inf]
1407: (12)              out = [np.nan, np.nan, np.inf, -1.0]
1408: (12)              for dt in ['e', 'f', 'd']:
1409: (16)                in_arr = np.array(in_, dtype=dt)
1410: (16)                out_arr = np.array(out, dtype=dt)

```

```

1411: (16)                         assert_equal(np.expm1(in_arr), out_arr)
1412: (8)                          for value in [200.0, 2000.0]:
1413: (12)                         with np.errstate(over='raise'):
1414: (16)                           for dt in ['e', 'f']:
1415: (20)                             assert_raises(FloatingPointError, np.expm1,
1416: (34)   np.array(value, dtype=dt))
1417: (4) INF_INVALID_ERR = [
1418: (8)   np.cos, np.sin, np.tan, np.arccos, np.arcsin, np.spacing, np.arctanh
1419: (4)
1420: (4) ]
1421: (8) NEG_INVALID_ERR = [
1422: (8)   np.log, np.log2, np.log10, np.log1p, np.sqrt, np.arccosh,
1423: (4)   np.arctanh
1424: (4) ]
1425: (8) ONE_INVALID_ERR = [
1426: (4)   np.arctanh
1427: (4) ]
1428: (8) LTONE_INVALID_ERR = [
1429: (4)   np.arccosh
1430: (4) ]
1431: (8) BYZERO_ERR = [
1432: (4)   np.log, np.log2, np.log10, np.reciprocal, np.arccosh
1433: (4) ]
1434: (24) @pytest.mark.skipif(sys.platform == "win32" and sys.maxsize < 2**31 + 1,
1435: (4)   reason='failures on 32-bit Python, see FIXME below')
1436: (4) @pytest.mark.parametrize("ufunc", UFUNCS_UNARY_FP)
1437: (4) @pytest.mark.parametrize("dtype", ('e', 'f', 'd'))
1438: (8) @pytest.mark.parametrize("data, escape", (
1439: (8)   ([0.03], LTONE_INVALID_ERR),
1440: (8)   ([0.03]*32, LTONE_INVALID_ERR),
1441: (8)   ([-1.0], NEG_INVALID_ERR),
1442: (8)   ([-1.0]*32, NEG_INVALID_ERR),
1443: (8)   ([1.0], ONE_INVALID_ERR),
1444: (8)   ([1.0]*32, ONE_INVALID_ERR),
1445: (8)   ([0.0], BYZERO_ERR),
1446: (8)   ([0.0]*32, BYZERO_ERR),
1447: (8)   ([-0.0], BYZERO_ERR),
1448: (8)   ([-0.0]*32, BYZERO_ERR),
1449: (8)   ([[0.5, 0.5, 0.5, np.nan], LTONE_INVALID_ERR),
1450: (8)   ([[0.5, 0.5, 0.5, np.nan]*32, LTONE_INVALID_ERR),
1451: (8)   ([[np.nan, 1.0, 1.0, 1.0], ONE_INVALID_ERR),
1452: (8)   ([[np.nan, 1.0, 1.0, 1.0]*32, ONE_INVALID_ERR),
1453: (8)   ([[np.nan], []],
1454: (8)   ([[np.nan]*32, []],
1455: (8)   ([[0.5, 0.5, 0.5, np.inf], INF_INVALID_ERR + LTONE_INVALID_ERR),
1456: (8)   ([[0.5, 0.5, 0.5, np.inf]*32, INF_INVALID_ERR + LTONE_INVALID_ERR),
1457: (8)   ([[np.inf, 1.0, 1.0, 1.0], INF_INVALID_ERR),
1458: (8)   ([[np.inf, 1.0, 1.0, 1.0]*32, INF_INVALID_ERR),
1459: (8)   ([[np.inf], INF_INVALID_ERR),
1460: (8)   ([[np.inf]*32, INF_INVALID_ERR),
1461: (9)   ([0.5, 0.5, 0.5, -np.inf],
1462: (8)     NEG_INVALID_ERR + INF_INVALID_ERR + LTONE_INVALID_ERR),
1463: (9)   ([0.5, 0.5, 0.5, -np.inf]*32,
1464: (8)     NEG_INVALID_ERR + INF_INVALID_ERR + LTONE_INVALID_ERR),
1465: (8)   ([-np.inf, 1.0, 1.0, 1.0], NEG_INVALID_ERR + INF_INVALID_ERR),
1466: (8)   ([-np.inf, 1.0, 1.0, 1.0]*32, NEG_INVALID_ERR + INF_INVALID_ERR),
1467: (8)   ([-np.inf], NEG_INVALID_ERR + INF_INVALID_ERR),
1468: (4)   ([-np.inf]*32, NEG_INVALID_ERR + INF_INVALID_ERR),
1469: (4) )
1470: (8) def test Unary_spurious_fpexception(self, ufunc, dtype, data, escape):
1471: (12)   if escape and ufunc in escape:
1472: (8)     return
1473: (12)   if ufunc in (np.spacing, np.ceil) and dtype == 'e':
1474: (8)     return
1475: (8)   array = np.array(data, dtype=dtype)
1476: (12)   with assert_no_warnings():
1477: (4)     ufunc(array)
1478: (4) @pytest.mark.parametrize("dtype", ('e', 'f', 'd'))
1479: (8) def test Divide_spurious_fpexception(self, dtype):
1480: (4)   dt = np.dtype(dtype)

```

```

1480: (8)           dt_info = np.finfo(dt)
1481: (8)           subnorm = dt_info.smallest_subnormal
1482: (8)           with assert_no_warnings():
1483: (12)             np.zeros(128 + 1, dtype=dt) / subnorm
1484: (0)           class TestFPClass:
1485: (4)             @pytest.mark.parametrize("stride", [-5, -4, -3, -2, -1, 1,
1486: (32)                           2, 4, 5, 6, 7, 8, 9, 10])
1487: (4)             def test_fpclass(self, stride):
1488: (8)               arr_f64 = np.array([np.nan, -np.nan, np.inf, -np.inf, -1.0, 1.0, -0.0,
1489: (8)                 0.0, 2.2251e-308, -2.2251e-308], dtype='d')
1490: (8)               arr_f32 = np.array([np.nan, -np.nan, np.inf, -np.inf, -1.0, 1.0, -0.0,
1491: (8)                 0.0, 1.4013e-045, -1.4013e-045], dtype='f')
1492: (8)               nan     = np.array([True, True, False, False, False, False,
1493: (8)                 False, False, False])
1494: (8)               inf    = np.array([False, False, True, True, False, False,
1495: (8)                 False, False, False])
1496: (8)               sign   = np.array([False, True, False, True, True, False, True,
1497: (8)                 False, False, False])
1498: (8)               finite = np.array([False, False, False, False, True, True, True,
1499: (8)                 True, True, True])
1500: (8)               assert_equal(np.isnan(arr_f32[::stride]), nan[::stride])
1501: (8)               assert_equal(np.isnan(arr_f64[::stride]), nan[::stride])
1502: (4)               assert_equal(np.isinf(arr_f32[::stride]), inf[::stride])
1503: (4)               assert_equal(np.isinf(arr_f64[::stride]), inf[::stride])
1504: (8)               assert_equal(np.signbit(arr_f32[::stride]), sign[::stride])
1505: (28)              assert_equal(np.signbit(arr_f64[::stride]), sign[::stride])
1506: (28)              assert_equal(np.isfinite(arr_f32[::stride]), finite[::stride])
1507: (28)              assert_equal(np.isfinite(arr_f64[::stride]), finite[::stride])
1508: (28)              @pytest.mark.parametrize("dtype", ['d', 'f'])
1509: (8)              def test_fp_noncontiguous(self, dtype):
1510: (28)                data = np.array([np.nan, -np.nan, np.inf, -np.inf, -1.0,
1511: (28)                  1.0, -0.0, 0.0, 2.2251e-308,
1512: (28)                  -2.2251e-308], dtype=dtype)
1513: (8)                nan = np.array([True, True, False, False, False, False,
1514: (28)                  False, False, False, False])
1515: (8)                inf = np.array([False, False, True, True, False, False,
1516: (8)                  False, False, False, False])
1517: (8)                sign = np.array([False, True, False, True, True, False,
1518: (8)                  True, False, False, True])
1519: (8)                finite = np.array([False, False, False, False, True, True,
1520: (8)                  True, True, True, True])
1521: (8)                out = np.ndarray(data.shape, dtype='bool')
1522: (8)                ncontig_in = data[1::3]
1523: (8)                ncontig_out = out[1::3]
1524: (8)                contig_in = np.array(ncontig_in)
1525: (8)                assert_equal(ncontig_in.flags.c_contiguous, False)
1526: (8)                assert_equal(ncontig_out.flags.c_contiguous, False)
1527: (8)                assert_equal(contig_in.flags.c_contiguous, True)
1528: (8)                assert_equal(np.isnan(ncontig_in, out=ncontig_out), nan[1::3])
1529: (8)                assert_equal(np.isinf(ncontig_in, out=ncontig_out), inf[1::3])
1530: (8)                assert_equal(np.signbit(ncontig_in, out=ncontig_out), sign[1::3])
1531: (8)                assert_equal(np.isfinite(ncontig_in, out=ncontig_out), finite[1::3])
1532: (8)                assert_equal(np.isnan(contig_in, out=ncontig_out), nan[1::3])
1533: (8)                assert_equal(np.isinf(contig_in, out=ncontig_out), inf[1::3])
1534: (8)                assert_equal(np.signbit(contig_in, out=ncontig_out), sign[1::3])
1535: (8)                assert_equal(np.isfinite(contig_in, out=ncontig_out), finite[1::3])
1536: (8)                assert_equal(np.isnan(ncontig_in), nan[1::3])
1537: (8)                assert_equal(np.isinf(ncontig_in), inf[1::3])
1538: (8)                assert_equal(np.signbit(ncontig_in), sign[1::3])
1539: (8)                assert_equal(np.isfinite(ncontig_in), finite[1::3])
1540: (8)                data_split = np.array(np.array_split(data, 2))
1541: (8)                nan_split = np.array(np.array_split(nan, 2))
1542: (8)                inf_split = np.array(np.array_split(inf, 2))
1543: (8)                sign_split = np.array(np.array_split(sign, 2))
1544: (8)                finite_split = np.array(np.array_split(finite, 2))
1545: (8)                assert_equal(np.isnan(data_split), nan_split)
1546: (8)                assert_equal(np.isinf(data_split), inf_split)
1547: (8)                assert_equal(np.signbit(data_split), sign_split)
1548: (8)                assert_equal(np.isfinite(data_split), finite_split)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1543: (0)
1544: (4)
1545: (4)
1546: (4)
1547: (8)
1548: (8)
1549: (8)
1550: (8)
out=out[:::stride]), np.ones(8, dtype=dtype)[:::stride])
1551: (8)
1552: (0)    class TestFRExp:
1553: (4)        @pytest.mark.parametrize("stride", [-4,-2,-1,1,2,4])
1554: (4)        @pytest.mark.parametrize("dtype", ['f', 'd'])
1555: (4)        def test_ldexp(self, dtype, stride):
1556: (4)            mant = np.array([0.125, 0.25, 0.5, 1., 1., 2., 4., 8.], dtype=dtype)
1557: (4)            exp = np.array([3, 2, 1, 0, 0, -1, -2, -3], dtype='i')
1558: (4)            out = np.zeros(8, dtype=dtype)
1559: (8)            assert_equal(np.ldexp(mant[:::stride], exp[:::stride],
on windows and linux")
1560: (8)
1561: (8)
1562: (8)
1563: (8)
1564: (8)
out_exp[:::stride]))
1565: (8)
1566: (8)
1567: (8)
1568: (8)
1569: (0)    avx_ufuncs = {'sqrt'      :[1,  0.,  100.],
1570: (14)          'absolute'   :[0, -100., 100.],
1571: (14)          'reciprocal' :[1,  1.,  100.],
1572: (14)          'square'     :[1, -100., 100.],
1573: (14)          'rint'       :[0, -100., 100.],
1574: (14)          'floor'      :[0, -100., 100.],
1575: (14)          'ceil'       :[0, -100., 100.],
1576: (14)          'trunc'      :[0, -100., 100.]}
1577: (0)    class TestAVXUfuncs:
1578: (4)        def test_avx_based_ufunc(self):
1579: (8)            strides = np.array([-4,-3,-2,-1,1,2,3,4])
1580: (8)            np.random.seed(42)
1581: (8)            for func, prop in avx_ufuncs.items():
1582: (12)                maxulperr = prop[0]
1583: (12)                minval = prop[1]
1584: (12)                maxval = prop[2]
1585: (12)                for size in range(1,32):
1586: (16)                    myfunc = getattr(np, func)
1587: (16)                    x_f32 = np.float32(np.random.uniform(low=minval, high=maxval,
1588: (20)                        size=size))
1589: (16)                    x_f64 = np.float64(x_f32)
1590: (16)                    x_f128 = np.longdouble(x_f32)
1591: (16)                    y_true128 = myfunc(x_f128)
1592: (16)                    if maxulperr == 0:
1593: (20)                        assert_equal(myfunc(x_f32), np.float32(y_true128))
1594: (20)                        assert_equal(myfunc(x_f64), np.float64(y_true128))
1595: (16)                    else:
1596: (20)                        assert_array_max_ulp(myfunc(x_f32), np.float32(y_true128),
1597: (28)                            maxulp=maxulperr)
1598: (20)                        assert_array_max_ulp(myfunc(x_f64), np.float64(y_true128),
1599: (28)                            maxulp=maxulperr)
1600: (16)                    if size > 1:
1601: (20)                        y_true32 = myfunc(x_f32)
1602: (20)                        y_true64 = myfunc(x_f64)
1603: (20)                        for jj in strides:
1604: (24)                            assert_equal(myfunc(x_f64[:::jj]), y_true64[:::jj])
1605: (24)                            assert_equal(myfunc(x_f32[:::jj]), y_true32[:::jj])
1606: (0)    class TestAVXFloat32Transcendental:

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1607: (4)           def test_exp_float32(self):
1608: (8)             np.random.seed(42)
1609: (8)             x_f32 = np.float32(np.random.uniform(low=0.0,high=88.1,size=1000000))
1610: (8)             x_f64 = np.float64(x_f32)
1611: (8)             assert_array_max_ulp(np.exp(x_f32), np.float32(np.exp(x_f64)),
maxulp=3)
1612: (4)           def test_log_float32(self):
1613: (8)             np.random.seed(42)
1614: (8)             x_f32 = np.float32(np.random.uniform(low=0.0,high=1000,size=1000000))
1615: (8)             x_f64 = np.float64(x_f32)
1616: (8)             assert_array_max_ulp(np.log(x_f32), np.float32(np.log(x_f64)),
maxulp=4)
1617: (4)           def test_sincos_float32(self):
1618: (8)             np.random.seed(42)
1619: (8)             N = 1000000
1620: (8)             M = np.int_(N/20)
1621: (8)             index = np.random.randint(low=0, high=N, size=M)
1622: (8)             x_f32 = np.float32(np.random.uniform(low=-100.,high=100.,size=N))
1623: (8)             if not _glibc_older_than("2.17"):
1624: (12)               x_f32[index] = np.float32(10E+10*np.random.rand(M))
1625: (8)             x_f64 = np.float64(x_f32)
1626: (8)             assert_array_max_ulp(np.sin(x_f32), np.float32(np.sin(x_f64)),
maxulp=2)
1627: (8)             assert_array_max_ulp(np.cos(x_f32), np.float32(np.cos(x_f64)),
maxulp=2)
1628: (8)             tx_f32 = x_f32.copy()
1629: (8)             assert_array_max_ulp(np.sin(x_f32, out=x_f32),
np.float32(np.sin(x_f64)), maxulp=2)
1630: (8)             assert_array_max_ulp(np.cos(tx_f32, out=tx_f32),
np.float32(np.cos(x_f64)), maxulp=2)
1631: (4)           def test_strided_float32(self):
1632: (8)             np.random.seed(42)
1633: (8)             strides = np.array([-4,-3,-2,-1,1,2,3,4])
1634: (8)             sizes = np.arange(2,100)
1635: (8)             for ii in sizes:
1636: (12)               x_f32 = np.float32(np.random.uniform(low=0.01,high=88.1,size=ii))
1637: (12)               x_f32_large = x_f32.copy()
1638: (12)               x_f32_large[3:-1:4] = 120000.0
1639: (12)               exp_true = np.exp(x_f32)
1640: (12)               log_true = np.log(x_f32)
1641: (12)               sin_true = np.sin(x_f32_large)
1642: (12)               cos_true = np.cos(x_f32_large)
1643: (12)               for jj in strides:
1644: (16)                 assert_array_almost_equal_nulp(np.exp(x_f32[::-jj]),
exp_true[::-jj], nulp=2)
1645: (16)                 assert_array_almost_equal_nulp(np.log(x_f32[::-jj]),
log_true[::-jj], nulp=2)
1646: (16)                 assert_array_almost_equal_nulp(np.sin(x_f32_large[::-jj]),
sin_true[::-jj], nulp=2)
1647: (16)                 assert_array_almost_equal_nulp(np.cos(x_f32_large[::-jj]),
cos_true[::-jj], nulp=2)
1648: (0)           class TestLogAddExp(_FilterInvalids):
1649: (4)             def test_logaddexp_values(self):
1650: (8)               x = [1, 2, 3, 4, 5]
1651: (8)               y = [5, 4, 3, 2, 1]
1652: (8)               z = [6, 6, 6, 6, 6]
1653: (8)               for dt, dec_ in zip(['f', 'd', 'g'], [6, 15, 15]):
1654: (12)                 xf = np.log(np.array(x, dtype=dt))
1655: (12)                 yf = np.log(np.array(y, dtype=dt))
1656: (12)                 zf = np.log(np.array(z, dtype=dt))
1657: (12)                 assert_almost_equal(np.logaddexp(xf, yf), zf, decimal=dec_)
1658: (4)             def test_logaddexp_range(self):
1659: (8)               x = [1000000, -1000000, 1000200, -1000200]
1660: (8)               y = [1000200, -1000200, 1000000, -1000000]
1661: (8)               z = [1000200, -1000000, 1000200, -1000000]
1662: (8)               for dt in ['f', 'd', 'g']:
1663: (12)                 logxf = np.array(x, dtype=dt)
1664: (12)                 logyf = np.array(y, dtype=dt)
1665: (12)                 logzf = np.array(z, dtype=dt)

```

```

1666: (12)                     assert_almost_equal(np.logaddexp(logxf, logyf), logzf)
1667: (4)          def test_inf(self):
1668: (8)            inf = np.inf
1669: (8)            x = [inf, -inf, inf, -inf, inf, 1, -inf, 1]
1670: (8)            y = [inf, inf, -inf, -inf, 1, inf, 1, -inf]
1671: (8)            z = [inf, inf, inf, -inf, inf, inf, 1, 1]
1672: (8)            with np.errstate(invalid='raise'):
1673: (12)              for dt in ['f', 'd', 'g']:
1674: (16)                logxf = np.array(x, dtype=dt)
1675: (16)                logyf = np.array(y, dtype=dt)
1676: (16)                logzf = np.array(z, dtype=dt)
1677: (16)                assert_equal(np.logaddexp(logxf, logyf), logzf)
1678: (4)          def test_nan(self):
1679: (8)            assert_(np.isnan(np.logaddexp(np.nan, np.inf)))
1680: (8)            assert_(np.isnan(np.logaddexp(np.inf, np.nan)))
1681: (8)            assert_(np.isnan(np.logaddexp(np.nan, 0)))
1682: (8)            assert_(np.isnan(np.logaddexp(0, np.nan)))
1683: (8)            assert_(np.isnan(np.logaddexp(np.nan, np.nan)))
1684: (4)          def test_reduce(self):
1685: (8)            assert_equal(np.logaddexp.identity, -np.inf)
1686: (8)            assert_equal(np.logaddexp.reduce([]), -np.inf)
1687: (0)        class TestLog1p:
1688: (4)          def test_log1p(self):
1689: (8)            assert_almost_equal(ncu.log1p(0.2), ncu.log(1.2))
1690: (8)            assert_almost_equal(ncu.log1p(1e-6), ncu.log(1+1e-6))
1691: (4)          def test_special(self):
1692: (8)            with np.errstate(invalid="ignore", divide="ignore"):
1693: (12)              assert_equal(ncu.log1p(np.nan), np.nan)
1694: (12)              assert_equal(ncu.log1p(np.inf), np.inf)
1695: (12)              assert_equal(ncu.log1p(-1.), -np.inf)
1696: (12)              assert_equal(ncu.log1p(-2.), np.nan)
1697: (12)              assert_equal(ncu.log1p(-np.inf), np.nan)
1698: (0)        class TestExpm1:
1699: (4)          def test_expm1(self):
1700: (8)            assert_almost_equal(ncu.expm1(0.2), ncu.exp(0.2)-1)
1701: (8)            assert_almost_equal(ncu.expm1(1e-6), ncu.exp(1e-6)-1)
1702: (4)          def test_special(self):
1703: (8)            assert_equal(ncu.expm1(np.inf), np.inf)
1704: (8)            assert_equal(ncu.expm1(0.), 0.)
1705: (8)            assert_equal(ncu.expm1(-0.), -0.)
1706: (8)            assert_equal(ncu.expm1(np.inf), np.inf)
1707: (8)            assert_equal(ncu.expm1(-np.inf), -1.)
1708: (4)          def test_complex(self):
1709: (8)            x = np.asarray(1e-12)
1710: (8)            assert_allclose(x, ncu.expm1(x))
1711: (8)            x = x.astype(np.complex128)
1712: (8)            assert_allclose(x, ncu.expm1(x))
1713: (0)        class TestHypot:
1714: (4)          def test_simple(self):
1715: (8)            assert_almost_equal(ncu.hypot(1, 1), ncu.sqrt(2))
1716: (8)            assert_almost_equal(ncu.hypot(0, 0), 0)
1717: (4)          def test_reduce(self):
1718: (8)            assert_almost_equal(ncu.hypot.reduce([3.0, 4.0]), 5.0)
1719: (8)            assert_almost_equal(ncu.hypot.reduce([3.0, 4.0, 0]), 5.0)
1720: (8)            assert_almost_equal(ncu.hypot.reduce([9.0, 12.0, 20.0]), 25.0)
1721: (8)            assert_equal(ncu.hypot.reduce([]), 0.0)
1722: (0)          def assert_hypot_isnan(x, y):
1723: (4)            with np.errstate(invalid='ignore'):
1724: (8)              assert_(np.isnan(ncu.hypot(x, y)),
1725: (16)                  "hypot(%s, %s) is %s, not nan" % (x, y, ncu.hypot(x, y)))
1726: (0)          def assert_hypot_isinf(x, y):
1727: (4)            with np.errstate(invalid='ignore'):
1728: (8)              assert_(np.isinf(ncu.hypot(x, y)),
1729: (16)                  "hypot(%s, %s) is %s, not inf" % (x, y, ncu.hypot(x, y)))
1730: (0)        class TestHypotSpecialValues:
1731: (4)          def test_nan_outputs(self):
1732: (8)            assert_hypot_isnan(np.nan, np.nan)
1733: (8)            assert_hypot_isnan(np.nan, 1)
1734: (4)          def test_nan_outputs2(self):

```

```

1735: (8)             assert_hypot_isinf(np.nan, np.inf)
1736: (8)             assert_hypot_isinf(np.inf, np.nan)
1737: (8)             assert_hypot_isinf(np.inf, 0)
1738: (8)             assert_hypot_isinf(0, np.inf)
1739: (8)             assert_hypot_isinf(np.inf, np.inf)
1740: (8)             assert_hypot_isinf(np.inf, 23.0)
1741: (4)             def test_no_fpe(self):
1742: (8)                 assert_no_warnings(ncu.hypot, np.inf, 0)
1743: (0)             def assert_arctan2_isnan(x, y):
1744: (4)                 assert_(np.isnan(ncu.arctan2(x, y)), "arctan(%s, %s) is %s, not nan" % (x,
y, ncu.arctan2(x, y)))
1745: (0)             def assert_arctan2_ispinf(x, y):
1746: (4)                 assert_((np.isinf(ncu.arctan2(x, y)) and ncu.arctan2(x, y) > 0),
"arctan(%s, %s) is %s, not +inf" % (x, y, ncu.arctan2(x, y)))
1747: (0)             def assert_arctan2_isninf(x, y):
1748: (4)                 assert_((np.isinf(ncu.arctan2(x, y)) and ncu.arctan2(x, y) < 0),
"arctan(%s, %s) is %s, not -inf" % (x, y, ncu.arctan2(x, y)))
1749: (0)             def assert_arctan2_ispzero(x, y):
1750: (4)                 assert_((ncu.arctan2(x, y) == 0 and not np.signbit(ncu.arctan2(x, y))),
"arctan(%s, %s) is %s, not +0" % (x, y, ncu.arctan2(x, y)))
1751: (0)             def assert_arctan2_isnzero(x, y):
1752: (4)                 assert_((ncu.arctan2(x, y) == 0 and np.signbit(ncu.arctan2(x, y))),
"arctan(%s, %s) is %s, not -0" % (x, y, ncu.arctan2(x, y)))
1753: (0)             class TestArctan2SpecialValues:
1754: (4)                 def test_one_one(self):
1755: (8)                     assert_almost_equal(ncu.arctan2(1, 1), 0.25 * np.pi)
1756: (8)                     assert_almost_equal(ncu.arctan2(-1, 1), -0.25 * np.pi)
1757: (8)                     assert_almost_equal(ncu.arctan2(1, -1), 0.75 * np.pi)
1758: (4)                 def test_zero_nzero(self):
1759: (8)                     assert_almost_equal(ncu.arctan2(np.PZERO, np.NZERO), np.pi)
1760: (8)                     assert_almost_equal(ncu.arctan2(np.NZERO, np.NZERO), -np.pi)
1761: (4)                 def test_zero_pzero(self):
1762: (8)                     assert_arctan2_ispzero(np.PZERO, np.PZERO)
1763: (8)                     assert_arctan2_isnzero(np.NZERO, np.PZERO)
1764: (4)                 def test_zero_negative(self):
1765: (8)                     assert_almost_equal(ncu.arctan2(np.PZERO, -1), np.pi)
1766: (8)                     assert_almost_equal(ncu.arctan2(np.NZERO, -1), -np.pi)
1767: (4)                 def test_zero_positive(self):
1768: (8)                     assert_arctan2_ispzero(np.PZERO, 1)
1769: (8)                     assert_arctan2_isnzero(np.NZERO, 1)
1770: (4)                 def test_positive_zero(self):
1771: (8)                     assert_almost_equal(ncu.arctan2(1, np.PZERO), 0.5 * np.pi)
1772: (8)                     assert_almost_equal(ncu.arctan2(1, np.NZERO), 0.5 * np.pi)
1773: (4)                 def test_negative_zero(self):
1774: (8)                     assert_almost_equal(ncu.arctan2(-1, np.PZERO), -0.5 * np.pi)
1775: (8)                     assert_almost_equal(ncu.arctan2(-1, np.NZERO), -0.5 * np.pi)
1776: (4)                 def test_any_ninf(self):
1777: (8)                     assert_almost_equal(ncu.arctan2(1, np.NINF), np.pi)
1778: (8)                     assert_almost_equal(ncu.arctan2(-1, np.NINF), -np.pi)
1779: (4)                 def test_any_pinf(self):
1780: (8)                     assert_arctan2_ispzero(1, np.inf)
1781: (8)                     assert_arctan2_isnzero(-1, np.inf)
1782: (4)                 def test_inf_any(self):
1783: (8)                     assert_almost_equal(ncu.arctan2(np.inf, 1), 0.5 * np.pi)
1784: (8)                     assert_almost_equal(ncu.arctan2(-np.inf, 1), -0.5 * np.pi)
1785: (4)                 def test_inf_ninf(self):
1786: (8)                     assert_almost_equal(ncu.arctan2(np.inf, -np.inf), 0.75 * np.pi)
1787: (8)                     assert_almost_equal(ncu.arctan2(-np.inf, -np.inf), -0.75 * np.pi)
1788: (4)                 def test_inf_pinf(self):
1789: (8)                     assert_almost_equal(ncu.arctan2(np.inf, np.inf), 0.25 * np.pi)
1790: (8)                     assert_almost_equal(ncu.arctan2(-np.inf, np.inf), -0.25 * np.pi)
1791: (4)                 def test_nan_any(self):
1792: (8)                     assert_arctan2_isnan(np.nan, np.inf)
1793: (8)                     assert_arctan2_isnan(np.inf, np.nan)
1794: (8)                     assert_arctan2_isnan(np.nan, np.nan)
1795: (0)             class TestLdexp:
1796: (4)                 def _check_ldexp(self, tp):
1797: (8)                     assert_almost_equal(ncu.ldexp(np.array(2., np.float32),
np.array(3, tp)), 16.)
1798: (38)

```

```

1799: (8) assert_almost_equal(ncu.ldexp(np.array(2., np.float64),
1800: (38)                                     np.array(3, tp)), 16.)
1801: (8) assert_almost_equal(ncu.ldexp(np.array(2., np.longdouble),
1802: (38)                                     np.array(3, tp)), 16.)
1803: (4) def test_ldexp(self):
1804: (8)     assert_almost_equal(ncu.ldexp(2., 3), 16.)
1805: (8)     self._check_ldexp(np.int8)
1806: (8)     self._check_ldexp(np.int16)
1807: (8)     self._check_ldexp(np.int32)
1808: (8)     self._check_ldexp('i')
1809: (8)     self._check_ldexp('l')
1810: (4) def test_ldexp_overflow(self):
1811: (8)     with np.errstate(over="ignore"):
1812: (12)         imax = np.iinfo(np.dtype('l')).max
1813: (12)         imin = np.iinfo(np.dtype('l')).min
1814: (12)         assert_equal(ncu.ldexp(2., imax), np.inf)
1815: (12)         assert_equal(ncu.ldexp(2., imin), 0)
1816: (0) class TestMaximum(_FilterInvalids):
1817: (4)     def test_reduce(self):
1818: (8)         dfilt = np.typecodes['AllFloat']
1819: (8)         dint = np.typecodes['AllInteger']
1820: (8)         seq1 = np.arange(11)
1821: (8)         seq2 = seq1[::-1]
1822: (8)         func = np.maximum.reduce
1823: (8)         for dt in dint:
1824: (12)             tmp1 = seq1.astype(dt)
1825: (12)             tmp2 = seq2.astype(dt)
1826: (12)             assert_equal(func(tmp1), 10)
1827: (12)             assert_equal(func(tmp2), 10)
1828: (8)         for dt in dfilt:
1829: (12)             tmp1 = seq1.astype(dt)
1830: (12)             tmp2 = seq2.astype(dt)
1831: (12)             assert_equal(func(tmp1), 10)
1832: (12)             assert_equal(func(tmp2), 10)
1833: (12)             tmp1[::2] = np.nan
1834: (12)             tmp2[::2] = np.nan
1835: (12)             assert_equal(func(tmp1), np.nan)
1836: (12)             assert_equal(func(tmp2), np.nan)
1837: (4)     def test_reduce_complex(self):
1838: (8)         assert_equal(np.maximum.reduce([1, 2j]), 1)
1839: (8)         assert_equal(np.maximum.reduce([1+3j, 2j]), 1+3j)
1840: (4)     def test_float_nans(self):
1841: (8)         nan = np.nan
1842: (8)         arg1 = np.array([0, nan, nan])
1843: (8)         arg2 = np.array([nan, 0, nan])
1844: (8)         out = np.array([nan, nan, nan])
1845: (8)         assert_equal(np.maximum(arg1, arg2), out)
1846: (4)     def test_object_nans(self):
1847: (8)         for i in range(1):
1848: (12)             x = np.array(float('nan'), object)
1849: (12)             y = 1.0
1850: (12)             z = np.array(float('nan'), object)
1851: (12)             assert_(np.maximum(x, y) == 1.0)
1852: (12)             assert_(np.maximum(z, y) == 1.0)
1853: (4)     def test_complex_nans(self):
1854: (8)         nan = np.nan
1855: (8)         for cnan in [complex(nan, 0), complex(0, nan), complex(nan, nan)]:
1856: (12)             arg1 = np.array([0, cnan, cnan], dtype=complex)
1857: (12)             arg2 = np.array([cnan, 0, cnan], dtype=complex)
1858: (12)             out = np.array([nan, nan, nan], dtype=complex)
1859: (12)             assert_equal(np.maximum(arg1, arg2), out)
1860: (4)     def test_object_array(self):
1861: (8)         arg1 = np.arange(5, dtype=object)
1862: (8)         arg2 = arg1 + 1
1863: (8)         assert_equal(np.maximum(arg1, arg2), arg2)
1864: (4)     def test_strided_array(self):
1865: (8)         arr1 = np.array([-4.0, 1.0, 10.0, 0.0, np.nan, -np.nan, np.inf, -
1866: (8)                                     np.inf])
1867: (8)         arr2 = np.array([-2.0, -1.0, np.nan, 1.0, 0.0, np.nan, 1.0,
```

```

-3.0])
1867: (8)           maxtrue = np.array([-2.0, 1.0, np.nan, 1.0, np.nan, np.nan, np.inf,
-3.0])
1868: (8)           out = np.ones(8)
1869: (8)           out_maxtrue = np.array([-2.0, 1.0, 1.0, 10.0, 1.0, 1.0, np.nan, 1.0])
1870: (8)           assert_equal(np.maximum(arr1,arr2), maxtrue)
1871: (8)           assert_equal(np.maximum(arr1[::-2],arr2[::-2]), maxtrue[::-2])
1872: (8)           assert_equal(np.maximum(arr1[:4:], arr2[::-2]), np.array([-2.0, np.nan,
10.0, 1.0]))
1873: (8)           assert_equal(np.maximum(arr1[::-3], arr2[:3:]), np.array([-2.0, 0.0,
np.nan]))
1874: (8)           assert_equal(np.maximum(arr1[:6:2], arr2[::-3]), out=out[::-3]),
np.array([-2.0, 10., np.nan]))
1875: (8)           assert_equal(out, out_maxtrue)
1876: (4)           def test_precision(self):
1877: (8)               dtypes = [np.float16, np.float32, np.float64, np.longdouble]
1878: (8)               for dt in dtypes:
1879: (12)                   dtmin = np.finfo(dt).min
1880: (12)                   dtmax = np.finfo(dt).max
1881: (12)                   d1 = dt(0.1)
1882: (12)                   d1_next = np.nextafter(d1, np.inf)
1883: (12)                   test_cases = [
1884: (16)                       (dtmin, -np.inf,      dtmin),
1885: (16)                       (dtmax, -np.inf,      dtmax),
1886: (16)                       (d1,      d1_next,     d1_next),
1887: (16)                       (dtmax, np.nan,       np.nan),
1888: (12)                   ]
1889: (12)                   for v1, v2, expected in test_cases:
1890: (16)                       assert_equal(np.maximum([v1], [v2]), [expected])
1891: (16)                       assert_equal(np.maximum.reduce([v1, v2]), expected)
1892: (0)           class TestMinimum(_FilterInvalids):
1893: (4)               def test_reduce(self):
1894: (8)                   dfilt = np.typecodes['AllFloat']
1895: (8)                   dint = np.typecodes['AllInteger']
1896: (8)                   seq1 = np.arange(11)
1897: (8)                   seq2 = seq1[::-1]
1898: (8)                   func = np.minimum.reduce
1899: (8)                   for dt in dint:
1900: (12)                       tmp1 = seq1.astype(dt)
1901: (12)                       tmp2 = seq2.astype(dt)
1902: (12)                       assert_equal(func(tmp1), 0)
1903: (12)                       assert_equal(func(tmp2), 0)
1904: (8)                   for dt in dfilt:
1905: (12)                       tmp1 = seq1.astype(dt)
1906: (12)                       tmp2 = seq2.astype(dt)
1907: (12)                       assert_equal(func(tmp1), 0)
1908: (12)                       assert_equal(func(tmp2), 0)
1909: (12)                       tmp1[::-2] = np.nan
1910: (12)                       tmp2[::-2] = np.nan
1911: (12)                       assert_equal(func(tmp1), np.nan)
1912: (12)                       assert_equal(func(tmp2), np.nan)
1913: (4)               def test_reduce_complex(self):
1914: (8)                   assert_equal(np.minimum.reduce([1, 2j]), 2j)
1915: (8)                   assert_equal(np.minimum.reduce([1+3j, 2j]), 2j)
1916: (4)               def test_float_nans(self):
1917: (8)                   nan = np.nan
1918: (8)                   arg1 = np.array([0,    nan,  nan])
1919: (8)                   arg2 = np.array([nan,  0,    nan])
1920: (8)                   out = np.array([nan,  nan,  nan])
1921: (8)                   assert_equal(np.minimum(arg1, arg2), out)
1922: (4)               def test_object_nans(self):
1923: (8)                   for i in range(1):
1924: (12)                       x = np.array(float('nan'), object)
1925: (12)                       y = 1.0
1926: (12)                       z = np.array(float('nan'), object)
1927: (12)                       assert_(np.minimum(x, y) == 1.0)
1928: (12)                       assert_(np.minimum(z, y) == 1.0)
1929: (4)               def test_complex_nans(self):
1930: (8)                   nan = np.nan

```

```

1931: (8)             for cnan in [complex(nan, 0), complex(0, nan), complex(nan, nan)]:
1932: (12)            arg1 = np.array([0, cnan, cnan], dtype=complex)
1933: (12)            arg2 = np.array([cnan, 0, cnan], dtype=complex)
1934: (12)            out = np.array([nan, nan, nan], dtype=complex)
1935: (12)            assert_equal(np.minimum(arg1, arg2), out)
1936: (4)             def test_object_array(self):
1937: (8)               arg1 = np.arange(5, dtype=object)
1938: (8)               arg2 = arg1 + 1
1939: (8)               assert_equal(np.minimum(arg1, arg2), arg1)
1940: (4)             def test_strided_array(self):
1941: (8)               arr1 = np.array([-4.0, 1.0, 10.0, 0.0, np.nan, -np.nan, np.inf, -
1942: (8)                 np.inf])
1943: (8)               arr2 = np.array([-2.0, -1.0, np.nan, 1.0, 0.0, np.nan, 1.0,
1944: (8)                 -3.0])
1945: (8)               mintrue = np.array([-4.0, -1.0, np.nan, 0.0, np.nan, np.nan, 1.0, -
1946: (8)                 out = np.ones(8)
1947: (8)               out_mintrue = np.array([-4.0, 1.0, 1.0, 1.0, 1.0, 1.0, np.nan, 1.0])
1948: (8)               assert_equal(np.minimum(arr1, arr2), mintrue)
1949: (8)               assert_equal(np.minimum(arr1[:2], arr2[:2]), mintrue[:2])
1950: (8)               assert_equal(np.minimum(arr1[4:], arr2[2:]), np.array([-4.0, np.nan,
1951: (8)                 0.0, 0.0]))
1952: (4)             assert_equal(np.minimum(arr1[::3], arr2[::3]), np.array([-4.0, -1.0,
1953: (8)                 np.nan]))
1954: (8)             assert_equal(np.minimum(arr1[::6], arr2[::3], out=out[::3]),
1955: (12)                np.array([-4.0, 1.0, np.nan]))
1956: (12)                assert_equal(out, out_mintrue)
1957: (12)             def test_precision(self):
1958: (12)               dtypes = [np.float16, np.float32, np.float64, np.longdouble]
1959: (12)               for dt in dtypes:
1960: (16)                 dtmin = np.finfo(dt).min
1961: (16)                 dtmax = np.finfo(dt).max
1962: (16)                 d1 = dt(0.1)
1963: (16)                 d1_next = np.nextafter(d1, np.inf)
1964: (12)                 test_cases = [
1965: (12)                   (dtmin, np.inf, dtmin),
1966: (16)                   (dtmax, np.inf, dtmax),
1967: (16)                   (d1, d1_next, d1),
1968: (0)                    (dtmin, np.nan, np.nan),
1969: (4)                 ]
1970: (8)                 for v1, v2, expected in test_cases:
1971: (8)                   assert_equal(np.minimum([v1], [v2]), [expected])
1972: (8)                   assert_equal(np.minimum.reduce([v1, v2]), expected)
1973: (8)             class TestFmax(_FilterInvalids):
1974: (8)               def test_reduce(self):
1975: (8)                 dflt = np.typecodes['AllFloat']
1976: (12)                 dint = np.typecodes['AllInteger']
1977: (12)                 seq1 = np.arange(11)
1978: (12)                 seq2 = seq1[::-1]
1979: (12)                 func = np.fmax.reduce
1980: (8)                 for dt in dint:
1981: (12)                   tmp1 = seq1.astype(dt)
1982: (12)                   tmp2 = seq2.astype(dt)
1983: (12)                   assert_equal(func(tmp1), 10)
1984: (12)                   assert_equal(func(tmp2), 10)
1985: (12)                 for dt in dflt:
1986: (12)                   tmp1 = seq1.astype(dt)
1987: (12)                   tmp2 = seq2.astype(dt)
1988: (12)                   assert_equal(func(tmp1), 10)
1989: (4)                     assert_equal(func(tmp2), 10)
1990: (8)             def test_reduce_complex(self):
1991: (8)               assert_equal(np.fmax.reduce([1, 2j]), 1)
1992: (4)               assert_equal(np.fmax.reduce([1+3j, 2j]), 1+3j)
1993: (8)             def test_float_nans(self):
1994: (8)               nan = np.nan

```

```

1994: (8)             arg1 = np.array([0,    nan,    nan])
1995: (8)             arg2 = np.array([nan,  0,    nan])
1996: (8)             out = np.array([0,    0,    nan])
1997: (8)             assert_equal(np.fmax(arg1, arg2), out)
1998: (4)             def test_complex_nans(self):
1999: (8)                 nan = np.nan
2000: (8)                 for cnan in [complex(nan, 0), complex(0, nan), complex(nan, nan)]:
2001: (12)                     arg1 = np.array([0, cnan, cnan], dtype=complex)
2002: (12)                     arg2 = np.array([cnan, 0, cnan], dtype=complex)
2003: (12)                     out = np.array([0,    0, nan], dtype=complex)
2004: (12)                     assert_equal(np.fmax(arg1, arg2), out)
2005: (4)             def test_precision(self):
2006: (8)                 dtypes = [np.float16, np.float32, np.float64, np.longdouble]
2007: (8)                 for dt in dtypes:
2008: (12)                     dtmin = np.finfo(dt).min
2009: (12)                     dtmax = np.finfo(dt).max
2010: (12)                     d1 = dt(0.1)
2011: (12)                     d1_next = np.nextafter(d1, np.inf)
2012: (12)                     test_cases = [
2013: (16)                         (dtmin, -np.inf,      dtmin),
2014: (16)                         (dtmax, -np.inf,      dtmax),
2015: (16)                         (d1,      d1_next,     d1_next),
2016: (16)                         (dtmax, np.nan,       dtmax),
2017: (12)
2018: (12)                     ]
2019: (16)                     for v1, v2, expected in test_cases:
2020: (16)                         assert_equal(np.fmax([v1], [v2]), [expected])
2021: (16)                         assert_equal(np.fmax.reduce([v1, v2]), expected)
2022: (0)             class TestFmin(_FilterInvalids):
2023: (4)                 def test_reduce(self):
2024: (8)                     dfilt = np.typecodes['AllFloat']
2025: (8)                     dint = np.typecodes['AllInteger']
2026: (8)                     seq1 = np.arange(11)
2027: (8)                     seq2 = seq1[::-1]
2028: (8)                     func = np.fmin.reduce
2029: (12)                     for dt in dint:
2030: (12)                         tmp1 = seq1.astype(dt)
2031: (12)                         tmp2 = seq2.astype(dt)
2032: (12)                         assert_equal(func(tmp1), 0)
2033: (8)                         assert_equal(func(tmp2), 0)
2034: (12)                     for dt in dfilt:
2035: (12)                         tmp1 = seq1.astype(dt)
2036: (12)                         tmp2 = seq2.astype(dt)
2037: (12)                         assert_equal(func(tmp1), 0)
2038: (12)                         assert_equal(func(tmp2), 0)
2039: (12)                         tmp1[::2] = np.nan
2040: (12)                         tmp2[::2] = np.nan
2041: (12)                         assert_equal(func(tmp1), 1)
2042: (12)                         assert_equal(func(tmp2), 1)
2043: (4)                 def test_reduce_complex(self):
2044: (8)                     assert_equal(np.fmin.reduce([1, 2j]), 2j)
2045: (8)                     assert_equal(np.fmin.reduce([1+3j, 2j]), 2j)
2046: (4)                 def test_float_nans(self):
2047: (8)                     nan = np.nan
2048: (8)                     arg1 = np.array([0,    nan,    nan])
2049: (8)                     arg2 = np.array([nan,  0,    nan])
2050: (8)                     out = np.array([0,    0,    nan])
2051: (4)                     assert_equal(np.fmin(arg1, arg2), out)
2052: (8)                 def test_complex_nans(self):
2053: (8)                     nan = np.nan
2054: (12)                     for cnan in [complex(nan, 0), complex(0, nan), complex(nan, nan)]:
2055: (12)                         arg1 = np.array([0, cnan, cnan], dtype=complex)
2056: (12)                         arg2 = np.array([cnan, 0, cnan], dtype=complex)
2057: (12)                         out = np.array([0,    0, nan], dtype=complex)
2058: (4)                         assert_equal(np.fmin(arg1, arg2), out)
2059: (8)                 def test_precision(self):
2060: (8)                     dtypes = [np.float16, np.float32, np.float64, np.longdouble]
2061: (12)                     for dt in dtypes:
2062: (12)                         dtmin = np.finfo(dt).min
2063: (12)                         dtmax = np.finfo(dt).max

```

```

2063: (12)             d1 = dt(0.1)
2064: (12)             d1_next = np.nextafter(d1, np.inf)
2065: (12)             test_cases = [
2066: (16)                 (dtmin, np.inf,      dtmin),
2067: (16)                 (dtmax, np.inf,      dtmax),
2068: (16)                 (d1,      d1_next,     d1),
2069: (16)                 (dtmin, np.nan,     dtmin),
2070: (12)             ]
2071: (12)             for v1, v2, expected in test_cases:
2072: (16)                 assert_equal(np.fmin([v1], [v2]), [expected])
2073: (16)                 assert_equal(np.fmin.reduce([v1, v2]), expected)
2074: (0)              class TestBool:
2075: (4)                def test_exceptions(self):
2076: (8)                  a = np.ones(1, dtype=np.bool_)
2077: (8)                  assert_raises(TypeError, np.negative, a)
2078: (8)                  assert_raises(TypeError, np.positive, a)
2079: (8)                  assert_raises(TypeError, np.subtract, a, a)
2080: (4)                def test_truth_table_logical(self):
2081: (8)                  input1 = [0, 0, 3, 2]
2082: (8)                  input2 = [0, 4, 0, 2]
2083: (8)                  typecodes = (np.typecodes['AllFloat']
2084: (21)                      + np.typecodes['AllInteger']
2085: (21)                      + '?')      # boolean
2086: (8)                  for dtype in map(np.dtype, typecodes):
2087: (12)                      arg1 = np.asarray(input1, dtype=dtype)
2088: (12)                      arg2 = np.asarray(input2, dtype=dtype)
2089: (12)                      out = [False, True, True, True]
2090: (12)                      for func in (np.logical_or, np.maximum):
2091: (16)                          assert_equal(func(arg1, arg2).astype(bool), out)
2092: (12)                      out = [False, False, False, True]
2093: (12)                      for func in (np.logical_and, np.minimum):
2094: (16)                          assert_equal(func(arg1, arg2).astype(bool), out)
2095: (12)                      out = [False, True, True, False]
2096: (12)                      for func in (np.logical_xor, np.not_equal):
2097: (16)                          assert_equal(func(arg1, arg2).astype(bool), out)
2098: (4)                def test_truth_table_bitwise(self):
2099: (8)                  arg1 = [False, False, True, True]
2100: (8)                  arg2 = [False, True, False, True]
2101: (8)                  out = [False, True, True, True]
2102: (8)                  assert_equal(np.bitwise_or(arg1, arg2), out)
2103: (8)                  out = [False, False, False, True]
2104: (8)                  assert_equal(np.bitwise_and(arg1, arg2), out)
2105: (8)                  out = [False, True, True, False]
2106: (8)                  assert_equal(np.bitwise_xor(arg1, arg2), out)
2107: (4)                def test_reduce(self):
2108: (8)                  none = np.array([0, 0, 0, 0], bool)
2109: (8)                  some = np.array([1, 0, 1, 1], bool)
2110: (8)                  every = np.array([1, 1, 1, 1], bool)
2111: (8)                  empty = np.array([], bool)
2112: (8)                  arrs = [none, some, every, empty]
2113: (8)                  for arr in arrs:
2114: (12)                      assert_equal(np.logical_and.reduce(arr), all(arr))
2115: (8)                  for arr in arrs:
2116: (12)                      assert_equal(np.logical_or.reduce(arr), any(arr))
2117: (8)                  for arr in arrs:
2118: (12)                      assert_equal(np.logical_xor.reduce(arr), arr.sum() % 2 == 1)
2119: (0)              class TestBitwiseUFuncs:
2120: (4)                bitwise_types = [np.dtype(c) for c in '?bBhHiIlLqQ' + 'O']
2121: (4)                def test_values(self):
2122: (8)                  for dt in self.bitwise_types:
2123: (12)                      zeros = np.array([0], dtype=dt)
2124: (12)                      ones = np.array([-1]).astype(dt)
2125: (12)                      msg = "dt = '%s'" % dt.char
2126: (12)                      assert_equal(np.bitwise_not(zeros), ones, err_msg=msg)
2127: (12)                      assert_equal(np.bitwise_not(ones), zeros, err_msg=msg)
2128: (12)                      assert_equal(np.bitwise_or(zeros, zeros), zeros, err_msg=msg)
2129: (12)                      assert_equal(np.bitwise_or(zeros, ones), ones, err_msg=msg)
2130: (12)                      assert_equal(np.bitwise_or(ones, zeros), ones, err_msg=msg)
2131: (12)                      assert_equal(np.bitwise_or(ones, ones), ones, err_msg=msg)

```

```

2132: (12) assert_equal(np.bitwise_xor(zeros, zeros), zeros, err_msg=msg)
2133: (12) assert_equal(np.bitwise_xor(zeros, ones), ones, err_msg=msg)
2134: (12) assert_equal(np.bitwise_xor(ones, zeros), ones, err_msg=msg)
2135: (12) assert_equal(np.bitwise_xor(ones, ones), zeros, err_msg=msg)
2136: (12) assert_equal(np.bitwise_and(zeros, zeros), zeros, err_msg=msg)
2137: (12) assert_equal(np.bitwise_and(zeros, ones), zeros, err_msg=msg)
2138: (12) assert_equal(np.bitwise_and(ones, zeros), zeros, err_msg=msg)
2139: (12) assert_equal(np.bitwise_and(ones, ones), ones, err_msg=msg)
2140: (4) def test_types(self):
2141: (8)     for dt in self.bitwise_types:
2142: (12)         zeros = np.array([0], dtype=dt)
2143: (12)         ones = np.array([-1]).astype(dt)
2144: (12)         msg = "dt = '%s'" % dt.char
2145: (12)         assert_(np.bitwise_not(zeros).dtype == dt, msg)
2146: (12)         assert_(np.bitwise_or(zeros, zeros).dtype == dt, msg)
2147: (12)         assert_(np.bitwise_xor(zeros, zeros).dtype == dt, msg)
2148: (12)         assert_(np.bitwise_and(zeros, zeros).dtype == dt, msg)
2149: (4) def test_identity(self):
2150: (8)     assert_(np.bitwise_or.identity == 0, 'bitwise_or')
2151: (8)     assert_(np.bitwise_xor.identity == 0, 'bitwise_xor')
2152: (8)     assert_(np.bitwise_and.identity == -1, 'bitwise_and')
2153: (4) def test_reduction(self):
2154: (8)     binary_funcs = (np.bitwise_or, np.bitwise_xor, np.bitwise_and)
2155: (8)     for dt in self.bitwise_types:
2156: (12)         zeros = np.array([0], dtype=dt)
2157: (12)         ones = np.array([-1]).astype(dt)
2158: (12)         for f in binary_funcs:
2159: (16)             msg = "dt: '%s', f: '%s'" % (dt, f)
2160: (16)             assert_equal(f.reduce(zeros), zeros, err_msg=msg)
2161: (16)             assert_equal(f.reduce(ones), ones, err_msg=msg)
2162: (8)         for dt in self.bitwise_types[:-1]:
2163: (12)             empty = np.array([], dtype=dt)
2164: (12)             for f in binary_funcs:
2165: (16)                 msg = "dt: '%s', f: '%s'" % (dt, f)
2166: (16)                 tgt = np.array(f.identity).astype(dt)
2167: (16)                 res = f.reduce(empty)
2168: (16)                 assert_equal(res, tgt, err_msg=msg)
2169: (16)                 assert_(res.dtype == tgt.dtype, msg)
2170: (8)             for f in binary_funcs:
2171: (12)                 msg = "dt: '%s'" % (f,)
2172: (12)                 empty = np.array([], dtype=object)
2173: (12)                 tgt = f.identity
2174: (12)                 res = f.reduce(empty)
2175: (12)                 assert_equal(res, tgt, err_msg=msg)
2176: (8)             for f in binary_funcs:
2177: (12)                 msg = "dt: '%s'" % (f,)
2178: (12)                 btype = np.array([True], dtype=object)
2179: (12)                 assert_(type(f.reduce(btype)) is bool, msg)
2180: (0) class TestInt:
2181: (4)     def test_logical_not(self):
2182: (8)         x = np.ones(10, dtype=np.int16)
2183: (8)         o = np.ones(10 * 2, dtype=bool)
2184: (8)         tgt = o.copy()
2185: (8)         tgt[::2] = False
2186: (8)         os = o[::2]
2187: (8)         assert_array_equal(np.logical_not(x, out=os), False)
2188: (8)         assert_array_equal(o, tgt)
2189: (0) class TestFloatingPoint:
2190: (4)     def test_floating_point(self):
2191: (8)         assert_equal(ncu.FLOATING_POINT_SUPPORT, 1)
2192: (0) class TestDegrees:
2193: (4)     def test_degrees(self):
2194: (8)         assert_almost_equal(ncu.degrees(np.pi), 180.0)
2195: (8)         assert_almost_equal(ncu.degrees(-0.5*np.pi), -90.0)
2196: (0) class TestRadians:
2197: (4)     def test_radians(self):
2198: (8)         assert_almost_equal(ncu.radians(180.0), np.pi)
2199: (8)         assert_almost_equal(ncu.radians(-90.0), -0.5*np.pi)
2200: (0) class TestHeavside:

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

2201: (4)
2202: (8)
np.inf]])
2203: (8)
0.0)])
2204: (8)
2205: (8)
2206: (8)
2207: (8)
2208: (8)
2209: (8)
2210: (8)
2211: (8)
2212: (8)
2213: (8)
2214: (8)
2215: (0)
2216: (4)
2217: (8)
2218: (8)
2219: (8)
2220: (8)
2221: (12)
2222: (12)
2223: (12)
2224: (12)
2225: (12)
2226: (4)
2227: (8)
2228: (8)
2229: (8)
2230: (8)
2231: (4)
2232: (8)
2233: (12)
2234: (12)
2235: (8)
2236: (0)
2237: (4)
2238: (8)
2239: (12)
2240: (53)
2241: (16)
2242: (20)
2243: (20)
2244: (20)
2245: (20)
2246: (24)
2247: (35)
2248: (24)
2249: (24)
2250: (20)
2251: (20)
2252: (20)
2253: (20)
2254: (4)
2255: (8)
2256: (8)
2257: (8)
2258: (4)
2259: (8)
2260: (12)
2261: (16)
2262: (20)
2263: (4)
2264: (8)
2265: (8)
2266: (0)
2267: (4)

def test_heaviside(self):
    x = np.array([-30.0, -0.1, 0.0, 0.2], [7.5, np.nan, np.inf, -
expectedhalf = np.array([[0.0, 0.0, 0.5, 1.0], [1.0, np.nan, 1.0,
0.0]])
expected1 = expectedhalf.copy()
expected1[0, 2] = 1
h = ncu.heaviside(x, 0.5)
assert_equal(h, expectedhalf)
h = ncu.heaviside(x, 1.0)
assert_equal(h, expected1)
x = x.astype(np.float32)
h = ncu.heaviside(x, np.float32(0.5))
assert_equal(h, expectedhalf.astype(np.float32))
h = ncu.heaviside(x, np.float32(1.0))
assert_equal(h, expected1.astype(np.float32))

class TestSign:
    def test_sign(self):
        a = np.array([np.inf, -np.inf, np.nan, 0.0, 3.0, -3.0])
        out = np.zeros(a.shape)
        tgt = np.array([1., -1., np.nan, 0.0, 1.0, -1.0])
        with np.errstate(invalid='ignore'):
            res = ncu.sign(a)
            assert_equal(res, tgt)
            res = ncu.sign(a, out)
            assert_equal(res, tgt)
            assert_equal(out, tgt)

    def test_sign_dtype_object(self):
        foo = np.array([-1, 0, 1])
        a = np.sign(foo.astype(object))
        b = np.sign(foo)
        assert_array_equal(a, b)

    def test_sign_dtype_nan_object(self):
        def test_nan():
            foo = np.array([np.nan])
            a = np.sign(foo.astype(object))
            assert_raises(TypeError, test_nan)

class TestMinMax:
    def test_minmax_blocked(self):
        for dt, sz in [(np.float32, 15), (np.float64, 7)]:
            for out, inp, msg in _gen_alignment_data(dtype=dt, type='unary',
max_size=sz):
                for i in range(inp.size):
                    inp[:] = np.arange(inp.size, dtype=dt)
                    inp[i] = np.nan
                    emsg = lambda: '%r\n%s' % (inp, msg)
                    with suppress_warnings() as sup:
                        sup.filter(RuntimeWarning,
                                   "invalid value encountered in reduce")
                        assert_(np.isnan(inp.max()), msg=emsg)
                        assert_(np.isnan(inp.min()), msg=emsg)
                    inp[i] = 1e10
                    assert_equal(inp.max(), 1e10, err_msg=msg)
                    inp[i] = -1e10
                    assert_equal(inp.min(), -1e10, err_msg=msg)

    def test_lower_align(self):
        d = np.zeros(23 * 8, dtype=np.int8)[4:-4].view(np.float64)
        assert_equal(d.max(), d[0])
        assert_equal(d.min(), d[0])

    def test_reduce_reorder(self):
        for n in (2, 4, 8, 16, 32):
            for dt in (np.float32, np.float16, np.complex64):
                for r in np.diagflat(np.array([np.nan] * n, dtype=dt)):
                    assert_equal(np.min(r), np.nan)

    def test_minimize_no_warns(self):
        a = np.minimum(np.nan, 1)
        assert_equal(a, np.nan)

class TestAbsoluteNegative:
    def test_abs_neg_blocked(self):

```

```

2268: (8)           for dt, sz in [(np.float32, 11), (np.float64, 5)]:
2269: (12)             for out, inp, msg in _gen_alignment_data(dtype=dt, type='unary',
2270: (53)                           max_size=sz):
2271: (16)               tgt = [ncu.absolute(i) for i in inp]
2272: (16)               np.absolute(inp, out=out)
2273: (16)               assert_equal(out, tgt, err_msg=msg)
2274: (16)               assert_(out >= 0).all()
2275: (16)               tgt = [-1*(i) for i in inp]
2276: (16)               np.negative(inp, out=out)
2277: (16)               assert_equal(out, tgt, err_msg=msg)
2278: (16)               for v in [np.nan, -np.inf, np.inf]:
2279: (20)                 for i in range(inp.size):
2280: (24)                   d = np.arange(inp.size, dtype=dt)
2281: (24)                   inp[:] = -d
2282: (24)                   inp[i] = v
2283: (24)                   d[i] = -v if v == -np.inf else v
2284: (24)                   assert_array_equal(np.abs(inp), d, err_msg=msg)
2285: (24)                   np.abs(inp, out=out)
2286: (24)                   assert_array_equal(out, d, err_msg=msg)
2287: (24)                   assert_array_equal(-inp, -1*inp, err_msg=msg)
2288: (24)                   d = -1 * inp
2289: (24)                   np.negative(inp, out=out)
2290: (24)                   assert_array_equal(out, d, err_msg=msg)
2291: (4)           def test_lower_align(self):
2292: (8)             d = np.zeros(23 * 8, dtype=np.int8)[4:-4].view(np.float64)
2293: (8)             assert_equal(np.abs(d), d)
2294: (8)             assert_equal(np.negative(d), -d)
2295: (8)             np.negative(d, out=d)
2296: (8)             np.negative(np.ones_like(d), out=d)
2297: (8)             np.abs(d, out=d)
2298: (8)             np.abs(np.ones_like(d), out=d)
2299: (4)             @pytest.mark.parametrize("dtype", ['d', 'f', 'int32', 'int64'])
2300: (4)             @pytest.mark.parametrize("big", [True, False])
2301: (4)             def test_noncontiguous(self, dtype, big):
2302: (8)               data = np.array([-1.0, 1.0, -0.0, 0.0, 2.2251e-308, -2.5, 2.5, -6,
2303: (28)                         6, -2.2251e-308, -8, 10], dtype=dtype)
2304: (8)               expect = np.array([1.0, -1.0, 0.0, -0.0, -2.2251e-308, 2.5, -2.5, 6,
2305: (28)                         -6, 2.2251e-308, 8, -10], dtype=dtype)
2306: (8)               if big:
2307: (12)                 data = np.repeat(data, 10)
2308: (12)                 expect = np.repeat(expect, 10)
2309: (8)               out = np.ndarray(data.shape, dtype=dtype)
2310: (8)               ncontig_in = data[1::2]
2311: (8)               ncontig_out = out[1::2]
2312: (8)               contig_in = np.array(ncontig_in)
2313: (8)               assert_array_equal(np.negative(contig_in), expect[1::2])
2314: (8)               assert_array_equal(np.negative(contig_in, out=ncontig_out),
2315: (32)                             expect[1::2])
2316: (8)               assert_array_equal(np.negative(ncontig_in), expect[1::2])
2317: (8)               assert_array_equal(np.negative(ncontig_in, out=ncontig_out),
2318: (32)                             expect[1::2])
2319: (8)               data_split = np.array(np.array_split(data, 2))
2320: (8)               expect_split = np.array(np.array_split(expect, 2))
2321: (8)               assert_equal(np.negative(data_split), expect_split)
2322: (0)           class TestPositive:
2323: (4)             def test_valid(self):
2324: (8)               valid_dtypes = [int, float, complex, object]
2325: (8)               for dtype in valid_dtypes:
2326: (12)                 x = np.arange(5, dtype=dtype)
2327: (12)                 result = np.positive(x)
2328: (12)                 assert_equal(x, result, err_msg=str(dtype))
2329: (4)             def test_invalid(self):
2330: (8)               with assert_raises(TypeError):
2331: (12)                 np.positive(True)
2332: (8)               with assert_raises(TypeError):
2333: (12)                 np.positive(np.datetime64('2000-01-01'))
2334: (8)               with assert_raises(TypeError):
2335: (12)                 np.positive(np.array(['foo']), dtype=str))
2336: (8)               with assert_raises(TypeError):

```

```

2337: (12) np.positive(np.array(['bar'], dtype=object))
2338: (0)
2339: (4)
2340: (8)
2341: (12)
2342: (16)
2343: (12)
2344: (16)
2345: (16)
2346: (16)
2347: (16)
2348: (8)
2349: (8)
2350: (8) x = ncu.minimum(a, a)
2351: (8) assert_equal(x.arr, np.zeros(1))
2352: (8) func, args, i = x.context
2353: (8) assert_(func is ncu.minimum)
2354: (8) assert_equal(len(args), 2)
2355: (8) assert_equal(args[0], a)
2356: (8) assert_equal(args[1], a)
2357: (8) assert_equal(i, 0)
2358: (4) def test_wrap_and_prepare_out(self):
2359: (8)     class StoreArrayPrepareWrap(np.ndarray):
2360: (12)         _wrap_args = None
2361: (12)         _prepare_args = None
2362: (12)         def __new__(cls):
2363: (16)             return np.zeros().view(cls)
2364: (12)         def __array_wrap__(self, obj, context):
2365: (16)             self._wrap_args = context[1]
2366: (12)             return obj
2367: (16)         def __array_prepare__(self, obj, context):
2368: (16)             self._prepare_args = context[1]
2369: (12)             return obj
2370: (12)         @property
2371: (16)         def args(self):
2372: (12)             return (self._prepare_args, self._wrap_args)
2373: (16)         def __repr__(self):
2374: (8)             return "a" # for short test output
2375: (12)     def do_test(f_call, f_expected):
2376: (12)         a = StoreArrayPrepareWrap()
2377: (12)         f_call(a)
2378: (12)         p, w = a.args
2379: (12)         expected = f_expected(a)
2380: (12)         try:
2381: (16)             assert_equal(p, expected)
2382: (12)             assert_equal(w, expected)
2383: (16)         except AssertionError as e:
2384: (20)             raise AssertionError("\n".join([
2385: (20)                 "Bad arguments passed in ufunc call",
2386: (20)                 " expected:           {}".format(expected),
2387: (20)                 " __array_prepare__ got: {}".format(p),
2388: (16)                 " __array_wrap__ got:   {}".format(w)
2389: (8)             ]))
2390: (8)         do_test(lambda a: np.add(a, 0), lambda a: (a, 0))
2391: (8)         do_test(lambda a: np.add(a, 0, None), lambda a: (a, 0))
2392: (8)         do_test(lambda a: np.add(a, 0, out=None), lambda a: (a, 0))
2393: (8)         do_test(lambda a: np.add(a, 0, out=(None,)), lambda a: (a, 0))
2394: (8)         do_test(lambda a: np.add(0, 0, a), lambda a: (0, 0, a))
2395: (8)         do_test(lambda a: np.add(0, 0, out=a), lambda a: (0, 0, a))
2396: (8)         do_test(lambda a: np.add(0, 0, out=(a,)), lambda a: (0, 0, a))
2397: (8)         do_test(lambda a: np.add(a, 0, where=False), lambda a: (a, 0))
2398: (4)         do_test(lambda a: np.add(0, 0, a, where=False), lambda a: (0, 0, a))
2399: (8)     def test_wrap_with_iterable(self):
2400: (12)         class with_wrap(np.ndarray):
2401: (12)             __array_priority__ = 10
2402: (16)             def __new__(cls):
2403: (12)                 return np.asarray(1).view(cls).copy()
2404: (16)             def __array_wrap__(self, arr, context):
2405: (8)                 return arr.view(type(self))

```

```

2406: (8)           x = ncu.multiply(a, (1, 2, 3))
2407: (8)           assert_(isinstance(x, with_wrap))
2408: (8)           assert_array_equal(x, np.array((1, 2, 3)))
2409: (4)           def test_priority_with_scalar(self):
2410: (8)             class A(np.ndarray):
2411: (12)               __array_priority__ = 10
2412: (12)               def __new__(cls):
2413: (16)                 return np.asarray(1.0, 'float64').view(cls).copy()
2414: (8)
2415: (8)             a = A()
2416: (8)             x = np.float64(1)*a
2417: (8)             assert_(isinstance(x, A))
2418: (4)             assert_array_equal(x, np.array(1))
2419: (8)           def test_old_wrap(self):
2420: (12)             class with_wrap:
2421: (16)               def __array__(self):
2422: (12)                 return np.zeros(1)
2423: (16)               def __array_wrap__(self, arr):
2424: (16)                 r = with_wrap()
2425: (16)                 r.arr = arr
2426: (16)                 return r
2427: (8)             a = with_wrap()
2428: (8)             x = ncu.minimum(a, a)
2429: (8)             assert_equal(x.arr, np.zeros(1))
2430: (8)           def test_priority(self):
2431: (12)             class A:
2432: (16)               def __array__(self):
2433: (12)                 return np.zeros(1)
2434: (16)               def __array_wrap__(self, arr, context):
2435: (16)                 r = type(self)()
2436: (16)                 r.arr = arr
2437: (16)                 r.context = context
2438: (8)                 return r
2439: (12)             class B(A):
2440: (8)               __array_priority__ = 20.
2441: (12)             class C(A):
2442: (8)               __array_priority__ = 40.
2443: (8)             x = np.zeros(1)
2444: (8)             a = A()
2445: (8)             b = B()
2446: (8)             c = C()
2447: (8)             f = ncu.minimum
2448: (8)             assert_(type(f(x, x)) is np.ndarray)
2449: (8)             assert_(type(f(x, a)) is A)
2450: (8)             assert_(type(f(x, b)) is B)
2451: (8)             assert_(type(f(x, c)) is C)
2452: (8)             assert_(type(f(a, x)) is A)
2453: (8)             assert_(type(f(b, x)) is B)
2454: (8)             assert_(type(f(c, x)) is C)
2455: (8)             assert_(type(f(a, a)) is A)
2456: (8)             assert_(type(f(a, b)) is B)
2457: (8)             assert_(type(f(b, a)) is B)
2458: (8)             assert_(type(f(b, b)) is B)
2459: (8)             assert_(type(f(b, c)) is C)
2460: (8)             assert_(type(f(c, b)) is C)
2461: (8)             assert_(type(f(c, c)) is C)
2462: (8)             assert_(type(ncu.exp(a) is A))
2463: (8)             assert_(type(ncu.exp(b) is B))
2464: (4)             assert_(type(ncu.exp(c) is C))
2465: (8)           def test_failing_wrap(self):
2466: (12)             class A:
2467: (16)               def __array__(self):
2468: (12)                 return np.zeros(2)
2469: (16)               def __array_wrap__(self, arr, context):
2470: (8)                 raise RuntimeError
2471: (8)             a = A()
2472: (8)             assert_raises(RuntimeError, ncu.maximum, a, a)
2473: (8)             assert_raises(RuntimeError, ncu.maximum.reduce, a)
2474: (4)           def test_failing_out_wrap(self):
2475: (8)             singleton = np.array([1.0])

```

```

2475: (8)         class Ok(np.ndarray):
2476: (12)           def __array_wrap__(self, obj):
2477: (16)             return singleton
2478: (8)         class Bad(np.ndarray):
2479: (12)           def __array_wrap__(self, obj):
2480: (16)             raise RuntimeError
2481: (8)           ok = np.empty(1).view(Ok)
2482: (8)           bad = np.empty(1).view(Bad)
2483: (8)           for i in range(10):
2484: (12)             assert_raises(RuntimeError, ncu.frexp, 1, ok, bad)
2485: (4)       def test_none_wrap(self):
2486: (8)           class A:
2487: (12)             def __array__(self):
2488: (16)               return np.zeros(1)
2489: (12)             def __array_wrap__(self, arr, context=None):
2490: (16)               return None
2491: (8)           a = A()
2492: (8)           assert_equal(ncu.maximum(a, a), None)
2493: (4)       def test_default_prepare(self):
2494: (8)           class with_wrap:
2495: (12)             __array_priority__ = 10
2496: (12)             def __array__(self):
2497: (16)               return np.zeros(1)
2498: (12)             def __array_wrap__(self, arr, context):
2499: (16)               return arr
2500: (8)           a = with_wrap()
2501: (8)           x = ncu.minimum(a, a)
2502: (8)           assert_equal(x, np.zeros(1))
2503: (8)           assert_equal(type(x), np.ndarray)
2504: (4) @pytest.mark.parametrize("use_where", [True, False])
2505: (4)       def test_prepare(self, use_where):
2506: (8)           class with_prepare(np.ndarray):
2507: (12)             __array_priority__ = 10
2508: (12)             def __array_prepare__(self, arr, context):
2509: (16)               return np.array(arr).view(type=with_prepare)
2510: (8)           a = np.array(1).view(type=with_prepare)
2511: (8)           if use_where:
2512: (12)             x = np.add(a, a, where=np.array(True))
2513: (8)           else:
2514: (12)             x = np.add(a, a)
2515: (8)           assert_equal(x, np.array(2))
2516: (8)           assert_equal(type(x), with_prepare)
2517: (4) @pytest.mark.parametrize("use_where", [True, False])
2518: (4)       def test_prepare_out(self, use_where):
2519: (8)           class with_prepare(np.ndarray):
2520: (12)             __array_priority__ = 10
2521: (12)             def __array_prepare__(self, arr, context):
2522: (16)               return np.array(arr).view(type=with_prepare)
2523: (8)           a = np.array([1]).view(type=with_prepare)
2524: (8)           if use_where:
2525: (12)             x = np.add(a, a, a, where=[True])
2526: (8)           else:
2527: (12)             x = np.add(a, a, a)
2528: (8)           assert_(not np.shares_memory(x, a))
2529: (8)           assert_equal(x, np.array([2]))
2530: (8)           assert_equal(type(x), with_prepare)
2531: (4)       def test_failing_prepare(self):
2532: (8)           class A:
2533: (12)             def __array__(self):
2534: (16)               return np.zeros(1)
2535: (12)             def __array_prepare__(self, arr, context=None):
2536: (16)               raise RuntimeError
2537: (8)           a = A()
2538: (8)           assert_raises(RuntimeError, ncu.maximum, a, a)
2539: (8)           assert_raises(RuntimeError, ncu.maximum, a, a, where=False)
2540: (4)       def test_array_too_many_args(self):
2541: (8)           class A:
2542: (12)             def __array__(self, dtype, context):
2543: (16)               return np.zeros(1)

```

```

2544: (8)             a = A()
2545: (8)             assert_raises_regex(TypeError, '2 required positional', np.sum, a)
2546: (4)             def test_ufunc_override(self):
2547: (8)                 class A:
2548: (12)                     def __array_ufunc__(self, func, method, *inputs, **kwargs):
2549: (16)                         return self, func, method, inputs, kwargs
2550: (8)             class MyNDArray(np.ndarray):
2551: (12)                 __array_priority__ = 100
2552: (8)             a = A()
2553: (8)             b = np.array([1]).view(MyNDArray)
2554: (8)             res0 = np.multiply(a, b)
2555: (8)             res1 = np.multiply(b, b, out=a)
2556: (8)             assert_equal(res0[0], a)
2557: (8)             assert_equal(res1[0], a)
2558: (8)             assert_equal(res0[1], np.multiply)
2559: (8)             assert_equal(res1[1], np.multiply)
2560: (8)             assert_equal(res0[2], '__call__')
2561: (8)             assert_equal(res1[2], '__call__')
2562: (8)             assert_equal(res0[3], (a, b))
2563: (8)             assert_equal(res1[3], (b, b))
2564: (8)             assert_equal(res0[4], {})
2565: (8)             assert_equal(res1[4], {'out': (a,)})
2566: (4)             def test_ufunc_override_mro(self):
2567: (8)                 def tres_mul(a, b, c):
2568: (12)                     return a * b * c
2569: (8)                 def quattro_mul(a, b, c, d):
2570: (12)                     return a * b * c * d
2571: (8)             three_mul_ufunc = np.frompyfunc(tres_mul, 3, 1)
2572: (8)             four_mul_ufunc = np.frompyfunc(quattro_mul, 4, 1)
2573: (8)             class A:
2574: (12)                 def __array_ufunc__(self, func, method, *inputs, **kwargs):
2575: (16)                     return "A"
2576: (8)             class ASub(A):
2577: (12)                 def __array_ufunc__(self, func, method, *inputs, **kwargs):
2578: (16)                     return "ASub"
2579: (8)             class B:
2580: (12)                 def __array_ufunc__(self, func, method, *inputs, **kwargs):
2581: (16)                     return "B"
2582: (8)             class C:
2583: (12)                 def __init__(self):
2584: (16)                     self.count = 0
2585: (12)                 def __array_ufunc__(self, func, method, *inputs, **kwargs):
2586: (16)                     self.count += 1
2587: (16)                     return NotImplemented
2588: (8)             class CSub(C):
2589: (12)                 def __array_ufunc__(self, func, method, *inputs, **kwargs):
2590: (16)                     self.count += 1
2591: (16)                     return NotImplemented
2592: (8)             a = A()
2593: (8)             a_sub = ASub()
2594: (8)             b = B()
2595: (8)             c = C()
2596: (8)             res = np.multiply(a, a_sub)
2597: (8)             assert_equal(res, "ASub")
2598: (8)             res = np.multiply(a_sub, b)
2599: (8)             assert_equal(res, "ASub")
2600: (8)             res = np.multiply(c, a)
2601: (8)             assert_equal(res, "A")
2602: (8)             assert_equal(c.count, 1)
2603: (8)             res = np.multiply(c, a)
2604: (8)             assert_equal(c.count, 2)
2605: (8)             c = C()
2606: (8)             c_sub = CSub()
2607: (8)             assert_raises(TypeError, np.multiply, c, c_sub)
2608: (8)             assert_equal(c.count, 1)
2609: (8)             assert_equal(c_sub.count, 1)
2610: (8)             c.count = c_sub.count = 0
2611: (8)             assert_raises(TypeError, np.multiply, c_sub, c)
2612: (8)             assert_equal(c.count, 1)

```

```

2613: (8)             assert_equal(c_sub.count, 1)
2614: (8)             c.count = 0
2615: (8)             assert_raises(TypeError, np.multiply, c, c)
2616: (8)             assert_equal(c.count, 1)
2617: (8)             c.count = 0
2618: (8)             assert_raises(TypeError, np.multiply, 2, c)
2619: (8)             assert_equal(c.count, 1)
2620: (8)             assert_equal(three_mul_ufunc(a, 1, 2), "A")
2621: (8)             assert_equal(three_mul_ufunc(1, a, 2), "A")
2622: (8)             assert_equal(three_mul_ufunc(1, 2, a), "A")
2623: (8)             assert_equal(three_mul_ufunc(a, a, 6), "A")
2624: (8)             assert_equal(three_mul_ufunc(a, 2, a), "A")
2625: (8)             assert_equal(three_mul_ufunc(a, 2, b), "A")
2626: (8)             assert_equal(three_mul_ufunc(a, 2, a_sub), "ASub")
2627: (8)             assert_equal(three_mul_ufunc(a, a_sub, 3), "ASub")
2628: (8)             c.count = 0
2629: (8)             assert_equal(three_mul_ufunc(c, a_sub, 3), "ASub")
2630: (8)             assert_equal(c.count, 1)
2631: (8)             c.count = 0
2632: (8)             assert_equal(three_mul_ufunc(1, a_sub, c), "ASub")
2633: (8)             assert_equal(c.count, 0)
2634: (8)             c.count = 0
2635: (8)             assert_equal(three_mul_ufunc(a, b, c), "A")
2636: (8)             assert_equal(c.count, 0)
2637: (8)             c_sub.count = 0
2638: (8)             assert_equal(three_mul_ufunc(a, b, c_sub), "A")
2639: (8)             assert_equal(c_sub.count, 0)
2640: (8)             assert_equal(three_mul_ufunc(1, 2, b), "B")
2641: (8)             assert_raises(TypeError, three_mul_ufunc, 1, 2, c)
2642: (8)             assert_raises(TypeError, three_mul_ufunc, c_sub, 2, c)
2643: (8)             assert_raises(TypeError, three_mul_ufunc, c_sub, 2, 3)
2644: (8)             assert_equal(four_mul_ufunc(a, 1, 2, 3), "A")
2645: (8)             assert_equal(four_mul_ufunc(1, a, 2, 3), "A")
2646: (8)             assert_equal(four_mul_ufunc(1, 1, a, 3), "A")
2647: (8)             assert_equal(four_mul_ufunc(1, 1, 2, a), "A")
2648: (8)             assert_equal(four_mul_ufunc(a, b, 2, 3), "A")
2649: (8)             assert_equal(four_mul_ufunc(1, a, 2, b), "A")
2650: (8)             assert_equal(four_mul_ufunc(b, 1, a, 3), "B")
2651: (8)             assert_equal(four_mul_ufunc(a_sub, 1, 2, a), "ASub")
2652: (8)             assert_equal(four_mul_ufunc(a, 1, 2, a_sub), "ASub")
2653: (8)             c = C()
2654: (8)             c_sub = CSub()
2655: (8)             assert_raises(TypeError, four_mul_ufunc, 1, 2, 3, c)
2656: (8)             assert_equal(c.count, 1)
2657: (8)             c.count = 0
2658: (8)             assert_raises(TypeError, four_mul_ufunc, 1, 2, c_sub, c)
2659: (8)             assert_equal(c_sub.count, 1)
2660: (8)             assert_equal(c.count, 1)
2661: (8)             c2 = C()
2662: (8)             c.count = c_sub.count = 0
2663: (8)             assert_raises(TypeError, four_mul_ufunc, 1, c, c_sub, c2)
2664: (8)             assert_equal(c_sub.count, 1)
2665: (8)             assert_equal(c.count, 1)
2666: (8)             assert_equal(c2.count, 0)
2667: (8)             c.count = c2.count = c_sub.count = 0
2668: (8)             assert_raises(TypeError, four_mul_ufunc, c2, c, c_sub, c)
2669: (8)             assert_equal(c_sub.count, 1)
2670: (8)             assert_equal(c.count, 0)
2671: (8)             assert_equal(c2.count, 1)
2672: (4)             def test_ufunc_override_methods(self):
2673: (8)                 class A:
2674: (12)                     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
2675: (16)                         return self, ufunc, method, inputs, kwargs
2676: (8)                     a = A()
2677: (8)                     with assert_raises(TypeError):
2678: (12)                         np.multiply.__call__(1, a, foo='bar', answer=42)
2679: (8)                         res = np.multiply.__call__(1, a, subok='bar', where=42)
2680: (8)                         assert_equal(res[0], a)
2681: (8)                         assert_equal(res[1], np.multiply)

```

```

2682: (8) assert_equal(res[2], '__call__')
2683: (8) assert_equal(res[3], (1, a))
2684: (8) assert_equal(res[4], {'subok': 'bar', 'where': 42})
2685: (8) assert_raises(TypeError, np.multiply, a)
2686: (8) assert_raises(TypeError, np.multiply, a, a, a, a)
2687: (8) assert_raises(TypeError, np.multiply, a, a, sig='a', signature='a')
2688: (8) assert_raises(TypeError, ncu_tests.inner1d, a, a, axis=0, axes=[0, 0])
2689: (8) res = np.multiply.reduce(a, 'axis0', 'dtype0', 'out0', 'keep0')
2690: (8) assert_equal(res[0], a)
2691: (8) assert_equal(res[1], np.multiply)
2692: (8) assert_equal(res[2], 'reduce')
2693: (8) assert_equal(res[3], (a,))
2694: (8) assert_equal(res[4], {'dtype': 'dtype0',
2695: (30) 'out': ('out0',),
2696: (30) 'keepdims': 'keep0',
2697: (30) 'axis': 'axis0'})
2698: (8) res = np.multiply.reduce(a, axis='axis0', dtype='dtype0', out='out0',
2699: (33) keepdims='keep0', initial='init0',
2700: (33) where='where0')
2701: (8) assert_equal(res[0], a)
2702: (8) assert_equal(res[1], np.multiply)
2703: (8) assert_equal(res[2], 'reduce')
2704: (8) assert_equal(res[3], (a,))
2705: (8) assert_equal(res[4], {'dtype': 'dtype0',
2706: (30) 'out': ('out0',),
2707: (30) 'keepdims': 'keep0',
2708: (30) 'axis': 'axis0',
2709: (30) 'initial': 'init0',
2710: (30) 'where': 'where0'})
2711: (8) res = np.multiply.reduce(a, 0, None, None, False)
2712: (8) assert_equal(res[4], {'axis': 0, 'dtype': None, 'keepdims': False})
2713: (8) res = np.multiply.reduce(a, out=None, axis=0, keepdims=True)
2714: (8) assert_equal(res[4], {'axis': 0, 'keepdims': True})
2715: (8) res = np.multiply.reduce(a, None, out=(None,), dtype=None)
2716: (8) assert_equal(res[4], {'axis': None, 'dtype': None})
2717: (8) res = np.multiply.reduce(a, 0, None, None, False, 2, True)
2718: (8) assert_equal(res[4], {'axis': 0, 'dtype': None, 'keepdims': False,
2719: (30) 'initial': 2, 'where': True})
2720: (8) res = np.multiply.reduce(a, 0, None, None, False,
2721: (33) np._NoValue, True)
2722: (8) assert_equal(res[4], {'axis': 0, 'dtype': None, 'keepdims': False,
2723: (30) 'where': True})
2724: (8) res = np.multiply.reduce(a, 0, None, None, False, None, True)
2725: (8) assert_equal(res[4], {'axis': 0, 'dtype': None, 'keepdims': False,
2726: (30) 'initial': None, 'where': True})
2727: (8) assert_raises(ValueError, np.multiply.reduce, a, out=())
2728: (8) assert_raises(ValueError, np.multiply.reduce, a, out=('out0', 'out1'))
2729: (8) assert_raises(TypeError, np.multiply.reduce, a, 'axis0', axis='axis0')
2730: (8) res = np.multiply.accumulate(a, 'axis0', 'dtype0', 'out0')
2731: (8) assert_equal(res[0], a)
2732: (8) assert_equal(res[1], np.multiply)
2733: (8) assert_equal(res[2], 'accumulate')
2734: (8) assert_equal(res[3], (a,))
2735: (8) assert_equal(res[4], {'dtype': 'dtype0',
2736: (30) 'out': ('out0',),
2737: (30) 'axis': 'axis0'})
2738: (8) res = np.multiply.accumulate(a, axis='axis0', dtype='dtype0',
2739: (37) out='out0')
2740: (8) assert_equal(res[0], a)
2741: (8) assert_equal(res[1], np.multiply)
2742: (8) assert_equal(res[2], 'accumulate')
2743: (8) assert_equal(res[3], (a,))
2744: (8) assert_equal(res[4], {'dtype': 'dtype0',
2745: (30) 'out': ('out0',),
2746: (30) 'axis': 'axis0'})
2747: (8) res = np.multiply.accumulate(a, 0, None, None)
2748: (8) assert_equal(res[4], {'axis': 0, 'dtype': None})
2749: (8) res = np.multiply.accumulate(a, out=None, axis=0, dtype='dtype1')
2750: (8) assert_equal(res[4], {'axis': 0, 'dtype': 'dtype1'})

```

```

2751: (8)             res = np.multiply.accumulate(a, None, out=(None,), dtype=None)
2752: (8)             assert_equal(res[4], {'axis': None, 'dtype': None})
2753: (8)             assert_raises(ValueError, np.multiply.accumulate, a, out=())
2754: (8)             assert_raises(ValueError, np.multiply.accumulate, a,
2755: (22)                 out=('out0', 'out1'))
2756: (8)             assert_raises(TypeError, np.multiply.accumulate, a,
2757: (22)                 'axis0', axis='axis0')
2758: (8)             res = np.multiply.reduceat(a, [4, 2], 'axis0', 'dtype0', 'out0')
2759: (8)             assert_equal(res[0], a)
2760: (8)             assert_equal(res[1], np.multiply)
2761: (8)             assert_equal(res[2], 'reduceat')
2762: (8)             assert_equal(res[3], (a, [4, 2]))
2763: (8)             assert_equal(res[4], {'dtype': 'dtype0',
2764: (30)                 'out': ('out0',),
2765: (30)                 'axis': 'axis0'})
2766: (8)             res = np.multiply.reduceat(a, [4, 2], axis='axis0', dtype='dtype0',
2767: (35)                 out='out0')
2768: (8)             assert_equal(res[0], a)
2769: (8)             assert_equal(res[1], np.multiply)
2770: (8)             assert_equal(res[2], 'reduceat')
2771: (8)             assert_equal(res[3], (a, [4, 2]))
2772: (8)             assert_equal(res[4], {'dtype': 'dtype0',
2773: (30)                 'out': ('out0',),
2774: (30)                 'axis': 'axis0'})
2775: (8)             res = np.multiply.reduceat(a, [4, 2], 0, None, None)
2776: (8)             assert_equal(res[4], {'axis': 0, 'dtype': None})
2777: (8)             res = np.multiply.reduceat(a, [4, 2], axis=None, out=None, dtype='dt')
2778: (8)             assert_equal(res[4], {'axis': None, 'dtype': 'dt'})
2779: (8)             res = np.multiply.reduceat(a, [4, 2], None, None, out=(None,))
2780: (8)             assert_equal(res[4], {'axis': None, 'dtype': None})
2781: (8)             assert_raises(ValueError, np.multiply.reduce, a, [4, 2], out=())
2782: (8)             assert_raises(ValueError, np.multiply.reduce, a, [4, 2],
2783: (22)                 out=('out0', 'out1'))
2784: (8)             assert_raises(TypeError, np.multiply.reduce, a, [4, 2],
2785: (22)                 'axis0', axis='axis0')
2786: (8)             res = np.multiply.outer(a, 42)
2787: (8)             assert_equal(res[0], a)
2788: (8)             assert_equal(res[1], np.multiply)
2789: (8)             assert_equal(res[2], 'outer')
2790: (8)             assert_equal(res[3], (a, 42))
2791: (8)             assert_equal(res[4], {})
2792: (8)             assert_raises(TypeError, np.multiply.outer, a)
2793: (8)             assert_raises(TypeError, np.multiply.outer, a, a, a, a)
2794: (8)             assert_raises(TypeError, np.multiply.outer, a, a, sig='a',
signature='a')

2795: (8)             res = np.multiply.at(a, [4, 2], 'b0')
2796: (8)             assert_equal(res[0], a)
2797: (8)             assert_equal(res[1], np.multiply)
2798: (8)             assert_equal(res[2], 'at')
2799: (8)             assert_equal(res[3], (a, [4, 2], 'b0'))
2800: (8)             assert_raises(TypeError, np.multiply.at, a)
2801: (8)             assert_raises(TypeError, np.multiply.at, a, a, a, a)
2802: (4)             def test_ufunc_override_out(self):
2803: (8)                 class A:
2804: (12)                     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
2805: (16)                         return kwargs
2806: (8)                 class B:
2807: (12)                     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
2808: (16)                         return kwargs
2809: (8)                 a = A()
2810: (8)                 b = B()
2811: (8)                 res0 = np.multiply(a, b, 'out_arg')
2812: (8)                 res1 = np.multiply(a, b, out='out_arg')
2813: (8)                 res2 = np.multiply(2, b, 'out_arg')
2814: (8)                 res3 = np.multiply(3, b, out='out_arg')
2815: (8)                 res4 = np.multiply(a, 4, 'out_arg')
2816: (8)                 res5 = np.multiply(a, 5, out='out_arg')
2817: (8)                 assert_equal(res0['out'][0], 'out_arg')
2818: (8)                 assert_equal(res1['out'][0], 'out_arg')

```

```

2819: (8)             assert_equal(res2['out'][0], 'out_arg')
2820: (8)             assert_equal(res3['out'][0], 'out_arg')
2821: (8)             assert_equal(res4['out'][0], 'out_arg')
2822: (8)             assert_equal(res5['out'][0], 'out_arg')
2823: (8)             res6 = np.modf(a, 'out0', 'out1')
2824: (8)             res7 = np.frexp(a, 'out0', 'out1')
2825: (8)             assert_equal(res6['out'][0], 'out0')
2826: (8)             assert_equal(res6['out'][1], 'out1')
2827: (8)             assert_equal(res7['out'][0], 'out0')
2828: (8)             assert_equal(res7['out'][1], 'out1')
2829: (8)             assert_(np.sin(a, None) == {})
2830: (8)             assert_(np.sin(a, out=None) == {})
2831: (8)             assert_(np.sin(a, out=(None,)) == {})
2832: (8)             assert_(np.modf(a, None) == {})
2833: (8)             assert_(np.modf(a, None, None) == {})
2834: (8)             assert_(np.modf(a, out=(None, None)) == {})
2835: (8)             with assert_raises(TypeError):
2836: (12)                 np.modf(a, out=None)
2837: (8)             assert_raises(TypeError, np.multiply, a, b, 'one', out='two')
2838: (8)             assert_raises(TypeError, np.multiply, a, b, 'one', 'two')
2839: (8)             assert_raises(ValueError, np.multiply, a, b, out=('one', 'two'))
2840: (8)             assert_raises(TypeError, np.multiply, a, out=())
2841: (8)             assert_raises(TypeError, np.modf, a, 'one', out=('two', 'three'))
2842: (8)             assert_raises(TypeError, np.modf, a, 'one', 'two', 'three')
2843: (8)             assert_raises(ValueError, np.modf, a, out=('one', 'two', 'three'))
2844: (8)             assert_raises(ValueError, np.modf, a, out=('one',))
2845: (4)             def test_ufunc_override_where(self):
2846: (8)                 class OverriddenArrayOld(np.ndarray):
2847: (12)                     def __init__(self, obj):
2848: (16)                         self._obj = obj
2849: (16)                     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
2850: (16)                         if "out" in kwargs:
2851: (20)                             kwargs["out"] = self._unwrap(kwargs["out"])
2852: (24)                             if kwargs["out"] is NotImplemented:
2853: (20)                                 return NotImplemented
2854: (24)                             result.append(kwargs["out"])
2855: (20)                         return result
2856: (16)                     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
2857: (12)                         inputs = self._unwrap(inputs)
2858: (16)                         if inputs is NotImplemented:
2859: (16)                             return NotImplemented
2860: (20)                         r = super().__array_ufunc__(ufunc, method, *inputs, **kwargs)
2861: (16)                         if r is not NotImplemented:
2862: (16)                             r = r.view(type(self))
2863: (20)                         return r
2864: (20)                     else:
2865: (24)                         if "where" in kwargs:
2866: (16)                             kwargs["where"] = self._unwrap(kwargs["where"])
2867: (16)                             if kwargs["where"] is NotImplemented:
2868: (20)                                 return NotImplemented
2869: (16)                             r = r.view(type(self))
2870: (8)                         return r
2871: (12)                     class OverriddenArrayNew(OverriddenArrayOld):
2872: (16)                         def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
2873: (16)                             kwargs = kwargs.copy()
2874: (16)                             if "where" in kwargs:
2875: (20)                                 kwargs["where"] = self._unwrap(kwargs["where"])
2876: (24)                                 if kwargs["where"] is NotImplemented:
2877: (20)                                     return NotImplemented
2878: (24)                                 else:
2879: (16)                                     kwargs["where"] = kwargs["where"][0]
2880: (16)                             r = super().__array_ufunc__(ufunc, method, *inputs, **kwargs)
2881: (20)                             if r is not NotImplemented:
2882: (16)                                 r = r.view(type(self))
2883: (8)                             return r
2884: (8)             ufunc = np.negative
2885: (8)             array = np.array([1, 2, 3])
2886: (8)             where = np.array([True, False, True])
2887: (8)             expected = ufunc(array, where=where)

```

```

2888: (12)           ufunc(array, where=where.view(OverriddenArrayOld))
2889: (8)            result_1 = ufunc(
2890: (12)              array,
2891: (12)              where=where.view(OverriddenArrayNew)
2892: (8)            )
2893: (8)            assert isinstance(result_1, OverriddenArrayNew)
2894: (8)            assert np.all(np.array(result_1) == expected, where=where)
2895: (8)            result_2 = ufunc(
2896: (12)              array.view(OverriddenArrayNew),
2897: (12)              where=where.view(OverriddenArrayNew)
2898: (8)            )
2899: (8)            assert isinstance(result_2, OverriddenArrayNew)
2900: (8)            assert np.all(np.array(result_2) == expected, where=where)
2901: (4)          def test_ufunc_override_exception(self):
2902: (8)            class A:
2903: (12)              def __array_ufunc__(self, *a, **kwargs):
2904: (16)                  raise ValueError("oops")
2905: (8)            a = A()
2906: (8)            assert_raises(ValueError, np.negative, 1, out=a)
2907: (8)            assert_raises(ValueError, np.negative, a)
2908: (8)            assert_raises(ValueError, np.divide, 1., a)
2909: (4)          def test_ufunc_override_not_implemented(self):
2910: (8)            class A:
2911: (12)              def __array_ufunc__(self, *args, **kwargs):
2912: (16)                  return NotImplemented
2913: (8)            msg = ("operand type(s) all returned NotImplemented from "
2914: (15)                "__array_ufunc__(<ufunc 'negative">, '__call__', <*>): 'A''")
2915: (8)            with assert_raises_regex(TypeError, fnmatch.translate(msg)):
2916: (12)                np.negative(A())
2917: (8)            msg = ("operand type(s) all returned NotImplemented from "
2918: (15)                "__array_ufunc__(<ufunc 'add'">, '__call__', <*>, <object *>, "
2919: (15)                  "out=(1,)): 'A', 'object', 'int'"')
2920: (8)            with assert_raises_regex(TypeError, fnmatch.translate(msg)):
2921: (12)                np.add(A(), object(), out=1)
2922: (4)          def test_ufunc_override_disabled(self):
2923: (8)            class OptOut:
2924: (12)              __array_ufunc__ = None
2925: (8)            opt_out = OptOut()
2926: (8)            msg = "operand 'OptOut' does not support ufuncs"
2927: (8)            with assert_raises_regex(TypeError, msg):
2928: (12)                np.add(opt_out, 1)
2929: (8)            with assert_raises_regex(TypeError, msg):
2930: (12)                np.add(1, opt_out)
2931: (8)            with assert_raises_regex(TypeError, msg):
2932: (12)                np.negative(opt_out)
2933: (8)            class GreedyArray:
2934: (12)              def __array_ufunc__(self, *args, **kwargs):
2935: (16)                  return self
2936: (8)            greedy = GreedyArray()
2937: (8)            assert_(np.negative(greedy) is greedy)
2938: (8)            with assert_raises_regex(TypeError, msg):
2939: (12)                np.add(greedy, opt_out)
2940: (8)            with assert_raises_regex(TypeError, msg):
2941: (12)                np.add(greedy, 1, out=opt_out)
2942: (4)          def test_gufunc_override(self):
2943: (8)            class A:
2944: (12)              def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
2945: (16)                  return self, ufunc, method, inputs, kwargs
2946: (8)            inner1d = ncu_tests.inner1d
2947: (8)            a = A()
2948: (8)            res = inner1d(a, a)
2949: (8)            assert_equal(res[0], a)
2950: (8)            assert_equal(res[1], inner1d)
2951: (8)            assert_equal(res[2], '__call__')
2952: (8)            assert_equal(res[3], (a, a))
2953: (8)            assert_equal(res[4], {})
2954: (8)            res = inner1d(1, 1, out=a)
2955: (8)            assert_equal(res[0], a)
2956: (8)            assert_equal(res[1], inner1d)

```

```

2957: (8)             assert_equal(res[2], '__call__')
2958: (8)             assert_equal(res[3], (1, 1))
2959: (8)             assert_equal(res[4], {'out': (a,)})
2960: (8)             assert_raises(TypeError, inner1d, a, out='two')
2961: (8)             assert_raises(TypeError, inner1d, a, a, 'one', out='two')
2962: (8)             assert_raises(TypeError, inner1d, a, a, 'one', 'two')
2963: (8)             assert_raises(ValueError, inner1d, a, a, out=('one', 'two'))
2964: (8)             assert_raises(ValueError, inner1d, a, a, out=())
2965: (4)              def test_ufunc_override_with_super(self):
2966: (8)                  class A(np.ndarray):
2967: (12)                      def __array_ufunc__(self, ufunc, method, *inputs, out=None,
2968: (16)                          **kwargs):
2969: (16)                          args = []
2970: (16)                          in_no = []
2971: (20)                          for i, input_ in enumerate(inputs):
2972: (24)                              if isinstance(input_, A):
2973: (24)                                  in_no.append(i)
2974: (20)                                  args.append(input_.view(np.ndarray))
2975: (24)                          else:
2976: (16)                              args.append(input_)
2977: (16)                          outputs = out
2978: (16)                          out_no = []
2979: (20)                          if outputs:
2980: (20)                              out_args = []
2981: (24)                              for j, output in enumerate(outputs):
2982: (28)                                  if isinstance(output, A):
2983: (28)                                      out_no.append(j)
2984: (24)                                      out_args.append(output.view(np.ndarray))
2985: (28)                                  else:
2986: (20)                                      out_args.append(output)
2987: (16)                          kwargs['out'] = tuple(out_args)
2988: (20)                      else:
2989: (16)                          outputs = (None,) * ufunc.nout
2990: (16)                          info = {}
2991: (20)                          if in_no:
2992: (16)                              info['inputs'] = in_no
2993: (20)                          if out_no:
2994: (16)                              info['outputs'] = out_no
2995: (50)                          results = super().__array_ufunc__(ufunc, method,
2996: (16)   *args, **kwargs)
2997: (20)                          if results is NotImplemented:
2998: (16)                              return NotImplemented
2999: (20)                          if method == 'at':
3000: (24)                              if isinstance(inputs[0], A):
3001: (20)                                  inputs[0].info = info
3002: (16)                                  return
3003: (20)                              results = (results,)
3004: (16)                          results = tuple((np.asarray(result).view(A)
3005: (33)  if output is None else output)
3006: (32)  for result, output in zip(results, outputs)))
3007: (16)                          if results and isinstance(results[0], A):
3008: (20)                              results[0].info = info
3009: (16)                          return results[0] if len(results) == 1 else results
3010: (8)              class B:
3011: (12)                  def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
3012: (16)                      if any(isinstance(input_, A) for input_ in inputs):
3013: (20)                          return "A!"
3014: (16)                      else:
3015: (20)                          return NotImplemented
3016: (8)                      d = np.arange(5.)
3017: (8)                      a = np.arange(5.).view(A)
3018: (8)                      b = np.sin(a)
3019: (8)                      check = np.sin(d)
3020: (8)                      assert_(np.all(check == b))
3021: (8)                      assert_equal(b.info, {'inputs': [0]})
3022: (8)                      b = np.sin(d, out=(a,))
3023: (8)                      assert_(np.all(check == b))
3024: (8)                      assert_equal(b.info, {'outputs': [0]})
```

```
3025: (8) assert_(b is a)
3026: (8) a = np.arange(5.).view(A)
3027: (8) b = np.sin(a, out=a)
3028: (8) assert_(np.all(check == b))
3029: (8) assert_equal(b.info, {'inputs': [0], 'outputs': [0]}) a = np.arange(5.).view(A)
3030: (8) b1, b2 = np.modf(a)
3031: (8) assert_equal(b1.info, {'inputs': [0]}) b1, b2 = np.modf(d, out=(None, a))
3032: (8) assert_(b2 is a)
3033: (8) assert_equal(b1.info, {'outputs': [1]}) a = np.arange(5.).view(A)
3034: (8) b = np.arange(5.).view(A)
3035: (8) c1, c2 = np.modf(a, out=(a, b))
3036: (8) assert_(c1 is a)
3037: (8) assert_(c2 is b)
3038: (8) assert_equal(c1.info, {'inputs': [0], 'outputs': [0, 1]}) a = np.arange(5.).view(A)
3039: (8) b = np.arange(5.).view(A)
3040: (8) c = np.add(a, b, out=a)
3041: (8) assert_(c is a)
3042: (8) assert_equal(c.info, {'inputs': [0, 1], 'outputs': [0]}) a = np.arange(5.)
3043: (8) b = B()
3044: (8) assert_(a.__array_ufunc__(np.add, '__call__', a, b) is NotImplemented)
3045: (8) assert_(b.__array_ufunc__(np.add, '__call__', a, b) is NotImplemented)
3046: (8) assert_raises(TypeError, np.add, a, b)
3047: (8) a = a.view(A)
3048: (8) assert_(a.__array_ufunc__(np.add, '__call__', a, b) is NotImplemented)
3049: (8) assert_(b.__array_ufunc__(np.add, '__call__', a, b) is NotImplemented)
3050: (8) assert_(np.add(a, b) == "A!")
3051: (8) d = np.array([[1, 2, 3], [1, 2, 3]])
3052: (8) a = d.view(A)
3053: (8) a = d.view(A)
3054: (8) assert_(a.__array_ufunc__(np.add, '__call__', a, b) is NotImplemented)
3055: (8) assert_(b.__array_ufunc__(np.add, '__call__', a, b) == "A!")
3056: (8) assert_(np.add(a, b) == "A!")
3057: (8) a = d.view(A)
3058: (8) c = a.any()
3059: (8) check = d.any()
3060: (8) assert_equal(c, check)
3061: (8) assert_(c.info, {'inputs': [0]}) c = a.max()
3062: (8) check = d.max()
3063: (8) assert_equal(c, check)
3064: (8) assert_(c.info, {'inputs': [0]}) c = a.max(out=b)
3065: (8) assert_(c.info, {'inputs': [0]}) b = np.array(0).view(A)
3066: (8) c = a.max(out=b)
3067: (8) assert_equal(c, check)
3068: (8) assert_(c.info, {'inputs': [0]}) c = a.max(axis=0)
3069: (8) assert_(c.info, {'inputs': [0]}) check = a.max(axis=0)
3070: (8) b = np.zeros_like(check).view(A)
3071: (8) c = a.max(axis=0, out=b)
3072: (8) assert_equal(c, check)
3073: (8) assert_(c.info, {'inputs': [0]}) c = a.max(axis=0)
3074: (8) assert_(c.info, {'inputs': [0]}) assert_(c is b)
3075: (8) assert_(c.info, {'inputs': [0]}) assert_(c is b)
3076: (8) check = np.add.reduce(d, axis=1)
3077: (8) c = np.add.reduce(a, axis=1)
3078: (8) assert_equal(c, check)
3079: (8) assert_(c.info, {'inputs': [0]}) assert_(c.info, {'inputs': [0]}) b = np.zeros_like(c)
3080: (8) c = np.add.reduce(a, 1, None, b)
3081: (8) assert_equal(c, check)
3082: (8) assert_(c.info, {'inputs': [0]}) assert_(c.info, {'inputs': [0]}) c = np.add.reduce(a, 0, None, b)
3083: (8) assert_equal(c, check)
3084: (8) assert_(c.info, {'inputs': [0]}) assert_(c.info, {'inputs': [0]}) check = np.add.accumulate(d, axis=0)
3085: (8) c = np.add.accumulate(a, axis=0)
3086: (8) assert_equal(c, check)
3087: (8) assert_(c.info, {'inputs': [0]}) assert_(c.info, {'inputs': [0]}) b = np.zeros_like(c)
3088: (8) c = np.add.accumulate(a, 0, None, b)
3089: (8) assert_equal(c, check)
3090: (8) assert_(c.info, {'inputs': [0]}) assert_(c.info, {'inputs': [0]}) c = np.add.accumulate(a, 0, None, b)
3091: (8) assert_equal(c, check)
3092: (8) assert_(c.info, {'inputs': [0]}) assert_(c.info, {'inputs': [0]}) assert_(c is b)
```

```

3094: (8) assert_(c.info, {'inputs': [0], 'outputs': [0]})  

3095: (8) indices = [0, 2, 1]  

3096: (8) check = np.add.reduceat(d, indices, axis=1)  

3097: (8) c = np.add.reduceat(a, indices, axis=1)  

3098: (8) assert_equal(c, check)  

3099: (8) assert_(c.info, {'inputs': [0]})  

3100: (8) b = np.zeros_like(c)  

3101: (8) c = np.add.reduceat(a, indices, 1, None, b)  

3102: (8) assert_equal(c, check)  

3103: (8) assert_(c is b)  

3104: (8) assert_(c.info, {'inputs': [0], 'outputs': [0]})  

3105: (8) d = np.array([[1, 2, 3], [1, 2, 3]])  

3106: (8) check = d.copy()  

3107: (8) a = d.copy().view(A)  

3108: (8) np.add.at(check, ([0, 1], [0, 2]), 1.)  

3109: (8) np.add.at(a, ([0, 1], [0, 2]), 1.)  

3110: (8) assert_equal(a, check)  

3111: (8) assert_(a.info, {'inputs': [0]})  

3112: (8) b = np.array(1.).view(A)  

3113: (8) a = d.copy().view(A)  

3114: (8) np.add.at(a, ([0, 1], [0, 2]), b)  

3115: (8) assert_equal(a, check)  

3116: (8) assert_(a.info, {'inputs': [0, 2]})  

3117: (4) def test_array_ufunc_direct_call(self):  

3118: (8) a = np.array(1)  

3119: (8) with pytest.raises(TypeError):  

3120: (12)     a.__array_ufunc__()  

3121: (8) with pytest.raises(TypeError):  

3122: (12)     a.__array_ufunc__(1, 2)  

3123: (8) res = a.__array_ufunc__(np.add, "__call__", a, a)  

3124: (8) assert_array_equal(res, a + a)  

3125: (0) class TestChoose:  

3126: (4)     def test_mixed(self):  

3127: (8)         c = np.array([True, True])  

3128: (8)         a = np.array([True, True])  

3129: (8)         assert_equal(np.choose(c, (a, 1)), np.array([1, 1]))  

3130: (0) class TestRationalFunctions:  

3131: (4)     def test_lcm(self):  

3132: (8)         self._test_lcm_inner(np.int16)  

3133: (8)         self._test_lcm_inner(np.uint16)  

3134: (4)     def test_lcm_object(self):  

3135: (8)         self._test_lcm_inner(np.object_)  

3136: (4)     def test_gcd(self):  

3137: (8)         self._test_gcd_inner(np.int16)  

3138: (8)         self._test_lcm_inner(np.uint16)  

3139: (4)     def test_gcd_object(self):  

3140: (8)         self._test_gcd_inner(np.object_)  

3141: (4)     def _test_lcm_inner(self, dtype):  

3142: (8)         a = np.array([12, 120], dtype=dtype)  

3143: (8)         b = np.array([20, 200], dtype=dtype)  

3144: (8)         assert_equal(np.lcm(a, b), [60, 600])  

3145: (8)         if not issubclass(dtype, np.unsignedinteger):  

3146: (12)             a = np.array([12, -12, 12, -12], dtype=dtype)  

3147: (12)             b = np.array([20, 20, -20, -20], dtype=dtype)  

3148: (12)             assert_equal(np.lcm(a, b), [60]*4)  

3149: (8)         a = np.array([3, 12, 20], dtype=dtype)  

3150: (8)         assert_equal(np.lcm.reduce([3, 12, 20]), 60)  

3151: (8)         a = np.arange(6).astype(dtype)  

3152: (8)         b = 20  

3153: (8)         assert_equal(np.lcm(a, b), [0, 20, 20, 60, 20, 20])  

3154: (4)     def _test_gcd_inner(self, dtype):  

3155: (8)         a = np.array([12, 120], dtype=dtype)  

3156: (8)         b = np.array([20, 200], dtype=dtype)  

3157: (8)         assert_equal(np.gcd(a, b), [4, 40])  

3158: (8)         if not issubclass(dtype, np.unsignedinteger):  

3159: (12)             a = np.array([12, -12, 12, -12], dtype=dtype)  

3160: (12)             b = np.array([20, 20, -20, -20], dtype=dtype)  

3161: (12)             assert_equal(np.gcd(a, b), [4]*4)  

3162: (8)             a = np.array([15, 25, 35], dtype=dtype)

```

```

3163: (8)             assert_equal(np.gcd.reduce(a), 5)
3164: (8)             a = np.arange(6).astype(dtype)
3165: (8)             b = 20
3166: (8)             assert_equal(np.gcd(a, b), [20, 1, 2, 1, 4, 5])
3167: (4)             def test_lcm_overflow(self):
3168: (8)                 big = np.int32(np.iinfo(np.int32).max // 11)
3169: (8)                 a = 2*big
3170: (8)                 b = 5*big
3171: (8)                 assert_equal(np.lcm(a, b), 10*big)
3172: (4)             def test_gcd_overflow(self):
3173: (8)                 for dtype in (np.int32, np.int64):
3174: (12)                     a = dtype(np.iinfo(dtype).min) # negative power of two
3175: (12)                     q = -(a // 4)
3176: (12)                     assert_equal(np.gcd(a, q*3), q)
3177: (12)                     assert_equal(np.gcd(a, -q*3), q)
3178: (4)             def test_decimal(self):
3179: (8)                 from decimal import Decimal
3180: (8)                 a = np.array([1, 1, -1, -1]) * Decimal('0.20')
3181: (8)                 b = np.array([1, -1, 1, -1]) * Decimal('0.12')
3182: (8)                 assert_equal(np.gcd(a, b), 4*[Decimal('0.04')])
3183: (8)                 assert_equal(np.lcm(a, b), 4*[Decimal('0.60')])
3184: (4)             def test_float(self):
3185: (8)                 assert_raises(TypeError, np.gcd, 0.3, 0.4)
3186: (8)                 assert_raises(TypeError, np.lcm, 0.3, 0.4)
3187: (4)             def test_builtin_long(self):
3188: (8)                 assert_equal(np.array(2**200).item(), 2**200)
3189: (8)                 a = np.array(2**100 * 3**5)
3190: (8)                 b = np.array([2**100 * 5**7, 2**50 * 3**10])
3191: (8)                 assert_equal(np.gcd(a, b), [2**100, 2**50 * 3**5])
3192: (8)                 assert_equal(np.lcm(a, b), [2**100 * 3**5 * 5**7, 2**100 * 3**10])
3193: (8)                 assert_equal(np.gcd(2**100, 3**100), 1)
3194: (0)             class TestRoundingFunctions:
3195: (4)             def test_object_direct(self):
3196: (8)                 """ test direct implementation of these magic methods """
3197: (8)                 class C:
3198: (12)                     def __floor__(self):
3199: (16)                         return 1
3200: (12)                     def __ceil__(self):
3201: (16)                         return 2
3202: (12)                     def __trunc__(self):
3203: (16)                         return 3
3204: (8)                     arr = np.array([C(), C()])
3205: (8)                     assert_equal(np.floor(arr), [1, 1])
3206: (8)                     assert_equal(np.ceil(arr), [2, 2])
3207: (8)                     assert_equal(np.trunc(arr), [3, 3])
3208: (4)             def test_object_indirect(self):
3209: (8)                 """ test implementations via __float__ """
3210: (8)                 class C:
3211: (12)                     def __float__(self):
3212: (16)                         return -2.5
3213: (8)                     arr = np.array([C(), C()])
3214: (8)                     assert_equal(np.floor(arr), [-3, -3])
3215: (8)                     assert_equal(np.ceil(arr), [-2, -2])
3216: (8)                     with pytest.raises(TypeError):
3217: (12)                         np.trunc(arr) # consistent with math.trunc
3218: (4)             def test_fraction(self):
3219: (8)                 f = Fraction(-4, 3)
3220: (8)                 assert_equal(np.floor(f), -2)
3221: (8)                 assert_equal(np.ceil(f), -1)
3222: (8)                 assert_equal(np.trunc(f), -1)
3223: (0)             class TestComplexFunctions:
3224: (4)                 funcs = [np.arcsin, np.arccos, np.arctan, np.arcsinh, np.arccosh,
3225: (13)                     np.arctanh, np.sin, np.cos, np.tan, np.exp,
3226: (13)                     np.exp2, np.log, np.sqrt, np.log10, np.log2,
3227: (13)                     np.log1p]
3228: (4)             def test_it(self):
3229: (8)                 for f in self.funcs:
3230: (12)                     if f is np.arccosh:
3231: (16)                         x = 1.5

```

```

3232: (12)           else:
3233: (16)             x = .5
3234: (12)             fr = f(x)
3235: (12)             fz = f(complex(x))
3236: (12)             assert_almost_equal(fz.real, fr, err_msg='real part %s' % f)
3237: (12)             assert_almost_equal(fz.imag, 0., err_msg='imag part %s' % f)
3238: (4) @pytest.mark.xfail(IS_MUSL, reason="gh23049")
3239: (4) @pytest.mark.xfail(IS_WASM, reason="doesn't work")
3240: (4) def test_precisions_consistent(self):
3241: (8)   z = 1 + 1j
3242: (8)   for f in self.funcs:
3243: (12)     fcf = f(np.csingle(z))
3244: (12)     fcd = f(np.cdouble(z))
3245: (12)     fcl = f(np.clongdouble(z))
3246: (12)     assert_almost_equal(fcf, fcd, decimal=6, err_msg='fch-fcd %s' % f)
3247: (12)     assert_almost_equal(fcl, fcd, decimal=15, err_msg='fch-fcl %s' % f)
3248: (4) @pytest.mark.xfail(IS_MUSL, reason="gh23049")
3249: (4) @pytest.mark.xfail(IS_WASM, reason="doesn't work")
3250: (4) def test_branch_cuts(self):
3251: (8)   _check_branch_cut(np.log, -0.5, 1j, 1, -1, True)
3252: (8)   _check_branch_cut(np.log2, -0.5, 1j, 1, -1, True)
3253: (8)   _check_branch_cut(np.log10, -0.5, 1j, 1, -1, True)
3254: (8)   _check_branch_cut(np.log1p, -1.5, 1j, 1, -1, True)
3255: (8)   _check_branch_cut(np.sqrt, -0.5, 1j, 1, -1, True)
3256: (8)   _check_branch_cut(np.arcsin, [-2, 2], [1j, 1j], 1, -1, True)
3257: (8)   _check_branch_cut(np.arccos, [-2, 2], [1j, 1j], 1, -1, True)
3258: (8)   _check_branch_cut(np.arctan, [0-2j, 2j], [1, 1], -1, 1, True)
3259: (8)   _check_branch_cut(np.arcsinh, [0-2j, 2j], [1, 1], -1, 1, True)
3260: (8)   _check_branch_cut(np.arccosh, [-1, 0.5], [1j, 1j], 1, -1, True)
3261: (8)   _check_branch_cut(np.arctanh, [-2, 2], [1j, 1j], 1, -1, True)
3262: (8)   _check_branch_cut(np.arcsin, [0-2j, 2j], [1, 1], 1, 1)
3263: (8)   _check_branch_cut(np.arccos, [0-2j, 2j], [1, 1], 1, 1)
3264: (8)   _check_branch_cut(np.arctan, [-2, 2], [1j, 1j], 1, 1)
3265: (8)   _check_branch_cut(np.arcsinh, [-2, 2, 0], [1j, 1j, 1], 1, 1)
3266: (8)   _check_branch_cut(np.arccosh, [0-2j, 2j, 2], [1, 1, 1j], 1, 1)
3267: (8)   _check_branch_cut(np.arctanh, [0-2j, 2j, 0], [1, 1, 1j], 1, 1)
3268: (4) @pytest.mark.xfail(IS_MUSL, reason="gh23049")
3269: (4) @pytest.mark.xfail(IS_WASM, reason="doesn't work")
3270: (4) def test_branch_cuts_complex64(self):
3271: (8)   _check_branch_cut(np.log, -0.5, 1j, 1, -1, True, np.complex64)
3272: (8)   _check_branch_cut(np.log2, -0.5, 1j, 1, -1, True, np.complex64)
3273: (8)   _check_branch_cut(np.log10, -0.5, 1j, 1, -1, True, np.complex64)
3274: (8)   _check_branch_cut(np.log1p, -1.5, 1j, 1, -1, True, np.complex64)
3275: (8)   _check_branch_cut(np.sqrt, -0.5, 1j, 1, -1, True, np.complex64)
3276: (8)   _check_branch_cut(np.arcsin, [-2, 2], [1j, 1j], 1, -1, True,
3277: (8)     np.complex64)
3278: (8)   _check_branch_cut(np.arccos, [-2, 2], [1j, 1j], 1, -1, True,
3279: (8)     np.complex64)
3280: (8)   _check_branch_cut(np.arctan, [0-2j, 2j], [1, 1], -1, 1, True,
3281: (8)     np.complex64)
3282: (8)   _check_branch_cut(np.arcsinh, [0-2j, 2j], [1, 1], -1, 1, True,
3283: (8)     np.complex64)
3284: (8)   _check_branch_cut(np.arccosh, [-1, 0.5], [1j, 1j], 1, -1, True,
3285: (8)     np.complex64)
3286: (8)   _check_branch_cut(np.arctanh, [-2, 2], [1j, 1j], 1, -1, True,
3287: (8)     np.complex64)
3288: (8)   _check_branch_cut(np.arcsin, [0-2j, 2j], [1, 1], 1, 1, False,
3289: (8)     np.complex64)
3290: (8)   _check_branch_cut(np.arccos, [0-2j, 2j], [1, 1], 1, 1, False,
3291: (8)     np.complex64)
3292: (8)   _check_branch_cut(np.arctan, [-2, 2], [1j, 1j], 1, 1, False,
3293: (8)     np.complex64)
3294: (8)   _check_branch_cut(np.arcsinh, [-2, 2, 0], [1j, 1j, 1], 1, 1, False,
3295: (8)     np.complex64)
3296: (8)   _check_branch_cut(np.arccosh, [0-2j, 2j, 2], [1, 1, 1j], 1, 1,
3297: (8)     np.complex64)
3298: (8)   _check_branch_cut(np.arctanh, [0-2j, 2j, 0], [1, 1, 1j], 1, 1,
3299: (8)     np.complex64)

```

```

3288: (4)
3289: (8)
3290: (8)
3291: (8)
3292: (20)
'atanh'}
3293: (8)
3294: (8)
3295: (12)
3296: (12)
3297: (12)
3298: (16)
3299: (12)
3300: (16)
3301: (12)
3302: (16)
3303: (16)
3304: (16)
3305: (20)
3306: (20)
3307: (16)
3308: (4)
3309: (8)
3310: (8)
3311: (4)
3312: (4)
3313: (4)
3314: (4)
np.longcomplex])
3315: (4)
3316: (8)
3317: (8)
3318: (8)
3319: (8)
3320: (8)
3321: (12)
3322: (12)
3323: (12)
3324: (12)
3325: (38)
3326: (12)
3327: (12)
3328: (12)
3329: (38)
3330: (12)
3331: (12)
3332: (12)
3333: (38)
3334: (12)
3335: (12)
3336: (12)
3337: (38)
3338: (8)
3339: (8)
3340: (8)
3341: (12)
3342: (16)
3343: (28)
3344: (28)
3345: (12)
3346: (8)
3347: (12)
3348: (8)
3349: (8)
3350: (8)
3351: (8)
3352: (8)
3353: (8)
3354: (8)

        def test_against_cmath(self):
            import cmath
            points = [-1-1j, -1+1j, +1-1j, +1+1j]
            name_map = {'arcsin': 'asin', 'arccos': 'acos', 'arctan': 'atan',
                        'arcsinh': 'asinh', 'arccosh': 'acosh', 'arctanh':
                        'atanh'}
            atol = 4*np.finfo(complex).eps
            for func in self.funcs:
                fname = func.__name__.split('.')[ -1]
                cname = name_map.get(fname, fname)
                try:
                    cfunc = getattr(cmath, cname)
                except AttributeError:
                    continue
                for p in points:
                    a = complex(func(np.complex_(p)))
                    b = cfunc(p)
                    assert_(
                        abs(a - b) < atol,
                        "%s %s: %s; cmath: %s" % (fname, p, a, b)
                    )
            @pytest.mark.xfail(
                _glibc_older_than("2.18"),
                reason="Older glibc versions are imprecise (maybe passes with SIMD?)"
            )
            @pytest.mark.xfail(IS_MUSL, reason="gh23049")
            @pytest.mark.xfail(IS_WASM, reason="doesn't work")
            @pytest.mark.parametrize('dtype', [np.complex64, np.complex_,
  np.longcomplex])
            def test_loss_of_precision(self, dtype):
                """Check loss of precision in complex arc* functions"""
                info = np.finfo(dtype)
                real_dtype = dtype(0.).real.dtype
                eps = info.eps
                def check(x, rtol):
                    x = x.astype(real_dtype)
                    z = x.astype(dtype)
                    d = np.absolute(np.arcsinh(x)/np.arcsinh(z).real - 1)
                    assert_(np.all(d < rtol), (np.argmax(d), x[np.argmax(d)], d.max(),
  'arcsinh'))
                    z = (1j*x).astype(dtype)
                    d = np.absolute(np.arcsinh(x)/np.arcsin(z).imag - 1)
                    assert_(np.all(d < rtol), (np.argmax(d), x[np.argmax(d)], d.max(),
  'arcsin'))
                    z = x.astype(dtype)
                    d = np.absolute(np.arctanh(x)/np.arctanh(z).real - 1)
                    assert_(np.all(d < rtol), (np.argmax(d), x[np.argmax(d)], d.max(),
  'arctanh'))
                    z = (1j*x).astype(dtype)
                    d = np.absolute(np.arctanh(x)/np.arctan(z).imag - 1)
                    assert_(np.all(d < rtol), (np.argmax(d), x[np.argmax(d)], d.max(),
  'arctan'))
                    x_series = np.logspace(-20, -3.001, 200)
                    x_basic = np.logspace(-2.999, 0, 10, endpoint=False)
                    if dtype is np.longcomplex:
                        if bad_arcsinh():
                            pytest.skip("Trig functions of np.longcomplex values known "
  "to be inaccurate on aarch64 and PPC for some "
  "compilation configurations.")
                        check(x_series, 50.0*eps)
                    else:
                        check(x_series, 2.1*eps)
                    check(x_basic, 2.0*eps/1e-3)
                    z = np.array([1e-5*(1+1j)], dtype=dtype)
                    p = 9.9999999993333333e-6 + 1.00000000066666666e-5j
                    d = np.absolute(1-np.arctanh(z)/p)
                    assert_(np.all(d < 1e-15))
                    p = 1.0000000003333333e-5 + 9.9999999996666667e-6j
                    d = np.absolute(1-np.arcsinh(z)/p)

```

```

3355: (8) assert_(np.all(d < 1e-15))
3356: (8) p = 9.9999999993333333e-6j + 1.0000000006666666e-5
3357: (8) d = np.absolute(1-np.arctan(z)/p)
3358: (8) assert_(np.all(d < 1e-15))
3359: (8) p = 1.00000000033333333e-5j + 9.99999999966666667e-6
3360: (8) d = np.absolute(1-np.arcsin(z)/p)
3361: (8) assert_(np.all(d < 1e-15))
3362: (8) def check(func, z0, d=1):
3363: (12)     z0 = np.asarray(z0, dtype=dtype)
3364: (12)     zp = z0 + abs(z0) * d * eps * 2
3365: (12)     zm = z0 - abs(z0) * d * eps * 2
3366: (12)     assert_(np.all(zp != zm), (zp, zm))
3367: (12)     good = (abs(func(zp) - func(zm)) < 2*eps)
3368: (12)     assert_(np.all(good), (func, z0[~good]))
3369: (8)     for func in (np.arcsinh, np.arcsinh, np.arcsin, np.arctanh,
np.arctan):
3370: (12)         pts = [rp+1j*ip for rp in (-1e-3, 0, 1e-3) for ip in(-1e-3, 0, 1e-
3)
3371: (19)             if rp != 0 or ip != 0]
3372: (12)         check(func, pts, 1)
3373: (12)         check(func, pts, 1j)
3374: (12)         check(func, pts, 1+1j)
3375: (4) @np.errstate(all="ignore")
3376: (4) def test_promotion_corner_cases(self):
3377: (8)     for func in self.funcs:
3378: (12)         assert func(np.float16(1)).dtype == np.float16
3379: (12)         assert func(np.uint8(1)).dtype == np.float16
3380: (12)         assert func(np.int16(1)).dtype == np.float32
3381: (0) class TestAttributes:
3382: (4)     def test_attributes(self):
3383: (8)         add = ncu.add
3384: (8)         assert_equal(add.__name__, 'add')
3385: (8)         assert_(add.ntypes >= 18) # don't fail if types added
3386: (8)         assert_('ii->i' in add.types)
3387: (8)         assert_equal(add.nin, 2)
3388: (8)         assert_equal(add.nout, 1)
3389: (8)         assert_equal(add.identity, 0)
3390: (4)     def test_doc(self):
3391: (8)         assert_(ncu.add.__doc__.startswith(
3392: (12)             "add(x1, x2, /, out=None, *, where=True"))
3393: (8)         assert_(ncu.frexp.__doc__.startswith(
3394: (12)             "frexp(x[, out1, out2], / [, out=(None, None)], *, where=True"))
3395: (0) class TestSubclass:
3396: (4)     def test_subclass_op(self):
3397: (8)         class simple(np.ndarray):
3398: (12)             def __new__(subtype, shape):
3399: (16)                 self = np.ndarray.__new__(subtype, shape, dtype=object)
3400: (16)                 self.fill(0)
3401: (16)                 return self
3402: (8)             a = simple((3, 4))
3403: (8)             assert_equal(a+a, a)
3404: (0)         class TestFrompyfunc:
3405: (4)             def test_identity(self):
3406: (8)                 def mul(a, b):
3407: (12)                     return a * b
3408: (8)                 mul_ufunc = np.frompyfunc(mul, nin=2, nout=1, identity=1)
3409: (8)                 assert_equal(mul_ufunc.reduce([2, 3, 4]), 24)
3410: (8)                 assert_equal(mul_ufunc.reduce(np.ones((2, 2))), axis=(0, 1)), 1)
3411: (8)                 assert_equal(mul_ufunc.reduce([]), 1)
3412: (8)                 mul_ufunc = np.frompyfunc(mul, nin=2, nout=1, identity=None)
3413: (8)                 assert_equal(mul_ufunc.reduce([2, 3, 4]), 24)
3414: (8)                 assert_equal(mul_ufunc.reduce(np.ones((2, 2))), axis=(0, 1)), 1)
3415: (8)                 assert_raises(ValueError, lambda: mul_ufunc.reduce([]))
3416: (8)                 mul_ufunc = np.frompyfunc(mul, nin=2, nout=1)
3417: (8)                 assert_equal(mul_ufunc.reduce([2, 3, 4]), 24)
3418: (8)                 assert_raises(ValueError, lambda: mul_ufunc.reduce(np.ones((2, 2)),
axis=(0, 1)))
3419: (8)             assert_raises(ValueError, lambda: mul_ufunc.reduce([]))
3420: (0)             def _check_branch_cut(f, x0, dx, re_sign=1, im_sign=-1, sig_zero_ok=False,

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

3421: (22)                               dtype=complex):
3422: (4)                                 """
3423: (4)                                 Check for a branch cut in a function.
3424: (4)                                 Assert that `x0` lies on a branch cut of function `f` and `f` is
3425: (4)                                 continuous from the direction `dx`.
3426: (4)                                 Parameters
3427: (4)                                 -----
3428: (4)                                 f : func
3429: (8)                                 Function to check
3430: (4)                                 x0 : array-like
3431: (8)                                 Point on branch cut
3432: (4)                                 dx : array-like
3433: (8)                                 Direction to check continuity in
3434: (4)                                 re_sign, im_sign : {1, -1}
3435: (8)                                 Change of sign of the real or imaginary part expected
3436: (4)                                 sig_zero_ok : bool
3437: (8)                                 Whether to check if the branch cut respects signed zero (if
applicable)
3438: (4)                                 dtype : dtype
3439: (8)                                 Dtype to check (should be complex)
3440: (4)                                 """
3441: (4)                                 x0 = np.atleast_1d(x0).astype(dtype)
3442: (4)                                 dx = np.atleast_1d(dx).astype(dtype)
3443: (4)                                 if np.dtype(dtype).char == 'F':
3444: (8)                                 scale = np.finfo(dtype).eps * 1e2
3445: (8)                                 atol = np.float32(1e-2)
3446: (4)                                 else:
3447: (8)                                 scale = np.finfo(dtype).eps * 1e3
3448: (8)                                 atol = 1e-4
3449: (4)                                 y0 = f(x0)
3450: (4)                                 yp = f(x0 + dx*scale*np.absolute(x0)/np.absolute(dx))
3451: (4)                                 ym = f(x0 - dx*scale*np.absolute(x0)/np.absolute(dx))
3452: (4)                                 assert_(np.all(np.absolute(y0.real - yp.real) < atol), (y0, yp))
3453: (4)                                 assert_(np.all(np.absolute(y0.imag - yp.imag) < atol), (y0, yp))
3454: (4)                                 assert_(np.all(np.absolute(y0.real - ym.real*re_sign) < atol), (y0, ym))
3455: (4)                                 assert_(np.all(np.absolute(y0.imag - ym.imag*im_sign) < atol), (y0, ym))
3456: (4)                                 if sig_zero_ok:
3457: (8)                                 jr = (x0.real == 0) & (dx.real != 0)
3458: (8)                                 ji = (x0.imag == 0) & (dx.imag != 0)
3459: (8)                                 if np.any(jr):
3460: (12)                                 x = x0[jr]
3461: (12)                                 x.real = np.NZERO
3462: (12)                                 ym = f(x)
3463: (12)                                 assert_(np.all(np.absolute(y0[jr].real - ym.real*re_sign) < atol),
(y0[jr], ym))
3464: (12)                                 assert_(np.all(np.absolute(y0[jr].imag - ym.imag*im_sign) < atol),
(y0[jr], ym))
3465: (8)                                 if np.any(ji):
3466: (12)                                 x = x0[ji]
3467: (12)                                 x.imag = np.NZERO
3468: (12)                                 ym = f(x)
3469: (12)                                 assert_(np.all(np.absolute(y0[ji].real - ym.real*re_sign) < atol),
(y0[ji], ym))
3470: (12)                                 assert_(np.all(np.absolute(y0[ji].imag - ym.imag*im_sign) < atol),
(y0[ji], ym))
3471: (0)                                 def test_copysign():
3472: (4)                                 assert_(np.copysign(1, -1) == -1)
3473: (4)                                 with np.errstate(divide="ignore"):
3474: (8)                                 assert_(1 / np.copysign(0, -1) < 0)
3475: (8)                                 assert_(1 / np.copysign(0, 1) > 0)
3476: (4)                                 assert_(np.signbit(np.copysign(np.nan, -1)))
3477: (4)                                 assert_(not np.signbit(np.copysign(np.nan, 1)))
3478: (0)                                 def _test_nextafter(t):
3479: (4)                                 one = t(1)
3480: (4)                                 two = t(2)
3481: (4)                                 zero = t(0)
3482: (4)                                 eps = np.finfo(t).eps
3483: (4)                                 assert_(np.nextafter(one, two) - one == eps)
3484: (4)                                 assert_(np.nextafter(one, zero) - one < 0)

```

```

3485: (4) assert_(np.isnan(np.nextafter(np.nan, one)))
3486: (4) assert_(np.isnan(np.nextafter(one, np.nan)))
3487: (4) assert_(np.nextafter(one, one) == one)
3488: (0) def test_nextafter():
3489: (4)     return _test_nextafter(np.float64)
3490: (0) def test_nextafterf():
3491: (4)     return _test_nextafter(np.float32)
3492: (0) @pytest.mark.skipif(np.finfo(np.double) == np.finfo(np.longdouble),
3493: (20)                     reason="long double is same as double")
3494: (0) @pytest.mark.xfail(condition=platform.machine().startswith("ppc64"),
3495: (20)                     reason="IBM double double")
3496: (0) def test_nextafterl():
3497: (4)     return _test_nextafter(np.longdouble)
3498: (0) def test_nextafter_0():
3499: (4)     for t, direction in itertools.product(np.sctypes['float'], (1, -1)):
3500: (8)         with suppress_warnings() as sup:
3501: (12)             sup.filter(UserWarning)
3502: (12)             if not np.isnan(np.finfo(t).tiny):
3503: (16)                 tiny = np.finfo(t).tiny
3504: (16)                 assert_
3505: (20)                 0. < direction * np.nextafter(t(0), t(direction)) < tiny
3506: (8)                 assert_equal(np.nextafter(t(0), t(direction)) / t(2.1), direction *
0.0)
3507: (0) def _test_spacing(t):
3508: (4)     one = t(1)
3509: (4)     eps = np.finfo(t).eps
3510: (4)     nan = t(np.nan)
3511: (4)     inf = t(np.inf)
3512: (4)     with np.errstate(invalid='ignore'):
3513: (8)         assert_equal(np.spacing(one), eps)
3514: (8)         assert_(np.isnan(np.spacing(nan)))
3515: (8)         assert_(np.isnan(np.spacing(inf)))
3516: (8)         assert_(np.isnan(np.spacing(-inf)))
3517: (8)         assert_(np.spacing(t(1e30)) != 0)
3518: (0) def test_spacing():
3519: (4)     return _test_spacing(np.float64)
3520: (0) def test_spacingf():
3521: (4)     return _test_spacing(np.float32)
3522: (0) @pytest.mark.skipif(np.finfo(np.double) == np.finfo(np.longdouble),
3523: (20)                     reason="long double is same as double")
3524: (0) @pytest.mark.xfail(condition=platform.machine().startswith("ppc64"),
3525: (20)                     reason="IBM double double")
3526: (0) def test_spacingle():
3527: (4)     return _test_spacing(np.longdouble)
3528: (0) def test_spacing_gfortran():
3529: (4)     ref = {np.float64: [1.69406589450860068E-021,
3530: (24)                     2.22044604925031308E-016,
3531: (24)                     1.13686837721616030E-013,
3532: (24)                     1.81898940354585648E-012],
3533: (11)                     np.float32: [9.09494702E-13,
3534: (24)                         1.19209290E-07,
3535: (24)                         6.10351563E-05,
3536: (24)                         9.76562500E-04]}
3537: (4)     for dt, dec_ in zip([np.float32, np.float64], (10, 20)):
3538: (8)         x = np.array([1e-5, 1, 1000, 10500], dtype=dt)
3539: (8)         assert_array_almost_equal(np.spacing(x), ref[dt], decimal=dec_)
3540: (0) def test_nextafter_vs_spacing():
3541: (4)     for t in [np.float32, np.float64]:
3542: (8)         for _f in [1, 1e-5, 1000]:
3543: (12)             f = t(_f)
3544: (12)             f1 = t(_f + 1)
3545: (12)             assert_(np.nextafter(f, f1) - f == np.spacing(f))
3546: (0) def test_pos_nan():
3547: (4)     """Check np.nan is a positive nan."""
3548: (4)     assert_(np.signbit(np.nan) == 0)
3549: (0) def test_reduceat():
3550: (4)     """Test bug in reduceat when structured arrays are not copied."""
3551: (4)     db = np.dtype([('name', 'S11'), ('time', np.int64), ('value',
np.float32)])

```

```

3552: (4)          a = np.empty([100], dtype=db)
3553: (4)          a['name'] = 'Simple'
3554: (4)          a['time'] = 10
3555: (4)          a['value'] = 100
3556: (4)          indx = [0, 7, 15, 25]
3557: (4)          h2 = []
3558: (4)          val1 = indx[0]
3559: (4)          for val2 in indx[1:]:
3560: (8)              h2.append(np.add.reduce(a['value'][val1:val2]))
3561: (8)              val1 = val2
3562: (4)          h2.append(np.add.reduce(a['value'][val1:]))
3563: (4)          h2 = np.array(h2)
3564: (4)          h1 = np.add.reduceat(a['value'], indx)
3565: (4)          assert_array_almost_equal(h1, h2)
3566: (4)          np.setbufsize(32)
3567: (4)          h1 = np.add.reduceat(a['value'], indx)
3568: (4)          np.setbufsize(np.UFUNC_BUFSIZE_DEFAULT)
3569: (4)          assert_array_almost_equal(h1, h2)
3570: (0)          def test_reduceat_empty():
3571: (4)              """Reduceat should work with empty arrays"""
3572: (4)              indices = np.array([], 'i4')
3573: (4)              x = np.array([], 'f8')
3574: (4)              result = np.add.reduceat(x, indices)
3575: (4)              assert_equal(result.dtype, x.dtype)
3576: (4)              assert_equal(result.shape, (0,))
3577: (4)              x = np.ones((5, 2))
3578: (4)              result = np.add.reduceat(x, [], axis=0)
3579: (4)              assert_equal(result.dtype, x.dtype)
3580: (4)              assert_equal(result.shape, (0, 2))
3581: (4)              result = np.add.reduceat(x, [], axis=1)
3582: (4)              assert_equal(result.dtype, x.dtype)
3583: (4)              assert_equal(result.shape, (5, 0))
3584: (0)          def test_complex_nan_comparisons():
3585: (4)              nans = [complex(np.nan, 0), complex(0, np.nan), complex(np.nan, np.nan)]
3586: (4)              fins = [complex(1, 0), complex(-1, 0), complex(0, 1), complex(0, -1),
3587: (12)                  complex(1, 1), complex(-1, -1), complex(0, 0)]
3588: (4)              with np.errstate(invalid='ignore'):
3589: (8)                  for x in nans + fins:
3590: (12)                      x = np.array([x])
3591: (12)                      for y in nans + fins:
3592: (16)                          y = np.array([y])
3593: (16)                          if np.isfinite(x) and np.isfinite(y):
3594: (20)                              continue
3595: (16)                          assert_equal(x < y, False, err_msg="%r < %r" % (x, y))
3596: (16)                          assert_equal(x > y, False, err_msg="%r > %r" % (x, y))
3597: (16)                          assert_equal(x <= y, False, err_msg="%r <= %r" % (x, y))
3598: (16)                          assert_equal(x >= y, False, err_msg="%r >= %r" % (x, y))
3599: (16)                          assert_equal(x == y, False, err_msg="%r == %r" % (x, y))
3600: (0)          def test_rint_big_int():
3601: (4)              val = 4607998452777363968
3602: (4)              assert_equal(val, int(float(val)))
3603: (4)              assert_equal(val, np.rint(val))
3604: (0)          @pytest.mark.parametrize('ftype', [np.float32, np.float64])
3605: (0)          def test_memoverlap_accumulate(ftype):
3606: (4)              arr = np.array([0.61, 0.60, 0.77, 0.41, 0.19], dtype=ftype)
3607: (4)              out_max = np.array([0.61, 0.61, 0.77, 0.77, 0.77], dtype=ftype)
3608: (4)              out_min = np.array([0.61, 0.60, 0.60, 0.41, 0.19], dtype=ftype)
3609: (4)              assert_equal(np.maximum.accumulate(arr), out_max)
3610: (4)              assert_equal(np.minimum.accumulate(arr), out_min)
3611: (0)          @pytest.mark.parametrize("ufunc, dtype", [
3612: (4)              (ufunc, t[0])
3613: (4)              for ufunc in UFUNCS_BINARY_ACC
3614: (4)              for t in ufunc.types
3615: (4)              if t[-1] == '?' and t[0] not in 'DFGMmO'
3616: (0)
3617: (0)
3618: (4)
3619: (8)
3620: (4)
3621: (0)
3622: (0)
3623: (0)
3624: (0)
3625: (0)
3626: (0)
3627: (0)
3628: (0)
3629: (0)
3630: (0)
3631: (0)
3632: (0)
3633: (0)
3634: (0)
3635: (0)
3636: (0)
3637: (0)
3638: (0)
3639: (0)
3640: (0)
3641: (0)
3642: (0)
3643: (0)
3644: (0)
3645: (0)
3646: (0)
3647: (0)
3648: (0)
3649: (0)
3650: (0)
3651: (0)
3652: (0)
3653: (0)
3654: (0)
3655: (0)
3656: (0)
3657: (0)
3658: (0)
3659: (0)
3660: (0)
3661: (0)
3662: (0)
3663: (0)
3664: (0)
3665: (0)
3666: (0)
3667: (0)
3668: (0)
3669: (0)
3670: (0)
3671: (0)
3672: (0)
3673: (0)
3674: (0)
3675: (0)
3676: (0)
3677: (0)
3678: (0)
3679: (0)
3680: (0)
3681: (0)
3682: (0)
3683: (0)
3684: (0)
3685: (0)
3686: (0)
3687: (0)
3688: (0)
3689: (0)
3690: (0)
3691: (0)
3692: (0)
3693: (0)
3694: (0)
3695: (0)
3696: (0)
3697: (0)
3698: (0)
3699: (0)
3700: (0)
3701: (0)
3702: (0)
3703: (0)
3704: (0)
3705: (0)
3706: (0)
3707: (0)
3708: (0)
3709: (0)
3710: (0)
3711: (0)
3712: (0)
3713: (0)
3714: (0)
3715: (0)
3716: (0)
3717: (0)
3718: (0)
3719: (0)
3720: (0)
3721: (0)
3722: (0)
3723: (0)
3724: (0)
3725: (0)
3726: (0)
3727: (0)
3728: (0)
3729: (0)
3730: (0)
3731: (0)
3732: (0)
3733: (0)
3734: (0)
3735: (0)
3736: (0)
3737: (0)
3738: (0)
3739: (0)
3740: (0)
3741: (0)
3742: (0)
3743: (0)
3744: (0)
3745: (0)
3746: (0)
3747: (0)
3748: (0)
3749: (0)
3750: (0)
3751: (0)
3752: (0)
3753: (0)
3754: (0)
3755: (0)
3756: (0)
3757: (0)
3758: (0)
3759: (0)
3760: (0)
3761: (0)
3762: (0)
3763: (0)
3764: (0)
3765: (0)
3766: (0)
3767: (0)
3768: (0)
3769: (0)
3770: (0)
3771: (0)
3772: (0)
3773: (0)
3774: (0)
3775: (0)
3776: (0)
3777: (0)
3778: (0)
3779: (0)
3780: (0)
3781: (0)
3782: (0)
3783: (0)
3784: (0)
3785: (0)
3786: (0)
3787: (0)
3788: (0)
3789: (0)
3790: (0)
3791: (0)
3792: (0)
3793: (0)
3794: (0)
3795: (0)
3796: (0)
3797: (0)
3798: (0)
3799: (0)
3800: (0)
3801: (0)
3802: (0)
3803: (0)
3804: (0)
3805: (0)
3806: (0)
3807: (0)
3808: (0)
3809: (0)
3810: (0)
3811: (0)
3812: (0)
3813: (0)
3814: (0)
3815: (0)
3816: (0)
3817: (0)
3818: (0)
3819: (0)
3820: (0)
3821: (0)
3822: (0)
3823: (0)
3824: (0)
3825: (0)
3826: (0)
3827: (0)
3828: (0)
3829: (0)
3830: (0)
3831: (0)
3832: (0)
3833: (0)
3834: (0)
3835: (0)
3836: (0)
3837: (0)
3838: (0)
3839: (0)
3840: (0)
3841: (0)
3842: (0)
3843: (0)
3844: (0)
3845: (0)
3846: (0)
3847: (0)
3848: (0)
3849: (0)
3850: (0)
3851: (0)
3852: (0)
3853: (0)
3854: (0)
3855: (0)
3856: (0)
3857: (0)
3858: (0)
3859: (0)
3860: (0)
3861: (0)
3862: (0)
3863: (0)
3864: (0)
3865: (0)
3866: (0)
3867: (0)
3868: (0)
3869: (0)
3870: (0)
3871: (0)
3872: (0)
3873: (0)
3874: (0)
3875: (0)
3876: (0)
3877: (0)
3878: (0)
3879: (0)
3880: (0)
3881: (0)
3882: (0)
3883: (0)
3884: (0)
3885: (0)
3886: (0)
3887: (0)
3888: (0)
3889: (0)
3890: (0)
3891: (0)
3892: (0)
3893: (0)
3894: (0)
3895: (0)
3896: (0)
3897: (0)
3898: (0)
3899: (0)
3900: (0)
3901: (0)
3902: (0)
3903: (0)
3904: (0)
3905: (0)
3906: (0)
3907: (0)
3908: (0)
3909: (0)
3910: (0)
3911: (0)
3912: (0)
3913: (0)
3914: (0)
3915: (0)
3916: (0)
3917: (0)
3918: (0)
3919: (0)
3920: (0)
3921: (0)
3922: (0)
3923: (0)
3924: (0)
3925: (0)
3926: (0)
3927: (0)
3928: (0)
3929: (0)
3930: (0)
3931: (0)
3932: (0)
3933: (0)
3934: (0)
3935: (0)
3936: (0)
3937: (0)
3938: (0)
3939: (0)
3940: (0)
3941: (0)
3942: (0)
3943: (0)
3944: (0)
3945: (0)
3946: (0)
3947: (0)
3948: (0)
3949: (0)
3950: (0)
3951: (0)
3952: (0)
3953: (0)
3954: (0)
3955: (0)
3956: (0)
3957: (0)
3958: (0)
3959: (0)
3960: (0)
3961: (0)
3962: (0)
3963: (0)
3964: (0)
3965: (0)
3966: (0)
3967: (0)
3968: (0)
3969: (0)
3970: (0)
3971: (0)
3972: (0)
3973: (0)
3974: (0)
3975: (0)
3976: (0)
3977: (0)
3978: (0)
3979: (0)
3980: (0)
3981: (0)
3982: (0)
3983: (0)
3984: (0)
3985: (0)
3986: (0)
3987: (0)
3988: (0)
3989: (0)
3990: (0)
3991: (0)
3992: (0)
3993: (0)
3994: (0)
3995: (0)
3996: (0)
3997: (0)
3998: (0)
3999: (0)
4000: (0)
4001: (0)
4002: (0)
4003: (0)
4004: (0)
4005: (0)
4006: (0)
4007: (0)
4008: (0)
4009: (0)
4010: (0)
4011: (0)
4012: (0)
4013: (0)
4014: (0)
4015: (0)
4016: (0)
4017: (0)
4018: (0)
4019: (0)
4020: (0)
4021: (0)
4022: (0)
4023: (0)
4024: (0)
4025: (0)
4026: (0)
4027: (0)
4028: (0)
4029: (0)
4030: (0)
4031: (0)
4032: (0)
4033: (0)
4034: (0)
4035: (0)
4036: (0)
4037: (0)
4038: (0)
4039: (0)
4040: (0)
4041: (0)
4042: (0)
4043: (0)
4044: (0)
4045: (0)
4046: (0)
4047: (0)
4048: (0)
4049: (0)
4050: (0)
4051: (0)
4052: (0)
4053: (0)
4054: (0)
4055: (0)
4056: (0)
4057: (0)
4058: (0)
4059: (0)
4060: (0)
4061: (0)
4062: (0)
4063: (0)
4064: (0)
4065: (0)
4066: (0)
4067: (0)
4068: (0)
4069: (0)
4070: (0)
4071: (0)
4072: (0)
4073: (0)
4074: (0)
4075: (0)
4076: (0)
4077: (0)
4078: (0)
4079: (0)
4080: (0)
4081: (0)
4082: (0)
4083: (0)
4084: (0)
4085: (0)
4086: (0)
4087: (0)
4088: (0)
4089: (0)
4090: (0)
4091: (0)
4092: (0)
4093: (0)
4094: (0)
4095: (0)
4096: (0)
4097: (0)
4098: (0)
4099: (0)
4100: (0)
4101: (0)
4102: (0)
4103: (0)
4104: (0)
4105: (0)
4106: (0)
4107: (0)
4108: (0)
4109: (0)
4110: (0)
4111: (0)
4112: (0)
4113: (0)
4114: (0)
4115: (0)
4116: (0)
4117: (0)
4118: (0)
4119: (0)
4120: (0)
4121: (0)
4122: (0)
4123: (0)
4124: (0)
4125: (0)
4126: (0)
4127: (0)
4128: (0)
4129: (0)
4130: (0)
4131: (0)
4132: (0)
4133: (0)
4134: (0)
4135: (0)
4136: (0)
4137: (0)
4138: (0)
4139: (0)
4140: (0)
4141: (0)
4142: (0)
4143: (0)
4144: (0)
4145: (0)
4146: (0)
4147: (0)
4148: (0)
4149: (0)
4150: (0)
4151: (0)
4152: (0)
4153: (0)
4154: (0)
4155: (0)
4156: (0)
4157: (0)
4158: (0)
4159: (0)
4160: (0)
4161: (0)
4162: (0)
4163: (0)
4164: (0)
4165: (0)
4166: (0)
4167: (0)
4168: (0)
4169: (0)
4170: (0)
4171: (0)
4172: (0)
4173: (0)
4174: (0)
4175: (0)
4176: (0)
4177: (0)
4178: (0)
4179: (0)
4180: (0)
4181: (0)
4182: (0)
4183: (0)
4184: (0)
4185: (0)
4186: (0)
4187: (0)
4188: (0)
4189: (0)
4190: (0)
4191: (0)
4192: (0)
4193: (0)
4194: (0)
4195: (0)
4196: (0)
4197: (0)
4198: (0)
4199: (0)
4200: (0)
4201: (0)
4202: (0)
4203: (0)
4204: (0)
4205: (0)
4206: (0)
4207: (0)
4208: (0)
4209: (0)
4210: (0)
4211: (0)
4212: (0)
4213: (0)
4214: (0)
4215: (0)
4216: (0)
4217: (0)
4218: (0)
4219: (0)
4220: (0)
4221: (0)
4222: (0)
4223: (0)
4224: (0)
4225: (0)
4226: (0)
4227: (0)
4228: (0)
4229: (0)
4230: (0)
4231: (0)
4232: (0)
4233: (0)
4234: (0)
4235: (0)
4236: (0)
4237: (0)
4238: (0)
4239: (0)
4240: (0)
4241: (0)
4242: (0)
4243: (0)
4244: (0)
4245: (0)
4246: (0)
4247: (0)
4248: (0)
4249: (0)
4250: (0)
4251: (0)
4252: (0)
4253: (0)
4254: (0)
4255: (0)
4256: (0)
4257: (0)
4258: (0)
4259: (0)
4260: (0)
4261: (0)
4262: (0)
4263: (0)
4264: (0)
4265: (0)
4266: (0)
4267: (0)
4268: (0)
4269: (0)
4270: (0)
4271: (0)
4272: (0)
4273: (0)
4274: (0)
4275: (0)
4276: (0)
4277: (0)
4278: (0)
4279: (0)
4280: (0)
4281: (0)
4282: (0)
4283: (0)
4284: (0)
4285: (0)
4286: (0)
4287: (0)
4288: (0)
4289: (0)
4290: (0)
4291: (0)
4292: (0)
4293: (0)
4294: (0)
4295: (0)
4296: (0)
4297: (0)
4298: (0)
4299: (0)
4300: (0)
4301: (0)
4302: (0)
4303: (0)
4304: (0)
4305: (0)
4306: (0)
4307: (0)
4308: (0)
4309: (0)
4310: (0)
4311: (0)
4312: (0)
4313: (0)
4314: (0)
4315: (0)
4316: (0)
4317: (0)
4318: (0)
4319: (0)
4320: (0)
4321: (0)
4322: (0)
4323: (0)
4324: (0)
4325: (0)
4326: (0)
4327: (0)
4328: (0)
4329: (0)
4330: (0)
4331: (0)
4332: (0)
4333: (0)
4334: (0)
4335: (0)
4336: (0)
4337: (0)
4338: (0)
4339: (0)
4340: (0)
4341: (0)
4342: (0)
4343: (0)
4344: (0)
4345: (0)
4346: (0)
4347: (0)
4348: (0)
4349: (0)
4350: (0)
4351: (0)
4352: (0)
4353: (0)
4354: (0)
4355: (0)
4356: (0)
4357: (0)
4358: (0)
4359: (0)
4360: (0)
4361: (0)
4362: (0)
4363: (0)
4364: (0)
4365: (0)
4366: (0)
4367: (0)
4368: (0)
4369: (0)
4370: (0)
4371: (0)
4372: (0)
4373: (0)
4374: (0)
4375: (0)
4376: (0)
4377: (0)
4378: (0)
4379: (0)
4380: (0)
4381: (0)
4382: (0)
4383: (0)
4384: (0)
4385: (0)
4386: (0)
4387: (0)
4388: (0)
4389: (0)
4390: (0)
4391: (0)
4392: (0)
4393: (0)
4394: (0)
4395: (0)
4396: (0)
4397: (0)
4398: (0)
4399: (0)
4400: (0)
4401: (0)
4402: (0)
4403: (0)
4404: (0)
4405: (0)
4406: (0)
4407: (0)
4408: (0)
4409: (0)
4410: (0)
4411: (0)
4412: (0)
4413: (0)
4414: (0)
4415: (0)
4416: (0)
4417: (0)
4418: (0)
4419: (0)
4420: (0)
4421: (0)
4422: (0)
4423: (0)
4424: (0)
4425: (0)
4426: (0)
4427: (0)
4428: (0)
4429: (0)
4430: (0)
4431: (0)
4432: (0)
4433: (0)
4434: (0)
4435: (0)
4436: (0)
4437: (0)
4438: (0)
4439: (0)
4440: (0)
4441: (0)
4442: (0)
4443: (0)
4444: (0)
4445: (0)
4446: (0)
4447: (0)
4448: (0)
4449: (0)
4450: (0)
4451: (0)
4452: (0)
4453: (0)
4454: (0)
4455: (0)
4456: (0)
4457: (0)
4458: (0)
4459: (0)
4460: (0)
4461: (0)
4462: (0)
4463: (0)
4464: (0)
4465: (0)
4466: (0)
4467: (0)
4468: (0)
4469: (0)
4470: (0)
4471: (0)
4472: (0)
4473: (0)
4474: (0)
4475: (0)
4476: (0)
4477: (0)
4478: (0)
4479: (0)
4480: (0)
4481: (0)
4482: (0)
4483: (0)
4484: (0)
4485: (0)
4486: (0)
4487: (0)
4488: (0)
4489: (0)
4490: (0)
4491: (0)
4492: (0)
4493: (0)
4494: (0)
4495: (0)
4496: (0)
4497: (0)
4498: (0)
4499: (0)
4500: (0)
4501: (0)
4502: (0)
4503: (0)
4504: (0)
4505: (0)
4506: (0)
4507: (0)
4508: (0)
4509: (0)
4510: (0)
4511: (0)
4512: (0)
4513: (0)
4514: (0)
4515: (0)
4516: (0)
4517: (0)
4518: (0)
4519: (0)
4520: (0)
4521: (0)
4522: (0)
4523: (0)
4524: (0)
4525: (0)
4526: (0)
4527: (0)
4528: (0)
4529: (0)
4530: (0)
4531: (0)
4532: (0)
4533: (0)
4534: (0)
4535: (0)
4536: (0)
4537: (0)
4538: (0)
4539: (0)
4540: (0)
4541: (0)
4542: (0)
4543: (0)
4544: (0)
4545: (0)
4546: (0)
4547: (0)
4548: (0)
4549: (0)
4550: (0)
4551: (0)
4552: (0)
4553: (0)
4554: (0)
4555: (0)
4556: (0)
4557: (0)
4558: (0)
4559: (0)
4560: (0)
4561: (0)
4562: (0)
4563: (0)
4564: (0)
4565: (0)
4566: (0)
4567: (0)
4568: (0)
4569: (0)
4570: (0)
4571: (0)
4572: (0)
4573: (0)
4574: (0)
4575: (0)
4576: (0)
4577: (0)
4578: (0)
4579: (0)
4580: (0)
4581: (0)
4582: (0)
4583: (0)
4584: (0)
4585: (0)
4586: (0)
4587: (0)
4588: (0)
4589: (0)
4590: (0)
4591: (0)
4592: (0)
4593: (0)
4594: (0)
4595: (0)
4596: (0)
4597: (0)
4598: (0)
4599: (0)
4600: (0)
4601: (0)
4602: (0)
4603: (0)
4604: (0)
4605: (0)
4606: (0)
4607: (0)
4608: (0)
4609: (0)
4610: (0)
4611: (0)
4612: (0)
4613: (0)
4614: (0)
4615: (0)
4616: (0)
4617: (0)
4618: (0)
4619: (0)
4620: (0)
4621: (0)
4622: (0)
4623: (0)
4624: (0)
4625: (0)
4626: (0)
4627: (0)
4628: (0)
4629: (0)
4630: (0)
4631: (0)
4632: (0)
4633: (0)
4634: (0)
4635: (0)
4636: (0)
4637: (0)
4638: (0)
4639: (0)
4640: (0)
4641: (0)
4642: (0)
4643: (0)
4644: (0)
4645: (0)
4646: (0)
4647: (0)
4648: (0)
4649: (0)
4650: (0)
4651: (0)
4652: (0)
4653: (0)
4654: (0)
4655: (0)
4656: (0)
4657: (0)
4658: (0)
4659: (0)
4660: (0)
4661: (0)
4662: (0)
4663: (0)
4664: (0)
4665: (0)
4666: (0)
4667: (0)
4668: (0)
4669: (0)
4670: (0)
4671: (0)
4672: (0)
4673: (0)
4674: (0)
4675: (0)
4676: (0)
4677: (0)
4678: (0)
4679: (0)
4680: (0)
4681: (0)
4682: (0)
4683: (0)
4684: (0)
4685: (0)
4686: (0)
4687: (0)
4688: (0)
4689: (0)
4690: (0)
4691: (0)
4692: (0)
4693: (0)
4694: (0)
4695: (0)
4696: (0)
4697: (0)
4698: (0)
4699: (0)
4700: (0)
4701: (0)
4702: (0)
4703: (0)
4704: (0)
4705: (0)
4706: (0)
4707: (0)
4708: (0)
4709: (0)
4710: (0)
4711: (0)
4712: (0)
4713: (0)
4714: (0)
4715: (0)
4716: (0)
4717: (0)
4718: (0)
4719: (0)
4720: (0)
4721: (0)
4722: (0)
4723: (0)
4724: (0)
4725: (0)
4726: (0)
4727: (0)
4728: (0)
4729: (0)
4730: (0)
4731: (0)
4732: (0)
4733: (0)
4734: (0)
4735: (0)
4736: (0)
4737: (0)
4738: (0)
4739: (0)
4740: (0)
4741: (0)
4742: (0)
4743: (0)
4744: (0)
4745: (0)
4746: (0)
4747: (0)
4748: (0)
4749: (0)
4750: (0)
4751: (0)
4752: (0)
4753: (0)
4754: (0)
4755: (0)
4756: (0)
4757: (0)
4758: (0)
4759: (0)
4760: (0)
4761: (0)
4762: (0)
4763: (0)
4764: (0)
4765: (0)
4766: (0)
4767: (0)
4768: (0)
4769: (0)
4770: (0)
4771: (0)
4772: (0)
4773: (0)
4774: (0)
4775: (0)
4776: (0)
4777: (0)
4778: (0)
4779: (0)
4780: (0)
4781: (0)
4782: (0)
4783: (0)
4784: (0)
4785: (0)
4786: (0)
4787: (0)
4788: (0)
4789: (0)
4790: (0)
4791: (0)
4792: (0)
4793: (0)
4794: (0)
4795: (0)
4796: (0)
4797: (0)
4798: (0)
4799: (0)
4800: (0)
4801: (0)
4802: (0)
4803: (0)
4804: (0)
4805: (0)
4806: (0)
4807: (0)
4808: (0)
4809: (0)
4810: (0)
4811: (0)
4812: (0)
4813: (0)
4814: (0)
4815: (0)
4816: (0)
4817: (0)
481
```

```

3621: (8) arr = np.array([0, 1, 1]*size, dtype=dtype)
3622: (8) acc = ufunc.accumulate(arr, dtype='?')
3623: (8) acc_u8 = acc.view(np.uint8)
3624: (8) exp = np.array(list(itertools.accumulate(arr, ufunc)), dtype=np.uint8)
3625: (8) assert_equal(exp, acc_u8)
3626: (0) @pytest.mark.parametrize("ufunc, dtype", [
3627: (4)     (ufunc, t[0])
3628: (4)     for ufunc in UFUNCS_BINARY_ACC
3629: (4)     for t in ufunc.types
3630: (4)     if t[0] == t[1] and t[0] == t[-1] and t[0] not in 'DFGMmO?'
3631: (0) ])
3632: (0) def test_memoverlap_accumulate_symmetric(ufunc, dtype):
3633: (4)     if ufunc.signature:
3634: (8)         pytest.skip('For generic signatures only')
3635: (4)     with np.errstate(all='ignore'):
3636: (8)         for size in (2, 8, 32, 64, 128, 256):
3637: (12)             arr = np.array([0, 1, 2]*size).astype(dtype)
3638: (12)             acc = ufunc.accumulate(arr, dtype=dtype)
3639: (12)             exp = np.array(list(itertools.accumulate(arr, ufunc)),
3640: (12)                         dtype=dtype)
3641: (0)             assert_equal(exp, acc)
3642: (4)     def test_signaling_nan_exceptions():
3643: (8)         with assert_no_warnings():
3644: (8)             a = np.ndarray(shape=(), dtype='float32', buffer=b'\x00\xe0\xbf\xff')
3645: (0)             np.isnan(a)
3646: (4)     @pytest.mark.parametrize("arr", [
3647: (4)         np.arange(2),
3648: (4)         np.matrix([0, 1]),
3649: (4)         np.matrix([[0, 1], [2, 5]]),
3650: (0)     ])
3651: (4)     def test_outer_subclass_preserve(arr):
3652: (4)         class foo(np.ndarray): pass
3653: (4)         actual = np.multiply.outer(arr.view(foo), arr.view(foo))
3654: (0)         assert actual.__class__.__name__ == 'foo'
3655: (4)     def test_outer_bad_subclass():
3656: (8)         class BadArr1(np.ndarray):
3657: (12)             def __array_finalize__(self, obj):
3658: (16)                 if self.ndim == 3:
3659: (8)                     self.shape = self.shape + (1,)
3660: (12)             def __array_prepare__(self, obj, context=None):
3661: (4)                 return obj
3662: (8)         class BadArr2(np.ndarray):
3663: (12)             def __array_finalize__(self, obj):
3664: (16)                 if isinstance(obj, BadArr2):
3665: (20)                     if self.shape[-1] == 1:
3666: (8)                         self.shape = self.shape[::-1]
3667: (12)             def __array_prepare__(self, obj, context=None):
3668: (4)                 return obj
3669: (8)         for cls in [BadArr1, BadArr2]:
3670: (8)             arr = np.ones((2, 3)).view(cls)
3671: (12)             with assert_raises(TypeError) as a:
3672: (8)                 np.add.outer(arr, [1, 2])
3673: (8)             arr = np.ones((2, 3)).view(cls)
3674: (0)             assert type(np.add.outer([1, 2], arr)) is cls
3675: (4)     def test_outer_exceeds_maxdims():
3676: (4)         deep = np.ones((1,) * 17)
3677: (8)         with assert_raises(ValueError):
3678: (0)             np.add.outer(deep, deep)
3679: (4)     def test_bad_legacy_ufunc_silent_errors():
3680: (4)         arr = np.arange(3).astype(np.float64)
3681: (8)         with pytest.raises(RuntimeError, match=r"How unexpected :\)!"):
3682: (4)             ncu_tests.always_error(arr, arr)
3683: (8)         with pytest.raises(RuntimeError, match=r"How unexpected :\)!"):
3684: (8)             non_contig = arr.repeat(20).reshape(-1, 6)[:, ::2]
3685: (4)             ncu_tests.always_error(non_contig, arr)
3686: (8)         with pytest.raises(RuntimeError, match=r"How unexpected :\)!"):
3687: (4)             ncu_tests.always_error.outer(arr, arr)
3688: (8)         with pytest.raises(RuntimeError, match=r"How unexpected :\)!"):
3689: (4)             ncu_tests.always_error.reduce(arr)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3689: (4)             with pytest.raises(RuntimeError, match=r"How unexpected :\!)!"):
3690: (8)                 ncu_tests.always_error.reduceat(arr, [0, 1])
3691: (4)             with pytest.raises(RuntimeError, match=r"How unexpected :\!)!"):
3692: (8)                 ncu_tests.always_error.accumulate(arr)
3693: (4)             with pytest.raises(RuntimeError, match=r"How unexpected :\!)!"):
3694: (8)                 ncu_tests.always_error.at(arr, [0, 1, 2], arr)
3695: (0) @pytest.mark.parametrize('x1', [np.arange(3.0), [0.0, 1.0, 2.0]])
3696: (0) def test_bad_legacy_gufunc_silent_errors(x1):
3697: (4)             with pytest.raises(RuntimeError, match=r"How unexpected :\!)!"):
3698: (8)                 ncu_tests.always_error_gufunc(x1, 0.0)

```

---

## File 131 - test\_umath\_accuracy.py:

```

1: (0)             import numpy as np
2: (0)             import os
3: (0)             from os import path
4: (0)             import sys
5: (0)             import pytest
6: (0)             from ctypes import c_longlong, c_double, c_float, c_int, cast, pointer,
POINTER
7: (0)             from numpy.testing import assert_array_max_ulp
8: (0)             from numpy.testing._private.utils import _glibc_older_than
9: (0)             from numpy.core._multiarray_umath import __cpu_features__
10: (0)             UNARY_UFUNCS = [obj for obj in np.core.umath.__dict__.values() if
11: (8)                 isinstance(obj, np.ufunc)]
12: (0)             UNARY_OBJECT_UFUNCS = [uf for uf in UNARY_UFUNCS if "0->0" in uf.types]
13: (0)             UNARY_OBJECT_UFUNCS.remove(getattr(np, 'invert'))
14: (0)             IS_AVX = __cpu_features__.get('AVX512F', False) or \
15: (8)                 (__cpu_features__.get('FMA3', False) and __cpu_features__.get('AVX2',
False))
16: (0)             runtest = (sys.platform.startswith('linux')
17: (11)                 and IS_AVX and not _glibc_older_than("2.17"))
18: (0)             platform_skip = pytest.mark.skipif(not runtest,
19: (35)                             reason="avoid testing inconsistent platform
"
20: (35)
21: (0)             "library implementations")
22: (4)             def convert(s, datatype="np.float32"):
23: (4)                 i = int(s, 16)                      # convert from hex to a Python int
24: (8)                 cp = pointer(c_longlong(i))           # make this into a c long long
integer
25: (8)                 fp = cast(cp, POINTER(c_double))    # cast the int pointer to a double
pointer
26: (4)
27: (8)                 else:
28: (8)                     cp = pointer(c_int(i))            # make this into a c integer
29: (8)                     fp = cast(cp, POINTER(c_float))   # cast the int pointer to a float
30: (0)             return fp.contents.value        # dereference the pointer, get the float
str_to_float = np.vectorize(convert)
31: (0)             class TestAccuracy:
32: (4)                 @platform_skip
33: (4)                 def test_validate_transcendentals(self):
34: (8)                     with np.errstate(all='ignore'):
35: (12)                         data_dir = path.join(path.dirname(__file__), 'data')
36: (12)                         files = os.listdir(data_dir)
37: (12)                         files = list(filter(lambda f: f.endswith('.csv'), files))
38: (12)                         for filename in files:
39: (16)                             filepath = path.join(data_dir, filename)
40: (16)                             with open(filepath) as fid:
41: (20)                                 file_without_comments = (r for r in fid if not r[0] in
('$', '#'))
42: (20)                                 data = np.genfromtxt(file_without_comments,
43: (41)                                     dtype=('|S39','|S39','|S39',int),
44: (41)                                     names=
('type','input','output','ulperr'),
45: (41)                                     delimiter=',',
46: (41)                                     skip_header=1)

```

```

47: (20)                     npname = path.splitext(filename)[0].split('-')[3]
48: (20)                     npfunc = getattr(np, npname)
49: (20)                     for datatype in np.unique(data['type']):
50: (24)                         data_subset = data[data['type'] == datatype]
51: (24)                         inval =
np.array(str_to_float(data_subset['input'].astype(str), data_subset['type'].astype(str)),
dtype=eval(datatype))
52: (24)                         outval =
np.array(str_to_float(data_subset['output'].astype(str), data_subset['type'].astype(str)),
dtype=eval(datatype))
53: (24)                     perm = np.random.permutation(len(inval))
54: (24)                     inval = inval[perm]
55: (24)                     outval = outval[perm]
56: (24)                     maxulperr = data_subset['ulperr'].max()
57: (24)                     assert_array_max_ulp(npfunc(inval), outval, maxulperr)
58: (4) @pytest.mark.parametrize("ufunc", UNARY_OBJECT_UFUNCS)
59: (4) def test_validate_fp16_transcendentals(self, ufunc):
60: (8)     with np.errstate(all='ignore'):
61: (12)         arr = np.arange(65536, dtype=np.int16)
62: (12)         datafp16 = np.frombuffer(arr.tobytes(), dtype=np.float16)
63: (12)         datafp32 = datafp16.astype(np.float32)
64: (12)         assert_array_max_ulp(ufunc(datafp16), ufunc(datafp32),
65: (20)                         maxulp=1, dtype=np.float16)

```

---

## File 132 - test\_umath\_complex.py:

```

1: (0)         import sys
2: (0)         import platform
3: (0)         import pytest
4: (0)         import numpy as np
5: (0)         import numpy.core._multiarray_umath as ncu
6: (0)         from numpy.testing import (
7: (4)             assert_raises, assert_equal, assert_array_equal, assert_almost_equal,
assert_array_max_ulp
8: (4)         )
9: (0)         with np.errstate(all='ignore'):
10: (4)             functions_seem_flaky = ((np.exp(complex(np.inf, 0)).imag != 0)
11: (28)                 or (np.log(complex(np.NZERO, 0)).imag != np.pi))
12: (0)             xfail_complex_tests = (not sys.platform.startswith('linux') or
functions_seem_flaky)
13: (0)             platform_skip = pytest.mark.skipif(xfail_complex_tests,
14: (35)                             reason="Inadequate C99 complex support")
15: (0)             class TestCexp:
16: (4)                 def test_simple(self):
17: (8)                     check = check_complex_value
18: (8)                     f = np.exp
19: (8)                     check(f, 1, 0, np.exp(1), 0, False)
20: (8)                     check(f, 0, 1, np.cos(1), np.sin(1), False)
21: (8)                     ref = np.exp(1) * complex(np.cos(1), np.sin(1))
22: (8)                     check(f, 1, 1, ref.real, ref.imag, False)
23: (4) @platform_skip
24: (4)                 def test_special_values(self):
25: (8)                     check = check_complex_value
26: (8)                     f = np.exp
27: (8)                     check(f, np.PZERO, 0, 1, 0, False)
28: (8)                     check(f, np.NZERO, 0, 1, 0, False)
29: (8)                     check(f, 1, np.inf, np.nan, np.nan)
30: (8)                     check(f, -1, np.inf, np.nan, np.nan)
31: (8)                     check(f, 0, np.inf, np.nan, np.nan)
32: (8)                     check(f, np.inf, 0, np.inf, 0)
33: (8)                     check(f, -np.inf, 1, np.PZERO, np.PZERO)
34: (8)                     check(f, -np.inf, 0.75 * np.pi, np.NZERO, np.PZERO)
35: (8)                     check(f, np.inf, 1, np.inf, np.inf)
36: (8)                     check(f, np.inf, 0.75 * np.pi, -np.inf, np.inf)
37: (8)                     def _check_ninf_inf(dummy):
38: (12)                         msgform = "cexp(-inf, inf) is (%f, %f), expected (+-0, +-0)"
39: (12)                         with np.errstate(invalid='ignore'):

```

```

40: (16)             z = f(np.array(complex(-np.inf, np.inf)))
41: (16)             if z.real != 0 or z.imag != 0:
42: (20)                 raise AssertionError(msgform % (z.real, z.imag))
43: (8)             _check_ninf_inf(None)
44: (8)         def _check_inf_inf(dummy):
45: (12)             msgform = "cexp(inf, inf) is (%f, %f), expected (+-inf, nan)"
46: (12)             with np.errstate(invalid='ignore'):
47: (16)                 z = f(np.array(complex(np.inf, np.inf)))
48: (16)                 if not np.isinf(z.real) or not np.isnan(z.imag):
49: (20)                     raise AssertionError(msgform % (z.real, z.imag))
50: (8)             _check_inf_inf(None)
51: (8)         def _check_ninf_nan(dummy):
52: (12)             msgform = "cexp(-inf, nan) is (%f, %f), expected (+-0, +-0)"
53: (12)             with np.errstate(invalid='ignore'):
54: (16)                 z = f(np.array(complex(-np.inf, np.nan)))
55: (16)                 if z.real != 0 or z.imag != 0:
56: (20)                     raise AssertionError(msgform % (z.real, z.imag))
57: (8)             _check_ninf_nan(None)
58: (8)         def _check_inf_nan(dummy):
59: (12)             msgform = "cexp(-inf, nan) is (%f, %f), expected (+-inf, nan)"
60: (12)             with np.errstate(invalid='ignore'):
61: (16)                 z = f(np.array(complex(np.inf, np.nan)))
62: (16)                 if not np.isinf(z.real) or not np.isnan(z.imag):
63: (20)                     raise AssertionError(msgform % (z.real, z.imag))
64: (8)             _check_inf_nan(None)
65: (8)         check(f, np.nan, 1, np.nan, np.nan)
66: (8)         check(f, np.nan, -1, np.nan, np.nan)
67: (8)         check(f, np.nan, np.inf, np.nan, np.nan)
68: (8)         check(f, np.nan, -np.inf, np.nan, np.nan)
69: (8)         check(f, np.nan, np.nan, np.nan, np.nan)
70: (4)     @pytest.mark.skip(reason="cexp(nan + 0I) is wrong on most platforms")
71: (4)     def test_special_values2(self):
72: (8)         check = check_complex_value
73: (8)         f = np.exp
74: (8)         check(f, np.nan, 0, np.nan, 0)
75: (0)     class TestClog:
76: (4)         def test_simple(self):
77: (8)             x = np.array([1+0j, 1+2j])
78: (8)             y_r = np.log(np.abs(x)) + 1j * np.angle(x)
79: (8)             y = np.log(x)
80: (8)             assert_almost_equal(y, y_r)
81: (4)     @platform_skip
82: (4)     @pytest.mark.skipif(platform.machine() == "armv5tel", reason="See gh-
413.")
83: (4)     def test_special_values(self):
84: (8)         xl = []
85: (8)         yl = []
86: (8)         with np.errstate(divide='raise'):
87: (12)             x = np.array([np.NZERO], dtype=complex)
88: (12)             y = complex(-np.inf, np.pi)
89: (12)             assert_raises(FloatingPointError, np.log, x)
90: (8)         with np.errstate(divide='ignore'):
91: (12)             assert_almost_equal(np.log(x), y)
92: (8)             xl.append(x)
93: (8)             yl.append(y)
94: (8)         with np.errstate(divide='raise'):
95: (12)             x = np.array([0], dtype=complex)
96: (12)             y = complex(-np.inf, 0)
97: (12)             assert_raises(FloatingPointError, np.log, x)
98: (8)         with np.errstate(divide='ignore'):
99: (12)             assert_almost_equal(np.log(x), y)
100: (8)             xl.append(x)
101: (8)             yl.append(y)
102: (8)             x = np.array([complex(1, np.inf)], dtype=complex)
103: (8)             y = complex(np.inf, 0.5 * np.pi)
104: (8)             assert_almost_equal(np.log(x), y)
105: (8)             xl.append(x)
106: (8)             yl.append(y)
107: (8)             x = np.array([complex(-1, np.inf)], dtype=complex)

```

```

108: (8)           assert_almost_equal(np.log(x), y)
109: (8)           xl.append(x)
110: (8)           yl.append(y)
111: (8)           with np.errstate(invalid='raise'):
112: (12)             x = np.array([complex(1., np.nan)], dtype=complex)
113: (12)             y = complex(np.nan, np.nan)
114: (8)             with np.errstate(invalid='ignore'):
115: (12)               assert_almost_equal(np.log(x), y)
116: (8)               xl.append(x)
117: (8)               yl.append(y)
118: (8)               with np.errstate(invalid='raise'):
119: (12)                 x = np.array([np.inf + 1j * np.nan], dtype=complex)
120: (8)                 with np.errstate(invalid='ignore'):
121: (12)                   assert_almost_equal(np.log(x), y)
122: (8)                   xl.append(x)
123: (8)                   yl.append(y)
124: (8)                   x = np.array([-np.inf + 1j], dtype=complex)
125: (8)                   y = complex(np.inf, np.pi)
126: (8)                   assert_almost_equal(np.log(x), y)
127: (8)                   xl.append(x)
128: (8)                   yl.append(y)
129: (8)                   x = np.array([np.inf + 1j], dtype=complex)
130: (8)                   y = complex(np.inf, 0)
131: (8)                   assert_almost_equal(np.log(x), y)
132: (8)                   xl.append(x)
133: (8)                   yl.append(y)
134: (8)                   x = np.array([complex(-np.inf, np.inf)], dtype=complex)
135: (8)                   y = complex(np.inf, 0.75 * np.pi)
136: (8)                   assert_almost_equal(np.log(x), y)
137: (8)                   xl.append(x)
138: (8)                   yl.append(y)
139: (8)                   x = np.array([complex(np.inf, np.inf)], dtype=complex)
140: (8)                   y = complex(np.inf, 0.25 * np.pi)
141: (8)                   assert_almost_equal(np.log(x), y)
142: (8)                   xl.append(x)
143: (8)                   yl.append(y)
144: (8)                   x = np.array([complex(np.inf, np.nan)], dtype=complex)
145: (8)                   y = complex(np.inf, np.nan)
146: (8)                   assert_almost_equal(np.log(x), y)
147: (8)                   xl.append(x)
148: (8)                   yl.append(y)
149: (8)                   x = np.array([complex(-np.inf, np.nan)], dtype=complex)
150: (8)                   assert_almost_equal(np.log(x), y)
151: (8)                   xl.append(x)
152: (8)                   yl.append(y)
153: (8)                   x = np.array([complex(np.nan, 1)], dtype=complex)
154: (8)                   y = complex(np.nan, np.nan)
155: (8)                   assert_almost_equal(np.log(x), y)
156: (8)                   xl.append(x)
157: (8)                   yl.append(y)
158: (8)                   x = np.array([complex(np.nan, np.inf)], dtype=complex)
159: (8)                   y = complex(np.inf, np.nan)
160: (8)                   assert_almost_equal(np.log(x), y)
161: (8)                   xl.append(x)
162: (8)                   yl.append(y)
163: (8)                   x = np.array([complex(np.nan, np.nan)], dtype=complex)
164: (8)                   y = complex(np.nan, np.nan)
165: (8)                   assert_almost_equal(np.log(x), y)
166: (8)                   xl.append(x)
167: (8)                   yl.append(y)
168: (8)                   xa = np.array(xl, dtype=complex)
169: (8)                   ya = np.array(yl, dtype=complex)
170: (8)                   with np.errstate(divide='ignore'):
171: (12)                     for i in range(len(xa)):
172: (16)                       assert_almost_equal(np.log(xa[i].conj()), ya[i].conj())
173: (0)           class TestCsqrt:
174: (4)             def test_simple(self):
175: (8)               check_complex_value(np.sqrt, 1, 0, 1, 0)
176: (8)               rres = 0.5*np.sqrt(2)

```

```

177: (8)             ires = rres
178: (8)             check_complex_value(np.sqrt, 0, 1, rres, ires, False)
179: (8)             check_complex_value(np.sqrt, -1, 0, 0, 1)
180: (4)             def test_simple_conjugate(self):
181: (8)                 ref = np.conj(np.sqrt(complex(1, 1)))
182: (8)                 def f(z):
183: (12)                     return np.sqrt(np.conj(z))
184: (8)                     check_complex_value(f, 1, 1, ref.real, ref.imag, False)
185: (4)             @platform_skip
186: (4)             def test_special_values(self):
187: (8)                 check = check_complex_value
188: (8)                 f = np.sqrt
189: (8)                 check(f, np.PZERO, 0, 0, 0)
190: (8)                 check(f, np.NZERO, 0, 0, 0)
191: (8)                 check(f, 1, np.inf, np.inf, np.inf)
192: (8)                 check(f, -1, np.inf, np.inf, np.inf)
193: (8)                 check(f, np.PZERO, np.inf, np.inf, np.inf)
194: (8)                 check(f, np.NZERO, np.inf, np.inf, np.inf)
195: (8)                 check(f, np.inf, np.inf, np.inf, np.inf)
196: (8)                 check(f, -np.inf, np.inf, np.inf, np.inf)
197: (8)                 check(f, -np.nan, np.inf, np.inf, np.inf)
198: (8)                 check(f, 1, np.nan, np.nan, np.nan)
199: (8)                 check(f, -1, np.nan, np.nan, np.nan)
200: (8)                 check(f, 0, np.nan, np.nan, np.nan)
201: (8)                 check(f, -np.inf, 1, np.PZERO, np.inf)
202: (8)                 check(f, np.inf, 1, np.inf, np.PZERO)
203: (8)             def _check_ninf_nan(dummy):
204: (12)                 msgform = "csqrt(-inf, nan) is (%f, %f), expected (nan, +-inf)"
205: (12)                 z = np.sqrt(np.array(complex(-np.inf, np.nan)))
206: (12)                 with np.errstate(invalid='ignore'):
207: (16)                     if not (np.isnan(z.real) and np.isinf(z.imag)):
208: (20)                         raise AssertionError(msgform % (z.real, z.imag))
209: (8)             _check_ninf_nan(None)
210: (8)             check(f, np.inf, np.nan, np.inf, np.nan)
211: (8)             check(f, np.nan, 0, np.nan, np.nan)
212: (8)             check(f, np.nan, 1, np.nan, np.nan)
213: (8)             check(f, np.nan, np.nan, np.nan, np.nan)
214: (0)             class TestCpow:
215: (4)                 def setup_method(self):
216: (8)                     self.olderr = np.seterr(invalid='ignore')
217: (4)                 def teardown_method(self):
218: (8)                     np.seterr(**self.olderr)
219: (4)                 def test_simple(self):
220: (8)                     x = np.array([1+1j, 0+2j, 1+2j, np.inf, np.nan])
221: (8)                     y_r = x ** 2
222: (8)                     y = np.power(x, 2)
223: (8)                     assert_almost_equal(y, y_r)
224: (4)                 def test_scalar(self):
225: (8)                     x = np.array([1, 1j, 2, 2.5+.37j, np.inf, np.nan])
226: (8)                     y = np.array([1, 1j, -0.5+1.5j, -0.5+1.5j, 2, 3])
227: (8)                     lx = list(range(len(x)))
228: (8)                     p_r = [
229: (12)                         1+0j,
230: (12)                         0.20787957635076193+0j,
231: (12)                         0.35812203996480685+0.6097119028618724j,
232: (12)                         0.12659112128185032+0.48847676699581527j,
233: (12)                         complex(np.inf, np.nan),
234: (12)                         complex(np.nan, np.nan),
235: (8)                     ]
236: (8)                     n_r = [x[i] ** y[i] for i in lx]
237: (8)                     for i in lx:
238: (12)                         assert_almost_equal(n_r[i], p_r[i], err_msg='Loop %d\n' % i)
239: (4)                 def test_array(self):
240: (8)                     x = np.array([1, 1j, 2, 2.5+.37j, np.inf, np.nan])
241: (8)                     y = np.array([1, 1j, -0.5+1.5j, -0.5+1.5j, 2, 3])
242: (8)                     lx = list(range(len(x)))
243: (8)                     p_r = [
244: (12)                         1+0j,
245: (12)                         0.20787957635076193+0j,

```

```

246: (12)          0.35812203996480685+0.6097119028618724j,
247: (12)          0.12659112128185032+0.48847676699581527j,
248: (12)          complex(np.inf, np.nan),
249: (12)          complex(np.nan, np.nan),
250: (8)           ]
251: (8)          n_r = x ** y
252: (8)          for i in lx:
253: (12)          assert_almost_equal(n_r[i], p_r[i], err_msg='Loop %d\n' % i)
254: (0)           class TestCabs:
255: (4)             def setup_method(self):
256: (8)               self.olderr = np.seterr(invalid='ignore')
257: (4)             def teardown_method(self):
258: (8)               np.seterr(**self.olderr)
259: (4)             def test_simple(self):
260: (8)               x = np.array([1+1j, 0+2j, 1+2j, np.inf, np.nan])
261: (8)               y_r = np.array([np.sqrt(2.), 2, np.sqrt(5), np.inf, np.nan])
262: (8)               y = np.abs(x)
263: (8)               assert_almost_equal(y, y_r)
264: (4)             def test_fabs(self):
265: (8)               x = np.array([1+0j], dtype=complex)
266: (8)               assert_array_equal(np.abs(x), np.real(x))
267: (8)               x = np.array([complex(1, np.NZERO)], dtype=complex)
268: (8)               assert_array_equal(np.abs(x), np.real(x))
269: (8)               x = np.array([complex(np.inf, np.NZERO)], dtype=complex)
270: (8)               assert_array_equal(np.abs(x), np.real(x))
271: (8)               x = np.array([complex(np.nan, np.NZERO)], dtype=complex)
272: (8)               assert_array_equal(np.abs(x), np.real(x))
273: (4)             def test_cabs_inf_nan(self):
274: (8)               x, y = [], []
275: (8)               x.append(np.nan)
276: (8)               y.append(np.nan)
277: (8)               check_real_value(np.abs, np.nan, np.nan, np.nan)
278: (8)               x.append(np.nan)
279: (8)               y.append(-np.nan)
280: (8)               check_real_value(np.abs, -np.nan, np.nan, np.nan)
281: (8)               x.append(np.inf)
282: (8)               y.append(np.nan)
283: (8)               check_real_value(np.abs, np.inf, np.nan, np.inf)
284: (8)               x.append(-np.inf)
285: (8)               y.append(np.nan)
286: (8)               check_real_value(np.abs, -np.inf, np.nan, np.inf)
287: (8)             def f(a):
288: (12)               return np.abs(np.conj(a))
289: (8)             def g(a, b):
290: (12)               return np.abs(complex(a, b))
291: (8)               xa = np.array(x, dtype=complex)
292: (8)               assert len(xa) == len(x) == len(y)
293: (8)               for xi, yi in zip(x, y):
294: (12)                 ref = g(xi, yi)
295: (12)                 check_real_value(f, xi, yi, ref)
296: (0)           class TestCarg:
297: (4)             def test_simple(self):
298: (8)               check_real_value(ncu._arg, 1, 0, 0, False)
299: (8)               check_real_value(ncu._arg, 0, 1, 0.5*np.pi, False)
300: (8)               check_real_value(ncu._arg, 1, 1, 0.25*np.pi, False)
301: (8)               check_real_value(ncu._arg, np.PZERO, np.PZERO, np.PZERO)
302: (4)             @pytest.mark.skip(
303: (8)               reason="Complex arithmetic with signed zero fails on most platforms")
304: (4)             def test_zero(self):
305: (8)               check_real_value(ncu._arg, np.NZERO, np.PZERO, np.pi, False)
306: (8)               check_real_value(ncu._arg, np.NZERO, np.NZERO, -np.pi, False)
307: (8)               check_real_value(ncu._arg, np.PZERO, np.PZERO, np.PZERO)
308: (8)               check_real_value(ncu._arg, np.PZERO, np.NZERO, np.NZERO)
309: (8)               check_real_value(ncu._arg, 1, np.PZERO, np.PZERO, False)
310: (8)               check_real_value(ncu._arg, 1, np.NZERO, np.NZERO, False)
311: (8)               check_real_value(ncu._arg, -1, np.PZERO, np.pi, False)
312: (8)               check_real_value(ncu._arg, -1, np.NZERO, -np.pi, False)
313: (8)               check_real_value(ncu._arg, np.PZERO, 1, 0.5 * np.pi, False)
314: (8)               check_real_value(ncu._arg, np.NZERO, 1, 0.5 * np.pi, False)

```

```

315: (8)             check_real_value(ncu._arg, np.PZERO, -1, 0.5 * np.pi, False)
316: (8)             check_real_value(ncu._arg, np.NZERO, -1, -0.5 * np.pi, False)
317: (4)             def test_special_values(self):
318: (8)                 check_real_value(ncu._arg, -np.inf, 1, np.pi, False)
319: (8)                 check_real_value(ncu._arg, -np.inf, -1, -np.pi, False)
320: (8)                 check_real_value(ncu._arg, np.inf, 1, np.PZERO, False)
321: (8)                 check_real_value(ncu._arg, np.inf, -1, np.NZERO, False)
322: (8)                 check_real_value(ncu._arg, 1, np.inf, 0.5 * np.pi, False)
323: (8)                 check_real_value(ncu._arg, 1, -np.inf, -0.5 * np.pi, False)
324: (8)                 check_real_value(ncu._arg, -np.inf, np.inf, 0.75 * np.pi, False)
325: (8)                 check_real_value(ncu._arg, -np.inf, -np.inf, -0.75 * np.pi, False)
326: (8)                 check_real_value(ncu._arg, np.inf, np.inf, 0.25 * np.pi, False)
327: (8)                 check_real_value(ncu._arg, np.inf, -np.inf, -0.25 * np.pi, False)
328: (8)                 check_real_value(ncu._arg, np.nan, 0, np.nan, False)
329: (8)                 check_real_value(ncu._arg, 0, np.nan, np.nan, False)
330: (8)                 check_real_value(ncu._arg, np.nan, np.inf, np.nan, False)
331: (8)                 check_real_value(ncu._arg, np.inf, np.nan, np.nan, False)
332: (0)             def check_real_value(f, x1, y1, x, exact=True):
333: (4)                 z1 = np.array([complex(x1, y1)])
334: (4)                 if exact:
335: (8)                     assert_equal(f(z1), x)
336: (4)                 else:
337: (8)                     assert_almost_equal(f(z1), x)
338: (0)             def check_complex_value(f, x1, y1, x2, y2, exact=True):
339: (4)                 z1 = np.array([complex(x1, y1)])
340: (4)                 z2 = complex(x2, y2)
341: (4)                 with np.errstate(invalid='ignore'):
342: (8)                     if exact:
343: (12)                         assert_equal(f(z1), z2)
344: (8)                     else:
345: (12)                         assert_almost_equal(f(z1), z2)
346: (0)             class TestSpecialComplexAVX:
347: (4)                 @pytest.mark.parametrize("stride", [-4, -2, -1, 1, 2, 4])
348: (4)                 @pytest.mark.parametrize("astype", [np.complex64, np.complex128])
349: (4)                 def test_array(self, stride, astype):
350: (8)                     arr = np.array([complex(np.nan, np.nan),
351: (24)                         complex(np.nan, np.inf),
352: (24)                         complex(np.inf, np.nan),
353: (24)                         complex(np.inf, np.inf),
354: (24)                         complex(0., np.inf),
355: (24)                         complex(np.inf, 0.),
356: (24)                         complex(0., 0.),
357: (24)                         complex(0., np.nan),
358: (24)                         complex(np.nan, 0.)], dtype=astype)
359: (8)                     abs_true = np.array([np.nan, np.inf, np.inf, np.inf, np.inf,
0., np.nan, np.nan], dtype=arr.real.dtype)
360: (8)                     sq_true = np.array([complex(np.nan, np.nan),
361: (28)                         complex(np.nan, np.nan),
362: (28)                         complex(np.nan, np.nan),
363: (28)                         complex(np.nan, np.inf),
364: (28)                         complex(-np.inf, np.nan),
365: (28)                         complex(np.inf, np.nan),
366: (28)                         complex(0., 0.),
367: (28)                         complex(np.nan, np.nan),
368: (28)                         complex(np.nan, np.nan)], dtype=astype)
369: (8)                     with np.errstate(invalid='ignore'):
370: (12)                         assert_equal(np.abs(arr[::-stride]), abs_true[::-stride])
371: (12)                         assert_equal(np.square(arr[::-stride]), sq_true[::-stride])
372: (0)             class TestComplexAbsoluteAVX:
373: (4)                 @pytest.mark.parametrize("arraysize",
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15, 17, 18, 19])
374: (4)                 @pytest.mark.parametrize("stride", [-4, -3, -2, -1, 1, 2, 3, 4])
375: (4)                 @pytest.mark.parametrize("astype", [np.complex64, np.complex128])
376: (4)                 def test_array(self, arraysize, stride, astype):
377: (8)                     arr = np.ones(arraysize, dtype=astype)
378: (8)                     abs_true = np.ones(arraysize, dtype=arr.real.dtype)
379: (8)                     assert_equal(np.abs(arr[::-stride]), abs_true[::-stride])
380: (0)             class TestComplexAbsoluteMixedDTypes:
381: (4)                 @pytest.mark.parametrize("stride", [-4, -3, -2, -1, 1, 2, 3, 4])

```

```

382: (4)          @pytest.mark.parametrize("astype", [np.complex64, np.complex128])
383: (4)          @pytest.mark.parametrize("func", ['abs', 'square', 'conjugate'])
384: (4)          def test_array(self, stride, astype, func):
385: (8)              dtype = [('template_id', '<i8'), ('bank_chisq','<f4'),
386: (17)                  ('bank_chisq_dof','<i8'), ('chisq', '<f4'),
387: (17)                  ('chisq_dof','<i8'), ('cont_chisq', '<f4'), ('psd_var_val', '<f4'),
388: (17)                  ('sg_chisq','<f4'), ('mycomplex', astype), ('time_index', '<i8')]
389: (8)          vec = np.array([
390: (15)              (0, 0., 0, -31.666483, 200, 0., 0., 1., 3.0+4.0j),
613090),
391: (15)              (1, 0., 0, 260.91525, 42, 0., 0., 1., 5.0+12.0j),
787315),
392: (15)              (1, 0., 0, 52.15155, 42, 0., 0., 1., 8.0+15.0j),
806641),
393: (15)              (1, 0., 0, 52.430195, 42, 0., 0., 1., 7.0+24.0j),
1363540),
394: (15)              (2, 0., 0, 304.43646, 58, 0., 0., 1., 20.0+21.0j),
787323),
395: (15)              (3, 0., 0, 299.42108, 52, 0., 0., 1., 12.0+35.0j),
787332),
396: (15)              (4, 0., 0, 39.4836, 28, 0., 0., 9.182192, 9.0+40.0j),
787304),
397: (15)              (4, 0., 0, 76.83787, 28, 0., 0., 1., 28.0+45.0j),
1321869),
398: (15)              (5, 0., 0, 143.26366, 24, 0., 0., 10.996129, 11.0+60.0j),
787299)], dtype=dtype)
399: (8)          myfunc = getattr(np, func)
400: (8)          a = vec['mycomplex']
401: (8)          g = myfunc(a[::-stride])
402: (8)          b = vec['mycomplex'].copy()
403: (8)          h = myfunc(b[::-stride])
404: (8)          assert_array_max_ulp(h.real, g.real, 1)
405: (8)          assert_array_max_ulp(h.imag, g.imag, 1)

```

---

File 133 - test\_unicode.py:

```

1: (0)          import pytest
2: (0)          import numpy as np
3: (0)          from numpy.testing import assert_, assert_equal, assert_array_equal
4: (0)          def buffer_length(arr):
5: (4)              if isinstance(arr, str):
6: (8)                  if not arr:
7: (12)                      charmax = 0
8: (8)                  else:
9: (12)                      charmax = max([ord(c) for c in arr])
10: (8)                  if charmax < 256:
11: (12)                      size = 1
12: (8)                  elif charmax < 65536:
13: (12)                      size = 2
14: (8)                  else:
15: (12)                      size = 4
16: (8)                  return size * len(arr)
17: (4)                  v = memoryview(arr)
18: (4)                  if v.shape is None:
19: (8)                      return len(v) * v.itemsize
20: (4)                  else:
21: (8)                      return np.prod(v.shape) * v.itemsize
22: (0)          ucs2_value = '\u0900'
23: (0)          ucs4_value = '\U00100900'
24: (0)          def test_string_cast():
25: (4)              str_arr = np.array(["1234", "1234\0\0"], dtype='S')
26: (4)              uni_arr1 = str_arr.astype('>U')
27: (4)              uni_arr2 = str_arr.astype('<U')
28: (4)              assert_array_equal(str_arr != uni_arr1, np.ones(2, dtype=bool))
29: (4)              assert_array_equal(uni_arr1 != str_arr, np.ones(2, dtype=bool))

```

```

30: (4) assert_array_equal(str_arr == uni_arr1, np.zeros(2, dtype=bool))
31: (4) assert_array_equal(uni_arr1 == str_arr, np.zeros(2, dtype=bool))
32: (4) assert_array_equal(uni_arr1, uni_arr2)
33: (0)
34: (4)     """Check the creation of zero-valued arrays"""
35: (4)     def content_check(self, ua, ua_scalar, nbytes):
36: (8)         assert_(int(ua.dtype.str[2:]) == self.ulen)
37: (8)         assert_(buffer_length(ua) == nbytes)
38: (8)         assert_(ua_scalar == '')
39: (8)         assert_(ua_scalar.encode('ascii') == b'')
40: (8)         assert_(buffer_length(ua_scalar) == 0)
41: (4)     def test_zeros0D(self):
42: (8)         ua = np.zeros(()), dtype='U%s' % self.ulen)
43: (8)         self.content_check(ua, ua[()], 4*self.ulen)
44: (4)     def test_zerosSD(self):
45: (8)         ua = np.zeros((2,), dtype='U%s' % self.ulen)
46: (8)         self.content_check(ua, ua[0], 4*self.ulen*2)
47: (8)         self.content_check(ua, ua[1], 4*self.ulen*2)
48: (4)     def test_zerosMD(self):
49: (8)         ua = np.zeros((2, 3, 4), dtype='U%s' % self.ulen)
50: (8)         self.content_check(ua, ua[0, 0, 0], 4*self.ulen*2*3*4)
51: (8)         self.content_check(ua, ua[-1, -1, -1], 4*self.ulen*2*3*4)
52: (0)
53: (4) class TestCreateZeros_1(CreateZeros):
54: (4)     """Check the creation of zero-valued arrays (size 1)"""
55: (0)     ulen = 1
56: (4) class TestCreateZeros_2(CreateZeros):
57: (4)     """Check the creation of zero-valued arrays (size 2)"""
58: (0)     ulen = 2
59: (4) class TestCreateZeros_1009(CreateZeros):
60: (4)     """Check the creation of zero-valued arrays (size 1009)"""
61: (0)     ulen = 1009
62: (4) class CreateValues:
63: (4)     """Check the creation of unicode arrays with values"""
64: (8)     def content_check(self, ua, ua_scalar, nbytes):
65: (8)         assert_(int(ua.dtype.str[2:]) == self.ulen)
66: (8)         assert_(buffer_length(ua) == nbytes)
67: (8)         assert_(ua_scalar == self.ucs_value*self.ulen)
68: (8)         assert_(ua_scalar.encode('utf-8') ==
69: (8)             (self.ucs_value*self.ulen).encode('utf-8'))
70: (12)         if self.ucs_value == ucs4_value:
71: (8)             assert_(buffer_length(ua_scalar) == 2*2*self.ulen)
72: (12)         else:
73: (4)             assert_(buffer_length(ua_scalar) == 2*self.ulen)
74: (8)     def test_values0D(self):
75: (8)         ua = np.array(self.ucs_value*self.ulen, dtype='U%s' % self.ulen)
76: (8)         self.content_check(ua, ua[()], 4*self.ulen)
77: (8)     def test_valuesSD(self):
78: (8)         ua = np.array([self.ucs_value*self.ulen]*2, dtype='U%s' % self.ulen)
79: (8)         self.content_check(ua, ua[0], 4*self.ulen*2)
80: (8)         self.content_check(ua, ua[1], 4*self.ulen*2)
81: (8)     def test_valuesMD(self):
82: (8)         ua = np.array([[self.ucs_value*self.ulen]*2]*3]*4, dtype='U%s' %
83: (8)             self.content_check(ua, ua[0, 0, 0], 4*self.ulen*2*3*4)
84: (8)             self.content_check(ua, ua[-1, -1, -1], 4*self.ulen*2*3*4)
85: (0) class TestCreateValues_1_UCS2(CreateValues):
86: (4)     """Check the creation of valued arrays (size 1, UCS2 values)"""
87: (4)     ulen = 1
88: (0)     ucs_value = ucs2_value
89: (4) class TestCreateValues_1_UCS4(CreateValues):
90: (4)     """Check the creation of valued arrays (size 1, UCS4 values)"""
91: (4)     ulen = 1
92: (0)     ucs_value = ucs4_value
93: (4) class TestCreateValues_2_UCS2(CreateValues):
94: (4)     """Check the creation of valued arrays (size 2, UCS2 values)"""
95: (4)     ulen = 2
96: (0)     ucs_value = ucs2_value
97: (4) class TestCreateValues_2_UCS4(CreateValues):
98: (4)     """Check the creation of valued arrays (size 2, UCS4 values)"""

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

98: (4)
99: (4)
100: (0)
101: (4)
102: (4)
103: (4)
104: (0)
105: (4)
106: (4)
107: (4)
108: (0)
109: (4)
110: (4)
111: (8)
112: (8)
113: (8)
114: (8)
115: (24)
116: (8)
117: (12)
118: (8)
119: (12)
120: (4)
121: (8)
122: (8)
123: (8)
124: (4)
125: (8)
126: (8)
127: (8)
128: (8)
129: (8)
130: (4)
131: (8)
132: (8)
133: (8)
134: (8)
135: (8)
136: (0)
137: (4)
138: (4)
139: (4)
140: (0)
141: (4)
142: (4)
143: (4)
144: (0)
145: (4)
146: (4)
147: (4)
148: (0)
149: (4)
150: (4)
151: (4)
152: (0)
153: (4)
154: (4)
155: (4)
156: (0)
157: (4)
158: (4)
159: (4)
160: (0)
161: (4)
162: (4)
163: (8)
164: (8)
165: (8)
166: (8)

        ulen = 2
        ucs_value = ucs4_value
class TestCreateValues_1009_UCS2(CreateValues):
    """Check the creation of valued arrays (size 1009, UCS2 values)"""
    ulen = 1009
    ucs_value = ucs2_value
class TestCreateValues_1009_UCS4(CreateValues):
    """Check the creation of valued arrays (size 1009, UCS4 values)"""
    ulen = 1009
    ucs_value = ucs4_value
class AssignValues:
    """Check the assignment of unicode arrays with values"""
    def content_check(self, ua, ua_scalar, nbytes):
        assert_(int(ua.dtype.str[2:]) == self.ulen)
        assert_(buffer_length(ua) == nbytes)
        assert_(ua_scalar == self.ucs_value*self.ulen)
        assert_(ua_scalar.encode('utf-8') ==
                (self.ucs_value*self.ulen).encode('utf-8'))
        if self.ucs_value == ucs4_value:
            assert_(buffer_length(ua_scalar) == 2*2*self.ulen)
        else:
            assert_(buffer_length(ua_scalar) == 2*self.ulen)
    def test_values0D(self):
        ua = np.zeros(), dtype='U%s' % self.ulen
        ua[:] = self.ucs_value*self.ulen
        self.content_check(ua, ua[:], 4*self.ulen)
    def test_valuesSD(self):
        ua = np.zeros((2,), dtype='U%s' % self.ulen)
        ua[0] = self.ucs_value*self.ulen
        self.content_check(ua, ua[0], 4*self.ulen*2)
        ua[1] = self.ucs_value*self.ulen
        self.content_check(ua, ua[1], 4*self.ulen*2)
    def test_valuesMD(self):
        ua = np.zeros((2, 3, 4), dtype='U%s' % self.ulen)
        ua[0, 0, 0] = self.ucs_value*self.ulen
        self.content_check(ua, ua[0, 0, 0], 4*self.ulen*2*3*4)
        ua[-1, -1, -1] = self.ucs_value*self.ulen
        self.content_check(ua, ua[-1, -1, -1], 4*self.ulen*2*3*4)
class TestAssignValues_1_UCS2(AssignValues):
    """Check the assignment of valued arrays (size 1, UCS2 values)"""
    ulen = 1
    ucs_value = ucs2_value
class TestAssignValues_1_UCS4(AssignValues):
    """Check the assignment of valued arrays (size 1, UCS4 values)"""
    ulen = 1
    ucs_value = ucs4_value
class TestAssignValues_2_UCS2(AssignValues):
    """Check the assignment of valued arrays (size 2, UCS2 values)"""
    ulen = 2
    ucs_value = ucs2_value
class TestAssignValues_2_UCS4(AssignValues):
    """Check the assignment of valued arrays (size 2, UCS4 values)"""
    ulen = 2
    ucs_value = ucs4_value
class TestAssignValues_1009_UCS2(AssignValues):
    """Check the assignment of valued arrays (size 1009, UCS2 values)"""
    ulen = 1009
    ucs_value = ucs2_value
class TestAssignValues_1009_UCS4(AssignValues):
    """Check the assignment of valued arrays (size 1009, UCS4 values)"""
    ulen = 1009
    ucs_value = ucs4_value
class ByteorderValues:
    """Check the byteorder of unicode arrays in round-trip conversions"""
    def test_values0D(self):
        ua = np.array(self.ucs_value*self.ulen, dtype='U%s' % self.ulen)
        ua2 = ua.newbyteorder()
        assert_(ua[:] != ua2[:])
        ua3 = ua2.newbyteorder()

```

```

167: (8)             assert_equal(ua, ua3)
168: (4)             def test_valuesSD(self):
169: (8)                 ua = np.array([self.ucs_value*self.ulen]*2, dtype='U%s' % self.ulen)
170: (8)                 ua2 = ua.newbyteorder()
171: (8)                 assert_(ua != ua2).all()
172: (8)                 assert_(ua[-1] != ua2[-1])
173: (8)                 ua3 = ua2.newbyteorder()
174: (8)                 assert_equal(ua, ua3)
175: (4)             def test_valuesMD(self):
176: (8)                 ua = np.array([[self.ucs_value*self.ulen]*2]*3)*4,
177: (22)                           dtype='U%s' % self.ulen)
178: (8)                 ua2 = ua.newbyteorder()
179: (8)                 assert_(ua != ua2).all()
180: (8)                 assert_(ua[-1, -1, -1] != ua2[-1, -1, -1])
181: (8)                 ua3 = ua2.newbyteorder()
182: (8)                 assert_equal(ua, ua3)
183: (4)             def test_values_cast(self):
184: (8)                 test1 = np.array([self.ucs_value*self.ulen]*2, dtype='U%s' %
self.ulen)
185: (8)                 test2 = np.repeat(test1, 2)[::2]
186: (8)                 for ua in (test1, test2):
187: (12)                     ua2 = ua.astype(dtype=ua.dtype.newbyteorder())
188: (12)                     assert_(ua == ua2).all()
189: (12)                     assert_(ua[-1] == ua2[-1])
190: (12)                     ua3 = ua2.astype(dtype=ua.dtype)
191: (12)                     assert_equal(ua, ua3)
192: (4)             def test_values_updowncast(self):
193: (8)                 test1 = np.array([self.ucs_value*self.ulen]*2, dtype='U%s' %
self.ulen)
194: (8)                 test2 = np.repeat(test1, 2)[::2]
195: (8)                 for ua in (test1, test2):
196: (12)                     longer_type = np.dtype('U%s' % (self.ulen+1)).newbyteorder()
197: (12)                     ua2 = ua.astype(dtype=longer_type)
198: (12)                     assert_(ua == ua2).all()
199: (12)                     assert_(ua[-1] == ua2[-1])
200: (12)                     ua3 = ua2.astype(dtype=ua.dtype)
201: (12)                     assert_equal(ua, ua3)
202: (0)             class TestByteorder_1_UCS2(ByteorderValues):
203: (4)                 """Check the byteorder in unicode (size 1, UCS2 values)"""
204: (4)                 ulen = 1
205: (4)                 ucs_value = ucs2_value
206: (0)             class TestByteorder_1_UCS4(ByteorderValues):
207: (4)                 """Check the byteorder in unicode (size 1, UCS4 values)"""
208: (4)                 ulen = 1
209: (4)                 ucs_value = ucs4_value
210: (0)             class TestByteorder_2_UCS2(ByteorderValues):
211: (4)                 """Check the byteorder in unicode (size 2, UCS2 values)"""
212: (4)                 ulen = 2
213: (4)                 ucs_value = ucs2_value
214: (0)             class TestByteorder_2_UCS4(ByteorderValues):
215: (4)                 """Check the byteorder in unicode (size 2, UCS4 values)"""
216: (4)                 ulen = 2
217: (4)                 ucs_value = ucs4_value
218: (0)             class TestByteorder_1009_UCS2(ByteorderValues):
219: (4)                 """Check the byteorder in unicode (size 1009, UCS2 values)"""
220: (4)                 ulen = 1009
221: (4)                 ucs_value = ucs2_value
222: (0)             class TestByteorder_1009_UCS4(ByteorderValues):
223: (4)                 """Check the byteorder in unicode (size 1009, UCS4 values)"""
224: (4)                 ulen = 1009
225: (4)                 ucs_value = ucs4_value
-----
```

## File 134 - \_locales.py:

```

1: (0)             """Provide class for testing in French locale
2: (0)             """
3: (0)             import sys
```

```

4: (0)         import locale
5: (0)         import pytest
6: (0)         __ALL__ = ['CommaDecimalPointLocale']
7: (0)         def find_comma_decimal_point_locale():
8: (4)             """See if platform has a decimal point as comma locale.
9: (4)             Find a locale that uses a comma instead of a period as the
10: (4)                decimal point.
11: (4)                Returns
12: (4)                -----
13: (4)                old_locale: str
14: (8)                    Locale when the function was called.
15: (4)                new_locale: {str, None}
16: (8)                    First French locale found, None if none found.
17: (4)
18: (4)            if sys.platform == 'win32':
19: (8)                locales = ['FRENCH']
20: (4)            else:
21: (8)                locales = ['fr_FR', 'fr_FR.UTF-8', 'fi_FI', 'fi_FI.UTF-8']
22: (4)            old_locale = locale.getlocale(locale.LC_NUMERIC)
23: (4)            new_locale = None
24: (4)            try:
25: (8)                for loc in locales:
26: (12)                    try:
27: (16)                        locale.setlocale(locale.LC_NUMERIC, loc)
28: (16)                        new_locale = loc
29: (16)                        break
30: (12)                    except locale.Error:
31: (16)                        pass
32: (4)            finally:
33: (8)                locale.setlocale(locale.LC_NUMERIC, locale=old_locale)
34: (4)            return old_locale, new_locale
35: (0)        class CommaDecimalPointLocale:
36: (4)            """Sets LC_NUMERIC to a locale with comma as decimal point.
37: (4)            Classes derived from this class have setup and teardown methods that run
38: (4)            tests with locale.LC_NUMERIC set to a locale where commas (',') are used
as
39: (4)            the decimal point instead of periods ('.'). On exit the locale is restored
40: (4)            to the initial locale. It also serves as context manager with the same
41: (4)            effect. If no such locale is available, the test is skipped.
42: (4)            .. versionadded:: 1.15.0
43: (4)
44: (4)            (cur_locale, tst_locale) = find_comma_decimal_point_locale()
45: (4)            def setup_method(self):
46: (8)                if self.tst_locale is None:
47: (12)                    pytest.skip("No French locale available")
48: (8)                    locale.setlocale(locale.LC_NUMERIC, locale=self.tst_locale)
49: (4)            def teardown_method(self):
50: (8)                locale.setlocale(locale.LC_NUMERIC, locale=self.cur_locale)
51: (4)            def __enter__(self):
52: (8)                if self.tst_locale is None:
53: (12)                    pytest.skip("No French locale available")
54: (8)                    locale.setlocale(locale.LC_NUMERIC, locale=self.tst_locale)
55: (4)            def __exit__(self, type, value, traceback):
56: (8)                locale.setlocale(locale.LC_NUMERIC, locale=self.cur_locale)

-----

```

## File 135 - test\_exceptions.py:

```

1: (0)         """
2: (0)             Tests of the .exceptions module. Primarily for exercising the __str__
methods.
3: (0)             """
4: (0)             import pickle
5: (0)             import pytest
6: (0)             import numpy as np
7: (0)             _ArrayMemoryError = np.core._exceptions._ArrayMemoryError
8: (0)             _UFuncNoLoopError = np.core._exceptions._UFuncNoLoopError
9: (0)             class TestArrayMemoryError:

```

```

10: (4)          def test_pickling(self):
11: (8)            """ Test that _ArrayMemoryError can be pickled """
12: (8)            error = _ArrayMemoryError((1023,), np.dtype(np.uint8))
13: (8)            res = pickle.loads(pickle.dumps(error))
14: (8)            assert res._total_size == error._total_size
15: (4)          def test_str(self):
16: (8)            e = _ArrayMemoryError((1023,), np.dtype(np.uint8))
17: (8)            str(e) # not crashing is enough
18: (4)          def test_size_to_string(self):
19: (8)            """ Test e._size_to_string """
20: (8)            f = _ArrayMemoryError._size_to_string
21: (8)            Ki = 1024
22: (8)            assert f(0) == '0 bytes'
23: (8)            assert f(1) == '1 bytes'
24: (8)            assert f(1023) == '1023 bytes'
25: (8)            assert f(Ki) == '1.00 KiB'
26: (8)            assert f(Ki+1) == '1.00 KiB'
27: (8)            assert f(10*Ki) == '10.0 KiB'
28: (8)            assert f(int(999.4*Ki)) == '999. KiB'
29: (8)            assert f(int(1023.4*Ki)) == '1023. KiB'
30: (8)            assert f(int(1023.5*Ki)) == '1.00 MiB'
31: (8)            assert f(Ki*Ki) == '1.00 MiB'
32: (8)            assert f(int(Ki*Ki*Ki*Ki*Ki*0.9999)) == '1.00 GiB'
33: (8)            assert f(Ki*Ki*Ki*Ki*Ki*Ki) == '1.00 EiB'
34: (8)            assert f(Ki*Ki*Ki*Ki*Ki*Ki*123456) == '123456. EiB'
35: (4)          def test_total_size(self):
36: (8)            """ Test e._total_size """
37: (8)            e = _ArrayMemoryError((1,), np.dtype(np.uint8))
38: (8)            assert e._total_size == 1
39: (8)            e = _ArrayMemoryError((2, 4), np.dtype((np.uint64, 16)))
40: (8)            assert e._total_size == 1024
41: (0)          class TestUFuncNoLoopError:
42: (4)            def test_pickling(self):
43: (8)              """ Test that _UFuncNoLoopError can be pickled """
44: (8)              assert isinstance(pickle.dumps(_UFuncNoLoopError), bytes)
45: (0)          @pytest.mark.parametrize("args", [
46: (4)            (2, 1, None),
47: (4)            (2, 1, "test_prefix"),
48: (4)            ("test message",),
49: (0)
50: (0)        ])
51: (4)        class TestAxisError:
52: (8)          def test_attr(self, args):
53: (8)            """Validate attribute types."""
54: (8)            exc = np.AxisError(*args)
55: (12)           if len(args) == 1:
56: (12)             assert exc.axis is None
57: (12)             assert exc.ndim is None
58: (12)           else:
59: (12)             axis, ndim, *_ = args
60: (12)             assert exc.axis == axis
61: (12)             assert exc.ndim == ndim
61: (4)          def test_pickling(self, args):
62: (8)            """Test that `AxisError` can be pickled."""
63: (8)            exc = np.AxisError(*args)
64: (8)            exc2 = pickle.loads(pickle.dumps(exc))
65: (8)            assert type(exc) is type(exc2)
66: (8)            for name in ("axis", "ndim", "args"):
67: (12)              attr1 = getattr(exc, name)
68: (12)              attr2 = getattr(exc2, name)
69: (12)              assert attr1 == attr2, name

```

-----  
File 136 - \_\_init\_\_.py:

```
1: (0)
```

## File 137 - setup.py:

```

1: (0)      """
2: (0)      Provide python-space access to the functions exposed in numpy/__init__.pxd
3: (0)      for testing.
4: (0)
5: (0)      import numpy as np
6: (0)      from distutils.core import setup
7: (0)      from Cython.Build import cythonize
8: (0)      from setuptools.extension import Extension
9: (0)      import os
10: (0)     macros = [("NPY_NO_DEPRECATED_API", 0)]
11: (0)     checks = Extension(
12: (4)         "checks",
13: (4)         sources=[os.path.join('.', "checks.pyx")],
14: (4)         include_dirs=[np.get_include()],
15: (4)         define_macros=macros,
16: (0)
17: (0)     )
18: (0)     extensions = [checks]
19: (4)     setup(
20: (0)         ext_modules=cythonize(extensions)
)
-----
```

## File 138 - setup.py:

```

1: (0)      """
2: (0)      Build an example package using the limited Python C API.
3: (0)
4: (0)
5: (0)      import numpy as np
6: (0)      from setuptools import setup, Extension
7: (0)      import os
8: (0)      macros = [("NPY_NO_DEPRECATED_API", 0), ("Py_LIMITED_API", "0x03060000")]
9: (0)      limited_api = Extension(
10: (4)        "limited_api",
11: (4)        sources=[os.path.join('.', "limited_api.c")],
12: (4)        include_dirs=[np.get_include()],
13: (4)        define_macros=macros,
14: (0)
15: (0)      )
16: (4)      extensions = [limited_api]
17: (0)      setup(
18: (0)          ext_modules=extensions
)
-----
```

## File 139 - constants.py:

```

1: (0)      """
2: (0)      =====
3: (0)      Constants
4: (0)      =====
5: (0)      .. currentmodule:: numpy
6: (0)      NumPy includes several constants:
7: (0)      %(constant_list)s
8: (0)
9: (0)      import re
10: (0)      import textwrap
11: (0)      constants = []
12: (0)      def add_newdoc(module, name, doc):
13: (4)          constants.append((name, doc))
14: (0)      add_newdoc('numpy', 'pi',
15: (4)          """
16: (4)          ``pi = 3.1415926535897932384626433...``
17: (4)          References
18: (4)          -----
19: (4)          https://en.wikipedia.org/wiki/Pi
20: (4)          """")
-----
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```
21: (0)
22: (4)    add_newdoc('numpy', 'e',
23: (4)        """
24: (4)            Euler's constant, base of natural logarithms, Napier's constant.
25: (4)            ``e = 2.71828182845904523536028747135266249775724709369995...``
26: (4)            See Also
27: (4)                -----
28: (4)                    exp : Exponential function
29: (4)                    log : Natural logarithm
30: (4)                    References
31: (4)                -----
32: (4)                https://en.wikipedia.org/wiki/E_%28mathematical_constant%29
33: (0)    add_newdoc('numpy', 'euler_gamma',
34: (4)        """
35: (4)            ``Î³ = 0.5772156649015328606065120900824024310421...``
36: (4)            References
37: (4)                -----
38: (4)                https://en.wikipedia.org/wiki/Euler-Mascheroni_constant
39: (4)            """
40: (0)    add_newdoc('numpy', 'inf',
41: (4)        """
42: (4)            IEEE 754 floating point representation of (positive) infinity.
43: (4)            Returns
44: (4)                -----
45: (4)                y : float
46: (8)                    A floating point representation of positive infinity.
47: (4)            See Also
48: (4)                -----
49: (4)                isnan : Shows which elements are positive or negative infinity
50: (4)                isposinf : Shows which elements are positive infinity
51: (4)                isneginf : Shows which elements are negative infinity
52: (4)                isnan : Shows which elements are Not a Number
53: (4)                isfinite : Shows which elements are finite (not one of Not a Number,
54: (4)                    positive infinity and negative infinity)
55: (4)                Notes
56: (4)                -----
57: (4)                NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
58: (4)                (IEEE 754). This means that Not a Number is not equivalent to infinity.
59: (4)                Also that positive infinity is not equivalent to negative infinity. But
60: (4)                infinity is equivalent to positive infinity.
61: (4)                `Inf`, `Infinity`, `PINF` and `infty` are aliases for `inf`.
62: (4)                Examples
63: (4)                -----
64: (4)                >>> np.inf
65: (4)                    inf
66: (4)                >>> np.array([1]) / 0.
67: (4)                    array([ Inf])
68: (4)            """
69: (0)    add_newdoc('numpy', 'nan',
70: (4)        """
71: (4)            IEEE 754 floating point representation of Not a Number (NaN).
72: (4)            Returns
73: (4)                -----
74: (4)                y : A floating point representation of Not a Number.
75: (4)            See Also
76: (4)                -----
77: (4)                isnan : Shows which elements are Not a Number.
78: (4)                isfinite : Shows which elements are finite (not one of
79: (4)                    Not a Number, positive infinity and negative infinity)
80: (4)                Notes
81: (4)                -----
82: (4)                NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
83: (4)                (IEEE 754). This means that Not a Number is not equivalent to infinity.
84: (4)                `NaN` and `NAN` are aliases of `nan`.
85: (4)                Examples
86: (4)                -----
87: (4)                >>> np.nan
88: (4)                    nan
89: (4)                >>> np.log(-1)
```

```

90: (4)             nan
91: (4)             >>> np.log([-1, 1, 2])
92: (4)             array([      NaN,  0.          ,  0.69314718])
93: (4)             """
94: (0)             add_newdoc('numpy', 'newaxis',
95: (4)             """
96: (4)             A convenient alias for None, useful for indexing arrays.
97: (4)             Examples
98: (4)             -----
99: (4)             >>> newaxis is None
100: (4)            True
101: (4)            >>> x = np.arange(3)
102: (4)            >>> x
103: (4)            array([0, 1, 2])
104: (4)            >>> x[:, newaxis]
105: (4)            array([[0],
106: (4)              [1],
107: (4)              [2]])
108: (4)            >>> x[:, newaxis, newaxis]
109: (4)            array([[[0]],
110: (4)              [[1]],
111: (4)              [[2]]])
112: (4)            >>> x[:, newaxis] * x
113: (4)            array([[0, 0, 0],
114: (4)              [0, 1, 2],
115: (4)              [0, 2, 4]])
116: (4)            Outer product, same as ``outer(x, y)``:
117: (4)            >>> y = np.arange(3, 6)
118: (4)            >>> x[:, newaxis] * y
119: (4)            array([[ 0,  0,  0],
120: (4)              [ 3,  4,  5],
121: (4)              [ 6,  8, 10]])
122: (4)            ``x[newaxis, :]`` is equivalent to ``x[newaxis]`` and ``x[None]``:
123: (4)            >>> x[newaxis, :].shape
124: (4)            (1, 3)
125: (4)            >>> x[newaxis].shape
126: (4)            (1, 3)
127: (4)            >>> x[None].shape
128: (4)            (1, 3)
129: (4)            >>> x[:, newaxis].shape
130: (4)            (3, 1)
131: (4)            """
132: (0)            add_newdoc('numpy', 'NZERO',
133: (4)            """
134: (4)            IEEE 754 floating point representation of negative zero.
135: (4)            Returns
136: (4)            -----
137: (4)            y : float
138: (8)            A floating point representation of negative zero.
139: (4)            See Also
140: (4)            -----
141: (4)            PZERO : Defines positive zero.
142: (4)            isnf : Shows which elements are positive or negative infinity.
143: (4)            isposinf : Shows which elements are positive infinity.
144: (4)            isneginf : Shows which elements are negative infinity.
145: (4)            isnan : Shows which elements are Not a Number.
146: (4)            isfinite : Shows which elements are finite - not one of
147: (15)              Not a Number, positive infinity and negative infinity.
148: (4)            Notes
149: (4)            -----
150: (4)            NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
151: (4)            (IEEE 754). Negative zero is considered to be a finite number.
152: (4)            Examples
153: (4)            -----
154: (4)            >>> np.NZERO
155: (4)            -0.0
156: (4)            >>> np.PZERO
157: (4)            0.0
158: (4)            >>> np.isfinite([np.NZERO])

```

```

159: (4)           array([ True])
160: (4)           >>> np.isnan([np.NZERO])
161: (4)           array([False])
162: (4)           >>> np.isinf([np.NZERO])
163: (4)           array([False])
164: (4)           """
165: (0)           add_newdoc('numpy', 'PZERO',
166: (4)           """
167: (4)           IEEE 754 floating point representation of positive zero.
168: (4)           Returns
169: (4)           -----
170: (4)           y : float
171: (8)           A floating point representation of positive zero.
172: (4)           See Also
173: (4)           -----
174: (4)           NZERO : Defines negative zero.
175: (4)           isnan : Shows which elements are positive or negative infinity.
176: (4)           isposinf : Shows which elements are positive infinity.
177: (4)           isneginf : Shows which elements are negative infinity.
178: (4)           isinf : Shows which elements are finite - not one of
179: (4)           Not a Number, positive infinity and negative infinity.
180: (15)
181: (4)           Notes
182: (4)           -----
183: (4)           NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
184: (4)           (IEEE 754). Positive zero is considered to be a finite number.
185: (4)           Examples
186: (4)           -----
187: (4)           >>> np.PZERO
188: (4)           0.0
189: (4)           >>> np.NZERO
190: (4)           -0.0
191: (4)           >>> np.isfinite([np.PZERO])
192: (4)           array([ True])
193: (4)           >>> np.isnan([np.PZERO])
194: (4)           array([False])
195: (4)           >>> np.isinf([np.PZERO])
196: (4)           array([False])
197: (4)           """
198: (0)           add_newdoc('numpy', 'NAN',
199: (4)           """
200: (4)           IEEE 754 floating point representation of Not a Number (NaN).
201: (4)           `NaN` and `NAN` are equivalent definitions of `nan`. Please use
202: (4)           `nan` instead of `NAN`.
203: (4)           See Also
204: (4)           -----
205: (4)           nan
206: (4)           """
207: (0)           add_newdoc('numpy', 'NaN',
208: (4)           """
209: (4)           IEEE 754 floating point representation of Not a Number (NaN).
210: (4)           `NaN` and `NAN` are equivalent definitions of `nan`. Please use
211: (4)           `nan` instead of `NAN`.
212: (4)           See Also
213: (4)           -----
214: (4)           nan
215: (4)           """
216: (0)           add_newdoc('numpy', 'NINF',
217: (4)           """
218: (4)           IEEE 754 floating point representation of negative infinity.
219: (4)           Returns
220: (4)           -----
221: (4)           y : float
222: (8)           A floating point representation of negative infinity.
223: (4)           See Also
224: (4)           -----
225: (4)           isinf : Shows which elements are positive or negative infinity
226: (4)           isposinf : Shows which elements are positive infinity
227: (4)           isneginf : Shows which elements are negative infinity

```

```

228: (4)           isnan : Shows which elements are Not a Number
229: (4)           isfinite : Shows which elements are finite (not one of Not a Number,
230: (4)           positive infinity and negative infinity)
231: (4)           Notes
232: (4)           -----
233: (4)           NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
234: (4)           (IEEE 754). This means that Not a Number is not equivalent to infinity.
235: (4)           Also that positive infinity is not equivalent to negative infinity. But
236: (4)           infinity is equivalent to positive infinity.
237: (4)           Examples
238: (4)           -----
239: (4)           >>> np.NINF
240: (4)           -inf
241: (4)           >>> np.log(0)
242: (4)           -inf
243: (4)           """
244: (0)           add_newdoc('numpy', 'PINF',
245: (4)           """
246: (4)           IEEE 754 floating point representation of (positive) infinity.
247: (4)           Use `inf` because `Inf`, `Infinity`, `PINF` and `infty` are aliases for
248: (4)           `inf`. For more details, see `inf`.
249: (4)           See Also
250: (4)           -----
251: (4)           inf
252: (4)           """
253: (0)           add_newdoc('numpy', 'infty',
254: (4)           """
255: (4)           IEEE 754 floating point representation of (positive) infinity.
256: (4)           Use `inf` because `Inf`, `Infinity`, `PINF` and `infty` are aliases for
257: (4)           `inf`. For more details, see `inf`.
258: (4)           See Also
259: (4)           -----
260: (4)           inf
261: (4)           """
262: (0)           add_newdoc('numpy', 'Inf',
263: (4)           """
264: (4)           IEEE 754 floating point representation of (positive) infinity.
265: (4)           Use `inf` because `Inf`, `Infinity`, `PINF` and `infty` are aliases for
266: (4)           `inf`. For more details, see `inf`.
267: (4)           See Also
268: (4)           -----
269: (4)           inf
270: (4)           """
271: (0)           add_newdoc('numpy', 'Infinity',
272: (4)           """
273: (4)           IEEE 754 floating point representation of (positive) infinity.
274: (4)           Use `inf` because `Inf`, `Infinity`, `PINF` and `infty` are aliases for
275: (4)           `inf`. For more details, see `inf`.
276: (4)           See Also
277: (4)           -----
278: (4)           inf
279: (4)           """
280: (0)           if __doc__:
281: (4)             constants_str = []
282: (4)             constants.sort()
283: (4)             for name, doc in constants:
284: (8)               s = textwrap.dedent(doc).replace("\n", "\n      ")
285: (8)               lines = s.split("\n")
286: (8)               new_lines = []
287: (8)               for line in lines:
288: (12)                 m = re.match(r'^(\s+)[-]+\s*$', line)
289: (12)                 if m and new_lines:
290: (16)                   prev = textwrap.dedent(new_lines.pop())
291: (16)                   new_lines.append('%s.. rubric:: %s' % (m.group(1), prev))
292: (16)                   new_lines.append('')
293: (12)                 else:
294: (16)                   new_lines.append(line)
295: (8)               s = "\n".join(new_lines)
296: (8)               constants_str.append("'''.. data:: %s\n      %s''' % (name, s))
```

```

297: (4)         constants_str = "\n".join(constants_str)
298: (4)         __doc__ = __doc__ % dict(constant_list=constants_str)
299: (4)         del constants_str, name, doc
300: (4)         del line, lines, new_lines, m, s, prev
301: (0)         del constants, add_newdoc
-----
```

## File 140 - ufuncs.py:

```

1: (0)         """
2: (0)         =====
3: (0)         Universal Functions
4: (0)         =====
5: (0)         Ufuncs are, generally speaking, mathematical functions or operations that are
6: (0)         applied element-by-element to the contents of an array. That is, the result
7: (0)         in each output array element only depends on the value in the corresponding
8: (0)         input array (or arrays) and on no other array elements. NumPy comes with a
9: (0)         large suite of ufuncs, and scipy extends that suite substantially. The
simplest
10: (0)        example is the addition operator: :::
11: (1)        >>> np.array([0,2,3,4]) + np.array([1,1,-1,2])
12: (1)        array([1, 3, 2, 6])
13: (0)        The ufunc module lists all the available ufuncs in numpy. Documentation on
14: (0)        the specific ufuncs may be found in those modules. This documentation is
15: (0)        intended to address the more general aspects of ufuncs common to most of
16: (0)        them. All of the ufuncs that make use of Python operators (e.g., +, -, etc.)
17: (0)        have equivalent functions defined (e.g. add() for +)
18: (0)        Type coercion
19: (0)        =====
20: (0)        What happens when a binary operator (e.g., +, -, \*, /, etc) deals with arrays
of
21: (0)        two different types? What is the type of the result? Typically, the result is
22: (0)        the higher of the two types. For example: :::
23: (1)        float32 + float64 -> float64
24: (1)        int8 + int32 -> int32
25: (1)        int16 + float32 -> float32
26: (1)        float32 + complex64 -> complex64
27: (0)        There are some less obvious cases generally involving mixes of types
28: (0)        (e.g. uints, ints and floats) where equal bit sizes for each are not
29: (0)        capable of saving all the information in a different type of equivalent
30: (0)        bit size. Some examples are int32 vs float32 or uint32 vs int32.
31: (0)        Generally, the result is the higher type of larger size than both
32: (0)        (if available). So: :::
33: (1)        int32 + float32 -> float64
34: (1)        uint32 + int32 -> int64
35: (0)        Finally, the type coercion behavior when expressions involve Python
36: (0)        scalars is different than that seen for arrays. Since Python has a
37: (0)        limited number of types, combining a Python int with a dtype=np.int8
38: (0)        array does not coerce to the higher type but instead, the type of the
39: (0)        array prevails. So the rules for Python scalars combined with arrays is
40: (0)        that the result will be that of the array equivalent the Python scalar
41: (0)        if the Python scalar is of a higher 'kind' than the array (e.g., float
42: (0)        vs. int), otherwise the resultant type will be that of the array.
43: (0)        For example: :::
44: (2)        Python int + int8 -> int8
45: (2)        Python float + int8 -> float64
46: (0)        ufunc methods
47: (0)        =====
48: (0)        Binary ufuncs support 4 methods.
49: (0)        **.reduce(arr)** applies the binary operator to elements of the array in
50: (2)        sequence. For example: :::
51: (1)        >>> np.add.reduce(np.arange(10)) # adds all elements of array
52: (1)        45
53: (0)        For multidimensional arrays, the first dimension is reduced by default: :::
54: (1)        >>> np.add.reduce(np.arange(10).reshape(2,5))
55: (5)        array([ 5,  7,  9, 11, 13])
56: (0)        The axis keyword can be used to specify different axes to reduce: :::
57: (1)        >>> np.add.reduce(np.arange(10).reshape(2,5),axis=1)
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

58: (1) array([10, 35])
59: (0) **.accumulate(arr)** applies the binary operator and generates an
60: (0) equivalently shaped array that includes the accumulated amount for each
61: (0) element of the array. A couple examples: :::
62: (1) >>> np.add.accumulate(np.arange(10))
63: (1) array([ 0,  1,  3,  6, 10, 15, 21, 28, 36, 45])
64: (1) >>> np.multiply.accumulate(np.arange(1,9))
65: (1) array([ 1,   2,   6,  24, 120, 720, 5040, 40320])
66: (0) The behavior for multidimensional arrays is the same as for .reduce(), as is the use of the axis keyword).
67: (0)
68: (0) **.reduceat(arr,indices)** allows one to apply reduce to selected parts
69: (2) of an array. It is a difficult method to understand. See the documentation
70: (2) at:
71: (0) **.outer(arr1,arr2)** generates an outer operation on the two arrays arr1 and
72: (2) arr2. It will work on multidimensional arrays (the shape of the result is
73: (2) the concatenation of the two input shapes.:::
74: (1) >>> np.multiply.outer(np.arange(3),np.arange(4))
75: (1) array([[0, 0, 0, 0],
76: (8)         [0, 1, 2, 3],
77: (8)         [0, 2, 4, 6]])
78: (0) Output arguments
79: (0) =====
80: (0) All ufuncs accept an optional output array. The array must be of the expected
81: (0) output shape. Beware that if the type of the output array is of a different
82: (0) (and lower) type than the output result, the results may be silently truncated
83: (0) or otherwise corrupted in the downcast to the lower type. This usage is useful
84: (0) when one wants to avoid creating large temporary arrays and instead allows one
85: (0) to reuse the same array memory repeatedly (at the expense of not being able to
86: (0) use more convenient operator notation in expressions). Note that when the
87: (0) output argument is used, the ufunc still returns a reference to the result.
88: (1) >>> x = np.arange(2)
89: (1) >>> np.add(np.arange(2),np.arange(2.),x)
90: (1) array([0, 2])
91: (1) >>> x
92: (1) array([0, 2])
93: (0) and & or as ufuncs
94: (0) =====
95: (0) Invariably people try to use the python 'and' and 'or' as logical operators
96: (0) (and quite understandably). But these operators do not behave as normal
97: (0) operators since Python treats these quite differently. They cannot be
98: (0) overloaded with array equivalents. Thus using 'and' or 'or' with an array
99: (0) results in an error. There are two alternatives:
100: (1) 1) use the ufunc functions logical_and() and logical_or().
101: (1) 2) use the bitwise operators & and \|. The drawback of these is that if
102: (4) the arguments to these operators are not boolean arrays, the result is
103: (4) likely incorrect. On the other hand, most usages of logical_and and
104: (4) logical_or are with boolean arrays. As long as one is careful, this is
105: (4) a convenient way to apply these operators.
106: (0)

```

---

File 141 - \_\_init\_\_.py:

```

1: (0) import os
2: (0) ref_dir = os.path.join(os.path.dirname(__file__))
3: (0) __all__ = sorted(f[:-3] for f in os.listdir(ref_dir) if f.endswith('.py') and
4: (11) not f.startswith('__'))
5: (0) for f in __all__:
6: (4)     __import__(__name__ + '.' + f)
7: (0) del f, ref_dir
8: (0) __doc__ = """\
9: (0) Topical documentation
10: (0) =====
11: (0) The following topics are available:
12: (0) %s
13: (0) You can view them by
14: (0) >>> help(np.doc.TOPIC) #doctest: +SKIP
15: (0) """% '\n- '.join([''] + __all__)

```

16: (0)

`__all__.extend(['__doc__'])`-----  
File 142 - auxfuncs.py:

```

1: (0)
2: (0)    """
3: (0)    Auxiliary functions for f2py2e.
4: (0)    Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
5: (0)    Copyright 2011 -- present NumPy Developers.
6: (0)    Permission to use, modify, and distribute this software is given under the
7: (0)    terms of the NumPy (BSD style) LICENSE.
8: (0)    NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
9: (0)    """
10: (0)   import pprint
11: (0)   import sys
12: (0)   import re
13: (0)   import types
14: (0)   from functools import reduce
15: (0)   from copy import deepcopy
16: (0)   from . import __version__
17: (0)   from . import cfuncs
18: (0)   __all__ = [
19: (4)     'applyrules', 'debugcapi', 'dictappend', 'errmess', 'gentitle',
20: (4)     'getargs2', 'getcallprotoargument', 'getcallstatement',
21: (4)     'getfortranname', 'getpymethoddef', 'getrestdoc', 'getusercode',
22: (4)     'getusercode1', 'getdimension', 'hasbody', 'hascalldstatement',
23: (4)     'hascommon',
24: (4)     'hasexternals', 'hasinitvalue', 'hasnote', 'hasresultnote',
25: (4)     'isallocatable', 'isarray', 'isarrayofstrings',
26: (4)     'ischaracter', 'ischaracterarray', 'ischaracter_or_characterarray',
27: (4)     'iscomplex',
28: (4)     'iscomplexarray', 'iscomplexfunction', 'iscomplexfunction_warn',
29: (4)     'isdouble', 'isdummyroutine', 'isexternal', 'isfunction',
30: (4)     'isfunction_wrap', 'isint1', 'isint1array', 'isinteger', 'isintent_aux',
31: (4)     'isintent_c', 'isintent_callback', 'isintent_copy', 'isintent_dict',
32: (4)     'isintent_hide', 'isintent_in', 'isintent_inout', 'isintent_inplace',
33: (4)     'isintent_nothide', 'isintent_out', 'isintent_overwrite', 'islogical',
34: (4)     'islogicalfunction', 'islong_complex', 'islong_double',
35: (4)     'islong_doublefunction', 'islong_long', 'islong_longfunction',
36: (4)     'ismodule', 'ismoduleroutine', 'isoptional', 'isprivate', 'isrequired',
37: (4)     'isroutine', 'isscalar', 'issigned_long_longarray', 'isstring',
38: (4)     'isstringarray', 'isstring_or_stringarray', 'isstringfunction',
39: (4)     'issubroutine', 'get_f2py_modulename',
40: (4)     'issubroutine_wrap', 'isthreadsafe', 'isunsigned', 'isunsigned_char',
41: (4)     'isunsigned_chararray', 'isunsigned_long_long',
42: (4)     'isunsigned_long_longarray', 'isunsigned_short',
43: (4)     'isunsigned_shortarray', 'l_and', 'l_not', 'l_or', 'outmess',
44: (4)     'replace', 'show', 'stripcomma', 'throw_error', 'isattr_value',
45: (4)     'getuseblocks', 'process_f2cmap_dict'
46: (0)   ]
47: (0)   f2py_version = __version__.version
48: (0)   errmess = sys.stderr.write
49: (0)   show = pprint.pprint
50: (0)   options = {}
51: (0)   debugoptions = []
52: (0)   wrapfuncs = 1
53: (0)   def outmess(t):
54: (4)     if options.get('verbose', 1):
55: (8)       sys.stdout.write(t)
56: (0)   def debugcapi(var):
57: (4)     return 'capi' in debugoptions
58: (0)   def _ischaracter(var):
59: (4)     return 'typespec' in var and var['typespec'] == 'character' and \
60: (11)       not isexternal(var)
61: (0)   def _isstring(var):
62: (4)     return 'typespec' in var and var['typespec'] == 'character' and \
63: (11)       not isexternal(var)
64: (0)   def ischaracter_or_characterarray(var):
```

```

63: (4)           return _ischaracter(var) and 'charselector' not in var
64: (0)
65: (4)           return ischaracter_or_characterarray(var) and not isarray(var)
66: (0)
67: (4)           return ischaracter_or_characterarray(var) and isarray(var)
68: (0)
69: (4)           return _ischaracter(var) and 'charselector' in var
70: (0)
71: (4)           return isstring_or_stringarray(var) and not isarray(var)
72: (0)
73: (4)           return isstring_or_stringarray(var) and isarray(var)
74: (0)
75: (4)           return isstringarray(var) and var['dimension'][-1] == '(*)'
76: (0)
77: (4)           return 'dimension' in var and not isexternal(var)
78: (0)
79: (4)           return not (isarray(var) or isstring(var) or isexternal(var))
80: (0)
81: (4)           return isscalar(var)
82: (11)          return isscalar(var) and \
83: (0)             var.get('typespec') in ['complex', 'double complex']
84: (4)           return isscalar(var) and var.get('typespec') == 'logical'
85: (0)
86: (4)           return isscalar(var) and var.get('typespec') == 'integer'
87: (0)
88: (4)           return isscalar(var) and var.get('typespec') == 'real'
89: (0)
90: (4)           def get_kind(var):
91: (8)             try:
92: (4)               return var['kindselector']['*']
93: (8)             except KeyError:
94: (12)               try:
95: (8)                 return var['kindselector']['kind']
96: (12)               except KeyError:
97: (0)                 pass
98: (4)           def isint1(var):
99: (8)             return var.get('typespec') == 'integer' \
100: (0)               and get_kind(var) == '1' and not isarray(var)
101: (4)           def islong_long(var):
102: (8)             if not isscalar(var):
103: (4)               return 0
104: (8)             if var.get('typespec') not in ['integer', 'logical']:
105: (4)               return 0
106: (0)             return get_kind(var) == '8'
107: (4)           def isunsigned_char(var):
108: (8)             if not isscalar(var):
109: (4)               return 0
110: (8)             if var.get('typespec') != 'integer':
111: (4)               return 0
112: (0)             return get_kind(var) == '-1'
113: (4)           def isunsigned_short(var):
114: (8)             if not isscalar(var):
115: (4)               return 0
116: (8)             if var.get('typespec') != 'integer':
117: (4)               return 0
118: (0)             return get_kind(var) == '-2'
119: (4)           def isunsigned(var):
120: (8)             if not isscalar(var):
121: (4)               return 0
122: (8)             if var.get('typespec') != 'integer':
123: (4)               return 0
124: (0)             return get_kind(var) == '-4'
125: (4)           def isunsigned_long_long(var):
126: (8)             if not isscalar(var):
127: (4)               return 0
128: (8)             if var.get('typespec') != 'integer':
129: (4)               return 0
130: (0)             return get_kind(var) == '-8'
131: (4)           def isdouble(var):
132: (0)             if not isscalar(var):

```

```

132: (8)             return 0
133: (4)             if not var.get('typespec') == 'real':
134: (8)                 return 0
135: (4)             return get_kind(var) == '8'
136: (0)             def islong_double(var):
137: (4)                 if not isscalar(var):
138: (8)                     return 0
139: (4)                 if not var.get('typespec') == 'real':
140: (8)                     return 0
141: (4)                 return get_kind(var) == '16'
142: (0)             def islong_complex(var):
143: (4)                 if not iscomplex(var):
144: (8)                     return 0
145: (4)                 return get_kind(var) == '32'
146: (0)             def iscomplexarray(var):
147: (4)                 return isarray(var) and \
148: (11)                   var.get('typespec') in ['complex', 'double complex']
149: (0)             def isint1array(var):
150: (4)                 return isarray(var) and var.get('typespec') == 'integer' \
151: (8)                   and get_kind(var) == '1'
152: (0)             def isunsigned_chararray(var):
153: (4)                 return isarray(var) and var.get('typespec') in ['integer', 'logical']\
154: (8)                   and get_kind(var) == '-1'
155: (0)             def isunsigned_shortarray(var):
156: (4)                 return isarray(var) and var.get('typespec') in ['integer', 'logical']\
157: (8)                   and get_kind(var) == '-2'
158: (0)             def isunsignededarray(var):
159: (4)                 return isarray(var) and var.get('typespec') in ['integer', 'logical']\
160: (8)                   and get_kind(var) == '-4'
161: (0)             def isunsigned_long_longarray(var):
162: (4)                 return isarray(var) and var.get('typespec') in ['integer', 'logical']\
163: (8)                   and get_kind(var) == '-8'
164: (0)             def issigned_chararray(var):
165: (4)                 return isarray(var) and var.get('typespec') in ['integer', 'logical']\
166: (8)                   and get_kind(var) == '1'
167: (0)             def issigned_shortarray(var):
168: (4)                 return isarray(var) and var.get('typespec') in ['integer', 'logical']\
169: (8)                   and get_kind(var) == '2'
170: (0)             def issigned_array(var):
171: (4)                 return isarray(var) and var.get('typespec') in ['integer', 'logical']\
172: (8)                   and get_kind(var) == '4'
173: (0)             def issigned_long_longarray(var):
174: (4)                 return isarray(var) and var.get('typespec') in ['integer', 'logical']\
175: (8)                   and get_kind(var) == '8'
176: (0)             def isallocatable(var):
177: (4)                 return 'attrspec' in var and 'allocatable' in var['attrspec']
178: (0)             def ismutable(var):
179: (4)                 return not ('dimension' not in var or isstring(var))
180: (0)             def ismoduleroutine(rout):
181: (4)                 return 'modulename' in rout
182: (0)             def ismodule(rout):
183: (4)                 return 'block' in rout and 'module' == rout['block']
184: (0)             def isfunction(rout):
185: (4)                 return 'block' in rout and 'function' == rout['block']
186: (0)             def isfunction_wrap(rout):
187: (4)                 if isintent_c(rout):
188: (8)                     return 0
189: (4)                     return wrapfuncs and isfunction(rout) and (not isexternal(rout))
190: (0)             def issubroutine(rout):
191: (4)                 return 'block' in rout and 'subroutine' == rout['block']
192: (0)             def issubroutine_wrap(rout):
193: (4)                 if isintent_c(rout):
194: (8)                     return 0
195: (4)                     return issubroutine(rout) and hasassumedshape(rout)
196: (0)             def isattr_value(var):
197: (4)                 return 'value' in var.get('attrspec', [])
198: (0)             def hasassumedshape(rout):
199: (4)                 if rout.get('hasassumedshape'):
200: (8)                     return True

```

```

201: (4)
202: (8)
203: (12)
204: (16)
205: (16)
206: (4)
207: (0)
208: (4)
209: (0)
210: (4)
211: (0)
212: (4)
213: (8)
214: (4)
215: (8)
216: (4)
217: (8)
218: (4)
219: (8)
220: (4)
221: (0)
222: (4)
223: (8)
224: (4)
225: (8)
226: (4)
227: (8)
228: (4)
229: (8)
230: (4)
231: (0)
232: (4)
233: (8)
234: (4)
235: (8)
236: (4)
237: (8)
238: (4)
239: (8)
240: (4)
241: (0)
242: (4)
243: (8)
244: (4)
245: (8)
246: (4)
247: (8)
248: (4)
249: (8)
250: (4)
251: (0)
252: (4)
253: (8)
254: (4)
255: (8)
256: (8)
257: (8)
258: (8)
259: (8)
260: (4)
261: (8)
262: (4)
263: (0)
264: (4)
265: (8)
266: (4)
267: (8)
268: (4)
269: (8)

    for a in rout['args']:
        for d in rout['vars'].get(a, {}).get('dimension', []):
            if d == ':':
                rout['hasassumedshape'] = True
                return True
        return False
    def requiresf90wrapper(rout):
        return ismoduleroutine(rout) or hasassumedshape(rout)
    def isroutine(rout):
        return isfunction(rout) or issubroutine(rout)
    def islogicalfunction(rout):
        if not isfunction(rout):
            return 0
        if 'result' in rout:
            a = rout['result']
        else:
            a = rout['name']
        if a in rout['vars']:
            return islogical(rout['vars'][a])
        return 0
    def islong_longfunction(rout):
        if not isfunction(rout):
            return 0
        if 'result' in rout:
            a = rout['result']
        else:
            a = rout['name']
        if a in rout['vars']:
            return islong_long(rout['vars'][a])
        return 0
    def islong_doublefunction(rout):
        if not isfunction(rout):
            return 0
        if 'result' in rout:
            a = rout['result']
        else:
            a = rout['name']
        if a in rout['vars']:
            return islong_double(rout['vars'][a])
        return 0
    def iscomplexfunction(rout):
        if not isfunction(rout):
            return 0
        if 'result' in rout:
            a = rout['result']
        else:
            a = rout['name']
        if a in rout['vars']:
            return iscomplex(rout['vars'][a])
        return 0
    def iscomplexfunction_warn(rout):
        if iscomplexfunction(rout):
            outmess("""\n*****
                         Warning: code with a function returning complex value
                         may not work correctly with your Fortran compiler.
                         When using GNU gcc/g77 compilers, codes should work
                         correctly for callbacks with:
                         f2py -c -DF2PY_CB_RETURNCOMPLEX
*****\n""")
```

```

270: (4)           if a in rout['vars']:
271: (8)             return isstring(rout['vars'][a])
272: (4)             return 0
273: (0)           def hasexternals(rout):
274: (4)             return 'externals' in rout and rout['externals']
275: (0)           def isthreadsafe(rout):
276: (4)             return 'f2pyenhancements' in rout and \
277: (11)               'threadsafe' in rout['f2pyenhancements']
278: (0)           def hasvariables(rout):
279: (4)             return 'vars' in rout and rout['vars']
280: (0)           def isoptional(var):
281: (4)             return ('attrspec' in var and 'optional' in var['attrspec'] and
282: (12)               'required' not in var['attrspec']) and isintent_nothide(var)
283: (0)           def isexternal(var):
284: (4)             return 'attrspec' in var and 'external' in var['attrspec']
285: (0)           def getdimension(var):
286: (4)             dimpattern = r"\((.*?)\)"
287: (4)             if 'attrspec' in var.keys():
288: (8)               if any('dimension' in s for s in var['attrspec']):
289: (12)                 return [re.findall(dimpattern, v) for v in var['attrspec']][0]
290: (0)           def isrequired(var):
291: (4)             return not isoptional(var) and isintent_nothide(var)
292: (0)           def isintent_in(var):
293: (4)             if 'intent' not in var:
294: (8)               return 1
295: (4)             if 'hide' in var['intent']:
296: (8)               return 0
297: (4)             if 'inplace' in var['intent']:
298: (8)               return 0
299: (4)             if 'in' in var['intent']:
300: (8)               return 1
301: (4)             if 'out' in var['intent']:
302: (8)               return 0
303: (4)             if 'inout' in var['intent']:
304: (8)               return 0
305: (4)             if 'outin' in var['intent']:
306: (8)               return 0
307: (4)             return 1
308: (0)           def isintent inout(var):
309: (4)             return ('intent' in var and ('inout' in var['intent'] or
310: (12)               'outin' in var['intent']) and 'in' not in var['intent'] and
311: (12)               'hide' not in var['intent'] and 'inplace' not in var['intent'])
312: (0)           def isintent_out(var):
313: (4)             return 'out' in var.get('intent', [])
314: (0)           def isintent_hide(var):
315: (4)             return ('intent' in var and ('hide' in var['intent'] or
316: (12)               ('out' in var['intent'] and 'in' not in var['intent'] and
317: (16)                 (not l_or(isintent inout, isintent inplace))(var))))
318: (0)           def isintent_nothide(var):
319: (4)             return not isintent_hide(var)
320: (0)           def isintent_c(var):
321: (4)             return 'c' in var.get('intent', [])
322: (0)           def isintent_cache(var):
323: (4)             return 'cache' in var.get('intent', [])
324: (0)           def isintent_copy(var):
325: (4)             return 'copy' in var.get('intent', [])
326: (0)           def isintent_overwrite(var):
327: (4)             return 'overwrite' in var.get('intent', [])
328: (0)           def isintent_callback(var):
329: (4)             return 'callback' in var.get('intent', [])
330: (0)           def isintent inplace(var):
331: (4)             return 'inplace' in var.get('intent', [])
332: (0)           def isintent aux(var):
333: (4)             return 'aux' in var.get('intent', [])
334: (0)           def isintent aligned4(var):
335: (4)             return 'aligned4' in var.get('intent', [])
336: (0)           def isintent aligned8(var):
337: (4)             return 'aligned8' in var.get('intent', [])
338: (0)           def isintent aligned16(var):

```

```

339: (4)             return 'aligned16' in var.get('intent', [])
340: (0)             isintent_dict = {isintent_in: 'INTENT_IN', isintent_inout: 'INTENT_INOUT',
341: (17)             isintent_out: 'INTENT_OUT', isintent_hide: 'INTENT_HIDE',
342: (17)             isintent_cache: 'INTENT_CACHE',
343: (17)             isintent_c: 'INTENT_C', isoptional: 'OPTIONAL',
344: (17)             isintent_inplace: 'INTENT_INPLACE',
345: (17)             isintent_aligned4: 'INTENT_ALIGNED4',
346: (17)             isintent_aligned8: 'INTENT_ALIGNED8',
347: (17)             isintent_aligned16: 'INTENT_ALIGNED16',
348: (17)             }
349: (0)             def isprivate(var):
350: (4)                 return 'attrspec' in var and 'private' in var['attrspec']
351: (0)             def hasinitvalue(var):
352: (4)                 return '=' in var
353: (0)             def hasinitvalueasString(var):
354: (4)                 if not hasinitvalue(var):
355: (8)                     return 0
356: (4)                 return var['='][0] in ['"', "'"]
357: (0)             def hasnote(var):
358: (4)                 return 'note' in var
359: (0)             def hasresultnote(rout):
360: (4)                 if not isfunction(rout):
361: (8)                     return 0
362: (4)                 if 'result' in rout:
363: (8)                     a = rout['result']
364: (4)                 else:
365: (8)                     a = rout['name']
366: (4)                 if a in rout['vars']:
367: (8)                     return hasnote(rout['vars'][a])
368: (4)                 return 0
369: (0)             def hascommon(rout):
370: (4)                 return 'common' in rout
371: (0)             def containscommon(rout):
372: (4)                 if hascommon(rout):
373: (8)                     return 1
374: (4)                 if hasbody(rout):
375: (8)                     for b in rout['body']:
376: (12)                         if containscommon(b):
377: (16)                             return 1
378: (4)                         return 0
379: (0)             def containsmodule(block):
380: (4)                 if ismodule(block):
381: (8)                     return 1
382: (4)                 if not hasbody(block):
383: (8)                     return 0
384: (4)                 for b in block['body']:
385: (8)                     if containsmodule(b):
386: (12)                         return 1
387: (4)                         return 0
388: (0)             def hasbody(rout):
389: (4)                 return 'body' in rout
390: (0)             def hascallstatement(rout):
391: (4)                 return getcallstatement(rout) is not None
392: (0)             def istrue(var):
393: (4)                 return 1
394: (0)             def isfalse(var):
395: (4)                 return 0
396: (0)             class F2PYError(Exception):
397: (4)                 pass
398: (0)             class throw_error:
399: (4)                 def __init__(self, mess):
400: (8)                     self.mess = mess
401: (4)                 def __call__(self, var):
402: (8)                     mess = '\n\n var = %s\n Message: %s\n' % (var, self.mess)
403: (8)                     raise F2PYError(mess)
404: (0)             def l_and(*f):
405: (4)                 l1, l2 = 'lambda v', []
406: (4)                 for i in range(len(f)):
407: (8)                     l1 = '%s,f%d=f[%d]' % (l1, i, i)

```

```

408: (8)           12.append('f%d(v)' % (i))
409: (4)           return eval('%s:%s' % (11, ' and '.join(12)))
410: (0)           def l_or(*f):
411: (4)             11, 12 = 'lambda v', []
412: (4)             for i in range(len(f)):
413: (8)               11 = '%s,f%d=f[%d]' % (11, i, i)
414: (8)               12.append('f%d(v)' % (i))
415: (4)             return eval('%s:%s' % (11, ' or '.join(12)))
416: (0)           def l_not(f):
417: (4)             return eval('lambda v,f=f:not f(v)')
418: (0)           def isdummyroutine(rout):
419: (4)             try:
420: (8)               return rout['f2pyenhancements']['fortranname'] == ''
421: (4)             except KeyError:
422: (8)               return 0
423: (0)           def getfortranname(rout):
424: (4)             try:
425: (8)               name = rout['f2pyenhancements']['fortranname']
426: (8)               if name == '':
427: (12)                 raise KeyError
428: (8)               if not name:
429: (12)                 errmess('Failed to use fortranname from %s\n' %
430: (20)                   (rout['f2pyenhancements']))
431: (12)                 raise KeyError
432: (4)             except KeyError:
433: (8)               name = rout['name']
434: (4)             return name
435: (0)           def getmultilineblock(rout, blockname, comment=1, counter=0):
436: (4)             try:
437: (8)               r = rout['f2pyenhancements'].get(blockname)
438: (4)             except KeyError:
439: (8)               return
440: (4)             if not r:
441: (8)               return
442: (4)             if counter > 0 and isinstance(r, str):
443: (8)               return
444: (4)             if isinstance(r, list):
445: (8)               if counter >= len(r):
446: (12)                 return
447: (8)               r = r[counter]
448: (4)             if r[:3] == "'''":
449: (8)               if comment:
450: (12)                 r = '\t/* start ' + blockname + \
451: (16)                   ' multiline (' + repr(counter) + ') */\n' + r[3:]
452: (8)             else:
453: (12)               r = r[3:]
454: (8)             if r[-3:] == "'''":
455: (12)               if comment:
456: (16)                 r = r[:-3] + '\n\t/* end multiline (' + repr(counter) + ') */'
457: (12)               else:
458: (16)                 r = r[:-3]
459: (8)             else:
460: (12)               errmess("%s multiline block should end with ````: %s\n" %
461: (20)                   (blockname, repr(r)))
462: (4)             return r
463: (0)           def getcallstatement(rout):
464: (4)             return getmultilineblock(rout, 'callstatement')
465: (0)           def getcallprotoargument(rout, cb_map={}):
466: (4)             r = getmultilineblock(rout, 'callprotoargument', comment=0)
467: (4)             if r:
468: (8)               return r
469: (4)             if hascallstatement(rout):
470: (8)               outmess(
471: (12)                 'warning: callstatement is defined without callprotoargument\n')
472: (8)             return
473: (4)             from .capi_maps import getctype
474: (4)             arg_types, arg_types2 = [], []
475: (4)             if l_and(isstringfunction, l_not(isfunction_wrap))(rout):
476: (8)               arg_types.extend(['char*', 'size_t'])

```

```

477: (4)           for n in rout['args']:
478: (8)             var = rout['vars'][n]
479: (8)               if isintent_callback(var):
480: (12)                 continue
481: (8)               if n in cb_map:
482: (12)                 ctype = cb_map[n] + '_typedef'
483: (8)               else:
484: (12)                 ctype = getctype(var)
485: (12)                 if l_and(isintent_c, l_or(isscalar, iscomplex))(var):
486: (16)                   pass
487: (12)                 elif isstring(var):
488: (16)                   pass
489: (12)                 else:
490: (16)                   if not isattr_value(var):
491: (20)                     ctype = ctype + '*'
492: (12)                   if ((isstring(var)
493: (17)                     or isarrayofstrings(var) # obsolete?
494: (17)                     or isstringarray(var))):
495: (16)                       arg_types2.append('size_t')
496: (8)                     arg_types.append(ctype)
497: (4)                     proto_args = ','.join(arg_types + arg_types2)
498: (4)                   if not proto_args:
499: (8)                     proto_args = 'void'
500: (4)                   return proto_args
501: (0)     def getusercode(rout):
502: (4)       return getmultilineblock(rout, 'usercode')
503: (0)     def getusercode1(rout):
504: (4)       return getmultilineblock(rout, 'usercode', counter=1)
505: (0)     def getpymethoddef(rout):
506: (4)       return getmultilineblock(rout, 'pymethoddef')
507: (0)     def getargs(rout):
508: (4)       sortargs, args = [], []
509: (4)       if 'args' in rout:
510: (8)         args = rout['args']
511: (8)         if 'sortvars' in rout:
512: (12)           for a in rout['sortvars']:
513: (16)             if a in args:
514: (20)               sortargs.append(a)
515: (12)           for a in args:
516: (16)             if a not in sortargs:
517: (20)               sortargs.append(a)
518: (8)         else:
519: (12)           sortargs = rout['args']
520: (4)       return args, sortargs
521: (0)     def getargs2(rout):
522: (4)       sortargs, args = [], rout.get('args', [])
523: (4)       auxvars = [a for a in rout['vars'].keys() if isintent_aux(rout['vars'][a])
524: (15)                     and a not in args]
525: (4)       args = auxvars + args
526: (4)       if 'sortvars' in rout:
527: (8)         for a in rout['sortvars']:
528: (12)           if a in args:
529: (16)             sortargs.append(a)
530: (8)           for a in args:
531: (12)             if a not in sortargs:
532: (16)               sortargs.append(a)
533: (4)         else:
534: (8)           sortargs = auxvars + rout['args']
535: (4)       return args, sortargs
536: (0)     def getrestdoc(rout):
537: (4)       if 'f2pymultilines' not in rout:
538: (8)         return None
539: (4)       k = None
540: (4)       if rout['block'] == 'python module':
541: (8)         k = rout['block'], rout['name']
542: (4)       return rout['f2pymultilines'].get(k, None)
543: (0)     def gentitle(name):
544: (4)       ln = (80 - len(name) - 6) // 2
545: (4)       return '/%s %s %s*' % (ln * '*', name, ln * '*')

```

```

546: (0)
547: (4)
548: (8)
549: (4)
550: (0)
551: (4)
552: (8)
553: (4)
554: (0)
555: (4)
556: (8)
557: (4)
558: (8)
559: (4)
560: (8)
561: (12)
562: (8)
563: (12)
564: (8)
565: (12)
566: (8)
567: (12)
568: (8)
569: (12)
570: (4)
571: (0)
572: (4)
573: (8)
574: (12)
575: (8)
576: (4)
577: (8)
578: (12)
579: (8)
580: (12)
581: (16)
582: (12)
583: (16)
584: (20)
585: (16)
586: (20)
587: (12)
588: (16)
589: (20)
590: (24)
591: (28)
592: (32)
593: (20)
594: (24)
595: (8)
596: (12)
597: (4)
598: (0)
599: (4)
600: (4)
601: (8)
602: (12)
603: (12)
604: (12)
605: (16)
606: (8)
607: (4)
608: (8)
609: (4)
610: (8)
611: (8)
612: (12)
613: (4)
614: (8)

def flatlist(lst):
    if isinstance(lst, list):
        return reduce(lambda x, y, f=flatlist: x + f(y), lst, [])
    return [lst]

def stripcomma(s):
    if s and s[-1] == ',':
        return s[:-1]
    return s

def replace(str, d, defaultsep=''):
    if isinstance(d, list):
        return [replace(str, _m, defaultsep) for _m in d]
    if isinstance(str, list):
        return [replace(_m, d, defaultsep) for _m in str]
    for k in 2 * list(d.keys()):
        if k == 'separatorsfor':
            continue
        if 'separatorsfor' in d and k in d['separatorsfor']:
            sep = d['separatorsfor'][k]
        else:
            sep = defaultsep
        if isinstance(d[k], list):
            str = str.replace('#%s#' % (k), sep.join(flatlist(d[k])))
        else:
            str = str.replace('#%s#' % (k), d[k])
    return str

def dictappend(rd, ar):
    if isinstance(ar, list):
        for a in ar:
            rd = dictappend(rd, a)
        return rd
    for k in ar.keys():
        if k[0] == '_':
            continue
        if k in rd:
            if isinstance(rd[k], str):
                rd[k] = [rd[k]]
            if isinstance(rd[k], list):
                if isinstance(ar[k], list):
                    rd[k] = rd[k] + ar[k]
                else:
                    rd[k].append(ar[k])
            elif isinstance(ar[k], dict):
                if k == 'separatorsfor':
                    for k1 in ar[k].keys():
                        if k1 not in rd[k]:
                            rd[k][k1] = ar[k][k1]
                else:
                    rd[k] = dictappend(rd[k], ar[k])
            else:
                rd[k] = ar[k]
        return rd
    def applyrules(rules, d, var={}):
        ret = {}
        if isinstance(rules, list):
            for r in rules:
                rr = applyrules(r, d, var)
                ret = dictappend(ret, rr)
                if '_break' in rr:
                    break
            return ret
        if '_check' in rules and (not rules['_check'](var)):
            return ret
        if 'need' in rules:
            res = applyrules({'needs': rules['need']}, d, var)
            if 'needs' in res:
                cfuncs.append_needs(res['needs'])
        for k in rules.keys():
            if k == 'separatorsfor':

```

```

615: (12)                                ret[k] = rules[k]
616: (12)                                continue
617: (8)                                 if isinstance(rules[k], str):
618: (12)                                ret[k] = replace(rules[k], d)
619: (8)                                 elif isinstance(rules[k], list):
620: (12)                                ret[k] = []
621: (12)                                for i in rules[k]:
622: (16)                                  ar = applyrules({k: i}, d, var)
623: (16)                                  if k in ar:
624: (20)                                    ret[k].append(ar[k])
625: (8)                                 elif k[0] == '_':
626: (12)                                continue
627: (8)                                 elif isinstance(rules[k], dict):
628: (12)                                ret[k] = []
629: (12)                                for k1 in rules[k].keys():
630: (16)                                  if isinstance(k1, types.FunctionType) and k1(var):
631: (20)                                    if isinstance(rules[k][k1], list):
632: (24)                                      for i in rules[k][k1]:
633: (28)  if isinstance(i, dict):
634: (32)  res = applyrules({'supertext': i}, d, var)
635: (32)  if 'supertext' in res:
636: (36)  i = res['supertext']
637: (32)  else:
638: (36)  i = ''
639: (28)  ret[k].append(replace(i, d))
640: (20)                                      else:
641: (24)  i = rules[k][k1]
642: (24)  if isinstance(i, dict):
643: (28)  res = applyrules({'supertext': i}, d)
644: (28)  if 'supertext' in res:
645: (32)  i = res['supertext']
646: (28)  else:
647: (32)  i = ''
648: (24)  ret[k].append(replace(i, d))
649: (8)                                     else:
650: (12)                                       errmess('applyrules: ignoring rule %s.\n' % repr(rules[k]))
651: (8)                                     if isinstance(ret[k], list):
652: (12)                                       if len(ret[k]) == 1:
653: (16)   ret[k] = ret[k][0]
654: (12)                                       if ret[k] == []:
655: (16)   del ret[k]
656: (4)                                     return ret
657: (0) _f2py_module_name_match = re.compile(r'\s*python\s*module\s*(?P<name>[\w_]+)', re.I).match
658: (37) _f2py_user_module_name_match = re.compile(r'\s*python\s*module\s*(?P<name>[\w_]*?'
659: (0) [_w_]*)', re.I).match
660: (42)
661: (0) def get_f2py_modulename(source):
662: (4)   name = None
663: (4)   with open(source) as f:
664: (8)     for line in f:
665: (12)       m = _f2py_module_name_match(line)
666: (12)       if m:
667: (16)         if _f2py_user_module_name_match(line): # skip *__user__* names
668: (20)           continue
669: (16)         name = m.group('name')
670: (16)         break
671: (4)   return name
672: (0) def getuseblocks(pymod):
673: (4)   all_uses = []
674: (4)   for inner in pymod['body']:
675: (8)     for modblock in inner['body']:
676: (12)       if modblock.get('use'):
677: (16)         all_uses.extend([x for x in modblock.get("use").keys() if "__"
not in x])
678: (4)   return all_uses
679: (0) def process_f2cmap_dict(f2cmap_all, new_map, c2py_map, verbose = False):
680: (4)   """
681: (4)   Update the Fortran-to-C type mapping dictionary with new mappings and

```

```

682: (4) return a list of successfully mapped C types.
683: (4) This function integrates a new mapping dictionary into an existing
684: (4) Fortran-to-C type mapping dictionary. It ensures that all keys are in
685: (4) lowercase and validates new entries against a given C-to-Python mapping
686: (4) dictionary. Redefinitions and invalid entries are reported with a warning.
687: (4) Parameters
688: (4) -----
689: (4) f2cmap_all : dict
690: (8) The existing Fortran-to-C type mapping dictionary that will be
updated.
691: (8) It should be a dictionary of dictionaries where the main keys
represent
692: (8) Fortran types and the nested dictionaries map Fortran type specifiers
693: (8) to corresponding C types.
694: (4) new_map : dict
695: (8) A dictionary containing new type mappings to be added to `f2cmap_all`.
696: (8) The structure should be similar to `f2cmap_all`, with keys
representing
697: (8) Fortran types and values being dictionaries of type specifiers and
their
698: (8) C type equivalents.
699: (4) c2py_map : dict
700: (8) A dictionary used for validating the C types in `new_map`. It maps C
701: (8) types to corresponding Python types and is used to ensure that the C
702: (8) types specified in `new_map` are valid.
703: (4) verbose : boolean
704: (8) A flag used to provide information about the types mapped
Returns
705: (4) -----
706: (4) tuple of (dict, list)
707: (4) The updated Fortran-to-C type mapping dictionary and a list of
708: (8) successfully mapped C types.
709: (4) """
710: (4) f2cmap_mapped = []
711: (4) new_map_lower = {}
712: (4) for k, d1 in new_map.items():
713: (8)     d1_lower = {k1.lower(): v1 for k1, v1 in d1.items()}
714: (8)     new_map_lower[k.lower()] = d1_lower
715: (4) for k, d1 in new_map_lower.items():
716: (8)     if k not in f2cmap_all:
717: (12)         f2cmap_all[k] = {}
718: (8)         for k1, v1 in d1.items():
719: (12)             if v1 in c2py_map:
720: (16)                 if k1 in f2cmap_all[k]:
721: (20)                     outmess(
722: (24)                         "\tWarning: redefinition of {'%s': {'%s': '%s' -"
723: (24) > '%s'}}\n"
724: (24) )
725: (20)             % (k, k1, f2cmap_all[k][k1], v1)
726: (16)         )
727: (16)         f2cmap_all[k][k1] = v1
728: (20)         if verbose:
729: (16)             outmess('\tMapping "%s(kind=%s)" to "%s"\n' % (k, k1, v1))
730: (12)             f2cmap_mapped.append(v1)
731: (16)         else:
732: (20)             if verbose:
733: (24)                 errmess(
734: (24)                     "\tIgnoring map {'%s': {'%s': '%s'}}: '%s' must be in
735: (20) %s\n"
736: (4)                 % (k, k1, v1, v1, list(c2py_map.keys()))
)
return f2cmap_all, f2cmap_mapped

```

## File 143 - capi maps.py:

```
1: (0)          """
2: (0)          Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
3: (0)          Copyright 2011 -- present NumPy Developers.
```

```

4: (0) Permission to use, modify, and distribute this software is given under the
5: (0) terms of the NumPy License.
6: (0) NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
7: (0)
8: (0)
9: (0) """
10: (0) from . import __version__
11: (0) f2py_version = __version__.version
12: (0) import copy
13: (0) import re
14: (0) import os
15: (0) from .crackfortran import markoutercomma
16: (0) from . import cb_rules
17: (0) from ._isocbind import iso_c_binding_map, isoc_c2pycode_map, iso_c2py_map
18: (0) from .auxfuncs import *
19: (0) __all__ = [
20: (4)     'getctype', 'getstrlength', 'getarrdims', 'getpydocsign',
21: (4)     'getarrdocsign', 'getinit', 'sign2map', 'routsign2map', 'modsign2map',
22: (4)     'cb_sign2map', 'cb_routsign2map', 'common_sign2map', 'process_f2cmap_dict'
23: (0)
24: (0) depargs = []
25: (0) lcb_map = {}
26: (0) lcb2_map = {}
27: (0) c2py_map = {'double': 'float',
28: (12)             'float': 'float',                                # forced casting
29: (12)             'long_double': 'float',                            # forced casting
30: (12)             'char': 'int',                                 # forced casting
31: (12)             'signed_char': 'int',                            # forced casting
32: (12)             'unsigned_char': 'int',                           # forced casting
33: (12)             'short': 'int',                               # forced casting
34: (12)             'unsigned_short': 'int',                          # forced casting
35: (12)             'int': 'int',                                # forced casting
36: (12)             'long': 'int',                               # forced casting
37: (12)             'long_long': 'long',                             # forced casting
38: (12)             'unsigned': 'int',                             # forced casting
39: (12)             'complex_float': 'complex',                         # forced casting
40: (12)             'complex_double': 'complex',                        # forced casting
41: (12)             'complex_long_double': 'complex',                      # forced casting
42: (12)             'string': 'string',
43: (0)             'character': 'bytes',
44: (0)         }
45: (0)         c2capi_map = {'double': 'NPY_DOUBLE',
46: (16)             'float': 'NPY_FLOAT',
47: (16)             'long_double': 'NPY_LONGLONG',
48: (16)             'char': 'NPY_BYTE',
49: (16)             'unsigned_char': 'NPY_UBYTE',
50: (16)             'signed_char': 'NPY_BYTE',
51: (16)             'short': 'NPY_SHORT',
52: (16)             'unsigned_short': 'NPY USHORT',
53: (16)             'int': 'NPY_INT',
54: (16)             'unsigned': 'NPY_UINT',
55: (16)             'long': 'NPY_LONG',
56: (16)             'unsigned_long': 'NPY ULONG',
57: (16)             'long_long': 'NPY LONGLONG',
58: (16)             'unsigned_long_long': 'NPY ULLONG',
59: (16)             'complex_float': 'NPY_CFLOAT',
60: (16)             'complex_double': 'NPY_CDOUBLE',
61: (16)             'complex_long_double': 'NPY_CDOUBLE',
62: (0)             'string': 'NPY_STRING',
63: (0)             'character': 'NPY_STRING'}
64: (0)         c2pycode_map = {'double': 'd',
65: (16)             'float': 'f',
66: (16)             'long_double': 'g',
67: (16)             'char': 'b',
68: (16)             'unsigned_char': 'B',
69: (16)             'signed_char': 'b',
70: (16)             'short': 'h',
71: (16)             'unsigned_short': 'H',
72: (16)             'int': 'i',
73: (16)             'unsigned': 'I',
74: (16)             'long': 'l',

```

```

73: (16)                         'unsigned_long': 'L',
74: (16)                         'long_long': 'q',
75: (16)                         'unsigned_long_long': 'Q',
76: (16)                         'complex_float': 'F',
77: (16)                         'complex_double': 'D',
78: (16)                         'complex_long_double': 'G',
79: (16)                         'string': 'S',
80: (16)                         'character': 'c'}
81: (0)                          c2buildvalue_map = {'double': 'd',
82: (20)                           'float': 'f',
83: (20)                           'char': 'b',
84: (20)                           'signed_char': 'b',
85: (20)                           'short': 'h',
86: (20)                           'int': 'i',
87: (20)                           'long': 'l',
88: (20)                           'long_long': 'L',
89: (20)                           'complex_float': 'N',
90: (20)                           'complex_double': 'N',
91: (20)                           'complex_long_double': 'N',
92: (20)                           'string': 'y',
93: (20)                           'character': 'c'}
94: (0)                          f2cmap_all = {'real': {'': 'float', '4': 'float', '8': 'double',
95: (23)                           '12': 'long_double', '16': 'long_double'},
96: (14)                           'integer': {'': 'int', '1': 'signed_char', '2': 'short',
97: (26)                           '4': 'int', '8': 'long_long',
98: (26)                           '-1': 'unsigned_char', '-2': 'unsigned_short',
99: (26)                           '-4': 'unsigned', '-8': 'unsigned_long_long'},
100: (14)                          'complex': {'': 'complex_float', '8': 'complex_float',
101: (26)                           '16': 'complex_double', '24': 'complex_long_double',
102: (26)                           '32': 'complex_long_double'},
103: (14)                          'complexkind': {'': 'complex_float', '4': 'complex_float',
104: (30)                           '8': 'complex_double', '12':
'complex_long_double',
105: (30)                           '16': 'complex_long_double'},
106: (14)                          'logical': {'': 'int', '1': 'char', '2': 'short', '4': 'int',
107: (26)                           '8': 'long_long'},
108: (14)                          'double complex': {'': 'complex_double'},
109: (14)                          'double precision': {'': 'double'},
110: (14)                          'byte': {'': 'char'},
111: (14) }
112: (0)                          c2pycode_map.update(isoc_c2pycode_map)
113: (0)                          c2py_map.update(iso_c2py_map)
114: (0)                          f2cmap_all, _ = process_f2cmap_dict(f2cmap_all, iso_c_binding_map, c2py_map)
115: (0)                          f2cmap_default = copy.deepcopy(f2cmap_all)
116: (0)                          f2cmap_mapped = []
117: (0)                          def load_f2cmap_file(f2cmap_file):
118: (4)                            global f2cmap_all, f2cmap_mapped
119: (4)                            f2cmap_all = copy.deepcopy(f2cmap_default)
120: (4)                            if f2cmap_file is None:
121: (8)                              f2cmap_file = '.f2py_f2cmap'
122: (8)                              if not os.path.isfile(f2cmap_file):
123: (12)                                return
124: (4)                            try:
125: (8)                              outmess('Reading f2cmap from {!r} ...'.format(f2cmap_file))
126: (8)                              with open(f2cmap_file) as f:
127: (12)                                d = eval(f.read().lower(), {}, {})
128: (8)                                f2cmap_all, f2cmap_mapped = process_f2cmap_dict(f2cmap_all, d,
c2py_map, True)
129: (8)                                outmess('Successfully applied user defined f2cmap changes\n')
130: (4)                            except Exception as msg:
131: (8)                              errmess('Failed to apply user defined f2cmap changes: %s. Skipping.\n'
% (msg))
132: (0)                            cformat_map = {'double': '%g',
133: (15)                           'float': '%g',
134: (15)                           'long_double': '%Lg',
135: (15)                           'char': '%d',
136: (15)                           'signed_char': '%d',
137: (15)                           'unsigned_char': '%hu',
138: (15)                           'short': '%hd',

```

```

139: (15)                         'unsigned_short': '%hu',
140: (15)                         'int': '%d',
141: (15)                         'unsigned': '%u',
142: (15)                         'long': '%ld',
143: (15)                         'unsigned_long': '%lu',
144: (15)                         'long_long': '%ld',
145: (15)                         'complex_float': '(%g,%g)',
146: (15)                         'complex_double': '(%g,%g)',
147: (15)                         'complex_long_double': '(%Lg,%Lg)',
148: (15)                         'string': '\\\"%s\\\"',
149: (15)                         'character': "'%c'",
150: (15)
151: (0)  def getctype(var):
152: (4)      """
153: (4)          Determines C type
154: (4)      """
155: (4)      ctype = 'void'
156: (4)      if isfunction(var):
157: (8)          if 'result' in var:
158: (12)              a = var['result']
159: (8)          else:
160: (12)              a = var['name']
161: (8)          if a in var['vars']:
162: (12)              return getctype(var['vars'][a])
163: (8)          else:
164: (12)              errmess('getctype: function %s has no return value?!\\n' % a)
165: (4)      elif issubroutine(var):
166: (8)          return ctype
167: (4)      elif ischaracter_or_characterarray(var):
168: (8)          return 'character'
169: (4)      elif isstring_or_stringarray(var):
170: (8)          return 'string'
171: (4)      elif 'typespec' in var and var['typespec'].lower() in f2cmap_all:
172: (8)          typespec = var['typespec'].lower()
173: (8)          f2cmap = f2cmap_all[typespec]
174: (8)          ctype = f2cmap[''] # default type
175: (8)          if 'kindselector' in var:
176: (12)              if '*' in var['kindselector']:
177: (16)                  try:
178: (20)                      ctype = f2cmap[var['kindselector']['*']]
179: (16)                  except KeyError:
180: (20)                      errmess('getctype: "%s %s %s" not supported.\\n' %
181: (28)                          (var['typespec'], '*', var['kindselector']['*']))
182: (12)                  elif 'kind' in var['kindselector']:
183: (16)                      if typespec + 'kind' in f2cmap_all:
184: (20)                          f2cmap = f2cmap_all[typespec + 'kind']
185: (16)
186: (20)                      try:
187: (16)                          ctype = f2cmap[var['kindselector']['kind']]
188: (20)                      except KeyError:
189: (24)                          if typespec in f2cmap_all:
190: (20)                              f2cmap = f2cmap_all[typespec]
191: (24)                              try:
192: (20)                                  ctype = f2cmap[str(var['kindselector']['kind'])]
193: (24)                                  except KeyError:
194: (32)                                      errmess('getctype: "%s(kind=%s)" is mapped to C "%s"'
(to override define dict(%s = dict(%s=<C typespec>)) in %s/.f2py_f2cmap file).\\n'
195: (35)  % (typespec, var['kindselector']['kind'],
ctype,
196: (4)  typespec, var['kindselector']['kind']),
os.getcwd()))
197: (8)                  else:
198: (12)                      if not isexternal(var):
199: (4)                          errmess('getctype: No C-type found in "%s", assuming void.\\n' %
var)
200: (0)                          return ctype
def f2cexpr(expr):
201: (4)      """Rewrite Fortran expression as f2py supported C expression.
202: (4)      Due to the lack of a proper expression parser in f2py, this
203: (4)      function uses a heuristic approach that assumes that Fortran

```

```

204: (4)           arithmetic expressions are valid C arithmetic expressions when
205: (4)           mapping Fortran function calls to the corresponding C function/CPP
206: (4)           macros calls.
207: (4)
208: (4)           """
209: (4)           expr = re.sub(r'\blen\b', 'f2py_slen', expr)
210: (0)           return expr
211: (4)           def getstrlength(var):
212: (8)             if isstringfunction(var):
213: (12)               if 'result' in var:
214: (8)                 a = var['result']
215: (12)               else:
216: (8)                 a = var['name']
217: (12)               if a in var['vars']:
218: (8)                 return getstrlength(var['vars'][a])
219: (12)               else:
220: (4)                 errmess('getstrlength: function %s has no return value?!\\n' % a)
221: (8)             if not isstring(var):
222: (12)               errmess(
223: (8)                 'getstrlength: expected a signature of a string but got: %s\\n' %
224: (12)                 (repr(var)))
225: (4)               len = '1'
226: (8)               if 'charelector' in var:
227: (12)                 a = var['charelector']
228: (8)                 if '*' in a:
229: (12)                   len = a['*']
230: (4)                   elif 'len' in a:
231: (8)                     len = f2cexpr(a['len'])
232: (12)                   if re.match(r'\\(\s*(\\*|:)\\s*)', len) or re.match(r'\\(*|:)\\', len):
233: (8)                     if isintent_hide(var):
234: (12)                       errmess('getstrlength:intent(hide): expected a string with defined
235: (4)                         length but got: %s\\n' %
236: (0)                           (repr(var)))
237: (4)                         len = '-1'
238: (8)                         return len
239: (4)           def getarrdims(a, var, verbose=0):
240: (8)             ret = {}
241: (8)             if isstring(var) and not isarray(var):
242: (12)               ret['size'] = getstrlength(var)
243: (8)               ret['rank'] = '0'
244: (8)               ret['dims'] = ''
245: (8)             elif isscalar(var):
246: (12)               ret['size'] = '1'
247: (8)               ret['rank'] = '0'
248: (8)               ret['dims'] = ''
249: (8)
250: (12)             elif isarray(var):
251: (8)               dim = copy.copy(var['dimension'])
252: (12)               ret['size'] = '*'.join(dim)
253: (8)               try:
254: (12)                 ret['size'] = repr(eval(ret['size']))
255: (8)               except Exception:
256: (12)                 pass
257: (8)                 ret['dims'] = ','.join(dim)
258: (8)                 ret['rank'] = repr(len(dim))
259: (8)                 ret['rank*[-1]'] = repr(len(dim) * [-1])[1:-1]
260: (8)                 for i in range(len(dim)): # solve dim for dependencies
261: (12)                   v = []
262: (12)                   if dim[i] in depargs:
263: (16)                     v = [dim[i]]
264: (12)                   else:
265: (16)                     for va in depargs:
266: (20)                       if re.match(r'\\.\\*?\\b%s\\b\\.\\*' % va, dim[i]):
267: (24)                         v.append(va)
268: (12)                     for va in v:
269: (16)                       if depargs.index(va) > depargs.index(a):
270: (20)                         dim[i] = '*'
271: (8)                         break
272: (8)               ret['setdims'], i = '', -1
273: (8)               for d in dim:
274: (12)                 i = i + 1

```

```

271: (12)           if d not in ['*', ':', '(*)', '( :)']:
272: (16)             ret['setdims'] = '%s#varname#_Dims[%d]=%s,' % (
273: (20)               ret['setdims'], i, d)
274: (8)             if ret['setdims']:
275: (12)               ret['setdims'] = ret['setdims'][:-1]
276: (8)             ret['cbsetdims'], i = '', -1
277: (8)             for d in var['dimension']:
278: (12)               i = i + 1
279: (12)               if d not in ['*', ':', '(*)', '( :)']:
280: (16)                 ret['cbsetdims'] = '%s#varname#_Dims[%d]=%s,' % (
281: (20)                   ret['cbsetdims'], i, d)
282: (12)             elif isintent_in(var):
283: (16)               outmess('getarrdims:warning: assumed shape array, using 0
instead of %r\n'
284: (24)                           % (d))
285: (16)             ret['cbsetdims'] = '%s#varname#_Dims[%d]=%s,' % (
286: (20)               ret['cbsetdims'], i, 0)
287: (12)             elif verbose:
288: (16)               errmess(
289: (20)                 'getarrdims: If in call-back function: array argument %s
must have bounded dimensions: got %s\n' % (repr(a), repr(d)))
290: (8)             if ret['cbsetdims']:
291: (12)               ret['cbsetdims'] = ret['cbsetdims'][:-1]
292: (4)             return ret
293: (0)             def getpydocsign(a, var):
294: (4)               global lcb_map
295: (4)               if isfunction(var):
296: (8)                 if 'result' in var:
297: (12)                   af = var['result']
298: (8)                 else:
299: (12)                   af = var['name']
300: (8)                 if af in var['vars']:
301: (12)                   return getpydocsign(af, var['vars'][af])
302: (8)                 else:
303: (12)                   errmess('getctype: function %s has no return value?!\\n' % af)
304: (8)                 return ''
305: (4)               sig, sigout = a, a
306: (4)               opt = ''
307: (4)               if isintent_in(var):
308: (8)                 opt = 'input'
309: (4)               elif isintent_inout(var):
310: (8)                 opt = 'in/output'
311: (4)               out_a = a
312: (4)               if isintent_out(var):
313: (8)                 for k in var['intent']:
314: (12)                   if k[:4] == 'out=':
315: (16)                     out_a = k[4:]
316: (16)                     break
317: (4)               init = ''
318: (4)               ctype = getctype(var)
319: (4)               if hasinitvalue(var):
320: (8)                 init, showinit = getinit(a, var)
321: (8)                 init = ', optional\\n      Default: %s' % showinit
322: (4)               if isscalar(var):
323: (8)                 if isintent_inout(var):
324: (12)                   sig = '%s : %s rank-0 array(%s,\\'%s\\')%s' % (a, opt,
c2py_map[ctype],
325: (57)   c2pycode_map[ctype],
init)
326: (8)               else:
327: (12)                 sig = '%s : %s %s%s' % (a, opt, c2py_map[ctype], init)
328: (8)                 sigout = '%s : %s' % (out_a, c2py_map[ctype])
329: (4)               elif isstring(var):
330: (8)                 if isintent_inout(var):
331: (12)                   sig = '%s : %s rank-0 array(string(len=%s),\\'c\\')%s' % (
332: (16)                     a, opt, getstrlen(var), init)
333: (8)                 else:
334: (12)                   sig = '%s : %s string(len=%s)%s' % (
335: (16)                     a, opt, getstrlen(var), init)

```

```

336: (8)             sigout = '%s : string(len=%s)' % (out_a, getstrlength(var))
337: (4)             elif isarray(var):
338: (8)                 dim = var['dimension']
339: (8)                 rank = repr(len(dim))
340: (8)                 sig = '%s : %s rank-%s array(\'%s\') with bounds (%s)%s' % (a, opt,
rank,
341: (68)
c2pycode_map[
342: (72)
ctype],
343: (68)
','.join(dim), init)
344: (8)             if a == out_a:
345: (12)                 sigout = '%s : rank-%s array(\'%s\') with bounds (%s)\'\
346: (16)                     % (a, rank, c2pycode_map[ctype], ','.join(dim))
347: (8)             else:
348: (12)                 sigout = '%s : rank-%s array(\'%s\') with bounds (%s) and %s
storage\'\
349: (16)                     % (out_a, rank, c2pycode_map[ctype], ','.join(dim), a)
350: (4)             elif isexternal(var):
351: (8)                 ua = ''
352: (8)                 if a in lcb_map and lcb_map[a] in lcb2_map and 'argname' in
lcb2_map[lcb_map[a]]:
353: (12)                     ua = lcb2_map[lcb_map[a]]['argname']
354: (12)                     if not ua == a:
355: (16)                         ua = ' => %s' % ua
356: (12)                     else:
357: (16)                         ua = ''
358: (8)                     sig = '%s : call-back function%s' % (a, ua)
359: (8)                     sigout = sig
360: (4)             else:
361: (8)                 errmess(
362: (12)                     'getpydocsig: Could not resolve docsignature for "%s".\n' % a)
363: (4)             return sig, sigout
364: (0)             def getarrdocsig(a, var):
365: (4)                 ctype = getctype(var)
366: (4)                 if isstring(var) and (not isarray(var)):
367: (8)                     sig = '%s : rank-0 array(string(len=%s),\'c\')' % (a,
368: (59)                                     getstrlength(var))
369: (4)                 elif isscalar(var):
370: (8)                     sig = '%s : rank-0 array(%s,\'%s\')' % (a, c2py_map[ctype],
371: (48)                                     c2pycode_map[ctype],)
372: (4)                 elif isarray(var):
373: (8)                     dim = var['dimension']
374: (8)                     rank = repr(len(dim))
375: (8)                     sig = '%s : rank-%s array(\'%s\') with bounds (%s)' % (a, rank,
376: (63)                                     c2pycode_map[
377: (67)   ctype],
378: (63)   ','.join(dim)))
379: (4)             return sig
380: (0)             def getinit(a, var):
381: (4)                 if isstring(var):
382: (8)                     init, showinit = "'''", "''"
383: (4)                 else:
384: (8)                     init, showinit = '', ''
385: (4)                 if hasinitvalue(var):
386: (8)                     init = var['=']
387: (8)                     showinit = init
388: (8)                     if iscomplex(var) or iscomplexarray(var):
389: (12)                         ret = {}
390: (12)                         try:
391: (16)                             v = var["="]
392: (16)                             if ',' in v:
393: (20)                                 ret['init.r'], ret['init.i'] = markoutercomma(
394: (24)                                     v[1:-1]).split('@,@')
395: (16)
396: (20)
397: (20)
398: (12)
except Exception:

```

```

399: (16)                     raise ValueError(
400: (20)                         'getinit: expected complex number `(r,i)\` but got `%s\`'
as initial value of %r.' % (init, a))
401: (12)                     if isarray(var):
402: (16)                         init = '(capi_c.r=%s,capi_c.i=%s,capi_c)' % (
403: (20)                             ret['init.r'], ret['init.i'])
404: (8)                     elif isstring(var):
405: (12)                         if not init:
406: (16)                             init, showinit = '""', """
407: (12)                         if init[0] == "":
408: (16)                             init = '%s' % (init[1:-1].replace('\"', '\\\"'))
409: (12)                         if init[0] == '':
410: (16)                             showinit = "%s" % (init[1:-1])
411: (4)                     return init, showinit
412: (0) def get_elsize(var):
413: (4)     if isstring(var) or isstringarray(var):
414: (8)         elsize = getstrlen(var)
415: (8)         elsize = var['cheselector'].get('f2py_len', elsize)
416: (8)         return elsize
417: (4)     if ischaracter(var) or ischaracterarray(var):
418: (8)         return '1'
419: (4)     return '1'
420: (0) def sign2map(a, var):
421: (4)     """
422: (4)         varname,ctype,atype
423: (4)         init,init.r,init.i,pytype
424: (4)         vardebuginfo,vardebugshowvalue,varshowvalue
425: (4)         varrformat
426: (4)         intent
427: (4)         """
428: (4)         out_a = a
429: (4)         if isintent_out(var):
430: (8)             for k in var['intent']:
431: (12)                 if k[:4] == 'out=':
432: (16)                     out_a = k[4:]
433: (16)                     break
434: (4)         ret = {'varname': a, 'outvarname': out_a, 'ctype': getctype(var)}
435: (4)         intent_flags = []
436: (4)         for f, s in isintent_dict.items():
437: (8)             if f(var):
438: (12)                 intent_flags.append('F2PY_%s' % s)
439: (4)         if intent_flags:
440: (8)             ret['intent'] = '|'.join(intent_flags)
441: (4)         else:
442: (8)             ret['intent'] = 'F2PY_INTENT_IN'
443: (4)         if isarray(var):
444: (8)             ret['varrformat'] = 'N'
445: (4)             elif ret['ctype'] in c2buildvalue_map:
446: (8)                 ret['varrformat'] = c2buildvalue_map[ret['ctype']]
447: (4)             else:
448: (8)                 ret['varrformat'] = 'O'
449: (4)             ret['init'], ret['showinit'] = getinit(a, var)
450: (4)             if hasinitvalue(var) and iscomplex(var) and not isarray(var):
451: (8)                 ret['init.r'], ret['init.i'] = markoutercomma(
452: (12)                     ret['init'][1:-1]).split('@,@')
453: (4)             if isexternal(var):
454: (8)                 ret['cbnamekey'] = a
455: (8)                 if a in lcb_map:
456: (12)                     ret['cbname'] = lcb_map[a]
457: (12)                     ret['maxnofargs'] = lcb2_map[lcb_map[a]]['maxnofargs']
458: (12)                     ret['nofoptargs'] = lcb2_map[lcb_map[a]]['nofoptargs']
459: (12)                     ret['cbdocstr'] = lcb2_map[lcb_map[a]]['docstr']
460: (12)                     ret['cblatedocstr'] = lcb2_map[lcb_map[a]]['latedocstr']
461: (8)                 else:
462: (12)                     ret['cbname'] = a
463: (12)                     errmess('sign2map: Confused: external %s is not in lcb_map%s.\n' %
(
464: (16)                         a, list(lcb_map.keys())))
465: (4)                     if isstring(var):

```

```

466: (8)           ret['length'] = getstrlength(var)
467: (4)           if isArray(var):
468: (8)             ret = dictappend(ret, getarrdims(a, var))
469: (8)             dim = copy.copy(var['dimension'])
470: (4)             if ret['ctype'] in c2capi_map:
471: (8)               ret['atype'] = c2capi_map[ret['ctype']]
472: (8)               ret['elsize'] = get_elsize(var)
473: (4)             if debugcapi(var):
474: (8)               il = [isintent_in, 'input', isintent_out, 'output',
475: (14)                 isintent_inout, 'inoutput', isrequired, 'required',
476: (14)                 isoptional, 'optional', isintent_hide, 'hidden',
477: (14)                 iscomplex, 'complex scalar',
478: (14)                 l_and(isscalar, l_not(iscomplex)), 'scalar',
479: (14)                 issstring, 'string', isarray, 'array',
480: (14)                 iscomplexarray, 'complex array', isstringarray, 'string array',
481: (14)                 iscomplexfunction, 'complex function',
482: (14)                 l_and(isfunction, l_not(iscomplexfunction)), 'function',
483: (14)                 isexternal, 'callback',
484: (14)                 isintent_callback, 'callback',
485: (14)                 isintent_aux, 'auxiliary',
486: (14)               ]
487: (8)             rl = []
488: (8)             for i in range(0, len(il), 2):
489: (12)               if il[i](var):
490: (16)                 rl.append(il[i + 1])
491: (8)             if issstring(var):
492: (12)               rl.append('slen(%s)=%s' % (a, ret['length']))
493: (8)             if isArray(var):
494: (12)               ddim = ','.join(
495: (16)                 map(lambda x, y: '%s|%s' % (x, y), var['dimension'], dim))
496: (12)               rl.append('dims(%s)' % ddim)
497: (8)             if isexternal(var):
498: (12)               ret['vardebuginfo'] = 'debug-capi:%s=>%s:%s' % (
499: (16)                 a, ret['cbname'], ','.join(rl))
500: (8)             else:
501: (12)               ret['vardebuginfo'] = 'debug-capi:%s %s=%s:%s' % (
502: (16)                 ret['ctype'], a, ret['showinit'], ','.join(rl))
503: (8)             if isscalar(var):
504: (12)               if ret['ctype'] in cformat_map:
505: (16)                 ret['vardebugshowvalue'] = 'debug-capi:%s=%s' % (
506: (20)                   a, cformat_map[ret['ctype']])
507: (8)             if issstring(var):
508: (12)               ret['vardebugshowvalue'] = 'debug-capi:slen(%s)=%%d %s=\\"%%s\\\\"'
509: (16)                   a, a)
510: (8)             if isexternal(var):
511: (12)               ret['vardebugshowvalue'] = 'debug-capi:%s=%%p' % (a)
512: (4)             if ret['ctype'] in cformat_map:
513: (8)               ret['varshowvalue'] = '#name#:%s=%s' % (a, cformat_map[ret['ctype']])
514: (8)               ret['showvalueformat'] = '%s' % (cformat_map[ret['ctype']])
515: (4)             if issstring(var):
516: (8)               ret['varshowvalue'] = '#name#:slen(%s)=%%d %s=\\"%%s\\\\"' % (a, a)
517: (4)             ret['pydocsign'], ret['pydocsignout'] = getpydocsign(a, var)
518: (4)             if hasnote(var):
519: (8)               ret['note'] = var['note']
520: (4)             return ret
521: (0)           def routsign2map(rout):
522: (4)             """
523: (4)               name,NAME,begintitle,endtitle
524: (4)               rname,ctype,rformat
525: (4)               routdebugshowvalue
526: (4)             """
527: (4)             global lcb_map
528: (4)             name = rout['name']
529: (4)             fname = getfortranname(rout)
530: (4)             ret = {'name': name,
531: (11)               'texname': name.replace('_', '\\_'),
532: (11)               'name_lower': name.lower(),
533: (11)               'NAME': name.upper(),

```

```

534: (11)             'begintitle': gentitle(name),
535: (11)             'endtitle': gentitle('end of %s' % name),
536: (11)             'fortranname': fname,
537: (11)             'FORTRANNAME': fname.upper(),
538: (11)             'callstatement': getcallstatement(rout) or '',
539: (11)             'usercode': getusercode(rout) or '',
540: (11)             'usercode1': getusercode1(rout) or '',
541: (11)             }
542: (4)             if '_' in fname:
543: (8)                 ret['F_FUNC'] = 'F_FUNC_US'
544: (4)             else:
545: (8)                 ret['F_FUNC'] = 'F_FUNC'
546: (4)             if '_' in name:
547: (8)                 ret['F_WRAPPEDFUNC'] = 'F_WRAPPEDFUNC_US'
548: (4)             else:
549: (8)                 ret['F_WRAPPEDFUNC'] = 'F_WRAPPEDFUNC'
550: (4)             lcb_map = {}
551: (4)             if 'use' in rout:
552: (8)                 for u in rout['use'].keys():
553: (12)                     if u in cb_rules.cb_map:
554: (16)                         for un in cb_rules.cb_map[u]:
555: (20)                             ln = un[0]
556: (20)                             if 'map' in rout['use'][u]:
557: (24)                                 for k in rout['use'][u]['map'].keys():
558: (28)                                     if rout['use'][u]['map'][k] == un[0]:
559: (32)   ln = k
560: (32)   break
561: (20)                         lcb_map[ln] = un[1]
562: (4)             elif 'externals' in rout and rout['externals']:
563: (8)                 errmess('routsig2map: Confused: function %s has externals %s but no
"use" statement.\n' %
564: (12)                     ret['name'], repr(rout['externals']))
565: (4)             ret['callprotoargument'] = getcallprotoargument(rout, lcb_map) or ''
566: (4)             if isfunction(rout):
567: (8)                 if 'result' in rout:
568: (12)                     a = rout['result']
569: (8)                 else:
570: (12)                     a = rout['name']
571: (8)             ret['rname'] = a
572: (8)             ret['pydocsign'], ret['pydocsignout'] = getpydocsign(a, rout)
573: (8)             ret['ctype'] = getctype(rout['vars'][a])
574: (8)             if hasresultnote(rout):
575: (12)                 ret['resultnote'] = rout['vars'][a]['note']
576: (12)                 rout['vars'][a]['note'] = ['See elsewhere.']
577: (8)             if ret['ctype'] in c2buildvalue_map:
578: (12)                 ret['rformat'] = c2buildvalue_map[ret['ctype']]
579: (8)             else:
580: (12)                 ret['rformat'] = '0'
581: (12)                 errmess('routsig2map: no c2buildvalue key for type %s\n' %
582: (20)                     (repr(ret['ctype'])))
583: (8)             if debugcapi(rout):
584: (12)                 if ret['ctype'] in cformat_map:
585: (16)                     ret['routdebugshowvalue'] = 'debug-capi:%s=%s' % (
586: (20)                         a, cformat_map[ret['ctype']])
587: (12)                     if isstringfunction(rout):
588: (16)                         ret['routdebugshowvalue'] = 'debug-capi:slen(%s)=%%d
%=%s\\\"%s\\\"' %
589: (20)                         a, a)
590: (8)             if isstringfunction(rout):
591: (12)                 ret['rlength'] = getstrlen(rout['vars'][a])
592: (12)                 if ret['rlength'] == '-1':
593: (16)                     errmess('routsig2map: expected explicit specification of the
length of the string returned by the fortran function %s; taking 10.\n' %
594: (20)                         repr(rout['name'])))
595: (16)                     ret['rlength'] = '10'
596: (4)             if hasnote(rout):
597: (8)                 ret['note'] = rout['note']
598: (8)                 rout['note'] = ['See elsewhere.']
599: (4)             return ret

```

```

600: (0)
601: (4)
602: (4)
603: (4)
604: (4)
605: (8)
606: (15)
607: (15)
608: (4)
609: (8)
610: (15)
611: (15)
612: (4)
613: (4)
614: (8)
615: (4)
616: (4)
617: (4)
618: (8)
619: (4)
620: (8)
621: (4)
622: (4)
623: (8)
624: (4)
625: (8)
626: (4)
627: (0)
628: (4)
629: (4)
630: (4)
631: (4)
632: (8)
633: (8)
634: (4)
635: (8)
636: (4)
637: (8)
638: (4)
639: (4)
640: (8)
641: (8)
642: (4)
643: (0)
644: (4)
645: (4)
646: (4)
647: (4)
648: (4)
649: (11)
650: (4)
651: (8)
652: (12)
653: (8)
654: (12)
655: (8)
656: (30)
657: (33)
658: (33)
659: (33)
660: (8)
661: (4)
662: (8)
663: (8)
664: (4)
665: (4)
666: (4)
667: (4)
668: (4)

def modsign2map(m):
    """
    modulename
    """
    if ismodule(m):
        ret = {'f90modulename': m['name'],
               'F90MODULENAME': m['name'].upper(),
               'texf90modulename': m['name'].replace('_', '\\_')}
    else:
        ret = {'modulename': m['name'],
               'MODULENAME': m['name'].upper(),
               'texmodulename': m['name'].replace('_', '\\_')}
    ret['restdoc'] = getrestdoc(m) or []
    if hasnote(m):
        ret['note'] = m['note']
    ret['usercode'] = getusercode(m) or ''
    ret['usercode1'] = getusercode1(m) or ''
    if m['body']:
        ret['interface_usercode'] = getusercode(m['body'][0]) or ''
    else:
        ret['interface_usercode'] = ''
    ret['pymethoddef'] = getpymethoddef(m) or ''
    if 'coutput' in m:
        ret['coutput'] = m['coutput']
    if 'f2py_wrapper_output' in m:
        ret['f2py_wrapper_output'] = m['f2py_wrapper_output']
    return ret

def cb_sign2map(a, var, index=None):
    ret = {'varname': a}
    ret['varname_i'] = ret['varname']
    ret['ctype'] = getctype(var)
    if ret['ctype'] in c2capi_map:
        ret['atype'] = c2capi_map[ret['ctype']]
        ret['elsize'] = get_elsize(var)
    if ret['ctype'] in cformat_map:
        ret['showvalueformat'] = '%s' % (cformat_map[ret['ctype']])
    if isarray(var):
        ret = dictappend(ret, getarrdims(a, var))
    ret['pydocsign'], ret['pydocsignout'] = getpydocsign(a, var)
    if hasnote(var):
        ret['note'] = var['note']
        var['note'] = ['See elsewhere.']
    return ret

def cb_routsign2map(rout, um):
    """
    name,begintitle,endtitle,argname
    ctype,rctype,maxnofargs,nofoptargs,returnncptr
    """
    ret = {'name': 'cb_%s_in_%s' % (rout['name'], um),
           'returnncptr': ''}
    if isintent_callback(rout):
        if '_' in rout['name']:
            F_FUNC = 'F_FUNC_US'
        else:
            F_FUNC = 'F_FUNC'
        ret['callbackname'] = '%s(%s,%s)' \
            % (F_FUNC,
               rout['name'].lower(),
               rout['name'].upper(),
               )
    ret['static'] = 'extern'
    else:
        ret['callbackname'] = ret['name']
        ret['static'] = 'static'
    ret['argname'] = rout['name']
    ret['begintitle'] = gentitle(ret['name'])
    ret['endtitle'] = gentitle('end of %s' % ret['name'])
    ret['ctype'] = getctype(rout)
    ret['rctype'] = 'void'

```

```

669: (4)           if ret['ctype'] == 'string':
670: (8)             ret['rctype'] = 'void'
671: (4)           else:
672: (8)             ret['rctype'] = ret['ctype']
673: (4)           if ret['rctype'] != 'void':
674: (8)             if iscomplexfunction(rout):
675: (12)               ret['returncptr'] = """
676: (0)           return_value=
677: (0)
678: (8)
679: (12)           """
680: (4)           if ret['ctype'] in cformat_map:
681: (8)             ret['showvalueformat'] = '%s' % (cformat_map[ret['ctype']])
682: (4)           if issstringfunction(rout):
683: (8)             ret['strlenth'] = getstrlenth(rout)
684: (4)           if isfunction(rout):
685: (8)             if 'result' in rout:
686: (12)               a = rout['result']
687: (8)             else:
688: (12)               a = rout['name']
689: (8)             if hasnote(rout['vars'][a]):
690: (12)               ret['note'] = rout['vars'][a]['note']
691: (12)               rout['vars'][a]['note'] = ['See elsewhere.']
692: (8)             ret['rname'] = a
693: (8)             ret['pydocsing'], ret['pydocsingout'] = getpydocsing(a, rout)
694: (8)             if iscomplexfunction(rout):
695: (12)               ret['rctype'] = """
696: (0)           void
697: (0)
698: (4)           else:
699: (8)             if hasnote(rout):
700: (12)               ret['note'] = rout['note']
701: (12)               rout['note'] = ['See elsewhere.']
702: (4)             nogargs = 0
703: (4)             nooptargs = 0
704: (4)             if 'args' in rout and 'vars' in rout:
705: (8)               for a in rout['args']:
706: (12)                 var = rout['vars'][a]
707: (12)                 if l_or(isintent_in, isintent_inout)(var):
708: (16)                   nogargs = nogargs + 1
709: (16)                   if isoptional(var):
710: (20)                     nooptargs = nooptargs + 1
711: (4)             ret['maxnofargs'] = repr(nogargs)
712: (4)             ret['nooptargs'] = repr(noptargs)
713: (4)             if hasnote(rout) and isfunction(rout) and 'result' in rout:
714: (8)               ret['routnote'] = rout['note']
715: (8)               rout['note'] = ['See elsewhere.']
716: (4)             return ret
717: (0)           def common_sign2map(a, var): # obsolete
718: (4)             ret = {'varname': a, 'ctype': getctype(var)}
719: (4)             if issstringarray(var):
720: (8)               ret['ctype'] = 'char'
721: (4)             if ret['ctype'] in c2capi_map:
722: (8)               ret['atype'] = c2capi_map[ret['ctype']]
723: (8)               ret['elsize'] = get_elsize(var)
724: (4)             if ret['ctype'] in cformat_map:
725: (8)               ret['showvalueformat'] = '%s' % (cformat_map[ret['ctype']])
726: (4)             if isArray(var):
727: (8)               ret = dictappend(ret, getarrdims(a, var))
728: (4)             elif issstring(var):
729: (8)               ret['size'] = getstrlenth(var)
730: (8)               ret['rank'] = '1'
731: (4)             ret['pydocsing'], ret['pydocsingout'] = getpydocsing(a, var)
732: (4)             if hasnote(var):
733: (8)               ret['note'] = var['note']
734: (8)               var['note'] = ['See elsewhere.']
735: (4)             ret['arrdocstr'] = getarrdocsing(a, var)
736: (4)             return ret

```

## File 144 - cb\_rules.py:

```

1: (0)      """
2: (0)      Build call-back mechanism for f2py2e.
3: (0)      Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
4: (0)      Copyright 2011 -- present NumPy Developers.
5: (0)      Permission to use, modify, and distribute this software is given under the
6: (0)      terms of the NumPy License.
7: (0)      NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
8: (0)
9: (0)      from . import __version__
10: (0)     from .auxfuncs import (
11: (4)         applyrules, debugcapi, dictappend, errmess, getargs, hasnote, isarray,
12: (4)         iscomplex, iscomplexarray, iscomplexfunction, isfunction, isintent_c,
13: (4)         isintent_hide, isintent_in, isintent_inout, isintent_nothide,
14: (4)         isintent_out, isoptional, isrequired, isscalar, isstring,
15: (4)         issstringfunction, issubroutine, l_and, l_not, l_or, outmess, replace,
16: (4)         stripcomma, throw_error
17: (0)     )
18: (0)     from . import cfuncs
19: (0)     f2py_version = __version__.version
20: (0)     cb_routine_rules = {
21: (4)         'cbtypedefs': 'typedef #rctype#(*#name#_typedef
(#optargs_td##args_td##strarglens_td##noargs#);',
22: (4)             'body': """
23: (0)             typedef struct {
24: (4)                 PyObject *capi;
25: (4)                 PyTupleObject *args_capi;
26: (4)                 int nofargs;
27: (4)                 jmp_buf jmpbuf;
28: (0)             } #name#_t;
29: (0)             static F2PY_THREAD_LOCAL_DECL #name#_t *_active_#name# = NULL;
30: (0)             static #name#_t *swap_active_#name#(#name#_t *ptr) {
31: (4)                 _active_#name# = ptr;
32: (4)                 return prev;
33: (0)             }
34: (0)             static #name#_t *get_active_#name#(void) {
35: (4)                 return _active_#name#;
36: (0)             }
37: (0)             static #name#_t *swap_active_#name#(#name#_t *ptr) {
38: (4)                 char *key = "__f2py_cb_##name#";
39: (4)                 return (#name#_t *)F2PySwapThreadLocalCallbackPtr(key, ptr);
40: (0)             }
41: (0)             static #name#_t *get_active_#name#(void) {
42: (4)                 char *key = "__f2py_cb_##name#";
43: (4)                 return (#name#_t *)F2PyGetThreadLocalCallbackPtr(key);
44: (0)             }
45: (0)             /*typedef #rctype#(*#name#_typedef
(#optargs_td##args_td##strarglens_td##noargs#);*/
46: (4)                 PyTupleObject *capi_arglist = NULL;
47: (4)                 PyObject *capi_return = NULL;
48: (4)                 PyObject *capi_tmp = NULL;
49: (4)                 PyObject *capi_arglist_list = NULL;
50: (4)                 int capi_j, capi_i = 0;
51: (4)                 int capi_longjmp_ok = 1;
52: (0)                 f2py_cb_start_clock();
53: (4)                 cb = get_active_#name#();
54: (4)                 if (cb == NULL) {
55: (8)                     capi_longjmp_ok = 0;
56: (8)                     cb = &cb_local;
57: (4)                 }
58: (4)                 capi_arglist = cb->args_capi;
59: (4)                 CFUNCSMESS(("cb:Call-back function #name# (maxnofargs=#maxnofargs#(-
#nofoptargs#))\\n\\n"));
60: (4)                 CFUNCSMESSPY("cb:#name#_capi=", cb->capi);
61: (4)                 if (cb->capi==NULL) {
62: (8)                     capi_longjmp_ok = 0;

```

```

63: (8)             cb->capi = PyObject_GetAttrString(#modulename#_module, "#argname#");
64: (8)             CFUNCSMESSPY("cb:#name#_capi=", cb->capi);
65: (4)
66: (4)             if (cb->capi==NULL) {
67: (8)                 PyErr_SetString(#modulename#_error, "cb: Callback #argname# not
defined (as an argument or module #modulename# attribute).\\n");
68: (8)                 goto capi_fail;
69: (4)
70: (4)             if (F2PyCapsule_Check(cb->capi)) {
71: (4)
72: (4)             if (capi_arglist==NULL) {
73: (8)                 capi_longjmp_ok = 0;
74: (8)                 capi_tmp =
PyObject_GetAttrString(#modulename#_module, "#argname#_extra_args");
75: (8)                 if (capi_tmp) {
76: (12)                     capi_arglist = (PyTupleObject *)PySequence_Tuple(capi_tmp);
77: (12)                     Py_DECREF(capi_tmp);
78: (12)                     if (capi_arglist==NULL) {
79: (16)                         PyErr_SetString(#modulename#_error, "Failed to convert
#modulename#.#argname#_extra_args to tuple.\\n");
80: (16)                         goto capi_fail;
81: (12)
82: (8)                 } else {
83: (12)                     PyErr_Clear();
84: (12)                     capi_arglist = (PyTupleObject *)Py_BuildValue("(())");
85: (8)
86: (4)
87: (4)                 if (capi_arglist == NULL) {
88: (8)                     PyErr_SetString(#modulename#_error, "Callback #argname# argument list
is not set.\\n");
89: (8)                     goto capi_fail;
90: (4)
91: (4)                 capi_arglist_list = PySequence_List(capi_arglist);
92: (4)                 if (capi_arglist_list == NULL) goto capi_fail;
93: (4)                 CFUNCSMESSPY("cb:capi_arglist=", capi_arglist_list);
94: (4)                 CFUNCSMESSPY("cb:capi_arglist=", capi_arglist);
95: (4)                 CFUNCSMESS("cb:Call-back calling Python function #argname#.\\n");
96: (0)                 f2py_cb_start_call_clock();
97: (4)                 capi_return = PyObject_CallObject(cb->capi, (PyObject *)capi_arglist_list);
98: (4)                 Py_DECREF(capi_arglist_list);
99: (4)                 capi_arglist_list = NULL;
100: (4)                 capi_return = PyObject_CallObject(cb->capi, (PyObject *)capi_arglist);
101: (0)                 f2py_cb_stop_call_clock();
102: (4)                 CFUNCSMESSPY("cb:capi_return=", capi_return);
103: (4)                 if (capi_return == NULL) {
104: (8)                     fprintf(stderr, "capi_return is NULL\\n");
105: (8)                     goto capi_fail;
106: (4)
107: (4)                 if (capi_return == Py_None) {
108: (8)                     Py_DECREF(capi_return);
109: (8)                     capi_return = Py_BuildValue("(())");
110: (4)
111: (4)                 else if (!PyTuple_Check(capi_return)) {
112: (8)                     capi_return = Py_BuildValue("(N)", capi_return);
113: (4)
114: (4)                     capi_j = PyTuple_Size(capi_return);
115: (4)                     capi_i = 0;
116: (4)                     CFUNCSMESS("cb:#name#:successful\\n");
117: (4)                     Py_DECREF(capi_return);
118: (0)                     f2py_cb_stop_clock();
119: (4)                     goto capi_return_pt;
120: (0)                 capi_fail:
121: (4)                     fprintf(stderr, "Call-back #name# failed.\\n");
122: (4)                     Py_XDECREF(capi_return);
123: (4)                     Py_XDECREF(capi_arglist_list);
124: (4)                     if (capi_longjmp_ok) {
125: (8)                         longjmp(cb->jmpbuf, -1);
126: (4)                     }
127: (0)                 capi_return_pt:

```

```

128: (4)
129: (0)
130: (0)
131: (4)
132: (4)
133: (4)
134: (4)
135: (4)
136: (4)
137: (0)
138: (4)
139: (0)
140: (0)
141: (4)
142: (8)
143: (26)
'freemem': '\n',
144: (26)
145: (26)
146: (26)
147: (26)
148: (26)
149: (26)
150: (26)
151: (26)
152: (8)
'/*frompyobj*/',
153: (8)
'/*freemem*/',
154: (8)
155: (8)
156: (8)
157: (8)
158: (8)
159: (8)
160: (8)
161: (8)
162: (8)
163: (8)
164: (8)
165: (8)
166: (8)
167: (8)
168: (8)
169: (4)
170: (8)
171: (8)
172: (12)
173: (12)
174: (4)
175: (8)
176: (10)
177: (10)
178: (4)
179: (8)
180: (23)
181: (4)
182: (12)
183: (13)
184: (8)
185: (8)
'GETSCALARFROMPYTUPLE'],
186: (8)
187: (8)
l_not(iscomplexfunction))
188: (4)
189: (4)
{ # String function
190: (8)
    'pyobjfrom': {debugcapi: '    fprintf(stderr,"debug-
capi:cb:#name%:d:\n",return_value_len);'},
    ;
}
"""
    'need': ['setjmp.h', 'CFUNCSMESS', 'F2PY_THREAD_LOCAL_DECL'],
    'maxnofargs': '#maxnofargs#',
    'nofoptargs': '#nofoptargs#',
    'docstr': """
        def #argname#(#docsignture#): return #doctreturn#\n\
            'latexdocstr': """
    {{}\nverb@def #argname#(#latexdocsignture#): return #doctreturn#@{}}
    'docstrshort': 'def #argname#(#docsignture#): return #doctreturn#'
}
cb_rout_rules = [
    { # Init
        'separatorsfor': {'decl': '\n',
                           'args': ',', 'optargs': '',
                           'pyobjfrom': '\n',
                           'args_td': ',', 'optargs_td': '',
                           'args_nm': ',', 'optargs_nm': '',
                           'frompyobj': '\n', 'setdims': '\n',
                           'docstrsigns': '\n\n\n',
                           'latexdocstrsigns': '\n',
                           'latexdocstrreq': '\n', 'latexdocstropt': '\n',
                           'latexdocstrout': '\n', 'latexdocstrcbs': '\n',
                           },
        'decl': /*decl*/,
        'pyobjfrom': /*pyobjfrom*/,
        'frompyobj':
        'args': [], 'optargs': '',
        'return': '',
        'strarglens': '',
        'freemem':
        'args_td': [],
        'optargs_td': '',
        'strarglens_td': '',
        'args_nm': [],
        'optargs_nm': '',
        'strarglens_nm': '',
        'noargs': '',
        'setdims': /*setdims*/,
        'docstrsigns': '',
        'latexdocstrsigns': '',
        'docstrreq': '    Required arguments:',
        'docstropt': '    Optional arguments:',
        'docstrout': '    Return objects:',
        'docstrcbs': '    Call-back functions:',
        'doctreturn': '',
        'docsignture': '',
        'docsignopt': '',
        'latexdocstrreq': '\nnoindent Required arguments:',
        'latexdocstropt': '\nnoindent Optional arguments:',
        'latexdocstrout': '\nnoindent Return objects:',
        'latexdocstrcbs': '\nnoindent Call-back functions:',
        'routnote': {hasnote: '--- #note#', l_not(hasnote): ''},
    }, { # Function
        'decl': '    #ctype# return_value = 0;',
        'frompyobj': [
            {debugcapi: '    CFUNCSMESS("cb:Getting return_value->");'},
            ''
        ]
        if (capi_j>capi_i) {
            GETSCALARFROMPYTUPLE(capi_return,capi_i++,&return_value,#ctype#,
                "#ctype#_from_pyobj failed in converting return_value of"
                " call-back function #name# to C #ctype#\n");
        } else {
            fprintf(stderr,"Warning: call-back function #name# did not provide"
                " return value (index=%d, type=#ctype#)\n",capi_i);
        }
        {debugcapi:
            fprintf(stderr,"#showvalueformat#\n",return_value);
        ],
        'need': ['#ctype#_from_pyobj', {debugcapi: 'CFUNCSMESS'},
        'return': '    return return_value;',
        '_check': l_and(isfunction, l_not(isstringfunction),
        l_not(iscomplexfunction)),
        },
        { # String function
        'pyobjfrom': {debugcapi: '    fprintf(stderr,"debug-
capi:cb:#name%:d:\n",return_value_len);'},
        }
    }
]

```

```

191: (8)                                'args': '#ctype# return_value,int return_value_len',
192: (8)                                'args_nm': 'return_value,&return_value_len',
193: (8)                                'args_td': '#ctype# ,int',
194: (8)                                'frompyobj': [
195: (12)                                 {debugcapi: '    CFUNCSMESS("cb:Getting return_value->\\\"");'},
196: (12)                                 """\
197: (4)                                if (capi_j>capi_i) {
198: (8)                                    GETSTRFROMPYTUPLE(capi_return,capi_i++,return_value,return_value_len);
199: (4)                                } else {
200: (8)                                    fprintf(stderr,"Warning: call-back function #name# did not provide"
201: (23)   " return value (index=%d, type=#ctype#)\\n",capi_i);
202: (4)                                }"""
203: (12)                                 {debugcapi:
204: (13)                                     fprintf(stderr,#showvalueformat#.\\n",return_value);'}
205: (8)                                ],
206: (8)                                'need': ['#ctype#_from_pyobj', {debugcapi: 'CFUNCSMESS'},
207: (17)   'string.h', 'GETSTRFROMPYTUPLE'],
208: (8)                                'return': 'return;',
209: (8)                                '_check': isstringfunction
210: (4),
211: (4)                                { # Complex function
212: (8)                                    'optargs': """
213: (0),
214: (8)                                    'optargs_nm': """
215: (0),
216: (0),
217: (8)                                    'optargs_td': """
218: (0),
219: (8)                                    'decl': """
220: (0),
221: (8)                                'frompyobj': [
222: (12)                                 {debugcapi: '    CFUNCSMESS("cb:Getting return_value->");'},
223: (12)                                 """\
224: (4)                                if (capi_j>capi_i) {
225: (8)                                    GETSCALARFROMPYTUPLE(capi_return,capi_i++,&return_value,#ctype#,
226: (10)   "#ctype#_from_pyobj failed in converting return_value of call-
back\\"
227: (10)   "\\ function #name# to C #ctype#\\n\\");
228: (8)   GETSCALARFROMPYTUPLE(capi_return,capi_i++,return_value,#ctype#,
229: (10)   "#ctype#_from_pyobj failed in converting return_value of call-
back\\"
230: (10)   "\\ function #name# to C #ctype#\\n\\");
231: (4),
232: (8)                                } else {
233: (16)                                    fprintf(stderr,
234: (16)   "Warning: call-back function #name# did not provide\\"
235: (4)   "\\ return value (index=%d, type=#ctype#)\\n\\",capi_i);
236: (12),
237: (4)
238: (4)                                (return_value).i);
239: (0)                                fprintf(stderr,#showvalueformat#.\\n\\",(return_value).r,
240: (8)                                fprintf(stderr,#showvalueformat#.\\n\\",(*return_value).r,
241: (8)                                """}
242: (4),
243: (4)
244: (0),
245: (8)                                'return': """
246: (17)                                return return_value;
247: (8)                                return;
248: (4),
249: (4)                                '_check': iscomplexfunction
250: (5),
251: (24)                                {debugcapi: """
252: (5)   '#pydocsignout#',
253: (5)   'latexdocstrout': ['\\item[]{}\\verb@#pydocsignout#@{}'],
254: (4)   {hasnote: '--- #note#'}],
255: (0)   'doctreturn': '#rname#,',
256: (4)   '_check': isfunction},
257: (4)   {'_check': issubroutine, 'return': 'return;'}
258: (0)                                ]

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

256: (0) cb_arg_rules = [
257: (4)     { # Doc
258: (8)         'docstropt': {l_and(isoptional, isintent_nothide): '
#pydocsign#},
259: (8)         'docstrreq': {l_and(isrequired, isintent_nothide): '
#pydocsign#},
260: (8)         'docstrout': {isintent_out: '          #pydocsignout#'},
261: (8)         'latexdocstropt': {l_and(isoptional, isintent_nothide): ['\\item[]'
{}\\verb@#pydocsign#@{}'],
262: (65)   {hasnote: '--'
- #note#']}},
263: (8)         'latexdocstrreq': {l_and(isrequired, isintent_nothide): ['\\item[]'
{}\\verb@#pydocsign#@{}'],
264: (65)   {hasnote: '--'
- #note#']}},
265: (8)         'latexdocstrout': {isintent_out: ['\\item[]'
{}\\verb@#pydocsignout#@{}'],
266: (42)   {l_and(hasnote, isintent_hide): '---'
#note#},
267: (43)   l_and(hasnote, isintent_nothide):
'--- See above.'}],
268: (8)         'docsign': {l_and(isrequired, isintent_nothide): '#varname#,'},
269: (8)         'docsignopt': {l_and(isoptional, isintent_nothide): '#varname#,'},
270: (8)         'depend': ''},
271: (4)     },
272: (4)     {
273: (8)         'args': {
274: (12)             l_and(isscalar, isintent_c): '#ctype# #varname_i#',
275: (12)             l_and(isscalar, l_not(isintent_c)): '#ctype#
*#varname_i#_cb_capi',
276: (12)             isarray: '#ctype# *#varname_i#',
277: (12)             isstring: '#ctype# #varname_i#'
},
278: (8),
279: (8),
280: (12),
281: (12),
282: (12),
283: (12),
284: (8),
285: (8),
286: (12),
287: (12),
288: (12),
289: (12),
290: (8),
291: (8),
292: (8),
293: (8),
294: (8),
295: (4),
296: (4),
297: (8),
(*#varname_i#_cb_capi);'},
298: (8),
299: (24),
scalar arguments')):
300: (18)
301: (8)         'error': {l_and(isintent_c, isintent_out,
302: (22)             throw_error('intent(c,out) is forbidden for callback
303: (23)             scalar arguments')):
304: (22)             ''},
305: (26)
fprintf(stderr, "#showvalueformat#.\\n",#varname_i#);'},
306: (22)
isintent_c)):

```

```

fprintf(stderr, "#showvalueformat#.\\n", *#varname_i#_cb_capi);'},
308: (22)                                {l_and(debugcapi, l_and(iscomplex, isintent_c)):
309: (26)                                ' fprintf(stderr, "#showvalueformat#.\\n",
310: (22)                                (#varname_i#).r, (#varname_i#).i);'},
311: (26)                                {l_and(debugcapi, l_and(iscomplex, l_not( isintent_c))):
312: (22)                                (*#varname_i#_cb_capi).r, (*#varname_i#_cb_capi).i);'},
313: (8)                                ],
314: (17)                                'need': [{isintent_out: ['#ctype#_from_pyobj',
'GETSCALARFROMPYTUPLE']},,
315: (8)                                {debugcapi: 'CFUNCSMESS'}],
316: (4)                                '_check': isscalar
317: (8)                                {pyobjfrom: [{isintent_in: """\
318: (4)                                if (cb->nofargs>capi_i)
319: (8)                                if (CAPI_ARGLIST_SETITEM(capi_i++,pyobj_from_#ctype#1(#varname_i#)))
320: (12)                                goto capi_fail;"""\n
321: (22)                                {isintent_inout: """\
322: (4)                                if (cb->nofargs>capi_i)
323: (8)                                if
(CAPI_ARGLIST_SETITEM(capi_i++,pyarr_from_p_#ctype#1(#varname_i#_cb_capi)))
324: (12)                                goto capi_fail;"""\n},
325: (8)                                'need': [{isintent_in: 'pyobj_from_#ctype#1'},
326: (17)                                {isintent_inout: 'pyarr_from_p_#ctype#1'},
327: (17)                                {iscomplex: '#ctype#'}],
328: (8)                                '_check': l_and(isscalar, isintent_nothide),
329: (8)                                '_optional': ''
330: (4)                                }, {# String
331: (8)                                'frompyobj': [{debugcapi: ' CFUNCSMESS("cb:Getting #varname#-'
>\"");'},
332: (22)                                """" if (capi_j>capi_i)
333: (8)
GETSTRFROMPYTUPLE(capi_return,capi_i++,#varname_i#,#varname_i#_cb_len);""",
334: (22)                                {debugcapi:
335: (23)
fprintf(stderr, "#showvalueformat#\\"%d:.\\n",#varname_i#,#varname_i#_cb_len);'},
336: (22)                                ],
337: (8)                                'need': ['#ctype#', 'GETSTRFROMPYTUPLE',
338: (17)                                {debugcapi: 'CFUNCSMESS'}, 'string.h'],
339: (8)                                '_check': l_and(isstring, isintent_out)
340: (4)                                },
341: (8)                                'pyobjfrom': [
342: (12)                                {debugcapi:
343: (13)                                (' fprintf(stderr,"debug-capi:cb:#varname#=#showvalueformat#:'%
344: (14)                                '%d:\\n",#varname_i#,#varname_i#_cb_len);')},
345: (12)                                {isintent_in: """\
346: (4)                                if (cb->nofargs>capi_i)
347: (8)                                if
(CAPI_ARGLIST_SETITEM(capi_i++,pyobj_from_#ctype#1size(#varname_i#,#varname_i#_cb_len)))
348: (12)                                goto capi_fail;"""\n
349: (22)                                {isintent_inout: """\
350: (4)                                if (cb->nofargs>capi_i) {
351: (8)                                int #varname_i#_cb_dims[] = {#varname_i#_cb_len};
352: (8)                                if
(CAPI_ARGLIST_SETITEM(capi_i++,pyarr_from_p_#ctype#1(#varname_i#,#varname_i#_cb_dims)))
353: (12)                                goto capi_fail;
354: (4)                                }"""\n},
355: (8)                                'need': [{isintent_in: 'pyobj_from_#ctype#1size'},
356: (17)                                {isintent_inout: 'pyarr_from_p_#ctype#1'}],
357: (8)                                '_check': l_and(isstring, isintent_nothide),
358: (8)                                '_optional': ''
359: (4)                                },
360: (4)                                {
361: (8)                                'decl': '    npy_intp #varname_i#_Dims[#rank#] = {#rank*[-1]#};',
362: (8)                                'setdims': '    #cbsetdims#;',
363: (8)                                '_check': isarray,
364: (8)                                '_depend': ''
365: (4)                                },
366: (4)                                {

```

```

367: (8)           'pyobjfrom': [{debugcapi: '    fprintf(stderr,"debug-
capi:cb:#varname#\n");'}, 
368: (22)           {isintent_c: """\
369: (4)             if (cb->nofargs>capi_i) {
370: (8)               /* tmp_arr will be inserted to capi_arglist_list that will be
371: (11)                 destroyed when leaving callback function wrapper together
372: (11)                 with tmp_arr. */
373: (8)               PyArrayObject *tmp_arr = (PyArrayObject *)PyArray_New(&PyArray_Type,
374: (10)                 NPY_ARRAY_CARRAY,NULL);
375: (0)             """
376: (23)           l_not(isintent_c): """\
377: (4)             if (cb->nofargs>capi_i) {
378: (8)               /* tmp_arr will be inserted to capi_arglist_list that will be
379: (11)                 destroyed when leaving callback function wrapper together
380: (11)                 with tmp_arr. */
381: (8)               PyArrayObject *tmp_arr = (PyArrayObject *)PyArray_New(&PyArray_Type,
382: (10)                 NPY_ARRAY_FARRAY,NULL);
383: (0)             """
384: (23)           },
385: (22)           """
386: (8)             if (tmp_arr==NULL)
387: (12)               goto capi_fail;
388: (8)             if (CAPI_ARGLIST_SETITEM(capi_i++,(PyObject *)tmp_arr))
389: (12)               goto capi_fail;
390: (0)           }"""], 
391: (8)             '_check': l_and(isarray, isintent_nothide, l_or(isintent_in,
392: (8)               '_optional': ''),
393: (4)             }, {
394: (8)               'frompyobj': [{debugcapi: '    CFUNCSMESS("cb:Getting #varname#-
>");'},
395: (22)                   """      if (capi_j>capi_i) {
396: (8)                     PyArrayObject *rv_cb_arr = NULL;
397: (8)                     if ((capi_tmp = PyTuple_GetItem(capi_return,capi_i++))==NULL) goto
398: (8)                       rv_cb_arr =
array_from_pyobj(#atype#,#varname_i#_Dims,#rank#,F2PY_INTENT_IN""",
399: (22)                         {isintent_c: '|F2PY_INTENT_C'},
400: (22)                         """,capi_tmp);
401: (8)                     if (rv_cb_arr == NULL) {
402: (12)                       fprintf(stderr,"rv_cb_arr is NULL\n");
403: (12)                       goto capi_fail;
404: (8)                     }
405: (8)                     MEMCOPY(#varname_i#,PyArray_DATA(rv_cb_arr),PyArray_NBYTES(rv_cb_arr));
406: (8)                     if (capi_tmp != (PyObject *)rv_cb_arr) {
407: (12)                       Py_DECREF(rv_cb_arr);
408: (8)                     }
409: (4)                   }""", 
410: (22)                     {debugcapi: '    fprintf(stderr,<-.\n');},
411: (22)                     ],
412: (8)                     'need': ['MEMCOPY', {iscomplexarray: '#ctype#'}],
413: (8)                     '_check': l_and(isarray, isintent_out)
414: (4)                   }, {
415: (8)                     'doreturn': '#varname#',
416: (8)                     '_check': isintent_out
417: (4)                   }
418: (0)                 ]
419: (0)                 cb_map = {}
420: (0)                 def buildcallbacks(m):
421: (4)                   cb_map[m['name']] = []
422: (4)                   for bi in m['body']:
423: (8)                     if bi['block'] == 'interface':
424: (12)                       for b in bi['body']:
425: (16)                         if b:
426: (20)                           buildcallback(b, m['name'])
427: (16)                         else:
428: (20)                           errmess('warning: empty body for %s\n' % (m['name']))
429: (0)                 def buildcallback(rout, um):
```

```

430: (4)           from . import capi_maps
431: (4)           outmess('    Constructing call-back function "cb_%s_in_%s"\n' %
432: (12)             (rout['name'], um))
433: (4)           args, depargs = getargs(rout)
434: (4)           capi_maps.depargs = depargs
435: (4)           var = rout['vars']
436: (4)           vrd = capi_maps.cb_routsig2map(rout, um)
437: (4)           rd = dictappend({}, vrd)
438: (4)           cb_map[um].append([rout['name'], rd['name']])
439: (4)           for r in cb_rout_rules:
440: (8)             if '_check' in r and r['_check'](rout) or ('_check' not in r):
441: (12)               ar = applyrules(r, vrd, rout)
442: (12)               rd = dictappend(rd, ar)
443: (4)           savevrd = {}
444: (4)           for i, a in enumerate(args):
445: (8)             vrd = capi_maps.cb_sign2map(a, var[a], index=i)
446: (8)             savevrd[a] = vrd
447: (8)             for r in cb_arg_rules:
448: (12)               if '_depend' in r:
449: (16)                 continue
450: (12)               if '_optional' in r and isoptional(var[a]):
451: (16)                 continue
452: (12)               if ('_check' in r and r['_check'](var[a])) or ('_check' not in r):
453: (16)                 ar = applyrules(r, vrd, var[a])
454: (16)                 rd = dictappend(rd, ar)
455: (16)                 if '_break' in r:
456: (20)                   break
457: (4)             for a in args:
458: (8)               vrd = savevrd[a]
459: (8)               for r in cb_arg_rules:
460: (12)                 if '_depend' in r:
461: (16)                   continue
462: (12)                 if ('_optional' not in r) or ('_optional' in r and
isrequired(var[a])):
463: (16)                   continue
464: (12)                 if ('_check' in r and r['_check'](var[a])) or ('_check' not in r):
465: (16)                   ar = applyrules(r, vrd, var[a])
466: (16)                   rd = dictappend(rd, ar)
467: (16)                   if '_break' in r:
468: (20)                     break
469: (4)             for a in depargs:
470: (8)               vrd = savevrd[a]
471: (8)               for r in cb_arg_rules:
472: (12)                 if '_depend' not in r:
473: (16)                   continue
474: (12)                 if '_optional' in r:
475: (16)                   continue
476: (12)                 if ('_check' in r and r['_check'](var[a])) or ('_check' not in r):
477: (16)                   ar = applyrules(r, vrd, var[a])
478: (16)                   rd = dictappend(rd, ar)
479: (16)                   if '_break' in r:
480: (20)                     break
481: (4)             if 'args' in rd and 'optargs' in rd:
482: (8)               if isinstance(rd['optargs'], list):
483: (12)                 rd['optargs'] = rd['optargs'] + ["""
484: (0)           ,
485: (0)           """
486: (12)           rd['optargs_nm'] = rd['optargs_nm'] + ["""
487: (0)           ,
488: (0)           """
489: (12)           rd['optargs_td'] = rd['optargs_td'] + ["""
490: (0)           ,
491: (0)           """
492: (4)           if isinstance(rd['doctreturn'], list):
493: (8)             rd['doctreturn'] = stripcomma(
494: (12)               replace('#doctreturn#', {'doctreturn': rd['doctreturn']}))
495: (4)             optargs = stripcomma(replace('#docsignopt#',
496: (33)                           {'docsignopt': rd['docsignopt']}))
497: (33)           ))

```

```

498: (4)             if optargs == '':
499: (8)                 rd['docsignture'] = stripcomma(
500: (12)                     replace('#docsign#', {'docsign': rd['docsign']}))
501: (4)
502: (8)             else:
503: (37)                 rd['docsignture'] = replace('#docsign#[#docsignopt#]',
504: (38)                             {'docsign': rd['docsign'],
505: (38)                             'docsignopt': optargs,
506: (38)                             })
507: (4)                 rd['latexdocsignture'] = rd['docsignture'].replace('_', '\\_')
508: (4)                 rd['docstrsigns'] = []
509: (4)                 rd['latexdocstrsigns'] = []
510: (4)                 for k in ['docstrreq', 'docstrop', 'docstrout', 'docstrcbs']:
511: (8)                     if k in rd and isinstance(rd[k], list):
512: (12)                         rd['docstrsigns'] = rd['docstrsigns'] + rd[k]
513: (8)                         k = 'latex' + k
514: (8)                         if k in rd and isinstance(rd[k], list):
515: (12)                             rd['latexdocstrsigns'] = rd['latexdocstrsigns'] + rd[k][0:1] +
516: (16)                                 ['\\begin{description}'] + rd[k][1:] + \
517: (16)                                 ['\\end{description}']
518: (4)                 if 'args' not in rd:
519: (8)                     rd['args'] = ''
520: (8)                     rd['args_td'] = ''
521: (8)                     rd['args_nm'] = ''
522: (4)                 if not (rd.get('args') or rd.get('optargs') or rd.get('strarglens')):
523: (8)                     rd['noargs'] = 'void'
524: (4)                 ar = applyrules(cb_routine_rules, rd)
525: (4)                 cfuncs.callbacks[rd['name']] = ar['body']
526: (4)                 if isinstance(ar['need'], str):
527: (8)                     ar['need'] = [ar['need']]
528: (4)                 if 'need' in rd:
529: (8)                     for t in cfuncs.typedefs.keys():
530: (12)                         if t in rd['need']:
531: (16)                             ar['need'].append(t)
532: (4)                 cfuncs.typedefs_generated[rd['name'] + '_typedef'] = ar['cbtypedefs']
533: (4)                 ar['need'].append(rd['name'] + '_typedef')
534: (4)                 cfuncs.needs[rd['name']] = ar['need']
535: (4)                 capi_maps.lcb2_map[rd['name']] = {'maxnofargs': ar['maxnofargs'],
536: (38)                               'nofoptargs': ar['nofoptargs'],
537: (38)                               'docstr': ar['docstr'],
538: (38)                               'latexdocstr': ar['latexdocstr'],
539: (38)                               'argname': rd['argname']}
540: (38)
541: (4)                 outmess('      %s\n' % (ar['docstrshort']))
542: (4)             return

```

-----  
File 145 - cfuncs.py:

```

1: (0)             """
2: (0)             C declarations, CPP macros, and C functions for f2py2e.
3: (0)             Only required declarations/macros/functions will be used.
4: (0)             Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
5: (0)             Copyright 2011 -- present NumPy Developers.
6: (0)             Permission to use, modify, and distribute this software is given under the
7: (0)             terms of the NumPy License.
8: (0)             NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
9: (0)             """
10: (0)             import sys
11: (0)             import copy
12: (0)             from . import __version__
13: (0)             f2py_version = __version__.version
14: (0)             errmess = sys.stderr.write
15: (0)             outneeds = {'includes0': [], 'includes': [], 'typedefs': [],
16: (12)                           'userincludes': [],
17: (12)                           'cppmacros': [], 'cfuncs': [], 'callbacks': [], 'f90modhooks': [],
18: (12)                           'commonhooks': []}

```

```

19: (0)           needs = {}
20: (0)           includes0 = {'includes0': '/*need_includes0*/'}
21: (0)           includes = {'includes': '/*need_includes*/'}
22: (0)           userincludes = {'userincludes': '/*need_userincludes*/'}
23: (0)           typedefs = {'typedefs': '/*need_typedefs*/'}
24: (0)           typedefs_generated = {'typedefs_generated': '/*need_typedefs_generated*/'}
25: (0)           cppmacros = {'cppmacros': '/*need_cppmacros*/'}
26: (0)           cfuncs = {'cfuncs': '/*need_cfuncs*/'}
27: (0)           callbacks = {'callbacks': '/*need_callbacks*/'}
28: (0)           f90modhooks = {'f90modhooks': '/*need_f90modhooks*/',
29: (15)             'initf90modhooksstatic': '/*initf90modhooksstatic*/',
30: (15)             'initf90modhooksdynamic': '/*initf90modhooksdynamic*/',
31: (15)           }
32: (0)           commonhooks = {'commonhooks': '/*need_commonhooks*/',
33: (15)             'initcommonhooks': '/*need_initcommonhooks*/',
34: (15)           }
35: (0)           includes0['math.h'] = '#include <math.h>'
36: (0)           includes0['string.h'] = '#include <string.h>'
37: (0)           includes0['setjmp.h'] = '#include <setjmp.h>'
38: (0)           includes['arrayobject.h'] = '''#define PY_ARRAY_UNIQUE_SYMBOL PyArray_API
39: (0)           includes['npy_math.h'] = '#include "numpy/npy_math.h"'
40: (0)           includes['arrayobject.h'] = '#include "fortranobject.h"'
41: (0)           includes['stdarg.h'] = '#include <stdarg.h>'
42: (0)           typedefs['unsigned_char'] = 'typedef unsigned char unsigned_char;'
43: (0)           typedefs['unsigned_short'] = 'typedef unsigned short unsigned_short;'
44: (0)           typedefs['unsigned_long'] = 'typedef unsigned long unsigned_long;'
45: (0)           typedefs['signed_char'] = 'typedef signed char signed_char;'
46: (0)           typedefs['long_long'] = """
47: (0)             typedef __int64 long_long;
48: (0)             typedef long long long_long;
49: (0)             typedef unsigned long long unsigned_long_long;
50: (0)           """
51: (0)           typedefs['unsigned_long_long'] = """
52: (0)             typedef __uint64 long_long;
53: (0)             typedef unsigned long long unsigned_long_long;
54: (0)           """
55: (0)           typedefs['long_double'] = """
56: (0)             typedef long double long_double;
57: (0)           """
58: (0)           typedefs[
59: (4)             'complex_long_double'] = 'typedef struct {long double r,i;}'
complex_long_double;'           typedefs['complex_float'] = 'typedef struct {float r,i;} complex_float;'
60: (0)           typedefs['complex_double'] = 'typedef struct {double r,i;} complex_double;'
61: (0)           typedefs['string'] = """typedef char * string;"""
62: (0)           typedefs['character'] = """typedef char character;"""
63: (0)           cppmacros['CFUNCSMESS'] = """
64: (0)             PyObject_Print((PyObject *)obj,stderr,Py_PRINT_RAW);\\\
65: (4)               fprintf(stderr, "\\n\\");
66: (4)             """
67: (0)           """
68: (0)           cppmacros['F_FUNC'] = """
69: (0)             """
70: (0)           cppmacros['F_WRAPPEDFUNC'] = """
71: (0)             """
72: (0)           cppmacros['F_MODFUNC'] = """
73: (0)             /*
74: (0)             */
75: (0)             """
76: (0)           cppmacros['SWAPUNSAFE'] = """
77: (1)             (size_t)(b) = ((size_t)(a) ^ (size_t)(b));\\\
78: (1)             (size_t)(a) = ((size_t)(a) ^ (size_t)(b))
79: (0)           """
80: (0)           cppmacros['SWAP'] = """
81: (4)             t *c;\\\
82: (4)             c = a;\\\
83: (4)             a = b;\\\
84: (4)             b = c;}
85: (0)           """
86: (0)           cppmacros['PRINTPYOBJERR'] = """

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

87: (4)         fprintf(stderr, "#modulename#.error is related to ");\\
88: (4)         PyObject_Print((PyObject *)obj, stderr, Py_PRINT_RAW);\\
89: (4)         fprintf(stderr, "\\n");
90: (0)
91: (0)         cppmacros[ 'MINMAX' ] = """
92: (0)         """
93: (0)         cppmacros[ 'len..' ] = """
94: (0)         /* See fortranobject.h for definitions. The macros here are provided for BC.
*/
95: (0)
96: (0)         cppmacros[ 'pyobj_from_char1' ] = r"""
97: (0)         """
98: (0)         cppmacros[ 'pyobj_from_short1' ] = r"""
99: (0)         """
100: (0)        needs[ 'pyobj_from_int1' ] = ['signed_char']
101: (0)        cppmacros[ 'pyobj_from_int1' ] = r"""
102: (0)        """
103: (0)        cppmacros[ 'pyobj_from_long1' ] = r"""
104: (0)        """
105: (0)        needs[ 'pyobj_from_long_long1' ] = ['long_long']
106: (0)        cppmacros[ 'pyobj_from_long_long1' ] = """
107: (0)        """
108: (0)        needs[ 'pyobj_from_long_double1' ] = ['long_double']
109: (0)        cppmacros[ 'pyobj_from_long_double1' ] = """
110: (0)        cppmacros[ 'pyobj_from_double1' ] = """
111: (0)        cppmacros[ 'pyobj_from_float1' ] = """
112: (0)        needs[ 'pyobj_from_complex_long_double1' ] = ['complex_long_double']
113: (0)        cppmacros[ 'pyobj_from_complex_long_double1' ] = """
114: (0)        needs[ 'pyobj_from_complex_double1' ] = ['complex_double']
115: (0)        cppmacros[ 'pyobj_from_complex_double1' ] = """
116: (0)        needs[ 'pyobj_from_complex_float1' ] = ['complex_float']
117: (0)        cppmacros[ 'pyobj_from_complex_float1' ] = """
118: (0)        needs[ 'pyobj_from_string1' ] = ['string']
119: (0)        cppmacros[ 'pyobj_from_string1' ] = """
120: (0)        needs[ 'pyobj_from_string1size' ] = ['string']
121: (0)        cppmacros[ 'pyobj_from_string1size' ] = """
122: (0)        needs[ 'TRYPYARRAYTEMPLATE' ] = ['PRINTPYOBJERR']
123: (0)        cppmacros[ 'TRYPYARRAYTEMPLATE' ] = """
124: (0)        /* New SciPy */
125: (8)          PyArrayObject *arr = NULL;\\
126: (8)          if (!obj) return -2;\\
127: (8)          if (!PyArray_Check(obj)) return -1;\\
128: (8)          if (!(arr=(PyArrayObject *)obj))
{fprintf(stderr,"TRYPYARRAYTEMPLATE:");PRINTPYOBJERR(obj);return 0;}\\
129: (8)          if (PyArray_DESCR(arr)->type==typecode) {*(ctype *)
(PyArray_DATA(arr))=*v; return 1;}\\
130: (8)          switch (PyArray_TYPE(arr)) {
131: (16)            case NPY_DOUBLE: *(npy_double *) (PyArray_DATA(arr))=*v;
break;\\
132: (16)
133: (16)            case NPY_INT: *(npy_int *) (PyArray_DATA(arr))=*v; break;\\
134: (16)            case NPY_LONG: *(npy_long *) (PyArray_DATA(arr))=*v; break;\\
135: (16)            case NPY_FLOAT: *(npy_float *) (PyArray_DATA(arr))=*v; break;\\
break;\\
136: (16)
break;\\
137: (16)
break;\\
138: (16)
139: (16)            case NPY_UBYTE: *(npy_ubyte *) (PyArray_DATA(arr))=*v; break;\\
140: (16)            case NPY_BYTE: *(npy_byte *) (PyArray_DATA(arr))=*v; break;\\
141: (16)            case NPY_SHORT: *(npy_short *) (PyArray_DATA(arr))=*v; break;\\
break;\\
142: (16)
143: (16)            case NPY USHORT: *(npy_ushort *) (PyArray_DATA(arr))=*v;
144: (16)
break;\\
145: (16)
break;\\

```

```

146: (16) case NPY_LONGLONG: *(npy_longdouble *)
(PyArray_DATA(arr))=*v; break; \\
147: (16) case NPY_CLONGDOUBLE: *(npy_longdouble *)
(PyArray_DATA(arr))=*v; break; \\
148: (16) case NPY_OBJECT: PyArray_SETITEM(arr, PyArray_DATA(arr),
pyobj_from_ ## ctype ## 1(*v)); break; \\
149: (8) default: return -2; \\
150: (8) }; \\
151: (8) return 1
152: (0)
"""
153: (0) needs[ 'TRYCOMPLEXPYARRAYTEMPLATE' ] = [ 'PRINTPYOBJERR' ]
154: (0) cppmacros[ 'TRYCOMPLEXPYARRAYTEMPLATE' ] = """
155: (8)     PyArrayObject *arr = NULL; \\
156: (8)     if (!obj) return -2; \\
157: (8)     if (!PyArray_Check(obj)) return -1; \\
158: (8)     if (!(arr=(PyArrayObject *)obj))
{fprintf(stderr,"TRYCOMPLEXPYARRAYTEMPLATE:\");PRINTPYOBJERR(obj);return 0;} \\
159: (8)     if (PyArray_DESCR(arr)->type==typecode) { \\
160: (12)         *(ctype *) (PyArray_DATA(arr))=(*v).r; \\
161: (12)         *(ctype *) (PyArray_DATA(arr)+sizeof(ctype))=(*v).i; \\
162: (12)         return 1; \\
163: (8)     } \\
164: (8)     switch (PyArray_TYPE(arr)) { \\
165: (16)         case NPY_CDOUBLE: *(npy_double *) (PyArray_DATA(arr))=(*v).r; \\
166: (34)         *(npy_double *) \\
(PyArray_DATA(arr)+sizeof(npy_double))=(*v).i; \\
167: (34)         break; \\
168: (16)         case NPY_CFLOAT: *(npy_float *) (PyArray_DATA(arr))=(*v).r; \\
169: (33)         *(npy_float *) \\
(PyArray_DATA(arr)+sizeof(npy_float))=(*v).i; \\
170: (33)         break; \\
171: (16)         case NPY_DOUBLE: *(npy_double *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
172: (16)         case NPY_LONG: *(npy_long *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
173: (16)         case NPY_FLOAT: *(npy_float *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
174: (16)         case NPY_INT: *(npy_int *) (PyArray_DATA(arr))=(*v).r; break; \\
175: (16)         case NPY_SHORT: *(npy_short *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
176: (16)         case NPY_UBYTE: *(npy_ubyte *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
177: (16)         case NPY_BYTE: *(npy_byte *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
178: (16)         case NPY_BOOL: *(npy_bool *) (PyArray_DATA(arr))=((*v).r!=0 &&
(*v).i!=0); break; \\
179: (16)         case NPY USHORT: *(npy_ushort *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
180: (16)         case NPY_UINT: *(npy_uint *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
181: (16)         case NPY ULONG: *(npy_ulong *) (PyArray_DATA(arr))=(*v).r; \\
break; \\
182: (16)         case NPY_LONGLONG: *(npy_longlong *) (PyArray_DATA(arr))=
(*v).r; break; \\
183: (16)         case NPY_ULONGLONG: *(npy_ulonglong *) (PyArray_DATA(arr))=
(*v).r; break; \\
184: (16)         case NPY_LONGLONG: *(npy_longdouble *) (PyArray_DATA(arr))=
(*v).r; break; \\
185: (16)         case NPY_CLONGDOUBLE: *(npy_longdouble *) (PyArray_DATA(arr))=
(*v).r; \\
186: (38)         *(npy_longdouble *) \\
(PyArray_DATA(arr)+sizeof(npy_longdouble))=(*v).i; \\
187: (38)         break; \\
188: (16)         case NPY_OBJECT: PyArray_SETITEM(arr, PyArray_DATA(arr),
pyobj_from_complex_ ## ctype ## 1((*v))); break; \\
189: (16)         default: return -2; \\
190: (8)     }; \\
191: (8)     return -1;
192: (0)
"""

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

193: (0)
194: (0)
195: (8)
196: (8)
197: (12)
198: (8)
199: (12)
200: (12)
(len));\\
201: (8)
202: (12)
203: (12)
204: (12)
205: (8)
206: (4)
207: (0)
208: (0)
209: (8)
capi_fail;\\
210: (8)
211: (12)
212: (4)
213: (0)
214: (0)
215: (4)
216: (8)
217: (8)
218: (4)
219: (0)
220: (0)
221: (0)
222: (0)
223: (4)
224: (0)
225: (0)
226: (4)
227: (8)
228: (8)
229: (4)
230: (8)
231: (4)
232: (0)
233: (0)
234: (0)
235: (0)
236: (0)
237: (0)
/*
238: (0)
STRINGPADN replaces null values with padding values from the right.
`to` must have size of at least N bytes.
If the `to[N-1]` has null value, then replace it and all the
preceding, nulls with the given padding.
STRINGPADN(to, N, PADDING, NULLVALUE) is an inverse operation.
*/
244: (4)
do {
    int _m = (N);
    char *_to = (to);
    for (_m -= 1; _m >= 0 && _to[_m] == NULLVALUE; _m--) {
        _to[_m] = PADDING;
    }
} while (0)
"""
252: (0)
needs['STRINGCOPYN'] = ['string.h', 'FAILNULL']
cppmacros['STRINGCOPYN'] = """
/*
255: (0)
STRINGCOPYN copies N bytes.
`to` and `from` buffers must have sizes of at least N bytes.
*/
258: (4)
do {
    int _m = (N);

```

```

260: (8)             char *_to = (to);                                \\
261: (8)             char *_from = (from);                            \\
262: (8)             FAILNULL(_to); FAILNULL(_from);                  \\
263: (8)             (void)strncpy(_to, _from, _m);                   \\
264: (4)             } while (0)                                     \\
265: (0)             """
266: (0)             needs['STRINGCOPY'] = ['string.h', 'FAILNULL'] \\
267: (0)             cppmacros['STRINGCOPY'] = """
268: (4)             do { FAILNULL(to); FAILNULL(from); (void)strcpy(to,from); } while (0) \\
269: (0)             """
270: (0)             cppmacros['CHECKGENERIC'] = """
271: (4)             if (!(check)) {\\"\\
272: (8)                 PyErr_SetString(#modulename#error, \"(\"tcheck\") failed for \\
273: (8)                     /*goto capi_fail;*/\\
274: (4)                 } else """
275: (0)                 cppmacros['CHECKARRAY'] = """
276: (4)                 if (!(check)) {\\"\\
277: (8)                     PyErr_SetString(#modulename#error, \"(\"tcheck\") failed for \\
278: (8)                         /*goto capi_fail;*/\\
279: (4)                     } else """
280: (0)                     cppmacros['CHECKSTRING'] = """
281: (4)                     if (!(check)) {\\"\\
282: (8)                         char errstring[256];\\
283: (8)                         sprintf(errstring, \"%s: \"show, \"(\"tcheck\") failed for \"name,
slen(var), var);\\\
284: (8)                         PyErr_SetString(#modulename#error, errstring);\\\
285: (8)                         /*goto capi_fail;*/\\
286: (4)                     } else """
287: (0)                     cppmacros['CHECKSCALAR'] = """
288: (4)                     if (!(check)) {\\"\\
289: (8)                         char errstring[256];\\
290: (8)                         sprintf(errstring, \"%s: \"show, \"(\"tcheck\") failed for \"name,
var);\\\
291: (8)                         PyErr_SetString(#modulename#error, errstring);\\\
292: (8)                         /*goto capi_fail;*/\\
293: (4)                     } else """
294: (0)                     cppmacros[
295: (4)                         'ARRSIZE'] = '#define ARRSIZE(dims,rank)
(_PyArray_multiply_list(dims,rank))'
296: (0)                     cppmacros['OLDPYNUM'] = """
297: (0)                     """
298: (0)                     cppmacros["F2PY_THREAD_LOCAL_DECL"] = """
299: (6)                         && (__STDC_VERSION__ >= 201112L) \\
300: (6)                         && !defined(__STDC_NO_THREADS__)
301: (6)                         && (!defined(__GLIBC__) || __GLIBC__ > 2 || __GLIBC__ == 2 &&
__GLIBC_MINOR__ > 12)) \\
302: (6)                         && !defined(NPY_OS_OPENBSD) && !defined(NPY_OS_HAIKU)
303: (0)                     /* __STDC_NO_THREADS__ was first defined in a maintenance release of glibc
2.12,
304: (3)                         see https://lists.gnu.org/archive/html/commit-hurd/2012-07/msg00180.html,
305: (3)                         so `!defined(__STDC_NO_THREADS__)` may give false positive for the
existence
306: (3)                         of `threads.h` when using an older release of glibc 2.12
307: (3)                         See gh-19437 for details on OpenBSD */
308: (6)                         && (__GNUC__ > 4 || (__GNUC__ == 4 && (__GNUC_MINOR__ >= 4)))
309: (0)                     """
310: (0)                     cfuncs['calcarrindex'] = """
311: (0)                     static int calcarrindex(int *i,PyArrayObject *arr) {
312: (4)                         int k,ii = i[0];
313: (4)                         for (k=1; k < PyArray_NDIM(arr); k++)
314: (8)                             ii += (ii*(PyArray_DIM(arr,k) - 1)+i[k]); /* assuming contiguous arr
*/
315: (4)                         return ii;
316: (0)                     }"""
317: (0)                     cfuncs['calcarrindextr'] = """
318: (0)                     static int calcarrindextr(int *i,PyArrayObject *arr) {
319: (4)                         int k,ii = i[PyArray_NDIM(arr)-1];

```

```

320: (4)             for (k=1; k < PyArray_NDIM(arr); k++)
321: (8)                 ii += (ii*(PyArray_DIM(arr,PyArray_NDIM(arr)-k-1) -
1)+i[PyArray_NDIM(arr)-k-1]); /* assuming contiguous arr */
322: (4)             return ii;
323: (0)         }"""
324: (0)         cfuncs['forcomb'] = """
325: (0)         static struct { int nd;npy_intp *d;int *i,*i_tr,tr; } forcombcache;
326: (0)         static int initforcomb(npy_intp *dims,int nd,int tr) {
327: (2)             int k;
328: (2)             if (dims==NULL) return 0;
329: (2)             if (nd<0) return 0;
330: (2)             forcombcache.nd = nd;
331: (2)             forcombcache.d = dims;
332: (2)             forcombcache.tr = tr;
333: (2)             if ((forcombcache.i = (int *)malloc(sizeof(int)*nd))==NULL) return 0;
334: (2)             if ((forcombcache.i_tr = (int *)malloc(sizeof(int)*nd))==NULL) return 0;
335: (2)             for (k=1;k<nd;k++) {
336: (4)                 forcombcache.i[k] = forcombcache.i_tr[nd-k-1] = 0;
337: (2)             }
338: (2)             forcombcache.i[0] = forcombcache.i_tr[nd-1] = -1;
339: (2)             return 1;
340: (0)         }
341: (0)         static int *nextforcomb(void) {
342: (2)             int j,*i,*i_tr,k;
343: (2)             int nd=forcombcache.nd;
344: (2)             if ((i=forcombcache.i) == NULL) return NULL;
345: (2)             if ((i_tr=forcombcache.i_tr) == NULL) return NULL;
346: (2)             if (forcombcache.d == NULL) return NULL;
347: (2)             i[0]++;
348: (2)             if (i[0]==forcombcache.d[0]) {
349: (4)                 j=1;
350: (4)                 while ((j<nd) && (i[j]==forcombcache.d[j]-1)) j++;
351: (4)                 if (j==nd) {
352: (6)                     free(i);
353: (6)                     free(i_tr);
354: (6)                     return NULL;
355: (4)                 }
356: (4)                 for (k=0;k<j;k++) i[k] = i_tr[nd-k-1] = 0;
357: (4)                 i[j]++;
358: (4)                 i_tr[nd-j-1]++;
359: (2)             } else
360: (4)                 i_tr[nd-1]++;
361: (2)             if (forcombcache.tr) return i_tr;
362: (2)             return i;
363: (0)         }"""
364: (0)         needs['try_pyarr_from_string'] = ['STRINGCOPYN', 'PRINTPYOBJERR', 'string']
365: (0)         cfuncs['try_pyarr_from_string'] = """
366: (0) /*
367: (2)     try_pyarr_from_string copies str[:len(obj)] to the data of an `ndarray`.
368: (2)     If obj is an `ndarray`, it is assumed to be contiguous.
369: (2)     If the specified len== -1, str must be null-terminated.
370: (0) */
371: (0)         static int try_pyarr_from_string(PyObject *obj,
372: (33)                                     const string str, const int len) {
373: (0)             fprintf(stderr, "try_pyarr_from_string(str='%s', len=%d, obj=%p)\\n",
374: (8)                             (char*)str,len, obj);
375: (4)             if (!obj) return -2; /* Object missing */
376: (4)             if (obj == Py_None) return -1; /* None */
377: (4)             if (!PyArray_Check(obj)) goto capi_fail; /* not an ndarray */
378: (4)             if (PyArray_Check(obj)) {
379: (8)                 PyArrayObject *arr = (PyArrayObject *)obj;
380: (8)                 assert(ISCONTIGUOUS(arr));
381: (8)                 string buf = PyArray_DATA(arr);
382: (8)                 npy_intp n = len;
383: (8)                 if (n == -1) {
384: (12)                     /* Assuming null-terminated str. */
385: (12)                     n = strlen(str);
386: (8)                 }
387: (8)                 if (n > PyArray_NBYTES(arr)) {

```

```

388: (12)          n = PyArray_NBYTES(arr);
389: (8)          }
390: (8)          STRINGCOPYN(buf, str, n);
391: (8)          return 1;
392: (4)      }
393: (0)  capi_fail:
394: (4)      PRINTPYOBJERR(obj);
395: (4)      PyErr_SetString(#modulename#_error, \\"try_pyarr_from_string failed\\");
396: (4)      return 0;
397: (0)
398: (0)
399: (0)  needs['string_from_pyobj'] = ['string', 'STRINGMALLOC', 'STRINGCOPYN']
400: (0)  cfuncs['string_from_pyobj'] = """
401: (0) /*
402: (2)     Create a new string buffer `str` of at most length `len` from a
403: (2)     Python string-like object `obj`.
404: (2)     The string buffer has given size (len) or the size of inistr when len== -1.
405: (2)     The string buffer is padded with blanks: in Fortran, trailing blanks
406: (2)     are insignificant contrary to C nulls.
407: (1)
408: (0)
409: (0)  static int
410: (18)  string_from_pyobj(string *str, int *len, const string inistr, PyObject *obj,
411: (0)          const char *errmess)
412: (0)  {
413: (4)      PyObject *tmp = NULL;
414: (4)      string buf = NULL;
415: (4)      npy_intp n = -1;
416: (15)      fprintf(stderr,\\"string_from_pyobj(str=%s',len=%d,inistr=%s',obj=%p)\\n\",
417: (4)                  (char*)str, *len, (char *)inistr, obj);
418: (8)      if (obj == Py_None) {
419: (8)          n = strlen(inistr);
420: (8)          buf = inistr;
421: (4)
422: (8)      else if (PyArray_Check(obj)) {
423: (8)          PyArrayObject *arr = (PyArrayObject *)obj;
424: (12)          if (!ISCONTIGUOUS(arr)) {
425: (28)              PyErr_SetString(PyExc_ValueError,
426: (12)                  \\"array object is non-contiguous.\\");
427: (8)              goto capi_fail;
428: (8)
429: (8)          n = PyArray_NBYTES(arr);
430: (8)          buf = PyArray_DATA(arr);
431: (8)          n = strnlen(buf, n);
432: (4)
433: (8)      else {
434: (12)          if (PyBytes_Check(obj)) {
435: (12)              tmp = obj;
436: (8)              Py_INCREF(tmp);
437: (8)
438: (12)          else if (PyUnicode_Check(obj)) {
439: (8)              tmp = PyUnicode_AsASCIIString(obj);
440: (8)
441: (12)          else {
442: (12)              PyObject *tmp2;
443: (12)              tmp2 = PyObject_Str(obj);
444: (16)              if (tmp2) {
445: (16)                  tmp = PyUnicode_AsASCIIString(tmp2);
446: (12)                  Py_DECREF(tmp2);
447: (12)
448: (16)                  else {
449: (12)                      tmp = NULL;
450: (8)
451: (8)                  if (tmp == NULL) goto capi_fail;
452: (8)                  n = PyBytes_GET_SIZE(tmp);
453: (8)                  buf = PyBytes_AS_STRING(tmp);
454: (4)
455: (4)                  if (*len == -1) {
456: (8)                      /* TODO: change the type of `len` so that we can remove this */

```

```

457: (8)             if (n > NPY_MAX_INT) {
458: (12)             PyErr_SetString(PyExc_OverflowError,
459: (28)                         "object too large for a 32-bit int");
460: (12)             goto capi_fail;
461: (8)         }
462: (8)         *len = n;
463: (4)
464: (4)     else if (*len < n) {
465: (8)         /* discard the last (len-n) bytes of input buf */
466: (8)         n = *len;
467: (4)
468: (4)     if (n < 0 || *len < 0 || buf == NULL) {
469: (8)         goto capi_fail;
470: (4)
471: (4)     STRINGMALLOC(*str, *len); // *str is allocated with size (*len + 1)
472: (4)     if (n < *len) {
473: (8)         /*
474: (10)             Pad fixed-width string with nulls. The caller will replace
475: (10)             nulls with blanks when the corresponding argument is not
476: (10)             intent(c).
477: (8)         */
478: (8)         memset(*str + n, '\\0', *len - n);
479: (4)
480: (4)     STRINGCOPYN(*str, buf, n);
481: (4)     Py_XDECREF(tmp);
482: (4)     return 1;
483: (0)   capi_fail:
484: (4)     Py_XDECREF(tmp);
485: (4)   {
486: (8)       PyObject* err = PyErr_Occurred();
487: (8)       if (err == NULL) {
488: (12)           err = #modulename#_error;
489: (8)       }
490: (8)       PyErr_SetString(err, errmess);
491: (4)
492: (4)       return 0;
493: (0)
494: (0)
495: (0) """
496: (0) static int
497: (0) character_from_pyobj(character* v, PyObject *obj, const char *errmess) {
498: (4)     if (PyBytes_Check(obj)) {
499: (8)         /* empty bytes has trailing null, so dereferencing is always safe */
500: (8)         *v = PyBytes_AS_STRING(obj)[0];
501: (8)         return 1;
502: (4)     } else if (PyUnicode_Check(obj)) {
503: (8)         PyObject* tmp = PyUnicode_AsASCIIString(obj);
504: (8)         if (tmp != NULL) {
505: (12)             *v = PyBytes_AS_STRING(tmp)[0];
506: (12)             Py_DECREF(tmp);
507: (12)             return 1;
508: (8)
509: (4)     } else if (PyArray_Check(obj)) {
510: (8)         PyArrayObject* arr = (PyArrayObject*)obj;
511: (8)         if (F2PY_ARRAY_IS_CHARACTER_COMPATIBLE(arr)) {
512: (12)             *v = PyArray_BYTES(arr)[0];
513: (12)             return 1;
514: (8)         } else if (F2PY_IS_UNICODE_ARRAY(arr)) {
515: (12)             // TODO: update when numpy will support 1-byte and
516: (12)             // 2-byte unicode dtypes
517: (12)             PyObject* tmp = PyUnicode_FromKindAndData(
518: (30)                 PyUnicode_4BYTE_KIND,
519: (30)                 PyArray_BYTES(arr),
520: (30)                 (PyArray_NBYTES(arr)>0?1:0));
521: (12)             if (tmp != NULL) {
522: (16)                 if (character_from_pyobj(v, tmp, errmess)) {
523: (20)                     Py_DECREF(tmp);
524: (20)                     return 1;
525: (16)             }

```

```

526: (16)                                Py_DECREF(tmp);
527: (12)                                }
528: (8)                                 }
529: (4)  } else if (PySequence_Check(obj)) {
530: (8)      PyObject* tmp = PySequence_GetItem(obj,0);
531: (8)      if (tmp != NULL) {
532: (12)          if (character_from_pyobj(v, tmp, errmess)) {
533: (16)              Py_DECREF(tmp);
534: (16)              return 1;
535: (12)          }
536: (12)          Py_DECREF(tmp);
537: (8)      }
538: (4)  }
539: (4)  {
540: (8)      /* TODO: This error (and most other) error handling needs cleaning. */
541: (8)      char mess[F2PY_MESSAGE_BUFFER_SIZE];
542: (8)      strcpy(mess, errmess);
543: (8)      PyObject* err = PyErr_Occurred();
544: (8)      if (err == NULL) {
545: (12)          err = PyExc_TypeError;
546: (12)          Py_INCREF(err);
547: (8)      }
548: (8)      else {
549: (12)          Py_INCREF(err);
550: (12)          PyErr_Clear();
551: (8)      }
552: (8)      sprintf(mess + strlen(mess),
553: (16)          " -- expected str|bytes|sequence-of-str-or-bytes, got ");
554: (8)      f2py_describe(obj, mess + strlen(mess));
555: (8)      PyErr_SetString(err, mess);
556: (8)      Py_DECREF(err);
557: (4)  }
558: (4)  return 0;
559: (0)
560: (0)
561: (0) """
562: (0) needs['char_from_pyobj'] = ['int_from_pyobj']
563: (0) cfuncs['char_from_pyobj'] = """
564: (0) static int
565: (4) char_from_pyobj(char* v, PyObject *obj, const char *errmess) {
566: (4)     int i = 0;
567: (4)     if (int_from_pyobj(&i, obj, errmess)) {
568: (8)         *v = (char)i;
569: (8)         return 1;
570: (4)     }
571: (4)     return 0;
572: (0)
573: (0) """
574: (0) needs['signed_char_from_pyobj'] = ['int_from_pyobj', 'signed_char']
575: (0) cfuncs['signed_char_from_pyobj'] = """
576: (0) static int
577: (4) signed_char_from_pyobj(signed_char* v, PyObject *obj, const char *errmess) {
578: (4)     int i = 0;
579: (4)     if (int_from_pyobj(&i, obj, errmess)) {
580: (8)         *v = (signed_char)i;
581: (8)         return 1;
582: (4)     }
583: (4)     return 0;
584: (0)
585: (0) """
586: (0) needs['short_from_pyobj'] = ['int_from_pyobj']
587: (0) cfuncs['short_from_pyobj'] = """
588: (0) static int
589: (4) short_from_pyobj(short* v, PyObject *obj, const char *errmess) {
590: (4)     int i = 0;
591: (4)     if (int_from_pyobj(&i, obj, errmess)) {
592: (8)         *v = (short)i;
593: (8)         return 1;
594: (4)     }
595: (4)     return 0;

```

```

595: (0)
596: (0)
597: (0)
598: (0)
599: (0)
600: (0)
601: (4)
602: (4)
603: (8)
604: (8)
605: (4)
606: (4)
607: (4)
608: (8)
609: (8)
610: (8)
611: (4)
612: (4)
613: (8)
614: (8)
615: (4)
616: (4)
617: (8)
618: (4)
619: (4)
620: (8)
621: (8)
622: (4)
623: (4)
624: (8)
625: (12)
626: (12)
627: (8)
628: (8)
629: (4)
630: (4)
631: (8)
632: (8)
633: (12)
634: (8)
635: (8)
636: (4)
637: (4)
638: (0)
639: (0)
640: (0)
641: (0)
642: (0)
643: (4)
644: (4)
645: (8)
646: (8)
647: (4)
648: (4)
649: (4)
650: (8)
651: (8)
652: (8)
653: (4)
654: (4)
655: (8)
656: (8)
657: (4)
658: (4)
659: (8)
660: (4)
661: (4)
662: (8)
663: (8)

}
"""

cfuncs['int_from_pyobj'] = """
static int
int_from_pyobj(int* v, PyObject *obj, const char *errmess)
{
    PyObject* tmp = NULL;
    if (PyLong_Check(obj)) {
        *v = Npy__PyLong_AsInt(obj);
        return !(*v == -1 && PyErr_Occurred());
    }
    tmp = PyNumber_Long(obj);
    if (tmp) {
        *v = Npy__PyLong_AsInt(tmp);
        Py_DECREF(tmp);
        return !(*v == -1 && PyErr_Occurred());
    }
    if (PyComplex_Check(obj)) {
        PyErr_Clear();
        tmp = PyObject_GetAttrString(obj, "real");
    }
    else if (PyBytes_Check(obj) || PyUnicode_Check(obj)) {
        /*pass*/
    }
    else if (PySequence_Check(obj)) {
        PyErr_Clear();
        tmp = PySequence_GetItem(obj, 0);
    }
    if (tmp) {
        if (int_from_pyobj(v, tmp, errmess)) {
            Py_DECREF(tmp);
            return 1;
        }
        Py_DECREF(tmp);
    }
    PyObject* err = PyErr_Occurred();
    if (err == NULL) {
        err = #modulename#_error;
    }
    PyErr_SetString(err, errmess);
}
return 0;
}

"""

cfuncs['long_from_pyobj'] = """
static int
long_from_pyobj(long* v, PyObject *obj, const char *errmess) {
    PyObject* tmp = NULL;
    if (PyLong_Check(obj)) {
        *v = PyLong_AsLong(obj);
        return !(*v == -1 && PyErr_Occurred());
    }
    tmp = PyNumber_Long(obj);
    if (tmp) {
        *v = PyLong_AsLong(tmp);
        Py_DECREF(tmp);
        return !(*v == -1 && PyErr_Occurred());
    }
    if (PyComplex_Check(obj)) {
        PyErr_Clear();
        tmp = PyObject_GetAttrString(obj, "real");
    }
    else if (PyBytes_Check(obj) || PyUnicode_Check(obj)) {
        /*pass*/
    }
    else if (PySequence_Check(obj)) {
        PyErr_Clear();
        tmp = PySequence_GetItem(obj, 0);
    }

```

```

664: (4)
665: (4)
666: (8)
667: (12)
668: (12)
669: (8)
670: (8)
671: (4)
672: (4)
673: {
674:     if (tmp) {
675:         if (long_long_from_pyobj(v, tmp, errmess)) {
676:             Py_DECREF(tmp);
677:             return 1;
678:         }
679:     }
680:     return 0;
681: }
682: """
683: needs['long_long_from_pyobj'] = ['long_long']
684: cfuncs['long_long_from_pyobj'] = """
685: static int
686: long_long_from_pyobj(long_long* v, PyObject *obj, const char *errmess)
687: {
688:     PyObject* tmp = NULL;
689:     if (PyLong_Check(obj)) {
690:         *v = PyLong_AsLongLong(obj);
691:         return !(*v == -1 && PyErr_Occurred());
692:     }
693:     tmp = PyNumber_Long(obj);
694:     if (tmp) {
695:         *v = PyLong_AsLongLong(tmp);
696:         Py_DECREF(tmp);
697:         return !(*v == -1 && PyErr_Occurred());
698:     }
699:     if (PyComplex_Check(obj)) {
700:         PyErr_Clear();
701:         tmp = PyObject_GetAttrString(obj, "real");
702:     }
703:     else if (PyBytes_Check(obj) || PyUnicode_Check(obj)) {
704:         /*pass*/
705:     }
706:     else if (PySequence_Check(obj)) {
707:         PyErr_Clear();
708:         tmp = PySequence_GetItem(obj, 0);
709:     }
710:     if (tmp) {
711:         if (long_long_from_pyobj(v, tmp, errmess)) {
712:             Py_DECREF(tmp);
713:             return 1;
714:         }
715:     }
716: }
717: """
718: needs['long_double_from_pyobj'] = ['double_from_pyobj', 'long_double']
719: cfuncs['long_double_from_pyobj'] = """
720: static int
721: long_double_from_pyobj(long_double* v, PyObject *obj, const char *errmess)
722: {
723:     double d=0;
724:     if (PyArray_CheckScalar(obj)){
725: """
726: needs['long_double_from_pyobj'] = ['double_from_pyobj', 'long_double']
727: cfuncs['long_double_from_pyobj'] = """
728: static int
729: long_double_from_pyobj(long_double* v, PyObject *obj, const char *errmess)
730: {
731:     double d=0;
732:     if (PyArray_CheckScalar(obj)){

```

```

733: (8)             if PyArray_IsScalar(obj, LongDouble) {
734: (12)             PyArray_ScalarAsCtype(obj, v);
735: (12)             return 1;
736: (8)
737: (8)             }
738: (12)             else if (PyArray_Check(obj) && PyArray_TYPE(obj) == NPY_LONGLONG) {
739: (12)                 (*v) = *((npy_longlong *)PyArray_DATA(obj));
740: (8)                 return 1;
741: (4)
742: (4)             }
743: (8)             if (double_from_pyobj(&d, obj, errmess)) {
744: (8)                 *v = (long_double)d;
745: (4)                 return 1;
746: (4)
747: (0)             }
748: (0)
749: (0)             """
750: (0)             static int
751: (0)             double_from_pyobj(double* v, PyObject *obj, const char *errmess)
752: (0)             {
753: (4)                 PyObject* tmp = NULL;
754: (4)                 if (PyFloat_Check(obj)) {
755: (8)                     *v = PyFloat_AsDouble(obj);
756: (8)                     return !(*v == -1.0 && PyErr_Occurred());
757: (4)
758: (4)                 tmp = PyNumber_Float(obj);
759: (4)                 if (tmp) {
760: (8)                     *v = PyFloat_AsDouble(tmp);
761: (8)                     Py_DECREF(tmp);
762: (8)                     return !(*v == -1.0 && PyErr_Occurred());
763: (4)
764: (4)                 if (PyComplex_Check(obj)) {
765: (8)                     PyErr_Clear();
766: (8)                     tmp = PyObject_GetAttrString(obj, \"real\");
767: (4)
768: (4)                 else if (PyBytes_Check(obj) || PyUnicode_Check(obj)) {
769: (8)                     /*pass*/
770: (4)
771: (4)                 else if (PySequence_Check(obj)) {
772: (8)                     PyErr_Clear();
773: (8)                     tmp = PySequence_GetItem(obj, 0);
774: (4)
775: (4)                 if (tmp) {
776: (8)                     if (double_from_pyobj(v, tmp, errmess)) {Py_DECREF(tmp); return 1;}
777: (8)                     Py_DECREF(tmp);
778: (4)
779: (4)
780: (8)                 PyObject* err = PyErr_Occurred();
781: (8)                 if (err==NULL) err = #modulename#_error;
782: (8)                 PyErr_SetString(err,errmess);
783: (4)
784: (4)
785: (0)             return 0;
786: (0)
787: (0)             """
788: (0)             needs['float_from_pyobj'] = ['double_from_pyobj']
789: (0)             cfuncs['float_from_pyobj'] = """
790: (0)             static int
791: (0)             float_from_pyobj(float* v, PyObject *obj, const char *errmess)
792: (4)             {
793: (4)                 double d=0.0;
794: (4)                 if (double_from_pyobj(&d,obj,errmess)) {
795: (8)                     *v = (float)d;
796: (8)                     return 1;
797: (4)
798: (4)                 return 0;
799: (0)
799: (0)             """
800: (0)             needs['complex_long_double_from_pyobj'] = ['complex_long_double',
'long_double'],

```

```

801: (43)                                'complex_double_from_pyobj',
'numpy_math.h']
802: (0)                                 cfuncs['complex_long_double_from_pyobj'] = """
803: (0)                                 static int
804: (0)                                 complex_long_double_from_pyobj(complex_long_double* v, PyObject *obj, const
char *errmess)
805: (0)                                 {
806: (4)                                   complex_double cd = {0.0,0.0};
807: (4)                                   if (PyArray_CheckScalar(obj)){
808: (8)                                     if PyArray_IsScalar(obj, CLongDouble) {
809: (12)                                       PyArray_ScalarAsCtype(obj, v);
810: (12)                                       return 1;
811: (8)                                     }
812: (8)                                   else if (PyArray_Check(obj) && PyArray_TYPE(obj)==NPY_CLONGDOUBLE) {
813: (12)                                     (*v).r = npy_creal((*(((npy_clongdouble *)PyArray_DATA(obj)))); 
814: (12)                                     (*v).i = npy_cimagl((*(((npy_clongdouble *)PyArray_DATA(obj)))); 
815: (12)                                     return 1;
816: (8)                                   }
817: (4)                                 }
818: (4)                                 if (complex_double_from_pyobj(&cd,obj,errmess)) {
819: (8)                                   (*v).r = (long_double)cd.r;
820: (8)                                   (*v).i = (long_double)cd.i;
821: (8)                                   return 1;
822: (4)                                 }
823: (4)                                 return 0;
824: (0)                               }
825: (0)                               """
needs['complex_double_from_pyobj'] = ['complex_double', 'numpy_math.h']
cfuncs['complex_double_from_pyobj'] = """
static int
complex_double_from_pyobj(complex_double* v, PyObject *obj, const char
*errmess) {
830: (4)                           Py_complex c;
831: (4)                           if (PyComplex_Check(obj)) {
832: (8)                             c = PyComplex_AsCComplex(obj);
833: (8)                             (*v).r = c.real;
834: (8)                             (*v).i = c.imag;
835: (8)                             return 1;
836: (4)                           }
837: (4)                           if (PyArray_IsScalar(obj, ComplexFloating)) {
838: (8)                             if (PyArray_IsScalar(obj, CFLOAT)) {
839: (12)                               npy_cfloat new;
840: (12)                               PyArray_ScalarAsCtype(obj, &new);
841: (12)                               (*v).r = (double)npy_crealf(new);
842: (12)                               (*v).i = (double)npy_cimagf(new);
843: (8)                             }
844: (8)                           else if (PyArray_IsScalar(obj, CLongDouble)) {
845: (12)                             npy_clongdouble new;
846: (12)                             PyArray_ScalarAsCtype(obj, &new);
847: (12)                             (*v).r = (double)npy_creal(new);
848: (12)                             (*v).i = (double)npy_cimagl(new);
849: (8)                           }
850: (8)                           else { /* if (PyArray_IsScalar(obj, CDouble)) */
851: (12)                             PyArray_ScalarAsCtype(obj, v);
852: (8)                           }
853: (8)                           return 1;
854: (4)                         }
855: (4)                         if (PyArray_CheckScalar(obj)) { /* 0-dim array or still array scalar */
856: (8)                           PyArrayObject *arr;
857: (8)                           if (PyArray_Check(obj)) {
858: (12)                             arr = (PyArrayObject *)PyArray_Cast((PyArrayObject *)obj,
NPY_CDOUBLE);
859: (8)                           }
860: (8)                           else {
861: (12)                             arr = (PyArrayObject *)PyArray_FromScalar(obj,
PyArray_DescrFromType(NPY_CDOUBLE));
862: (8)                           }
863: (8)                           if (arr == NULL) {
864: (12)                             return 0;

```

```

865: (8) }
866: (8) (*v).r = npy_creal(*(((npy_cdouble *)PyArray_DATA(arr)))); 
867: (8) (*v).i = npy_cimag(*(((npy_cdouble *)PyArray_DATA(arr)))); 
868: (8) Py_DECREF(arr);
869: (8) return 1;
870: (4) }
871: (4) /* Python does not provide PyNumber_Complex function :-( */
872: (4) (*v).i = 0.0;
873: (4) if (PyFloat_Check(obj)) {
874: (8) (*v).r = PyFloat_AsDouble(obj);
875: (8) return !((*v).r == -1.0 && PyErr_Occurred());
876: (4) }
877: (4) if (PyLong_Check(obj)) {
878: (8) (*v).r = PyLong_AsDouble(obj);
879: (8) return !((*v).r == -1.0 && PyErr_Occurred());
880: (4) }
881: (4) if (PySequence_Check(obj) && !(PyBytes_Check(obj) ||
PyUnicode_Check(obj))) {
882: (8) PyObject *tmp = PySequence_GetItem(obj,0);
883: (8) if (tmp) {
884: (12) if (complex_double_from_pyobj(v,tmp,errmess)) {
885: (16) Py_DECREF(tmp);
886: (16) return 1;
887: (12) }
888: (12) Py_DECREF(tmp);
889: (8) }
890: (4) }
891: (4) {
892: (8) PyObject* err = PyErr_Occurred();
893: (8) if (err==NULL)
894: (12) err = PyExc_TypeError;
895: (8) PyErr_SetString(err,errmess);
896: (4) }
897: (4) return 0;
898: (0) }
899: (0) """
900: (0) needs['complex_float_from_pyobj'] = [
901: (4) 'complex_float', 'complex_double_from_pyobj']
902: (0) cfuncs['complex_float_from_pyobj'] = """
903: (0) static int
904: (0) complex_float_from_pyobj(complex_float* v,PyObject *obj,const char *errmess)
905: (0) {
906: (4) complex_double cd={0.0,0.0};
907: (4) if (complex_double_from_pyobj(&cd,obj,errmess)) {
908: (8) (*v).r = (float)cd.r;
909: (8) (*v).i = (float)cd.i;
910: (8) return 1;
911: (4) }
912: (4) return 0;
913: (0) }
914: (0) """
915: (0) cfuncs['try_pyarr_from_character'] = """
916: (0) static int try_pyarr_from_character(PyObject* obj, character* v) {
917: (4) PyArrayObject *arr = (PyArrayObject*)obj;
918: (4) if (!obj) return -2;
919: (4) if (PyArray_Check(obj)) {
920: (8) if (F2PY_ARRAY_IS_CHARACTER_COMPATIBLE(arr)) {
921: (12) *(character *)PyArray_DATA(arr)) = *v;
922: (12) return 1;
923: (8) }
924: (4) }
925: (4) {
926: (8) char mess[F2PY_MESSAGE_BUFFER_SIZE];
927: (8) PyObject* err = PyErr_Occurred();
928: (8) if (err == NULL) {
929: (12) err = PyExc_ValueError;
930: (12) strcpy(mess, "try_pyarr_from_character failed"
931: (25) " -- expected bytes array-scalar|array, got ");
932: (12) f2py_describe(obj, mess + strlen(mess));

```

```

333: (12)           PyErr_SetString(err, mess);
934: (8)
935: (4)
936: (4)
937: (0)
938: (0)
939: (0)          needs['try_pyarr_from_char'] = ['pyobj_from_char1', 'TRYPYARRAYTEMPLATE']
940: (0)          cfuncs[
941: (4)            'try_pyarr_from_char'] = 'static int try_pyarr_from_char(PyObject*
obj,char* v) {\n    TRYPYARRAYTEMPLATE(char,\'c\');\n}\n'
942: (0)          needs['try_pyarr_from_signed_char'] = ['TRYPYARRAYTEMPLATE', 'unsigned_char']
943: (0)          cfuncs[
944: (4)            'try_pyarr_from_unsigned_char'] = 'static int
try_pyarr_from_unsigned_char(PyObject* obj,unsigned_char* v) {\n    TRYPYARRAYTEMPLATE(unsigned_char,\'b\');\n}\n'
945: (0)          needs['try_pyarr_from_signed_char'] = ['TRYPYARRAYTEMPLATE', 'signed_char']
946: (0)          cfuncs[
947: (4)            'try_pyarr_from_signed_char'] = 'static int
try_pyarr_from_signed_char(PyObject* obj,signed_char* v) {\n    TRYPYARRAYTEMPLATE(signed_char,\'1\');\n}\n'
948: (0)          needs['try_pyarr_from_short'] = ['pyobj_from_short1', 'TRYPYARRAYTEMPLATE']
949: (0)          cfuncs[
950: (4)            'try_pyarr_from_short'] = 'static int try_pyarr_from_short(PyObject*
obj,short* v) {\n    TRYPYARRAYTEMPLATE(short,\'s\');\n}\n'
951: (0)          needs['try_pyarr_from_int'] = ['pyobj_from_int1', 'TRYPYARRAYTEMPLATE']
952: (0)          cfuncs[
953: (4)            'try_pyarr_from_int'] = 'static int try_pyarr_from_int(PyObject* obj,int*
v) {\n    TRYPYARRAYTEMPLATE(int,\'i\');\n}\n'
954: (0)          needs['try_pyarr_from_long'] = ['pyobj_from_long1', 'TRYPYARRAYTEMPLATE']
955: (0)          cfuncs[
956: (4)            'try_pyarr_from_long'] = 'static int try_pyarr_from_long(PyObject*
obj,long* v) {\n    TRYPYARRAYTEMPLATE(long,\'l\');\n}\n'
957: (0)          needs['try_pyarr_from_long_long'] = [
958: (4)            'pyobj_from_long_long1', 'TRYPYARRAYTEMPLATE', 'long_long']
959: (0)          cfuncs[
960: (4)            'try_pyarr_from_long_long'] = 'static int
try_pyarr_from_long_long(PyObject* obj,long_long* v) {\n    TRYPYARRAYTEMPLATE(long_long,\'L\');\n}\n'
961: (0)          needs['try_pyarr_from_float'] = ['pyobj_from_float1', 'TRYPYARRAYTEMPLATE']
962: (0)          cfuncs[
963: (4)            'try_pyarr_from_float'] = 'static int try_pyarr_from_float(PyObject*
obj,float* v) {\n    TRYPYARRAYTEMPLATE(float,\'f\');\n}\n'
964: (0)          needs['try_pyarr_from_double'] = ['pyobj_from_double1', 'TRYPYARRAYTEMPLATE']
965: (0)          cfuncs[
966: (4)            'try_pyarr_from_double'] = 'static int try_pyarr_from_double(PyObject*
obj,double* v) {\n    TRYPYARRAYTEMPLATE(double,\'d\');\n}\n'
967: (0)          needs['try_pyarr_from_complex_float'] = [
968: (4)            'pyobj_from_complex_float1', 'TRYCOMPLEXPYARRAYTEMPLATE', 'complex_float']
969: (0)          cfuncs[
970: (4)            'try_pyarr_from_complex_float'] = 'static int
try_pyarr_from_complex_float(PyObject* obj,complex_float* v) {\n    TRYCOMPLEXPYARRAYTEMPLATE(float,\'F\');\n}\n'
971: (0)          needs['try_pyarr_from_complex_double'] = [
972: (4)            'pyobj_from_complex_double1', 'TRYCOMPLEXPYARRAYTEMPLATE',
'complex_double']
973: (0)          cfuncs[
974: (4)            'try_pyarr_from_complex_double'] = 'static int
try_pyarr_from_complex_double(PyObject* obj,complex_double* v) {\n    TRYCOMPLEXPYARRAYTEMPLATE(double,\'D\');\n}\n'
975: (0)          needs['create_cb_arglist'] = ['CFUNCSMESS', 'PRINTPYOBJERR', 'MINMAX']
976: (0)          cfuncs['create_cb_arglist'] = """
977: (0)          static int
978: (0)            create_cb_arglist(PyObject* fun, PyTupleObject* xa , const int maxnofargs,
979: (18)                  const int nooftargs, int *nofargs, PyTupleObject **args,
980: (18)                  const char *errmess)
981: (0)
982: (4)            PyObject *tmp = NULL;
983: (4)            PyObject *tmp_fun = NULL;
984: (4)            Py_ssize_t tot, opt, ext, siz, i, di = 0;

```

```

985: (4) CFUNCSMESS(\"create_cb_arglist\\n\");
986: (4) tot=opt=ext=siz=0;
987: (4) /* Get the total number of arguments */
988: (4) if (PyFunction_Check(fun)) {
989: (8)     tmp_fun = fun;
990: (8)     Py_INCREF(tmp_fun);
991: (4) }
992: (4) else {
993: (8)     di = 1;
994: (8)     if (PyObject_HasAttrString(fun,\"im_func\")) {
995: (12)         tmp_fun = PyObject_GetAttrString(fun,\"im_func\");
996: (8)     }
997: (8)     else if (PyObject_HasAttrString(fun,\"__call__\")) {
998: (12)         tmp = PyObject_GetAttrString(fun,\"__call__\");
999: (12)         if (PyObject_HasAttrString(tmp,\"im_func\")) {
1000: (16)             tmp_fun = PyObject_GetAttrString(tmp,\"im_func\");
1001: (12)         }
1002: (16)         else {
1003: (16)             tmp_fun = fun; /* built-in function */
1004: (16)             Py_INCREF(tmp_fun);
1005: (16)             tot = maxnofargs;
1006: (20)             if (PyCFunction_Check(fun)) {
1007: (20)                 /* In case the function has a co_argcount (like on PyPy)
1008: (16) */
1009: (16)                 di = 0;
1010: (20)             }
1011: (12)             if (xa != NULL)
1012: (12)                 tot += PyTuple_Size((PyObject *)xa);
1013: (8)         }
1014: (8)     else if (PyFortran_Check(fun) || PyFortran_Check1(fun)) {
1015: (12)         tot = maxnofargs;
1016: (12)         if (xa != NULL)
1017: (16)             tot += PyTuple_Size((PyObject *)xa);
1018: (12)         tmp_fun = fun;
1019: (12)         Py_INCREF(tmp_fun);
1020: (8)     }
1021: (8)     else if (F2PyCapsule_Check(fun)) {
1022: (12)         tot = maxnofargs;
1023: (12)         if (xa != NULL)
1024: (16)             ext = PyTuple_Size((PyObject *)xa);
1025: (12)         if(ext>0) {
1026: (16)             fprintf(stderr,\"extra arguments tuple cannot be used with
PyCapsule call-back\\n\");
1027: (16)             goto capi_fail;
1028: (12)         }
1029: (12)         tmp_fun = fun;
1030: (12)         Py_INCREF(tmp_fun);
1031: (8)     }
1032: (4) }
1033: (4)     if (tmp_fun == NULL) {
1034: (8)         fprintf(stderr,
1035: (16)             \"Call-back argument must be
function|instance|instance.__call__|f2py-function \
1036: (16)             \"but got %.\\n\",
1037: (16)             ((fun == NULL) ? \"NULL\" : Py_TYPE(fun)->tp_name));
1038: (8)         goto capi_fail;
1039: (4)     }
1040: (4)     if (PyObject_HasAttrString(tmp_fun,\"__code__\")) {
1041: (8)         if (PyObject_HasAttrString(tmp =
PyObject_GetAttrString(tmp_fun,\"__code__\"),\"co_argcount\")) {
1042: (12)             PyObject *tmp_argcount =
PyObject_GetAttrString(tmp,\"co_argcount\");
1043: (12)             Py_DECREF(tmp);
1044: (12)             if (tmp_argcount == NULL) {
1045: (16)                 goto capi_fail;
1046: (12)             }
1047: (12)             tot = PyLong_AsSsize_t(tmp_argcount) - di;
1048: (12)             Py_DECREF(tmp_argcount);

```

```

1049: (8)             }
1050: (4)         }
1051: (4)         /* Get the number of optional arguments */
1052: (4)         if (PyObject_HasAttrString(tmp_fun, "__defaults__")) {
1053: (8)             if (PyTuple_Check(tmp =
Py0bject_GetAttrString(tmp_fun, "__defaults__")))
1054: (12)                 opt = PyTuple_Size(tmp);
1055: (8)             Py_XDECREF(tmp);
1056: (4)         }
1057: (4)         /* Get the number of extra arguments */
1058: (4)         if (xa != NULL)
1059: (8)             ext = PyTuple_Size((PyObject *)xa);
1060: (4)         /* Calculate the size of call-backs argument list */
1061: (4)         siz = MIN(maxnofargs+ext,tot);
1062: (4)         *nofargs = MAX(0,siz-ext);
1063: (4)         fprintf(stderr,
1064: (12)             "\debug-capi:create_cb_arglist:maxnofargs(-nofoptargs),\""
1065: (12)             "\"tot,opt,ext,siz,nofargs = %d(-%d), %zd, %zd, %zd, %d\\n\",
1066: (12)             maxnofargs, nofoptargs, tot, opt, ext, siz, *nofargs);
1067: (4)         if (siz < tot-opt) {
1068: (8)             fprintf(stderr,
1069: (16)                 "create_cb_arglist: Failed to build argument list \
1070: (16)                 "(siz) with enough arguments (tot-opt) required by \
1071: (16)                 "user-supplied function (siz,tot,opt=%zd, %zd, %zd).\\n\",
1072: (16)                 siz, tot, opt);
1073: (8)             goto capi_fail;
1074: (4)         }
1075: (4)         /* Initialize argument list */
1076: (4)         *args = (PyTupleObject *)PyTuple_New(siz);
1077: (4)         for (i=0;i<*nofargs;i++) {
1078: (8)             Py_INCREF(Py_None);
1079: (8)             PyTuple_SET_ITEM((PyObject *)*args,i,Py_None);
1080: (4)         }
1081: (4)         if (xa != NULL)
1082: (8)             for (i=(*nofargs);i<siz;i++) {
1083: (12)                 tmp = PyTuple_GetItem((PyObject *)xa,i-(*nofargs));
1084: (12)                 Py_INCREF(tmp);
1085: (12)                 PyTuple_SET_ITEM(*args,i,tmp);
1086: (8)             }
1087: (4)             CFUNCSMESS(\"create_cb_arglist-end\\n\");
1088: (4)             Py_DECREF(tmp_fun);
1089: (4)             return 1;
1090: (0)         capi_fail:
1091: (4)             if (PyErr_Occurred() == NULL)
1092: (8)                 PyErr_SetString(#modulename#_error, errmess);
1093: (4)             Py_XDECREF(tmp_fun);
1094: (4)             return 0;
1095: (0)     }
1096: (0)     """
1097: (0)     def buildcfuncs():
1098: (4)         from .capi_maps import c2capi_map
1099: (4)         for k in c2capi_map.keys():
1100: (8)             m = 'pyarr_from_p_%s1' % k
1101: (8)             cppmacros[
1102: (12)                 m] = '#define %s(v) (PyArray_SimpleNewFromData(0,NULL,%s,(char
*)v))' % (m, c2capi_map[k])
1103: (4)                 k = 'string'
1104: (4)                 m = 'pyarr_from_p_%s1' % k
1105: (4)                 cppmacros[
1106: (8)                     m] = '#define %s(v,dims) (PyArray_New(&PyArray_Type, 1, dims,
NPY_STRING, NULL, v, 1, NPY_ARRAY_CARRAY, NULL))' % (m)
1107: (0)             def append_needs(need, flag=1):
1108: (4)                 if isinstance(need, list):
1109: (8)                     for n in need:
1110: (12)                         append_needs(n, flag)
1111: (4)                     elif isinstance(need, str):
1112: (8)                         if not need:
1113: (12)                             return
1114: (8)                         if need in includes0:

```

```

1115: (12)             n = 'includes0'
1116: (8)              elif need in includes:
1117: (12)                n = 'includes'
1118: (8)              elif need in typedefs:
1119: (12)                n = 'typedefs'
1120: (8)              elif need in typedefs_generated:
1121: (12)                n = 'typedefs_generated'
1122: (8)              elif need in cppmacros:
1123: (12)                n = 'cppmacros'
1124: (8)              elif need in cfuncs:
1125: (12)                n = 'cfuncs'
1126: (8)              elif need in callbacks:
1127: (12)                n = 'callbacks'
1128: (8)              elif need in f90modhooks:
1129: (12)                n = 'f90modhooks'
1130: (8)              elif need in commonhooks:
1131: (12)                n = 'commonhooks'
1132: (8)            else:
1133: (12)              errmess('append_needs: unknown need %s\n' % (repr(need)))
1134: (12)              return
1135: (8)            if need in outneeds[n]:
1136: (12)              return
1137: (8)            if flag:
1138: (12)              tmp = {}
1139: (12)              if need in needs:
1140: (16)                for nn in needs[need]:
1141: (20)                  t = append_needs(nn, 0)
1142: (20)                  if isinstance(t, dict):
1143: (24)                    for nnn in t.keys():
1144: (28)                      if nnn in tmp:
1145: (32)                        tmp[nnn] = tmp[nnn] + t[nnn]
1146: (28)                      else:
1147: (32)                        tmp[nnn] = t[nnn]
1148: (12)                for nn in tmp.keys():
1149: (16)                  for nnn in tmp[nn]:
1150: (20)                      if nnn not in outneeds[nn]:
1151: (24)                        outneeds[nn] = [nnn] + outneeds[nn]
1152: (12)                      outneeds[n].append(need)
1153: (8)            else:
1154: (12)              tmp = {}
1155: (12)              if need in needs:
1156: (16)                for nn in needs[need]:
1157: (20)                  t = append_needs(nn, flag)
1158: (20)                  if isinstance(t, dict):
1159: (24)                    for nnn in t.keys():
1160: (28)                      if nnn in tmp:
1161: (32)                        tmp[nnn] = t[nnn] + tmp[nnn]
1162: (28)                      else:
1163: (32)                        tmp[nnn] = t[nnn]
1164: (12)                  if n not in tmp:
1165: (16)                    tmp[n] = []
1166: (12)                    tmp[n].append(need)
1167: (12)                    return tmp
1168: (4)            else:
1169: (8)              errmess('append_needs: expected list or string but got :%s\n' %
1170: (16)                            (repr(need)))
1171: (0)            def get_needs():
1172: (4)              res = {}
1173: (4)              for n in outneeds.keys():
1174: (8)                out = []
1175: (8)                saveout = copy.copy(outneeds[n])
1176: (8)                while len(outneeds[n]) > 0:
1177: (12)                  if outneeds[n][0] not in needs:
1178: (16)                    out.append(outneeds[n][0])
1179: (16)                    del outneeds[n][0]
1180: (12)                  else:
1181: (16)                    flag = 0
1182: (16)                    for k in outneeds[n][1:]:
1183: (20)                      if k in needs[outneeds[n][0]]:
```

```

1184: (24)                     flag = 1
1185: (24)                     break
1186: (16)                     if flag:
1187: (20)                         outneeds[n] = outneeds[n][1:] + [outneeds[n][0]]
1188: (16)                     else:
1189: (20)                         out.append(outneeds[n][0])
1190: (20)                         del outneeds[n][0]
1191: (12)                     if saveout and (0 not in map(lambda x, y: x == y, saveout,
outneeds[n])) \
1192: (20)                         and outneeds[n] != []:
1193: (16)                         print(n, saveout)
1194: (16)                         errmess(
1195: (20)                             'get_needs: no progress in sorting needs, probably
circular dependence, skipping.\n')
1196: (16)                         out = out + saveout
1197: (16)                         break
1198: (12)                     saveout = copy.copy(outneeds[n])
1199: (8)                     if out == []:
1200: (12)                         out = [n]
1201: (8)                         res[n] = out
1202: (4)                     return res

```

---

## File 146 - common\_rules.py:

```

1: (0)             """
2: (0)             Build common block mechanism for f2py2e.
3: (0)             Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
4: (0)             Copyright 2011 -- present NumPy Developers.
5: (0)             Permission to use, modify, and distribute this software is given under the
6: (0)             terms of the NumPy License
7: (0)             NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
8: (0)             """
9: (0)             from . import __version__
10: (0)             f2py_version = __version__.version
11: (0)             from .auxfuncs import (
12: (4)                 hasbody, hascommon, hasnote, isintent_hide, outmess, getuseblocks
13: (0)             )
14: (0)             from . import capi_maps
15: (0)             from . import func2subr
16: (0)             from .crackfortran import rmbadname
17: (0)             def findcommonblocks(block, top=1):
18: (4)                 ret = []
19: (4)                 if hascommon(block):
20: (8)                     for key, value in block['common'].items():
21: (12)                         vars_ = {v: block['vars'][v] for v in value}
22: (12)                         ret.append((key, value, vars_))
23: (4)                 elif hasbody(block):
24: (8)                     for b in block['body']:
25: (12)                         ret = ret + findcommonblocks(b, 0)
26: (4)                 if top:
27: (8)                     tret = []
28: (8)                     names = []
29: (8)                     for t in ret:
30: (12)                         if t[0] not in names:
31: (16)                             names.append(t[0])
32: (16)                             tret.append(t)
33: (8)                     return tret
34: (4)                 return ret
35: (0)             def buildhooks(m):
36: (4)                 ret = {'commonhooks': [], 'initcommonhooks': [],
37: (11)                     'docs': ['"COMMON blocks:\\\\n"']}
38: (4)                 fwrap = []
39: (4)                 def fadd(line, s=fwrap):
40: (8)                     s[0] = '%s\\n      %s' % (s[0], line)
41: (4)                     chooks = []
42: (4)                     def cadd(line, s=chooks):
43: (8)                         s[0] = '%s\\n%s' % (s[0], line)

```

```

44: (4)           ihooks = []
45: (4)           def iadd(line, s=ihooks):
46: (8)             s[0] = '%s\n%s' % (s[0], line)
47: (4)           doc = []
48: (4)           def dadd(line, s=doc):
49: (8)             s[0] = '%s\n%s' % (s[0], line)
50: (4)             for (name, vnames, vars) in findcommonblocks(m):
51: (8)               lower_name = name.lower()
52: (8)               hnames, inames = [], []
53: (8)               for n in vnames:
54: (12)                 if isintent_hide(vars[n]):
55: (16)                   hnames.append(n)
56: (12)                 else:
57: (16)                   inames.append(n)
58: (8)               if hnames:
59: (12)                 outmess('\t\tConstructing COMMON block support for "%s"...\\n\\t\\t
%$\\n\\t\\t Hidden: %s\\n' % (
60: (16)                   name, ', '.join(inames), ', '.join(hnames)))
61: (8)             else:
62: (12)               outmess('\t\tConstructing COMMON block support for "%s"...\\n\\t\\t
%$\\n' % (
63: (16)                   name, ', '.join(inames)))
64: (8)             fadd('subroutine f2pyinit%s(setupfunc)' % name)
65: (8)             for username in getuseblocks(m):
66: (12)               fadd(f'use {username}')
67: (8)               fadd('external setupfunc')
68: (8)               for n in vnames:
69: (12)                 fadd(func2subr.var2fixfortran(vars, n))
70: (8)               if name == '_BLNK_':
71: (12)                 fadd('common %s' % (', '.join(vnames)))
72: (8)               else:
73: (12)                 fadd('common /%s/ %s' % (name, ', '.join(vnames)))
74: (8)               fadd('call setupfunc(%s)' % (', '.join(inames)))
75: (8)               fadd('end\\n')
76: (8)               cadd('static FortranDataDef f2py_%s_def[] = {' % (name))
77: (8)               idims = []
78: (8)               for n in inames:
79: (12)                 ct = capi_maps.getctype(vars[n])
80: (12)                 elsize = capi_maps.get_elsize(vars[n])
81: (12)                 at = capi_maps.c2capi_map[ct]
82: (12)                 dm = capi_maps.getarrdims(n, vars[n])
83: (12)                 if dm['dims']:
84: (16)                   idims.append('(%s)' % (dm['dims']))
85: (12)                 else:
86: (16)                   idims.append('')
87: (12)                 dms = dm['dims'].strip()
88: (12)                 if not dms:
89: (16)                   dms = '-1'
90: (12)                 cadd('\\t{\"%s\",%s,{%s},%s, %s},'
91: (17)                   % (n, dm['rank'], dms, at, elsize))
92: (8)               cadd('\\t{NULL}\\n};')
93: (8)               inames1 = rmbadname(inames)
94: (8)               inames1_tps = ', '.join(['char *' + s for s in inames1])
95: (8)               cadd('static void f2py_setup_%s(%s) {' % (name, inames1_tps))
96: (8)               cadd('\\tint i_f2py=0;')
97: (8)               for n in inames1:
98: (12)                 cadd('\\tf2py_%s_def[i_f2py++].data = %s;' % (name, n))
99: (8)               cadd('}')
100: (8)               if '_' in lower_name:
101: (12)                 F_FUNC = 'F_FUNC_US'
102: (8)               else:
103: (12)                 F_FUNC = 'F_FUNC'
104: (8)               cadd('extern void %s(f2pyinit%,F2PYINIT%) (void(*)(%s));'
105: (13)                 % (F_FUNC, lower_name, name.upper(),
106: (16)                   ', '.join(['char*' * len(inames1))))
107: (8)               cadd('static void f2py_init_%s(void) {' % name)
108: (8)               cadd('\\t%$(f2pyinit%,F2PYINIT%) (f2py_setup_%s);'
109: (13)                 % (F_FUNC, lower_name, name.upper(), name))
110: (8)               cadd('}\\n')
```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

111: (8) iadd('\ttmp = PyFortranObject_New(f2py_%s_def,f2py_init_%s);' % (name,
112: (8) name))
113: (8) iadd('\tif (tmp == NULL) return NULL;')
114: (13) iadd('\tif (F2PyDict_SetItemString(d, \"%s\", tmp) == -1) return
115: (8) % name)
116: (8) iadd('\tPy_DECREF(tmp);')
117: (8) tname = name.replace('_', '\\_')
118: (8) dadd('\\subsection{Common block \\texttt{%s}}\\n' % (tname))
119: (8) dadd('\\begin{description}')
120: (12) for n in inames:
121: (17)     dadd('\\item[]{{{}}\\verb@%s@{}}}' %
122: (12)         (capi_maps.getarrdocsign(n, vars[n])))
123: (16) if hasnote(vars[n]):
124: (16)     note = vars[n]['note']
125: (20)     if isinstance(note, list):
126: (16)         note = '\n'.join(note)
127: (8)     dadd('--- %s' % (note))
128: (8)     dadd('\\end{description}')
129: (12)     ret['docs'].append(
130: (4)         '"\\t/%s/ %s\\n"' % (name, ', '.join(map(lambda v, d: v + d, inames,
131: (4) idims))))
132: (4)     ret['commonhooks'] = chooks
133: (4)     ret['initcommonhooks'] = ihooks
134: (4)     ret['lateXdoc'] = doc[0]
135: (4)     if len(ret['docs']) <= 1:
136: (8)         ret['docs'] = ''
137: (4)     return ret, fwrap[0]
```

---

File 147 - crackfortran.py:

```
1: (0) """
2: (0)     crackfortran --- read fortran (77,90) code and extract declaration
information.
3: (0)     Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
4: (0)     Copyright 2011 -- present NumPy Developers.
5: (0)     Permission to use, modify, and distribute this software is given under the
6: (0)     terms of the NumPy License.
7: (0)     NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
8: (0)     Usage of crackfortran:
9: (0)     =====
10: (0)     Command line keys: -quiet,-verbose,-fix,-f77,-f90,-show,-h <pyffilename>
11: (19)             -m <module name for f77 routines>,--ignore-contains
12: (0)     Functions: crackfortran, crack2fortran
13: (0)     The following Fortran statements/constructions are supported
14: (0)     (or will be if needed):
15: (3)         block data,byte,call,character,common,complex,contains,data,
16: (3)         dimension,double complex,double precision,end,external,function,
17: (3)         implicit,integer,intent,interface,intrinsic,
18: (3)         logical,module,optional,parameter,private,public,
19: (3)         program,real,(sequence?),subroutine,type,use,virtual,
20: (3)         include,pythonmodule
21: (0)     Note: 'virtual' is mapped to 'dimension'.
22: (0)     Note: 'implicit integer (z) static (z)' is 'implicit static (z)' (this is
minor bug).
23: (0)     Note: code after 'contains' will be ignored until its scope ends.
24: (0)     Note: 'common' statement is extended: dimensions are moved to variable
definitions
25: (0)     Note: f2py directive: <commentchar>f2py<line> is read as <line>
26: (0)     Note: pythonmodule is introduced to represent Python module
27: (0)     Usage:
28: (2)         `postlist=crackfortran(files)`
29: (2)         `postlist` contains declaration information read from the list of files
`files`.
30: (2)         `crack2fortran(postlist)` returns a fortran code to be saved to pyf-file
31: (2)         `postlist` has the following structure:
32: (1)         *** it is a list of dictionaries containing `blocks`:
```

```

33: (5)           B =
{'block','body','vars','parent_block',['name','prefix','args','result',
34: (10)          'implicit','externals','interfaced','common','sortvars',
35: (10)          'commonvars','note']}
36: (5)          B['block'] = 'interface' | 'function' | 'subroutine' | 'module' |
37: (18)          'program' | 'block data' | 'type' | 'pythonmodule' |
38: (18)          'abstract interface'
39: (5)          B['body'] --- list containing `subblocks' with the same structure as
`blocks'
40: (5)          B['parent_block'] --- dictionary of a parent block:
41: (29)          C['body'][<index>]['parent_block'] is C
42: (5)          B['vars'] --- dictionary of variable definitions
43: (5)          B['sortvars'] --- dictionary of variable definitions sorted by dependence
(independent first)
44: (5)          B['name'] --- name of the block (not if B['block']=='interface')
45: (5)          B['prefix'] --- prefix string (only if B['block']=='function')
46: (5)          B['args'] --- list of argument names if B['block']=='function' |
47: (5)          B['result'] --- name of the return value (only if B['block']=='function')
48: (5)          B['implicit'] --- dictionary {'a':<variable definition>,'b':...} | None
49: (5)          B['externals'] --- list of variables being external
50: (5)          B['interfaced'] --- list of variables being external and defined
51: (5)          B['common'] --- dictionary of common blocks (list of objects)
52: (5)          B['commonvars'] --- list of variables used in common blocks (dimensions
are moved to variable definitions)
53: (5)          B['from'] --- string showing the 'parents' of the current block
54: (5)          B['use'] --- dictionary of modules used in current block:
55: (9)          {<modulename>:{['only':<0|1>],['map':{<local_name1>:
<use_name1>,...}]}}
56: (5)          B['note'] --- list of LaTeX comments on the block
57: (5)          B['f2pyenhancements'] --- optional dictionary
58: (10)         {'threadsafe':'','fortranname':<name>,
59: (11)         'callstatement':<C-expr>|<multi-line block>,
60: (11)         'callprotoargument':<C-expr-list>,
61: (11)         'usercode':<multi-line block>|<list of multi-line blocks>,
62: (11)         'pymethoddef':<multi-line block>
63: (11)         }
64: (5)          B['entry'] --- dictionary {entryname:argslist,...}
65: (5)          B['varnames'] --- list of variable names given in the order of reading
the
66: (23)         Fortran code, useful for derived types.
67: (5)          B['saved_interface'] --- a string of scanned routine signature, defines
explicit interface
68: (1)          *** Variable definition is a dictionary
69: (5)          D = B['vars'][<variable name>] =
70: (5)          {'typespec',[,'attrspec','kindselector','chaselector','=','typename']}
71: (5)          D['typespec'] = 'byte' | 'character' | 'complex' | 'double complex' |
72: (21)          'double precision' | 'integer' | 'logical' | 'real' |
73: (5)          'type'
73: (5)          D['attrspec'] --- list of attributes (e.g. 'dimension(<arrayspec>)',
74: (23)
'external','intent(in|out|inout|hide|c|callback|cache|aligned4|aligned8|aligned16)',
75: (23)          'optional','required', etc)
76: (5)          K = D['kindselector'] = {[*, 'kind']} (only if D['typespec'] =
77: (25)          'complex' | 'integer' | 'logical' | 'real' )
78: (5)          C = D['chaselector'] = {[*, 'len', 'kind', 'f2py_len']}
79: (29)          (only if D['typespec']=='character')
80: (5)          D['='] --- initialization expression string
81: (5)          D['typename'] --- name of the type if D['typespec']=='type'
82: (5)          D['dimension'] --- list of dimension bounds
83: (5)          D['intent'] --- list of intent specifications
84: (5)          D['depend'] --- list of variable names on which current variable depends
on
85: (5)          D['check'] --- list of C-expressions; if C-expr returns zero, exception
is raised
86: (5)          D['note'] --- list of LaTeX comments on the variable
87: (1)          *** Meaning of kind/char selectors (few examples):
88: (5)          D['typespec']*[K[*']]
89: (5)          D['typespec'](kind=K['kind'])

```

```

90: (5)          character*C['*']
91: (5)          character(len=C['len'],kind=C['kind'], f2py_len=C['f2py_len'])
92: (5)          (see also fortran type declaration statement formats below)
93: (0)          Fortran 90 type declaration statement format (F77 is subset of F90)
94: (0)          =====
95: (0)          (Main source: IBM XL Fortran 5.1 Language Reference Manual)
96: (0)          type declaration = <typespec> [[<attrspec>:::] <entitydecl>
97: (0)          <typespec> = byte
98: (13)          character[<charselctor>]
99: (13)          complex[<kindselctor>]
100: (13)          double complex
101: (13)          double precision
102: (13)          integer[<kindselctor>]
103: (13)          logical[<kindselctor>]
104: (13)          real[<kindselctor>]
105: (13)          type(<typename>)
106: (0)          <charselctor> = * <charlen>
107: (13)          ([len=]<len>[,,[kind=]<kind>])
108: (13)          (kind=<kind>[,,<len>])
109: (0)          <kindselctor> = * <intlen>
110: (13)          ([kind=]<kind>)
111: (0)          <attrspec> = comma separated list of attributes.
112: (13)          Only the following attributes are used in
113: (13)          building up the interface:
114: (16)          external
115: (16)          (parameter --- affects '=' key)
116: (16)          optional
117: (16)          intent
118: (13)          Other attributes are ignored.
119: (0)          <intentspec> = in | out | inout
120: (0)          <arrayspec> = comma separated list of dimension bounds.
121: (0)          <entitydecl> = <name> [[*<charlen>][(<arrayspec>)] | [(<arrayspec>)]*
<charlen>]
122: (22)          [/<init_expr>/ | =<init_expr>] [,<entitydecl>]
123: (0)          In addition, the following attributes are used: check,depend,note
124: (0)          TODO:
125: (4)          * Apply 'parameter' attribute (e.g. 'integer parameter :: i=2' 'real x(i)'
126: (35)          -> 'real x(2)')
127: (4)          The above may be solved by creating appropriate preprocessor program, for
example.
128: (0)          """
129: (0)          import sys
130: (0)          import string
131: (0)          import fileinput
132: (0)          import re
133: (0)          import os
134: (0)          import copy
135: (0)          import platform
136: (0)          import codecs
137: (0)          from pathlib import Path
138: (0)          try:
139: (4)          import charset_normalizer
140: (0)          except ImportError:
141: (4)          charset_normalizer = None
142: (0)          from . import __version__
143: (0)          from .auxfuncs import *
144: (0)          from . import symbolic
145: (0)          f2py_version = __version__.version
146: (0)          strictf77 = 1          # Ignore `!' comments unless line[0]=='!'
147: (0)          sourcecodeform = 'fix'  # 'fix','free'
148: (0)          quiet = 0           # Be verbose if 0 (Obsolete: not used any more)
149: (0)          verbose = 1          # Be quiet if 0, extra verbose if > 1.
150: (0)          tabchar = 4 * ' '
151: (0)          pyffilename = ''
152: (0)          f77modulename = ''
153: (0)          skipemptyends = 0    # for old F77 programs without 'program' statement
154: (0)          ignorecontains = 1
155: (0)          dolowercase = 1
156: (0)          debug = []

```

```

157: (0) beginpattern = ''
158: (0) currentfilename = ''
159: (0) expectbegin = 1
160: (0) f90modulevars = {}
161: (0) filepositiontext = ''
162: (0) gotnextfile = 1
163: (0) groupcache = None
164: (0) groupcounter = 0
165: (0) grouplist = {groupcounter: []}
166: (0) groupname = ''
167: (0) include_paths = []
168: (0) neededmodule = -1
169: (0) onlyfuncs = []
170: (0) previous_context = None
171: (0) skipblocksuntil = -1
172: (0) skipfuncs = []
173: (0) skipfunctions = []
174: (0) usermodules = []
175: (0) def reset_global_f2py_vars():
176: (4)     global groupcounter, grouplist, neededmodule, expectbegin
177: (4)     global skipblocksuntil, usermodules, f90modulevars, gotnextfile
178: (4)     global filepositiontext, currentfilename, skipfunctions, skipfuncs
179: (4)     global onlyfuncs, include_paths, previous_context
180: (4)     global strictf77, sourcecodeform, quiet, verbose, tabchar, pyffilename
181: (4)     global f77modulename, skipemptyends, ignorecontains, dolowercase, debug
182: (4)     strictf77 = 1
183: (4)     sourcecodeform = 'fix'
184: (4)     quiet = 0
185: (4)     verbose = 1
186: (4)     tabchar = 4 * ' '
187: (4)     pyffilename = ''
188: (4)     f77modulename = ''
189: (4)     skipemptyends = 0
190: (4)     ignorecontains = 1
191: (4)     dolowercase = 1
192: (4)     debug = []
193: (4)     groupcounter = 0
194: (4)     grouplist = {groupcounter: []}
195: (4)     neededmodule = -1
196: (4)     expectbegin = 1
197: (4)     skipblocksuntil = -1
198: (4)     usermodules = []
199: (4)     f90modulevars = {}
200: (4)     gotnextfile = 1
201: (4)     filepositiontext = ''
202: (4)     currentfilename = ''
203: (4)     skipfunctions = []
204: (4)     skipfuncs = []
205: (4)     onlyfuncs = []
206: (4)     include_paths = []
207: (4)     previous_context = None
208: (0)     def outmess(line, flag=1):
209: (4)         global filepositiontext
210: (4)         if not verbose:
211: (8)             return
212: (4)         if not quiet:
213: (8)             if flag:
214: (12)                 sys.stdout.write(filepositiontext)
215: (8)                 sys.stdout.write(line)
216: (0)             re._MAXCACHE = 50
217: (0)             defaultimplicitrules = {}
218: (0)             for c in "abcdefghijklmnopqrstuvwxyz$":
219: (4)                 defaultimplicitrules[c] = {'typespec': 'real'}
220: (0)             for c in "ijklmn":
221: (4)                 defaultimplicitrules[c] = {'typespec': 'integer'}
222: (0)             badnames = {}
223: (0)             invbadnames = {}
224: (0)             for n in ['int', 'double', 'float', 'char', 'short', 'long', 'void', 'case',
'while',

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

225: (10)                                'return', 'signed', 'unsigned', 'if', 'for', 'typedef', 'sizeof',
'union',
226: (10)                                'struct', 'static', 'register', 'new', 'break', 'do', 'goto',
'switch',
227: (10)                                'continue', 'else', 'inline', 'extern', 'delete', 'const', 'auto',
228: (10)                                'len', 'rank', 'shape', 'index', 'slen', 'size', '_i',
229: (10)                                'max', 'min',
230: (10)                                'flen', 'fshape',
231: (10)                                'string', 'complex_double', 'float_double', 'stdin', 'stderr',
'stdout',
232: (10)                                'type', 'default']:
233: (4)                                 badnames[n] = n + '_bn'
234: (4)                                 invbadnames[n + '_bn'] = n
235: (0)                                 def rmbadname1(name):
236: (4)                                   if name in badnames:
237: (8)                                     errmess('rmbadname1: Replacing "%s" with "%s".\n' %
238: (16)                                       (name, badnames[name]))
239: (8)                                     return badnames[name]
240: (4)                                   return name
241: (0)                                 def rmbadname(names):
242: (4)                                   return [rmbadname1(_m) for _m in names]
243: (0)                                 def undo_rmbadname1(name):
244: (4)                                   if name in invbadnames:
245: (8)                                     errmess('undo_rmbadname1: Replacing "%s" with "%s".\n' %
246: (16)                                       (name, invbadnames[name]))
247: (8)                                     return invbadnames[name]
248: (4)                                   return name
249: (0)                                 def undo_rmbadname(names):
250: (4)                                   return [undo_rmbadname1(_m) for _m in names]
251: (0)                                 _has_f_header = re.compile(r'-\*-s*fortran\s*-`-\*', re.I).search
252: (0)                                 _has_f90_header = re.compile(r'-\*-s*f90\s*-`-\*', re.I).search
253: (0)                                 _has_fix_header = re.compile(r'-\*-s*fix\s*-`-\*', re.I).search
254: (0)                                 _free_f90_start = re.compile(r'^[^\s]*[\^s]\d\t]', re.I).match
255: (0)                                 COMMON_FREE_EXTENSIONS = ['.f90', '.f95', '.f03', '.f08']
256: (0)                                 COMMON_FIXED_EXTENSIONS = ['.for', '.ftn', '.f77', '.f']
257: (0)                                 def openhook(filename, mode):
258: (4)                                   """Ensures that filename is opened with correct encoding parameter.
259: (4)                                   This function uses charset_normalizer package, when available, for
260: (4)                                   determining the encoding of the file to be opened. When charset_normalizer
261: (4)                                   is not available, the function detects only UTF encodings, otherwise,
ASCII
262: (4)                                   encoding is used as fallback.
263: (4)                                   """
264: (4)                                   if charset_normalizer is not None:
265: (8)                                     encoding = charset_normalizer.from_path(filename).best().encoding
266: (4)                                   else:
267: (8)                                     nbytes = min(32, os.path.getsize(filename))
268: (8)                                     with open(filename, 'rb') as fhandle:
269: (12)                                       raw = fhandle.read(nbytes)
270: (12)                                       if raw.startswith(codecs.BOM_UTF8):
271: (16)   encoding = 'UTF-8-SIG'
272: (12)   elif raw.startswith((codecs.BOM_UTF32_LE, codecs.BOM_UTF32_BE)):
273: (16)   encoding = 'UTF-32'
274: (12)   elif raw.startswith((codecs.BOM_LE, codecs.BOM_BE)):
275: (16)   encoding = 'UTF-16'
276: (12)   else:
277: (16)   encoding = 'ascii'
278: (4)                                     return open(filename, mode, encoding=encoding)
279: (0)                                 def is_free_format(fname):
280: (4)                                   """Check if file is in free format Fortran."""
281: (4)                                   result = False
282: (4)                                   if Path(fname).suffix.lower() in COMMON_FREE_EXTENSIONS:
283: (8)                                     result = True
284: (4)                                   with openhook(fname, 'r') as fhandle:
285: (8)                                     line = fhandle.readline()
286: (8)                                     n = 15 # the number of non-comment lines to scan for hints
287: (8)                                     if _has_f_header(line):
288: (12)                                       n = 0
289: (8)                                     elif _has_f90_header(line):

```

```

290: (12)          n = 0
291: (12)          result = True
292: (8)           while n > 0 and line:
293: (12)             if line[0] != '!' and line.strip():
294: (16)               n -= 1
295: (16)               if (line[0] != '\t' and _free_f90_start(line[:5])) or
line[-2:-1] == '&':
296: (20)                 result = True
297: (20)                 break
298: (12)                 line = fhandle.readline()
299: (4)                  return result
300: (0)  def readfortrancode(ffile, dowithline=show, istop=1):
301: (4)    """
302: (4)      Read fortran codes from files and
303: (5)        1) Get rid of comments, line continuations, and empty lines; lower cases.
304: (5)        2) Call dowithline(line) on every line.
305: (5)        3) Recursively call itself when statement \"include '<filename>'\" is
met.
306: (4)
307: (4)  global gotnextfile, filepositiontext, currentfilename, sourcecodeform,
strictf77
308: (4)  global beginpattern, quiet, verbose, dolowercase, include_paths
309: (4)  if not istop:
310: (8)    saveglobals = gotnextfile, filepositiontext, currentfilename,
sourcecodeform, strictf77,\ \
311: (12)          beginpattern, quiet, verbose, dolowercase
312: (4)  if ffile == []:
313: (8)    return
314: (4)  localdolowercase = dolowercase
315: (4)  cont = False
316: (4)  finalline = ''
317: (4)  ll = ''
318: (4)  includeline = re.compile(
319: (8)    r'\s*include\s*(\\"|\")(?P<name>[^\\"]*)(\\\"|")', re.I)
320: (4)  cont1 = re.compile(r'(?P<line>.*)&\s*\Z')
321: (4)  cont2 = re.compile(r'(\s*&|)(?P<line>.*)')
322: (4)  mline_mark = re.compile(r".*?''''")
323: (4)  if istop:
324: (8)    dowithline('', -1)
325: (4)  ll, ll = '', ''
326: (4)  spacedigits = [' '] + [str(_m) for _m in range(10)]
327: (4)  filepositiontext = ''
328: (4)  fin = fileinput.FileInput(ffile, openhook=openhook)
329: (4)  while True:
330: (8)
331: (12)    try:
332: (8)      l = fin.readline()
333: (12)    except UnicodeDecodeError as msg:
334: (16)      raise Exception(
335: (16)        f'readfortrancode: reading {fin.filename()}#{fin.lineno()}'
336: (16)        f' failed with\n{msg}.\nIt is likely that installing
337: (16)          package will help f2py determine the input file encoding'
338: (8)          ' correctly.')
339: (12)    if not l:
340: (8)      break
341: (12)    if fin.isfirstline():
342: (12)      filepositiontext = ''
343: (12)      currentfilename = fin.filename()
344: (12)      gotnextfile = 1
345: (12)      ll = 1
346: (12)      strictf77 = 0
347: (12)      sourcecodeform = 'fix'
348: (12)      ext = os.path.splitext(currentfilename)[1]
349: (20)      if Path(currentfilename).suffix.lower() in COMMON_FIXED_EXTENSIONS
and \
350: (16)          not (_has_f90_header(l) or _has_fix_header(l)):
351: (12)          strictf77 = 1
352: (16)          elif is_free_format(currentfilename) and not _has_fix_header(l):
353: (16)              sourcecodeform = 'free'
```

```

353: (12)
354: (16)
355: (12)
356: (16)
357: (12)
358: (20)
359: (23)
360: (8)
361: (8)
362: (12)
363: (16)
364: (12)
365: (8)
366: (12)
367: (12)
368: (12)
369: (16)
370: (8)
371: (12)
372: (16)
373: (12)
374: (16)
375: (12)
376: (8)
377: (12)
378: (16)
379: (20)
380: (16)
381: (20)
382: (20)
383: (12)
384: (16)
385: (20)
386: (12)
387: (16)
'
388: (32)
389: (32)
repr(1))
390: (12)
391: (16)
392: (16)
393: (16)
394: (12)
395: (16)
396: (20)
397: (20)
398: (24)
399: (20)
400: (24)
401: (24)
402: (24)
403: (20)
404: (24)
405: (24)
406: (28)
407: (24)
408: (28)
409: (24)
410: (24)
411: (20)
412: (16)
413: (20)
414: (20)
415: (24)
416: (20)
417: (24)
418: (20)
419: (20)

            if strictf77:
                beginpattern = beginpattern77
            else:
                beginpattern = beginpattern90
            outmess('\tReading file %s (format:%s%s)\n'
                    % (repr(currentfilename), sourcecodeform,
                       strictf77 and ',strict' or ''))
            l = l.expandtabs().replace('\xa0', ' ')
        while not l == '':
            if l[-1] not in "\n\r\f":
                break
            l = l[:-1]
        if not strictf77:
            (l, rl) = split_by_unquoted(l, '!')
            l += ' '
            if rl[:5].lower() == '!f2py': # f2py directive
                l, _ = split_by_unquoted(l + 4 * ' ' + rl[5:], '!')
        if l.strip() == '': # Skip empty line
            if sourcecodeform == 'free':
                pass
            else:
                cont = False
            continue
        if sourcecodeform == 'fix':
            if l[0] in ['*', 'c', '!', 'C', '#']:
                if l[1:5].lower() == 'f2py': # f2py directive
                    l = ' ' + l[5:]
                else: # Skip comment line
                    cont = False
                continue
            elif strictf77:
                if len(l) > 72:
                    l = l[:72]
            if not (l[0] in spacedigits):
                raise Exception('readfortrancode: Found non-(space,digit) char '
                                'in the first column.\n\tAre you sure that '
                                'this code is in fix form?\n\tline=%s' %
                                l)
            if (not cont or strictf77) and (len(l) > 5 and not l[5] == ' '):
                ll = ll + l[6:]
                finalline = ''
                origfinalline = ''
            else:
                if not strictf77:
                    r = cont1.match(l)
                    if r:
                        l = r.group('line') # Continuation follows ..
                    if cont:
                        ll = ll + cont2.match(l).group('line')
                        finalline = ''
                        origfinalline = ''
                    else:
                        l = ' ' + l[5:]
                        if localdolowercase:
                            finalline = ll.lower()
                        else:
                            finalline = ll
                        origfinalline = ll
                        ll = l
                    cont = (r is not None)
                else:
                    l = ' ' + l[5:]
                    if localdolowercase:
                        finalline = ll.lower()
                    else:
                        finalline = ll
                    origfinalline = ll
                    ll = l

```

```

420: (8)           elif sourcecodeform == 'free':
421: (12)             if not cont and ext == '.pyf' and mline_mark.match(l):
422: (16)               l = l + '\n'
423: (16)               while True:
424: (20)                 lc = fin.readline()
425: (20)                 if not lc:
426: (24)                   errmess(
427: (28)                     'Unexpected end of file when reading multiline\n')
428: (24)
429: (20)                     break
430: (20)                     l = l + lc
431: (24)                     if mline_mark.match(lc):
432: (16)                       break
433: (12)                     l = l.rstrip()
434: (12)                     r = cont1.match(l)
435: (16)                     if r:
436: (12)                       l = r.group('line') # Continuation follows ..
437: (16)                     if cont:
438: (16)                         ll = ll + cont2.match(l).group('line')
439: (16)                         finalline = ''
440: (12)                         origfinalline = ''
441: (16)                     else:
442: (20)                         if localdolowercase:
443: (16)                           finalline = ll.lower()
444: (20)                         else:
445: (16)                           finalline = ll
446: (16)                           origfinalline = ll
447: (12)                           ll = l
448: (8)                           cont = (r is not None)
449: (12)                     else:
450: (16)                         raise ValueError(
451: (8)                           "Flag sourcecodeform must be either 'fix' or 'free': %s" %
repr(sourcecodeform))
452: (12)
453: (8)
454: (8)
455: (12)
456: (12)
457: (16)
458: (12)
459: (16)
460: (20)
461: (16)
462: (16)
463: (20)
464: (20)
465: (24)
466: (24)
467: (24)
468: (16)
469: (20)
in %s. Ignoring.\n' %
470: (24)           filepositiontext = 'Line #%d in %s:\n\t' % (
471: (8)             fin.filelineno() - 1, currentfilename, ll)
472: (12)             m = includeline.match(origfinalline)
473: (8)             if m:
474: (4)               fn = m.group('name')
475: (8)               if os.path.isfile(fn):
476: (4)                 readfortrancode(fn, dowithline=dowithline, istop=0)
477: (8)               else:
478: (4)                 include_dirs = [
479: (8)                   os.path.dirname(currentfilename)] + include_paths
480: (8)                 foundfile = 0
481: (4)                 for inc_dir in include_dirs:
482: (8)                   fn1 = os.path.join(inc_dir, fn)
483: (8)                   if os.path.isfile(fn1):
484: (8)                     foundfile = 1
485: (12)                     readfortrancode(fn1, dowithline=dowithline, istop=0)
486: (8)                     break
487: (8)                     if not foundfile:
488: (8)                       outmess('readfortrancode: could not find include file %s
489: (8)                         repr(fn), os.pathsep.join(include_dirs)))
490: (8)                     else:
491: (12)                       dowithline(finalline)
492: (8)                     ll = ll
493: (4)                     if localdolowercase:
494: (8)                       finalline = ll.lower()
495: (4)                     else:
496: (8)                       finalline = ll
497: (4)                     origfinalline = ll
498: (4)                     filepositiontext = 'Line #%d in %s:\n\t' % (
499: (8)                       fin.filelineno() - 1, currentfilename, ll)
500: (4)                     m = includeline.match(origfinalline)
501: (4)                     if m:
502: (8)                       fn = m.group('name')
503: (8)                       if os.path.isfile(fn):
504: (8)                         readfortrancode(fn, dowithline=dowithline, istop=0)
505: (8)                         else:

```

```

487: (12)           include_dirs = [os.path.dirname(currentfilename)] + include_paths
488: (12)           foundfile = 0
489: (12)           for inc_dir in include_dirs:
490: (16)             fn1 = os.path.join(inc_dir, fn)
491: (16)             if os.path.isfile(fn1):
492: (20)               foundfile = 1
493: (20)               readfortrancode(fn1, dowithline=dowithline, istop=0)
494: (20)               break
495: (12)           if not foundfile:
496: (16)             outmess('readfortrancode: could not find include file %s in
%$. Ignoring.\n' % (
497: (20)               repr(fn), os.pathsep.join(include_dirs)))
498: (4)           else:
499: (8)             dowithline(finalline)
500: (4)             filepositiontext = ''
501: (4)             fin.close()
502: (4)             if istop:
503: (8)               dowithline('', 1)
504: (4)             else:
505: (8)               gotnextfile, filepositiontext, currentfilename, sourcecodeform,
strictf77,\           beginpattern, quiet, verbose, dolowercase = saveglobals
506: (12)           beforethisafter = r'\s*(?P<before>%s(=\s*(\b(%s)\b)))' + \
507: (0)             r'\s*(?P<this>(\b(%s)\b))' + \
508: (4)             r'\s*(?P<after>%s)\s*\Z'
509: (4)           fortrantypes = r'character|logical|integer|real|complex|double\s*(precision\s*\
510: (0) (complex)|complex)|type(?\s*([_\w\s,=(*])*\b))|byte'
511: (0)           typespattern = re.compile(
512: (4)             beforethisafter % ('', fortrantypes, fortrantypes, '.'), re.I), 'type'
513: (0)           typespattern4implicit = re.compile(beforethisafter % (
514: (4)             '', fortrantypes + '|static|automatic|undefined', fortrantypes + \
'|static|automatic|undefined', '.'), re.I)
515: (0)           functionpattern = re.compile(beforethisafter % (
516: (4)             r'([a-z]+[\w\s(=+-/)]*?)', 'function', 'function', '.'), re.I), 'begin'
517: (0)           subroutinepattern = re.compile(beforethisafter % (
518: (4)             r'[a-z\s]*?', 'subroutine', 'subroutine', '.'), re.I), 'begin'
519: (0)           groupbegins77 = r'program|block\s*data'
520: (0)           beginpattern77 = re.compile(
521: (4)             beforethisafter % ('', groupbegins77, groupbegins77, '.'), re.I), 'begin'
522: (0)           groupbegins90 = groupbegins77 + \
523: (4)             r'|module(?!s*procedure)|python\s*module|(abstract|)\s*interface|' + \
524: (4)             r'type(?!s*\b)'
525: (0)           beginpattern90 = re.compile(
526: (4)             beforethisafter % ('', groupbegins90, groupbegins90, '.'), re.I), 'begin'
527: (0)           groupends = (r'end|endprogram|endblockdata|endmodule|endpythonmodule|' \
528: (13)             r'endinterface|endsubroutine|endfunction')
529: (0)           endpattern = re.compile(
530: (4)             beforethisafter % ('', groupends, groupends, '.'), re.I), 'end'
531: (0)           endifs = r'end\s*(if|do|where|select|while|forall|associate|' + \
532: (9)             r'critical|enum|team)'
533: (0)           endifpattern = re.compile(
534: (4)             beforethisafter % (r'[\w]*?', endifs, endifs, '.'), re.I), 'endif'
535: (0)           moduleprocedures = r'module\s*procedure'
536: (0)           moduleprocedurepattern = re.compile(
537: (4)             beforethisafter % ('', moduleprocedures, moduleprocedures, '.'), re.I), \
538: (4)             'moduleprocedure'
539: (0)           implicitpattern = re.compile(
540: (4)             beforethisafter % ('', 'implicit', 'implicit', '.'), re.I), 'implicit'
541: (0)           dimensionpattern = re.compile(beforethisafter % (
542: (4)             '', 'dimension|virtual', 'dimension|virtual', '.'), re.I), 'dimension'
543: (0)           externalpattern = re.compile(
544: (4)             beforethisafter % ('', 'external', 'external', '.'), re.I), 'external'
545: (0)           optionalpattern = re.compile(
546: (4)             beforethisafter % ('', 'optional', 'optional', '.'), re.I), 'optional'
547: (0)           requiredpattern = re.compile(
548: (4)             beforethisafter % ('', 'required', 'required', '.'), re.I), 'required'
549: (0)           publicpattern = re.compile(
550: (4)             beforethisafter % ('', 'public', 'public', '.'), re.I), 'public'
551: (0)           privatepattern = re.compile(

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

552: (4)           beforethisafter % ('', 'private', 'private', '.*'), re.I), 'private'
553: (0)           intrinsicpattern = re.compile(
554: (4)             beforethisafter % ('', 'intrinsic', 'intrinsic', '.*'), re.I), 'intrinsic'
555: (0)           intentpattern = re.compile(beforethisafter % (
556: (4)               '', 'intent|depend|note|check', 'intent|depend|note|check', r'\s*(.*?
557: (0)           ), re.I), 'intent'
558: (4)           parameterpattern = re.compile(
559: (0)             beforethisafter % ('', 'parameter', 'parameter', r'\s*(.*'), re.I),
560: (4)           'parameter'
561: (0)           datapattern = re.compile(
562: (4)             beforethisafter % ('', 'data', 'data', '.*'), re.I), 'data'
563: (0)           callpattern = re.compile(
564: (4)             beforethisafter % ('', 'call', 'call', '.*'), re.I), 'call'
565: (0)           entrypattern = re.compile(
566: (4)             beforethisafter % ('', 'entry', 'entry', '.*'), re.I), 'entry'
567: (0)           callfunpattern = re.compile(
568: (4)             beforethisafter % ('', 'callfun', 'callfun', '.*'), re.I), 'callfun'
569: (0)           commonpattern = re.compile(
570: (4)             beforethisafter % ('', 'common', 'common', '.*'), re.I), 'common'
571: (0)           usepattern = re.compile(
572: (4)             beforethisafter % ('', 'use', 'use', '.*'), re.I), 'use'
573: (0)           containspattern = re.compile(
574: (4)             beforethisafter % ('', 'contains', 'contains', ''), re.I), 'contains'
575: (0)           formatpattern = re.compile(
576: (56)           beforethisafter % ('', 'format', 'format', '.*'), re.I), 'format'
577: (0)           f2pyenhancementspattern = re.compile(beforethisafter % '',
578: (4)           'threadsafe|fortranname|callstatement|callprotoargument|usercode|pymethoddef',
579: (0)           're.S', 'f2pyenhancements')
580: (4)           multilinepattern = re.compile(
581: (4)             r"\s*(?P<before>'')(?P<this>.*)?(?P<after>'')\s*\Z", re.S), 'multiline'
582: (0)           def split_by_unquoted(line, characters):
583: (4)             """
584: (4)               Splits the line into (line[:i], line[i:]),
585: (4)               where i is the index of first occurrence of one of the characters
586: (4)               not within quotes, or len(line) if no such index exists
587: (8)             """
588: (8)             assert not (set("'\\"") & set(characters)), "cannot split by unquoted
589: (12)             quotes"
590: (12)             r = re.compile(
591: (12)               r"\A(?P<before>({single_quoted}|{double_quoted}|{not_quoted}))"
592: (12)               r"(?P<after>{char}.*)\Z".format(
593: (4)                 not_quoted="[^'{}]\.".format(re.escape(characters)),
594: (4)                 char="{}\.".format(re.escape(characters)),
595: (8)                 single_quoted=r"('([^\\\]|(\\\.)*)'",
596: (8)                 double_quoted=r'"([^\\\]|(\\\.)*)"')
597: (4)             m = r.match(line)
598: (0)             if m:
599: (4)               d = m.groupdict()
600: (4)               return (d["before"], d["after"])
601: (8)             return (line, "")
602: (12)             def _simplifyargs(argsline):
603: (8)               a = []
604: (4)               for n in markoutercomma(argsline).split('@,@'):
605: (0)                 for r in '(),':
606: (0)                   n = n.replace(r, '_')
607: (0)                   a.append(n)
608: (0)               return ','.join(a)
609: (4)             crackline_re_1 = re.compile(r'\s*(?P<result>\b[a-z]+\w*\b)\s*.*', re.I)
610: (4)             crackline_bind_1 = re.compile(r'\s*(?P<bind>\b[a-z]+\w*\b)\s*.*', re.I)
611: (0)             crackline_bindlang = re.compile(r'\s*bind\(\s*(?P<lang>
612: (4)               [^,]+)\s*,\s*name\s*=\s*"(?P<lang_name>[^"]+)\s*\)', re.I)
613: (4)             def crackline(line, reset=0):
614: (0)               """
615: (4)                 reset=-1 --- initialize
616: (4)                 reset=0 --- crack the line
617: (4)                 reset=1 --- final check if mismatch of blocks occurred
618: (4)               Cracked data is saved in grouplist[0].

```

```

614: (4)
615: (4)
616: (4)
617: (4)
618: (4)
619: (4)
620: (31)
621: (8)
622: (8)
623: (8)
624: (12)
625: (12)
626: (8)
627: (8)
628: (4)
629: (8)
630: (8)
631: (8)
632: (8)
633: (8)
634: (8)
635: (8)
636: (8)
637: (8)
638: (8)
639: (8)
640: (4)
641: (8)
642: (8)
643: (12)
644: (8)
645: (12)
646: (20)
647: (12)
648: (16)
by assuming "end" statement.\n')
649: (12)
650: (12)
651: (12)
652: (12)
653: (8)
654: (12)
655: (12)
656: (12)
657: (12)
658: (12)
659: (12)
660: (12)
661: (12)
662: (12)
663: (8)
664: (4)
665: (8)
666: (4)
667: (4)
optionalpattern,
668: (16)
669: (16)
670: (16)
671: (16)
672: (16)
673: (16)
674: (16)
675: (16)
676: (16)
677: (16)
678: (16)
679: (16)
680: (16)

        """
        global beginpattern, groupcounter, groupname, groupcache, grouplist
        global filepositiontext, currentfilename, neededmodule, expectbegin
        global skipblocksuntil, skipemptyends, previous_context, gotnextfile
        _, has_semicolon = split_by_unquoted(line, ";")
        if has_semicolon and not (f2pyenhancementspattern[0].match(line) or
                                   multilinepattern[0].match(line)):
            assert reset == 0, repr(reset)
            line, semicolon_line = split_by_unquoted(line, ";")
            while semicolon_line:
                crackline(line, reset)
                line, semicolon_line = split_by_unquoted(semicolon_line[1:], ";")
            crackline(line, reset)
            return
        if reset < 0:
            groupcounter = 0
            groupname = {groupcounter: ''}
            groupcache = {groupcounter: {}}
            grouplist = {groupcounter: []}
            groupcache[groupcounter]['body'] = []
            groupcache[groupcounter]['vars'] = {}
            groupcache[groupcounter]['block'] = ''
            groupcache[groupcounter]['name'] = ''
            neededmodule = -1
            skipblocksuntil = -1
            return
        if reset > 0:
            f1 = 0
            if f77modulename and neededmodule == groupcounter:
                f1 = 2
            while groupcounter > f1:
                outmess('crackline: groupcounter=%s groupname=%s\n' %
                        (repr(groupcounter), repr(groupname)))
                outmess(
                    'crackline: Mismatch of blocks encountered. Trying to fix it
by assuming "end" statement.\n')
                grouplist[groupcounter - 1].append(groupcache[groupcounter])
                grouplist[groupcounter - 1][-1]['body'] = grouplist[groupcounter]
                del grouplist[groupcounter]
                groupcounter = groupcounter - 1
            if f77modulename and neededmodule == groupcounter:
                grouplist[groupcounter - 1].append(groupcache[groupcounter])
                grouplist[groupcounter - 1][-1]['body'] = grouplist[groupcounter]
                del grouplist[groupcounter]
                groupcounter = groupcounter - 1 # end interface
                grouplist[groupcounter - 1].append(groupcache[groupcounter])
                grouplist[groupcounter - 1][-1]['body'] = grouplist[groupcounter]
                del grouplist[groupcounter]
                groupcounter = groupcounter - 1 # end module
                neededmodule = -1
            return
        if line == '':
            return
        flag = 0
        for pat in [dimensionpattern, externalpattern, intentpattern,
                    optionalpattern,
                    requiredpattern,
                    parameterpattern, datapattern, publicpattern, privatepattern,
                    intrinsicpattern,
                    endifpattern, endpattern,
                    formatpattern,
                    beginpattern, functionpattern, subroutinepattern,
                    implicitpattern, typespattern, commonpattern,
                    callpattern, usepattern, containspattern,
                    entrypattern,
                    f2pyenhancementspattern,
                    multilinepattern,
                    moduleprocedurepattern
                    ]:

```

```

681: (8)             m = pat[0].match(line)
682: (8)             if m:
683: (12)                 break
684: (8)             flag = flag + 1
685: (4)             if not m:
686: (8)                 re_1 = crackline_re_1
687: (8)                 if 0 <= skipblocksuntil <= groupcounter:
688: (12)                     return
689: (8)                 if 'externals' in groupcache[groupcounter]:
690: (12)                     for name in groupcache[groupcounter]['externals']:
691: (16)                         if name in invbadnames:
692: (20)                             name = invbadnames[name]
693: (16)                         if 'interfaced' in groupcache[groupcounter] and name in
groupcache[groupcounter]['interfaced']:
694: (20)                             continue
695: (16)                         m1 = re.match(
696: (20)                             r'(?P<before>[^"]*)\b%s\b\s*@\(@(?P<args>[@]*)@)\@.*\Z' %
name, markouterparen(line), re.I)
697: (16)                         if m1:
698: (20)                             m2 = re_1.match(m1.group('before'))
699: (20)                             a = _simplifyargs(m1.group('args'))
700: (20)                             if m2:
701: (24)                                 line = 'callfun %s(%s) result (%s)' % (
702: (28)                                     name, a, m2.group('result'))
703: (20)                             else:
704: (24)                                 line = 'callfun %s(%s)' % (name, a)
705: (20)                             m = callfunpattern[0].match(line)
706: (20)                             if not m:
707: (24)                                 outmess(
708: (28)                                     'crackline: could not resolve function call for
line=%s.\n' % repr(line))
709: (24)                                 return
710: (20)                             analyzeline(m, 'callfun', line)
711: (20)                             return
712: (8)             if verbose > 1 or (verbose == 1 and
currentfilename.lower().endswith('.pyf')):
713: (12)                 previous_context = None
714: (12)                 outmess('crackline:%d: No pattern for line\n' % (groupcounter))
715: (8)                 return
716: (4)             elif pat[1] == 'end':
717: (8)                 if 0 <= skipblocksuntil < groupcounter:
718: (12)                     groupcounter = groupcounter - 1
719: (12)                     if skipblocksuntil <= groupcounter:
720: (16)                         return
721: (8)                     if groupcounter <= 0:
722: (12)                         raise Exception('crackline: groupcounter(=%s) is nonpositive. '
723: (28)                             'Check the blocks.' %
(groupcounter))
724: (28)                     m1 = beginpattern[0].match((line))
725: (8)                     if (m1) and (not m1.group('this') == groupname[groupcounter]):
726: (8)                         raise Exception('crackline: End group %s does not match with '
727: (12)                             'previous Begin group %s\n\t%s' %
(repr(m1.group('this'))),
728: (28)                             repr(groupname[groupcounter])),
729: (28)   filepositiontext)
730: (29)   )
731: (28)   if skipblocksuntil == groupcounter:
732: (8)   skipblocksuntil = -1
733: (12)   grouplist[groupcounter - 1].append(groupcache[groupcounter])
734: (8)   grouplist[groupcounter - 1][-1]['body'] = grouplist[groupcounter]
735: (8)   del grouplist[groupcounter]
736: (8)   groupcounter = groupcounter - 1
737: (8)   if not skipemptyends:
738: (8)   expectbegin = 1
739: (12)   elif pat[1] == 'begin':
740: (4)   if 0 <= skipblocksuntil <= groupcounter:
741: (8)   groupcounter = groupcounter + 1
742: (12)   return
743: (12)   gotnextfile = 0

```

```

745: (8)             analyzeline(m, pat[1], line)
746: (8)             expectbegin = 0
747: (4)             elif pat[1] == 'endif':
748: (8)                 pass
749: (4)             elif pat[1] == 'moduleprocedure':
750: (8)                 analyzeline(m, pat[1], line)
751: (4)             elif pat[1] == 'contains':
752: (8)                 if ignorecontains:
753: (12)                     return
754: (8)                 if 0 <= skipblocksuntil <= groupcounter:
755: (12)                     return
756: (8)                     skipblocksuntil = groupcounter
757: (4)             else:
758: (8)                 if 0 <= skipblocksuntil <= groupcounter:
759: (12)                     return
760: (8)                 analyzeline(m, pat[1], line)
761: (0)             def markouterparen(line):
762: (4)                 l = ''
763: (4)                 f = 0
764: (4)                 for c in line:
765: (8)                     if c == '(':
766: (12)                         f = f + 1
767: (12)                         if f == 1:
768: (16)                             l = l + '@(@'
769: (16)                             continue
770: (8)                     elif c == ')':
771: (12)                         f = f - 1
772: (12)                         if f == 0:
773: (16)                             l = l + '@)@'
774: (16)                             continue
775: (8)                     l = l + c
776: (4)                 return l
777: (0)             def markoutercomma(line, comma=','):
778: (4)                 l = ''
779: (4)                 f = 0
780: (4)                 before, after = split_by_unquoted(line, comma + '()')
781: (4)                 l += before
782: (4)                 while after:
783: (8)                     if (after[0] == comma) and (f == 0):
784: (12)                         l += '@' + comma + '@'
785: (8)                     else:
786: (12)                         l += after[0]
787: (12)                         if after[0] == '(':
788: (16)                             f += 1
789: (12)                         elif after[0] == ')':
790: (16)                             f -= 1
791: (8)                         before, after = split_by_unquoted(after[1:], comma + '()')
792: (8)                         l += before
793: (4)                     assert not f, repr((f, line, l))
794: (4)                     return l
795: (0)             def unmarkouterparen(line):
796: (4)                 r = line.replace('@(@', '(').replace('@)@', ')')
797: (4)                 return r
798: (0)             def appenddecl(decl, decl2, force=1):
799: (4)                 if not decl:
800: (8)                     decl = {}
801: (4)                 if not decl2:
802: (8)                     return decl
803: (4)                 if decl is decl2:
804: (8)                     return decl
805: (4)                 for k in list(decl2.keys()):
806: (8)                     if k == 'typespec':
807: (12)                         if force or k not in decl:
808: (16)                             decl[k] = decl2[k]
809: (8)                     elif k == 'attrspec':
810: (12)                         for l in decl2[k]:
811: (16)                             decl = setattrspec(decl, l, force)
812: (8)                     elif k == 'kindselector':
813: (12)                         decl = setkindselector(decl, decl2[k], force)

```

```

814: (8)           elif k == 'charselector':
815: (12)          decl = setcharselector(decl, decl2[k], force)
816: (8)          elif k in ['=', 'typename']:
817: (12)            if force or k not in decl:
818: (16)              decl[k] = decl2[k]
819: (8)          elif k == 'note':
820: (12)            pass
821: (8)          elif k in ['intent', 'check', 'dimension', 'optional',
822: (19)            'required', 'depend']:
823: (12)            errmess('appenddecl: "%s" not implemented.\n' % k)
824: (8)          else:
825: (12)            raise Exception('appenddecl: Unknown variable definition key: ' +
826: (28)                      str(k))
827: (4)          return decl
828: (0)          selectpattern = re.compile(
829: (4)            r'\s*(?P<this>(@(\.*?\@)|\*[\d*]+|\*\s*@\(\.*?\@|)) (?P<after>.*))\Z',
830: (0)          re.I)
831: (4)          typedefpattern = re.compile(
832: (4)            r'(?:(?P<attributes>[\w(),]+)?(:)?(?P<name>\b[a-z$_][\w$]*\b)|
833: (0)              r'(?:(?P<params>[\w,]*))?\Z', re.I)
834: (4)          nameargspattern = re.compile(
835: ((result(\s*@\(\@\s*(?P<result>\b[\w$]+)\b)\s*(@\(\@\s*(?P<args>[\w\s,]*))\s*@\()@|))\s*|bind\s*@\(\@\s*(?P<bind>(?:
836: (?!)@\().*)\s*@\())@\)*\s*\Z', re.I)
837: (0)          operatorpattern = re.compile(
838: (4)            r'\s*(?P<scheme>(operator|assignment))|
839: (4)              r'@\(\@\s*(?P<name>[^])+\s*@\()@\s*\Z', re.I)
840: (0)          callnameargspattern = re.compile(
841: (4)            r'\s*(?P<name>\b[\w$]+)\b)\s*@\(\@\s*(?P<args>.*))\s*@\()@\s*\Z', re.I)
842: (0)          real16pattern = re.compile(
843: (4)            r'([-+]?(:\d+(:\.\d*)?|\d*\.\d+))[dD]((?:[-+]?\d+)?))')
844: (0)          real8pattern = re.compile(
845: (4)            r'([-+]?((?:\d+(:\.\d*)?|\d*\.\d+))[eE]((?:[-+]?\d+)?))|(\d+\.\d*))')
846: (0)          _intentcallbackpattern = re.compile(r'intent\s*\(.*\bcallback\b', re.I)
847: (0)          def _is_intent_callback(vdecl):
848: (12)            for a in vdecl.get('attrspec', []):
849: (8)              if _intentcallbackpattern.match(a):
850: (4)                return 1
851: (8)            return 0
852: (0)          def _resolvetypedefpattern(line):
853: (4)            line = ''.join(line.split()) # removes whitespace
854: (4)            m1 = typedefpattern.match(line)
855: (4)            print(line, m1)
856: (4)            if m1:
857: (8)              attrs = m1.group('attributes')
858: (8)              attrs = [a.lower() for a in attrs.split(',') if attrs else []]
859: (8)              return m1.group('name'), attrs, m1.group('params')
860: (0)            return None, [], None
861: (0)          def parse_name_for_bind(line):
862: (4)            pattern = re.compile(r'bind\(\s*(?P<lang>[^,]+)(?:\s*,\s*name\s*=\s*["\']?
863: ((result(\s*@\(\@\s*(?P<result>\b[\w$]+)\b)\s*(@\(\@\s*(?P<args>[\w\s,]*))\s*@\()@|))\s*|bind\s*@\(\@\s*(?P<bind>(?:
864: (?!)@\().*)\s*@\())@\)*\s*\Z', re.I)
865: (0)            match = pattern.search(line)
866: (0)            bind_statement = None
867: (0)            if match:
868: (4)              bind_statement = match.group(0)
869: (4)              line = line[:match.start()] + line[match.end():]
870: (4)            return line, bind_statement
871: (0)          def _resolvenameargspattern(line):
872: (4)            line, bind_cname = parse_name_for_bind(line)
873: (4)            line = markouterparen(line)
874: (4)            m1 = nameargspattern.match(line)
875: (4)            if m1:
876: (8)              return m1.group('name'), m1.group('args'), m1.group('result'),
877: (0)            m1 = operatorpattern.match(line)
878: (4)            if m1:
879: (8)              name = m1.group('scheme') + '(' + m1.group('name') + ')'
880: (8)              return name, [], None, None
881: (4)            m1 = callnameargspattern.match(line)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

878: (4)
879: (8)
880: (4)
881: (0)
882: (4)
883: (4)
884: (4)
885: (4)
886: (4)
887: (4)
888: (4)
889: (4)
890: (4)
891: (4)
892: (4)
893: (4)
894: (4)
895: (8)
896: (4)
897: (7)
898: (8)
899: (8)
900: (12)
"%s.\n' % newname)
901: (8)
902: (8)
903: (8)
904: (8)
905: (8)
906: (8)
907: (8)
908: (8)
909: (8)
910: (8)
911: (8)
912: (4)
913: (8)
914: (8)
915: (12)
916: (8)
917: (12)
918: (8)
919: (12)
920: (8)
921: (12)
922: (12)
923: (12)
924: (12)
925: (8)
926: (12)
_resolvenameargspattern(m.group('after'))
927: (8)
if name is None:
    if block == 'block data':
        name = '_BLOCK_DATA_'
    else:
        name = ''
    if block not in ['interface', 'block data', 'abstract interface']:
        outmess('analyzeline: No name/args pattern found for line.\n')
previous_context = (block, name, groupcounter)
if args:
    args = rmbadname([x.strip()
                      for x in markoutercomma(args).split('@,@')])
else:
    args = []
if '' in args:
    while '' in args:
        args.remove('')
    outmess(
        'analyzeline: argument list is malformed (missing

```

```

argument).\n')
945: (8)             needmodule = 0
946: (8)             needinterface = 0
947: (8)             if case in ['call', 'callfun']:
948: (12)               needinterface = 1
949: (12)               if 'args' not in groupcache[groupcounter]:
950: (16)                 return
951: (12)               if name not in groupcache[groupcounter]['args']:
952: (16)                 return
953: (12)               for it in grouplist[groupcounter]:
954: (16)                 if it['name'] == name:
955: (20)                   return
956: (12)               if name in groupcache[groupcounter]['interfaced']:
957: (16)                   return
958: (12)               block = {'call': 'subroutine', 'callfun': 'function'}[case]
959: (8)             if f77modulename and neededmodule == -1 and groupcounter <= 1:
960: (12)               neededmodule = groupcounter + 2
961: (12)               needmodule = 1
962: (12)               if block not in ['interface', 'abstract interface']:
963: (16)                 needinterface = 1
964: (8)             groupcounter = groupcounter + 1
965: (8)             groupcache[groupcounter] = {}
966: (8)             grouplist[groupcounter] = []
967: (8)             if needmodule:
968: (12)               if verbose > 1:
969: (16)                 outmess('analyzeline: Creating module block %s\n' %
970: (24)                   repr(f77modulename), 0)
971: (12)               groupname[groupcounter] = 'module'
972: (12)               groupcache[groupcounter]['block'] = 'python module'
973: (12)               groupcache[groupcounter]['name'] = f77modulename
974: (12)               groupcache[groupcounter]['from'] = ''
975: (12)               groupcache[groupcounter]['body'] = []
976: (12)               groupcache[groupcounter]['externals'] = []
977: (12)               groupcache[groupcounter]['interfaced'] = []
978: (12)               groupcache[groupcounter]['vars'] = {}
979: (12)               groupcounter = groupcounter + 1
980: (12)               groupcache[groupcounter] = {}
981: (12)               grouplist[groupcounter] = []
982: (8)             if needinterface:
983: (12)               if verbose > 1:
984: (16)                 outmess('analyzeline: Creating additional interface block
(groupcounter=%s).\n' % (
985: (20)                   groupcounter), 0)
986: (12)               groupname[groupcounter] = 'interface'
987: (12)               groupcache[groupcounter]['block'] = 'interface'
988: (12)               groupcache[groupcounter]['name'] = 'unknown_interface'
989: (12)               groupcache[groupcounter]['from'] = '%s:%s' % (
990: (16)                   groupcache[groupcounter - 1]['from'], groupcache[groupcounter
- 1]['name'])
991: (12)               groupcache[groupcounter]['body'] = []
992: (12)               groupcache[groupcounter]['externals'] = []
993: (12)               groupcache[groupcounter]['interfaced'] = []
994: (12)               groupcache[groupcounter]['vars'] = {}
995: (12)               groupcounter = groupcounter + 1
996: (12)               groupcache[groupcounter] = {}
997: (12)               grouplist[groupcounter] = []
998: (8)             groupname[groupcounter] = block
999: (8)             groupcache[groupcounter]['block'] = block
1000: (8)             if not name:
1001: (12)               name = 'unknown_' + block.replace(' ', '_')
1002: (8)               groupcache[groupcounter]['prefix'] = m.group('before')
1003: (8)               groupcache[groupcounter]['name'] = rmbadname1(name)
1004: (8)               groupcache[groupcounter]['result'] = result
1005: (8)             if groupcounter == 1:
1006: (12)               groupcache[groupcounter]['from'] = currentfilename
1007: (8)             else:
1008: (12)               if f77modulename and groupcounter == 3:
1009: (16)                 groupcache[groupcounter]['from'] = '%s:%s' % (
1010: (20)                   groupcache[groupcounter - 1]['from'], currentfilename)

```

```

1011: (12)           else:
1012: (16)             groupcache[groupcounter]['from'] = '%s:%s' % (
1013: (20)               groupcache[groupcounter - 1]['from'],
groupcache[groupcounter - 1]['name'])
1014: (8)               for k in list(groupcache[groupcounter].keys()):
1015: (12)                 if not groupcache[groupcounter][k]:
1016: (16)                   del groupcache[groupcounter][k]
1017: (8)               groupcache[groupcounter]['args'] = args
1018: (8)               groupcache[groupcounter]['body'] = []
1019: (8)               groupcache[groupcounter]['externals'] = []
1020: (8)               groupcache[groupcounter]['interfaced'] = []
1021: (8)               groupcache[groupcounter]['vars'] = {}
1022: (8)               groupcache[groupcounter]['entry'] = {}
1023: (8)             if block == 'type':
1024: (12)               groupcache[groupcounter]['varnames'] = []
1025: (8)             if case in ['call', 'callfun']: # set parents variables
1026: (12)               if name not in groupcache[groupcounter - 2]['externals']:
1027: (16)                 groupcache[groupcounter - 2]['externals'].append(name)
1028: (12)               groupcache[groupcounter]['vars'] = copy.deepcopy(
1029: (16)                 groupcache[groupcounter - 2]['vars'])
1030: (12)             try:
1031: (16)               del groupcache[groupcounter]['vars'][name][
1032: (20)                 groupcache[groupcounter]['vars'][name]
['attrspec'].index('external')]

1033: (12)             except Exception:
1034: (16)               pass
1035: (8)             if block in ['function', 'subroutine']: # set global attributes
1036: (12)               if bindcline:
1037: (16)                 bindcdat = re.search(crackline_bindlang, bindcline)
1038: (16)                 if bindcdat:
1039: (20)                   groupcache[groupcounter]['bindlang'] = {name : {}}
1040: (20)                   groupcache[groupcounter]['bindlang'][name]["lang"] =
bindcdat.group('lang')
1041: (20)                   if bindcdat.group('lang_name'):
1042: (24)                     groupcache[groupcounter]['bindlang'][name]["name"] =
bindcdat.group('lang_name')

1043: (12)             try:
1044: (16)               groupcache[groupcounter]['vars'][name] = appenddecl(
1045: (20)                 groupcache[groupcounter]['vars'][name],
groupcache[groupcounter - 2]['vars'][()])
1046: (12)             except Exception:
1047: (16)               pass
1048: (12)             if case == 'callfun': # return type
1049: (16)               if result and result in groupcache[groupcounter]['vars']:
1050: (20)                 if not name == result:
1051: (24)                   groupcache[groupcounter]['vars'][name] = appenddecl(
1052: (28)                     groupcache[groupcounter]['vars'][name],
groupcache[groupcounter]['vars'][result])

1053: (12)             try:
1054: (16)               groupcache[groupcounter - 2]['interfaced'].append(name)
1055: (12)             except Exception:
1056: (16)               pass
1057: (8)             if block == 'function':
1058: (12)               t = typespattern[0].match(m.group('before') + ' ' + name)
1059: (12)               if t:
1060: (16)                 typespec, selector, attr, edecl = cracktypespec0(
1061: (20)                   t.group('this'), t.group('after'))
1062: (16)                   updatevars(typespec, selector, attr, edecl)
1063: (8)             if case in ['call', 'callfun']:
1064: (12)               grouplist[groupcounter - 1].append(groupcache[groupcounter])
1065: (12)               grouplist[groupcounter - 1][-1]['body'] = grouplist[groupcounter]
1066: (12)               del grouplist[groupcounter]
1067: (12)               groupcounter = groupcounter - 1 # end routine
1068: (12)               grouplist[groupcounter - 1].append(groupcache[groupcounter])
1069: (12)               grouplist[groupcounter - 1][-1]['body'] = grouplist[groupcounter]
1070: (12)               del grouplist[groupcounter]
1071: (12)               groupcounter = groupcounter - 1 # end interface
1072: (4)             elif case == 'entry':
1073: (8)               name, args, result, _ = _resolvenameargspattern(m.group('after'))

```

```

1074: (8)             if name is not None:
1075: (12)             if args:
1076: (16)                 args = rmbadname([x.strip()
1077: (34)                     for x in markoutercomma(args).split('@,@')]))
1078: (12)             else:
1079: (16)                 args = []
1080: (12)             assert result is None, repr(result)
1081: (12)             groupcache[groupcounter]['entry'][name] = args
1082: (12)             previous_context = ('entry', name, groupcounter)
1083: (4)             elif case == 'type':
1084: (8)                 typespec, selector, attr, edecl = cracktypespec0(
1085: (12)                     block, m.group('after'))
1086: (8)                 last_name = updatevars(typespec, selector, attr, edecl)
1087: (8)                 if last_name is not None:
1088: (12)                     previous_context = ('variable', last_name, groupcounter)
1089: (4)                 elif case in ['dimension', 'intent', 'optional', 'required', 'external',
1090: (8)                     'public', 'private', 'intrinsic']:
1091: (8)                     edecl = groupcache[groupcounter]['vars']
1092: (8)                     ll = m.group('after').strip()
1093: (8)                     i = ll.find('::')
1094: (12)                     if i < 0 and case == 'intent':
1095: (12)                         i = markouterparen(ll).find('@)@') - 2
1096: (12)                         ll = ll[:i + 1] + '::' + ll[i + 1:]
1097: (12)                         i = ll.find('::')
1098: (16)                         if ll[i:] == '::' and 'args' in groupcache[groupcounter]:
1099: (24)                             outmess('All arguments will have attribute %s%s\n' %
1100: (16)                                 (m.group('this'), ll[:i]))
1101: (8)                             ll = ll + ','.join(groupcache[groupcounter]['args'])
1102: (12)                     if i < 0:
1103: (12)                         i = 0
1104: (8)                         pl = ''
1105: (12)                     else:
1106: (12)                         pl = ll[:i].strip()
1107: (8)                         ll = ll[i + 2:]
1108: (8)                         ch = markoutercomma(pl).split('@,@')
1109: (12)                         if len(ch) > 1:
1110: (12)                             pl = ch[0]
1111: (16)                             outmess('analyzeline: cannot handle multiple attributes without
type specification. Ignoring %r.\n' (
1112: (8)                                 ',','.join(ch[1:])))

last_name = None
for e in [x.strip() for x in markoutercomma(ll).split('@,@')]:
    m1 = namepattern.match(e)
    if not m1:
        if case in ['public', 'private']:
            k = ''
        else:
            print(m.groupdict())
            outmess('analyzeline: no name pattern found in %s
statement for %s. Skipping.\n' %
1113: (8)                                 case, repr(e)))
1114: (12)                                 continue
1115: (12)             else:
1116: (16)                 k = rmbadname1(m1.group('name'))
1117: (20)                 if case in ['public', 'private'] and \
1118: (16)                     (k == 'operator' or k == 'assignment'):
1119: (20)                         k += m1.group('after')
1120: (20)                         if k not in edecl:
1121: (24)                             edecl[k] = {}
1122: (20)                         if case == 'dimension':
1123: (12)                             ap = case + m1.group('after')
1124: (16)                         if case == 'intent':
1125: (12)                             ap = m.group('this') + pl
1126: (15)                             if _intentcallbackpattern.match(ap):
1127: (16)                                 if k not in groupcache[groupcounter]['args']:
1128: (12)                                     if groupcounter > 1:
1129: (16)   if '__user__' not in groupcache[groupcounter - 2]
1130: (12)   outmess(
1131: (16)
1132: (12)
1133: (16)
1134: (16)
1135: (20)
1136: (24)
1137: (28)
['name']:
1138: (32)

```

```

1139: (36)   'analyzeline: missing __user__ module
(could be nothing)\n')
1140: (28)   if k != groupcache[groupcounter]['name']:
1141: (32)   outmess('analyzeline: appending
intent(callback) %s'   ' to %s arguments\n' % (k,
1142: (40)   groupcache[groupcounter]['args'].append(k)
groupcache[groupcounter]['name'])) 1143: (32)   else:
1144: (24)   errmess(
1145: (28)   'analyzeline: intent(callback) %s is
1146: (32)   ignored\n' % (k))
1147: (20)   else:
1148: (24)   errmess('analyzeline: intent(callback) %s is already
1149: (32)   ' in argument list\n' % (k))
1150: (12)   if case in ['optional', 'required', 'public', 'external',
'private', 'intrinsic']:
1151: (16)   ap = case
1152: (12)   if 'attrspec' in edecl[k]:
1153: (16)   edecl[k]['attrspec'].append(ap)
1154: (12)   else:
1155: (16)   edecl[k]['attrspec'] = [ap]
1156: (12)   if case == 'external':
1157: (16)   if groupcache[groupcounter]['block'] == 'program':
1158: (20)   outmess('analyzeline: ignoring program arguments\n')
1159: (20)   continue
1160: (16)   if k not in groupcache[groupcounter]['args']:
1161: (20)   continue
1162: (16)   if 'externals' not in groupcache[groupcounter]:
1163: (20)   groupcache[groupcounter]['externals'] = []
1164: (16)   groupcache[groupcounter]['externals'].append(k)
1165: (12)   last_name = k
1166: (8)   groupcache[groupcounter]['vars'] = edecl
1167: (8)   if last_name is not None:
1168: (12)   previous_context = ('variable', last_name, groupcounter)
1169: (4)   elif case == 'moduleprocedure':
1170: (8)   groupcache[groupcounter]['implementedby'] = \
1171: (12)   [x.strip() for x in m.group('after').split(',')]

1172: (4)   elif case == 'parameter':
1173: (8)   edecl = groupcache[groupcounter]['vars']
1174: (8)   ll = m.group('after').strip()[1:-1]
1175: (8)   last_name = None
1176: (8)   for e in markoutercomma(ll).split('@,@'):
1177: (12)   try:
1178: (16)   k, initexpr = [x.strip() for x in e.split('=')]
1179: (12)   except Exception:
1180: (16)   outmess(
1181: (20)   'analyzeline: could not extract name,expr in parameter
statement "%s" of "%s"\n' % (e, ll))
1182: (16)   continue
1183: (12)   params = get_parameters(edecl)
1184: (12)   k = rmbadname1(k)
1185: (12)   if k not in edecl:
1186: (16)   edecl[k] = {}
1187: (12)   if '=' in edecl[k] and (not edecl[k]['='] == initexpr):
1188: (16)   outmess('analyzeline: Overwriting the value of parameter "%s"
("%s") with "%s".\n' % (
1189: (20)   k, edecl[k]['='], initexpr))
1190: (12)   t = determineexprtype(initexpr, params)
1191: (12)   if t:
1192: (16)   if t.get('typespec') == 'real':
1193: (20)   tt = list(initexpr)
1194: (20)   for m in real16pattern.finditer(initexpr):
1195: (24)   tt[m.start():m.end()] = list(
1196: (28)   initexpr[m.start():m.end()].lower().replace('d',
'e'))
1197: (20)   initexpr = ''.join(tt)
1198: (16)   elif t.get('typespec') == 'complex':
1199: (20)   initexpr = initexpr[1:].lower().replace('d', 'e').\

```

```

1200: (24)                                replace(',', '+1j*(')
1201: (12)                                try:
1202: (16)                                v = eval(initexpr, {}, params)
1203: (12)                                except (SyntaxError, NameError, TypeError) as msg:
1204: (16)                                  errmess('analyzeline: Failed to evaluate %r. Ignoring: %s\n' %
1205: (24)                                      % (initexpr, msg))
1206: (16)                                continue
1207: (12)                                edecl[k]['='] = repr(v)
1208: (12)                                if 'attrspec' in edecl[k]:
1209: (16)                                  edecl[k]['attrspec'].append('parameter')
1210: (12)                                else:
1211: (16)                                  edecl[k]['attrspec'] = ['parameter']
1212: (12)                                last_name = k
1213: (8)                                 groupcache[groupcounter]['vars'] = edecl
1214: (8)                                if last_name is not None:
1215: (12)                                  previous_context = ('variable', last_name, groupcounter)
1216: (4)                                 elif case == 'implicit':
1217: (8)                                   if m.group('after').strip().lower() == 'none':
1218: (12)                                     groupcache[groupcounter]['implicit'] = None
1219: (8)                                   elif m.group('after'):
1220: (12)                                     if 'implicit' in groupcache[groupcounter]:
1221: (16)                                       impl = groupcache[groupcounter]['implicit']
1222: (12)                                     else:
1223: (16)                                       impl = {}
1224: (12)                                     if impl is None:
1225: (16)                                       outmess(
1226: (20)   'analyzeline: Overwriting earlier "implicit none" statement.\n')
1227: (16)                                       impl = {}
1228: (12)                                       for e in markoutercomma(m.group('after')).split('@,@'):
1229: (16)   decl = {}
1230: (16)   m1 = re.match(
1231: (20)   r'\s*(?P<this>.*?)\s*(\(\s*(?P<after>[a-zA-Z-]+)\s*\)\s*|)\s*\)|\Z', e, re.I)
1232: (16)   if not m1:
1233: (20)   outmess(
1234: (24)   'analyzeline: could not extract info of implicit statement part "%s"\n' % (e))
1235: (20)   continue
1236: (16)   m2 = typespattern4implicit.match(m1.group('this'))
1237: (16)   if not m2:
1238: (20)   outmess(
1239: (24)   'analyzeline: could not extract types pattern of implicit statement part "%s"\n' % (e))
1240: (20)   continue
1241: (16)   typespec, selector, attr, edecl = cracktypespec0(
1242: (20)   m2.group('this'), m2.group('after'))
1243: (16)   kindselect, chiselect, typename = cracktypespec(
1244: (20)   typespec, selector)
1245: (16)   decl['typespec'] = typespec
1246: (16)   decl['kindselector'] = kindselect
1247: (16)   decl['cheselect'] = chiselect
1248: (16)   decl['typename'] = typename
1249: (16)   for k in list(decl.keys()):
1250: (20)   if not decl[k]:
1251: (24)   del decl[k]
1252: (16)   for r in markoutercomma(m1.group('after')).split('@,@'):
1253: (20)   if '-' in r:
1254: (24)   try:
1255: (28)   begc, endc = [x.strip() for x in r.split('-')]
1256: (24)   except Exception:
1257: (28)   outmess(
1258: (32)   'analyzeline: expected "<char>-<char>" instead of "%s" in range list of implicit statement\n' % r)
1259: (28)   continue
1260: (20)   else:
1261: (24)   begc = endc = r.strip()
1262: (20)   if not len(begc) == len(endc) == 1:
1263: (24)   outmess(

```

```

1264: (28)                                'analyzeline: expected "<char>-<char>" instead of
"%s" in range list of implicit statement (2)\n' % r)
1265: (24)                                continue
1266: (20)                                for o in range(ord(begc), ord(endc) + 1):
1267: (24)                                impl[chr(o)] = decl
1268: (12)                                groupcache[groupcounter]['implicit'] = impl
1269: (4)                                 elif case == 'data':
1270: (8)                                 ll = []
1271: (8)                                 dl = ''
1272: (8)                                 il = ''
1273: (8)                                 f = 0
1274: (8)                                 fc = 1
1275: (8)                                 inp = 0
1276: (8)                                 for c in m.group('after'):
1277: (12)                                if not inp:
1278: (16)                                if c == '':
1279: (20)                                fc = not fc
1280: (16)                                if c == '/' and fc:
1281: (20)                                f = f + 1
1282: (20)                                continue
1283: (12)                                if c == '(':
1284: (16)                                inp = inp + 1
1285: (12)                                elif c == ')':
1286: (16)                                inp = inp - 1
1287: (12)                                if f == 0:
1288: (16)                                dl = dl + c
1289: (12)                                elif f == 1:
1290: (16)                                il = il + c
1291: (12)                                elif f == 2:
1292: (16)                                dl = dl.strip()
1293: (16)                                if dl.startswith(','):
1294: (20)                                dl = dl[1:].strip()
1295: (16)                                ll.append([dl, il])
1296: (16)                                dl = c
1297: (16)                                il = ''
1298: (16)                                f = 0
1299: (8)                                 if f == 2:
1300: (12)                                dl = dl.strip()
1301: (12)                                if dl.startswith(','):
1302: (16)                                dl = dl[1:].strip()
1303: (12)                                ll.append([dl, il])
1304: (8)                                 vars = groupcache[groupcounter].get('vars', {})
1305: (8)                                 last_name = None
1306: (8)                                 for l in ll:
1307: (12)                                l[0], l[1] = l[0].strip(), l[1].strip()
1308: (12)                                if l[0].startswith(','):
1309: (16)                                l[0] = l[0][1:]
1310: (12)                                if l[0].startswith('('):
1311: (16)                                outmess('analyzeline: implied-DO list "%s" is not supported.
Skipping.\n' % l[0])
1312: (16)                                continue
1313: (12)                                for idx, v in enumerate(rmbadname([x.strip() for x in
markoutercomma(l[0]).split('@,@')]))) :
1314: (16)                                if v.startswith('('):
1315: (20)                                outmess('analyzeline: implied-DO list "%s" is not
supported. Skipping.\n' % v)
1316: (20)                                continue
1317: (16)                                if '!' in l[1]:
1318: (20)                                outmess('Comment line in declaration "%s" is not
supported. Skipping.\n' % l[1])
1319: (20)                                continue
1320: (16)                                vars.setdefault(v, {})
1321: (16)                                vtype = vars[v].get('typespec')
1322: (16)                                vdim = getdimension(vars[v])
1323: (16)                                matches = re.findall(r"\(.?\)", l[1]) if vtype == 'complex'
else l[1].split(',')
1324: (16)                                try:
1325: (20)                                    new_val = "({})".format(", ".join(matches)) if vdim else
matches[idx]

```

```

1326: (16)             except IndexError:
1327: (20)                 if any("*" in m for m in matches):
1328: (24)                     expanded_list = []
1329: (24)                     for match in matches:
1330: (28)                         if "*" in match:
1331: (32)                             try:
1332: (36)                                 multiplier, value = match.split("*")
1333: (36)                                 expanded_list.extend([value.strip()])
1334: (32)             except ValueError: # if int(multiplier) fails
1335: (36)                     expanded_list.append(match.strip())
1336: (28)             else:
1337: (32)                     expanded_list.append(match.strip())
1338: (24)                     matches = expanded_list
1339: (20)                     new_val = "/{}/".format(", ".join(matches)) if vdim else
matches[idx]
1340: (16)                     current_val = vars[v].get('=')
1341: (16)                     if current_val and (current_val != new_val):
1342: (20)                         outmess('analyzeline: changing init expression of "%s"
("%s") to "%s"\n' % (v, current_val, new_val))
1343: (16)                         vars[v]['='] = new_val
1344: (16)                         last_name = v
1345: (8)                         groupcache[groupcounter]['vars'] = vars
1346: (8)                         if last_name:
1347: (12)                             previous_context = ('variable', last_name, groupcounter)
1348: (4)                         elif case == 'common':
1349: (8)                             line = m.group('after').strip()
1350: (8)                             if not line[0] == '/':
1351: (12)                                 line = '//' + line
1352: (8)                             cl = []
1353: (8)                             f = 0
1354: (8)                             bn = ''
1355: (8)                             ol = ''
1356: (8)                             for c in line:
1357: (12)                                 if c == '/':
1358: (16)                                     f = f + 1
1359: (16)                                     continue
1360: (12)                                 if f >= 3:
1361: (16)                                     bn = bn.strip()
1362: (16)                                     if not bn:
1363: (20)   bn = '_BLNK_'
1364: (16)   cl.append([bn, ol])
1365: (16)   f = f - 2
1366: (16)   bn = ''
1367: (16)   ol = ''
1368: (12)   if f % 2:
1369: (16)   bn = bn + c
1370: (12)   else:
1371: (16)   ol = ol + c
1372: (8)   bn = bn.strip()
1373: (8)   if not bn:
1374: (12)   bn = '_BLNK_'
1375: (8)   cl.append([bn, ol])
1376: (8)   commonkey = {}
1377: (8)   if 'common' in groupcache[groupcounter]:
1378: (12)   commonkey = groupcache[groupcounter]['common']
1379: (8)   for c in cl:
1380: (12)   if c[0] not in commonkey:
1381: (16)   commonkey[c[0]] = []
1382: (12)   for i in [x.strip() for x in markoutercomma(c[1]).split('@,@')]:
1383: (16)   if i:
1384: (20)   commonkey[c[0]].append(i)
1385: (8)   groupcache[groupcounter]['common'] = commonkey
1386: (8)   previous_context = ('common', bn, groupcounter)
1387: (4)   elif case == 'use':
1388: (8)   m1 = re.match(
1389: (12)   r'\A\s*(?P<name>\b\w+\b)\s*((,( \s*\bonly\b\s*:|(?P<notonly>)))\s*(?
P<list>.*))|\s*\Z', m.group('after'), re.I)
1390: (8)   if m1:

```

```

1391: (12)             mm = m1.groupdict()
1392: (12)             if 'use' not in groupcache[groupcounter]:
1393: (16)                 groupcache[groupcounter]['use'] = {}
1394: (12)             name = m1.group('name')
1395: (12)             groupcache[groupcounter]['use'][name] = {}
1396: (12)             isonly = 0
1397: (12)             if 'list' in mm and mm['list'] is not None:
1398: (16)                 if 'notonly' in mm and mm['notonly'] is None:
1399: (20)                     isonly = 1
1400: (16)                 groupcache[groupcounter]['use'][name]['only'] = isonly
1401: (16)                 ll = [x.strip() for x in mm['list'].split(',')]
1402: (16)                 rl = {}
1403: (16)                 for l in ll:
1404: (20)                     if '=' in l:
1405: (24)                         m2 = re.match(
1406: (28)                             r'^\A\s*(?P<local>\b\w+\b)\s*=\s*\s*(?
P<use>\b\w+\b)\s*\Z', l, re.I)
1407: (24)                         if m2:
1408: (28)                             rl[m2.group('local').strip()] = m2.group(
1409: (32)                                 'use').strip()
1410: (24)                         else:
1411: (28)                             outmess(
1412: (32)                                 'analyzeline: Not local=>use pattern found in
%$n' % repr(l))
1413: (20)                         else:
1414: (24)                             rl[l] = 1
1415: (20)                         groupcache[groupcounter]['use'][name]['map'] = rl
1416: (12)                         else:
1417: (16)                             pass
1418: (8)                         else:
1419: (12)                             print(m.groupdict())
1420: (12)                             outmess('analyzeline: Could not crack the use statement.\n')
1421: (4)                         elif case in ['f2pyenhancements']:
1422: (8)                             if 'f2pyenhancements' not in groupcache[groupcounter]:
1423: (12)                                 groupcache[groupcounter]['f2pyenhancements'] = {}
1424: (8)                             d = groupcache[groupcounter]['f2pyenhancements']
1425: (8)                             if m.group('this') == 'usercode' and 'usercode' in d:
1426: (12)                                 if isinstance(d['usercode'], str):
1427: (16)                                     d['usercode'] = [d['usercode']]
1428: (12)                                     d['usercode'].append(m.group('after'))
1429: (8)                             else:
1430: (12)                                 d[m.group('this')] = m.group('after')
1431: (4)                         elif case == 'multiline':
1432: (8)                             if previous_context is None:
1433: (12)                                 if verbose:
1434: (16)                                     outmess('analyzeline: No context for multiline block.\n')
1435: (12)                                 return
1436: (8)                             gc = groupcounter
1437: (8)                             appendmultiline(groupcache[gc],
1438: (24)                                 previous_context[:2],
1439: (24)                                 m.group('this'))
1440: (4)                         else:
1441: (8)                             if verbose > 1:
1442: (12)                                 print(m.groupdict())
1443: (12)                                 outmess('analyzeline: No code implemented for line.\n')
1444: (0)                         def appendmultiline(group, context_name, ml):
1445: (4)                             if 'f2pymultilines' not in group:
1446: (8)                                 group['f2pymultilines'] = {}
1447: (4)                             d = group['f2pymultilines']
1448: (4)                             if context_name not in d:
1449: (8)                                 d[context_name] = []
1450: (4)                             d[context_name].append(ml)
1451: (4)                             return
1452: (0)                         def cracktypespec0(typespec, ll):
1453: (4)                             selector = None
1454: (4)                             attr = None
1455: (4)                             if re.match(r'double\s*complex', typespec, re.I):
1456: (8)                                 typespec = 'double complex'
1457: (4)                             elif re.match(r'double\s*precision', typespec, re.I):

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1458: (8)           typespec = 'double precision'
1459: (4)           else:
1460: (8)             typespec = typespec.strip().lower()
1461: (4)             m1 = selectpattern.match(markouterparen(l1))
1462: (4)             if not m1:
1463: (8)               outmess(
1464: (12)                 'cracktypespec0: no kind/char_selector pattern found for line.\n')
1465: (8)               return
1466: (4)             d = m1.groupdict()
1467: (4)             for k in list(d.keys()):
1468: (8)               d[k] = unmarkouterparen(d[k])
1469: (4)             if typespec in ['complex', 'integer', 'logical', 'real', 'character',
1470: 'type']:
1471: (8)               selector = d['this']
1472: (4)               ll = d['after']
1473: (4)               i = ll.find('::')
1474: (8)               if i >= 0:
1475: (8)                 attr = ll[:i].strip()
1476: (4)                 ll = ll[i + 2:]
1477: (0)               return typespec, selector, attr, ll
1478: (0)             namepattern = re.compile(r'\s*(?P<name>\b\w+\b)\s*(?P<after>.*+)\s*\Z', re.I)
1479: (4)             kindselector = re.compile(
1480: r'\s*(\(\s*(kind\s*=)?\s*(?P<kind>.*+)\s*\)|\*\s*(?P<kind2>.*?))\s*\Z',
re.I)
1481: (4)             charselector = re.compile(
1482: r'\s*(\(((?P<lenkind>.*))|\*\s*(?P<charlen>.*))\s*\Z', re.I)
1483: (4)             lenkindpattern = re.compile(
1484: r'\s*(kind\s*=?\s*(?P<kind>.*?))\s*(@,@\s*len\s*=\s*(?P<len>.*))|'
1485: r'|(len\s*=\s*|)(?P<len2>.*?)\s*(@,@\s*(kind\s*=\s*)|(?P<kind2>.*?))|'
1486: r'|(f2py_len\s*=\s*(?P<f2py_len>.*))|))\s*\Z', re.I)
1487: (4)             lenarraypattern = re.compile(
1488: r'\s*(@\(@\s*(?!)/)\s*(?P<array>.*?)\s*@\()@\s*\*\s*(?P<len>.*?)|(\*\s*(?
P<len2>.*?))\s*(@\(@\s*(?!)/)\s*(?P<array2>.*?)\s*@\()@\))\s*(=\s*(?P<init>.*?))|(@\(@|)/\s*(?
P<init2>.*?))\s*/(@\(@|))\s*\Z', re.I)
1489: (0)             def removespaces(expr):
1490: (4)               expr = expr.strip()
1491: (4)               if len(expr) <= 1:
1492: (8)                 return expr
1493: (4)               expr2 = expr[0]
1494: (8)               for i in range(1, len(expr) - 1):
1495: (12)                 if (expr[i] == ' ' and
1496: ((expr[i + 1] in "([{}]=-/* ") or
1497: (expr[i - 1] in "([{}]=-/* "))) :
1498: (8)                   continue
1499: (4)                 expr2 = expr2 + expr[i]
1500: (4)               expr2 = expr2 + expr[-1]
1501: (0)             return expr2
1502: (4)             def markinnerspaces(line):
1503: (4)               """
1504: (4)                 The function replace all spaces in the input variable line which are
1505: (4)                 surrounded with quotation marks, with the triplet "@_@".
1506: (4)                 For instance, for the input "a 'b c'" the function returns "a 'b@_@c'"
1507: (4)                 Parameters
1508: (4)                   -----
1509: (4)                   line : str
1510: (4)                   Returns
1511: (4)                   -----
1512: (4)                   str
1513: (4)                   """
1514: (4)                   fragment = ''
1515: (4)                   inside = False
1516: (4)                   current_quote = None
1517: (4)                   escaped = ''
1518: (8)                   for c in line:
1519: (12)                     if escaped == '\\\' and c in ['\\\'', '\\'', '\"']:
1520: (12)                       fragment += c
1521: (12)                       escaped = c
1522: (8)                       continue
1523: (8)                     if not inside and c in ['\\'', '\"']:

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1523: (12)           current_quote = c
1524: (8)            if c == current_quote:
1525: (12)              inside = not inside
1526: (8)            elif c == ' ' and inside:
1527: (12)                fragment += '@_@'
1528: (12)                continue
1529: (8)            fragment += c
1530: (8)            escaped = c # reset to non-backslash
1531: (4)          return fragment
1532: (0)      def updatevars(typespec, selector, attrspec, entitydecl):
1533: (4)          """
1534: (4)              Returns last_name, the variable name without special chars, parenthesis
1535: (8)                  or dimension specifiers.
1536: (4)              Alters groupcache to add the name, typespec, attrspec (and possibly value)
1537: (4)                  of current variable.
1538: (4)          """
1539: (4)          global groupcache, groupcounter
1540: (4)          last_name = None
1541: (4)          kindselect, chiselect, typename = cracktypespec(typespec, selector)
1542: (4)          if attrspec:
1543: (8)              attrspec = [x.strip() for x in markoutercomma(attrspec).split('@,@')]
1544: (8)              l = []
1545: (8)              c = re.compile(r'(?P<start>[a-zA-Z]+)')
1546: (8)              for a in attrspec:
1547: (12)                  if not a:
1548: (16)                      continue
1549: (12)                  m = c.match(a)
1550: (12)                  if m:
1551: (16)                      s = m.group('start').lower()
1552: (16)                      a = s + a[len(s):]
1553: (12)                      l.append(a)
1554: (8)              attrspec = l
1555: (4)          el = [x.strip() for x in markoutercomma(entitydecl).split('@,@')]
1556: (4)          el1 = []
1557: (4)          for e in el:
1558: (8)              for e1 in [x.strip() for x in
markoutercomma(removespaces(markinnerspaces(e)), comma=' ').split('@ @')]:
1559: (12)                  if e1:
1560: (16)                      el1.append(e1.replace('@_@', ' '))
1561: (4)          for e in el1:
1562: (8)              m = namepattern.match(e)
1563: (8)              if not m:
1564: (12)                  outmess(
1565: (16)                      'updatevars: no name pattern found for entity=%s. Skipping.\n'
% (repr(e)))
1566: (12)                  continue
1567: (8)              ename = rmbadname1(m.group('name'))
1568: (8)              edecl = {}
1569: (8)              if ename in groupcache[groupcounter]['vars']:
1570: (12)                  edecl = groupcache[groupcounter]['vars'][ename].copy()
1571: (12)                  not_has_typespec = 'typespec' not in edecl
1572: (12)                  if not_has_typespec:
1573: (16)                      edecl['typespec'] = typespec
1574: (12)                  elif typespec and (not typespec == edecl['typespec']):
1575: (16)                      outmess('updatevars: attempt to change the type of "%s" ("%s")
to "%s". Ignoring.\n' %
1576: (20)                          ename, edecl['typespec'], typespec))
1577: (12)                  if 'kindselector' not in edecl:
1578: (16)                      edecl['kindselector'] = copy.copy(kindselect)
1579: (12)                  elif kindselect:
1580: (16)                      for k in list(kindselect.keys()):
1581: (20)                          if k in edecl['kindselector'] and (not kindselect[k] ==
edecl['kindselector'][k]):
1582: (24)                              outmess('updatevars: attempt to change the
kindselector "%s" of "%s" ("%s") to "%s". Ignoring.\n' %
1583: (28)                                  k, ename, edecl['kindselector'][k],
kindselect[k]))
1584: (20)
1585: (24)                          else:
edecl['kindselector'][k] = copy.copy(kindselect[k])

```

```

1586: (12)             if 'charselector' not in edecl and charselect:
1587: (16)                 if not_has_typespec:
1588: (20)                     edecl['charselector'] = charselect
1589: (16)                 else:
1590: (20)                     errmess('updatevars:%s: attempt to change empty
charselector to %r. Ignoring.\n'                                % (ename, charselect))
1591: (28)                         % (ename, charselect)
1592: (12)                     elif charselect:
1593: (16)                         for k in list(charselect.keys()):
1594: (20)                             if k in edecl['charselector'] and (not charselect[k] ==
edecl['charselector'][k]):
1595: (24)                                 outmess('updatevars: attempt to change the
charselector "%s" of "%s" ("%s") to "%s". Ignoring.\n' % (
1596: (28)   k, ename, edecl['charselector'][k],
charselect[k]))
1597: (20)                             else:
1598: (24)                                 edecl['charselector'][k] = copy.copy(charselect[k])
1599: (12)                         if 'typename' not in edecl:
1600: (16)                             edecl['typename'] = typename
1601: (12)                         elif typename and (not edecl['typename'] == typename):
1602: (16)                             outmess('updatevars: attempt to change the typename of "%s"
("%s") to "%s". Ignoring.\n' % (
1603: (20)   ename, edecl['typename'], typename))
1604: (12)                         if 'attrspec' not in edecl:
1605: (16)                             edecl['attrspec'] = copy.copy(attrspec)
1606: (12)                         elif attrspec:
1607: (16)                             for a in attrspec:
1608: (20)                                 if a not in edecl['attrspec']:
1609: (24)                                     edecl['attrspec'].append(a)
1610: (8)                         else:
1611: (12)                             edecl['typespec'] = copy.copy(typespec)
1612: (12)                             edecl['kindselector'] = copy.copy(kindselect)
1613: (12)                             edecl['charselector'] = copy.copy(charselect)
1614: (12)                             edecl['typename'] = typename
1615: (12)                             edecl['attrspec'] = copy.copy(attrspec)
1616: (8)                         if 'external' in (edecl.get('attrspec') or []) and e in
groupcache[groupcounter]['args']:
1617: (12)                             if 'externals' not in groupcache[groupcounter]:
1618: (16)                                 groupcache[groupcounter]['externals'] = []
1619: (12)                                 groupcache[groupcounter]['externals'].append(e)
1620: (8)                         if m.group('after'):
1621: (12)                             m1 = lenarraypattern.match(markouterparen(m.group('after')))
1622: (12)                             if m1:
1623: (16)                                 d1 = m1.groupdict()
1624: (16)                                 for lk in ['len', 'array', 'init']:
1625: (20)                                     if d1[lk + '2'] is not None:
1626: (24)   d1[lk] = d1[lk + '2']
1627: (24)   del d1[lk + '2']
1628: (16)                                     for k in list(d1.keys()):
1629: (20)   if d1[k] is not None:
1630: (24)   d1[k] = unmarkouterparen(d1[k])
1631: (20)   else:
1632: (24)   del d1[k]
1633: (16)                                     if 'len' in d1 and 'array' in d1:
1634: (20)   if d1['len'] == '':
1635: (24)   d1['len'] = d1['array']
1636: (24)   del d1['array']
1637: (20)   elif typespec == 'character':
1638: (24)   if ('charselector' not in edecl) or (not
edecl['charselector']):
1639: (28)   edecl['charselector'] = {}
1640: (24)   if 'len' in edecl['charselector']:
1641: (28)   del edecl['charselector']['len']
1642: (24)   edecl['charselector']['*'] = d1['len']
1643: (24)   del d1['len']
1644: (20)   else:
1645: (24)   d1['array'] = d1['array'] + ',' + d1['len']
1646: (24)   del d1['len']
1647: (24)   errmess('updatevars: "%s %s" is mapped to "%s'

```

```
%s(%s)"\n' % (
1648: (28)                                typespec, e, typespec, ename, d1['array']))
1649: (16)                                if 'len' in d1:
1650: (20)                                if typespec in ['complex', 'integer', 'logical', 'real']:
1651: (24)                                if ('kindselector' not in edecl) or (not
edecl['kindselector']):
1652: (28)                                edecl['kindselector'] = {}
1653: (24)                                edecl['kindselector']['*'] = d1['len']
1654: (24)                                del d1['len']
1655: (20)                                elif typespec == 'character':
1656: (24)                                if ('charselector' not in edecl) or (not
edecl['charselector']):
1657: (28)                                edecl['charselector'] = {}
1658: (24)                                if 'len' in edecl['charselector']:
1659: (28)                                    del edecl['charselector']['len']
1660: (24)                                    edecl['charselector']['*'] = d1['len']
1661: (24)                                    del d1['len']
1662: (16)                                if 'init' in d1:
1663: (20)                                if '=' in edecl and (not edecl['='] == d1['init']):
1664: (24)                                outmess('updatevars: attempt to change the init
expression of "%s" ("%s") to "%s". Ignoring.\n' %
1665: (28)                                    ename, edecl['='], d1['init']))
1666: (20)                                else:
1667: (24)                                    edecl['='] = d1['init']
1668: (16)                                if 'array' in d1:
1669: (20)                                    dm = 'dimension(%s)' % d1['array']
1670: (20)                                    if 'attrspec' not in edecl or (not edecl['attrspec']):
1671: (24)  edecl['attrspec'] = [dm]
1672: (20)                                else:
1673: (24)                                    edecl['attrspec'].append(dm)
1674: (24)                                    for dm1 in edecl['attrspec']:
1675: (28)  if dm1[:9] == 'dimension' and dm1 != dm:
1676: (32)  del edecl['attrspec'][-1]
1677: (32)  errmess('updatevars:%s: attempt to change %r
to %r. Ignoring.\n' %
1678: (40)  % (ename, dm1, dm))
1679: (32)  break
1680: (12)                                else:
1681: (16)                                outmess('updatevars: could not crack entity declaration "%s".
Ignoring.\n' %
1682: (20)                                    ename + m.group('after')))
1683: (8)                                for k in list(edecl.keys()):
1684: (12)                                    if not edecl[k]:
1685: (16)  del edecl[k]
1686: (8)                                groupcache[groupcounter]['vars'][ename] = edecl
1687: (8)                                if 'varnames' in groupcache[groupcounter]:
1688: (12)                                    groupcache[groupcounter]['varnames'].append(ename)
1689: (8)                                    last_name = ename
1690: (4)                                return last_name
1691: (0)                                def cracktypespec(typespec, selector):
1692: (4)                                    kindselect = None
1693: (4)                                    charselect = None
1694: (4)                                    typename = None
1695: (4)                                    if selector:
1696: (8)  if typespec in ['complex', 'integer', 'logical', 'real']:
1697: (12)  kindselect = kindselector.match(selector)
1698: (12)  if not kindselect:
1699: (16)  outmess(
1700: (20)  'cracktypespec: no kindselector pattern found for %s\n' %
(repr(selector)))
1701: (16)  return
1702: (12)  kindselect = kindselect.groupby()
1703: (12)  kindselect['*'] = kindselect['kind2']
1704: (12)  del kindselect['kind2']
1705: (12)  for k in list(kindselect.keys()):
1706: (16)  if not kindselect[k]:
1707: (20)  del kindselect[k]
1708: (12)  for k, i in list(kindselect.items()):
1709: (16)  kindselect[k] = rmbadname1(i)
```

```

1710: (8)           elif typespec == 'character':
1711: (12)          chiselect = chiselect.match(selector)
1712: (12)          if not chiselect:
1713: (16)            outmess(
1714: (20)              'cracktypespec: no chiselect pattern found for %s\n' %
(repr(selector)))
1715: (16)            return
1716: (12)          chiselect = chiselect.groupdict()
1717: (12)          chiselect['*'] = chiselect['charlen']
1718: (12)          del chiselect['charlen']
1719: (12)          if chiselect['lenkind']:
1720: (16)            lenkind = lenkindpattern.match(
1721: (20)              markoutercomma(chiselect['lenkind']))
1722: (16)            lenkind = lenkind.groupdict()
1723: (16)            for lk in ['len', 'kind']:
1724: (20)              if lenkind[lk + '2']:
1725: (24)                lenkind[lk] = lenkind[lk + '2']
1726: (20)                chiselect[lk] = lenkind[lk]
1727: (20)                del lenkind[lk + '2']
1728: (16)              if lenkind['f2py_len'] is not None:
1729: (20)                chiselect['f2py_len'] = lenkind['f2py_len']
1730: (12)              del chiselect['lenkind']
1731: (12)              for k in list(chiselect.keys()):
1732: (16)                if not chiselect[k]:
1733: (20)                  del chiselect[k]
1734: (12)                for k, i in list(chiselect.items()):
1735: (16)                  chiselect[k] = rmbadname1(i)
1736: (8)          elif typespec == 'type':
1737: (12)            typename = re.match(r'\s*\(\s*(?P<name>\w+)\s*\)', selector, re.I)
1738: (12)            if typename:
1739: (16)              typename = typename.group('name')
1740: (12)            else:
1741: (16)              outmess('cracktypespec: no typename found in %s\n' %
1742: (24)                  (repr(typespec + selector)))
1743: (8)            else:
1744: (12)              outmess('cracktypespec: no selector used for %s\n' %
1745: (20)                  (repr(selector)))
1746: (4)          return kindselect, chiselect, typename
1747: (0)      def setattrspec decl, attr, force=0):
1748: (4)        if not decl:
1749: (8)          decl = {}
1750: (4)        if not attr:
1751: (8)          return decl
1752: (4)        if 'attrspec' not in decl:
1753: (8)          decl['attrspec'] = [attr]
1754: (8)          return decl
1755: (4)        if force:
1756: (8)          decl['attrspec'].append(attr)
1757: (4)        if attr in decl['attrspec']:
1758: (8)          return decl
1759: (4)        if attr == 'static' and 'automatic' not in decl['attrspec']:
1760: (8)          decl['attrspec'].append(attr)
1761: (4)        elif attr == 'automatic' and 'static' not in decl['attrspec']:
1762: (8)          decl['attrspec'].append(attr)
1763: (4)        elif attr == 'public':
1764: (8)          if 'private' not in decl['attrspec']:
1765: (12)            decl['attrspec'].append(attr)
1766: (4)        elif attr == 'private':
1767: (8)          if 'public' not in decl['attrspec']:
1768: (12)            decl['attrspec'].append(attr)
1769: (4)        else:
1770: (8)          decl['attrspec'].append(attr)
1771: (4)        return decl
1772: (0)      def setkindselector decl, sel, force=0):
1773: (4)        if not decl:
1774: (8)          decl = {}
1775: (4)        if not sel:
1776: (8)          return decl
1777: (4)        if 'kindselector' not in decl:

```

```

1778: (8)             decl['kindselector'] = sel
1779: (8)             return decl
1780: (4)             for k in list(sel.keys()):
1781: (8)                 if force or k not in decl['kindselector']:
1782: (12)                   decl['kindselector'][k] = sel[k]
1783: (4)             return decl
1784: (0)             def setcharselector(decl, sel, force=0):
1785: (4)                 if not decl:
1786: (8)                     decl = {}
1787: (4)                 if not sel:
1788: (8)                     return decl
1789: (4)                 if 'charselector' not in decl:
1790: (8)                     decl['charselector'] = sel
1791: (8)                     return decl
1792: (4)                 for k in list(sel.keys()):
1793: (8)                     if force or k not in decl['charselector']:
1794: (12)                       decl['charselector'][k] = sel[k]
1795: (4)             return decl
1796: (0)             def getblockname(block, unknown='unknown'):
1797: (4)                 if 'name' in block:
1798: (8)                     return block['name']
1799: (4)                 return unknown
1800: (0)             def setmesstext(block):
1801: (4)                 global filepositiontext
1802: (4)                 try:
1803: (8)                     filepositiontext = 'In: %s:%s\n' % (block['from'], block['name'])
1804: (4)                 except Exception:
1805: (8)                     pass
1806: (0)             def get_usedict(block):
1807: (4)                 usedict = {}
1808: (4)                 if 'parent_block' in block:
1809: (8)                     usedict = get_usedict(block['parent_block'])
1810: (4)                 if 'use' in block:
1811: (8)                     usedict.update(block['use'])
1812: (4)             return usedict
1813: (0)             def get_useparameters(block, param_map=None):
1814: (4)                 global f90modulevars
1815: (4)                 if param_map is None:
1816: (8)                     param_map = {}
1817: (4)                 usedict = get_usedict(block)
1818: (4)                 if not usedict:
1819: (8)                     return param_map
1820: (4)                 for usename, mapping in list(usedict.items()):
1821: (8)                     usename = usename.lower()
1822: (8)                     if usename not in f90modulevars:
1823: (12)                         outmess('get_useparameters: no module %s info used by %s\n' %
1824: (20)                             (usename, block.get('name')))
1825: (12)                         continue
1826: (8)                     mvars = f90modulevars[usename]
1827: (8)                     params = get_parameters(mvars)
1828: (8)                     if not params:
1829: (12)                         continue
1830: (8)                     if mapping:
1831: (12)                         errmess('get_useparameters: mapping for %s not impl.\n' %
1832: (mapping))
1833: (8)                         for k, v in list(params.items()):
1834: (12)                             if k in param_map:
1835: (16)                                 outmess('get_useparameters: overriding parameter %s with'
1836: (24)                                     ' value from module %s\n' % (repr(k), repr(usename)))
1837: (12)                                     param_map[k] = v
1838: (4)                         return param_map
1839: (0)             def postcrack2(block, tab='', param_map=None):
1840: (4)                 global f90modulevars
1841: (4)                 if not f90modulevars:
1842: (8)                     return block
1843: (4)                 if isinstance(block, list):
1844: (8)                     ret = [postcrack2(g, tab=tab + '\t', param_map=param_map)
1845: (15)                         for g in block]
1846: (8)                     return ret

```

```

1846: (4)             setmesstext(block)
1847: (4)             outmess('%s\n' % (tab, block['name']), 0)
1848: (4)             if param_map is None:
1849: (8)                 param_map = get_useparameters(block)
1850: (4)             if param_map is not None and 'vars' in block:
1851: (8)                 vars = block['vars']
1852: (8)                 for n in list(vars.keys()):
1853: (12)                     var = vars[n]
1854: (12)                     if 'kindselector' in var:
1855: (16)                         kind = var['kindselector']
1856: (16)                         if 'kind' in kind:
1857: (20)                             val = kind['kind']
1858: (20)                             if val in param_map:
1859: (24)                                 kind['kind'] = param_map[val]
1860: (4)             new_body = [postcrack2(b, tab=tab + '\t', param_map=param_map)
1861: (16)                         for b in block['body']]
1862: (4)             block['body'] = new_body
1863: (4)             return block
1864: (0)             def postcrack(block, args=None, tab=''):
1865: (4)                 """
1866: (4)                 TODO:
1867: (10)                     function return values
1868: (10)                     determine expression types if in argument list
1869: (4)                 """
1870: (4)             global usermodules, onlyfunctions
1871: (4)             if isinstance(block, list):
1872: (8)                 gret = []
1873: (8)                 uret = []
1874: (8)                 for g in block:
1875: (12)                     setmesstext(g)
1876: (12)                     g = postcrack(g, tab=tab + '\t')
1877: (12)                     if 'name' in g and '__user__' in g['name']:
1878: (16)                         uret.append(g)
1879: (12)                     else:
1880: (16)                         gret.append(g)
1881: (8)                 return uret + gret
1882: (4)             setmesstext(block)
1883: (4)             if not isinstance(block, dict) and 'block' not in block:
1884: (8)                 raise Exception('postcrack: Expected block dictionary instead of ' +
1885: (24)                     str(block))
1886: (4)             if 'name' in block and not block['name'] == 'unknown_interface':
1887: (8)                 outmess('%s\n' % (tab, block['name']), 0)
1888: (4)             block = analyzeargs(block)
1889: (4)             block = analyzecommon(block)
1890: (4)             block['vars'] = analyzevars(block)
1891: (4)             block['sortvars'] = sortvarnames(block['vars'])
1892: (4)             if 'args' in block and block['args']:
1893: (8)                 args = block['args']
1894: (4)             block['body'] = analyzebody(block, args, tab=tab)
1895: (4)             userisdefined = []
1896: (4)             if 'use' in block:
1897: (8)                 useblock = block['use']
1898: (8)                 for k in list(useblock.keys()):
1899: (12)                     if '__user__' in k:
1900: (16)                         userisdefined.append(k)
1901: (4)             else:
1902: (8)                 useblock = {}
1903: (4)                 name = ''
1904: (4)                 if 'name' in block:
1905: (8)                     name = block['name']
1906: (4)                 if 'externals' in block and block['externals']:
1907: (8)                     interfaced = []
1908: (8)                     if 'interfaced' in block:
1909: (12)                         interfaced = block['interfaced']
1910: (8)                     mvars = copy.copy(block['vars'])
1911: (8)                     if name:
1912: (12)                         mname = name + '__user_routines'
1913: (8)                     else:
1914: (12)                         mname = 'unknown__user_routines'

```

```

1915: (8)             if mname in userisdefined:
1916: (12)             i = 1
1917: (12)             while '%s_%i' % (mname, i) in userisdefined:
1918: (16)                 i = i + 1
1919: (12)             mname = '%s_%i' % (mname, i)
1920: (8)             interface = {'block': 'interface', 'body': [], 
1921: (21)                     'vars': {}, 'name': name + '_user_interface'}
1922: (8)             for e in block['externals']:
1923: (12)                 if e in interfaced:
1924: (16)                     edef = []
1925: (16)                     j = -1
1926: (16)                     for b in block['body']:
1927: (20)                         j = j + 1
1928: (20)                         if b['block'] == 'interface':
1929: (24)                             i = -1
1930: (24)                             for bb in b['body']:
1931: (28)                                 i = i + 1
1932: (28)                                 if 'name' in bb and bb['name'] == e:
1933: (32)                                     edef = copy.copy(bb)
1934: (32)                                     del b['body'][i]
1935: (32)                                     break
1936: (24)                             if edef:
1937: (28)                                 if not b['body']:
1938: (32)                                     del block['body'][j]
1939: (28)                                     del interfaced[interfaced.index(e)]
1940: (28)                                     break
1941: (16)                             interface['body'].append(edef)
1942: (12)             else:
1943: (16)                 if e in mvars and not isexternal(mvars[e]):
1944: (20)                     interface['vars'][e] = mvars[e]
1945: (8)             if interface['vars'] or interface['body']:
1946: (12)                 block['interfaced'] = interfaced
1947: (12)                 mblock = {'block': 'python module', 'body': [
1948: (16)                     interface], 'vars': {}, 'name': mname, 'interfaced':
block['externals']}
1949: (12)                     useblock[mname] = {}
1950: (12)                     usermodules.append(mblock)
1951: (4)             if useblock:
1952: (8)                 block['use'] = useblock
1953: (4)             return block
1954: (0)             def sortvarnames(vars):
1955: (4)                 indep = []
1956: (4)                 dep = []
1957: (4)                 for v in list(vars.keys()):
1958: (8)                     if 'depend' in vars[v] and vars[v]['depend']:
1959: (12)                         dep.append(v)
1960: (8)                     else:
1961: (12)                         indep.append(v)
1962: (4)             n = len(dep)
1963: (4)             i = 0
1964: (4)             while dep: # XXX: How to catch dependence cycles correctly?
1965: (8)                 v = dep[0]
1966: (8)                 fl = 0
1967: (8)                 for w in dep[1:]:
1968: (12)                     if w in vars[v]['depend']:
1969: (16)                         fl = 1
1970: (16)                         break
1971: (8)             if fl:
1972: (12)                 dep = dep[1:] + [v]
1973: (12)                 i = i + 1
1974: (12)                 if i > n:
1975: (16)                     errmess('sortvarnames: failed to compute dependencies because'
1976: (24)                         ' of cyclic dependencies between '
1977: (24)                         '+ ', '.join(dep) + '\n')
1978: (16)                     indep = indep + dep
1979: (16)                     break
1980: (8)             else:
1981: (12)                 indep.append(v)
1982: (12)                 dep = dep[1:]

```

```

1983: (12)             n = len(dep)
1984: (12)             i = 0
1985: (4)              return indep
1986: (0)               def analyzecommon(block):
1987: (4)                 if not hascommon(block):
1988: (8)                   return block
1989: (4)                 commonvars = []
1990: (4)                 for k in list(block['common'].keys()):
1991: (8)                   comvars = []
1992: (8)                   for e in block['common'][k]:
1993: (12)                     m = re.match(
1994: (16)                       r'\A\s*\b(?P<name>.*?)\b\s*(\((?P<dims>.*?)\))\s*\Z', e,
re.I)
1995: (12)                     if m:
1996: (16)                       dims = []
1997: (16)                       if m.group('dims'):
1998: (20)                           dims = [x.strip()
1999: (28)                               for x in
markoutercomma(m.group('dims')).split('@,@')]
2000: (16)                           n = rmbadname1(m.group('name').strip())
2001: (16)                           if n in block['vars']:
2002: (20)                             if 'attrspec' in block['vars'][n]:
2003: (24)                               block['vars'][n]['attrspec'].append(
2004: (28)                                 'dimension(%s)' % (','.join(dims)))
2005: (20)                             else:
2006: (24)                               block['vars'][n]['attrspec'] = [
2007: (28)                                 'dimension(%s)' % (','.join(dims))]
2008: (16)                             else:
2009: (20)                               if dims:
2010: (24)                                 block['vars'][n] = {
2011: (28)                                   'attrspec': ['dimension(%s)' % (','.join(dims))]}
2012: (20)                               else:
2013: (24)                                 block['vars'][n] = {}
2014: (16)                               if n not in commonvars:
2015: (20)                                 commonvars.append(n)
2016: (12)                               else:
2017: (16)                                 n = e
2018: (16)                                 errmess(
2019: (20)                                   'analyzecommon: failed to extract "<name>[(<dims>)]" from
"%s" in common /%s/.\\n' % (e, k))
2020: (12)                                 comvars.append(n)
2021: (8)                                 block['common'][k] = comvars
2022: (4)                                 if 'commonvars' not in block:
2023: (8)                                   block['commonvars'] = commonvars
2024: (4)                                 else:
2025: (8)                                   block['commonvars'] = block['commonvars'] + commonvars
2026: (4)                                 return block
2027: (0)               def analyzebody(block, args, tab=''):
2028: (4)                 global usermodules, skipfuncs, onlyfuncs, f90modulevars
2029: (4)                 setmessstext(block)
2030: (4)                 maybe_private = {
2031: (8)                   key: value
2032: (8)                   for key, value in block['vars'].items()
2033: (8)                   if 'attrspec' not in value or 'public' not in value['attrspec']
2034: (4)                 }
2035: (4)                 body = []
2036: (4)                 for b in block['body']:
2037: (8)                   b['parent_block'] = block
2038: (8)                   if b['block'] in ['function', 'subroutine']:
2039: (12)                     if args is not None and b['name'] not in args:
2040: (16)                       continue
2041: (12)                     else:
2042: (16)                       as_ = b['args']
2043: (12)                       if b['name'] in maybe_private.keys():
2044: (16)                           skipfuncs.append(b['name'])
2045: (12)                           if b['name'] in skipfuncs:
2046: (16)                             continue
2047: (12)                             if onlyfuncs and b['name'] not in onlyfuncs:
2048: (16)                               continue

```

```

2049: (12)             b['saved_interface'] = crack2fortranen(
2050: (16)                 b, '\n' + ' ' * 6, as_interface=True)
2051: (8)             else:
2052: (12)                 as_ = args
2053: (8)             b = postcrack(b, as_, tab=tab + '\t')
2054: (8)             if b['block'] in ['interface', 'abstract interface'] and \
2055: (11)                 not b['body'] and not b.get('implementedby'):
2056: (12)                 if 'f2pyenhancements' not in b:
2057: (16)                     continue
2058: (8)             if b['block'].replace(' ', '') == 'pythonmodule':
2059: (12)                 usermodules.append(b)
2060: (8)             else:
2061: (12)                 if b['block'] == 'module':
2062: (16)                     f90modulevars[b['name']] = b['vars']
2063: (12)                     body.append(b)
2064: (4)             return body
2065: (0)         def buildimplicitrules(block):
2066: (4)             setmesstext(block)
2067: (4)             implicitrules = defaultimplicitrules
2068: (4)             attrrules = {}
2069: (4)             if 'implicit' in block:
2070: (8)                 if block['implicit'] is None:
2071: (12)                     implicitrules = None
2072: (12)                     if verbose > 1:
2073: (16)                         outmess(
2074: (20)                             'buildimplicitrules: no implicit rules for routine %s.\n'
% repr(block['name']))
2075: (8)             else:
2076: (12)                 for k in list(block['implicit'].keys()):
2077: (16)                     if block['implicit'][k].get('typespec') not in ['static',
2078: (20)                         'automatic']:
2079: (16)                         implicitrules[k] = block['implicit'][k]
2080: (20)                     else:
2081: (4)                         attrrules[k] = block['implicit'][k]['typespec']
2082: (0)             return implicitrules, attrrules
2083: (4)         def myeval(e, g=None, l=None):
2084: (4)             """ Like `eval` but returns only integers and floats """
2085: (4)             r = eval(e, g, l)
2086: (8)             if type(r) in [int, float]:
2087: (4)                 return r
2088: (0)             raise ValueError('r=%r' % (r))
2089: (0)         getlincoef_re_1 = re.compile(r'\A\b\w+\b\Z', re.I)
2090: (4)         def getlincoef(e, xset): # e = a*x+b ; x in xset
2091: (4)             """
2092: (4)                 Obtain ``a`` and ``b`` when ``e == "a*x+b"`` , where ``x`` is a symbol in
2093: (4)                 xset.
2094: (4)                 >>> getlincoef('2*x + 1', {'x'})
2095: (4)                 (2, 1, 'x')
2096: (4)                 >>> getlincoef('3*x + x*2 + 2 + 1', {'x'})
2097: (4)                 (5, 3, 'x')
2098: (4)                 >>> getlincoef('0', {'x'})
2099: (4)                 (0, 0, None)
2100: (4)                 >>> getlincoef('0*x', {'x'})
2101: (4)                 (0, 0, 'x')
2102: (4)                 >>> getlincoef('x*x', {'x'})
2103: (4)                 (None, None, None)
2104: (4)                 This can be tricked by sufficiently complex expressions
2105: (4)                 >>> getlincoef('(x - 0.5)*(x - 1.5)*(x - 1)*x + 2*x + 3', {'x'})
2106: (4)                 (2.0, 3.0, 'x')
2107: (4)                 """
2108: (8)                 try:
2109: (8)                     c = int(myeval(e, {}, {}))
2110: (4)                     return 0, c, None
2111: (8)                 except Exception:
2112: (4)                     pass
2113: (8)                 if getlincoef_re_1.match(e):
2114: (4)                     return 1, 0, e
2115: (4)                 len_e = len(e)
2116: (4)                 for x in xset:

```

```

2116: (8)           if len(x) > len_e:
2117: (12)         continue
2118: (8)         if re.search(r'\w\s*\([^\)]*\b' + x + r'\b', e):
2119: (12)           continue
2120: (8)         re_1 = re.compile(r'(?P<before>.*?)\b' + x + r'\b(?P<after>.*?)', re.I)
2121: (8)         m = re_1.match(e)
2122: (8)         if m:
2123: (12)           try:
2124: (16)             m1 = re_1.match(e)
2125: (16)             while m1:
2126: (20)               ee = '%s(%s)%s' % (
2127: (24)                 m1.group('before'), 0, m1.group('after'))
2128: (20)               m1 = re_1.match(ee)
2129: (16)               b = myeval(ee, {}, {})
2130: (16)               m1 = re_1.match(e)
2131: (16)               while m1:
2132: (20)                 ee = '%s(%s)%s' % (
2133: (24)                   m1.group('before'), 1, m1.group('after'))
2134: (20)                   m1 = re_1.match(ee)
2135: (16)                   a = myeval(ee, {}, {}) - b
2136: (16)                   m1 = re_1.match(e)
2137: (16)                   while m1:
2138: (20)                     ee = '%s(%s)%s' % (
2139: (24)                       m1.group('before'), 0.5, m1.group('after'))
2140: (20)                       m1 = re_1.match(ee)
2141: (16)                       c = myeval(ee, {}, {})
2142: (16)                       m1 = re_1.match(e)
2143: (16)                       while m1:
2144: (20)                         ee = '%s(%s)%s' % (
2145: (24)                           m1.group('before'), 1.5, m1.group('after'))
2146: (20)                           m1 = re_1.match(ee)
2147: (16)                           c2 = myeval(ee, {}, {})
2148: (16)                           if (a * 0.5 + b == c and a * 1.5 + b == c2):
2149: (20)                             return a, b, x
2150: (12)             except Exception:
2151: (16)               pass
2152: (12)             break
2153: (4)           return None, None, None
2154: (0)           word_pattern = re.compile(r'\b[a-z][\w$]*\b', re.I)
2155: (0)           def _get_depend_dict(name, vars, deps):
2156: (4)             if name in vars:
2157: (8)               words = vars[name].get('depend', [])
2158: (8)               if '=' in vars[name] and not isstring(vars[name]):
2159: (12)                 for word in word_pattern.findall(vars[name]['=']):
2160: (16)                   if word not in words and word in vars and word != name:
2161: (20)                     words.append(word)
2162: (8)               for word in words[:]:
2163: (12)                 for w in deps.get(word, []) \
2164: (20)                   or _get_depend_dict(word, vars, deps):
2165: (16)                     if w not in words:
2166: (20)                       words.append(w)
2167: (4)             else:
2168: (8)               outmess('_get_depend_dict: no dependence info for %s\n' %
2169: (8)                 (repr(name)))
2170: (4)               words = []
2171: (4)               deps[name] = words
2172: (4)               return words
2173: (0)           def _calc_depend_dict(vars):
2174: (4)             names = list(vars.keys())
2175: (4)             depend_dict = {}
2176: (4)             for n in names:
2177: (8)               _get_depend_dict(n, vars, depend_dict)
2178: (4)             return depend_dict
2179: (0)           def get_sorted_names(vars):
2180: (4)             depend_dict = _calc_depend_dict(vars)
2181: (4)             names = []
2182: (8)             for name in list(depend_dict.keys()):
2183: (12)               if not depend_dict[name]:
2184: (8)                 names.append(name)

```

```

2184: (12)           del depend_dict[name]
2185: (4)            while depend_dict:
2186: (8)             for name, lst in list(depend_dict.items()):
2187: (12)               new_lst = [n for n in lst if n in depend_dict]
2188: (12)               if not new_lst:
2189: (16)                 names.append(name)
2190: (16)                 del depend_dict[name]
2191: (12)               else:
2192: (16)                 depend_dict[name] = new_lst
2193: (4)             return [name for name in names if name in vars]
2194: (0)            def _kind_func(string):
2195: (4)              if string[0] in "'\"":
2196: (8)                string = string[1:-1]
2197: (4)              if real16pattern.match(string):
2198: (8)                return 8
2199: (4)              elif real8pattern.match(string):
2200: (8)                return 4
2201: (4)              return 'kind(' + string + ')'
2202: (0)            def _selected_int_kind_func(r):
2203: (4)              m = 10 ** r
2204: (4)              if m <= 2 ** 8:
2205: (8)                return 1
2206: (4)              if m <= 2 ** 16:
2207: (8)                return 2
2208: (4)              if m <= 2 ** 32:
2209: (8)                return 4
2210: (4)              if m <= 2 ** 63:
2211: (8)                return 8
2212: (4)              if m <= 2 ** 128:
2213: (8)                return 16
2214: (4)              return -1
2215: (0)            def _selected_real_kind_func(p, r=0, radix=0):
2216: (4)              if p < 7:
2217: (8)                return 4
2218: (4)              if p < 16:
2219: (8)                return 8
2220: (4)              machine = platform.machine().lower()
2221: (4)              if machine.startswith(('aarch64', 'alpha', 'arm64', 'loongarch', 'mips',
'power', 'ppc', 'riscv', 's390x', 'sparc')):
2222: (8)                if p <= 33:
2223: (12)                  return 16
2224: (4)                else:
2225: (8)                  if p < 19:
2226: (12)                    return 10
2227: (8)                    elif p <= 33:
2228: (12)                      return 16
2229: (4)                      return -1
2230: (0)            def get_parameters(vars, global_params={}):
2231: (4)              params = copy.copy(global_params)
2232: (4)              g_params = copy.copy(global_params)
2233: (4)              for name, func in [('_kind', _kind_func),
2234: (23)                  ('selected_int_kind', _selected_int_kind_func),
2235: (23)                  ('selected_real_kind', _selected_real_kind_func), ]:
2236: (8)                if name not in g_params:
2237: (12)                  g_params[name] = func
2238: (4)              param_names = []
2239: (4)              for n in get_sorted_names(vars):
2240: (8)                if 'attrspec' in vars[n] and 'parameter' in vars[n]['attrspec']:
2241: (12)                  param_names.append(n)
2242: (4)              kind_re = re.compile(r'\bkind\s*\(\s*(?P<value>.*\s*)\s*\)', re.I)
2243: (4)              selected_int_kind_re = re.compile(
2244: (8)                r'\bselected_int_kind\s*\(\s*(?P<value>.*\s*)\s*\)', re.I)
2245: (4)              selected_kind_re = re.compile(
2246: (8)                r'\bselected_(int|real)_kind\s*\(\s*(?P<value>.*\s*)\s*\)', re.I)
2247: (4)              for n in param_names:
2248: (8)                if '=' in vars[n]:
2249: (12)                  v = vars[n]['=']
2250: (12)                  if islogical(vars[n]):
2251: (16)                      v = v.lower()

```

```

2252: (16)             for repl in [
2253: (20)                 ('.false.', 'False'),
2254: (20)                 ('.true.', 'True'),
2255: (16)             ]:
2256: (20)                 v = v.replace(*repl)
2257: (12)             v = kind_re.sub(r'kind("\1")', v)
2258: (12)             v = selected_int_kind_re.sub(r'selected_int_kind(\1)', v)
2259: (12)             is_replaced = False
2260: (12)             if 'kindselector' in vars[n]:
2261: (16)                 if 'kind' in vars[n]['kindselector']:
2262: (20)                     orig_v_len = len(v)
2263: (20)                     v = v.replace('_' + vars[n]['kindselector']['kind'], '')
2264: (20)                     is_replaced = len(v) < orig_v_len
2265: (12)             if not is_replaced:
2266: (16)                 if not selected_kind_re.match(v):
2267: (20)                     v_ = v.split('_')
2268: (20)                     if len(v_) > 1:
2269: (24)                         v = ''.join(v_[:-1]).lower().replace(v_[-1].lower(),
2270: (12) ')
2271: (16)             if isdouble(vars[n]):
2272: (16)                 tt = list(v)
2273: (20)                 for m in real16pattern.finditer(v):
2274: (24)                     tt[m.start():m.end()] = list(
2275: (16)                         v[m.start():m.end()].lower().replace('d', 'e'))
2276: (12)             elif iscomplex(vars[n]):
2277: (16)                 outmess(f'get_parameters[TODO]: '
2278: (24)                     f'implement evaluation of complex expression {v}\n')
2279: (12)             dimspect = ([s.lstrip('dimension').strip()
2280: (24)                 for s in vars[n]['attrspec']
2281: (23)                     if s.startswith('dimension')] or [None])[0]
2282: (12)             if real16pattern.search(v):
2283: (16)                 v = 8
2284: (12)             elif real8pattern.search(v):
2285: (16)                 v = 4
2286: (12)             try:
2287: (16)                 params[n] = param_eval(v, g_params, params, dimspect=dimspect)
2288: (12)             except Exception as msg:
2289: (16)                 params[n] = v
2290: (16)                 outmess(f'get_parameters: got "{msg}" on {n!r}\n')
2291: (12)             if isstring(vars[n]) and isinstance(params[n], int):
2292: (16)                 params[n] = chr(params[n])
2293: (12)             nl = n.lower()
2294: (12)             if nl != n:
2295: (16)                 params[nl] = params[n]
2296: (8)             else:
2297: (12)                 print(vars[n])
2298: (12)                 outmess(f'get_parameters: parameter {n!r} does not have value?!\n')
2299: (4)             return params
2300: (0)         def _eval_length(length, params):
2301: (4)             if length in [':', '(*', '*']:
2302: (8)                 return '(*'
2303: (4)             return _eval_scalar(length, params)
2304: (0)         _is_kind_number = re.compile(r'\d+_').match
2305: (0)         def _eval_scalar(value, params):
2306: (4)             if _is_kind_number(value):
2307: (8)                 value = value.split('_')[0]
2308: (4)             try:
2309: (8)                 value = eval(value, {}, params)
2310: (8)                 value = (repr if isinstance(value, str) else str)(value)
2311: (4)             except (NameError, SyntaxError, TypeError):
2312: (8)                 return value
2313: (4)             except Exception as msg:
2314: (8)                 errmess('"%s" in evaluating %r '
2315: (16)                     '(available names: %s)\n'
2316: (16)                     % (msg, value, list(params.keys())))
2317: (4)             return value
2318: (0)         def analyzevars(block):
2319: (4)             """

```

```

2320: (4)          Sets correct dimension information for each variable/parameter
2321: (4)
2322: (4)
2323: (4)
2324: (4)
2325: (4)
2326: (4)
2327: (8)
2328: (4)
2329: (8)
2330: (8)
2331: (12)
2332: (12)
2333: (16)
2334: (20)
2335: (24)
2336: (4)
2337: (4)
2338: (4)
2339: (8)
2340: (12)
2341: (12)
2342: (8)
2343: (12)
2344: (4)
2345: (8)
2346: (12)
2347: (4)
2348: (4)
2349: (4)
2350: (4)
2351: (8)
2352: (8)
2353: (12)
2354: (12)
2355: (16)
2356: (12)
2357: (16)
2358: (4)
2359: (8)
2360: (12)
2361: (8)
2362: (12)
['attrspec']):
2363: (16)
2364: (20)
2365: (20)
2366: (24)
'undefined':
2367: (28)
2368: (24)
2369: (28)
2370: (24)
2371: (28)
2372: (32)
2373: (16)
2374: (20)
defined in routine %s.\n' %
2375: (24)          % (repr(n), block['name']))
2376: (8)          if 'charselector' in vars[n]:
2377: (12)          if 'len' in vars[n]['charselector']:
2378: (16)          l = vars[n]['charselector']['len']
2379: (16)          try:
2380: (20)              l = str(eval(l, {}, params))
2381: (16)          except Exception:
2382: (20)              pass
2383: (16)          vars[n]['charselector']['len'] = l
2384: (8)          if 'kindselector' in vars[n]:
2385: (12)              if 'kind' in vars[n]['kindselector']:

```

```

2386: (16)             l = vars[n]['kindselector']['kind']
2387: (16)         try:
2388: (20)             l = str(eval(l, {}, params))
2389: (16)         except Exception:
2390: (20)             pass
2391: (16)             vars[n]['kindselector']['kind'] = l
2392: (8)             dimension_expressions = {}
2393: (8)         if 'attrspec' in vars[n]:
2394: (12)             attr = vars[n]['attrspec']
2395: (12)             attr.reverse()
2396: (12)             vars[n]['attrspec'] = []
2397: (12)             dim, intent, depend, check, note = None, None, None, None, None
2398: (12)             for a in attr:
2399: (16)                 if a[:9] == 'dimension':
2400: (20)                     dim = (a[9:].strip())[1:-1]
2401: (16)                 elif a[:6] == 'intent':
2402: (20)                     intent = (a[6:].strip())[1:-1]
2403: (16)                 elif a[:6] == 'depend':
2404: (20)                     depend = (a[6:].strip())[1:-1]
2405: (16)                 elif a[:5] == 'check':
2406: (20)                     check = (a[5:].strip())[1:-1]
2407: (16)                 elif a[:4] == 'note':
2408: (20)                     note = (a[4:].strip())[1:-1]
2409: (16)             else:
2410: (20)                 vars[n] = setattrspec(vars[n], a)
2411: (16)         if intent:
2412: (20)             if 'intent' not in vars[n]:
2413: (24)                 vars[n]['intent'] = []
2414: (20)                 for c in [x.strip() for x in
markoutercomma(intent).split('@,@')]:
2415: (24)                     tmp = c.replace(' ', '')
2416: (24)                     if tmp not in vars[n]['intent']:
2417: (28)                         vars[n]['intent'].append(tmp)
2418: (20)                     intent = None
2419: (16)                 if note:
2420: (20)                     note = note.replace('\\n\\n', '\n\n')
2421: (20)                     note = note.replace('\\n ', '\n')
2422: (20)                     if 'note' not in vars[n]:
2423: (24)                         vars[n]['note'] = [note]
2424: (20)                     else:
2425: (24)                         vars[n]['note'].append(note)
2426: (20)                     note = None
2427: (16)                 if depend is not None:
2428: (20)                     if 'depend' not in vars[n]:
2429: (24)                         vars[n]['depend'] = []
2430: (20)                         for c in rmbadname([x.strip() for x in
markoutercomma(depend).split('@,@')]):
2431: (24)                             if c not in vars[n]['depend']:
2432: (28)                                 vars[n]['depend'].append(c)
2433: (20)                             depend = None
2434: (16)                         if check is not None:
2435: (20)                             if 'check' not in vars[n]:
2436: (24)                                 vars[n]['check'] = []
2437: (20)                                 for c in [x.strip() for x in
markoutercomma(check).split('@,@')]:
2438: (24)                                     if c not in vars[n]['check']:
2439: (28)   vars[n]['check'].append(c)
2440: (20)   check = None
2441: (12)   if dim and 'dimension' not in vars[n]:
2442: (16)   vars[n]['dimension'] = []
2443: (16)   for d in rmbadname(
2444: (24)   [x.strip() for x in markoutercomma(dim).split('@,@')])
2445: (16)   ):
2446: (20)   try:
2447: (24)   d = param_parse(d, params)
2448: (20)   except (ValueError, IndexError, KeyError):
2449: (24)   outmess(
2450: (28)   ('analyzevars: could not parse dimension for '
2451: (28)   f'variable {d!r}\n')

```

```

2452: (24)
2453: (20)
2454: (20)
2455: (24)
2456: (20)
2457: (24)
2458: (20)
2459: (24)
2460: (24)
2461: (20)
2462: (24)
2463: (20)
2464: (24)
2465: (24)
2466: (24)
2467: (24)
2468: (24)
2469: (28)
2470: (28)
2471: (32)
2472: (36)
2473: (36)
2474: (40)
2475: (36)
2476: (36)
2477: (32)
2478: (36)
2479: (36)
2480: (32)
2481: (36)
2482: (36)
2483: (32)
2484: (24)
2485: (20)
2486: (8)
2487: (12)
2488: (12)
2489: (12)
2490: (30)
2491: (12)
2492: (16)
2493: (20)
2494: (20)
2495: (24)
2496: (20)
2497: (24)
2498: (28)
2499: (32)
[1]:
2500: (36)
2501: (40)
2502: (40)
2503: (28)
2504: (28)
2505: (28)
2506: (33)
2507: (33)
2508: (32)
2509: (28)
2510: (32)
2511: (32)
2512: (36)
2513: (32)
2514: (36)
2515: (32)
2516: (32)
2517: (32)
2518: (36)
2519: (40)

)
dim_char = ':' if d == ':' else '*'
if d == dim_char:
    dl = [dim_char]
else:
    dl = markoutercomma(d, ':').split('@:@')
if len(dl) == 2 and '*' in dl: # e.g. dimension(5:*)
    dl = ['*']
    d = '*'
if len(dl) == 1 and dl[0] != dim_char:
    dl = ['1', dl[0]]
if len(dl) == 2:
    d1, d2 = map(symbolic.Expr.parse, dl)
    dsize = d2 - d1 + 1
    d = dsize.tostring(language=symbolic.Language.C)
    solver_and_deps = {}
    for v in block['vars']:
        s = symbolic.as_symbol(v)
        if dsize.contains(s):
            try:
                a, b = dsize.linear_solve(s)
                def solve_v(s, a=a, b=b):
                    return (s - b) / a
                all_symbols = set(a.symbols())
                all_symbols.update(b.symbols())
            except RuntimeError as msg:
                solve_v = None
                all_symbols = set(dsize.symbols())
            v_deps = set()
            s.data for s in all_symbols
            if s.data in vars):
                solver_and_deps[v] = solve_v, list(v_deps)
dimension_exprs[d] = solver_and_deps
vars[n]['dimension'].append(d)
if 'check' not in vars[n] and 'args' in block and n in block['args']:
    n_deps = vars[n].get('depend', [])
    n_checks = []
    n_is_input = l_or(isintent_in, isintent inout,
                      isintent inplace)(vars[n])
    if isArray(vars[n]): # n is array
        for i, d in enumerate(vars[n]['dimension']):
            coeffs_and_deps = dimension_exprs.get(d)
            if coeffs_and_deps is None:
                pass
            elif n_is_input:
                for v, (solver, deps) in coeffs_and_deps.items():
                    def compute_deps(v, deps):
                        for v1 in coeffs_and_deps.get(v, [None, []]):
                            if v1 not in deps:
                                deps.add(v1)
                                compute_deps(v1, deps)
all_deps = set()
compute_deps(v, all_deps)
if ((v in n_deps
      or '=' in vars[v]
      or 'depend' in vars[v])):
    continue
if solver is not None and v not in all_deps:
    is_required = False
    init = solver(symbolic.as_symbol(
        f'shape({n}, {i})'))
    init = init.tostring(
        language=symbolic.Language.C)
    vars[v]['='] = init
    vars[v]['depend'] = [n] + deps
    if 'check' not in vars[v]:
        vars[v]['check'] = [
            f'shape({n}, {i}) == {d}']
```

```

2520: (28)
2521: (32)
2522: (32)
2523: (36)
2524: (32)
2525: (36)
2526: (32)
2527: (32)
2528: (36)
2529: (28)
2530: (28)
2531: (36)
2532: (32)
2533: (36)
2534: (28)
2535: (32)
2536: (20)
2537: (24)
2538: (28)
2539: (28)
2540: (32)
2541: (36)
2542: (36)
2543: (28)
2544: (32)
2545: (28)
2546: (32)
2547: (12)
2548: (16)
2549: (20)
2550: (24)
2551: (46)
2552: (24)
2553: (20)
2554: (24)
2555: (46)
2556: (24)
2557: (24)
2558: (12)
2559: (16)
2560: (12)
2561: (16)
2562: (8)
2563: (12)
2564: (16)
2565: (12)
2566: (15)
2567: (16)
2568: (12)
2569: (16)
2570: (16)
2571: (20)
2572: (24)
2573: (16)
2574: (20)
2575: (12)
2576: (16)
2577: (4)
2578: (8)
2579: (12)
2580: (16)
2581: (12)
2582: (16)
2583: (20)
2584: (16)
2585: (20)
2586: (20)
2587: (20)
2588: (20)

        else:
            is_required = True
            if 'intent' not in vars[v]:
                vars[v]['intent'] = []
            if 'in' not in vars[v]['intent']:
                vars[v]['intent'].append('in')
            n_deps.append(v)
            n_checks.append(
                f'shape({n}, {i}) == {d}')
        v_attr = vars[v].get('attrspec', [])
        if not ('optional' in v_attr
                or 'required' in v_attr):
            v_attr.append(
                'required' if is_required else 'optional')
    if v_attr:
        vars[v]['attrspec'] = v_attr
    if coeffs_and_deps is not None:
        for v, (solver, deps) in coeffs_and_deps.items():
            v_deps = vars[v].get('depend', [])
            for aa in vars[v].get('attrspec', []):
                if aa.startswith('depend'):
                    aa = ''.join(aa.split())
                    v_deps.extend(aa[7:-1].split(','))
            if v_deps:
                vars[v]['depend'] = list(set(v_deps))
            if n not in v_deps:
                n_deps.append(v)
    elif isstring(vars[n]):
        if 'charselector' in vars[n]:
            if '*' in vars[n]['charselector']:
                length = _eval_length(vars[n]['charselector']['*'],
                                      params)
                vars[n]['charselector']['*'] = length
            elif 'len' in vars[n]['charselector']:
                length = _eval_length(vars[n]['charselector']['len'],
                                      params)
                del vars[n]['charselector']['len']
                vars[n]['charselector']['*'] = length
        if n_checks:
            vars[n]['check'] = n_checks
        if n_deps:
            vars[n]['depend'] = list(set(n_deps))
    if '=' in vars[n]:
        if 'attrspec' not in vars[n]:
            vars[n]['attrspec'] = []
        if ('optional' not in vars[n]['attrspec']) and \
           ('required' not in vars[n]['attrspec']):
            vars[n]['attrspec'].append('optional')
    if 'depend' not in vars[n]:
        vars[n]['depend'] = []
        for v, m in list(dep_matches.items()):
            if m(vars[n]['=']):
                vars[n]['depend'].append(v)
        if not vars[n]['depend']:
            del vars[n]['depend']
    if isscalar(vars[n]):
        vars[n]['='] = _eval_scalar(vars[n]['='], params)
for n in list(vars.keys()):
    if n == block['name']: # n is block name
        if 'note' in vars[n]:
            block['note'] = vars[n]['note']
        if block['block'] == 'function':
            if 'result' in block and block['result'] in vars:
                vars[n] = appenddecl(vars[n], vars[block['result']])
            if 'prefix' in block:
                pr = block['prefix']
                pr1 = pr.replace('pure', '')
                ispure = (not pr == pr1)
                pr = pr1.replace('recursive', '')

```

```

2589: (20)                     isrec = (not pr == pr1)
2590: (20)                     m = typespattern[0].match(pr)
2591: (20)                     if m:
2592: (24)                         typespec, selector, attr, edecl = cracktypespec0(
2593: (28)                             m.group('this'), m.group('after'))
2594: (24)                         kindselect, chiselect, typename = cracktypespec(
2595: (28)                             typespec, selector)
2596: (24)                         vars[n]['typespec'] = typespec
2597: (24)                     try:
2598: (28)                         if block['result']:
2599: (32)                             vars[block['result']]['typespec'] = typespec
2600: (24)                     except Exception:
2601: (28)                         pass
2602: (24)                     if kindselect:
2603: (28)                         if 'kind' in kindselect:
2604: (32)                             try:
2605: (36)                                 kindselect['kind'] = eval(
2606: (40)                                     kindselect['kind'], {}, params)
2607: (32)                             except Exception:
2608: (36)                                 pass
2609: (28)                             vars[n]['kindselector'] = kindselect
2610: (24)                     if chiselect:
2611: (28)                         vars[n]['cheselect'] = chiselect
2612: (24)                     if typename:
2613: (28)                         vars[n]['typename'] = typename
2614: (24)                     if ispure:
2615: (28)                         vars[n] = setattrspec(vars[n], 'pure')
2616: (24)                     if isrec:
2617: (28)                         vars[n] = setattrspec(vars[n], 'recursive')
2618: (20)                     else:
2619: (24)                         outmess(
2620: (28)                             'analyzevars: prefix (%s) were not used\n' %
repr(block['prefix']))
2621: (4)                     if not block['block'] in ['module', 'pythonmodule', 'python module',
'block data']:
2622: (8)                     if 'commonvars' in block:
2623: (12)                         neededvars = copy.copy(block['args'] + block['commonvars'])
2624: (8)
2625: (12)                     else:
2626: (8)                         neededvars = copy.copy(block['args'])
2627: (12)                     for n in list(vars.keys()):
2628: (16)                         if l_or(isintent_callback, isintent_aux)(vars[n]):
2629: (8)                             neededvars.append(n)
2630: (12)                     if 'entry' in block:
2631: (12)                         neededvars.extend(list(block['entry'].keys()))
2632: (16)                         for k in list(block['entry'].keys()):
2633: (20)                             for n in block['entry'][k]:
2634: (24)                                 if n not in neededvars:
2635: (8)                                     neededvars.append(n)
2636: (12)                     if block['block'] == 'function':
2637: (16)                         if 'result' in block:
2638: (12)                             neededvars.append(block['result'])
2639: (16)                         else:
2640: (8)                             neededvars.append(block['name'])
2641: (12)                     if block['block'] in ['subroutine', 'function']:
2642: (12)                         name = block['name']
2643: (16)                         if name in vars and 'intent' in vars[name]:
2644: (8)                             block['intent'] = vars[name]['intent']
2645: (12)                     if block['block'] == 'type':
2646: (8)                         neededvars.extend(list(vars.keys()))
2647: (12)                     for n in list(vars.keys()):
2648: (16)                         if n not in neededvars:
2649: (4)                             del vars[n]
2650: (0)                     return vars
2651: (0)                     analyzeargs_re_1 = re.compile(r'\A[a-z]+[\w$]*\Z', re.I)
2652: (4)                     def param_eval(v, g_params, params, dimspec=None):
2653: (4)                         """
2654: (4)                             Creates a dictionary of indices and values for each parameter in a
2655: (4)                             parameter array to be evaluated later.
2656: (4)                             WARNING: It is not possible to initialize multidimensional array

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2656: (4) parameters e.g. dimension(-3:1, 4, 3:5) at this point. This is because in
2657: (4) Fortran initialization through array constructor requires the RESHAPE
2658: (4) intrinsic function. Since the right-hand side of the parameter declaration
2659: (4) is not executed in f2py, but rather at the compiled c/fortran extension,
2660: (4) later, it is not possible to execute a reshape of a parameter array.
2661: (4) One issue remains: if the user wants to access the array parameter from
2662: (4) python, we should either
2663: (4) 1) allow them to access the parameter array using python standard indexing
2664: (7) (which is often incompatible with the original fortran indexing)
2665: (4) 2) allow the parameter array to be accessed in python as a dictionary with
2666: (7) fortran indices as keys
2667: (4) We are choosing 2 for now.
2668: (4)
2669: (4) """
2670: (8)     if dimspec is None:
2671: (12)         try:
2672: (8)             p = eval(v, g_params, params)
2673: (12)         except Exception as msg:
2674: (8)             p = v
2675: (12)             outmess(f'param_eval: got "{msg}" on {v!r}\n')
2676: (8)         return p
2677: (4)     if len(dimspec) < 2 or dimspec[::len(dimspec)-1] != "()":
2678: (8)         raise ValueError(f'param_eval: dimension {dimspec} can\'t be parsed')
2679: (4)     dimrange = dimspec[1:-1].split(',')
2680: (4)     if len(dimrange) == 1:
2681: (8)         dimrange = dimrange[0].split(':')
2682: (12)         if len(dimrange) == 1:
2683: (12)             bound = param_parse(dimrange[0], params)
2684: (8)             dimrange = range(1, int(bound)+1)
2685: (12)         else:
2686: (12)             lbound = param_parse(dimrange[0], params)
2687: (12)             ubound = param_parse(dimrange[1], params)
2688: (4)             dimrange = range(int(lbound), int(ubound)+1)
2689: (8)     else:
2690: (25)         raise ValueError(f'param_eval: multidimensional array parameters '
2691: (4)                         '{dimspec} not supported')
2692: (4)     v = (v[2:-2] if v.startswith('/') else v).split(',')
2693: (4)     v_eval = []
2694: (8)     for item in v:
2695: (12)         try:
2696: (8)             item = eval(item, g_params, params)
2697: (12)         except Exception as msg:
2698: (8)             outmess(f'param_eval: got "{msg}" on {item!r}\n')
2699: (4)             v_eval.append(item)
2700: (4)     p = dict(zip(dimrange, v_eval))
2701: (0)     return p
2702: (4) def param_parse(d, params):
2703: (4)     """Recursively parse array dimensions.
2704: (4)     Parses the declaration of an array variable or parameter
2705: (4)     `dimension` keyword, and is called recursively if the
2706: (4)     dimension for this array is a previously defined parameter
2707: (4)     (found in `params`).
2708: (4)     Parameters
2709: (4)     -----
2710: (8)         d : str
2711: (4)             Fortran expression describing the dimension of an array.
2712: (8)         params : dict
2713: (4)             Previously parsed parameters declared in the Fortran source file.
2714: (4)     Returns
2715: (4)     -----
2716: (8)         out : str
2717: (4)             Parsed dimension expression.
2718: (4)     Examples
2719: (4)     -----
2720: (6)         * If the line being analyzed is
2721: (6)             `integer, parameter, dimension(2) :: pa = (/ 3, 5 /)`
2722: (6)             then `d = 2` and we return immediately, with
2723: (4)             >>> d = '2'
2724: (4)             >>> param_parse(d, params)
2725: (4)             2

```

```

2725: (4) * If the line being analyzed is
2726: (6)   `integer, parameter, dimension(pa) :: pb = (/1, 2, 3/)`
2727: (6)   then `d = 'pa'`; since `pa` is a previously parsed parameter,
2728: (6)   and `pa = 3`, we call `param_parse` recursively, to obtain
2729: (4)   >>> d = 'pa'
2730: (4)   >>> params = {'pa': 3}
2731: (4)   >>> param_parse(d, params)
2732: (4)   3
2733: (4)   * If the line being analyzed is
2734: (6)   `integer, parameter, dimension(pa(1)) :: pb = (/1, 2, 3/)`
2735: (6)   then `d = 'pa(1)'`; since `pa` is a previously parsed parameter,
2736: (6)   and `pa(1) = 3`, we call `param_parse` recursively, to obtain
2737: (4)   >>> d = 'pa(1)'
2738: (4)   >>> params = dict(pa={1: 3, 2: 5})
2739: (4)   >>> param_parse(d, params)
2740: (4)   3
2741: (4)   """
2742: (4)   if "(" in d:
2743: (8)     dname = d[:d.find("(")]
2744: (8)     ddims = d[d.find("(")+1:d.rfind(")")]
2745: (8)     index = int(param_parse(ddims, params))
2746: (8)     return str(params[dname][index])
2747: (4)   elif d in params:
2748: (8)     return str(params[d])
2749: (4)   else:
2750: (8)     for p in params:
2751: (12)       re_1 = re.compile(
2752: (16)         r'(?P<before>.*?)\b' + p + r'\b(?P<after>.*?)', re.I
2753: (12)       )
2754: (12)       m = re_1.match(d)
2755: (12)       while m:
2756: (16)         d = m.group('before') + \
2757: (20)           str(params[p]) + m.group('after')
2758: (16)         m = re_1.match(d)
2759: (8)     return d
2760: (0) def expr2name(a, block, args=[]):
2761: (4)   orig_a = a
2762: (4)   a_is_expr = not analyzeargs_re_1.match(a)
2763: (4)   if a_is_expr: # `a` is an expression
2764: (8)     implicitrules, attrrules = buildimplicitrules(block)
2765: (8)     at = determineexprtype(a, block['vars'], implicitrules)
2766: (8)     na = 'e_'
2767: (8)     for c in a:
2768: (12)       c = c.lower()
2769: (12)       if c not in string.ascii_lowercase + string.digits:
2770: (16)         c = '_'
2771: (12)         na = na + c
2772: (8)         if na[-1] == '_':
2773: (12)           na = na + 'e'
2774: (8)         else:
2775: (12)           na = na + '_e'
2776: (8)         a = na
2777: (8)         while a in block['vars'] or a in block['args']:
2778: (12)           a = a + 'r'
2779: (4)       if a in args:
2780: (8)         k = 1
2781: (8)         while a + str(k) in args:
2782: (12)           k = k + 1
2783: (8)           a = a + str(k)
2784: (4)       if a_is_expr:
2785: (8)         block['vars'][a] = at
2786: (4)       else:
2787: (8)         if a not in block['vars']:
2788: (12)           if orig_a in block['vars']:
2789: (16)             block['vars'][a] = block['vars'][orig_a]
2790: (12)           else:
2791: (16)             block['vars'][a] = {}
2792: (8)           if 'externals' in block and orig_a in block['externals'] +
block['interfaced']:

```

```

2793: (12)           block['vars'][a] = setattrspec(block['vars'][a], 'external')
2794: (4)           return a
2795: (0)           def analyzeargs(block):
2796: (4)             setmessstext(block)
2797: (4)             implicitrules, _ = buildimplicitrules(block)
2798: (4)             if 'args' not in block:
2799: (8)               block['args'] = []
2800: (4)             args = []
2801: (4)             for a in block['args']:
2802: (8)               a = expr2name(a, block, args)
2803: (8)               args.append(a)
2804: (4)             block['args'] = args
2805: (4)             if 'entry' in block:
2806: (8)               for k, args1 in list(block['entry'].items()):
2807: (12)                 for a in args1:
2808: (16)                   if a not in block['vars']:
2809: (20)                     block['vars'][a] = {}
2810: (4)             for b in block['body']:
2811: (8)               if b['name'] in args:
2812: (12)                 if 'externals' not in block:
2813: (16)                   block['externals'] = []
2814: (12)                 if b['name'] not in block['externals']:
2815: (16)                   block['externals'].append(b['name'])
2816: (4)             if 'result' in block and block['result'] not in block['vars']:
2817: (8)               block['vars'][block['result']] = {}
2818: (4)             return block
2819: (0)           determineexprtype_re_1 = re.compile(r'\A\(.+?,.+?\)\Z', re.I)
2820: (0)           determineexprtype_re_2 = re.compile(r'\A[+-]\?d+(_(?P<name>\w+)|)\Z', re.I)
2821: (0)           determineexprtype_re_3 = re.compile(
2822: (4)             r'\A[+-]\?[\d.]+[-\d+de.]*(_(?P<name>\w+)|)\Z', re.I)
2823: (0)           determineexprtype_re_4 = re.compile(r'\A\(.*\)\Z', re.I)
2824: (0)           determineexprtype_re_5 = re.compile(r'\A(?P<name>\w+)\s*\(\.*?\)\s*\Z', re.I)
2825: (0)           def _ensure_exprdict(r):
2826: (4)             if isinstance(r, int):
2827: (8)               return {'typespec': 'integer'}
2828: (4)             if isinstance(r, float):
2829: (8)               return {'typespec': 'real'}
2830: (4)             if isinstance(r, complex):
2831: (8)               return {'typespec': 'complex'}
2832: (4)             if isinstance(r, dict):
2833: (8)               return r
2834: (4)             raise AssertionError(repr(r))
2835: (0)           def determineexprtype(expr, vars, rules={}):
2836: (4)             if expr in vars:
2837: (8)               return _ensure_exprdict(vars[expr])
2838: (4)             expr = expr.strip()
2839: (4)             if determineexprtype_re_1.match(expr):
2840: (8)               return {'typespec': 'complex'}
2841: (4)             m = determineexprtype_re_2.match(expr)
2842: (4)             if m:
2843: (8)               if 'name' in m.groupdict() and m.group('name'):
2844: (12)                 outmess(
2845: (16)                   'determineexprtype: selected kind types not supported (%s)\n'
% repr(expr))
2846: (8)               return {'typespec': 'integer'}
2847: (4)             m = determineexprtype_re_3.match(expr)
2848: (4)             if m:
2849: (8)               if 'name' in m.groupdict() and m.group('name'):
2850: (12)                 outmess(
2851: (16)                   'determineexprtype: selected kind types not supported (%s)\n'
% repr(expr))
2852: (8)               return {'typespec': 'real'}
2853: (4)             for op in ['+', '-', '*', '/']:
2854: (8)               for e in [x.strip() for x in markoutercomma(expr, comma=op).split('@'
+ op + '@')]:
2855: (12)                 if e in vars:
2856: (16)                   return _ensure_exprdict(vars[e])
2857: (4)             t = {}
2858: (4)             if determineexprtype_re_4.match(expr): # in parenthesis

```

```

2859: (8)          t = determineexprtype(expr[1:-1], vars, rules)
2860: (4)          else:
2861: (8)              m = determineexprtype_re_5.match(expr)
2862: (8)              if m:
2863: (12)                  rn = m.group('name')
2864: (12)                  t = determineexprtype(m.group('name'), vars, rules)
2865: (12)                  if t and 'attrspec' in t:
2866: (16)                      del t['attrspec']
2867: (12)                  if not t:
2868: (16)                      if rn[0] in rules:
2869: (20)                          return _ensure_exprdict(rules[rn[0]])
2870: (4)          if expr[0] in '\'':
2871: (8)              return {'typespec': 'character', 'charselector': {'*': '*'}}
2872: (4)          if not t:
2873: (8)              outmess(
2874: (12)                  'determineexprtype: could not determine expressions (%s) type.\n'
% (repr(expr)))
2875: (4)          return t
2876: (0)      def crack2fortrangen(block, tab='\n', as_interface=False):
2877: (4)          global skipfuncs, onlyfuncs
2878: (4)          setmesstext(block)
2879: (4)          ret = ''
2880: (4)          if isinstance(block, list):
2881: (8)              for g in block:
2882: (12)                  if g and g['block'] in ['function', 'subroutine']:
2883: (16)                      if g['name'] in skipfuncs:
2884: (20)                          continue
2885: (16)                      if onlyfuncs and g['name'] not in onlyfuncs:
2886: (20)                          continue
2887: (12)                  ret = ret + crack2fortrangen(g, tab, as_interface=as_interface)
2888: (8)          return ret
2889: (4)          prefix = ''
2890: (4)          name = ''
2891: (4)          args = ''
2892: (4)          blocktype = block['block']
2893: (4)          if blocktype == 'program':
2894: (8)              return ''
2895: (4)          argsl = []
2896: (4)          if 'name' in block:
2897: (8)              name = block['name']
2898: (4)          if 'args' in block:
2899: (8)              vars = block['vars']
2900: (8)              for a in block['args']:
2901: (12)                  a = expr2name(a, block, argsl)
2902: (12)                  if not isintent_callback(vars[a]):
2903: (16)                      argsl.append(a)
2904: (8)          if block['block'] == 'function' or argsl:
2905: (12)              args = '(%s)' % ','.join(argsl)
2906: (4)          f2pyenhancements = ''
2907: (4)          if 'f2pyenhancements' in block:
2908: (8)              for k in list(block['f2pyenhancements'].keys()):
2909: (12)                  f2pyenhancements = '%s%s%s %s' % (
2910: (16)                      f2pyenhancements, tab + tabchar, k, block['f2pyenhancements'][k])
2911: (4)          intent_lst = block.get('intent', [])[:]
2912: (4)          if blocktype == 'function' and 'callback' in intent_lst:
2913: (8)              intent_lst.remove('callback')
2914: (4)          if intent_lst:
2915: (8)              f2pyenhancements = '%s%sintent(%s) %s' % \
2916: (27)                  (f2pyenhancements, tab + tabchar,
2917: (28)                      ', '.join(intent_lst), name)
2918: (4)          use = ''
2919: (4)          if 'use' in block:
2920: (8)              use = use2fortran(block['use'], tab + tabchar)
2921: (4)          common = ''
2922: (4)          if 'common' in block:
2923: (8)              common = common2fortran(block['common'], tab + tabchar)
2924: (4)          if name == 'unknown_interface':
2925: (8)              name = ''

```



```

2993: (4)             nout = []
2994: (4)             for a in args:
2995: (8)               if a in block['vars']:
2996: (12)                 nout.append(a)
2997: (4)             if 'commonvars' in block:
2998: (8)               for a in block['commonvars']:
2999: (12)                 if a in vars:
3000: (16)                   if a not in nout:
3001: (20)                     nout.append(a)
3002: (12)                 else:
3003: (16)                   errmess(
3004: (20)                     'vars2fortran: Confused?!: "%s" is not defined in vars.\n'
% a)
3005: (4)             if 'varnames' in block:
3006: (8)               nout.extend(block['varnames'])
3007: (4)             if not as_interface:
3008: (8)               for a in list(vars.keys()):
3009: (12)                 if a not in nout:
3010: (16)                   nout.append(a)
3011: (4)             for a in nout:
3012: (8)               if 'depend' in vars[a]:
3013: (12)                 for d in vars[a]['depend']:
3014: (16)                   if d in vars and 'depend' in vars[d] and a in vars[d]
['depend']:
3015: (20)                     errmess(
3016: (24)                       'vars2fortran: Warning: cross-dependence between
variables "%s" and "%s"\n' % (a, d))
3017: (8)             if 'externals' in block and a in block['externals']:
3018: (12)               if isintent_callback(vars[a]):
3019: (16)                 ret = '%s%sintent(callback) %s' % (ret, tab, a)
3020: (12)               ret = '%s%sexternal %s' % (ret, tab, a)
3021: (12)               if isoptional(vars[a]):
3022: (16)                 ret = '%s%soptional %s' % (ret, tab, a)
3023: (12)               if a in vars and 'typespec' not in vars[a]:
3024: (16)                 continue
3025: (12)               cont = 1
3026: (12)               for b in block['body']:
3027: (16)                 if a == b['name'] and b['block'] == 'function':
3028: (20)                   cont = 0
3029: (20)                   break
3030: (12)                 if cont:
3031: (16)                   continue
3032: (8)               if a not in vars:
3033: (12)                 show(vars)
3034: (12)                 outmess('vars2fortran: No definition for argument "%s".\n' % a)
3035: (12)
3036: (8)               if a == block['name']:
3037: (12)                 if block['block'] != 'function' or block.get('result'):
3038: (16)                   continue
3039: (8)               if 'typespec' not in vars[a]:
3040: (12)                 if 'attrspec' in vars[a] and 'external' in vars[a]['attrspec']:
3041: (16)                   if a in args:
3042: (20)                     ret = '%s%sexternal %s' % (ret, tab, a)
3043: (16)                   continue
3044: (12)                 show(vars[a])
3045: (12)                 outmess('vars2fortran: No typespec for argument "%s".\n' % a)
3046: (12)                 continue
3047: (8)               vardef = vars[a]['typespec']
3048: (8)               if vardef == 'type' and 'typename' in vars[a]:
3049: (12)                 vardef = '%s(%s)' % (vardef, vars[a]['typename'])
3050: (8)               selector = {}
3051: (8)               if 'kindselector' in vars[a]:
3052: (12)                 selector = vars[a]['kindselector']
3053: (8)               elif 'charselector' in vars[a]:
3054: (12)                 selector = vars[a]['charselector']
3055: (8)               if '*' in selector:
3056: (12)                 if selector['*'] in ['*', ':']:
3057: (16)                   vardef = '%s*(%s)' % (vardef, selector['*'])
3058: (12)                 else:

```

```

3059: (16)             vardef = '%s*%s' % (vardef, selector['*'])
3060: (8)              else:
3061: (12)                 if 'len' in selector:
3062: (16)                     vardef = '%s(len=%s' % (vardef, selector['len'])
3063: (16)                     if 'kind' in selector:
3064: (20)                         vardef = '%s,kind=%s)' % (vardef, selector['kind'])
3065: (16)                     else:
3066: (20)                         vardef = '%s)' % (vardef)
3067: (12)                 elif 'kind' in selector:
3068: (16)                     vardef = '%s(kind=%s)' % (vardef, selector['kind'])
3069: (8)                 c = ' '
3070: (8)                 if 'attrspec' in vars[a]:
3071: (12)                     attr = [l for l in vars[a]['attrspec']
3072: (20)                         if l not in ['external']]
3073: (12)                     if as_interface and 'intent(in)' in attr and 'intent(out)' in
3074: (16)                         attr.remove('intent(out)')
3075: (12)                     if attr:
3076: (16)                         vardef = '%s, %s' % (vardef, ','.join(attr))
3077: (16)                         c = ', '
3078: (8)                     if 'dimension' in vars[a]:
3079: (12)                         vardef = '%s%sdimension(%s)' % (
3080: (16)                             vardef, c, ','.join(vars[a]['dimension']))
3081: (12)                         c = ', '
3082: (8)                     if 'intent' in vars[a]:
3083: (12)                         lst = true_intent_list(vars[a])
3084: (12)                         if lst:
3085: (16)                             vardef = '%s%sintent(%s)' % (vardef, c, ','.join(lst))
3086: (12)                             c = ', '
3087: (8)                     if 'check' in vars[a]:
3088: (12)                         vardef = '%s%scheck(%s)' % (vardef, c, ','.join(vars[a]['check']))
3089: (12)                         c = ', '
3090: (8)                     if 'depend' in vars[a]:
3091: (12)                         vardef = '%s%sdepend(%s)' % (
3092: (16)                             vardef, c, ','.join(vars[a]['depend']))
3093: (12)                             c = ', '
3094: (8)                     if '=' in vars[a]:
3095: (12)                         v = vars[a]['=']
3096: (12)                         if vars[a]['typespec'] in ['complex', 'double complex']:
3097: (16)                             try:
3098: (20)                                 v = eval(v)
3099: (20)                                 v = '(%s,%s)' % (v.real, v.imag)
3100: (16)                             except Exception:
3101: (20)                                 pass
3102: (12)                             vardef = '%s :: %s=%s' % (vardef, a, v)
3103: (8)                     else:
3104: (12)                         vardef = '%s :: %s' % (vardef, a)
3105: (8)                         ret = '%s%s%s' % (ret, tab, vardef)
3106: (4)                     return ret
3107: (0)                 post_processing_hooks = []
3108: (0)             def crackfortran(files):
3109: (4)                 global usermodules, post_processing_hooks
3110: (4)                 outmess('Reading fortran codes...\n', 0)
3111: (4)                 readfortrancode(files, crackline)
3112: (4)                 outmess('Post-processing...\n', 0)
3113: (4)                 usermodules = []
3114: (4)                 postlist = postcrack(grouplist[0])
3115: (4)                 outmess('Applying post-processing hooks...\n', 0)
3116: (4)                 for hook in post_processing_hooks:
3117: (8)                     outmess(f' {hook.__name__}\n', 0)
3118: (8)                     postlist = traverse(postlist, hook)
3119: (4)                     outmess('Post-processing (stage 2)... \n', 0)
3120: (4)                     postlist = postcrack2(postlist)
3121: (4)                     return usermodules + postlist
3122: (0)             def crack2fortran(block):
3123: (4)                 global f2py_version
3124: (4)                 pyf = crack2fortrangen(block) + '\n'
3125: (4)                 header = """!      -*- f90 -*-"""
3126: (0)             ! Note: the context of this file is case sensitive.

```

```

3127: (0)
3128: (4)         footer = """
3129: (0)         ! This file was auto-generated with f2py (version:%s).
3130: (0)         ! See:
3131: (0)         !
3132: (0)         https://web.archive.org/web/20140822061353/http://cens.ioc.ee/projects/f2py2e
3133: (4)         """
3134: (0)         % (f2py_version)
3135: (4)         return header + pyf + footer
3136: (12)        def _is_visit_pair(obj):
3137: (12)         return (isinstance(obj, tuple)
3138: (0)             and len(obj) == 2
3139: (4)             and isinstance(obj[0], (int, str)))
3140: (4)         def traverse(obj, visit, parents=[], result=None, *args, **kwargs):
3141: (8)             """
3142: (8)                 parents is a list of key-"f2py data structure" pairs from which
3143: (8)                 items are taken from.
3144: (8)                 result is a f2py data structure that is filled with the
3145: (8)                 return value of the visit function.
3146: (8)                 item is 2-tuple (index, value) if parents[-1][1] is a list
3147: (8)                 item is 2-tuple (key, value) if parents[-1][1] is a dict
3148: (8)                 The return value of visit must be None, or of the same kind as
3149: (8)                 item, that is, if parents[-1] is a list, the return value must
3150: (8)                 be 2-tuple (new_index, new_value), or if parents[-1] is a
3151: (8)                 dict, the return value must be 2-tuple (new_key, new_value).
3152: (8)                 If new_index or new_value is None, the return value of visit
3153: (8)                 is ignored, that is, it will not be added to the result.
3154: (8)                 If the return value is None, the content of obj will be
3155: (8)                 traversed, otherwise not.
3156: (8)             """
3157: (4)         """
3158: (4)         if _is_visit_pair(obj):
3159: (8)             if obj[0] == 'parent_block':
3160: (12)                 return obj
3161: (8)             new_result = visit(obj, parents, result, *args, **kwargs)
3162: (8)             if new_result is not None:
3163: (12)                 assert _is_visit_pair(new_result)
3164: (12)                 return new_result
3165: (8)             parent = obj
3166: (8)             result_key, obj = obj
3167: (4)         else:
3168: (8)             parent = (None, obj)
3169: (8)             result_key = None
3170: (4)         if isinstance(obj, list):
3171: (8)             new_result = []
3172: (8)             for index, value in enumerate(obj):
3173: (12)                 new_index, new_item = traverse((index, value), visit,
3174: (43)                               parents=parents + [parent],
3175: (43)                               result=result, *args, **kwargs)
3176: (12)                 if new_index is not None:
3177: (16)                     new_result.append(new_item)
3178: (4)             elif isinstance(obj, dict):
3179: (8)                 new_result = dict()
3180: (8)                 for key, value in obj.items():
3181: (12)                     new_key, new_value = traverse((key, value), visit,
3182: (42)                               parents=parents + [parent],
3183: (42)                               result=result, *args, **kwargs)
3184: (12)                     if new_key is not None:
3185: (16)                         new_result[new_key] = new_value
3186: (4)                 else:
3187: (8)                     new_result = obj
3188: (4)                 if result_key is None:
3189: (8)                     return new_result
3190: (4)                 return result_key, new_result
3191: (0)             def character_backward_compatibility_hook(item, parents, result,
3192: (42)                               *args, **kwargs):
3193: (4)                 """
3194: (4)                     Previously, Fortran character was incorrectly treated as

```

character\*1. This hook fixes the usage of the corresponding

```

3195: (4) variables in `check`, `dimension`, `=`, and `callstatement`  

3196: (4) expressions.  

3197: (4) The usage of `char*` in `callprotoargument` expression can be left  

3198: (4) unchanged because C `character` is C typedef of `char`, although,  

3199: (4) new implementations should use `character*` in the corresponding  

3200: (4) expressions.  

3201: (4) See https://github.com/numpy/numpy/pull/19388 for more information.  

3202: (4)  

3203: (4) """  

3204: (4) parent_key, parent_value = parents[-1]  

3205: (4) key, value = item  

3206: (8) def fix_usage(varname, value):  

3207: (8)     value = re.sub(r'[^]*\s*\b' + varname + r'\b', varname, value)  

3208: (23)    value = re.sub(r'\b' + varname + r'\b\s*[\[]|\s*0\s*[\]]',  

3209: (8)                varname, value)  

3210: (4)    return value  

3211: (8) if parent_key in ['dimension', 'check']:  

3212: (8)     assert parents[-3][0] == 'vars'  

3213: (8)     vars_dict = parents[-3][1]  

3214: (4) elif key == '=':  

3215: (8)     assert parents[-2][0] == 'vars'  

3216: (4)     vars_dict = parents[-2][1]  

3217: (4) else:  

3218: (4)     vars_dict = None  

3219: (4) new_value = None  

3220: (8) if vars_dict is not None:  

3221: (8)     new_value = value  

3222: (12)    for varname, vd in vars_dict.items():  

3223: (16)        if ischaracter(vd):  

3224: (4)            new_value = fix_usage(varname, new_value)  

3225: (8) elif key == 'callstatement':  

3226: (8)     vars_dict = parents[-2][1]['vars']  

3227: (8)     new_value = value  

3228: (12)    for varname, vd in vars_dict.items():  

3229: (16)        if ischaracter(vd):  

3230: (20)            new_value = re.sub(  

3231: (4)                r'(?<![&])\b' + varname + r'\b', '&' + varname, new_value)  

3232: (8) if new_value is not None:  

3233: (12)    if new_value != value:  

3234: (20)        outmess(f'character_bc_hook[{parent_key}.{key}]:'  

3235: (8)            f' replaced `{{value}` -> `{{new_value}}`\n', 1)  

3236: (0)    return (key, new_value)  

3237: (0) post_processing_hooks.append(character_backward_compatibility_hook)  

3238: (4) if __name__ == "__main__":  

3239: (4)     files = []  

3240: (4)     funcs = []  

3241: (4)     f = 1  

3242: (4)     f2 = 0  

3243: (4)     f3 = 0  

3244: (4)     showblocklist = 0  

3245: (8)     for l in sys.argv[1:]:  

3246: (12)         if l == '':
3247: (8)             pass
3248: (12)         elif l[0] == ':':
3249: (8)             f = 0
3250: (12)         elif l == '-quiet':
3251: (12)             quiet = 1
3252: (8)             verbose = 0
3253: (12)         elif l == '-verbose':
3254: (12)             verbose = 2
3255: (8)             quiet = 0
3256: (12)         elif l == '-fix':
3257: (16)             if strictf77:
3258: (20)                 outmess(
3259: (4)                     'Use option -f90 before -fix if Fortran 90 code is in fix
form.\n', 0)
3260: (12)                 skipemptyends = 1
3261: (12)                 sourcecodeform = 'fix'
3262: (8)             elif l == '-skipemptyends':
3263: (12)                 skipemptyends = 1

```

```

3263: (8)           elif l == '--ignore-contains':
3264: (12)             ignorecontains = 1
3265: (8)           elif l == '-f77':
3266: (12)             strictf77 = 1
3267: (12)             sourcecodeform = 'fix'
3268: (8)           elif l == '-f90':
3269: (12)             strictf77 = 0
3270: (12)             sourcecodeform = 'free'
3271: (12)             skipemptyends = 1
3272: (8)           elif l == '-h':
3273: (12)             f2 = 1
3274: (8)           elif l == '-show':
3275: (12)             showblocklist = 1
3276: (8)           elif l == '-m':
3277: (12)             f3 = 1
3278: (8)           elif l[0] == '-':
3279: (12)             errmess('Unknown option %s\n' % repr(l))
3280: (8)           elif f2:
3281: (12)             f2 = 0
3282: (12)             pyffilename = 1
3283: (8)           elif f3:
3284: (12)             f3 = 0
3285: (12)             f77modulename = 1
3286: (8)           elif f:
3287: (12)             try:
3288: (16)               open(l).close()
3289: (16)               files.append(l)
3290: (12)             except OSError as detail:
3291: (16)               errmess(f'OSError: {detail!s}\n')
3292: (8)           else:
3293: (12)             funcs.append(l)
3294: (4)           if not strictf77 and f77modulename and not skipemptyends:
3295: (8)             outmess("""\
3296: (2)             Warning: You have specified module name for non Fortran 77 code that
3297: (2)             should not need one (expect if you are scanning F90 code for non
3298: (2)             module blocks but then you should use flag -skipemptyends and also
3299: (2)             be sure that the files do not contain programs without program
3300: (2)             statement).
3301: (0)           """", 0)
3302: (4)           postlist = crackfortran(files)
3303: (4)           if pyffilename:
3304: (8)             outmess('Writing fortran code to file %s\n' % repr(pyffilename), 0)
3305: (8)             pyf = crack2fortran(postlist)
3306: (8)             with open(pyffilename, 'w') as f:
3307: (12)               f.write(pyf)
3308: (4)           if showblocklist:
3309: (8)             show(postlist)

```

-----

## File 148 - diagnose.py:

```

1: (0)           import os
2: (0)           import sys
3: (0)           import tempfile
4: (0)           def run_command(cmd):
5: (4)             print('Running %r:' % (cmd))
6: (4)             os.system(cmd)
7: (4)             print('-----')
8: (0)           def run():
9: (4)             _path = os.getcwd()
10: (4)             os.chdir(tempfile.gettempdir())
11: (4)             print('-----')
12: (4)             print('os.name=%r' % (os.name))
13: (4)             print('-----')
14: (4)             print('sys.platform=%r' % (sys.platform))
15: (4)             print('-----')
16: (4)             print('sys.version:')
17: (4)             print(sys.version)

```

```

18: (4)          print('-----')
19: (4)          print('sys.prefix:')
20: (4)          print(sys.prefix)
21: (4)          print('-----')
22: (4)          print('sys.path=%r' % (':'.join(sys.path)))
23: (4)          print('-----')
24: (4)          try:
25: (8)              import numpy
26: (8)              has_newnumpy = 1
27: (4)          except ImportError as e:
28: (8)              print('Failed to import new numpy:', e)
29: (8)              has_newnumpy = 0
30: (4)          try:
31: (8)              from numpy.f2py import f2py2e
32: (8)              has_f2py2e = 1
33: (4)          except ImportError as e:
34: (8)              print('Failed to import f2py2e:', e)
35: (8)              has_f2py2e = 0
36: (4)          try:
37: (8)              import numpy.distutils
38: (8)              has_numpy_distutils = 2
39: (4)          except ImportError:
40: (8)              try:
41: (12)                  import numpy_distutils
42: (12)                  has_numpy_distutils = 1
43: (8)              except ImportError as e:
44: (12)                  print('Failed to import numpy_distutils:', e)
45: (12)                  has_numpy_distutils = 0
46: (4)          if has_newnumpy:
47: (8)              try:
48: (12)                  print('Found new numpy version %r in %s' %
49: (18)                      (numpy.__version__, numpy.__file__))
50: (8)              except Exception as msg:
51: (12)                  print('error:', msg)
52: (12)                  print('-----')
53: (4)          if has_f2py2e:
54: (8)              try:
55: (12)                  print('Found f2py2e version %r in %s' %
56: (18)                      (f2py2e.__version__.version, f2py2e.__file__))
57: (8)              except Exception as msg:
58: (12)                  print('error:', msg)
59: (12)                  print('-----')
60: (4)          if has_numpy_distutils:
61: (8)              try:
62: (12)                  if has_numpy_distutils == 2:
63: (16)                      print('Found numpy.distutils version %r in %r' % (
64: (20)                          numpy.distutils.__version__,
65: (20)                          numpy.distutils.__file__))
66: (12)                  else:
67: (16)                      print('Found numpy_distutils version %r in %r' % (
68: (20)
numpy_distutils.numpy_distutils_version.numpy_distutils_version,
69: (20)                          numpy_distutils.__file__))
70: (12)                  print('-----')
71: (8)              except Exception as msg:
72: (12)                  print('error:', msg)
73: (12)                  print('-----')
74: (8)              try:
75: (12)                  if has_numpy_distutils == 1:
76: (16)                      print(
77: (20)                          'Importing numpy_distutils.command.build_flib ...', end='
')
78: (16)                      import numpy_distutils.command.build_flib as build_flib
79: (16)                      print('ok')
80: (16)                      print('-----')
81: (16)                      try:
82: (20)                          print(
83: (24)                              'Checking availability of supported Fortran
compilers:')
```

```

84: (20)                     for compiler_class in build_flib.all_compilers:
85: (24)                         compiler_class(verbose=1).is_available()
86: (24)                             print('-----')
87: (16)             except Exception as msg:
88: (20)                 print('error:', msg)
89: (20)                 print('-----')
90: (8)         except Exception as msg:
91: (12)             print(
92: (16)                 'error:', msg, '(ignore it, build_flib is obsolete for
numpy.distutils 0.2.2 and up)')
93: (12)             print('-----')
94: (8)         try:
95: (12)             if has_numpy_distutils == 2:
96: (16)                 print('Importing numpy.distutils.fcompiler ...', end=' ')
97: (16)                 import numpy.distutils.fcompiler as fcompiler
98: (12)             else:
99: (16)                 print('Importing numpy_distutils.fcompiler ...', end=' ')
100: (16)                import numpy_distutils.fcompiler as fcompiler
101: (12)            print('ok')
102: (12)            print('-----')
103: (12)        try:
104: (16)            print('Checking availability of supported Fortran compilers:')
105: (16)            fcompiler.show_fcompilers()
106: (16)            print('-----')
107: (12)        except Exception as msg:
108: (16)            print('error:', msg)
109: (16)            print('-----')
110: (8)    except Exception as msg:
111: (12)        print('error:', msg)
112: (12)        print('-----')
113: (8)    try:
114: (12)        if has_numpy_distutils == 2:
115: (16)            print('Importing numpy.distutils.cpuinfo ...', end=' ')
116: (16)            from numpy.distutils.cpuinfo import cpuinfo
117: (16)            print('ok')
118: (16)            print('-----')
119: (12)        else:
120: (16)            try:
121: (20)                print(
122: (24)                  'Importing numpy_distutils.command.cpuinfo ...', end=' '
)
123: (20)                  from numpy_distutils.command.cpuinfo import cpuinfo
124: (20)                  print('ok')
125: (20)                  print('-----')
126: (16)            except Exception as msg:
127: (20)                print('error:', msg, '(ignore it)')
128: (20)                print('Importing numpy_distutils.cpuinfo ...', end=' ')
129: (20)                from numpy_distutils.cpuinfo import cpuinfo
130: (20)                print('ok')
131: (20)                print('-----')
132: (12)            cpu = cpuinfo()
133: (12)            print('CPU information:', end=' ')
134: (12)            for name in dir(cpuinfo):
135: (16)                if name[0] == '_' and name[1] != '_' and getattr(cpu,
name[1:]):
136: (20)                    print(name[1:], end=' ')
137: (12)                    print('-----')
138: (8)            except Exception as msg:
139: (12)                print('error:', msg)
140: (12)                print('-----')
141: (4)                os.chdir(_path)
142: (0)            if __name__ == "__main__":
143: (4)                run()

-----

```

File 149 - f2py2e.py:

```

1: (0)      """

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2: (0) f2py2e - Fortran to Python C/API generator. 2nd Edition.
3: (9)     See __usage__ below.
4: (0) Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
5: (0) Copyright 2011 -- present NumPy Developers.
6: (0) Permission to use, modify, and distribute this software is given under the
7: (0) terms of the NumPy License.
8: (0) NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
9: (0)
10: (0) """
11: (0) import sys
12: (0) import os
13: (0) import pprint
14: (0) import re
15: (0) from pathlib import Path
16: (0) from itertools import dropwhile
17: (0) import argparse
18: (0) import copy
19: (0) from . import crackfortran
20: (0) from . import rules
21: (0) from . import cb_rules
22: (0) from . import auxfuncs
23: (0) from . import cfuncs
24: (0) from . import f90mod_rules
25: (0) from . import __version__
26: (0) from . import capi_maps
27: (0) from numpy.f2py._backends import f2py_build_generator
28: (0) f2py_version = __version__.version
29: (0) numpy_version = __version__.version
29: (0) errmess = sys.stderr.write
30: (0) show = pprint.pprint
31: (0) outmess = auxfuncs.outmess
32: (0) MESON_ONLY_VER = (sys.version_info >= (3, 12))
33: (0) __usage__ =\
34: (0) f"""Usage:
35: (0) 1) To construct extension module sources:
36: (6)     f2py [<options>] <fortran files> [[[only:]]|[skip:]] \\
37: (40)                         <fortran functions> ] \\
38: (39)                         [: <fortran files> ...]
39: (0) 2) To compile fortran files and build extension modules:
40: (6)     f2py -c [<options>, <build_flib options>, <extra options>] <fortran
41: (0) files>
42: (6)
43: (0) 3) To generate signature files:
44: (13)     f2py -h <filename.pyf> ...< same options as in (1) >
45: (13) Description: This program generates a Python C/API file (<modulename>module.c)
46: (13) that contains wrappers for given fortran functions so that they
47: (0) can be called from Python. With the -c option the corresponding
47: (0) extension modules are built.
48: (2) Options:
49: (19)     -h <filename> Write signatures of the fortran routines to file <filename>
50: (19)             and exit. You can then edit <filename> and use it instead
51: (19)             of <fortran files>. If <filename>==stdout then the
52: (2)             signatures are printed to stdout.
53: (19)             <fortran functions> Names of fortran routines for which Python C/API
54: (19)             functions will be generated. Default is all that are found
55: (2)             in <fortran files>.
56: (19)             <fortran files> Paths to fortran/signature files that will be scanned for
57: (2)             <fortran functions> in order to determine their signatures.
58: (2)             skip: Ignore fortran functions that follow until `:'.
59: (2)             only: Use only fortran functions that follow until `:'.
60: (2)             : Get back to <fortran files> mode.
61: (19)             -m <modulename> Name of the module; f2py generates a Python/C API
62: (19)             file <modulename>module.c or extension module <modulename>.
63: (2)             Default is 'untitled'.
64: (22)             '-include<header>' Writes additional headers in the C wrapper, can be
65: (2)                 multiple times, generates #include <header> each time.
66: (19)             --[no-]lower Do [not] lower the cases in <fortran files>. By default,
key.               --lower is assumed with -h key, and --no-lower without -h
67: (2)             --build-dir <dirname> All f2py generated files are created in <dirname>.

```

```

68: (19)                               Default is tempfile.mkdtemp().
69: (2) --overwrite-signature Overwrite existing signature file.
70: (2) --[no-]latex-doc Create (or not) <modulename>module.tex.
71: (19)                               Default is --no-latex-doc.
72: (2) --short-latex Create 'incomplete' LaTeX document (without commands
73: (19)                               \\documentclass, \\tableofcontents, and
\\begin{{document}},                                \\end{{document}}).
74: (19) --[no-]rest-doc Create (or not) <modulename>module.rst.
75: (2)                               Default is --no-rest-doc.
76: (19) --debug-capi Create C/API code that reports the state of the wrappers
77: (2)                               during runtime. Useful for debugging.
78: (19) --[no-]wrap-functions Create Fortran subroutine wrappers to Fortran 77
79: (2)                               functions. --wrap-functions is default because it ensures
80: (19)                               maximum portability/compiler independence.
81: (19) --include-paths <path1>:<path2>:... Search include files from the given
82: (2)                               directories.
83: (19) --help-link [...] List system resources found by system_info.py. See also
84: (2)                               --link-<resource> switch below. [...] is optional list
85: (19)                               of resources names. E.g. try 'f2py --help-link lapack_opt'.
86: (19) --f2cmap <filename> Load Fortran-to-Python KIND specification from the
87: (2)                               given
88: (19)                               file. Default: .f2py_f2cmap in current directory.
89: (2) --quiet Run quietly.
90: (2) --verbose Run with extra verbosity.
91: (2) --skip-empty-wrappers Only generate wrapper files when needed.
92: (2) -v Print f2py version ID and exit.
93: (0) build backend options (only effective with -c)
94: (0) [NO_MESON] is used to indicate an option not meant to be used
95: (0) with the meson backend or above Python 3.12:
96: (2) --fcompiler= Specify Fortran compiler type by vendor [NO_MESON]
97: (2) --compiler= Specify distutils C compiler type [NO_MESON]
98: (2) --help-fcompiler List available Fortran compilers and exit [NO_MESON]
99: (2) --f77exec= Specify the path to F77 compiler [NO_MESON]
100: (2) --f90exec= Specify the path to F90 compiler [NO_MESON]
101: (2) --f77flags= Specify F77 compiler flags
102: (2) --f90flags= Specify F90 compiler flags
103: (2) --opt= Specify optimization flags [NO_MESON]
104: (2) --arch= Specify architecture specific optimization flags
[NO_MESON]
105: (2) --noopt Compile without optimization [NO_MESON]
106: (2) --noarch Compile without arch-dependent optimization [NO_MESON]
107: (2) --debug Compile with debugging information
108: (2) --dep <dependency>
109: (23) Specify a meson dependency for the module. This may
110: (23) be passed multiple times for multiple dependencies.
111: (23) Dependencies are stored in a list for further
processing.
112: (23) Example: --dep lapack --dep scalapack
113: (23) This will identify "lapack" and "scalapack" as
dependencies
114: (23) and remove them from argv, leaving a dependencies list
115: (23) containing ["lapack", "scalapack"].
116: (2) --backend <backend_type>
117: (23) Specify the build backend for the compilation process.
118: (23) The supported backends are 'meson' and 'distutils'.
119: (23) If not specified, defaults to 'distutils'. On
120: (23) Python 3.12 or higher, the default is 'meson'.
121: (0) Extra options (only effective with -c):
122: (2) --link-<resource> Link extension module with <resource> as defined
123: (23) by numpy.distutils/system_info.py. E.g. to link
124: (23) with optimized LAPACK libraries (veclib on MacOSX,
125: (23) ATLAS elsewhere), use --link-lapack_opt.
126: (23) See also --help-link switch. [NO_MESON]
127: (2) -L/path/to/lib/ -l<libname>
128: (2) -D<define> -U<name>
129: (2) -I/path/to/include/
130: (2) <filename>.o <filename>.so <filename>.a
131: (2) Using the following macros may be required with non-gcc Fortran

```

```

132: (2)
133: (4)
134: (4)
135: (2)
136: (2)
137: (2)
138: (2)
139: (2)
140: (2)
141: (0)
142: (0)
143: (0)
144: (0)
145: (0)
146: (0)
147: (0)
148: (4)
149: (4)
150: (4)
151: (4)
152: (4)
153: (4)
154: (4)
155: (4)
156: (4)
157: (4)
158: (4)
159: (4)
160: (15)
161: (15)
162: (4)
163: (8)
164: (12)
165: (8)
166: (12)
167: (8)
168: (12)
169: (8)
170: (12)
171: (8)
172: (12)
173: (8)
174: (12)
175: (8)
176: (12)
177: (8)
178: (12)
179: (8)
180: (12)
181: (8)
182: (12)
183: (8)
184: (12)
185: (8)
186: (12)
187: (8)
188: (12)
189: (8)
190: (12)
191: (8)
192: (12)
193: (8)
194: (12)
195: (8)
196: (12)
197: (8)
198: (12)
199: (8)
200: (12)

compilers:
    -DPREPEND_FORTRAN -DNO_APPEND_FORTRAN -DUPPERCASE_FORTRAN
    -DUNDERSCORE_G77
When using -DF2PY_REPORT_ATEXIT, a performance report of F2PY
interface is printed out at exit (platforms: Linux).
When using -DF2PY_REPORT_ON_ARRAY_COPY=<int>, a message is
sent to stderr whenever F2PY interface makes a copy of an
array. Integer <int> sets the threshold for array sizes when
a message should be shown.
Version: {f2py_version}
numpy Version: {numpy_version}
License: NumPy license (see LICENSE.txt in the NumPy source code)
Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
Copyright 2011 -- present NumPy Developers.
https://numpy.org/doc/stable/f2py/index.html\n"""
def scaninputline(inputline):
    files, skipfuncs, onlyfuncs, debug = [], [], [], []
    f, f2, f3, f5, f6, f8, f9, f10 = 1, 0, 0, 0, 0, 0, 0, 0
    verbose = 1
    emptygen = True
    dolc = -1
    dolatexdoc = 0
    dorestdoc = 0
    wrapfuncs = 1
    buildpath = '.'
    include_paths, inputline = get_includes(inputline)
    signsfile, modulename = None, None
    options = {'buildpath': buildpath,
               'coutput': None,
               'f2py_wrapper_output': None}
    for l in inputline:
        if l == '':
            pass
        elif l == 'only':
            f = 0
        elif l == 'skip':
            f = -1
        elif l == ':':
            f = 1
        elif l[:8] == '--debug-':
            debug.append(l[8:])
        elif l == '--lower':
            dolc = 1
        elif l == '--build-dir':
            f6 = 1
        elif l == '--no-lower':
            dolc = 0
        elif l == '--quiet':
            verbose = 0
        elif l == '--verbose':
            verbose += 1
        elif l == '--latex-doc':
            dolatexdoc = 1
        elif l == '--no-latex-doc':
            dolatexdoc = 0
        elif l == '--rest-doc':
            dorestdoc = 1
        elif l == '--no-rest-doc':
            dorestdoc = 0
        elif l == '--wrap-functions':
            wrapfuncs = 1
        elif l == '--no-wrap-functions':
            wrapfuncs = 0
        elif l == '--short-latex':
            options['shortlatex'] = 1
        elif l == '--coutput':
            f8 = 1
        elif l == '--f2py-wrapper-output':
            f9 = 1

```

```

201: (8)           elif l == '--f2cmap':
202: (12)          f10 = 1
203: (8)          elif l == '--overwrite-signature':
204: (12)          options['h-overwrite'] = 1
205: (8)          elif l == '-h':
206: (12)          f2 = 1
207: (8)          elif l == '-m':
208: (12)          f3 = 1
209: (8)          elif l[:2] == '-v':
210: (12)          print(f2py_version)
211: (12)          sys.exit()
212: (8)          elif l == '--show-compilers':
213: (12)          f5 = 1
214: (8)          elif l[:8] == '-include':
215: (12)          cfuncs.outneeds['userincludes'].append(l[9:-1])
216: (12)          cfuncs.userincludes[l[9:-1]] = '#include ' + l[8:]
217: (8)          elif l == '--skip-empty-wrappers':
218: (12)          emptygen = False
219: (8)          elif l[0] == '-':
220: (12)          errmess('Unknown option %s\n' % repr(l))
221: (12)          sys.exit()
222: (8)          elif f2:
223: (12)          f2 = 0
224: (12)          signsfile = 1
225: (8)          elif f3:
226: (12)          f3 = 0
227: (12)          modulename = 1
228: (8)          elif f6:
229: (12)          f6 = 0
230: (12)          buildpath = 1
231: (8)          elif f8:
232: (12)          f8 = 0
233: (12)          options["coutput"] = 1
234: (8)          elif f9:
235: (12)          f9 = 0
236: (12)          options["f2py_wrapper_output"] = 1
237: (8)          elif f10:
238: (12)          f10 = 0
239: (12)          options["f2cmap_file"] = 1
240: (8)          elif f == 1:
241: (12)          try:
242: (16)            with open(l):
243: (20)              pass
244: (16)              files.append(l)
245: (12)          except OSError as detail:
246: (16)            errmess(f'OSError: {detail!s}. Skipping file "{l!s}">\n')
247: (8)          elif f == -1:
248: (12)            skipfuncs.append(l)
249: (8)          elif f == 0:
250: (12)            onlyfuncs.append(l)
251: (4)          if not f5 and not files and not modulename:
252: (8)            print(__usage__)
253: (8)            sys.exit()
254: (4)          if not os.path.isdir(buildpath):
255: (8)            if not verbose:
256: (12)              outmess('Creating build directory %s\n' % (buildpath))
257: (8)              os.mkdir(buildpath)
258: (4)          if signsfile:
259: (8)            signsfile = os.path.join(buildpath, signsfile)
260: (4)          if signsfile and os.path.isfile(signsfile) and 'h-overwrite' not in
options:
261: (8)            errmess(
262: (12)              'Signature file "%s" exists!!! Use --overwrite-signature to
overwrite.\n' % (signsfile))
263: (8)            sys.exit()
264: (4)            options['emptygen'] = emptygen
265: (4)            options['debug'] = debug
266: (4)            options['verbose'] = verbose
267: (4)            if dolc == -1 and not signsfile:

```

```

268: (8)             options['do-lower'] = 0
269: (4)             else:
270: (8)                 options['do-lower'] = dolc
271: (4)             if modulename:
272: (8)                 options['module'] = modulename
273: (4)             if signsfile:
274: (8)                 options['signsfile'] = signsfile
275: (4)             if onlyfuncs:
276: (8)                 options['onlyfuncs'] = onlyfuncs
277: (4)             if skipfuncs:
278: (8)                 options['skipfuncs'] = skipfuncs
279: (4)             options['dolatexdoc'] = dolatexdoc
280: (4)             options['dorestdoc'] = dorestdoc
281: (4)             options['wrapfuncs'] = wrapfuncs
282: (4)             options['buildpath'] = buildpath
283: (4)             options['include_paths'] = include_paths
284: (4)             options.setdefault('f2cmap_file', None)
285: (4)             return files, options
286: (0)         def callcrackfortran(files, options):
287: (4)             rules.options = options
288: (4)             crackfortran.debug = options['debug']
289: (4)             crackfortran.verbose = options['verbose']
290: (4)             if 'module' in options:
291: (8)                 crackfortran.f77modulename = options['module']
292: (4)             if 'skipfuncs' in options:
293: (8)                 crackfortran.skipfuncs = options['skipfuncs']
294: (4)             if 'onlyfuncs' in options:
295: (8)                 crackfortran.onlyfuncs = options['onlyfuncs']
296: (4)             crackfortran.include_paths[:] = options['include_paths']
297: (4)             crackfortran.dolowercase = options['do-lower']
298: (4)             postlist = crackfortran.crackfortran(files)
299: (4)             if 'signsfile' in options:
300: (8)                 outmess('Saving signatures to file "%s"\n' % (options['signsfile']))
301: (8)                 pyf = crackfortran.crack2fortran(postlist)
302: (8)                 if options['signsfile'][~-6:] == 'stdout':
303: (12)                     sys.stdout.write(pyf)
304: (8)                 else:
305: (12)                     with open(options['signsfile'], 'w') as f:
306: (16)                         f.write(pyf)
307: (4)             if options["coutput"] is None:
308: (8)                 for mod in postlist:
309: (12)                     mod["coutput"] = "%smodule.c" % mod["name"]
310: (4)             else:
311: (8)                 for mod in postlist:
312: (12)                     mod["coutput"] = options["coutput"]
313: (4)             if options["f2py_wrapper_output"] is None:
314: (8)                 for mod in postlist:
315: (12)                     mod["f2py_wrapper_output"] = "%s-f2pywrappers.f" % mod["name"]
316: (4)             else:
317: (8)                 for mod in postlist:
318: (12)                     mod["f2py_wrapper_output"] = options["f2py_wrapper_output"]
319: (4)             return postlist
320: (0)         def buildmodules(lst):
321: (4)             cfuncs.buildcfuncs()
322: (4)             outmess('Building modules...\n')
323: (4)             modules, mnames, isusedby = [], [], {}
324: (4)             for item in lst:
325: (8)                 if '__user__' in item['name']:
326: (12)                     cb_rules.buildcallbacks(item)
327: (8)                 else:
328: (12)                     if 'use' in item:
329: (16)                         for u in item['use'].keys():
330: (20)                             if u not in isusedby:
331: (24)                                 isusedby[u] = []
332: (20)                                 isusedby[u].append(item['name'])
333: (12)                                 modules.append(item)
334: (12)                                 mnames.append(item['name'])
335: (4)             ret = {}
336: (4)             for module, name in zip(modules, mnames):

```

```

337: (8)
338: (12)
339: (16)
340: (8)
341: (12)
342: (12)
343: (16)
344: (20)
345: (24)
346: (20)
347: (24)
348: (28)
349: (28)
350: (12)
351: (12)
352: (4)
353: (0)
354: (4)
355: (8)
356: (12)
357: (8)
358: (12)
359: (8)
360: (12)
361: (0)
362: (4)
363: (4)
364: (8)
365: (4)
366: (4)
367: (4)
368: (4)
369: (4)
370: (4)
371: (4)
372: (4)
373: (4)
374: (4)
375: (4)
376: (8)
377: (4)
378: (4)
379: (4)
380: (4)
381: (4)
382: (4)
383: (4)
384: (4)
385: (4)
386: (8)
387: (12)
388: (16)
389: (12)
390: (8)
391: (12)
392: (16)
393: (12)
394: (8)
395: (4)
396: (4)
397: (4)
398: (4)
399: (4)
400: (4)
401: (8)
402: (12)
403: (16)
404: (20)

        if name in isusedby:
            outmess('\tSkipping module "%s" which is used by %s.\n' % (
                name, ', '.join(['"%s"' % s for s in isusedby[name]])))
        else:
            um = []
            if 'use' in module:
                for u in module['use'].keys():
                    if u in isusedby and u in mnames:
                        um.append(modules[mnames.index(u)])
                    else:
                        outmess(
                            f'\tModule "{name}" uses nonexisting "{u}" '
                            'which will be ignored.\n')
            ret[name] = {}
            dict_append(ret[name], rules.buildmodule(module, um))
    return ret

def dict_append(d_out, d_in):
    for (k, v) in d_in.items():
        if k not in d_out:
            d_out[k] = []
        if isinstance(v, list):
            d_out[k] = d_out[k] + v
        else:
            d_out[k].append(v)

def run_main(comline_list):
    """
    Equivalent to running::
        f2py <args>
    where ``<args>=string.join(<list>, ' ')``, but in Python. Unless
    ``-h`` is used, this function returns a dictionary containing
    information on generated modules and their dependencies on source
    files.
    You cannot build extension modules with this function, that is,
    using ``-c`` is not allowed. Use the ``compile`` command instead.
    Examples
    -----
    The command ``f2py -m scalar scalar.f`` can be executed from Python as
    follows.
    .. literalinclude:: ../../source/f2py/code/results/run_main_session.dat
        :language: python
    """
    crackfortran.reset_global_f2py_vars()
    f2pydir = os.path.dirname(os.path.abspath(cfuncs.__file__))
    fobjjhsrc = os.path.join(f2pydir, 'src', 'fortranobject.h')
    fobjjcsrc = os.path.join(f2pydir, 'src', 'fortranobject.c')
    parser = make_f2py_compile_parser()
    args, comline_list = parser.parse_known_args(comline_list)
    pyf_files, _ = filter_files("", "[.]pyf([.]src|)", comline_list)
    if args.module_name:
        if "-h" in comline_list:
            modname = (
                args.module_name
            ) # Directly use from args when -h is present
        else:
            modname = validate_modulename(
                pyf_files, args.module_name
            ) # Validate modname when -h is not present
        comline_list += ['-m', modname] # needed for the rest of
        scaninputline
    files, options = scaninputline(comline_list)
    auxfuncs.options = options
    capi_maps.load_f2cmap_file(options['f2cmap_file'])
    postlist = callcrackfortran(files, options)
    isusedby = {}
    for plist in postlist:
        if 'use' in plist:
            for u in plist['use'].keys():
                if u not in isusedby:
                    isusedby[u] = []

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

405: (16)           isusedby[u].append(plist['name'])
406: (4)             for plist in postlist:
407: (8)               if plist['block'] == 'python module' and '__user__' in plist['name']:
408: (12)                 if plist['name'] in isusedby:
409: (16)                   outmess(
410: (20)                     f'Skipping Makefile build for module "{plist["name"]}" '
411: (20)                     'which is used by {}\n'.format(
412: (24)                         ', '.join(f'"{s}"' for s in isusedby[plist['name']])))
413: (4)             if 'signsfile' in options:
414: (8)               if options['verbose'] > 1:
415: (12)                 outmess(
416: (16)                   'Stopping. Edit the signature file and then run f2py on the
signature file: ')
417: (12)           outmess('%s %s\n' %
418: (20)             (os.path.basename(sys.argv[0]), options['signsfile']))
419: (8)             return
420: (4)             for plist in postlist:
421: (8)               if plist['block'] != 'python module':
422: (12)                 if 'python module' not in options:
423: (16)                   errmess(
424: (20)                     'Tip: If your original code is Fortran source then you
must use -m option.\n')
425: (12)           raise TypeError('All blocks must be python module blocks but got
%s' % (
426: (16)             repr(plist['block'])))
427: (4)             auxfuncs.debugoptions = options['debug']
428: (4)             f90mod_rules.options = options
429: (4)             auxfuncs.wrapfuncs = options['wrapfuncs']
430: (4)             ret = buildmodules(postlist)
431: (4)             for mn in ret.keys():
432: (8)               dict_append(ret[mn], {'csrc': fobjcsrc, 'h': fobjhsr})
433: (4)             return ret
434: (0)             def filter_files(prefix, suffix, files, remove_prefix=None):
435: (4)               """
436: (4)                 Filter files by prefix and suffix.
437: (4)               """
438: (4)               filtered, rest = [], []
439: (4)               match = re.compile(prefix + r'.*' + suffix + r'\Z').match
440: (4)               if remove_prefix:
441: (8)                 ind = len(prefix)
442: (4)               else:
443: (8)                 ind = 0
444: (4)               for file in [x.strip() for x in files]:
445: (8)                 if match(file):
446: (12)                   filtered.append(file[ind:])
447: (8)                 else:
448: (12)                     rest.append(file)
449: (4)               return filtered, rest
450: (0)             def get_prefix(module):
451: (4)               p = os.path.dirname(os.path.dirname(module.__file__))
452: (4)               return p
453: (0)             class CombineIncludePaths(argparse.Action):
454: (4)               def __call__(self, parser, namespace, values, option_string=None):
455: (8)                 include_paths_set = set(getattr(namespace, 'include_paths', []))
456: (8)                 if option_string == "--include-paths":
457: (12)                   outmess("Use --include-paths or -I instead of --include-paths
which will be removed")
458: (8)                   if option_string == "--include-paths" or option_string == "--"
459: (12)                     include_paths_set.update(values.split(':'))
460: (8)                   else:
461: (12)                     include_paths_set.add(values)
462: (8)                     setattr(namespace, 'include_paths', list(include_paths_set))
463: (0)             def include_parser():
464: (4)               parser = argparse.ArgumentParser(add_help=False)
465: (4)               parser.add_argument("-I", dest="include_paths",
action=CombineIncludePaths)
466: (4)               parser.add_argument("--include-paths", dest="include_paths",
action=CombineIncludePaths)

```

```

467: (4)             parser.add_argument("--include_paths", dest="include_paths",
action=CombineIncludePaths)
468: (4)         return parser
469: (0)     def get_includes(iline):
470: (4)         iline = (' '.join(iline)).split()
471: (4)         parser = include_parser()
472: (4)         args, remain = parser.parse_known_args(iline)
473: (4)         ipaths = args.include_paths
474: (4)         if args.include_paths is None:
475: (8)             ipaths = []
476: (4)         return ipaths, remain
477: (0)     def make_f2py_compile_parser():
478: (4)         parser = argparse.ArgumentParser(add_help=False)
479: (4)         parser.add_argument("--dep", action="append", dest="dependencies")
480: (4)         parser.add_argument("--backend", choices=['meson', 'distutils'],
default='distutils')
481: (4)         parser.add_argument("-m", dest="module_name")
482: (4)         return parser
483: (0)     def preparse_sysargv():
484: (4)         parser = make_f2py_compile_parser()
485: (4)         args, remaining_argv = parser.parse_known_args()
486: (4)         sys.argv = [sys.argv[0]] + remaining_argv
487: (4)         backend_key = args.backend
488: (4)         if MESON_ONLY_VER and backend_key == 'distutils':
489: (8)             outmess("Cannot use distutils backend with Python>=3.12,"
490: (16)                 " using meson backend instead.\n")
491: (8)             backend_key = "meson"
492: (4)         return {
493: (8)             "dependencies": args.dependencies or [],
494: (8)             "backend": backend_key,
495: (8)             "modulename": args.module_name,
496: (4)         }
497: (0)     def run_compile():
498: (4)         """
499: (4)             Do it all in one call!
500: (4)         """
501: (4)             import tempfile
502: (4)             argy = preparse_sysargv()
503: (4)             modulename = argy["modulename"]
504: (4)             if modulename is None:
505: (8)                 modulename = 'untitled'
506: (4)             dependencies = argy["dependencies"]
507: (4)             backend_key = argy["backend"]
508: (4)             build_backend = f2py_build_generator(backend_key)
509: (4)             i = sys.argv.index('-c')
510: (4)             del sys.argv[i]
511: (4)             remove_build_dir = 0
512: (4)             try:
513: (8)                 i = sys.argv.index('--build-dir')
514: (4)             except ValueError:
515: (8)                 i = None
516: (4)             if i is not None:
517: (8)                 build_dir = sys.argv[i + 1]
518: (8)                 del sys.argv[i + 1]
519: (8)                 del sys.argv[i]
520: (4)             else:
521: (8)                 remove_build_dir = 1
522: (8)                 build_dir = tempfile.mkdtemp()
523: (4)                 _reg1 = re.compile(r'--link-')
524: (4)                 sysinfo_flags = [_m for _m in sys.argv[1:] if _reg1.match(_m)]
525: (4)                 sys.argv = [_m for _m in sys.argv if _m not in sysinfo_flags]
526: (4)                 if sysinfo_flags:
527: (8)                     sysinfo_flags = [f[7:] for f in sysinfo_flags]
528: (4)                     _reg2 = re.compile(
529: (8)                         r'--((no-|)(wrap-functions|lower)|debug-capi|quiet|skip-empty-
wrappers)|-include')
530: (4)                     f2py_flags = [_m for _m in sys.argv[1:] if _reg2.match(_m)]
531: (4)                     sys.argv = [_m for _m in sys.argv if _m not in f2py_flags]
532: (4)                     f2py_flags2 = []

```

```

533: (4)             f1 = 0
534: (4)             for a in sys.argv[1:]:
535: (8)                 if a in ['only:', 'skip:']:
536: (12)                     f1 = 1
537: (8)                 elif a == ':':
538: (12)                     f1 = 0
539: (8)                 if f1 or a == ':':
540: (12)                     f2py_flags2.append(a)
541: (4)             if f2py_flags2 and f2py_flags2[-1] != ':':
542: (8)                 f2py_flags2.append(':')
543: (4)             f2py_flags.extend(f2py_flags2)
544: (4)             sys.argv = [_m for _m in sys.argv if _m not in f2py_flags2]
545: (4)             _reg3 = re.compile(
546: (8)                 r'--((f(90)?compiler(-exec)|compiler)=|help-compiler)')
547: (4)             flib_flags = [_m for _m in sys.argv[1:] if _reg3.match(_m)]
548: (4)             sys.argv = [_m for _m in sys.argv if _m not in flib_flags]
549: (4)             _reg4 = re.compile(
550: (8)                 r'--((f(77|90)(flags|exec)|opt|arch)=|(debug|noopt|noarch|help-
fcompiler))')
551: (4)             fc_flags = [_m for _m in sys.argv[1:] if _reg4.match(_m)]
552: (4)             sys.argv = [_m for _m in sys.argv if _m not in fc_flags]
553: (4)             del_list = []
554: (4)             for s in flib_flags:
555: (8)                 v = '--fcompiler='
556: (8)                 if s[:len(v)] == v:
557: (12)                     if MESON_ONLY_VER or backend_key == 'meson':
558: (16)                         outmess(
559: (20)                             "--fcompiler cannot be used with meson,"
560: (20)                             "set compiler with the FC environment variable\n"
561: (20)                         )
562: (12)                 else:
563: (16)                     from numpy.distutils import fcompiler
564: (16)                     fcompiler.load_all_fcompiler_classes()
565: (16)                     allowed_keys = list(fcompiler.fcompiler_class.keys())
566: (16)                     nv = ov = s[len(v):].lower()
567: (16)                     if ov not in allowed_keys:
568: (20)                         vmap = {} # XXX
569: (20)                         try:
570: (24)                             nv = vmap[ov]
571: (20)                         except KeyError:
572: (24)                             if ov not in vmap.values():
573: (28)                                 print('Unknown vendor: "%s" % (s[len(v):]))')
574: (20)                             nv = ov
575: (16)                             i = flib_flags.index(s)
576: (16)                             flib_flags[i] = '--fcompiler=' + nv
577: (16)                             continue
578: (4)             for s in del_list:
579: (8)                 i = flib_flags.index(s)
580: (8)                 del flib_flags[i]
581: (4)             assert len(flib_flags) <= 2, repr(flib_flags)
582: (4)             _reg5 = re.compile(r'--(verbose)')
583: (4)             setup_flags = [_m for _m in sys.argv[1:] if _reg5.match(_m)]
584: (4)             sys.argv = [_m for _m in sys.argv if _m not in setup_flags]
585: (4)             if '--quiet' in f2py_flags:
586: (8)                 setup_flags.append('--quiet')
587: (4)             sources = sys.argv[1:]
588: (4)             f2cmapopt = '--f2cmap'
589: (4)             if f2cmapopt in sys.argv:
590: (8)                 i = sys.argv.index(f2cmapopt)
591: (8)                 f2py_flags.extend(sys.argv[i:i + 2])
592: (8)                 del sys.argv[i + 1], sys.argv[i]
593: (8)                 sources = sys.argv[1:]
594: (4)             pyf_files, _sources = filter_files("", "[.]pyf([.]src|)", sources)
595: (4)             sources = pyf_files + _sources
596: (4)             modulename = validate_modulename(pyf_files, modulename)
597: (4)             extra_objects, sources = filter_files('', '[.]o[|a|so|dylib]', sources)
598: (4)             library_dirs, sources = filter_files('-L', '', sources, remove_prefix=1)
599: (4)             libraries, sources = filter_files('-l', '', sources, remove_prefix=1)
600: (4)             undef_macros, sources = filter_files('-U', '', sources, remove_prefix=1)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

601: (4)             define_macros, sources = filter_files('-D', '', sources, remove_prefix=1)
602: (4)             for i in range(len(define_macros)):
603: (8)                 name_value = define_macros[i].split('=', 1)
604: (8)                 if len(name_value) == 1:
605: (12)                     name_value.append(None)
606: (8)                 if len(name_value) == 2:
607: (12)                     define_macros[i] = tuple(name_value)
608: (8)                 else:
609: (12)                     print('Invalid use of -D:', name_value)
610: (4)             if backend_key == 'meson':
611: (8)                 if not pyf_files:
612: (12)                     outmess('Using meson backend\nWill pass --lower to f2py\nSee
https://numpy.org/doc/stable/f2py/buildtools/meson.html\n')
613: (12)                     f2py_flags.append('--lower')
614: (12)                     run_main(f" {' '.join(f2py_flags)} -m {modulename} {'.
'.join(sources)}".split())
615: (8)             else:
616: (12)                 run_main(f" {' '.join(f2py_flags)} {' '.join(pyf_files)}".split())
617: (4)             include_dirs, sources = get_includes(sources)
618: (4)             builder = build_backend(
619: (8)                 modulename,
620: (8)                 sources,
621: (8)                 extra_objects,
622: (8)                 build_dir,
623: (8)                 include_dirs,
624: (8)                 library_dirs,
625: (8)                 libraries,
626: (8)                 define_macros,
627: (8)                 undef_macros,
628: (8)                 f2py_flags,
629: (8)                 sysinfo_flags,
630: (8)                 fc_flags,
631: (8)                 flib_flags,
632: (8)                 setup_flags,
633: (8)                 remove_build_dir,
634: (8)                 {"dependencies": dependencies},
635: (4)             )
636: (4)             builder.compile()
637: (0)             def validate_modulename(pyf_files, modulename='untitled'):
638: (4)                 if len(pyf_files) > 1:
639: (8)                     raise ValueError("Only one .pyf file per call")
640: (4)                 if pyf_files:
641: (8)                     pyff = pyf_files[0]
642: (8)                     pyf_modname = auxfuncs.get_f2py_modulename(pyff)
643: (8)                     if modulename != pyf_modname:
644: (12)                         outmess(
645: (16)                             f"Ignoring -m {modulename}.\n"
646: (16)                             f"{pyff} defines {pyf_modname} to be the modulename.\n"
647: (12)
648: (12)
649: (4)                         modulename = pyf_modname
650: (0)                     return modulename
651: (4)             def main():
652: (8)                 if '--help-link' in sys.argv[1:]:
653: (8)                     sys.argv.remove('--help-link')
654: (8)                     if MESON_ONLY_VER:
655: (12)                         outmess("Use --dep for meson builds\n")
656: (8)                     else:
657: (12)                         from numpy.distutils.system_info import show_all
658: (8)                         show_all()
659: (8)                     return
660: (4)                 if '-c' in sys.argv[1:]:
661: (8)                     run_compile()
662: (4)                 else:
663: (8)                     run_main(sys.argv[1:])

```

-----  
File 150 - func2subr.py:

```

1: (0)
2: (0)     """
3: (0)         Rules for building C/API module with f2py2e.
4: (0)         Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
5: (0)         Copyright 2011 -- present NumPy Developers.
6: (0)         Permission to use, modify, and distribute this software is given under the
7: (0)         terms of the NumPy License.
8: (0)         NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
9: (0)         """
10: (0)        import copy
11: (4)        from .auxfuncs import (
12: (4)            getfortranname, isexternal, isfunction, isfunction_wrap, isintent_in,
13: (4)            isintent_out, islogicalfunction, ismoduleroutine, isscalar,
14: (4)            isssubroutine, isssubroutine_wrap, outmess, show
15: (0)        )
16: (0)        from ._isocbind import isoc_kindmap
17: (0)        def var2fixfortran(vars, a, fa=None, f90mode=None):
18: (8)            if fa is None:
19: (8)                fa = a
20: (8)            if a not in vars:
21: (8)                show(vars)
22: (8)                outmess('var2fixfortran: No definition for argument "%s".\n' % a)
23: (8)                return ''
24: (8)            if 'typespec' not in vars[a]:
25: (8)                show(vars[a])
26: (8)                outmess('var2fixfortran: No typespec for argument "%s".\n' % a)
27: (8)                return ''
28: (4)            vardef = vars[a]['typespec']
29: (8)            if vardef == 'type' and 'typename' in vars[a]:
30: (8)                vardef = '%s(%s)' % (vardef, vars[a]['typename'])
31: (4)            selector = {}
32: (4)            lk = ''
33: (8)            if 'kindselector' in vars[a]:
34: (8)                selector = vars[a]['kindselector']
35: (8)                lk = 'kind'
36: (8)            elif 'charselector' in vars[a]:
37: (8)                selector = vars[a]['charselector']
38: (8)                lk = 'len'
39: (8)            if '*' in selector:
40: (12)                if f90mode:
41: (16)                    if selector['*'] in ['*', ':', '(*)']:
42: (12)                        vardef = '%s(len=*)' % (vardef)
43: (16)                    else:
44: (8)                        vardef = '%s(%s=%s)' % (vardef, lk, selector['*'])
45: (12)                else:
46: (16)                    if selector['*'] in ['*', ':']:
47: (12)                        vardef = '%s*(%s)' % (vardef, selector['*'])
48: (16)                    else:
49: (8)                        vardef = '%s*%s' % (vardef, selector['*'])
50: (8)            else:
51: (12)                if 'len' in selector:
52: (12)                    vardef = '%s(len=%s' % (vardef, selector['len']))
53: (16)                    if 'kind' in selector:
54: (12)                        vardef = '%s,kind=%s)' % (vardef, selector['kind'])
55: (16)                    else:
56: (8)                        vardef = '%s)' % (vardef)
57: (12)                elif 'kind' in selector:
58: (12)                    vardef = '%s(kind=%s)' % (vardef, selector['kind'])
59: (4)            vardef = '%s %s' % (vardef, fa)
60: (8)            if 'dimension' in vars[a]:
61: (8)                vardef = '%s(%s)' % (vardef, ','.join(vars[a]['dimension']))
62: (0)            return vardef
63: (4)        def useiso_c_binding(rout):
64: (4)            useisoc = False
65: (8)            for key, value in rout['vars'].items():
66: (8)                kind_value = value.get('kindselector', {}).get('kind')
67: (8)                if kind_value in isoc_kindmap:
68: (4)                    return True
69: (0)            return useisoc
70: (0)        def createfuncwrapper(rout, signature=0):

```

```

70: (4)             assert isfunction(rout)
71: (4)             extra_args = []
72: (4)             vars = rout['vars']
73: (4)             for a in rout['args']:
74: (8)                 v = rout['vars'][a]
75: (8)                 for i, d in enumerate(v.get('dimension', [])):
76: (12)                   if d == ':':
77: (16)                     dn = 'f2py_%s_d%s' % (a, i)
78: (16)                     dv = dict(typespec='integer', intent=['hide'])
79: (16)                     dv['='] = 'shape(%s, %s)' % (a, i)
80: (16)                     extra_args.append(dn)
81: (16)                     vars[dn] = dv
82: (16)                     v['dimension'][i] = dn
83: (4)             rout['args'].extend(extra_args)
84: (4)             need_interface = bool(extra_args)
85: (4)             ret = ['']
86: (4)             def add(line, ret=ret):
87: (8)                 ret[0] = '%s\n      %s' % (ret[0], line)
88: (4)             name = rout['name']
89: (4)             fortranname = getfortranname(rout)
90: (4)             f90mode = ismoduleroutine(rout)
91: (4)             newname = '%sf2pywrap' % (name)
92: (4)             if newname not in vars:
93: (8)                 vars[newname] = vars[name]
94: (8)                 args = [newname] + rout['args'][1:]
95: (4)             else:
96: (8)                 args = [newname] + rout['args']
97: (4)             l_tmpl = var2fixfortran(vars, name, '@@@NAME@@@', f90mode)
98: (4)             if l_tmpl[:13] == 'character(*)':
99: (8)                 if f90mode:
100: (12)                  l_tmpl = 'character(len=10)' + l_tmpl[13:]
101: (8)                 else:
102: (12)                  l_tmpl = 'character*10' + l_tmpl[13:]
103: (8)                 chiselect = vars[name]['chiselect']
104: (8)                 if chiselect.get('*', '') == '(*':
105: (12)                   chiselect['*'] = '10'
106: (4)                 l1 = l_tmpl.replace('@@@NAME@@@', newname)
107: (4)                 r1 = None
108: (4)                 useisoc = useiso_c_binding(rout)
109: (4)                 sargs = ', '.join(args)
110: (4)                 if f90mode:
111: (8)                     sargs = sargs.replace(f"{{name}}", ", ")
112: (8)                     args = [arg for arg in args if arg != name]
113: (8)                     rout['args'] = args
114: (8)                     add('subroutine f2pywrap%_s (%s)' %
115: (12)                       (rout['modulename'], name, sargs))
116: (8)                     if not signature:
117: (12)                       add('use %s, only : %s' % (rout['modulename'], fortranname))
118: (8)                     if useisoc:
119: (12)                       add('use iso_c_binding')
120: (4)                     else:
121: (8)                       add('subroutine f2pywrap%_s (%s)' % (name, sargs))
122: (8)                       if useisoc:
123: (12)                         add('use iso_c_binding')
124: (8)                         if not need_interface:
125: (12)                           add('external %s' % (fortranname))
126: (12)                           r1 = l_tmpl.replace('@@@NAME@@@', '') + ' ' + fortranname
127: (4)                         if need_interface:
128: (8)                           for line in rout['saved_interface'].split('\n'):
129: (12)                             if line.lstrip().startswith('use ') and '__user__' not in line:
130: (16)                               add(line)
131: (4)                             args = args[1:]
132: (4)                             dumped_args = []
133: (4)                             for a in args:
134: (8)                               if isexternal(vars[a]):
135: (12)                                 add('external %s' % (a))
136: (12)                                 dumped_args.append(a)
137: (4)                               for a in args:
138: (8)                                 if a in dumped_args:

```

```

139: (12)           continue
140: (8)            if isscalar(vars[a]):
141: (12)              add(var2fixfortran(vars, a, f90mode=f90mode))
142: (12)              dumped_args.append(a)
143: (4)            for a in args:
144: (8)              if a in dumped_args:
145: (12)                  continue
146: (8)              if isintent_in(vars[a]):
147: (12)                  add(var2fixfortran(vars, a, f90mode=f90mode))
148: (12)                  dumped_args.append(a)
149: (4)            for a in args:
150: (8)              if a in dumped_args:
151: (12)                  continue
152: (8)                  add(var2fixfortran(vars, a, f90mode=f90mode))
153: (4)            add(11)
154: (4)            if rl is not None:
155: (8)                add(rl)
156: (4)            if need_interface:
157: (8)                if f90mode:
158: (12)                    pass
159: (8)                else:
160: (12)                    add('interface')
161: (12)                    add(rout['saved_interface'].lstrip())
162: (12)                    add('end interface')
163: (4)            sargs = ', '.join([a for a in args if a not in extra_args])
164: (4)            if not signature:
165: (8)                if islogicalfunction(rout):
166: (12)                    add('%s = .not.(.not.%s(%s))' % (newname, fortranname, sargs))
167: (8)                else:
168: (12)                    add('%s = %s(%s)' % (newname, fortranname, sargs))
169: (4)            if f90mode:
170: (8)                add('end subroutine f2pywrap_%s_%s' % (rout['modulename'], name))
171: (4)            else:
172: (8)                add('end')
173: (4)            return ret[0]
174: (0)        def createsubrwrapper(rout, signature=0):
175: (4)            assert issubroutine(rout)
176: (4)            extra_args = []
177: (4)            vars = rout['vars']
178: (4)            for a in rout['args']:
179: (8)                v = rout['vars'][a]
180: (8)                for i, d in enumerate(v.get('dimension', [])):
181: (12)                    if d == ':':
182: (16)                        dn = 'f2py_%s_d%s' % (a, i)
183: (16)                        dv = dict(typespec='integer', intent=['hide'])
184: (16)                        dv['='] = 'shape(%s, %s)' % (a, i)
185: (16)                        extra_args.append(dn)
186: (16)                        vars[dn] = dv
187: (16)                        v['dimension'][i] = dn
188: (4)            rout['args'].extend(extra_args)
189: (4)            need_interface = bool(extra_args)
190: (4)            ret = ['']
191: (4)            def add(line, ret=ret):
192: (8)                ret[0] = '%s\n    %s' % (ret[0], line)
193: (4)                name = rout['name']
194: (4)                fortranname = getfortranname(rout)
195: (4)                f90mode = ismoduleroutine(rout)
196: (4)                args = rout['args']
197: (4)                useisoc = useiso_c_binding(rout)
198: (4)                sargs = ', '.join(args)
199: (4)                if f90mode:
200: (8)                    add('subroutine f2pywrap_%s_%s (%s)' %
201: (12)                      (rout['modulename'], name, sargs))
202: (8)                    if useisoc:
203: (12)                        add('use iso_c_binding')
204: (8)                        if not signature:
205: (12)                            add('use %s, only : %s' % (rout['modulename'], fortranname))
206: (4)                else:
207: (8)                    add('subroutine f2pywrap% (%s)' % (name, sargs))

```

```

208: (8)           if useisoc:
209: (12)             add('use iso_c_binding')
210: (8)           if not need_interface:
211: (12)             add('external %s' % (fortranname))
212: (4)           if need_interface:
213: (8)             for line in rout['saved_interface'].split('\n'):
214: (12)               if line.lstrip().startswith('use ') and '__user__' not in line:
215: (16)                 add(line)
216: (4)           dumped_args = []
217: (4)           for a in args:
218: (8)             if isexternal(vars[a]):
219: (12)               add('external %s' % (a))
220: (12)               dumped_args.append(a)
221: (4)           for a in args:
222: (8)             if a in dumped_args:
223: (12)               continue
224: (8)             if isscalar(vars[a]):
225: (12)               add(var2fixfortran(vars, a, f90mode=f90mode))
226: (12)               dumped_args.append(a)
227: (4)           for a in args:
228: (8)             if a in dumped_args:
229: (12)               continue
230: (8)             add(var2fixfortran(vars, a, f90mode=f90mode))
231: (4)           if need_interface:
232: (8)             if f90mode:
233: (12)               pass
234: (8)             else:
235: (12)               add('interface')
236: (12)               for line in rout['saved_interface'].split('\n'):
237: (16)                 if line.lstrip().startswith('use ') and '__user__' in line:
238: (20)                   continue
239: (16)                 add(line)
240: (12)               add('end interface')
241: (4)           sargs = ', '.join([a for a in args if a not in extra_args])
242: (4)           if not signature:
243: (8)             add('call %s(%s)' % (fortranname, sargs))
244: (4)           if f90mode:
245: (8)             add('end subroutine f2pywrap_%s_%s' % (rout['modulename'], name))
246: (4)           else:
247: (8)             add('end')
248: (4)           return ret[0]
249: (0)           def assubr(rout):
250: (4)             if isfunction_wrap(rout):
251: (8)               fortranname = getfortranname(rout)
252: (8)               name = rout['name']
253: (8)               outmess('\t\tCreating wrapper for Fortran function "%s"("%s")...\n' %
(
254: (12)                 name, fortranname))
255: (8)               rout = copy.copy(rout)
256: (8)               fname = name
257: (8)               rname = fname
258: (8)               if 'result' in rout:
259: (12)                 rname = rout['result']
260: (12)                 rout['vars'][fname] = rout['vars'][rname]
261: (8)               fvar = rout['vars'][fname]
262: (8)               if not isintent_out(fvar):
263: (12)                 if 'intent' not in fvar:
264: (16)                   fvar['intent'] = []
265: (12)                   fvar['intent'].append('out')
266: (12)                   flag = 1
267: (12)                   for i in fvar['intent']:
268: (16)                     if i.startswith('out='):
269: (20)                       flag = 0
270: (20)                       break
271: (12)                   if flag:
272: (16)                     fvar['intent'].append('out=%s' % (rname))
273: (8)                   rout['args'][ :] = [fname] + rout['args']
274: (8)                   return rout, createfuncwrapper(rout)
275: (4)           if issubroutine_wrap(rout):

```

```

276: (8)           fortranname = getfortranname(rout)
277: (8)           name = rout['name']
278: (8)           outmess('\t\tCreating wrapper for Fortran subroutine "%s"("%s")...\n'
279: (16)             % (name, fortranname))
280: (8)           rout = copy.copy(rout)
281: (8)           return rout, createsubrwrapper(rout)
282: (4)           return rout, ''

```

---

File 151 - f90mod\_rules.py:

```

1: (0)           """
2: (0)           Build F90 module support for f2py2e.
3: (0)           Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
4: (0)           Copyright 2011 -- present NumPy Developers.
5: (0)           Permission to use, modify, and distribute this software is given under the
6: (0)           terms of the NumPy License.
7: (0)           NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
8: (0)           """
9: (0)           __version__ = "$Revision: 1.27 $"[10:-1]
10: (0)          f2py_version = 'See `f2py -v`'
11: (0)          import numpy as np
12: (0)          from . import capi_maps
13: (0)          from . import func2subr
14: (0)          from .crackfortran import undo_rmbadname, undo_rmbadname1
15: (0)          from .auxfuncs import *
16: (0)          options = {}
17: (0)          def findf90modules(m):
18: (4)            if ismodule(m):
19: (8)              return [m]
20: (4)            if not hasbody(m):
21: (8)              return []
22: (4)            ret = []
23: (4)            for b in m['body']:
24: (8)              if ismodule(b):
25: (12)                ret.append(b)
26: (8)              else:
27: (12)                ret = ret + findf90modules(b)
28: (4)            return ret
29: (0)          fgetdims1 = """\
30: (6)            external f2pysetdata
31: (6)            logical ns
32: (6)            integer r,i
33: (6)            integer(%d) s(*)
34: (6)            ns = .FALSE.
35: (6)            if (allocated(d)) then
36: (9)              do i=1,r
37: (12)                if ((size(d,i).ne.s(i)).and.(s(i).ge.0)) then
38: (15)                  ns = .TRUE.
39: (12)                end if
40: (9)              end do
41: (9)              if (ns) then
42: (12)                deallocate(d)
43: (9)              end if
44: (6)            end if
45: (6)            if ((.not.allocated(d)).and.(s(1).ge.1)) then"""\% np.intp().itemsize
46: (0)          fgetdims2 = """\
47: (6)            end if
48: (6)            if (allocated(d)) then
49: (9)              do i=1,r
50: (12)                s(i) = size(d,i)
51: (9)              end do
52: (6)            end if
53: (6)            flag = 1
54: (6)            call f2pysetdata(d,allocated(d))"""
55: (0)          fgetdims2_sa = """\
56: (6)            end if
57: (6)            if (allocated(d)) then

```

```

58: (9)           do i=1,r
59: (12)         s(i) = size(d,i)
60: (9)       end do
61: (9)     !s(r) must be equal to len(d(1))
62: (6)   end if
63: (6)   flag = 2
64: (6)   call f2pysetdata(d,allocated(d))"""
65: (0) def buildhooks(pymod):
66: (4)   from . import rules
67: (4)   ret = {'f90modhooks': [], 'initf90modhooks': [], 'body': [],
68: (11)     'need': ['F_FUNC', 'arrayobject.h'],
69: (11)     'separatorsfor': {'includes0': '\n', 'includes': '\n'},
70: (11)     'docs': ['"Fortran 90/95 modules:\\n"'],
71: (11)     'latexdoc': []}
72: (4)   fhooks = ['']
73: (4)   def fadd(line, s=fhooks):
74: (8)     s[0] = '%s\n      %s' % (s[0], line)
75: (4)   doc = ['']
76: (4)   def dadd(line, s=doc):
77: (8)     s[0] = '%s\n%s' % (s[0], line)
78: (4)   usenames = getuseblocks(pymod)
79: (4)   for m in findf90modules(pymod):
80: (8)     sargs, fargs, efargs, modobjs, notvars, onlyvars = [], [], [], [], []
81: (12)     m['name']], []
82: (8)     sargsp = []
83: (8)     ifargs = []
84: (8)     mfargs = []
85: (8)     if hasbody(m):
86: (12)       for b in m['body']:
87: (16)         notvars.append(b['name'])
88: (8)     for n in m['vars'].keys():
89: (12)       var = m['vars'][n]
90: (12)       if (n not in notvars) and (not l_or(isintent_hide, isprivate)
(var)):
91: (16)         onlyvars.append(n)
92: (16)         mfargs.append(n)
93: (8)       outmess('\t\tConstructing F90 module support for "%s"...\\n' %
94: (16)         (m['name']))
95: (8)       if m['name'] in usenames and not onlyvars:
96: (12)         outmess(f"\t\t\tSkipping {m['name']} since it is in 'use'...\\n")
97: (12)         continue
98: (8)       if onlyvars:
99: (12)         outmess('\t\t Variables: %s\\n' % (' '.join(onlyvars)))
100: (8)       chooks = ['']
101: (8)       def cadd(line, s=chooks):
102: (12)         s[0] = '%s\\n%s' % (s[0], line)
103: (8)       ihooks = ['']
104: (8)       def iadd(line, s=ihooks):
105: (12)         s[0] = '%s\\n%s' % (s[0], line)
106: (8)       vrd = capi_maps.modsign2map(m)
107: (8)       cadd('static FortranDataDef f2py_%s_def[] = {' % (m['name']))
108: (8)       dadd('\\subsection{Fortran 90/95 module \\texttt{\\%s}}\\n' %
(m['name']))
109: (8)       if hasnote(m):
110: (12)         note = m['note']
111: (12)         if isinstance(note, list):
112: (16)           note = '\\n'.join(note)
113: (12)           dadd(note)
114: (8)         if onlyvars:
115: (12)           dadd('\\begin{description}')
116: (8)         for n in onlyvars:
117: (12)           var = m['vars'][n]
118: (12)           modobjs.append(n)
119: (12)           ct = capi_maps.getctype(var)
120: (12)           at = capi_maps.c2capi_map[ct]
121: (12)           dm = capi_maps.getarrdims(n, var)
122: (12)           dms = dm['dims'].replace('*', '-1').strip()
123: (12)           dms = dms.replace(':', '-1').strip()
124: (12)           if not dms:

```

```

125: (16)           dms = '-1'
126: (12)           use_fgetdims2 = fgetdims2
127: (12)           cadd('\t{"%s",%s,{%s}},%s,%s,' %
128: (17)             (undo_rmbadname1(n), dm['rank'], dms, at,
129: (18)               capi_maps.get_elsize(var)))
130: (12)           dadd('"\item[]{}"\verb@%s@{}" %
131: (17)             (capi_maps.getarrdocsing(n, var)))
132: (12)           if hasnote(var):
133: (16)             note = var['note']
134: (16)             if isinstance(note, list):
135: (20)               note = '\n'.join(note)
136: (16)               dadd('--- %s' % (note))
137: (12)           if isallocatable(var):
138: (16)             fargs.append('f2py_%s_getdims_%s' % (m['name'], n))
139: (16)             efargs.append(fargs[-1])
140: (16)             sargs.append(
141: (20)               'void (*%s)(int*,npy_intp*,void*)(char*,npy_intp*),int*)'
142: (16)             (char*,npy_intp*),int*)')
143: (16)           sargsp.append('void (*)(int*,npy_intp*,void(*)'
144: (16)             iadd('\tf2py_%s_def[i_f2py++].func = %s;' % (m['name'], n))
145: (16)             fadd('subroutine %s(r,s,f2pysetdata,flag)' % (fargs[-1]))
146: (16)             fadd('use %s, only: d => %s\n' %
147: (21)               (m['name'], undo_rmbadname1(n)))
148: (16)             fadd('integer flag\n')
149: (16)             fhooks[0] = fhooks[0] + fgetdims1
150: (16)             dms = range(1, int(dm['rank']) + 1)
151: (21)             fadd(' allocate(d(%s))\n' %
152: (16)               (','.join(['s(%s)' % i for i in dms])))
153: (16)             fhooks[0] = fhooks[0] + use_fgetdims2
154: (12)             fadd('end subroutine %s' % (fargs[-1]))
155: (16)           else:
156: (16)             fargs.append(n)
157: (16)             sargs.append('char *%s' % (n))
158: (16)             sargsp.append('char*')
159: (8)               iadd('\tf2py_%s_def[i_f2py++].data = %s;' % (m['name'], n))
160: (12)             if onlyvars:
161: (8)               dadd('"\end{description}')
162: (12)             if hasbody(m):
163: (16)               for b in m['body']:
164: (20)                 if not isroutine(b):
165: (28)                   outmess("f90mod_rules.buildhooks:"
166: (20)                     "f" skipping {b['block']} {b['name']}\n")
167: (16)                   continue
168: (16)                   modobjs.append('%s()' % (b['name']))
169: (16)                   b['modulename'] = m['name']
170: (16)                   api, wrap = rules.buildapi(b)
171: (20)                   if isfunction(b):
172: (20)                     fhooks[0] = fhooks[0] + wrap
173: (20)                     fargs.append('f2pywrap_%s_%s' % (m['name'], b['name']))
174: (16)                     ifargs.append(func2subr.createfuncwrapper(b, signature=1))
175: (20)                   else:
176: (24)                     if wrap:
177: (24)                       fhooks[0] = fhooks[0] + wrap
178: (24)                       fargs.append('f2pywrap_%s_%s' % (m['name'],
179: (28)                         ifargs.append(
180: (20)                           func2subr.createsubrwrapper(b, signature=1))
181: (24)                         else:
182: (24)                           fargs.append(b['name'])
183: (16)                           mfargs.append(fargs[-1])
184: (16)                           api['externroutines'] = []
185: (16)                           ar = applyrules(api, vrd)
186: (16)                           ar['docs'] = []
187: (16)                           ar['docshort'] = []
188: (16)                           ret = dictappend(ret, ar)
189: (22)                           cadd(('\t{"%s",-1,{%-1}},0,0,NULL,(void *)'
190: (22)                             'f2py_rout_#modulename#_%s_%s,'
191: (22)                             'doc_f2py_rout_#modulename#_%s_%s},')

```

```

191: (21)                                     % (b['name'], m['name'], b['name'], m['name'],
b['name']))
192: (16)                                     sargs.append('char *%s' % (b['name']))
193: (16)                                     sargsp.append('char *')
194: (16)                                     iadd('\tf2py_%s_def[i_f2py++].data = %s;' %
195: (21)                                     (m['name'], b['name']))
196: (8)                                      cadd('\t{NULL}\n};\n')
197: (8)                                      iadd('}')
198: (8)                                      ihooks[0] = 'static void f2py_setup_%s(%s) {\n\tint i_f2py=0;%s' %
199: (12)                                     m['name'], ',' .join(sargs), ihooks[0])
200: (8)                                     if '_' in m['name']:
201: (12)                                     F_FUNC = 'F_FUNC_US'
202: (8)                                     else:
203: (12)                                     F_FUNC = 'F_FUNC'
204: (8)                                     iadd('extern void %s(f2pyinit%s,F2PYINIT%s)(void (*)(%s));' %
205: (13)                                     % (F_FUNC, m['name'], m['name'].upper(), ',' .join(sargsp)))
206: (8)                                     iadd('static void f2py_init_%s(void) {' % (m['name']))
207: (8)                                     iadd('\t%s(f2pyinit%s,F2PYINIT%s)(f2py_setup_%s);' %
208: (13)                                     % (F_FUNC, m['name'], m['name'].upper(), m['name']))
209: (8)                                     iadd('}\n')
210: (8)                                     ret['f90modhooks'] = ret['f90modhooks'] + chooks + ihooks
211: (8)                                     ret['initf90modhooks'] = ['\tPyDict_SetItemString(d, "%s",
PyFortranObject_New(f2py_%s_def,f2py_init_%s));' %
212: (12)                                     m['name'], m['name'], m['name'])] + ret['initf90modhooks']
213: (8)                                     fadd('')
214: (8)                                     fadd('subroutine f2pyinit%s(f2pysetupfunc)' % (m['name']))
215: (8)                                     if mfargs:
216: (12)                                     for a in undo_rmbadname(mfargs):
217: (16)                                     fadd('use %s, only : %s' % (m['name'], a))
218: (8)
219: (12)                                     if ifargs:
220: (12)                                     fadd(' '.join(['interface'] + ifargs))
221: (8)                                     fadd('end interface')
222: (8)
223: (12)                                     if efargs:
224: (16)                                     for a in undo_rmbadname(efargs):
225: (8)                                     fadd('external %s' % (a))
226: (8)                                     fadd('call f2pysetupfunc(%s)' % (',' .join(undo_rmbadname(fargs))))
227: (8)                                     fadd('end subroutine f2pyinit%s\n' % (m['name']))
228: (12)                                     dadd('\n'.join(ret['latexdoc']).replace(
229: (8)                                     r'\subsection{', r'\subsubsection{'))
230: (8)                                     ret['latexdoc'] = []
231: (46)                                     ret['docs'].append("\t%s --- %s" % (m['name'],
232: (4)                                     ', '.join(undo_rmbadname(modobjs))))
233: (4)                                     ret['routine_defs'] = ''
234: (4)                                     ret['doc'] = []
235: (4)                                     ret['docshort'] = []
236: (4)                                     ret['latexdoc'] = doc[0]
237: (8)                                     if len(ret['docs']) <= 1:
238: (4)                                     ret['docs'] = ''
239: (4)                                     return ret, fhooks[0]

```

---

## File 152 - rules.py:

```

1: (0)                                     """
2: (0)                                     Rules for building C/API module with f2py2e.
3: (0)                                     Here is a skeleton of a new wrapper function (13Dec2001):
4: (0)                                     wrapper_function(args)
5: (2)                                     declarations
6: (2)                                     get_python_arguments, say, `a` and `b'
7: (2)                                     get_a_from_python
8: (2)                                     if (successful) {
9: (4)   get_b_from_python
10: (4)   if (successful) {
11: (6)   callfortran
12: (6)   if (successful) {
13: (8)   put_a_to_python

```

```

14: (8)             if (successful) {
15: (10)             put_b_to_python
16: (10)             if (successful) {
17: (12)                 buildvalue = ...
18: (10)             }
19: (8)         }
20: (6)     }
21: (4)   }
22: (4)   cleanup_b
23: (2)
24: (2)   cleanup_a
25: (2)   return buildvalue
26: (0) Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
27: (0) Copyright 2011 -- present NumPy Developers.
28: (0) Permission to use, modify, and distribute this software is given under the
29: (0) terms of the NumPy License.
30: (0) NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
31: (0)
32: (0) """
33: (0) import os, sys
34: (0) import time
35: (0) import copy
36: (0) from pathlib import Path
37: (0) from . import __version__
38: (0) from .auxfuncs import (
39: (4)     applyrules, debugcapi, dictappend, errmess, gentitle, getargs2,
40: (4)     hascallstatement, hasexternals, hasinitvalue, hasnote,
41: (4)     hasresultnote, isarray, isarrayofstrings, ischaracter,
42: (4)     ischaracterarray, ischaracter_or_characterarray, iscomplex,
43: (4)     iscomplexarray, iscomplexfunction, iscomplexfunction_warn,
44: (4)     isdummysubroutine, isexternal, isfunction, isfunction_wrap, isint1,
45: (4)     isint1array, isintent_aux, isintent_c, isintent_callback,
46: (4)     isintent_copy, isintent_hide, isintent_inout, isintent_nothide,
47: (4)     isintent_out, isintent_overwrite, islogical, islong_complex,
48: (4)     islong_double, islong_doublefunction, islong_long,
49: (4)     islong_longfunction, ismoduleroutine, isoptional, isrequired,
50: (4)     isscalar, issigned_long_longarray, isstring, isstringarray,
51: (4)     isstringfunction, issubroutine, isattr_value,
52: (4)     issubroutine_wrap, isthreadsafe, isunsigned, isunsigned_char,
53: (4)     isunsigned_chararray, isunsigned_long_long,
54: (4)     isunsigned_long_longarray, isunsigned_short, isunsigned_shortarray,
55: (4)     l_and, l_not, l_or, outmess, replace, stripcomma, requiresf90wrapper
56: (0) )
57: (0) from . import capi_maps
58: (0) from . import cfuncs
59: (0) from . import common_rules
60: (0) from . import use_rules
61: (0) from . import f90mod_rules
62: (0) from . import func2subr
63: (0) f2py_version = __version__.version
64: (0) numpy_version = __version__.version
65: (0) options = {}
66: (0) sepdict = {}
67: (0) for k in ['decl',
68: (10)     'frompyobj',
69: (10)     'cleanupfrompyobj',
70: (10)     'topyarr', 'method',
71: (10)     'pyobjfrom', 'closepyobjfrom',
72: (10)     'freemem',
73: (10)     'userincludes',
74: (10)     'includes0', 'includes', 'typedefs', 'typedefs_generated',
75: (10)     'cppmacros', 'cfuncs', 'callbacks',
76: (10)     'latexdoc',
77: (10)     'restdoc',
78: (10)     'routine_defs', 'externroutines',
79: (10)     'initf2pywraphooks',
80: (10)     'commonhooks', 'initcommonhooks',
81: (4)     'f90modhooks', 'initf90modhooks']:
82: (0)     sepdict[k] = '\n'

```

generationtime = int(os.environ.get('SOURCE\_DATE\_EPOCH', time.time()))

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

83: (0) module_rules = {
84: (4)     'modulebody': """\
85: (0)     /* File: #modulename#module.c
86: (1)     * This file is auto-generated with f2py (version:#f2py_version#).
87: (1)     * f2py is a Fortran to Python Interface Generator (FPIG), Second Edition,
88: (1)     * written by Pearu Peterson <pearu@cens.ioc.ee>.
89: (1)     * Generation date: """ + time.asctime(time.gmtime(generationtime)) + """
90: (1)     * Do not edit this file directly unless you know what you are doing!!!
91: (1) */
92: (0) extern \"C\" {
93: (0) /* Unconditionally included */
94: (0) """ + gentitle("See f2py2e/cfuncs.py: includes") + """
95: (0) """ + gentitle("See f2py2e/rules.py: mod_rules['modulebody']") + """
96: (0) static PyObject *#modulename#_error;
97: (0) static PyObject *#modulename#_module;
98: (0) """ + gentitle("See f2py2e/cfuncs.py: typedefs") + """
99: (0) """ + gentitle("See f2py2e/cfuncs.py: typedefs_generated") + """
100: (0) """ + gentitle("See f2py2e/cfuncs.py: cppmacros") + """
101: (0) """ + gentitle("See f2py2e/cfuncs.py: cfuncs") + """
102: (0) """ + gentitle("See f2py2e/cfuncs.py: userincludes") + """
103: (0) """ + gentitle("See f2py2e/capi_rules.py: usercode") + """
104: (0) /* See f2py2e/rules.py */
105: (0) """ + gentitle("See f2py2e/capi_rules.py: usercode1") + """
106: (0) """ + gentitle("See f2py2e/cb_rules.py: buildcallback") + """
107: (0) """ + gentitle("See f2py2e/rules.py: buildapi") + """
108: (0) """ + gentitle("See f2py2e/f90mod_rules.py: buildhooks") + """
109: (0) """ + gentitle("See f2py2e/rules.py: module_rules['modulebody']") + """
110: (0) """ + gentitle("See f2py2e/common_rules.py: buildhooks") + """
111: (0) """ + gentitle("See f2py2e/rules.py") + """
112: (0) static FortranDataDef f2py_routine_defs[] = {
113: (4)     {NULL}
114: (0) };
115: (0) static PyMethodDef f2py_module_methods[] = {
116: (4)     {NULL,NULL}
117: (0) };
118: (0) static struct PyModuleDef moduledef = {
119: (4)     PyModuleDef_HEAD_INIT,
120: (4)     "#modulename#",
121: (4)     NULL,
122: (4)     -1,
123: (4)     f2py_module_methods,
124: (4)     NULL,
125: (4)     NULL,
126: (4)     NULL,
127: (4)     NULL
128: (0) };
129: (0) PyMODINIT_FUNC PyInit_#modulename#(void) {
130: (4)     int i;
131: (4)     PyObject *m,*d, *s, *tmp;
132: (4)     m = #modulename#_module = PyModule_Create(&moduledef);
133: (4)     Py_SET_TYPE(&PyFortran_Type, &PyType_Type);
134: (4)     import_array();
135: (4)     if (PyErr_Occurred())
136: (8)         {PyErr_SetString(PyExc_ImportError, \"can't initialize module
#modulename# (failed to import numpy)\"); return m;}
137: (4)     d = PyModule_GetDict(m);
138: (4)     s = PyUnicode_FromString(\"#f2py_version#\");
139: (4)     PyDict_SetItemString(d, \"__version__\", s);
140: (4)     Py_DECREF(s);
141: (4)     s = PyUnicode_FromString(
142: (8)         \"This module '#modulename#' is auto-generated with f2py
(version:#f2py_version#).\nFunctions:\\n\\n#docs#\\n\");
143: (4)     PyDict_SetItemString(d, \"__doc__\", s);
144: (4)     Py_DECREF(s);
145: (4)     s = PyUnicode_FromString(\""" + numpy_version + ""\"\");
146: (4)     PyDict_SetItemString(d, \"__f2py_numpy_version__\", s);
147: (4)     Py_DECREF(s);
148: (4)     /*
149: (5)         * Store the error object inside the dict, so that it could get

```

```

deallocated.

150: (5)          * (in practice, this is a module, so it likely will not and cannot.)
151: (5)          */
152: (4)          PyDict_SetItemString(d, \"_#modulename#_error\", #modulename#_error);
153: (4)          Py_DECREF(#modulename#_error);
154: (4)          for(i=0;f2py_routine_defs[i].name!=NULL;i++) {
155: (8)              tmp = PyFortranObject_NewAsAttr(&f2py_routine_defs[i]);
156: (8)              PyDict_SetItemString(d, f2py_routine_defs[i].name, tmp);
157: (8)              Py_DECREF(tmp);
158: (4)
159: (4)          if (! PyErr_Occurred())
160: (8)              on_exit(f2py_report_on_exit,(void*)\"#modulename#\");
161: (4)          return m;
162: (0)
163: (0)
164: (0)
165: (4)          """
166: (22)          'separatorsfor': {'latexdoc': '\n\n',
167: (4)                  'restdoc': '\n\n'},
168: (17)          'latexdoc': ['\\section{Module \\texttt{#texmodulename#}}\n',
169: (17)                  '#modnote#\n',
170: (4)                  '#latexdoc#'],
171: (16)          'restdoc': ['Module #modulename#\n' + '=' * 80,
172: (0)                  '\n#restdoc#']
173: (0)
174: (4)          defmod_rules = [
175: (5)              {'body': '/*eof body*/',
176: (5)                  'method': '/*eof method*/',
177: (5)                  'externroutines': '/*eof externroutines*/',
178: (5)                  'routine_defs': '/*eof routine_defs*/',
179: (5)                  'initf90modhooks': '/*eof initf90modhooks*/',
180: (5)                  'initf2pyraphooks': '/*eof initf2pyraphooks*/',
181: (5)                  'initcommonhooks': '/*eof initcommonhooks*/',
182: (5)                  'latexdoc': '',
183: (5)                  'restdoc': '',
184: (5)                  'modnote': {hasnote: '#note#', l_not(hasnote): ''},
185: (0)
186: (0)
187: (4)          routine_rules = {
188: (4)              'separatorsfor': sepdict,
189: (0)              'body': """
190: (0)              static char doc_#apiname#[] = \"\\n#docreturn##name#
191: (#docsignatureshort#)\\n\\nWrapper for ``#name#``.\\n\\n#docstrsigns#\";
192: (0)              /* #declfortranroutine# */
193: (0)              static PyObject *#apiname#(const PyObject *capi_self,
194: (27)                  PyObject *capi_args,
195: (27)                  PyObject *capi_keywds,
196: (4)                  PyObject *volatile capi_buildvalue = NULL;
197: (4)                  volatile int f2py_success = 1;
198: (4)                  static char *capi_kwlist[] = {#kwlist##kwlistopt##kwlistxa#NULL};
199: (0)                  f2py_start_clock();
200: (4)                  if (!PyArg_ParseTupleAndKeywords(capi_args,capi_keywds,\ \
201: (8)                      \"#argformat#|#keyformat##xaformat#:#pyname#\",\ \
202: (8)                      capi_kwlist#args_capi##keys_capi##keys_xa#))\n                          return NULL;
203: (0)                  /*end of frompyobj*/
204: (0)                  f2py_start_call_clock();
205: (0)                  if (PyErr_Occurred())
206: (0)                      f2py_success = 0;
207: (0)                      f2py_stop_call_clock();
208: (0)                      /*end of callfortranroutine*/
209: (8)                      if (f2py_success) {
210: (8)                          CFUNCSMESS(\"Building return value.\\n\");
211: (0)                          capi_buildvalue = Py_BuildValue(\"#returnformat#\"#return#");
212: (8)                          /*closepyobjfrom*/
213: (0)                          } /*if (f2py_success) after callfortranroutine*/
214: (4)                          /*cleanupfrompyobj*/
215: (4)                          if (capi_buildvalue == NULL) {
216: (4)                              }

```

```

217: (4) CFUNCSMESS(\"Freeing memory.\n\");
218: (0) f2py_stop_clock();
219: (4)     return capi_buildvalue;
220: (0)
221: (0)
222: (4) """
223: (4)     'routine_defs': '#routine_def#',
224: (4)     'initf2pyraphooks': '#initf2pyraphook#',
225: (4)     'externroutines': '#declfortranroutine#',
226: (4)     'doc': '#doctreturn##name##docssignature#',
227: (4)     'docshort': '#doctreturn##name##docssignatureshort#',
228: (4)     'docs': '\"' #doctreturn##name##docssignature#\n\"\\n',
229: (4)     'need': ['arrayobject.h', 'CFUNCSMESS', 'MINMAX'],
230: (4)     'cppmacros': {debugcapi: '#define DEBUGCFUNCS'},
231: (4)     'latexdoc': ['\\subsection{Wrapper function \\texttt{#texname#}}\\n',
232: (17)         """]
233: (0)     '\\noindent{}\\verb@#doctreturn##name#@{}\\texttt{(#latextdocsignatureshort#)}}
234: (0) """
235: (4)     'restdoc': ['Wrapped function ``#name#``\\n' + '-' * 80,
236: (16)         ]
237: (0)
238: (4) rout_rules = [
239: (8)     { # Init
240: (26)         'separatorsfor': {'callfortranroutine': '\\n', 'routdebugenter': '\\n',
241: (26)             'routdebugleave': '\\n', 'routdebugfailure': '\\n',
242: (26)             'setjmpbuf': ' || ',
243: (26)             'docstrreq': '\\n', 'docstropt': '\\n', 'docstrout':
244: (26)                 '\\n',
245: (26)                 'docstrcbs': '\\n', 'docstrsigns': '\\n\\n\\n',
246: (26)                 'latextdocstrsigns': '\\n',
247: (26)                 'latextdocstrreq': '\\n', 'latextdocstropt': '\\n',
248: (8)                 'latextdocstrout': '\\n', 'latextdocstrcbs': '\\n',
249: (8)             },
250: (8)         'kwlist': '', 'kwlistopt': '', 'callfortran': '', 'callfortranappend':
251: (8)             '',
252: (8)             'docsign': '', 'docsignopt': '', 'decl': '/*decl*/',
253: (8)             'freemem': '/*freemem*/',
254: (8)             'docsignshort': '', 'docsignoptshort': '',
255: (8)             'docstrsigns': '', 'latextdocstrsigns': '',
256: (8)             'docstrreq': '\\nParameters\\n-----',
257: (8)             'docstropt': '\\nOther Parameters\\n-----',
258: (8)             'docstrout': '\\nReturns\\n-----',
259: (8)             'docstrcbs': '\\nNotes\\n----\\nCall-back functions:\\n',
260: (8)             'latextdocstrreq': '\\noindent Required arguments:',
261: (8)             'latextdocstropt': '\\noindent Optional arguments:',
262: (8)             'latextdocstrout': '\\noindent Return objects:',
263: (8)             'latextdocstrcbs': '\\noindent Call-back functions:',
264: (8)             'args_capi': '', 'keys_capi': '', 'functype': '',
265: (8)             'frompyobj': '/*frompyobj*/',
266: (8)             'cleanupfrompyobj': ['/end of cleanupfrompyobj*'],
267: (8)             'pyobjfrom': '/*pyobjfrom*/',
268: (8)             'closepyobjfrom': ['/end of closepyobjfrom*'],
269: (8)             'topyarr': '/*topyarr*/', 'routdebugleave': '/*routdebugleave*/',
270: (8)             'routdebugenter': '/*routdebugenter*/',
271: (8)             'routdebugfailure': '/*routdebugfailure*/',
272: (8)             'callfortranroutine': '/*callfortranroutine*/',
273: (8)             'argformat': '', 'keyformat': '', 'need_cfuncs': '',
274: (8)             'doctreturn': '', 'return': '', 'returnformat': '',
275: (4)             'routnote': {hasnote: '--- #note#', l_not(hasnote): ''},
276: (8)         },
277: (8)         'apiname': 'f2py_rout_#modulename#_#name#',
278: (8)         'pyname': '#modulename#.#name#',
279: (8)         'decl': '',
280: (4)         '_check': l_not(ismoduleroutine)
281: (8)     },
281: (8)         'apiname': 'f2py_rout_#modulename#_#f90modulename#_#name#',

```

```

282: (8)                      'pyname': '#modulename#.#f90modulename#.#name#',
283: (8)                      'decl': '',
284: (8)                      '_check': ismoduleroutine
285: (4)  }, { # Subroutine
286: (8)                      'functype': 'void',
287: (8)                      'declfortranroutine': {l_and(l_not(l_or(ismoduleroutine, isintent_c)),
1_not(isdummyroutine)): 'extern void #F_FUNC#(#fortranname#,#FORTRANNAME#)(#callprotoargument#);',
288: (31)                                l_and(l_not(ismoduleroutine), isintent_c,
1_not(isdummyroutine)): 'extern void #fortranname#(#callprotoargument#);',
289: (31)                                ismoduleroutine: '',
290: (31)                                isdummyroutine: ''
291: (31) },
292: (8)                      'routine_def': {
293: (12)                        l_not(l_or(ismoduleroutine, isintent_c, isdummyroutine)):
294: (12)                          {\'#name#\',-1,{\{-1\}},0,0,(char *)'
295: (12)                          '#F_FUNC#(#fortranname#,#FORTRANNAME#),'
296: (12)                          '(f2py_init_func)#apiname#,doc_#apiname#},',
297: (12)                        l_and(l_not(ismoduleroutine), isintent_c, l_not(isdummyroutine)):
298: (12)                          {\'#name#\',-1,{\{-1\}},0,0,(char *)#fortranname#,
299: (12)                          '(f2py_init_func)#apiname#,doc_#apiname#},',
300: (12)                        l_and(l_not(ismoduleroutine), isdummyroutine):
301: (12)                          {\'#name#\',-1,{\{-1\}},0,0,NULL,
302: (12)                          '(f2py_init_func)#apiname#,doc_#apiname#},',
303: (8) },
304: (8)                      'need': {l_and(l_not(l_or(ismoduleroutine, isintent_c)),
1_not(isdummyroutine)): 'F_FUNC'},
305: (8)                      'callfortranroutine': [
306: (12)                        {debugcapi: [
307: (16)                          """ fprintf(stderr,\"debug-capi:Fortran subroutine
`#fortranname#(#callfortran#)\\"\\n\\");"""
308: (12)                          {hasexternals: """
309: (8)                            if (#setjmpbuf#) {
310: (12)                              f2py_success = 0;
311: (8)                            } else """",
312: (12)                            {isthreadsafe: ' Py_BEGIN_ALLOW_THREADS',
313: (12)                            {haskellstatement: ' Py_callstatement#;
314: (16)                              /*(*f2py_func)(#callfortran#);*/""",
315: (12)                              l_not(l_or(haskellstatement, isdummyroutine))
316: (19)                                : ' (*f2py_func)(#callfortran#);',
317: (12)                                {isthreadsafe: ' Py_END_ALLOW_THREADS',
318: (12)                                {hasexternals: """
319: (8) ],
320: (8)                      '_check': l_and(issubroutine, l_not(issubroutine_wrap)),
321: (4)  }, { # Wrapped function
322: (8)                      'functype': 'void',
323: (8)                      'declfortranroutine': {l_not(l_or(ismoduleroutine, isdummyroutine)):
'extern void #F_WRAPPEDFUNC#(#name_lower#,#NAME#)(#callprotoargument#);',
324: (31)                                isdummyroutine: '',
325: (31) },
326: (8)                      'routine_def': {
327: (12)                        l_not(l_or(ismoduleroutine, isdummyroutine)):
328: (12)                          {\'#name#\',-1,{\{-1\}},0,0,(char *)'
329: (12)                          '#F_WRAPPEDFUNC#(#name_lower#,#NAME#),
330: (12)                          '(f2py_init_func)#apiname#,doc_#apiname#},',
331: (12)                        isdummyroutine:
332: (12)                          {\'#name#\',-1,{\{-1\}},0,0,NULL,
333: (12)                          '(f2py_init_func)#apiname#,doc_#apiname#},',
334: (8) },
335: (8)                      'initf2pyraphook': {l_not(l_or(ismoduleroutine, isdummyroutine)): '''
336: (4)  {
337: (6)                            extern #ctype# #F_FUNC#(#name_lower#,#NAME#)(void);
338: (6)                            PyObject* o = PyDict_GetItemString(d,"#name#");
339: (6)                            tmp = F2PyCapsule_FromVoidPtr((void*)#F_FUNC#
(#name_lower#,#NAME#),NULL);
340: (6)                            PyObject_SetAttrString(o,"_cpointer", tmp);
341: (6)                            Py_DECREF(tmp);
342: (6)                            s = PyUnicode_FromString("#name#");
343: (6)                            PyObject_SetAttrString(o,"__name__", s);
344: (6)                            Py_DECREF(s);

```

```

345: (4)
346: (4)
347: (8)           'need': {l_not(l_or(ismoduleroutine, isdummyroutine))}:
['F_WRAPEDFUNC', 'F_FUNC']],
348: (8)           'callfortranroutine': [
349: (12)             {debugcapi: [
350: (16)               """ fprintf(stderr,\"debug-capi:Fortran subroutine
`f2pywrap#name_lower#(#callfortran#)\\"\n\";"""]},
351: (12)               {hasexternals: """
352: (4)                 if (#setjmpbuf#) {
353: (8)                   f2py_success = 0;
354: (4)                 } else {""", [
355: (12)                   {isthreadsafe: '    Py_BEGIN_ALLOW_THREADS'},
356: (12)                   {l_not(l_or(hascalldstatement, isdummyroutine))
357: (19)                     : '    (*f2py_func)(#callfortran#);'},
358: (12)                     {hascalldstatement:
359: (16)                         '#callstatement#\n    /*(*f2py_func)(#callfortran#);*/',
360: (12)                         {isthreadsafe: '    Py_END_ALLOW_THREADS'},
361: (12)                         {hasexternals: '    }'}
362: (8)                     ],
363: (8)                     '_check': isfunction_wrap,
364: (4)                 }, { # Wrapped subroutine
365: (8)                   'functype': 'void',
366: (8)                   'declfortranroutine': {l_not(l_or(ismoduleroutine, isdummyroutine))}:
'extern void #F_WRAPEDFUNC#(#name_lower#,#NAME#)(#callprotoargument#);',
367: (31)                         isdummyroutine: '',
368: (31)                         },
369: (8)                   'routine_def': {
370: (12)                     l_not(l_or(ismoduleroutine, isdummyroutine)):
371: (12)                     '    {"#name#", -1, {-1}, 0, 0, (char *)'
372: (12)                     '    #F_WRAPEDFUNC#(#name_lower#,#NAME#), '
373: (12)                     '    (f2py_init_func)#apiname#, doc_#apiname#}, ',
374: (12)                     isdummyroutine:
375: (12)                     '    {"#name#", -1, {-1}, 0, 0, NULL,
376: (12)                     '    (f2py_init_func)#apiname#, doc_#apiname#}, ',
377: (8)                     },
378: (8)                     'initf2pyraphook': {l_not(l_or(ismoduleroutine, isdummyroutine)): ''}
379: (4)                 {
380: (6)                   extern void #F_FUNC#(#name_lower#,#NAME#)(void);
381: (6)                   PyObject* o = PyDict_GetItemString(d, "#name#");
382: (6)                   tmp = F2PyCapsule_FromVoidPtr((void*)#F_FUNC#
(#name_lower#,#NAME#),NULL);
383: (6)                   PyObject_SetAttrString(o, "_cpointer", tmp);
384: (6)                   Py_DECREF(tmp);
385: (6)                   s = PyUnicode_FromString("#name#");
386: (6)                   PyObject_SetAttrString(o, "__name__", s);
387: (6)                   Py_DECREF(s);
388: (4)                 },
389: (4)                 ''},
390: (8)           'need': {l_not(l_or(ismoduleroutine, isdummyroutine))}:
['F_WRAPEDFUNC', 'F_FUNC']],
391: (8)           'callfortranroutine': [
392: (12)             {debugcapi: [
393: (16)               """ fprintf(stderr,\"debug-capi:Fortran subroutine
`f2pywrap#name_lower#(#callfortran#)\\"\n\";"""]},
394: (12)               {hasexternals: """
395: (4)                 if (#setjmpbuf#) {
396: (8)                   f2py_success = 0;
397: (4)                 } else {""", [
398: (12)                   {isthreadsafe: '    Py_BEGIN_ALLOW_THREADS'},
399: (12)                   {l_not(l_or(hascalldstatement, isdummyroutine))
400: (19)                     : '    (*f2py_func)(#callfortran#);'},
401: (12)                     {hascalldstatement:
402: (16)                         '#callstatement#\n    /*(*f2py_func)(#callfortran#);*/',
403: (12)                         {isthreadsafe: '    Py_END_ALLOW_THREADS'},
404: (12)                         {hasexternals: '    }'}
405: (8)                     ],
406: (8)                     '_check': issubroutine_wrap,
407: (4)                 }, { # Function

```

```

408: (8)             'functype': '#ctype#',
409: (8)             'docreturn': {l_not(isintent_hide): '#rname#,'},
410: (8)             'docstrout': '#pydocsigntout#',
411: (8)             'latexdocstrout': ['\\item[]{}\\verb@#pydocsigntout#@{}'],
412: (27)             {hasresultnote: '--- #resultnote#'}],
413: (8)             'callfortranroutine': [{l_and(debugcapi, isstringfunction): """\
414: (4)                 fprintf(stderr,\"debug-capi:Fortran function #ctype# #fortranname#
415: (#callcompaqfortran#)\\n\");
416: (4)                 fprintf(stderr,\"debug-capi:Fortran function #ctype# #fortranname#
417: (#callfortran#)\\n\");
418: (0)                 """},
419: (31)             {l_and(debugcapi, l_not(isstringfunction))}: """
420: (4)                 fprintf(stderr,\"debug-capi:Fortran function #ctype# #fortranname#
421: (#callfortran#)\\n\");
422: (0)                 """}
423: (31)             ],
424: (8)             '_check': l_and(isfunction, l_not(isfunction_wrap))
425: (4)             }, { # Scalar function
426: (8)             'declfortranroutine': {l_and(l_not(l_or(ismoduleroutine, isintent_c)),
427: l_not(isdummyroutine)): 'extern #ctype# #F_FUNC#(#fortranname#,#FORTRANNAME#
428: (#callprotoargument#);',
429: (31)             l_and(l_not(ismoduleroutine), isintent_c,
430: l_not(isdummyroutine)): 'extern #ctype# #fortranname#(#callprotoargument#);',
431: (31)             isdummyroutine: ''
432: (31)             },
433: (8)             'routine_def': {
434: (12)             l_and(l_not(l_or(ismoduleroutine, isintent_c)),
435: l_not(isdummyroutine)):
436: ('    {"#name#", -1, {{-1}}, 0, 0, (char *)'
437:     '#F_FUNC#(#fortranname#,#FORTRANNAME#), '
438:     '(f2py_init_func)#apiname#,doc_#apiname#},'),
439: (12)             l_and(l_not(ismoduleroutine), isintent_c, l_not(isdummyroutine)):
440: ('    {"#name#", -1, {{-1}}, 0, 0, (char *)#fortranname#,
441:     '(f2py_init_func)#apiname#,doc_#apiname#},',
442: (12)             isdummyroutine:
443: ('        {"#name#", -1, {{-1}}, 0, 0, NULL,
444:         '(f2py_init_func)#apiname#,doc_#apiname#},',
445: (8)             'decl': [{iscomplexfunction_warn: '      #ctype# #name#_return_value=
446: {0,0};',
447: (18)             l_not(iscomplexfunction): '      #ctype#
448: #name#_return_value=0;'},
449: (17)             {iscomplexfunction:
450: ('          PyObject *#name#_return_value_capi = Py_None;');
451: (12)             }],
452: (8)             'callfortranroutine': [
453: (12)             {hasexternals: """\
454: if (#setjmpbuf#) {
455: f2py_success = 0;
456: } else {"""},
457: (12)             {isthreadsafe: '      Py_BEGIN_ALLOW_THREADS'},
458: (12)             {hascallstatement: '      #callstatement#';
459: (0)             /*      #name#_return_value = (*f2py_func)(#callfortran#);*/
460: (0)             ''',},
461: (12)             {l_not(l_or(hascalldstatement, isdummyroutine))
462: (19)             : '      #name#_return_value = (*f2py_func)
463: (#callfortran#);'},
464: (12)             {isthreadsafe: '      Py_END_ALLOW_THREADS'},
465: (12)             {hasexternals: '      }'},
466: (12)             {l_and(debugcapi, iscomplexfunction)
467: (19)             :
468: fprintf(stderr,"#routdebugshowvalue#\n",#name#_return_value.r,#name#_return_value.i);'},
469: (12)             {l_and(debugcapi, l_not(iscomplexfunction)): '
470: fprintf(stderr,"#routdebugshowvalue#\n",#name#_return_value);'},
471: (8)             'pyobjfrom': {iscomplexfunction: '      #name#_return_value_capi =
472: pyobj_from_#ctype#1(#name#_return_value);'},
473: (8)             'need': [{l_not(isdummyroutine): 'F_FUNC'},
474: (17)             {iscomplexfunction: 'pyobj_from_#ctype#1'}},

```

```

464: (17)                      {islong_longfunction: 'long_long'},
465: (17)                      {islong_doublefunction: 'long_double'}],
466: (8)                      'returnformat': {l_not(isintent_hide): '#rformat#'},
467: (8)                      'return': {iscomplexfunction: ',#name#_return_value_capi',
468: (19)                          l_not(l_or(iscomplexfunction, isintent_hide)):'#name#_return_value'},
469: (8)                      '_check': l_and(isfunction, l_not(isstringfunction),
l_not(isfunction_wrap))
470: (4)                      }, { # String function # in use for --no-wrap
471: (8)                          'declfortranroutine': 'extern void #F_FUNC#
(#fortranname#,#FORTRANNAME#)(#callprotoargument#);',
472: (8)                          'routine_def': {l_not(l_or(ismoduleroutine, isintent_c)):'#name#\',-1,{-1},0,0,(char *)#F_FUNC#
(#fortranname#,#FORTRANNAME#),(f2py_init_func)#apiname#,doc_#apiname#,',
473: (24)                          l_and(l_not(ismoduleroutine), isintent_c):'#name#\',-1,{-1},0,0,(char *)#fortranname#,
474: (24)                          (f2py_init_func)#apiname#,doc_#apiname#,',
475: (24)                          },
476: (24)                      },
477: (8)                      'decl': ['#ctype# #name#_return_value = NULL;',
478: (17)                          'int #name#_return_value_len = 0;'],
479: (8)                      'callfortran': '#name#_return_value,#name#_return_value_len',
480: (8)                      'callfortranroutine': ['#name#_return_value_len = #rlength#;',
481: (30)                          'if ((#name#_return_value = (string)malloc('
482: (30)                              '+ #name#_return_value_len+1) == NULL) {',
483: (30)                              'PyErr_SetString(PyExc_MemoryError,
\"out of memory\");',
484: (30)                          'f2py_success = 0;',
485: (30)                          '} else {',
486: (30)                              '#name#_return_value'
487: (30)                          },
488: (30)                          'if (f2py_success) {',
489: (30)                      '{hasexternals: """\
490: (8)                          if (#setjmpbuf#) {
491: (12)                              f2py_success = 0;
492: (8)                          } else {""",',
493: (30)                      {isthreadsafe: '
494: (30)                          """\'
495: (8)                      (*f2py_func)(#callcompaqfortran#);
496: (8)                      (*f2py_func)(#callfortran#);
497: (0)                      """',
498: (30)                      {isthreadsafe: 'Py_END_ALLOW_THREADS'},
499: (30)                      {hasexternals: '}'},
500: (30)                      {debugcapi:
501: (34)
fprintf(stderr,"#routdebugshowvalue#\n",#name#_return_value_len,#name#_return_value);'},
502: (30)                      '/* if (f2py_success) after (string)malloc
*/',
503: (30)                      ],
504: (8)                      'returnformat': '#rformat#',
505: (8)                      'return': ',#name#_return_value',
506: (8)                      'freemem': 'STRINGFREE(#name#_return_value);',
507: (8)                      'need': ['F_FUNC', '#ctype#', 'STRINGFREE'],
508: (8)                      '_check':l_and(isstringfunction, l_not(isfunction_wrap)) # ????
obsolete
509: (4)                      },
510: (4)                      { # Debugging
511: (8)                          'routdebugenter': 'fprintf(stderr,"debug-capi:Python C/API
function #modulename#.#name#(#docssignature#)\n");',
512: (8)                          'routdebugleave': 'fprintf(stderr,"debug-capi:Python C/API
function #modulename#.#name#: successful.\n");',
513: (8)                          'routdebugfailure': 'fprintf(stderr,"debug-capi:Python C/API
function #modulename#.#name#: failure.\n");',
514: (8)                          '_check': debugcapi
515: (4)                      }
516: (0)                      ]
517: (0)                      typedef_need_dict = {islong_long: 'long_long',
518: (21)                          islong_double: 'long_double',

```

```

519: (21)           islong_complex: 'complex_long_double',
520: (21)           isunsigned_char: 'unsigned_char',
521: (21)           isunsigned_short: 'unsigned_short',
522: (21)           isunsigned: 'unsigned',
523: (21)           isunsigned_long_long: 'unsigned_long_long',
524: (21)           isunsigned_chararray: 'unsigned_char',
525: (21)           isunsigned_shortarray: 'unsigned_short',
526: (21)           isunsigned_long_longarray: 'unsigned_long_long',
527: (21)           issigned_long_longarray: 'long_long',
528: (21)           isint1: 'signed_char',
529: (21)           ischaracter_or_characterarray: 'character',
530: (21)           }
531: (0)           aux_rules = [
532: (4)           {
533: (8)             'separatorsfor': sepdict
534: (4)           },
535: (4)           { # Common
536: (8)             'frompyobj': ['/* Processing auxiliary variable #varname# */',
537: (22)               {debugcapi: 'fprintf(stderr,#vardebuginfo#\n");'},
538: (8)             ],
539: (8)             'cleanupfrompyobj': '/* End of cleaning variable #varname# */',
540: (4)             'need': typedef_need_dict,
541: (4)           },
542: (4)           { # Common
543: (8)             'decl': '#ctype# #varname# = 0;',
544: (8)             'need': {hasinitvalue: 'math.h'},
545: (8)             'frompyobj': {hasinitvalue: '#varname# = #init#;'},
546: (4)             '_check': l_and(isscalar, l_not(iscomplex)),
547: (4)           },
548: (8)             'return': '#varname#',
549: (8)             'docstrout': '#pydocsignout#',
550: (8)             'doctype': '#outvarname#',
551: (8)             'returnformat': '#varrformat#',
552: (8)             '_check': l_and(isscalar, l_not(iscomplex), isintent_out),
553: (4)           },
554: (4)           { # Common
555: (8)             'decl': '#ctype# #varname#;',
556: (8)             'frompyobj': {hasinitvalue: '#varname#.r = #init.r#; #varname#.i = #init.i#;'},
557: (8)             '_check': iscomplex
558: (4)           },
559: (4)           { # Common
560: (8)             'decl': ['#ctype# #varname# = NULL;',
561: (17)               'int slen(#varname#);',
562: (17)               ],
563: (8)             'need': ['len..'],
564: (8)             '_check': isstring
565: (4)           },
566: (4)           { # Common
567: (8)             'decl': ['#ctype# *#varname# = NULL;',
568: (17)               'npy_intp #varname#_Dims[#rank#] = {#rank*[-1]#};',
569: (17)               'const int #varname#_Rank = #rank#;'],
570: (17)             'need': ['len..', {hasinitvalue: 'forcomb'}, {hasinitvalue:
571: (8)               'CFUNCSMESS'}],
572: (8)             '_check': isarray
573: (4)           },
574: (4)           { # Common
575: (8)             '_check': l_and(isarray, l_not(iscomplexarray))
576: (4)           },
577: (4)           { # Not hidden
578: (4)             '_check': l_and(isarray, l_not(iscomplexarray), isintent_nothide)
579: (4)           },
580: (5)           {'need': '#ctype#',
581: (5)             '_check': isint1array,
582: (5)             '_depend': ''}
583: (4)           {'need': '#ctype#',
584: (5)             '_check': l_or(isunsigned_chararray, isunsigned_char),

```

```

585: (5)           '_depend': '',
586: (5)           },
587: (4)           {'need': '#ctype#',
588: (5)           '_check': isunsigned_shortarray,
589: (5)           '_depend': ''
590: (5)           },
591: (4)           {'need': '#ctype#',
592: (5)           '_check': isunsigned_long_longarray,
593: (5)           '_depend': ''
594: (5)           },
595: (4)           {'need': '#ctype#',
596: (5)           '_check': iscomplexarray,
597: (5)           '_depend': ''
598: (5)           },
599: (4)           {
600: (8)             'callfortranappend': {isarrayofstrings: 'flen(#varname#),'},
601: (8)             'need': 'string',
602: (8)             '_check': issstringarray
603: (4)           }
604: (0)         ]
605: (0)     arg_rules = [
606: (4)       {
607: (8)         'separatorsfor': sepdict
608: (4)       },
609: (4)       { # Common
610: (8)         'frompyobj': ['/* Processing variable #varname# */',
611: (22)           {debugcapi: 'fprintf(stderr, "#vardebuginfo#\n");'},
612: (8)           'cleanupfrompyobj': '/* End of cleaning variable #varname# */',
613: (8)           '_depend': '',
614: (8)           'need': typeid_need_dict,
615: (4)       },
616: (4)       {
617: (8)         'docstropt': {l_and(isoptional, isintent_nothide): '#pydocsign#'},
618: (8)         'docstrreq': {l_and(isrequired, isintent_nothide): '#pydocsign#'},
619: (8)         'docstrout': {isintent_out: '#pydocsignout#'},
620: (8)         'latexdocstropt': {l_and(isoptional, isintent_nothide): ['\\item[]',
621: (65)           {'{}\\verb@#pydocsign#@{}'}],                                     {hasnote: '--'
622: (8)           '- #note#'}]},  'latextoc': {l_and(isoptional, isintent_nothide): ['\\item[]',
623: (65)           {'{}\\verb@#pydocsign#@{}'}],                                     {hasnote: '--'
624: (8)           '- #note#'}]},  'latextocreq': {l_and(isrequired, isintent_nothide): ['\\item[]',
625: (42)           '#note#',   'latextocreq': {l_and(isrequired, isintent_nothide): ['\\item[]',
626: (43)           '--- See above.'}],},   'hasnote: '---'
627: (8)           'depend': ''
628: (4)         },
629: (4)         {
630: (8)           'kwlist': '"#varname#",',
631: (8)           'docsign': '#varname#,',
632: (8)           '_check': l_and(isintent_nothide, l_not(isoptional))
633: (4)         },
634: (4)         {
635: (8)           'kwlistopt': '"#varname#",',
636: (8)           'docsignopt': '#varname##showinit#,',
637: (8)           'docsignoptshort': '#varname#,',
638: (8)           '_check': l_and(isintent_nothide, isoptional)
639: (4)         },
640: (4)         {
641: (8)           'dcreturn': '#outvarname#,',
642: (8)           'returnformat': '#varrformat#',
643: (8)           '_check': isintent_out
644: (4)         },
645: (4)         { # Common

```

```

646: (8)             'docsignxa': {isintent_nothide: '#varname#_extra_args=(),'},
647: (8)             'docsignxashort': {isintent_nothide: '#varname#_extra_args,'},
648: (8)             'docstropt': {isintent_nothide: '#varname#_extra_args : input tuple,
optional\n      Default: ()'},
649: (8)             'docstrcbs': '#cbdocstr#',
650: (8)             'latexdocstrcbs': '\\item[] #cblatexdocstr#',
651: (8)             'latexdocstropt': {isintent_nothide: '\\item[]'
{{}}\\verb@#varname#_extra_args := () input tuple@{{}} --- Extra arguments for call-back function
{{}}\\verb@#varname#@{{}}.'],
652: (8)             'decl': ['      #cbname#_t #varname#_cb = { Py_None, NULL, 0 };',
653: (17)             '      #cbname#_t *#varname#_cb_ptr = &#varname#_cb;',
654: (17)             '      PyTupleObject *#varname#_xa_capi = NULL;',
655: (17)             '{l_not(isintent_callback):
656: (18)             '      #cbname#_typedef #varname#_cptr;'}
657: (17)             ],
658: (8)             'kwlistxa': {isintent_nothide: '"#varname#_extra_args",'},
659: (8)             'argformat': {isrequired: '0'},
660: (8)             'keyformat': {isoptional: '0'},
661: (8)             'xaformat': {isintent_nothide: '0!'},
662: (8)             'args_capi': {isrequired: ',#varname#_cb.capi'},
663: (8)             'keys_capi': {isoptional: ',#varname#_cb.capi'},
664: (8)             'keys_xa': ',&PyTuple_Type,&#varname#_xa_capi',
665: (8)             'setjmpbuf': '(setjmp(#varname#_cb.jmpbuf))',
666: (8)             'callfortran': {l_not(isintent_callback): '#varname#_cptr,'},
667: (8)             'need': ['#cbname#', 'setjmp.h'],
668: (8)             '_check':isexternal
669: (4)
670: (4)
671: (8)             {
672: (0)             'frompyobj': [{l_not(isintent_callback): """\
if(F2PyCapsule_Check(#varname#_cb.capi)) {
673: (0)             }
674: (0)             }
675: (0)             """}, {isintent_callback: """\
if (#varname#_cb.capi==Py_None) {
676: (0)             if (#varname#_cb.capi) {
677: (2)               if (#varname#_xa_capi==NULL) {
678: (4)                 if
(PyObject_HasAttrString(#modulename#_module,\">#varname#_extra_args\"))
679: (6)
680: (8)                 PyObject* capi_tmp =
PyObject_GetAttrString(#modulename#_module,\">#varname#_extra_args\");
681: (8)                 if (capi_tmp) {
682: (10)                   Py_DECREF(capi_tmp);
683: (8)                 }
684: (8)                 else {
685: (8)                   }
686: (8)                   if (#varname#_xa_capi==NULL) {
687: (10)                     PyErr_SetString(#modulename#_error,\Failed to convert
#modulename#.#varname#_extra_args to tuple.\n\");
688: (10)                     return NULL;
689: (8)                   }
690: (6)
691: (4)
692: (2)
693: (2)                   if (#varname#_cb.capi==NULL) {
694: (4)                     PyErr_SetString(#modulename#_error,\Callback #varname# not defined (as an
argument or module #modulename# attribute).\n\");
695: (4)                     return NULL;
696: (2)
697: (0)
698: (0)
699: (12)                     """
700: (4)
701: (0)
702: (12)                     if
(create_cb_arglist(#varname#_cb.capi,#varname#_xa_capi,#maxnofargs#,#nofoptargs#,#varname#_cb.nof
args,&#varname#_cb.args_capi,\failed in processing argument list for call-back #varname#."))
703: (8)
704: (8)                     {debugcapi: """
fprintf(stderr,\debug-capi:Assuming %d arguments; at most
#maxnofargs#(-#nofoptargs#) is expected.\n",#varname#_cb.nofargs);
CFUNCSMESSPY(\"for #varname#=\",#varname#_cb.capi);"""
}
705: (8)
706: (8)
707: (8)
708: (8)
709: (8)
710: (8)
711: (8)
712: (8)
713: (8)
714: (8)
715: (8)
716: (8)
717: (8)
718: (8)
719: (8)
720: (8)
721: (8)
722: (8)
723: (8)
724: (8)
725: (8)
726: (8)
727: (8)
728: (8)
729: (8)
730: (8)
731: (8)
732: (8)
733: (8)
734: (8)
735: (8)
736: (8)
737: (8)
738: (8)
739: (8)
740: (8)
741: (8)
742: (8)
743: (8)
744: (8)
745: (8)
746: (8)
747: (8)
748: (8)
749: (8)
750: (8)
751: (8)
752: (8)
753: (8)
754: (8)
755: (8)
756: (8)
757: (8)
758: (8)
759: (8)
760: (8)
761: (8)
762: (8)
763: (8)
764: (8)
765: (8)
766: (8)
767: (8)
768: (8)
769: (8)
770: (8)
771: (8)
772: (8)
773: (8)
774: (8)
775: (8)
776: (8)
777: (8)
778: (8)
779: (8)
780: (8)
781: (8)
782: (8)
783: (8)
784: (8)
785: (8)
786: (8)
787: (8)
788: (8)
789: (8)
790: (8)
791: (8)
792: (8)
793: (8)
794: (8)
795: (8)
796: (8)
797: (8)
798: (8)
799: (8)
800: (8)
801: (8)
802: (8)
803: (8)
804: (8)
805: (8)
806: (8)
807: (8)
808: (8)
809: (8)
810: (8)
811: (8)
812: (8)
813: (8)
814: (8)
815: (8)
816: (8)
817: (8)
818: (8)
819: (8)
820: (8)
821: (8)
822: (8)
823: (8)
824: (8)
825: (8)
826: (8)
827: (8)
828: (8)
829: (8)
830: (8)
831: (8)
832: (8)
833: (8)
834: (8)
835: (8)
836: (8)
837: (8)
838: (8)
839: (8)
840: (8)
841: (8)
842: (8)
843: (8)
844: (8)
845: (8)
846: (8)
847: (8)
848: (8)
849: (8)
850: (8)
851: (8)
852: (8)
853: (8)
854: (8)
855: (8)
856: (8)
857: (8)
858: (8)
859: (8)
860: (8)
861: (8)
862: (8)
863: (8)
864: (8)
865: (8)
866: (8)
867: (8)
868: (8)
869: (8)
870: (8)
871: (8)
872: (8)
873: (8)
874: (8)
875: (8)
876: (8)
877: (8)
878: (8)
879: (8)
880: (8)
881: (8)
882: (8)
883: (8)
884: (8)
885: (8)
886: (8)
887: (8)
888: (8)
889: (8)
890: (8)
891: (8)
892: (8)
893: (8)
894: (8)
895: (8)
896: (8)
897: (8)
898: (8)
899: (8)
900: (8)
901: (8)
902: (8)
903: (8)
904: (8)
905: (8)
906: (8)
907: (8)
908: (8)
909: (8)
910: (8)
911: (8)
912: (8)
913: (8)
914: (8)
915: (8)
916: (8)
917: (8)
918: (8)
919: (8)
920: (8)
921: (8)
922: (8)
923: (8)
924: (8)
925: (8)
926: (8)
927: (8)
928: (8)
929: (8)
930: (8)
931: (8)
932: (8)
933: (8)
934: (8)
935: (8)
936: (8)
937: (8)
938: (8)
939: (8)
940: (8)
941: (8)
942: (8)
943: (8)
944: (8)
945: (8)
946: (8)
947: (8)
948: (8)
949: (8)
950: (8)
951: (8)
952: (8)
953: (8)
954: (8)
955: (8)
956: (8)
957: (8)
958: (8)
959: (8)
960: (8)
961: (8)
962: (8)
963: (8)
964: (8)
965: (8)
966: (8)
967: (8)
968: (8)
969: (8)
970: (8)
971: (8)
972: (8)
973: (8)
974: (8)
975: (8)
976: (8)
977: (8)
978: (8)
979: (8)
980: (8)
981: (8)
982: (8)
983: (8)
984: (8)
985: (8)
986: (8)
987: (8)
988: (8)
989: (8)
990: (8)
991: (8)
992: (8)
993: (8)
994: (8)
995: (8)
996: (8)
997: (8)
998: (8)
999: (8)
1000: (8)
1001: (8)
1002: (8)
1003: (8)
1004: (8)
1005: (8)
1006: (8)
1007: (8)
1008: (8)
1009: (8)
1010: (8)
1011: (8)
1012: (8)
1013: (8)
1014: (8)
1015: (8)
1016: (8)
1017: (8)
1018: (8)
1019: (8)
1020: (8)
1021: (8)
1022: (8)
1023: (8)
1024: (8)
1025: (8)
1026: (8)
1027: (8)
1028: (8)
1029: (8)
1030: (8)
1031: (8)
1032: (8)
1033: (8)
1034: (8)
1035: (8)
1036: (8)
1037: (8)
1038: (8)
1039: (8)
1040: (8)
1041: (8)
1042: (8)
1043: (8)
1044: (8)
1045: (8)
1046: (8)
1047: (8)
1048: (8)
1049: (8)
1050: (8)
1051: (8)
1052: (8)
1053: (8)
1054: (8)
1055: (8)
1056: (8)
1057: (8)
1058: (8)
1059: (8)
1060: (8)
1061: (8)
1062: (8)
1063: (8)
1064: (8)
1065: (8)
1066: (8)
1067: (8)
1068: (8)
1069: (8)
1070: (8)
1071: (8)
1072: (8)
1073: (8)
1074: (8)
1075: (8)
1076: (8)
1077: (8)
1078: (8)
1079: (8)
1080: (8)
1081: (8)
1082: (8)
1083: (8)
1084: (8)
1085: (8)
1086: (8)
1087: (8)
1088: (8)
1089: (8)
1090: (8)
1091: (8)
1092: (8)
1093: (8)
1094: (8)
1095: (8)
1096: (8)
1097: (8)
1098: (8)
1099: (8)
1100: (8)
1101: (8)
1102: (8)
1103: (8)
1104: (8)
1105: (8)
1106: (8)
1107: (8)
1108: (8)
1109: (8)
1110: (8)
1111: (8)
1112: (8)
1113: (8)
1114: (8)
1115: (8)
1116: (8)
1117: (8)
1118: (8)
1119: (8)
1120: (8)
1121: (8)
1122: (8)
1123: (8)
1124: (8)
1125: (8)
1126: (8)
1127: (8)
1128: (8)
1129: (8)
1130: (8)
1131: (8)
1132: (8)
1133: (8)
1134: (8)
1135: (8)
1136: (8)
1137: (8)
1138: (8)
1139: (8)
1140: (8)
1141: (8)
1142: (8)
1143: (8)
1144: (8)
1145: (8)
1146: (8)
1147: (8)
1148: (8)
1149: (8)
1150: (8)
1151: (8)
1152: (8)
1153: (8)
1154: (8)
1155: (8)
1156: (8)
1157: (8)
1158: (8)
1159: (8)
1160: (8)
1161: (8)
1162: (8)
1163: (8)
1164: (8)
1165: (8)
1166: (8)
1167: (8)
1168: (8)
1169: (8)
1170: (8)
1171: (8)
1172: (8)
1173: (8)
1174: (8)
1175: (8)
1176: (8)
1177: (8)
1178: (8)
1179: (8)
1180: (8)
1181: (8)
1182: (8)
1183: (8)
1184: (8)
1185: (8)
1186: (8)
1187: (8)
1188: (8)
1189: (8)
1190: (8)
1191: (8)
1192: (8)
1193: (8)
1194: (8)
1195: (8)
1196: (8)
1197: (8)
1198: (8)
1199: (8)
1200: (8)
1201: (8)
1202: (8)
1203: (8)
1204: (8)
1205: (8)
1206: (8)
1207: (8)
1208: (8)
1209: (8)
1210: (8)
1211: (8)
1212: (8)
1213: (8)
1214: (8)
1215: (8)
1216: (8)
1217: (8)
1218: (8)
1219: (8)
1220: (8)
1221: (8)
1222: (8)
1223: (8)
1224: (8)
1225: (8)
1226: (8)
1227: (8)
1228: (8)
1229: (8)
1230: (8)
1231: (8)
1232: (8)
1233: (8)
1234: (8)
1235: (8)
1236: (8)
1237: (8)
1238: (8)
1239: (8)
1240: (8)
1241: (8)
1242: (8)
1243: (8)
1244: (8)
1245: (8)
1246: (8)
1247: (8)
1248: (8)
1249: (8)
1250: (8)
1251: (8)
1252: (8)
1253: (8)
1254: (8)
1255: (8)
1256: (8)
1257: (8)
1258: (8)
1259: (8)
1260: (8)
1261: (8)
1262: (8)
1263: (8)
1264: (8)
1265: (8)
1266: (8)
1267: (8)
1268: (8)
1269: (8)
1270: (8)
1271: (8)
1272: (8)
1273: (8)
1274: (8)
1275: (8)
1276: (8)
1277: (8)
1278: (8)
1279: (8)
1280: (8)
1281: (8)
1282: (8)
1283: (8)
1284: (8)
1285: (8)
1286: (8)
1287: (8)
1288: (8)
1289: (8)
1290: (8)
1291: (8)
1292: (8)
1293: (8)
1294: (8)
1295: (8)
1296: (8)
1297: (8)
1298: (8)
1299: (8)
1300: (8)
1301: (8)
1302: (8)
1303: (8)
1304: (8)
1305: (8)
1306: (8)
1307: (8)
1308: (8)
1309: (8)
1310: (8)
1311: (8)
1312: (8)
1313: (8)
1314: (8)
1315: (8)
1316: (8)
1317: (8)
1318: (8)
1319: (8)
1320: (8)
1321: (8)
1322: (8)
1323: (8)
1324: (8)
1325: (8)
1326: (8)
1327: (8)
1328: (8)
1329: (8)
1330: (8)
1331: (8)
1332: (8)
1333: (8)
1334: (8)
1335: (8)
1336: (8)
1337: (8)
1338: (8)
1339: (8)
1340: (8)
1341: (8)
1342: (8)
1343: (8)
1344: (8)
1345: (8)
1346: (8)
1347: (8)
1348: (8)
1349: (8)
1350: (8)
1351: (8)
1352: (8)
1353: (8)
1354: (8)
1355: (8)
1356: (8)
1357: (8)
1358: (8)
1359: (8)
1360: (8)
1361: (8)
1362: (8)
1363: (8)
1364: (8)
1365: (8)
1366: (8)
1367: (8)
1368: (8)
1369: (8)
1370: (8)
1371: (8)
1372: (8)
1373: (8)
1374: (8)
1375: (8)
1376: (8)
1377: (8)
1378: (8)
1379: (8)
1380: (8)
1381: (8)
1382: (8)
1383: (8)
1384: (8)
1385: (8)
1386: (8)
1387: (8)
1388: (8)
1389: (8)
1390: (8)
1391: (8)
1392: (8)
1393: (8)
1394: (8)
1395: (8)
1396: (8)
1397: (8)
1398: (8)
1399: (8)
1400: (8)
1401: (8)
1402: (8)
1403: (8)
1404: (8)
1405: (8)
1406: (8)
1407: (8)
1408: (8)
1409: (8)
1410: (8)
1411: (8)
1412: (8)
1413: (8)
1414: (8)
1415: (8)
1416: (8)
1417: (8)
1418: (8)
1419: (8)
1420: (8)
1421: (8)
1422: (8)
1423: (8)
1424: (8)
1425: (8)
1426: (8)
1427: (8)
1428: (8)
1429: (8)
1430: (8)
1431: (8)
1432: (8)
1433: (8)
1434: (8)
1435: (8)
1436: (8)
1437: (8)
1438: (8)
1439: (8)
1440: (8)
1441: (8)
1442: (8)
1443: (8)
1444: (8)
1445: (8)
1446: (8)
1447: (8)
1448: (8)
1449: (8)
1450: (8)
1451: (8)
1452: (8)
1453: (8)
1454: (8)
1455: (8)
1456: (8)
1457: (8)
1458: (8)
1459: (8)
1460: (8)
1461: (8)
1462: (8)
1463: (8)
1464: (8)
1465: (8)
1466: (8)
1467: (8)
1468: (8)
1469: (8)
1470: (8)
1471: (8)
1472: (8)
1473: (8)
1474: (8)
1475: (8)
1476: (8)
1477: (8)
1478: (8)
1479: (8)
1480: (8)
1481: (8)
1482: (8)
1483: (8)
1484: (8)
1485: (8)
1486: (8)
1487: (8)
1488: (8)
1489: (8)
1490: (8)
1491: (8)
1492: (8)
1493: (8)
1494: (8)
1495: (8)
1496: (8)
1497: (8)
1498: (8)
1499: (8)
1500: (8)
1501: (8)
1502: (8)
1503: (8)
1504: (8)
1505: (8)
1506: (8)
1507: (8)
1508: (8)
1509: (8)
1510: (8)
1511: (8)
1512: (8)
1513: (8)
1514: (8)
1515: (8)
1516: (8)
1517: (8)
1518: (8)
1519: (8)
1520: (8)
1521: (8)
1522: (8)
1523: (8)
1524: (8)
1525: (8)
1526: (8)
1527: (8)
1528: (8)
1529: (8)
1530: (8)
1531: (8)
1532: (8)
1533: (8)
1534: (8)
1535: (8)
1536: (8)
1537: (8)
1538: (8)
1539: (8)
1540: (8)
1541: (8)
1542: (8)
1543: (8)
1544: (8)
1545: (8)
1546: (8)
1547: (8)
1548: (8)
1549: (8)
1550: (8)
1551: (8)
1552: (8)
1553: (8)
1554: (8)
1555: (8)
1556: (8)
1557: (8)
1558: (8)
1559: (8)
1560: (8)
1561: (8)
1562: (8)
1563: (8)
1564: (8)
1565: (8)
1566: (8)
1567: (8)
1568: (8)
1569: (8)
1570: (8)
1571: (8)
1572: (8)
1573: (8)
1574: (8)
1575: (8)
1576: (8)
1577: (8)
1578: (8)
1579: (8)
1580: (8)
1581: (8)
1582: (8)
1583: (8)
1584: (8)
1585: (8)
1586: (8)
1587: (8)
1588: (8)
1589: (8)
1590: (8)
1591: (8)
1592: (8)
1593: (8)
1594: (8)
1595: (8)
1596: (8)
1597: (8)
1598: (8)
1599: (8)
1600: (8)
1601: (8)
1602: (8)
1603: (8)
1604: (8)
1605: (8)
1606: (8)
1607: (8)
1608: (8)
1609: (8)
1610: (8)
1611: (8)
1612: (8)
1613: (8)
1614: (8)
1615: (8)
1616: (8)
1617: (8)
1618: (8)
1619: (8)
1620: (8)
1621: (8)
1622: (8)
1623: (8)
1624: (8)
1625: (8)
1626: (8)
1627: (8)
1628: (8)
1629: (8)
1630: (8)
1631: (8)
1632: (8)
1633: (8)
1634: (8)
1635: (8)
1636: (8)
1637: (8)
1638: (8)
1639: (8)
1640: (8)
1641: (8)
1642: (8)
1643: (8)
1644: (8)
1645: (8)
1646: (8)
1647: (8)
1648: (8)
1649: (8)
1650: (8)
1651: (8)
1652: (8)
1653: (8)
1654: (8)
1655: (8)
1656: (8)
1657: (8)
1658: (8)
1659: (8)
1660: (8)
1661: (8)
1662: (8)
1663: (8)
1664: (8)
1665: (8)
1666: (8)
1667: (8)
1668: (8)
1669: (8)
1670: (8)
1671: (8)
1672: (8)
1673: (8)
1674: (8)
1675: (8)
1676: (8)
1677: (8)
1678: (8)
1679: (8)
1680: (8)
1681: (8)
1682: (8)
1683: (8)
1684: (8)
1685: (8)
1686: (8)
1687: (8)
1688: (8)
1689: (8)
1690: (8)
1691: (8)
1692: (8)
1693: (8)
1694: (8)
1695: (8)
1696: (8)
1697: (8)
1698: (8)
1699: (8)
1700: (8)
1701: (8)
1702: (8)
1703: (8)
1704: (8)
1705: (8)
1706: (8)
1707: (8)
1708: (8)
1709: (8)
1710: (8)
1711: (8)
1712: (8)
1713: (8)
1714: (8)
1715: (8)
1716: (8)
1717: (8)
1718: (8)
1719: (8)
1720: (8)
1721: (8)
1722: (8)
1723: (8)
1724: (8)
1725: (8)
1726: (8)
1727: (8)
1728: (8)
1729: (8)
1730: (8)
1731: (8)
1732: (8)
1733: (8)
1734: (8)
1735: (8)
1736: (8)
1737: (8)
1738: (8)
1739: (8)
1740: (8)
1741: (8)
1742: (8)
1743: (8)
1744: (8)
1745: (8)
1746: (8)
1747: (8)
1748: (8)
1749: (8)
1750: (8)
1751: (8)
1752: (8)
1753: (8)
1754: (8)
1755: (8)
1756: (8)
1757: (8)
1758: (8)
1759: (8)
1760: (8)
1761: (8)
1762: (8)
1763: (8)
1764: (8)
1765: (8)
1766: (8)
1767: (8)
1768: (8)
1769: (8)
1770: (8)
1771: (8)
1772: (8)
1773: (8)
1774: (8)
1775: (8)
1776: (8)
1777: (8)
1778: (8)
1779: (8)
1780: (8)
1781: (8)
1782: (8)
1783: (8)
1784: (8)
1785: (8)
1786: (8)
1787: (8)
1788: (8)
1789: (8)
1790: (8)
1791: (8)
1792: (8)
1793: (8)
1794: (8)
1795: (8)
1796: (8)
1797: (8)
1798: (8)
1799: (8)
1800: (8)
1801: (8)
1802: (8)
1803: (8)
1804: (8)
1805: (8)
1806: (8)
1807: (8)
1808: (8)
1809: (8)
1810: (8)
1811: (8)
1812: (8)
1813: (8)
1814: (8)
1815: (8)
1816: (8)
1817: (8)
1818: (8)
1819: (8)
1820: (8)
1821: (8)
1822: (8)
1823: (8)
1824: (8)
1825: (8)
1826: (8)
1827: (8)
1828: (8)
1829: (8)
1830: (8)
1831: (8)
1832: (8)
1833: (8)
1834: (8)
1835: (8)
1836: (8)
1837: (8)
1838: (8)
1839: (8)
1840: (8)
1841: (8)
1842: (8)
1843: (8)
1844: (8)
1845: (8)
1846: (8)
1847: (8)
1848: (8)
1849: (8)
1850: (8)
1851: (8)
1852: (8)
1853: (8)
1854: (8)
1855: (8)
1856: (8)
1857: (8)
1858: (8)
1859: (8)
1860: (8)
1861: (8)
1862: (8)
1863: (8)
1864: (8)
1865: (8)
1866: (8)
1867: (8)
1868: (8)
1869: (8)
1870: (8)
1871: (8)
1872: (8)
1873: (8)
1874: (8)
1875: (8)
1876: (8)
1877: (8)
1878: (8)
1879: (8)
1880: (8)
1881: (8)
1882: (8)
1883: (8)
1884: (8)
1885: (8)
1886: (8)
1887: (8)
1888: (8)
1889: (8)
1890: (8)
1891: (8)
1892: (8)
1893: (8)
1894: (8)
1895: (8)
1896: (8)
1897: (8)
1898: (8)
1899: (8)
1900: (8)
1901: (8)
1902: (8)
1903: (8)
1904: (8)
1905: (8)
1906: (8)
1907: (8)
1908: (8)
1909: (8)
1910: (8)
1911: (8)
1912: (8)
1913: (8)
1914: (8)
1915: (8)
1916: (8)
1917: (8)
1918: (8)
1919: (8)
1920: (8)
1921: (8)
1922: (8)
1923: (8)
1924: (8)
1925: (8)
1926: (8)
1927: (8)
1928: (8)
1929: (8)
1930: (8)
1931: (8)
1932: (8)
1933: (8)
1934: (8)
1935: (8)
1936: (8)
1937: (8)
1938: (8)
1939: (8)
1940: (8)
1941: (8)
1942: (8)
1943: (8)
1944: (8)
1945: (8)
1946: (8)
1947: (8)
1948: (8)
1949: (8)
1950: (8)
1951: (8)
1952: (8)
1953: (8)
1954: (8)
1955: (8)
1956: (8)
1957: (8)
1958: (8)
1959: (8)
1960: (8)
1
```

```

705: (25)                                     l_not(isintent_callback): """
706: (12)                                     fprintf(stderr,\">#vardebugshowvalue# (call-back in C).\n,#cbname#);"""]},
707: (8)                                     """
708: (8)                                     CFUNCSMESS(\"Saving callback variables for `#varname#`.\n\");
709: (8)                                     ],
710: (8)                                     'cleanupfrompyobj':
711: (8)                                     """
712: (8)                                     CFUNCSMESS(\"Restoring callback variables for `#varname#`.\n\");
713: (8)                                     Py_DECREF(#varname#_cb.args_capi);
714: (4)                                     }"",
715: (8)                                     'need': ['SWAP', 'create_cb_arglist'],
716: (8)                                     '_check': isexternal,
717: (4)                                     '_depend': ''
718: (4)                                     },
719: (8)                                     { # Common
720: (8)                                     'decl': '#ctype# #varname# = 0;',
721: (8)                                     'pyobjcfrom': {debugcapi: '
fprintf(stderr,\"#vardebugshowvalue#\n\",#varname#);'},
722: (8)                                     'callfortran': {l_or(isintent_c, isattr_value): '#varname#,',
723: (8)                                     l_not(l_or(isintent_c, isattr_value)): '&#varname#,',
724: (4)                                     'return': {isintent_out: '#varname#'},
725: (8)                                     '_check': l_and(isscalar, l_not(iscomplex)),
726: (4)                                     },
727: (4)                                     { # Not hidden
728: (8)                                     'decl': ' PyObject *#varname#_capi = Py_None;',
729: (8)                                     'argformat': {isrequired: '0'},
730: (8)                                     'keyformat': {isoptional: '0'},
731: (8)                                     'args_capi': {isrequired: ',#varname#_capi'},
732: (8)                                     'keys_capi': {isoptional: ',#varname#_capi'},
733: (8)                                     'pyobjcfrom': {isintent_inout: ""\n
734: (4)                                     f2py_success = try_pyarr_from_#ctype#(#varname#_capi,&#varname#);
735: (4)                                     if (f2py_success) {"",
736: (8)                                     'closepyobjcfrom': {isintent_inout: " } /*if (f2py_success) of
#varname# pyobjcfrom*/},
737: (8)                                     'need': {isintent_inout: 'try_pyarr_from_#ctype#'},
738: (8)                                     '_check': l_and(isscalar, l_not(iscomplex), l_not(isstring),
739: (24)                                     isintent_nothide)
740: (4)                                     },
741: (8)                                     'frompyobj': [
742: (12)                                     {hasinitvalue: ' if (#varname#_capi == Py_None) #varname# =
#init#; else',
743: (13)                                     '_depend': ''},
744: (12)                                     {l_and(isoptional, l_not(hasinitvalue)): ' if (#varname#_capi
!= Py_None)',
745: (13)                                     '_depend': ''},
746: (12)                                     {l_not(islogical): ''\n
747: (8)                                     f2py_success = #ctype#_from_pyobj(&#varname#,#varname#_capi,"#pname#
() #nth# (#varname#) can't be converted to #ctype#");
748: (4)                                     if (f2py_success) {"",
749: (12)                                     {islogical: ''\n
750: (8)                                     f2py_success = 1;
751: (4)                                     if (f2py_success) {"",
752: (8)                                     ],
753: (8)                                     'cleanupfrompyobj': ' } /*if (f2py_success) of #varname#*/',
754: (8)                                     'need': {l_not(islogical): '#ctype#_from_pyobj'},
755: (8)                                     '_check': l_and(isscalar, l_not(iscomplex), isintent_nothide),
756: (8)                                     '_depend': ''
757: (4)                                     },
758: (8)                                     { # Hidden
759: (8)                                     'frompyobj': {hasinitvalue: ' #varname# = #init#;'},
760: (8)                                     'need': typeid_need_dict,
761: (8)                                     '_check': l_and(isscalar, l_not(iscomplex), isintent_hide),
762: (4)                                     '_depend': ''
763: (8)                                     'frompyobj': {debugcapi: '
fprintf(stderr,\"#vardebugshowvalue#\n\",#varname#);'},
764: (8)                                     '_check': l_and(isscalar, l_not(iscomplex)),
765: (8)                                     '_depend': ''

```

```

766: (4)
767: (4)        },
768: (8)        { # Common
769: (8)          'decl': '    #ctype# #varname#;',
770: (8)          'callfortran': {isintent_c: '#varname#, l_not(isintent_c):
771: (8)            '#varname#,'},
772: (8)          'pyobjcfrom': {debugcapi:
773: (4)            'printf(stderr,"#vardebugshowvalue#\n",#varname#.r,#varname#.i);'},
774: (8)            'return': {isintent_out: ',#varname#_capi'},
775: (8)            '_check': iscomplex
776: (4)        }, { # Not hidden
777: (8)          'decl': '    PyObject *#varname#_capi = Py_None;',
778: (8)          'argformat': {isrequired: '0'},
779: (8)          'keyformat': {isoptional: '0'},
780: (8)          'args_capi': {isrequired: ',#varname#_capi'},
781: (8)          'keys_capi': {isoptional: ',#varname#_capi'},
782: (8)          'need': {isintent_inout: 'try_pyarr_from_#ctype#'},
783: (8)          'pyobjcfrom': {isintent_inout: "    f2py_success =
784: (8)            try_pyarr_from_#ctype#(#varname#_capi,&#varname#);
785: (4)            if (f2py_success) {"}
786: (8)            'closepyobjcfrom': {isintent_inout: "        } /*if (f2py_success) of
787: (22)          '#varname# pyobjcfrom*/",
788: (29)            '_check': l_and(iscomplex, isintent_nothide)
789: (22)        },
790: (22)        'frompyobj': [{hasinitvalue: '        if (#varname#_capi==Py_None)
791: (8)          '#varname#.r = #init.r#, #varname#.i = #init.i#;} else'],
792: (8)            '_check': l_and(isoptional, l_not(hasinitvalue))
793: (8)            ': '        if (#varname#_capi != Py_None)'},
794: (8)            'f2py_success =
795: (4)        },
796: (8)        '#ctype#_from_pyobj(&#varname#,#varname#_capi,"#pyname#() #nth# (#varname#) can't be converted to
797: (8)        '#ctype#");'
798: (4)        },
799: (8)        'frompyobj': {hasinitvalue: '        #varname#.r = #init.r#, #varname#.i =
800: (8)          '#init.i#;'},
801: (8)            '_check': l_and(iscomplex, isintent_hide),
802: (4)            '_depend': ''
803: (8)        },
804: (8)        '# Common
805: (8)          'pyobjcfrom': {isintent_out: '    #varname#_capi =
806: (4)            pyobj_from_#ctype#1(#varname#);'},
807: (8)            'need': ['pyobj_from_#ctype#1'],
808: (8)            '_check': iscomplex
809: (8)        },
810: (4)        },
811: (4)        { # Common
812: (8)          'decl': ['    #ctype# #varname# = NULL;',
813: (17)            '        int slen(#varname#);',
814: (17)            '        PyObject *#varname#_capi = Py_None;'],
815: (8)          'callfortran': '#varname#,',
816: (8)          'callfortranappend': 'slen(#varname#),',
817: (8)          'pyobjcfrom': [
818: (12)            {debugcapi:
819: (13)              '        fprintf(stderr,
820: (13)                "#vardebugshowvalue#\n",slen(#varname#),#varname#);'},
821: (12)              '{l_and(isintent_out, l_not(isintent_c)):
822: (13)                "            STRINGPADN(#varname#, slen(#varname#), ' ', '\0');"},'],
823: (8)          'return': {isintent_out: ',#varname#'},
824: (8)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

825: (8)           'need': ['len..'],
826: (17)         {l_and(isintent_out, l_not(isintent_c)): 'STRINGPADN'}],
827: (8)           '_check': isstring
828: (4)           }, { # Common
829: (8)             'frompyobj': [
830: (12)               """\
831: (4)                 slen(#varname#) = #elsize#;
832: (4)                 f2py_success = #ctype#_from_pyobj(&#varname#, &slen(#varname#), #init#, """
833: (0)                   """#varname#_capi,\\"#ctype#_from_pyobj failed in converting #nth#"""
834: (0)                   """`#varname#\` of #pyname# to C #ctype#\\"");
835: (4)                   if (f2py_success) {""",
836: (12)                     {l_not(isintent_c):
837: (13)                       "          STRINGPADN(#varname#, slen(#varname#), '\\0', ' ') ;"},"
838: (8)                     ],
839: (8)                     'cleanupfrompyobj': """\
840: (8)                       STRINGFREE(#varname#);
841: (4)                   } /*if (f2py_success) of #varname#*/ """",
842: (8)                   'need': ['#ctype#_from_pyobj', 'len..', 'STRINGFREE',
843: (17)                     {l_not(isintent_c): 'STRINGPADN'}],
844: (8)                   '_check': isstring,
845: (8)                   '_depend': ''
846: (4)                   }, { # Not hidden
847: (8)                     'argformat': {isrequired: '0'},
848: (8)                     'keyformat': {isoptional: '0'},
849: (8)                     'args_capi': {isrequired: ',&#varname#_capi'},
850: (8)                     'keys_capi': {isoptional: ',&#varname#_capi'},
851: (8)                     'pyobjfrom': [
852: (12)                       {l_and(isintent_inout, l_not(isintent_c)):
853: (13)                         "          STRINGPADN(#varname#, slen(#varname#), ' ', '\\0') ;"},"
854: (12)                         {isintent_inout: '''\
855: (4)                           f2py_success = try_pyarr_from_#ctype#(#varname#_capi, #varname#,
856: (42)   slen(#varname#));
857: (4)                           if (f2py_success) {''''},
858: (8)                             'closepyobjfrom': {isintent_inout: '      } /*if (f2py_success) of
#varname# pyobjfrom*/' },
859: (8)                               'need': {isintent_inout: 'try_pyarr_from_#ctype#',
860: (17)                                 l_and(isintent_inout, l_not(isintent_c)): 'STRINGPADN'},
861: (8)                                 '_check': l_and(isstring, isintent_nothide)
862: (4)                               }, { # Hidden
863: (8)                                 '_check': l_and(isstring, isintent_hide)
864: (4)                               }, {
865: (8)                                 'frompyobj': {debugcapi: '
fprintf(stderr, "#vardebugshowvalue#\n", slen(#varname#), #varname#);'},
866: (8)                                   '_check': isstring,
867: (8)                                   '_depend': ''
868: (4)                               },
869: (4)                               { # Common
870: (8)                                 'decl': ['      #ctype# *#varname# = NULL;',
871: (17)                                   '      npy_intp #varname#_Dims[#rank#] = {#rank*[-1]#};',
872: (17)                                   '      const int #varname#_Rank = #rank#;',
873: (17)                                   '      PyArrayObject *capi_#varname#_as_array = NULL;',
874: (17)                                   '      int capi_#varname#_intent = 0;',
875: (17)                                   {isstringarray: '      int slen(#varname#) = 0;'},
876: (17)                                   ],
877: (8)                                 'callfortran': '#varname#,' ,
878: (8)                                 'callfortranappend': {isstringarray: 'slen(#varname#),'},
879: (8)                                 'return': {isintent_out: ',capi_#varname#_as_array'},
880: (8)                                 'need': 'len..',
881: (8)                                 '_check': isarray
882: (4)                               }, { # intent(overwrite) array
883: (8)                                 'decl': '      int capi_overwrite_#varname# = 1;',
884: (8)                                 'kwlistxa': "'overwrite_#varname#",',
885: (8)                                 'xaformat': 'i',
886: (8)                                 'keys_xa': ',&capi_overwrite_#varname#',
887: (8)                                 'docsignxa': 'overwrite_#varname#=1,' ,
888: (8)                                 'docsignxashort': 'overwrite_#varname#,',
889: (8)                                 'docstrop': 'overwrite_#varname# : input int, optional\\n      Default:
1',
890: (8)                                 '_check': l_and(isarray, isintent_overwrite),

```

```

891: (4)             }, {
892: (8)             'frompyobj': '    capi_##varname##_intent |= (capi_overwrite_##varname##?
0:F2PY_INTENT_COPY);',
893: (8)             '_check': l_and(isarray, isintent_overwrite),
894: (8)             '_depend': '',
895: (4)             },
896: (4)             { # intent(copy) array
897: (8)             'decl': '    int capi_overwrite_##varname## = 0;',
898: (8)             'kwlistxa': '"overwrite_##varname##",',
899: (8)             'xaformat': 'i',
900: (8)             'keys_xa': ',&capi_overwrite_##varname##',
901: (8)             'docsignxa': 'overwrite_##varname##=0',
902: (8)             'docsignxashort': 'overwrite_##varname##',
903: (8)             'docstrop': 'overwrite_##varname## : input int, optional\n      Default:
0',
904: (8)             '_check': l_and(isarray, isintent_copy),
905: (4)             },
906: (8)             { 'frompyobj': '    capi_##varname##_intent |= (capi_overwrite_##varname##?
0:F2PY_INTENT_COPY);',
907: (8)             '_check': l_and(isarray, isintent_copy),
908: (8)             '_depend': '',
909: (4)             },
910: (8)             { 'need': [{hasinitvalue: 'forcomb'}, {hasinitvalue: 'CFUNCSMESS'}],
911: (8)             '_check': isarray,
912: (8)             '_depend': ''
913: (4)             }, { # Not hidden
914: (8)             'decl': '    PyObject *#varname##_capi = Py_None;',
915: (8)             'argformat': {isrequired: '0'},
916: (8)             'keyformat': {isoptional: '0'},
917: (8)             'args_capi': {isrequired: ',#varname##_capi'},
918: (8)             'keys_capi': {isoptional: ',#varname##_capi'},
919: (8)             '_check': l_and(isarray, isintent_nothide)
920: (4)             },
921: (8)             { 'frompyobj': [
922: (12)                 '#setdims#;',
923: (12)                 '    capi_##varname##_intent |= #intent#;',
924: (12)                 '(' const char * capi_errmess = "#modulename#.pyname#:"
925: (13)                 ' failed to create array from the #nth# `#varname##`;',
926: (12)                 'isintent_hide:
927: (13)                 '    capi_##varname##_as_array = ndarray_from_pyobj('
928: (13)                 '    #atype#, #elsize#, #varname##_Dims, #varname##_Rank, '
929: (13)                 '    capi_##varname##_intent, Py_None, capi_errmess);',
930: (12)                 'isintent_nothide:
931: (13)                 '    capi_##varname##_as_array = ndarray_from_pyobj('
932: (13)                 '    #atype#, #elsize#, #varname##_Dims, #varname##_Rank, '
933: (13)                 '    capi_##varname##_intent, #varname##_capi, capi_errmess);',
934: (12)                 '"""\'
935: (4)             if (capi_##varname##_as_array == NULL) {
936: (8)                 PyObject* capi_err = PyErr_Occurred();
937: (8)                 if (capi_err == NULL) {
938: (12)                     capi_err = #modulename##_error;
939: (12)                     PyErr_SetString(capi_err, capi_errmess);
940: (8)                 }
941: (4)             } else {
942: (0)             """",
943: (12)                 {isstringarray:
944: (13)                     '    slen(#varname##) = f2py_itemsize(#varname##);'},
945: (12)                 {hasinitvalue:
946: (16)                     {isintent_nothide:
947: (17)                         '    if (#varname##_capi == Py_None) {}',
948: (16)                         {isintent_hide: '    {}',
949: (16)                         {iscomplexarray: '        #ctype# capi_c;'},
950: (16)                         '"""\'
951: (8)                     int *_i, capi_i=0;
952: (8)                     CFUNCSMESS("#name#: Initializing #varname##=##init#\n");
953: (8)                     if (initforcomb(PyArray_DIMS(capi_##varname##_as_array),
954: (24)                         PyArray_NDIM(capi_##varname##_as_array), 1)) {
955: (12)                         while ((_i = nextforcomb())))
956: (8)                     } else {

```

```

957: (12)          PyObject *exc, *val, *tb;
958: (12)          PyErr_Fetch(&exc, &val, &tb);
959: (12)          PyErr_SetString(exc ? exc : #modulename#_error,
960: (16)                  \Initialization of #nth# #varname# failed (initforcomb).\");
961: (12)          npy_PyErr_ChainExceptionsCause(exc, val, tb);
962: (12)          f2py_success = 0;
963: (8)      }
964: (4)  }
965: (4)  if (f2py_success) {"""}],
966: (22)          ],
967: (8)          'cleanupfrompyobj': [ # note that this list will be reversed
968: (12)                  ''],
969: (12)                  /* if (capi_##varname##_as_array == NULL) ... else of #varname#
*/',
970: (12)          {l_not(l_or(isintent_out, isintent_hide)): """\n
971: (4)  if((PyObject *)capi_##varname##_as_array!=#varname##_capi) {
972: (8)      Py_XDECREF(capi_##varname##_as_array); }"""},
973: (12)      {l_and(isintent_hide, l_not(isintent_out))
974: (19)          : """\n      Py_XDECREF(capi_##varname##_as_array);"""},
975: (12)          {hasinitvalue: ' ' } /*if (f2py_success) of #varname# init*/'},
976: (8)      ],
977: (8)      '_check': isarray,
978: (8)      '_depend': ''
979: (4) },
980: (4) {
981: (8)     # Common
982: (4)     '_check': l_and(isarray, l_not(iscomplexarray))
983: (8) }, { # Not hidden
984: (4)     '_check': l_and(isarray, l_not(iscomplexarray), isintent_nothide)
985: (4) },
986: (5)     {'need': '#ctype#',
987: (5)         '_check': isint1array,
988: (5)         '_depend': ''
989: (5) },
990: (5)     {'need': '#ctype#',
991: (5)         '_check': isunsigned_chararray,
992: (5)         '_depend': ''
993: (5) },
994: (5)     {'need': '#ctype#',
995: (5)         '_check': isunsigned_shortarray,
996: (5)         '_depend': ''
997: (5) },
998: (5)     {'need': '#ctype#',
999: (5)         '_check': isunsigned_long_longarray,
1000: (5)         '_depend': ''
1001: (5) },
1002: (5)     {'need': '#ctype#',
1003: (5)         '_check': iscomplexarray,
1004: (5)         '_depend': ''
1005: (4) },
1006: (8)     {'need': 'string',
1007: (8)         '_check': ischaracter,
1008: (4) },
1009: (4)     {'need': 'string',
1010: (8)         '_check': ischaracterarray,
1011: (8) },
1012: (4)     {'callfortranappend': {isarrayofstrings: 'flen(#varname#),'},
1013: (4)         'need': 'string',
1014: (8)         '_check': issstringarray
1015: (8)     },
1016: (8)     'need': 'string',
1017: (4)     '_check': issstringarray
1018: (0)   }
1019: (0)   check_rules = [
1020: (4)     {
1021: (8)         'frompyobj': {debugcapi: '    fprintf(stderr,\\"debug-capi:Checking
`#check#\\"\\n\\");'},
1022: (8)         'need': 'len..'
1023: (4)     }, {

```

```

1024: (8)             'frompyobj': '      CHECKSCALAR(#check#, "#check#", "#nth#"
#varname#", "#varshowvalue#", #varname#) {',
1025: (8)                 'cleanupfrompyobj': '      } /*CHECKSCALAR(#check#)*/',
1026: (8)                 'need': 'CHECKSCALAR',
1027: (8)                 '_check': l_and(isscalar, l_not(iscomplex)),
1028: (8)                 '_break': ''
1029: (4)             }, {
1030: (8)                 'frompyobj': '      CHECKSTRING(#check#, "#check#", "#nth#"
#varname#", "#varshowvalue#", #varname#) {',
1031: (8)                 'cleanupfrompyobj': '      } /*CHECKSTRING(#check#)*/',
1032: (8)                 'need': 'CHECKSTRING',
1033: (8)                 '_check': isstring,
1034: (8)                 '_break': ''
1035: (4)             }, {
1036: (8)                 'need': 'CHECKARRAY',
1037: (8)                 'frompyobj': '      CHECKARRAY(#check#, "#check#", "#nth# #varname#")'
',
1038: (8)                 'cleanupfrompyobj': '      } /*CHECKARRAY(#check#)*/',
1039: (8)                 '_check': isarray,
1040: (8)                 '_break': ''
1041: (4)             }, {
1042: (8)                 'need': 'CHECKGENERIC',
1043: (8)                 'frompyobj': '      CHECKGENERIC(#check#, "#check#", "#nth#"
#varname#") {',
1044: (8)                 'cleanupfrompyobj': '      } /*CHECKGENERIC(#check#)*/',
1045: (4)
1046: (0)
1047: (0)         def buildmodule(m, um):
1048: (4)             """
1049: (4)             Return
1050: (4)             """
1051: (4)             outmess('      Building module "%s"...\\n' % (m['name']))
1052: (4)             ret = {}
1053: (4)             mod_rules = defmod_rules[:]
1054: (4)             vrd = capi_maps.modsign2map(m)
1055: (4)             rd = dictappend({'f2py_version': f2py_version}, vrd)
1056: (4)             funcwrappers = []
1057: (4)             funcwrappers2 = [] # F90 codes
1058: (4)             for n in m['interfaced']:
1059: (8)                 nb = None
1060: (8)                 for bi in m['body']:
1061: (12)                     if bi['block'] not in ['interface', 'abstract interface']:
1062: (16)                         errmess('buildmodule: Expected interface block. Skipping.\\n')
1063: (16)                         continue
1064: (12)                     for b in bi['body']:
1065: (16)                         if b['name'] == n:
1066: (20)                             nb = b
1067: (20)                             break
1068: (8)                         if not nb:
1069: (12)                             print(
1070: (16)                                 'buildmodule: Could not find the body of interfaced routine
"%s". Skipping.\\n' % (n), file=sys.stderr)
1071: (12)                             continue
1072: (8)                             nb_list = [nb]
1073: (8)                             if 'entry' in nb:
1074: (12)                                 for k, a in nb['entry'].items():
1075: (16)                                     nb1 = copy.deepcopy(nb)
1076: (16)                                     del nb1['entry']
1077: (16)                                     nb1['name'] = k
1078: (16)                                     nb1['args'] = a
1079: (16)                                     nb_list.append(nb1)
1080: (8)                                 for nb in nb_list:
1081: (12)                                     isf90 = requiresf90wrapper(nb)
1082: (12)                                     if options['emptygen']:
1083: (16)   b_path = options['buildpath']
1084: (16)   m_name = vrd['modulename']
1085: (16)   outmess('      Generating possibly empty wrappers"\\n')
1086: (16)   Path(f"{b_path}/{vrd['coutput']}").touch()
1087: (16)   if isf90:

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1088: (20)           outmess(f'    Maybe empty "{m_name}-f2pywrappers2.f90"\n')
1089: (20)           Path(f'{b_path}/{m_name}-f2pywrappers2.f90').touch()
1090: (20)           outmess(f'    Maybe empty "{m_name}-f2pywrappers.f"\n')
1091: (20)           Path(f'{b_path}/{m_name}-f2pywrappers.f').touch()
1092: (16)           else:
1093: (20)               outmess(f'    Maybe empty "{m_name}-f2pywrappers.f"\n')
1094: (20)               Path(f'{b_path}/{m_name}-f2pywrappers.f').touch()
1095: (12)           api, wrap = buildapi(nb)
1096: (12)           if wrap:
1097: (16)               if isf90:
1098: (20)                   funcwrappers2.append(wrap)
1099: (16)               else:
1100: (20)                   funcwrappers.append(wrap)
1101: (12)           ar = applyrules(api, vrd)
1102: (12)           rd = dictappend(rd, ar)
1103: (4)            cr, wrap = common_rules.buildhooks(m)
1104: (4)            if wrap:
1105: (8)                funcwrappers.append(wrap)
1106: (4)                ar = applyrules(cr, vrd)
1107: (4)                rd = dictappend(rd, ar)
1108: (4)                mr, wrap = f90mod_rules.buildhooks(m)
1109: (4)                if wrap:
1110: (8)                    funcwrappers2.append(wrap)
1111: (4)                    ar = applyrules(mr, vrd)
1112: (4)                    rd = dictappend(rd, ar)
1113: (4)                    for u in um:
1114: (8)                        ar = use_rules.buildusevars(u, m['use'][u['name']])
1115: (8)                        rd = dictappend(rd, ar)
1116: (4)                    needs = cfuncs.get_needs()
1117: (4)                    needs['typedefs'] += [cvar for cvar in capi_maps.f2cmap_mapped # 
1118: (26)                                if cvar in typedef_need_dict.values()]
1119: (4)                    code = {}
1120: (4)                    for n in needs.keys():
1121: (8)                        code[n] = []
1122: (8)                        for k in needs[n]:
1123: (12)                            c = ''
1124: (12)                            if k in cfuncs.includes0:
1125: (16)                                c = cfuncs.includes0[k]
1126: (12)                            elif k in cfuncs.includes:
1127: (16)                                c = cfuncs.includes[k]
1128: (12)                            elif k in cfuncs.userincludes:
1129: (16)                                c = cfuncs.userincludes[k]
1130: (12)                            elif k in cfuncs	typedefs:
1131: (16)                                c = cfuncs	typedefs[k]
1132: (12)                            elif k in cfuncs	typedefs_generated:
1133: (16)                                c = cfuncs	typedefs_generated[k]
1134: (12)                            elif k in cfuncs.cppmacros:
1135: (16)                                c = cfuncs.cppmacros[k]
1136: (12)                            elif k in cfuncs.cfuncs:
1137: (16)                                c = cfuncs.cfuncs[k]
1138: (12)                            elif k in cfuncs.callbacks:
1139: (16)                                c = cfuncs.callbacks[k]
1140: (12)                            elif k in cfuncs.f90modhooks:
1141: (16)                                c = cfuncs.f90modhooks[k]
1142: (12)                            elif k in cfuncs.commonhooks:
1143: (16)                                c = cfuncs.commonhooks[k]
1144: (12)                            else:
1145: (16)                                errmess('buildmodule: unknown need %s.\n' % (repr(k)))
1146: (16)                                continue
1147: (12)                            code[n].append(c)
1148: (4)            mod_rules.append(code)
1149: (4)            for r in mod_rules:
1150: (8)                if ('_check' in r and r['_check'](m)) or ('_check' not in r):
1151: (12)                    ar = applyrules(r, vrd, m)
1152: (12)                    rd = dictappend(rd, ar)
1153: (4)                    ar = applyrules(module_rules, rd)
1154: (4)                    fn = os.path.join(options['buildpath'], vrd['coutput'])
1155: (4)                    ret['csrc'] = fn
1156: (4)                    with open(fn, 'w') as f:

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1157: (8)           f.write(ar['modulebody'].replace('\t', 2 * ' '))
1158: (4)           outmess('    Wrote C/API module "%s" to file "%s"\n' % (m['name'], fn))
1159: (4)           if options['dorestdoc']:
1160: (8)             fn = os.path.join(
1161: (12)               options['buildpath'], vrd['modulename'] + 'module.rest')
1162: (8)             with open(fn, 'w') as f:
1163: (12)               f.write('... -*- rest -*-\n')
1164: (12)               f.write('\n'.join(ar['restdoc']))
1165: (8)             outmess('    ReST Documentation is saved to file "%s/%smodule.rest"\n'
%                           (options['buildpath'], vrd['modulename']))
1166: (16)
1167: (4)           if options['dolatexdoc']:
1168: (8)             fn = os.path.join(
1169: (12)               options['buildpath'], vrd['modulename'] + 'module.tex')
1170: (8)             ret['ltx'] = fn
1171: (8)             with open(fn, 'w') as f:
1172: (12)               f.write(
1173: (16)                 '%% This file is auto-generated with f2py (version:%s)\n' %
(f2py_version)
1174: (12)               if 'shortlatex' not in options:
1175: (16)                 f.write(
1176: (20)
'\\documentclass{article}\n\\usepackage{a4wide}\n\\begin{document}\n\\tableofcontents\n\n'
1177: (16)                 f.write('\n'.join(ar['latexdoc']))
1178: (12)                 if 'shortlatex' not in options:
1179: (16)                   f.write('\\end{document}')
1180: (8)                 outmess('    Documentation is saved to file "%s/%smodule.tex"\n' %
1181: (16)                   (options['buildpath'], vrd['modulename']))
1182: (4)           if funcwrappers:
1183: (8)             wn = os.path.join(options['buildpath'], vrd['f2py_wrapper_output'])
1184: (8)             ret['fsrc'] = wn
1185: (8)             with open(wn, 'w') as f:
1186: (12)               f.write('C      -*- fortran -*-\n')
1187: (12)               f.write(
1188: (16)                 'C      This file is autogenerated with f2py (version:%s)\n' %
(f2py_version)
1189: (12)               f.write(
1190: (16)                 'C      It contains Fortran 77 wrappers to fortran
functions.\n')
1191: (12)               lines = []
1192: (12)               for l in ('\n\n'.join(funcwrappers) + '\n').split('\n'):
1193: (16)                 if 0 <= l.find('!') < 66:
1194: (20)                   lines.append(l + '\n')
1195: (16)                 elif l and l[0] == ' ':
1196: (20)                   while len(l) >= 66:
1197: (24)                     lines.append(l[:66] + '\n      &')
1198: (24)                     l = l[66:]
1199: (20)                   lines.append(l + '\n')
1200: (16)
1201: (20)
1202: (12)
1203: (12)
1204: (8)                   lines = ''.join(lines).replace('\n      &\n', '\n')
1205: (4)                   f.write(lines)
1206: (8)                   outmess('    Fortran 77 wrappers are saved to "%s"\n' % (wn))
1207: (12)           if funcwrappers2:
1208: (12)             wn = os.path.join(
1209: (12)               options['buildpath'], '%s-f2pywrappers2.f90' %
(vrd['modulename']))
1210: (8)             ret['fsrc'] = wn
1211: (8)             with open(wn, 'w') as f:
1212: (12)               f.write('!      -*- f90 -*-\n')
1213: (12)               f.write(
1214: (16)                 '!      This file is autogenerated with f2py (version:%s)\n' %
(f2py_version))
1215: (12)               f.write(
1216: (16)                 '!      It contains Fortran 90 wrappers to fortran
functions.\n')
1217: (12)               lines = []
1218: (12)               for l in ('\n\n'.join(funcwrappers2) + '\n').split('\n'):
1219: (16)                 if 0 <= l.find('!') < 72:

```

```

1218: (20)             lines.append(l + '\n')
1219: (16)             elif len(l) > 72 and l[0] == ' ':
1220: (20)                 lines.append(l[:72] + '&\n      &')
1221: (20)                 l = l[72:]
1222: (20)                 while len(l) > 66:
1223: (24)                     lines.append(l[:66] + '&\n      &')
1224: (24)                     l = l[66:]
1225: (20)                     lines.append(l + '\n')
1226: (16)                 else:
1227: (20)                     lines.append(l + '\n')
1228: (12)                 lines = ''.join(lines).replace('\n      &\n', '\n')
1229: (12)                 f.write(lines)
1230: (8)                 outmess('      Fortran 90 wrappers are saved to "%s"\n' % (wn))
1231: (4)                 return ret
1232: (0)             stnd = {1: 'st', 2: 'nd', 3: 'rd', 4: 'th', 5: 'th',
1233: (8)                 6: 'th', 7: 'th', 8: 'th', 9: 'th', 0: 'th'}
1234: (0)             def buildapi(rout):
1235: (4)                 rout, wrap = func2subr.assubr(rout)
1236: (4)                 args, depargs = getargs2(rout)
1237: (4)                 capi_maps.depargs = depargs
1238: (4)                 var = rout['vars']
1239: (4)                 if ismoduleroutine(rout):
1240: (8)                     outmess('      Constructing wrapper function "%s.%s"...\\n' %
1241: (16)                         (rout['modulename'], rout['name']))
1242: (4)                 else:
1243: (8)                     outmess('      Constructing wrapper function "%s"...\\n' %
(rout['name']))
1244: (4)             vrd = capi_maps.routsign2map(rout)
1245: (4)             rd = dictappend({}, vrd)
1246: (4)             for r in rout_rules:
1247: (8)                 if ('_check' in r and r['_check'](rout)) or ('_check' not in r):
1248: (12)                     ar = applyrules(r, vrd, rout)
1249: (12)                     rd = dictappend(rd, ar)
1250: (4)             nth, nthk = 0, 0
1251: (4)             savevrd = {}
1252: (4)             for a in args:
1253: (8)                 vrd = capi_maps.sign2map(a, var[a])
1254: (8)                 if isintent_aux(var[a]):
1255: (12)                     _rules = aux_rules
1256: (8)                 else:
1257: (12)                     _rules = arg_rules
1258: (12)                     if not isintent_hide(var[a]):
1259: (16)                         if not isoptional(var[a]):
1260: (20)                             nth = nth + 1
1261: (20)                             vrd['nth'] = repr(nth) + stnd[nth % 10] + ' argument'
1262: (16)                         else:
1263: (20)                             nthk = nthk + 1
1264: (20)                             vrd['nth'] = repr(nthk) + stnd[nthk % 10] + ' keyword'
1265: (12)                         else:
1266: (16)                             vrd['nth'] = 'hidden'
1267: (8)                         savevrd[a] = vrd
1268: (8)                         for r in _rules:
1269: (12)                             if '_depend' in r:
1270: (16)                                 continue
1271: (12)                             if ('_check' in r and r['_check'](var[a])) or ('_check' not in r):
1272: (16)                                 ar = applyrules(r, vrd, var[a])
1273: (16)                                 rd = dictappend(rd, ar)
1274: (16)                                 if '_break' in r:
1275: (20)                                     break
1276: (4)                         for a in depargs:
1277: (8)                             if isintent_aux(var[a]):
1278: (12)                                 _rules = aux_rules
1279: (8)                             else:
1280: (12)                                 _rules = arg_rules
1281: (8)                             vrd = savevrd[a]
1282: (8)                             for r in _rules:
1283: (12)                                 if '_depend' not in r:
1284: (16)                                     continue
1285: (12)                                 if ('_check' in r and r['_check'](var[a])) or ('_check' not in r):

```

```

1286: (16)             ar = applyrules(r, vrd, var[a])
1287: (16)             rd = dictappend(rd, ar)
1288: (16)             if '_break' in r:
1289: (20)                 break
1290: (8)             if 'check' in var[a]:
1291: (12)                 for c in var[a]['check']:
1292: (16)                     vrd['check'] = c
1293: (16)                     ar = applyrules(check_rules, vrd, var[a])
1294: (16)                     rd = dictappend(rd, ar)
1295: (4)             if isinstance(rd['cleanupfrompyobj'], list):
1296: (8)                 rd['cleanupfrompyobj'].reverse()
1297: (4)             if isinstance(rd['closepyobjfrom'], list):
1298: (8)                 rd['closepyobjfrom'].reverse()
1299: (4)             rd['docsignature'] =
stripcomma(replace('#docsign##docsignopt##docsignxa#',
1300: (44)                               {'docsign': rd['docsign']},
1301: (45)                               'docsignopt': rd['docsignopt'],
1302: (45)                               'docsignxa': rd['docsignxa']}))
1303: (4)             optargs = stripcomma(replace('#docsignopt##docsignxa#',
1304: (33)                               {'docsignxa': rd['docsignxashort'],
1305: (34)                               'docsignopt': rd['docsignoptshort']}))
1306: (33)
1307: (4)             if optargs == '':
1308: (8)                 rd['docsignatureshort'] = stripcomma(
1309: (12)                   replace('#docsign#', {'docsign': rd['docsign']}))
1310: (4)             else:
1311: (8)                 rd['docsignatureshort'] = replace('#docsign#[#docsignopt#]',
1312: (42)                               {'docsign': rd['docsign'],
1313: (43)                               'docsignopt': optargs,
1314: (43)
1315: (4)                               })
1316: (4)                 rd['latexdocsignatureshort'] = rd['docsignatureshort'].replace('_', '\\_')
1317: (8)                 rd['latexdocsignatureshort'] = rd[
1318: (4)                   'latexdocsignatureshort'].replace(',', ', ')
1319: (21)                 cfs = stripcomma(replace('#callfortran##callfortranappend#', {
1320: (4)                               'callfortran': rd['callfortran'], 'callfortranappend':
1321: (8)                                 rd['callfortranappend'])})
1322: (45)                               'callfortran': rd['callfortran'],
1323: (4)             'callfortranappend': rd['callfortranappend'])))
1324: (8)             else:
1325: (4)                 rd['callcompaqfortran'] = cfs
1326: (4)             rd['callfortran'] = cfs
1327: (4)             if isinstance(rd['doctreturn'], list):
1328: (8)                 rd['doctreturn'] = stripcomma(
1329: (12)                   replace('#doctreturn#', {'doctreturn': rd['doctreturn']})) + ' = '
1330: (4)             rd['docstrsigns'] = []
1331: (4)             rd['latexdocstrsigns'] = []
1332: (8)             for k in ['docstrreq', 'docstropt', 'docstrout', 'docstrcbs']:
1333: (12)                 if k in rd and isinstance(rd[k], list):
1334: (8)                     rd['docstrsigns'] = rd['docstrsigns'] + rd[k]
1335: (8)                 k = 'latex' + k
1336: (12)                 if k in rd and isinstance(rd[k], list):
1337: (16)                     rd['latexdocstrsigns'] = rd['latexdocstrsigns'] + rd[k][0:1] +
1338: (16)                         ['\\begin{description}'] + rd[k][1:] +
1339: (4)                           ['\\end{description}']
1340: (4)             ar = applyrules(routine_rules, rd)
1341: (8)             if ismoduleroutine(rout):
1342: (4)                 outmess('          %s\n' % (ar['docshort']))
1343: (8)             else:
1344: (4)                 outmess('          %s\n' % (ar['docshort']))
1344: (4)             return ar, wrap
-----
```

File 153 - setup.py:

```
1: (0)     """
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2: (0) setup.py for installing F2PY
3: (0) Usage:
4: (3)     pip install .
5: (0) Copyright 2001-2005 Pearu Peterson all rights reserved,
6: (0) Pearu Peterson <pearu@cens.ioc.ee>
7: (0) Permission to use, modify, and distribute this software is given under the
8: (0) terms of the NumPy License.
9: (0) NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
10: (0) $Revision: 1.32 $
11: (0) $Date: 2005/01/30 17:22:14 $
12: (0) Pearu Peterson
13: (0)
14: (0) """
15: (0) from numpy.distutils.core import setup
16: (0) from numpy.distutils.misc_util import Configuration
17: (0) from __version__ import version
18: (4) def configuration(parent_package='', top_path=None):
19: (4)     config = Configuration('f2py', parent_package, top_path)
20: (4)     config.add_subpackage('tests')
21: (4)     config.add_subpackage('_backends')
22: (4)     config.add_data_dir('tests/src')
23: (8)     config.add_data_files(
24: (8)         'src/fortranobject.c',
25: (8)         'src/fortranobject.h',
26: (8)         '_backends/meson.build.template',
27: (4)     )
28: (4)     config.add_data_files('*/*.pyi')
29: (4)     return config
30: (4) if __name__ == "__main__":
31: (4)     config = configuration(top_path='')
32: (4)     config = config.todict()
33: (8)     config['classifiers'] = [
34: (8)         'Development Status :: 5 - Production/Stable',
35: (8)         'Intended Audience :: Developers',
36: (8)         'Intended Audience :: Science/Research',
37: (8)         'License :: OSI Approved :: NumPy License',
38: (8)         'Natural Language :: English',
39: (8)         'Operating System :: OS Independent',
40: (8)         'Programming Language :: C',
41: (8)         'Programming Language :: Fortran',
42: (8)         'Programming Language :: Python',
43: (8)         'Topic :: Scientific/Engineering',
44: (8)         'Topic :: Software Development :: Code Generators',
45: (4)     ]
46: (10)     setup(version=version,
47: (10)             description="F2PY - Fortran to Python Interface Generator",
48: (10)             author="Pearu Peterson",
49: (10)             author_email="pearu@cens.ioc.ee",
50: (10)             maintainer="Pearu Peterson",
51: (10)             maintainer_email="pearu@cens.ioc.ee",
52: (10)             license="BSD",
53: (10)             platforms="Unix, Windows (mingw|cygwin), Mac OSX",
54: (0)             long_description=""")
55: (0)             The Fortran to Python Interface Generator, or F2PY for short, is a
56: (0)             command line tool (f2py) for generating Python C/API modules for
57: (0)             wrapping Fortran 77/90/95 subroutines, accessing common blocks from
58: (0)             Python, and calling Python functions from Fortran (call-backs).
59: (10)             Interfacing subroutines/data from Fortran 90/95 modules is supported."",
60: (10)             url="https://numpy.org/doc/stable/f2py/",
61: (10)             keywords=['Fortran', 'f2py'],
62: (10)             **config)

```

---

File 154 - symbolic.py:

```

1: (0) """
2: (0)     Fortran/C symbolic expressions
3: (0) References:
4: (0)     - J3/21-007: Draft Fortran 202x. https://j3-fortran.org/doc/year/21/21-007.pdf
4: (0)     Copyright 1999 -- 2011 Pearu Peterson all rights reserved.

```

```

5: (0) Copyright 2011 -- present NumPy Developers.
6: (0) Permission to use, modify, and distribute this software is given under the
7: (0) terms of the NumPy License.
8: (0) NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
9: (0)
10: (0)     """__all__ = ['Expr']
11: (0)     import re
12: (0)     import warnings
13: (0)     from enum import Enum
14: (0)     from math import gcd
15: (0)     class Language(Enum):
16: (4)         """
17: (4)             Used as Expr.tostring language argument.
18: (4)             """
19: (4)             Python = 0
20: (4)             Fortran = 1
21: (4)             C = 2
22: (0)         class Op(Enum):
23: (4)             """
24: (4)                 Used as Expr op attribute.
25: (4)                 """
26: (4)                 INTEGER = 10
27: (4)                 REAL = 12
28: (4)                 COMPLEX = 15
29: (4)                 STRING = 20
30: (4)                 ARRAY = 30
31: (4)                 SYMBOL = 40
32: (4)                 TERNARY = 100
33: (4)                 APPLY = 200
34: (4)                 INDEXING = 210
35: (4)                 CONCAT = 220
36: (4)                 RELATIONAL = 300
37: (4)                 TERMS = 1000
38: (4)                 FACTORS = 2000
39: (4)                 REF = 3000
40: (4)                 DEREF = 3001
41: (0)             class RelOp(Enum):
42: (4)                 """
43: (4)                     Used in Op.RELATIONAL expression to specify the function part.
44: (4)                     """
45: (4)                     EQ = 1
46: (4)                     NE = 2
47: (4)                     LT = 3
48: (4)                     LE = 4
49: (4)                     GT = 5
50: (4)                     GE = 6
51: (4)                     @classmethod
52: (4)                     def fromstring(cls, s, language=Language.C):
53: (8)                         if language is Language.Fortran:
54: (12)                             return {'eq.': RelOp.EQ, 'ne.': RelOp.NE,
55: (20)                                 '.lt.': RelOp.LT, '.le.': RelOp.LE,
56: (20)                                 '.gt.': RelOp.GT, '.ge.': RelOp.GE}[s.lower()]
57: (8)                         return {'==': RelOp.EQ, '!=': RelOp.NE, '<': RelOp.LT,
58: (16)                             '<=': RelOp.LE, '>': RelOp.GT, '>=': RelOp.GE}[s]
59: (4)                     def tostring(self, language=Language.C):
60: (8)                         if language is Language.Fortran:
61: (12)                             return {RelOp.EQ: 'eq.', RelOp.NE: 'ne.',
62: (20)                                 RelOp.LT: 'lt.', RelOp.LE: 'le.',
63: (20)                                 RelOp.GT: 'gt.', RelOp.GE: 'ge.'}[self]
64: (8)                         return {RelOp.EQ: '==', RelOp.NE: '!=',
65: (16)                             RelOp.LT: '<', RelOp.LE: '<=',
66: (16)                             RelOp.GT: '>', RelOp.GE: '>='}[self]
67: (0)             class ArithOp(Enum):
68: (4)                 """
69: (4)                     Used in Op.APPLY expression to specify the function part.
70: (4)                     """
71: (4)                     POS = 1
72: (4)                     NEG = 2
73: (4)                     ADD = 3

```

```

74: (4)           SUB = 4
75: (4)           MUL = 5
76: (4)           DIV = 6
77: (4)           POW = 7
78: (0)           class OpError(Exception):
79: (4)             pass
80: (0)           class Precedence(Enum):
81: (4)             """
82: (4)               Used as Expr.tostring precedence argument.
83: (4)             """
84: (4)             ATOM = 0
85: (4)             POWER = 1
86: (4)             UNARY = 2
87: (4)             PRODUCT = 3
88: (4)             SUM = 4
89: (4)             LT = 6
90: (4)             EQ = 7
91: (4)             LAND = 11
92: (4)             LOR = 12
93: (4)             TERNARY = 13
94: (4)             ASSIGN = 14
95: (4)             TUPLE = 15
96: (4)             NONE = 100
97: (0)           integer_types = (int,)
98: (0)           number_types = (int, float)
99: (0)           def _pairs_add(d, k, v):
100: (4)             c = d.get(k)
101: (4)             if c is None:
102: (8)               d[k] = v
103: (4)             else:
104: (8)               c = c + v
105: (8)               if c:
106: (12)                 d[k] = c
107: (8)               else:
108: (12)                 del d[k]
109: (0)           class ExprWarning(UserWarning):
110: (4)             pass
111: (0)           def ewarn(message):
112: (4)             warnings.warn(message, ExprWarning, stacklevel=2)
113: (0)           class Expr:
114: (4)             """Represents a Fortran expression as a op-data pair.
115: (4)             Expr instances are hashable and sortable.
116: (4)             """
117: (4)             @staticmethod
118: (4)             def parse(s, language=Language.C):
119: (8)               """Parse a Fortran expression to a Expr.
120: (8)               """
121: (8)               return fromstring(s, language=language)
122: (4)             def __init__(self, op, data):
123: (8)               assert isinstance(op, Op)
124: (8)               if op is Op.INTEGER:
125: (12)                 assert isinstance(data, tuple) and len(data) == 2
126: (12)                 assert isinstance(data[0], int)
127: (12)                 assert isinstance(data[1], (int, str)), data
128: (8)               elif op is Op.REAL:
129: (12)                 assert isinstance(data, tuple) and len(data) == 2
130: (12)                 assert isinstance(data[0], float)
131: (12)                 assert isinstance(data[1], (int, str)), data
132: (8)               elif op is Op.COMPLEX:
133: (12)                 assert isinstance(data, tuple) and len(data) == 2
134: (8)               elif op is Op.STRING:
135: (12)                 assert isinstance(data, tuple) and len(data) == 2
136: (12)                 assert (isinstance(data[0], str)
137: (20)                   and data[0][:len(data[0])-1] in ('"', "'", '@@'))
138: (12)                 assert isinstance(data[1], (int, str)), data
139: (8)               elif op is Op.SYMBOL:
140: (12)                 assert hash(data) is not None
141: (8)               elif op in (Op.ARRAY, Op.CONCAT):
142: (12)                 assert isinstance(data, tuple)

```

```

143: (12) assert all(isinstance(item, Expr) for item in data), data
144: (8) elif op in (Op.TERMS, Op.FACTORS):
145: (12) assert isinstance(data, dict)
146: (8) elif op is Op.APPLY:
147: (12) assert isinstance(data, tuple) and len(data) == 3
148: (12) assert hash(data[0]) is not None
149: (12) assert isinstance(data[1], tuple)
150: (12) assert isinstance(data[2], dict)
151: (8) elif op is Op.INDEXING:
152: (12) assert isinstance(data, tuple) and len(data) == 2
153: (12) assert hash(data[0]) is not None
154: (8) elif op is Op.TERNARY:
155: (12) assert isinstance(data, tuple) and len(data) == 3
156: (8) elif op in (Op.REF, Op.DEREF):
157: (12) assert isinstance(data, Expr)
158: (8) elif op is Op.RELATIONAL:
159: (12) assert isinstance(data, tuple) and len(data) == 3
160: (8) else:
161: (12) raise NotImplementedError(
162: (16)     f'unknown op or missing sanity check: {op}')
163: (8) self.op = op
164: (8) self.data = data
165: (4) def __eq__(self, other):
166: (8)     return (isinstance(other, Expr)
167: (16)         and self.op is other.op
168: (16)         and self.data == other.data)
169: (4) def __hash__(self):
170: (8)     if self.op in (Op.TERMS, Op.FACTORS):
171: (12)         data = tuple(sorted(self.data.items()))
172: (8)     elif self.op is Op.APPLY:
173: (12)         data = self.data[:2] + tuple(sorted(self.data[2].items()))
174: (8)     else:
175: (12)         data = self.data
176: (8)     return hash((self.op, data))
177: (4) def __lt__(self, other):
178: (8)     if isinstance(other, Expr):
179: (12)         if self.op is not other.op:
180: (16)             return self.op.value < other.op.value
181: (12)         if self.op in (Op.TERMS, Op.FACTORS):
182: (16)             return (tuple(sorted(self.data.items()))
183: (24)                 < tuple(sorted(other.data.items())))
184: (12)         if self.op is Op.APPLY:
185: (16)             if self.data[:2] != other.data[:2]:
186: (20)                 return self.data[:2] < other.data[:2]
187: (16)             return tuple(sorted(self.data[2].items())) < tuple(
188: (20)                 sorted(other.data[2].items()))
189: (12)             return self.data < other.data
190: (8)     return NotImplemented
191: (4) def __le__(self, other): return self == other or self < other
192: (4) def __gt__(self, other): return not (self <= other)
193: (4) def __ge__(self, other): return not (self < other)
194: (4) def __repr__(self):
195: (8)     return f'{type(self).__name__}({self.op}, {self.data!r})'
196: (4) def __str__(self):
197: (8)     return self.tostring()
198: (4) def tostring(self, parent_precedence=Precedence.NONE,
199: (17)                 language=Language.Fortran):
200: (8)     """Return a string representation of Expr.
201: (8)
202: (8)     if self.op in (Op.INTEGER, Op.REAL):
203: (12)         precedence = (Precedence.SUM if self.data[0] < 0
204: (26)                         else Precedence.ATOM)
205: (12)         r = str(self.data[0]) + (f'_{{self.data[1]}}'
206: (37)                         if self.data[1] != 4 else '')
207: (8)     elif self.op is Op.COMPLEX:
208: (12)         r = ', '.join(item.tostring(Precedence.TUPLE, language=language)
209: (26)                         for item in self.data)
210: (12)         r = '(' + r + ')'
211: (12)         precedence = Precedence.ATOM

```

```

212: (8)
213: (12)
214: (12)
215: (8)
216: (12)
217: (12)
218: (16)
219: (12)
220: (8)
221: (12)
222: (26)
223: (12)
224: (12)
225: (8)
226: (12)
227: (12)
228: (16)
229: (20)
230: (20)
231: (16)
232: (20)
233: (16)
234: (20)
235: (16)
236: (20)
237: (24)
238: (20)
239: (24)
240: (28)
241: (16)
242: (20)
243: (16)
244: (20)
245: (16)
246: (12)
247: (12)
248: (8)
249: (12)
250: (12)
251: (12)
252: (16)
253: (16)
254: (20)
255: (43)
256: (16)
257: (20)
258: (24)
259: (47)
260: (24)
261: (20)
262: (24)
263: (47)
264: (24)
265: (24)
266: (20)
267: (24)
268: (47)
269: (24)
270: (16)
271: (20)
272: (43)
273: (16)
274: (20)
275: (16)
276: (12)
277: (16)
278: (20)
279: (16)
280: (12)

        elif self.op is Op.SYMBOL:
            precedence = Precedence.ATOM
            r = str(self.data)
        elif self.op is Op.STRING:
            r = self.data[0]
            if self.data[1] != 1:
                r = self.data[1] + ' ' + r
            precedence = Precedence.ATOM
        elif self.op is Op.ARRAY:
            r = ', '.join(item.tostring(Precedence.TUPLE, language=language)
                          for item in self.data)
            r = '[' + r + ']'
            precedence = Precedence.ATOM
        elif self.op is Op.TERMS:
            terms = []
            for term, coeff in sorted(self.data.items()):
                if coeff < 0:
                    op = ' - '
                    coeff = -coeff
                else:
                    op = ' + '
                if coeff == 1:
                    term = term.tostring(Precedence.SUM, language=language)
                else:
                    if term == as_number(1):
                        term = str(coeff)
                    else:
                        term = f'{coeff} * {term.tostring(Precedence.PRODUCT, language=language)}'
                if terms:
                    terms.append(op)
                elif op == ' - ':
                    terms.append(' - ')
                terms.append(term)
            r = ''.join(terms) or '0'
            precedence = Precedence.SUM if terms else Precedence.ATOM
        elif self.op is Op.FACTORS:
            factors = []
            tail = []
            for base, exp in sorted(self.data.items()):
                op = ' * '
                if exp == 1:
                    factor = base.tostring(Precedence.PRODUCT, language=language)
                elif language is Language.C:
                    if exp in range(2, 10):
                        factor = base.tostring(Precedence.PRODUCT, language=language)
                    else:
                        factor = ' * '.join([factor] * exp)
                elif exp in range(-10, 0):
                    factor = base.tostring(Precedence.PRODUCT, language=language)
                    tail += [factor] * -exp
                    continue
                else:
                    factor = base.tostring(Precedence.TUPLE, language=language)
                    factor = f'pow({factor}, {exp})'
                else:
                    factor = base.tostring(Precedence.POWER, language=language) + f' ** {exp}'
                if factors:
                    factors.append(op)
                factors.append(factor)
            if tail:
                if not factors:
                    factors += ['1']
                factors += ['/ ', '(', ' * '.join(tail), ')']
            r = ''.join(factors) or '1'

```

```

281: (12)                               precedence = Precedence.PRODUCT if factors else Precedence.ATOM
282: (8)                                elif self.op is Op.APPLY:
283: (12)                                  name, args, kwargs = self.data
284: (12)                                  if name is ArithOp.DIV and language is Language.C:
285: (16)                                      numer, denom = [arg.tostring(Precedence.PRODUCT,
286: (45)  language=language)
287: (32)  for arg in args]
288: (16)                                      r = f'{numer} / {denom}'
289: (16)                                      precedence = Precedence.PRODUCT
290: (12)                                else:
291: (16)                                    args = [arg.tostring(Precedence.TUPLE, language=language)
292: (24)   for arg in args]
293: (16)                                    args += [k + '=' + v.tostring(Precedence.NONE)
294: (25)   for k, v in kwargs.items()]
295: (16)                                    r = f'{name}({", ".join(args)})'
296: (16)                                    precedence = Precedence.ATOM
297: (8)                                elif self.op is Op.INDEXING:
298: (12)                                    name = self.data[0]
299: (12)                                    args = [arg.tostring(Precedence.TUPLE, language=language)
300: (20)   for arg in self.data[1:]]
301: (12)                                    r = f'{name}({", ".join(args)})'
302: (12)                                    precedence = Precedence.ATOM
303: (8)                                elif self.op is Op.CONCAT:
304: (12)                                    args = [arg.tostring(Precedence.PRODUCT, language=language)
305: (20)   for arg in self.data]
306: (12)                                    r = " // ".join(args)
307: (12)                                    precedence = Precedence.PRODUCT
308: (8)                                elif self.op is Op.TERNARY:
309: (12)                                    cond, expr1, expr2 = [a.tostring(Precedence.TUPLE,
310: (45)   language=language)
311: (34)   for a in self.data]
312: (12)                                    if language is Language.C:
313: (16)                                      r = f'({cond}?{expr1}:{expr2})'
314: (12)                                    elif language is Language.Python:
315: (16)                                      r = f'{expr1} if {cond} else {expr2})'
316: (12)                                    elif language is Language.Fortran:
317: (16)                                      r = f'merge({expr1}, {expr2}, {cond})'
318: (12)                                else:
319: (16)                                    raise NotImplementedError(
320: (20)   f'tostring for {self.op} and {language}')
321: (12)                                    precedence = Precedence.ATOM
322: (8)                                elif self.op is Op.REF:
323: (12)                                    r = '&' + self.data.tostring(Precedence.UNARY, language=language)
324: (12)                                    precedence = Precedence.UNARY
325: (8)                                elif self.op is Op.DEREF:
326: (12)                                    r = '*' + self.data.tostring(Precedence.UNARY, language=language)
327: (12)                                    precedence = Precedence.UNARY
328: (8)                                elif self.op is Op.RELATIONAL:
329: (12)                                    rop, left, right = self.data
330: (12)                                    precedence = (Precedence.EQ if rop in (RelOp.EQ, RelOp.NE)
331: (26)   else Precedence.LT)
332: (12)                                    left = left.tostring(precedence, language=language)
333: (12)                                    right = right.tostring(precedence, language=language)
334: (12)                                    rop = rop.tostring(language=language)
335: (12)                                    r = f'{left} {rop} {right}'
336: (8)                                else:
337: (12)                                    raise NotImplementedError(f'tostring for op {self.op}')
338: (8)                                if parent_precedence.value < precedence.value:
339: (12)                                    return '(' + r + ')'
340: (8)                                return r
341: (4)                                def __pos__(self):
342: (8)                                    return self
343: (4)                                def __neg__(self):
344: (8)                                    return self * -1
345: (4)                                def __add__(self, other):
346: (8)                                    other = as_expr(other)
347: (8)                                    if isinstance(other, Expr):
348: (12)  if self.op is other.op:
349: (16)  if self.op in (Op.INTEGER, Op.REAL):

```

```

350: (20)
351: (24)
352: (24)
353: (16)
354: (20)
355: (20)
356: (20)
357: (16)
358: (20)
359: (20)
360: (24)
361: (20)
362: (12)
363: (16)
364: (12)
365: (16)
366: (12)
367: (16)
368: (12)
369: (16)
370: (12)
371: (8)
372: (4)
373: (8)
374: (12)
375: (8)
376: (4)
377: (8)
378: (4)
379: (8)
380: (12)
381: (8)
382: (4)
383: (8)
384: (8)
385: (12)
386: (16)
387: (20)
388: (37)
389: (16)
390: (20)
391: (20)
392: (20)
393: (16)
394: (20)
395: (20)
396: (24)
397: (20)
398: (16)
399: (20)
400: (20)
401: (24)
402: (28)
403: (20)
404: (12)
405: (16)
406: (12)
407: (16)
408: (12)
409: (16)
410: (12)
411: (16)
412: (12)
413: (16)
414: (12)
415: (16)
416: (12)
417: (8)
418: (4)

        return as_number(
            self.data[0] + other.data[0],
            max(self.data[1], other.data[1]))
    if self.op is Op.COMPLEX:
        r1, i1 = self.data
        r2, i2 = other.data
        return as_complex(r1 + r2, i1 + i2)
    if self.op is Op.TERMS:
        r = Expr(self.op, dict(self.data))
        for k, v in other.data.items():
            _pairs_add(r.data, k, v)
        return normalize(r)
    if self.op is Op.COMPLEX and other.op in (Op.INTEGER, Op.REAL):
        return self + as_complex(other)
    elif self.op in (Op.INTEGER, Op.REAL) and other.op is Op.COMPLEX:
        return as_complex(self) + other
    elif self.op is Op.REAL and other.op is Op.INTEGER:
        return self + as_real(other, kind=self.data[1])
    elif self.op is Op.INTEGER and other.op is Op.REAL:
        return as_real(self, kind=other.data[1]) + other
    return as_terms(self) + as_terms(other)
return NotImplemented
def __radd__(self, other):
    if isinstance(other, number_types):
        return as_number(other) + self
    return NotImplemented
def __sub__(self, other):
    return self + (-other)
def __rsub__(self, other):
    if isinstance(other, number_types):
        return as_number(other) - self
    return NotImplemented
def __mul__(self, other):
    other = as_expr(other)
    if isinstance(other, Expr):
        if self.op is other.op:
            if self.op in (Op.INTEGER, Op.REAL):
                return as_number(self.data[0] * other.data[0],
                                max(self.data[1], other.data[1]))
            elif self.op is Op.COMPLEX:
                r1, i1 = self.data
                r2, i2 = other.data
                return as_complex(r1 * r2 - i1 * i2, r1 * i2 + r2 * i1)
        if self.op is Op.FACTORS:
            r = Expr(self.op, dict(self.data))
            for k, v in other.data.items():
                _pairs_add(r.data, k, v)
            return normalize(r)
        elif self.op is Op.TERMS:
            r = Expr(self.op, {})
            for t1, c1 in self.data.items():
                for t2, c2 in other.data.items():
                    _pairs_add(r.data, t1 * t2, c1 * c2)
            return normalize(r)
        if self.op is Op.COMPLEX and other.op in (Op.INTEGER, Op.REAL):
            return self * as_complex(other)
        elif other.op is Op.COMPLEX and self.op in (Op.INTEGER, Op.REAL):
            return as_complex(self) * other
        elif self.op is Op.REAL and other.op is Op.INTEGER:
            return self * as_real(other, kind=self.data[1])
        elif self.op is Op.INTEGER and other.op is Op.REAL:
            return as_real(self, kind=other.data[1]) * other
        if self.op is Op.TERMS:
            return self * as_terms(other)
        elif other.op is Op.TERMS:
            return as_terms(self) * other
        return as_factors(self) * as_factors(other)
    return NotImplemented
def __rmul__(self, other):

```

```

419: (8)
420: (12)
421: (8)
422: (4)
423: (8)
424: (8)
425: (12)
426: (16)
427: (16)
428: (20)
429: (16)
430: (20)
431: (16)
432: (20)
433: (24)
434: (24)
435: (28)
436: (24)
437: (20)
438: (16)
439: (20)
440: (16)
441: (12)
442: (8)
443: (4)
444: (8)
445: (8)
446: (12)
447: (8)
448: (4)
449: (8)
450: (8)
451: (12)
452: (8)
453: (4)
454: (8)
455: (8)
456: (12)
457: (8)
458: (4)
459: (8)
460: (8)
461: (12)
462: (8)
463: (4)
464: (8)
465: (24)
466: (4)
467: (8)
468: (8)
469: (12)
470: (8)
471: (12)
472: (8)
473: (4)
474: (8)
475: (8)
476: (8)
477: (8)
478: (12)
479: (12)
480: (16)
481: (12)
482: (12)
483: (16)
484: (16)
485: (20)
486: (16)
487: (12)

        if isinstance(other, number_types):
            return as_number(other) * self
        return NotImplemented
    def __pow__(self, other):
        other = as_expr(other)
        if isinstance(other, Expr):
            if other.op is Op.INTEGER:
                exponent = other.data[0]
                if exponent == 0:
                    return as_number(1)
                if exponent == 1:
                    return self
                if exponent > 0:
                    if self.op is Op.FACTORS:
                        r = Expr(self.op, {})
                        for k, v in self.data.items():
                            r.data[k] = v * exponent
                        return normalize(r)
                    return self * (self ** (exponent - 1))
                elif exponent != -1:
                    return (self ** (-exponent)) ** -1
            return Expr(Op.FACTORS, {self: exponent})
        return as_apply(ArithOp.POW, self, other)
    return NotImplemented
    def __truediv__(self, other):
        other = as_expr(other)
        if isinstance(other, Expr):
            return normalize(as_apply(ArithOp.DIV, self, other))
        return NotImplemented
    def __rtruediv__(self, other):
        other = as_expr(other)
        if isinstance(other, Expr):
            return other / self
        return NotImplemented
    def __floordiv__(self, other):
        other = as_expr(other)
        if isinstance(other, Expr):
            return normalize(Expr(Op.CONCAT, (self, other)))
        return NotImplemented
    def __rfloordiv__(self, other):
        other = as_expr(other)
        if isinstance(other, Expr):
            return other // self
        return NotImplemented
    def __call__(self, *args, **kwargs):
        return as_apply(self, *map(as_expr, args),
                       **dict((k, as_expr(v)) for k, v in kwargs.items()))
    def __getitem__(self, index):
        index = as_expr(index)
        if not isinstance(index, tuple):
            index = index,
        if len(index) > 1:
            ewarn(f'C-index should be a single expression but got `{index}`')
        return Expr(Op.INDEXING, (self,) + index)
    def substitute(self, symbols_map):
        """Recursively substitute symbols with values in symbols map.
        Symbols map is a dictionary of symbol-expression pairs.
        """
        if self.op is Op.SYMBOL:
            value = symbols_map.get(self)
            if value is None:
                return self
            m = re.match(r'\A(@__f2py_PARENTHESIS_(\w+)_\d+@\)\Z', self.data)
            if m:
                items, paren = m.groups()
                if paren in ['ROUNDDIV', 'SQUARE']:
                    return as_array(value)
                assert paren == 'ROUND', (paren, value)
            return value

```

```

488: (8)
489: (12)
490: (8)
491: (12)
492: (39)
493: (8)
494: (12)
495: (49)
496: (8)
497: (12)
498: (12)
499: (16)
500: (20)
501: (16)
502: (20)
503: (12)
504: (16)
505: (22)
506: (16)
507: (12)
508: (8)
509: (12)
510: (12)
511: (16)
512: (20)
513: (16)
514: (20)
515: (12)
516: (16)
517: (22)
518: (16)
519: (12)
520: (8)
521: (12)
522: (12)
523: (16)
524: (12)
525: (12)
526: (26)
527: (12)
528: (8)
529: (12)
530: (12)
531: (16)
532: (12)
533: (12)
534: (8)
535: (12)
536: (12)
537: (8)
538: (12)
539: (8)
540: (12)
541: (12)
542: (12)
543: (12)
544: (8)
545: (4)
546: (8)
547: (8)
548: (8)
549: (8)
550: (8)
551: (8)
552: (8)
553: (8)
554: (8)
555: (12)

        if self.op in (Op.INTEGER, Op.REAL, Op.STRING):
            return self
        if self.op in (Op.ARRAY, Op.COMPLEX):
            return Expr(self.op, tuple(item.substitute(symbols_map)
   for item in self.data))
        if self.op is Op.CONCAT:
            return normalize(Expr(self.op, tuple(item.substitute(symbols_map)
   for item in self.data)))
        if self.op is Op.TERMS:
            r = None
            for term, coeff in self.data.items():
                if r is None:
                    r = term.substitute(symbols_map) * coeff
                else:
                    r += term.substitute(symbols_map) * coeff
            if r is None:
                ewarn('substitute: empty TERMS expression interpreted as'
                      ' int-literal 0')
                return as_number(0)
            return r
        if self.op is Op.FACTORS:
            r = None
            for base, exponent in self.data.items():
                if r is None:
                    r = base.substitute(symbols_map) ** exponent
                else:
                    r *= base.substitute(symbols_map) ** exponent
            if r is None:
                ewarn('substitute: empty FACTORS expression interpreted'
                      ' as int-literal 1')
                return as_number(1)
            return r
        if self.op is Op.APPLY:
            target, args, kwargs = self.data
            if isinstance(target, Expr):
                target = target.substitute(symbols_map)
            args = tuple(a.substitute(symbols_map) for a in args)
            kwargs = dict((k, v.substitute(symbols_map))
                           for k, v in kwargs.items())
            return normalize(Expr(self.op, (target, args, kwargs)))
        if self.op is Op.INDEXING:
            func = self.data[0]
            if isinstance(func, Expr):
                func = func.substitute(symbols_map)
            args = tuple(a.substitute(symbols_map) for a in self.data[1:])
            return normalize(Expr(self.op, (func,) + args))
        if self.op is Op.TERNARY:
            operands = tuple(a.substitute(symbols_map) for a in self.data)
            return normalize(Expr(self.op, operands))
        if self.op in (Op.REF, Op.DEREF):
            return normalize(Expr(self.op, self.data.substitute(symbols_map)))
        if self.op is Op.RELATIONAL:
            rop, left, right = self.data
            left = left.substitute(symbols_map)
            right = right.substitute(symbols_map)
            return normalize(Expr(self.op, (rop, left, right)))
        raise NotImplementedError(f'substitute method for {self.op}:'
                                  '{self!r}')
def traverse(self, visit, *args, **kwargs):
    """Traverse expression tree with visit function.
    The visit function is applied to an expression with given args
    and kwargs.
    Traverse call returns an expression returned by visit when not
    None, otherwise return a new normalized expression with
    traverse-visit sub-expressions.
    """
    result = visit(self, *args, **kwargs)
    if result is not None:
        return result

```

```

556: (8)           if self.op in (Op.INTEGER, Op.REAL, Op.STRING, Op.SYMBOL):
557: (12)          return self
558: (8)          elif self.op in (Op.COMPLEX, Op.ARRAY, Op.CONCAT, Op.TERNARY):
559: (12)             return normalize(Expr(self.op, tuple(
560: (16)               item.traverse(visit, *args, **kwargs)
561: (16)                 for item in self.data)))
562: (8)             elif self.op in (Op.TERMS, Op.FACTORS):
563: (12)               data = {}
564: (12)               for k, v in self.data.items():
565: (16)                 k = k.traverse(visit, *args, **kwargs)
566: (16)                 v = (v.traverse(visit, *args, **kwargs)
567: (21)                   if isinstance(v, Expr) else v)
568: (16)                   if k in data:
569: (20)                     v = data[k] + v
570: (16)                     data[k] = v
571: (12)               return normalize(Expr(self.op, data))
572: (8)             elif self.op is Op.APPLY:
573: (12)               obj = self.data[0]
574: (12)               func = (obj.traverse(visit, *args, **kwargs)
575: (20)                 if isinstance(obj, Expr) else obj)
576: (12)               operands = tuple(operand.traverse(visit, *args, **kwargs)
577: (29)                 for operand in self.data[1:])
578: (12)               kwoperands = dict((k, v.traverse(visit, *args, **kwargs))
579: (30)                 for k, v in self.data[2].items())
580: (12)               return normalize(Expr(self.op, (func, operands, kwoperands)))
581: (8)             elif self.op is Op.INDEXING:
582: (12)               obj = self.data[0]
583: (12)               obj = (obj.traverse(visit, *args, **kwargs)
584: (19)                 if isinstance(obj, Expr) else obj)
585: (12)               indices = tuple(index.traverse(visit, *args, **kwargs)
586: (28)                 for index in self.data[1:])
587: (12)               return normalize(Expr(self.op, (obj,) + indices))
588: (8)             elif self.op in (Op.REF, Op.DEREF):
589: (12)               return normalize(Expr(self.op,
590: (34)                 self.data.traverse(visit, *args, **kwargs)))
591: (8)             elif self.op is Op.RELATIONAL:
592: (12)               rop, left, right = self.data
593: (12)               left = left.traverse(visit, *args, **kwargs)
594: (12)               right = right.traverse(visit, *args, **kwargs)
595: (12)               return normalize(Expr(self.op, (rop, left, right)))
596: (8)             raise NotImplementedError(f'traverse method for {self.op}')
597: (4)           def contains(self, other):
598: (8)             """Check if self contains other.
599: (8)
600: (8)             found = []
601: (8)             def visit(expr, found=found):
602: (12)               if found:
603: (16)                 return expr
604: (12)               elif expr == other:
605: (16)                 found.append(1)
606: (16)                 return expr
607: (8)               self.traverse(visit)
608: (8)               return len(found) != 0
609: (4)           def symbols(self):
610: (8)             """Return a set of symbols contained in self.
611: (8)
612: (8)             found = set()
613: (8)             def visit(expr, found=found):
614: (12)               if expr.op is Op.SYMBOL:
615: (16)                 found.add(expr)
616: (8)               self.traverse(visit)
617: (8)               return found
618: (4)           def polynomial_atoms(self):
619: (8)             """Return a set of expressions used as atoms in polynomial self.
620: (8)
621: (8)             found = set()
622: (8)             def visit(expr, found=found):
623: (12)               if expr.op is Op.FACTORS:
624: (16)                 for b in expr.data:

```

```

625: (20)                                b.traverse(visit)
626: (16)                                return expr
627: (12)                                if expr.op in (Op.TERMS, Op.COMPLEX):
628: (16)                                return
629: (12)                                if expr.op is Op.APPLY and isinstance(expr.data[0], ArithOp):
630: (16)                                    if expr.data[0] is ArithOp.POW:
631: (20)  expr.data[1][0].traverse(visit)
632: (20)                                    return expr
633: (16)                                return
634: (12)                                if expr.op in (Op.INTEGER, Op.REAL):
635: (16)                                    return expr
636: (12)                                found.add(expr)
637: (12)                                if expr.op in (Op.INDEXING, Op.APPLY):
638: (16)                                    return expr
639: (8)                                 self.traverse(visit)
640: (8)                                return found
641: (4) def linear_solve(self, symbol):
642: (8)    """Return a, b such that a * symbol + b == self.
643: (8)    If self is not linear with respect to symbol, raise RuntimeError.
644: (8)    """
645: (8)    b = self.substitute({symbol: as_number(0)})
646: (8)    ax = self - b
647: (8)    a = ax.substitute({symbol: as_number(1)})
648: (8)    zero, _ = as_numer_denom(a * symbol - ax)
649: (8)    if zero != as_number(0):
650: (12)        raise RuntimeError(f'not a {symbol}-linear equation:'
651: (31)                           f' {a} * {symbol} + {b} == {self}')
652: (8)    return a, b
653: (0) def normalize(obj):
654: (4)    """Normalize Expr and apply basic evaluation methods.
655: (4)    """
656: (4)    if not isinstance(obj, Expr):
657: (8)        return obj
658: (4)    if obj.op is Op.TERMS:
659: (8)        d = {}
660: (8)        for t, c in obj.data.items():
661: (12)            if c == 0:
662: (16)                continue
663: (12)            if t.op is Op.COMPLEX and c != 1:
664: (16)                t = t * c
665: (16)                c = 1
666: (12)            if t.op is Op.TERMS:
667: (16)                for t1, c1 in t.data.items():
668: (20)                    _pairs_add(d, t1, c1 * c)
669: (12)            else:
670: (16)                _pairs_add(d, t, c)
671: (8)        if len(d) == 0:
672: (12)            return as_number(0)
673: (8)        elif len(d) == 1:
674: (12)            (t, c), = d.items()
675: (12)            if c == 1:
676: (16)                return t
677: (8)        return Expr(Op.TERMS, d)
678: (4)    if obj.op is Op.FACTORS:
679: (8)        coeff = 1
680: (8)        d = {}
681: (8)        for b, e in obj.data.items():
682: (12)            if e == 0:
683: (16)                continue
684: (12)            if b.op is Op.TERMS and isinstance(e, integer_types) and e > 1:
685: (16)                b = b * (b ** (e - 1))
686: (16)                e = 1
687: (12)            if b.op in (Op.INTEGER, Op.REAL):
688: (16)                if e == 1:
689: (20)                    coeff *= b.data[0]
690: (16)                elif e > 0:
691: (20)                    coeff *= b.data[0] ** e
692: (16)                else:
693: (20)                    _pairs_add(d, b, e)

```

```

694: (12)           elif b.op is Op.FACTORS:
695: (16)             if e > 0 and isinstance(e, integer_types):
696: (20)               for b1, e1 in b.data.items():
697: (24)                 _pairs_add(d, b1, e1 * e)
698: (16)             else:
699: (20)               _pairs_add(d, b, e)
700: (12)             else:
701: (16)               _pairs_add(d, b, e)
702: (8)             if len(d) == 0 or coeff == 0:
703: (12)               assert isinstance(coeff, number_types)
704: (12)               return as_number(coeff)
705: (8)             elif len(d) == 1:
706: (12)               (b, e), = d.items()
707: (12)               if e == 1:
708: (16)                 t = b
709: (12)               else:
710: (16)                 t = Expr(Op.FACTORS, d)
711: (12)               if coeff == 1:
712: (16)                 return t
713: (12)               return Expr(Op.TERMS, {t: coeff})
714: (8)             elif coeff == 1:
715: (12)               return Expr(Op.FACTORS, d)
716: (8)             else:
717: (12)               return Expr(Op.TERMS, {Expr(Op.FACTORS, d): coeff})
718: (4)             if obj.op is Op.APPLY and obj.data[0] is ArithOp.DIV:
719: (8)               dividend, divisor = obj.data[1]
720: (8)               t1, c1 = as_term_coeff(dividend)
721: (8)               t2, c2 = as_term_coeff(divisor)
722: (8)               if isinstance(c1, integer_types) and isinstance(c2, integer_types):
723: (12)                 g = gcd(c1, c2)
724: (12)                 c1, c2 = c1//g, c2//g
725: (8)             else:
726: (12)               c1, c2 = c1/c2, 1
727: (8)             if t1.op is Op.APPLY and t1.data[0] is ArithOp.DIV:
728: (12)               numer = t1.data[1][0] * c1
729: (12)               denom = t1.data[1][1] * t2 * c2
730: (12)               return as_apply(ArithOp.DIV, numer, denom)
731: (8)             if t2.op is Op.APPLY and t2.data[0] is ArithOp.DIV:
732: (12)               numer = t2.data[1][1] * t1 * c1
733: (12)               denom = t2.data[1][0] * c2
734: (12)               return as_apply(ArithOp.DIV, numer, denom)
735: (8)             d = dict(as_factors(t1).data)
736: (8)             for b, e in as_factors(t2).data.items():
737: (12)               _pairs_add(d, b, -e)
738: (8)             numer, denom = {}, {}
739: (8)             for b, e in d.items():
740: (12)               if e > 0:
741: (16)                 numer[b] = e
742: (12)               else:
743: (16)                 denom[b] = -e
744: (8)             numer = normalize(Expr(Op.FACTORS, numer)) * c1
745: (8)             denom = normalize(Expr(Op.FACTORS, denom)) * c2
746: (8)             if denom.op in (Op.INTEGER, Op.REAL) and denom.data[0] == 1:
747: (12)               return numer
748: (8)             return as_apply(ArithOp.DIV, numer, denom)
749: (4)             if obj.op is Op.CONCAT:
750: (8)               lst = [obj.data[0]]
751: (8)               for s in obj.data[1:]:
752: (12)                 last = lst[-1]
753: (12)                 if (
754: (20)                   last.op is Op.STRING
755: (20)                   and s.op is Op.STRING
756: (20)                   and last.data[0][0] in "'\""
757: (20)                   and s.data[0][0] == last.data[0][-1]
758: (12)                 ):
759: (16)                   new_last = as_string(last.data[0][:-1] + s.data[0][1:],
760: (37)                               max(last.data[1], s.data[1]))
761: (16)                   lst[-1] = new_last
762: (12)                 else:

```

```

763: (16)           lst.append(s)
764: (8)            if len(lst) == 1:
765: (12)           return lst[0]
766: (8)            return Expr(Op.CONCAT, tuple(lst))
767: (4)             if obj.op is Op.TERNARY:
768: (8)               cond, expr1, expr2 = map(normalize, obj.data)
769: (8)               if cond.op is Op.INTEGER:
770: (12)                 return expr1 if cond.data[0] else expr2
771: (8)               return Expr(Op.TERNARY, (cond, expr1, expr2))
772: (4)             return obj
773: (0)             def as_expr(obj):
774: (4)               """Convert non-Expr objects to Expr objects.
775: (4)
776: (4)               if isinstance(obj, complex):
777: (8)                 return as_complex(obj.real, obj.imag)
778: (4)               if isinstance(obj, number_types):
779: (8)                 return as_number(obj)
780: (4)               if isinstance(obj, str):
781: (8)                 return as_string(repr(obj))
782: (4)               if isinstance(obj, tuple):
783: (8)                 return tuple(map(as_expr, obj))
784: (4)             return obj
785: (0)             def as_symbol(obj):
786: (4)               """Return object as SYMBOL expression (variable or unparsed expression).
787: (4)
788: (4)               return Expr(Op.SYMBOL, obj)
789: (0)             def as_number(obj, kind=4):
790: (4)               """Return object as INTEGER or REAL constant.
791: (4)
792: (4)               if isinstance(obj, int):
793: (8)                 return Expr(Op.INTEGER, (obj, kind))
794: (4)               if isinstance(obj, float):
795: (8)                 return Expr(Op.REAL, (obj, kind))
796: (4)               if isinstance(obj, Expr):
797: (8)                 if obj.op in (Op.INTEGER, Op.REAL):
798: (12)                   return obj
799: (4)                 raise OpError(f'cannot convert {obj} to INTEGER or REAL constant')
800: (0)             def as_integer(obj, kind=4):
801: (4)               """Return object as INTEGER constant.
802: (4)
803: (4)               if isinstance(obj, int):
804: (8)                 return Expr(Op.INTEGER, (obj, kind))
805: (4)               if isinstance(obj, Expr):
806: (8)                 if obj.op is Op.INTEGER:
807: (12)                   return obj
808: (4)                 raise OpError(f'cannot convert {obj} to INTEGER constant')
809: (0)             def as_real(obj, kind=4):
810: (4)               """Return object as REAL constant.
811: (4)
812: (4)               if isinstance(obj, int):
813: (8)                 return Expr(Op.REAL, (float(obj), kind))
814: (4)               if isinstance(obj, float):
815: (8)                 return Expr(Op.REAL, (obj, kind))
816: (4)               if isinstance(obj, Expr):
817: (8)                 if obj.op is Op.REAL:
818: (12)                   return obj
819: (8)                 elif obj.op is Op.INTEGER:
820: (12)                     return Expr(Op.REAL, (float(obj.data[0]), kind))
821: (4)                 raise OpError(f'cannot convert {obj} to REAL constant')
822: (0)             def as_string(obj, kind=1):
823: (4)               """Return object as STRING expression (string literal constant).
824: (4)
825: (4)               return Expr(Op.STRING, (obj, kind))
826: (0)             def as_array(obj):
827: (4)               """Return object as ARRAY expression (array constant).
828: (4)
829: (4)               if isinstance(obj, Expr):
830: (8)                 obj = obj,
831: (4)               return Expr(Op.ARRAY, obj)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

832: (0)
833: (4)
834: (4)
835: (4)
836: (0)
837: (4)
838: (4)
839: (4)
840: (16)
841: (17)
842: (0)
843: (4)
844: (4)
845: (4)
846: (0)
847: (4)
848: (4)
849: (4)
850: (0)
851: (4)
852: (4)
853: (4)
854: (0)
855: (4)
856: (0)
857: (4)
858: (0)
859: (4)
860: (0)
861: (4)
862: (0)
863: (4)
864: (0)
865: (4)
866: (0)
867: (4)
868: (4)
869: (4)
870: (8)
871: (8)
872: (12)
873: (8)
874: (12)
875: (8)
876: (12)
877: (8)
878: (4)
879: (0)
880: (4)
881: (4)
882: (4)
883: (8)
884: (8)
885: (12)
886: (8)
887: (12)
888: (16)
889: (16)
890: (20)
891: (16)
892: (8)
893: (13)
894: (13)
895: (12)
896: (8)
897: (4)
898: (0)
899: (4)
900: (4)

def as_complex(real, imag=0):
    """Return object as COMPLEX expression (complex literal constant).
    """
    return Expr(Op.COMPLEX, (as_expr(real), as_expr(imag)))

def as_apply(func, *args, **kwargs):
    """Return object as APPLY expression (function call, constructor, etc.).
    """
    return Expr(Op.APPLY,
               (func, tuple(map(as_expr, args)),
                dict((k, as_expr(v)) for k, v in kwargs.items())))

def as_ternary(cond, expr1, expr2):
    """Return object as TERNARY expression (cond?expr1:expr2).
    """
    return Expr(Op.TERNARY, (cond, expr1, expr2))

def as_ref(expr):
    """Return object as referencing expression.
    """
    return Expr(Op.REF, expr)

def as_deref(expr):
    """Return object as dereferencing expression.
    """
    return Expr(Op.DEREF, expr)

def as_eq(left, right):
    return Expr(Op.RELATIONAL, (RelOp.EQ, left, right))

def as_ne(left, right):
    return Expr(Op.RELATIONAL, (RelOp.NE, left, right))

def as_lt(left, right):
    return Expr(Op.RELATIONAL, (RelOp.LT, left, right))

def as_le(left, right):
    return Expr(Op.RELATIONAL, (RelOp.LE, left, right))

def as_gt(left, right):
    return Expr(Op.RELATIONAL, (RelOp.GT, left, right))

def as_ge(left, right):
    return Expr(Op.RELATIONAL, (RelOp.GE, left, right))

def as_terms(obj):
    """Return expression as TERMS expression.
    """
    if isinstance(obj, Expr):
        obj = normalize(obj)
        if obj.op is Op.TERMS:
            return obj
        if obj.op is Op.INTEGER:
            return Expr(Op.TERMS, {as_integer(1, obj.data[1]): obj.data[0]})
        if obj.op is Op.REAL:
            return Expr(Op.TERMS, {as_real(1, obj.data[1]): obj.data[0]})
        return Expr(Op.TERMS, {obj: 1})
    raise OpError(f'cannot convert {type(obj)} to terms Expr')

def as_factors(obj):
    """Return expression as FACTORS expression.
    """
    if isinstance(obj, Expr):
        obj = normalize(obj)
        if obj.op is Op.FACTORS:
            return obj
        if obj.op is Op.TERMS:
            if len(obj.data) == 1:
                (term, coeff), = obj.data.items()
                if coeff == 1:
                    return Expr(Op.FACTORS, {term: 1})
            return Expr(Op.FACTORS, {term: 1, Expr.number(coeff): 1})
        if ((obj.op is Op.APPLY
             and obj.data[0] is ArithOp.DIV
             and not obj.data[2])):
            return Expr(Op.FACTORS, {obj.data[1][0]: 1, obj.data[1][1]: -1})
        return Expr(Op.FACTORS, {obj: 1})
    raise OpError(f'cannot convert {type(obj)} to factors Expr')

def as_term_coeff(obj):
    """Return expression as term-coefficient pair.
    """

```

```

901: (4)           if isinstance(obj, Expr):
902: (8)             obj = normalize(obj)
903: (8)             if obj.op is Op.INTEGER:
904: (12)               return as_integer(1, obj.data[1]), obj.data[0]
905: (8)             if obj.op is Op.REAL:
906: (12)               return as_real(1, obj.data[1]), obj.data[0]
907: (8)             if obj.op is Op.TERMS:
908: (12)               if len(obj.data) == 1:
909: (16)                 (term, coeff), = obj.data.items()
910: (16)                 return term, coeff
911: (8)             if obj.op is Op.APPLY and obj.data[0] is ArithOp.DIV:
912: (12)               t, c = as_term_coeff(obj.data[1][0])
913: (12)               return as_apply(ArithOp.DIV, t, obj.data[1][1]), c
914: (8)             return obj, 1
915: (4)           raise OpError(f'cannot convert {type(obj)} to term and coeff')
916: (0)           def as_numer_denom(obj):
917: (4)             """Return expression as numer-denom pair.
918: (4)
919: (4)             if isinstance(obj, Expr):
920: (8)               obj = normalize(obj)
921: (8)               if obj.op in (Op.INTEGER, Op.REAL, Op.COMPLEX, Op.SYMBOL,
922: (22)                 Op.INDEXING, Op.TERNARY):
923: (12)                 return obj, as_number(1)
924: (8)               elif obj.op is Op.APPLY:
925: (12)                 if obj.data[0] is ArithOp.DIV and not obj.data[2]:
926: (16)                   numer, denom = map(as_numer_denom, obj.data[1])
927: (16)                   return numer[0] * denom[1], numer[1] * denom[0]
928: (12)                 return obj, as_number(1)
929: (8)               elif obj.op is Op.TERMS:
930: (12)                 numer, denom = [], []
931: (12)                 for term, coeff in obj.data.items():
932: (16)                   n, d = as_numer_denom(term)
933: (16)                   n = n * coeff
934: (16)                   numer.append(n)
935: (16)                   denom.append(d)
936: (12)                 numer, denom = as_number(0), as_number(1)
937: (12)                 for i in range(len(numer)):
938: (16)                   n = numer[i]
939: (16)                   for j in range(len(numer)):
940: (20)                     if i != j:
941: (24)                       n *= denom[j]
942: (16)                     numer += n
943: (16)                     denom *= denom[i]
944: (12)                   if denom.op in (Op.INTEGER, Op.REAL) and denom.data[0] < 0:
945: (16)                     numer, denom = -numer, -denom
946: (12)                   return numer, denom
947: (8)               elif obj.op is Op.FACTORS:
948: (12)                 numer, denom = as_number(1), as_number(1)
949: (12)                 for b, e in obj.data.items():
950: (16)                   bnumer, bdenom = as_numer_denom(b)
951: (16)                   if e > 0:
952: (20)                     numer *= bnumer ** e
953: (20)                     denom *= bdenom ** e
954: (16)                   elif e < 0:
955: (20)                     numer *= bdenom ** (-e)
956: (20)                     denom *= bnumer ** (-e)
957: (12)                   return numer, denom
958: (4)           raise OpError(f'cannot convert {type(obj)} to numer and denom')
959: (0)           def _counter():
960: (4)             counter = 0
961: (4)             while True:
962: (8)               counter += 1
963: (8)               yield counter
964: (0)             COUNTER = _counter()
965: (0)             def eliminate_quotes(s):
966: (4)               """Replace quoted substrings of input string.
967: (4)               Return a new string and a mapping of replacements.
968: (4)
969: (4)               d = {}

```

```

970: (4) def repl(m):
971: (8)     kind, value = m.groups()[:2]
972: (8)     if kind:
973: (12)         kind = kind[:-1]
974: (8)         p = {'''": "SINGLE", '''": "DOUBLE"}[value[0]]
975: (8)         k = f'{kind}@__f2py_QUOTES_{p}__{COUNTER.__next__()}@'
976: (8)         d[k] = value
977: (8)         return k
978: (4)     new_s = re.sub(r'({kind}_|)({single_quoted}|{double_quoted})'.format(
979: (8)         kind=r'\w[\w\d_]*',
980: (8)         single_quoted=r"('([^\\\]|(\\\.)*)'",
981: (8)         double_quoted=r'("([^\\\]|(\\\.)*)')",
982: (8)         repl, s)
983: (4)     assert ''' not in new_s
984: (4)     assert '''' not in new_s
985: (4)     return new_s, d
986: (0) def insert_quotes(s, d):
987: (4)     """Inverse of eliminate_quotes.
988: (4) """
989: (4)     for k, v in d.items():
990: (8)         kind = k[:k.find('@')]
991: (8)         if kind:
992: (12)             kind += '_'
993: (8)             s = s.replace(k, kind + v)
994: (4)     return s
995: (0) def replace_parenthesis(s):
996: (4)     """Replace substrings of input that are enclosed in parenthesis.
997: (4)     Return a new string and a mapping of replacements.
998: (4) """
999: (4)     left, right = None, None
1000: (4)     mn_i = len(s)
1001: (4)     for left_, right_ in ((('(', '/'),,
1002: (26)                 ')'),
1003: (26)                 '{}', # to support C literal structs
1004: (26)                 '[]'):
1005: (8)         i = s.find(left_)
1006: (8)         if i == -1:
1007: (12)             continue
1008: (8)         if i < mn_i:
1009: (12)             mn_i = i
1010: (12)             left, right = left_, right_
1011: (4)     if left is None:
1012: (8)         return s, {}
1013: (4)     i = mn_i
1014: (4)     j = s.find(right, i)
1015: (4)     while s.count(left, i + 1, j) != s.count(right, i + 1, j):
1016: (8)         j = s.find(right, j + 1)
1017: (8)         if j == -1:
1018: (12)             raise ValueError(f'Mismatch of {left+right} parenthesis in {s!r}')
1019: (4)     p = {':': 'ROUND', '[': 'SQUARE', '{': 'CURLY', '/': 'ROUNDDIV'}[left]
1020: (4)     k = f'@__f2py_PARENTHESIS_{p}__{COUNTER.__next__()}@'
1021: (4)     v = s[i+len(left):j]
1022: (4)     r, d = replace_parenthesis(s[j+len(right):])
1023: (4)     d[k] = v
1024: (4)     return s[:i] + k + r, d
1025: (0) def _get_parenthesis_kind(s):
1026: (4)     assert s.startswith('__f2py_PARENTHESIS_'), s
1027: (4)     return s.split('_')[4]
1028: (0) def unreplace_parenthesis(s, d):
1029: (4)     """Inverse of replace_parenthesis.
1030: (4) """
1031: (4)     for k, v in d.items():
1032: (8)         p = _get_parenthesis_kind(k)
1033: (8)         left = dict(ROUND='(', SQUARE='[ ', CURLY='{ ', ROUNDDIV='(/')[p]
1034: (8)         right = dict(ROUND=')', SQUARE=' ]', CURLY=' }', ROUNDDIV=' /')[p]
1035: (8)         s = s.replace(k, left + v + right)
1036: (4)     return s
1037: (0) def fromstring(s, language=Language.C):
1038: (4)     """Create an expression from a string.

```

```

1039: (4) This is a "lazy" parser, that is, only arithmetic operations are
1040: (4) resolved, non-arithmetic operations are treated as symbols.
1041: (4)
1042: (4)
1043: (4)
1044: (8)
1045: (4) raise ValueError(f'failed to parse `{s}` to Expr instance: got `'{r}``')
1046: (0)
1047: (4)
1048: (8)
1049: (8)
1050: (4)
1051: (8)
1052: (8)
1053: (12)
1054: (8)
1055: (12)
1056: (8)
1057: (4)
1058: (8)
1059: (0)
1060: (4)
1061: (8)
1062: (8)
1063: (8)
1064: (4)
1065: (8)
1066: (4)
1067: (8)
1068: (8)
1069: (8)
1070: (4)
1071: (8)
1072: (8)
1073: (8)
1074: (8)
1075: (8)
1076: (8)
1077: (8)
1078: (8)
1079: (8)
1080: (12)
1081: (8)
1082: (8)
1083: (8)
1084: (8)
1085: (12)
1086: (16)
1087: (12)
1088: (8)
1089: (12)
1090: (12)
1091: (16)
1092: (12)
1093: (16)
1094: (20)
1095: (12)
1096: (16)
1097: (8)
1098: (8)
1099: (12)
1100: (12)
1101: (12)
1102: (12)
1103: (12)
1104: (12)
1105: (8)
1106: (12)
1107: (16)

    r = _FromStringWorker(language=language).parse(s)
    if isinstance(r, Expr):
        return r
    raise ValueError(f'failed to parse `{s}` to Expr instance: got `'{r}``')

class _Pair:
    def __init__(self, left, right):
        self.left = left
        self.right = right
    def substitute(self, symbols_map):
        left, right = self.left, self.right
        if isinstance(left, Expr):
            left = left.substitute(symbols_map)
        if isinstance(right, Expr):
            right = right.substitute(symbols_map)
        return _Pair(left, right)
    def __repr__(self):
        return f'{type(self).__name__}({self.left}, {self.right})'

class _FromStringWorker:
    def __init__(self, language=Language.C):
        self.original = None
        self.quotes_map = None
        self.language = language
    def finalize_string(self, s):
        return insert_quotes(s, self.quotes_map)
    def parse(self, inp):
        self.original = inp
        unquoted, self.quotes_map = eliminate_quotes(inp)
        return self.process(unquoted)
    def process(self, s, context='expr'):
        """Parse string within the given context.
        The context may define the result in case of ambiguous
        expressions. For instance, consider expressions `f(x, y)` and
        `(x, y) + (a, b)` where `f` is a function and pair `(x, y)`
        denotes complex number. Specifying context as "args" or
        "expr", the subexpression `(x, y)` will be parse to an
        argument list or to a complex number, respectively.
        """
        if isinstance(s, (list, tuple)):
            return type(s)(self.process(s_, context) for s_ in s)
        assert isinstance(s, str), (type(s), s)
        r, raw_symbols_map = replace_parenthesis(s)
        r = r.strip()
        def restore(r):
            if isinstance(r, (list, tuple)):
                return type(r)(map(restore, r))
            return unreplace_parenthesis(r, raw_symbols_map)
        if ',' in r:
            operands = restore(r.split(','))
            if context == 'args':
                return tuple(self.process(operands))
            if context == 'expr':
                if len(operands) == 2:
                    return as_complex(*self.process(operands))
            raise NotImplementedError(
                f'parsing comma-separated list (context={context}): {r}')
        m = re.match(r'\A([^\?]+)[?](?:[^\?]+)(?:(.+)\Z', r)
        if m:
            assert context == 'expr', context
            oper, expr1, expr2 = restore(m.groups())
            oper = self.process(oper)
            expr1 = self.process(expr1)
            expr2 = self.process(expr2)
            return as_ternary(oper, expr1, expr2)
        if self.language is Language.Fortran:
            m = re.match(
                r'\A(.+)\s*[.](eq|ne|lt|le|gt|ge)[.]\s*(.+)\Z', r, re.I)

```

```

1108: (8)
1109: (12)
1110: (16)
1111: (8)
1112: (12)
1113: (12)
1114: (16)
1115: (12)
1116: (12)
1117: (12)
1118: (8)
1119: (8)
1120: (12)
1121: (12)
1122: (12)
1123: (8)
1124: (8)
1125: (12)
1126: (12)
1127: (16)
1128: (16)
1129: (16)
1130: (20)
1131: (16)
1132: (20)
1133: (20)
1134: (12)
1135: (8)
1136: (12)
1137: (12)
1138: (24)
1139: (8)
1140: (28)
1141: (29)
1142: (8)
1143: (12)
1144: (12)
1145: (16)
1146: (28)
1147: (12)
1148: (12)
1149: (16)
1150: (16)
1151: (16)
1152: (20)
1153: (16)
1154: (20)
1155: (20)
1156: (12)
1157: (8)
1158: (12)
1159: (12)
1160: (12)
1161: (8)
1162: (12)
1163: (12)
1164: (12)
1165: (16)
1166: (16)
1167: (12)
1168: (8)
1169: (12)
1170: (12)
1171: (8)
1172: (12)
1173: (12)
1174: (16)
1175: (12)
1176: (8)

        else:
            m = re.match(
                r'\A(.+)\s*([=][=]|![=][=]|<[=]>|[<][=][>])\s*(.+)\Z', r)
        if m:
            left, rop, right = m.groups()
            if self.language is Language.Fortran:
                rop = '.' + rop + '.'
            left, right = self.process	restore(left, right))
            rop = RelOp.fromstring(rop, language=self.language)
            return Expr(Op.RELATIONAL, (rop, left, right))
        m = re.match(r'\A(\w[\w\d_]*)\s*[=](.*))\Z', r)
        if m:
            keyname, value = m.groups()
            value = restore(value)
            return _Pair(keyname, self.process(value))
    operands = re.split(r'((?<!`d[edED])[+-])', r)
    if len(operands) > 1:
        result = self.process(restore(operands[0] or '0'))
        for op, operand in zip(operands[1::2], operands[2::2]):
            operand = self.process(restore(operand))
            op = op.strip()
            if op == '+':
                result += operand
            else:
                assert op == '-'
                result -= operand
    return result
    if self.language is Language.Fortran and '//' in r:
        operands = restore(r.split('//'))
        return Expr(Op.CONCAT,
                    tuple(self.process(operands)))
    operands = re.split(r'(?<=[@\w\d_])\s*([*]//',
                        (r if self.language is Language.C
                         else r.replace('**', '@_f2py_DOUBLE_STAR@'))))
    if len(operands) > 1:
        operands = restore(operands)
        if self.language is not Language.C:
            operands = [operand.replace('@_f2py_DOUBLE_STAR@', '**')
                        for operand in operands]
        result = self.process(operands[0])
        for op, operand in zip(operands[1::2], operands[2::2]):
            operand = self.process(operand)
            op = op.strip()
            if op == '*':
                result *= operand
            else:
                assert op == '/'
                result /= operand
    return result
    if r.startswith('*') or r.startswith('&'):
        op = {'*': Op.DEREF, '&': Op.REF}[r[0]]
        operand = self.process(restore(r[1:]))
        return Expr(op, operand)
    if self.language is not Language.C and '**' in r:
        operands = list(reversed(restore(r.split('**'))))
        result = self.process(operands[0])
        for operand in operands[1:]:
            operand = self.process(operand)
            result = operand ** result
    return result
    m = re.match(r'\A({digit_string})({kind})|\)\Z'.format(
        digit_string=r'\d+',
        kind=r'_(\d+|\w[\w\d_]*)'), r)
    if m:
        value, _, kind = m.groups()
        if kind and kind.isdigit():
            kind = int(kind)
        return as_integer(int(value), kind or 4)
    m = re.match(r'\A({significant})({exponent})|\d+{exponent}\Z', r)

```

```

1: ({kind}|\)\Z'
1177: (21)                                .format(
1178: (25)                                significant=r'[.]\d+|\d+[.]\d*',
1179: (25)                                exponent=r'[edED][+-]?\d+',
1180: (25)                                kind=r'_(\d+|\w[\w\d_]*)'), r)
1181: (8)                                 if m:
1182: (12)                                 value, _, _, kind = m.groups()
1183: (12)                                 if kind and kind.isdigit():
1184: (16)                                 kind = int(kind)
1185: (12)                                 value = value.lower()
1186: (12)                                 if 'd' in value:
1187: (16)                                 return as_real(float(value.replace('d', 'e'))), kind or 8)
1188: (12)                                 return as_real(float(value)), kind or 4)
1189: (8)                                 if r in self.quotes_map:
1190: (12)                                 kind = r[:r.find('@')]
1191: (12)                                 return as_string(self.quotes_map[r], kind or 1)
1192: (8)                                 if r in raw_symbols_map:
1193: (12)                                 paren = _get_parenthesis_kind(r)
1194: (12)                                 items = self.process	restore(raw_symbols_map[r]),
1195: (33)   'expr' if paren == 'ROUND' else 'args')
1196: (12)                                 if paren == 'ROUND':
1197: (16)                                 if isinstance(items, Expr):
1198: (20)                                 return items
1199: (12)                                 if paren in ['ROUNDDIV', 'SQUARE']:
1200: (16)                                 if isinstance(items, Expr):
1201: (20)                                 items = (items,)
1202: (16)                                 return as_array(items)
1203: (8)                                 m = re.match(r'\A(.+)\s*(@_f2py_PARENTHESIS_(ROUND|SQUARE)_\d+@)\Z', r)
1204: (21)                                 if m:
1205: (8)                                 target, args, paren = m.groups()
1206: (12)                                 target = self.process	restore(target))
1207: (12)                                 args = self.process	restore(args)[1:-1], 'args')
1208: (12)                                 if not isinstance(args, tuple):
1209: (12)                                     args = args,
1210: (16)                                 if paren == 'ROUND':
1211: (12)                                     kwargs = dict((a.left, a.right) for a in args
1212: (16)   if isinstance(a, _Pair))
1213: (30)                                     args = tuple(a for a in args if not isinstance(a, _Pair))
1214: (16)                                     return as_apply(target, *args, **kwargs)
1215: (16)                                 else:
1216: (12)                                     assert paren == 'SQUARE'
1217: (16)                                     return target[args]
1218: (16)                                 m = re.match(r'\A\w[\w\d_]*\Z', r)
1219: (8)                                 if m:
1220: (8)                                     return as_symbol(r)
1221: (12)                                 r = self.finalize_string	restore(r))
1222: (8)                                 ewarn(
1223: (8)                                     f'fromstring: treating {r!r} as symbol (original=
1224: (12)   {self.original}))')
1225: (8)                                 return as_symbol(r)

```

---

## File 155 - use\_rules.py:

```

1: (0)                                """
2: (0)                                Build 'use others module data' mechanism for f2py2e.
3: (0)                                Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
4: (0)                                Copyright 2011 -- present NumPy Developers.
5: (0)                                Permission to use, modify, and distribute this software is given under the
6: (0)                                terms of the NumPy License.
7: (0)                                NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
8: (0)                                """
9: (0)                                __version__ = "$Revision: 1.3 $"[10:-1]
10: (0)                                f2py_version = 'See `f2py -v`'
11: (0)                                from .auxfuncs import (
12: (4)                                    applyrules, dictappend, gentitle, hasnote, outmess
13: (0)                                )

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

14: (0) usemodule_rules = {
15: (4)     'body': """
16: (0)     static char doc_#apiname#[ ] = \"
17: (0)         \\\nVariable wrapper signature:\\n\\\
18: (0)         \t #name# = get_#name#()\\n\\
19: (0)         Arguments:\\n\\
20: (0)         extern F_MODFUNC(#usemodulename#, #USEMODULENAME#, #realname#, #REALNAME#);
21: (0)         static PyObject *#apiname#(PyObject *capi_self, PyObject *capi_args) {
22: (0)             /*#decl*/
23: (0)             \tif (!PyArg_ParseTuple(capi_args, \"\")) goto capi_fail;
24: (0)             printf(\"c:
25: (0)             %d\\n\", F_MODFUNC(#usemodulename#, #USEMODULENAME#, #realname#, #REALNAME#));
26: (0)             \treturn Py_BuildValue(\"\");
27: (0)         capi_fail:
28: (0)             \treturn NULL;
29: (0)         }
29: (4)         'method':
30: (4)     '\t{\"get_#name#\", #apiname#, METH_VARARGS|METH_KEYWORDS, doc_#apiname#},',
31: (4)         'need': ['F_MODFUNC']
32: (0)     def buildusevars(m, r):
33: (4)         ret = {}
34: (4)         outmess(
35: (8)             '\t\tBuilding use variable hooks for module "%s" (feature only for
F90/F95)...\\n' % (m['name']))
36: (4)         varsmap = {}
37: (4)         revmap = {}
38: (4)         if 'map' in r:
39: (8)             for k in r['map'].keys():
40: (12)                 if r['map'][k] in revmap:
41: (16)                     outmess('\t\t\tVariable "%s=>%s" is already mapped by "%s".
Skipping.\\n' %
42: (20)                         r['map'][k], k, revmap[r['map'][k]])
43: (12)             else:
44: (16)                 revmap[r['map'][k]] = k
45: (4)             if 'only' in r and r['only']:
46: (8)                 for v in r['map'].keys():
47: (12)                     if r['map'][v] in m['vars']:
48: (16)                         if revmap[r['map'][v]] == v:
49: (20)                             varsmap[v] = r['map'][v]
50: (16)                         else:
51: (20)                             outmess('\t\t\tIgnoring map "%s=>%s". See above.\\n' %
52: (28)                                 (v, r['map'][v]))
53: (12)             else:
54: (16)                 outmess(
55: (20)                     '\t\t\tNo definition for variable "%s=>%s". Skipping.\\n' %
(v, r['map'][v]))
56: (4)             else:
57: (8)                 for v in m['vars'].keys():
58: (12)                     if v in revmap:
59: (16)                         varsmap[v] = revmap[v]
60: (12)                     else:
61: (16)                         varsmap[v] = v
62: (4)                 for v in varsmap.keys():
63: (8)                     ret = dictappend(ret, buildusevar(v, varsmap[v], m['vars'],
m['name']))
64: (4)             return ret
65: (0)     def buildusevar(name, realname, vars, usemodulename):
66: (4)         outmess('\t\t\tConstructing wrapper function for variable "%s=>%s"...\\n' %
(
67: (8)             name, realname))
68: (4)         ret = {}
69: (4)         vrd = {'name': name,
70: (11)             'realname': realname,
71: (11)             'REALNAME': realname.upper(),
72: (11)             'usemodulename': usemodulename,
73: (11)             'USEMODULENAME': usemodulename.upper(),
74: (11)             'texname': name.replace('_', '\\_'),
75: (11)             'begintitle': gentitle('%s=>%s' % (name, realname)),
```

```

76: (11)             'endtitle': gentitle('end of %s=>%s' % (name, realname)),
77: (11)             'apiname': '#modulename#_use_%s_from_%s' % (realname,
usemodulename)
78: (11)         }
79: (4)             nummap = {0: 'Ro', 1: 'Ri', 2: 'Rii', 3: 'Riii', 4: 'Riv',
80: (14)                 5: 'Rv', 6: 'Rvi', 7: 'Rvii', 8: 'Rviii', 9: 'Rix'}
81: (4)             vrd['texnamename'] = name
82: (4)             for i in nummap.keys():
83: (8)                 vrd['texnamename'] = vrd['texnamename'].replace(repr(i), nummap[i])
84: (4)             if hasnote(vars[realname]):
85: (8)                 vrd['note'] = vars[realname]['note']
86: (4)             rd = dictappend({}, vrd)
87: (4)             print(name, realname, vars[realname])
88: (4)             ret = applyrules(usemodule_rules, rd)
89: (4)             return ret
-----
```

## File 156 - \_isocbind.py:

```

1: (0) """
2: (0) ISO_C_BINDING maps for f2py2e.
3: (0) Only required declarations/macros/functions will be used.
4: (0) Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
5: (0) Copyright 2011 -- present NumPy Developers.
6: (0) Permission to use, modify, and distribute this software is given under the
7: (0) terms of the NumPy License.
8: (0) NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
9: (0) """
10: (0) iso_c_binding_map = {
11: (4)     'integer': {
12: (8)         'c_int': 'int',
13: (8)         'c_short': 'short', # 'short' <=> 'int' for now
14: (8)         'c_long': 'long', # 'long' <=> 'int' for now
15: (8)         'c_long_long': 'long_long',
16: (8)         'c_signed_char': 'signed_char',
17: (8)         'c_size_t': 'unsigned', # size_t <=> 'unsigned' for now
18: (8)         'c_int8_t': 'signed_char', # int8_t <=> 'signed_char' for now
19: (8)         'c_int16_t': 'short', # int16_t <=> 'short' for now
20: (8)         'c_int32_t': 'int', # int32_t <=> 'int' for now
21: (8)         'c_int64_t': 'long_long',
22: (8)         'c_int_least8_t': 'signed_char', # int_least8_t <=> 'signed_char' for
now
23: (8)         'c_int_least16_t': 'short', # int_least16_t <=> 'short' for now
24: (8)         'c_int_least32_t': 'int', # int_least32_t <=> 'int' for now
25: (8)         'c_int_least64_t': 'long_long',
26: (8)         'c_int_fast8_t': 'signed_char', # int_fast8_t <=> 'signed_char' for
now
27: (8)         'c_int_fast16_t': 'short', # int_fast16_t <=> 'short' for now
28: (8)         'c_int_fast32_t': 'int', # int_fast32_t <=> 'int' for now
29: (8)         'c_int_fast64_t': 'long_long',
30: (8)         'c_intmax_t': 'long_long', # intmax_t <=> 'long_long' for now
31: (8)         'c_intptr_t': 'long', # intptr_t <=> 'long' for now
32: (8)         'c_ptrdiff_t': 'long', # ptrdiff_t <=> 'long' for now
33: (4)     },
34: (4)     'real': {
35: (8)         'c_float': 'float',
36: (8)         'c_double': 'double',
37: (8)         'c_long_double': 'long_double'
38: (4)     },
39: (4)     'complex': {
40: (8)         'c_float_complex': 'complex_float',
41: (8)         'c_double_complex': 'complex_double',
42: (8)         'c_long_double_complex': 'complex_long_double'
43: (4)     },
44: (4)     'logical': {
45: (8)         'c_bool': 'unsigned_char' # _Bool <=> 'unsigned_char' for now
46: (4)     },
47: (4)     'character': {
```

```

48: (8)           'c_char': 'char'
49: (4)
50: (0)
51: (0)           isoc_c2pycode_map = {}
52: (0)           iso_c2py_map = {}
53: (0)           isoc_kindmap = {}
54: (0)           for fortran_type, c_type_dict in iso_c_binding_map.items():
55: (4)             for c_type in c_type_dict.keys():
56: (8)               isoc_kindmap[c_type] = fortran_type
-----
```

## File 157 - \_src\_pyf.py:

```

1: (0)           import re
2: (0)           """
3: (0)           process_file(filename)
4: (2)             takes templated file .xxx.src and produces .xxx file where .xxx
5: (2)             is .pyf .f90 or .f using the following template rules:
6: (2)             '<..>' denotes a template.
7: (2)             All function and subroutine blocks in a source file with names that
8: (2)             contain '<..>' will be replicated according to the rules in '<..>'.
9: (2)             The number of comma-separated words in '<..>' will determine the number of
10: (2)            replicates.
11: (2)             '<..>' may have two different forms, named and short. For example,
12: (2)            named:
13: (3)               <p=d,s,z,c> where anywhere inside a block '<p>' will be replaced with
14: (3)                 'd', 's', 'z', and 'c' for each replicate of the block.
15: (3)               <_c> is already defined: <_c=s,d,c,z>
16: (3)               <_t> is already defined: <_t=real,double precision,complex,double complex>
17: (2)            short:
18: (3)               <s,d,c,z>, a short form of the named, useful when no <p> appears inside
19: (3)                 a block.
20: (2)             In general, '<..>' contains a comma separated list of arbitrary
21: (2)             expressions. If these expression must contain a comma|leftarrow|rightarrow, then prepend the comma|leftarrow|rightarrow with a backslash.
22: (2)             If an expression matches '\<index>' then it will be replaced
23: (2)               by <index>-th expression.
24: (2)             Note that all '<..>' forms in a block must have the same number of
25: (2)               comma-separated entries.
26: (2)             Predefined named template rules:
27: (1)               <prefix=s,d,c,z>
28: (2)               <ftype=real,double precision,complex,double complex>
29: (2)               <ftypereal=real,double precision,\0,\1>
30: (2)               <cctype=float,double,complex_float,complex_double>
31: (2)               <cctypereal=float,double,\0,\1>
32: (2)
33: (0)               """
34: (0)             routine_start_re = re.compile(r'(\n|\A)((\$\|*)|)\s*'
35: (0)             (subroutine|function)\b', re.I)
36: (0)             routine_end_re = re.compile(r'\n\s*end\s*(subroutine|function)\b.*(\n|\Z)', re.I)
37: (0)             function_start_re = re.compile(r'\n      (\$\|*)\s*function\b', re.I)
38: (4)             def parse_structure(astr):
39: (4)               """ Return a list of tuples for each function or subroutine each
40: (4)                 tuple is the start and end of a subroutine or function to be
41: (4)                   expanded.
42: (4)               """
43: (4)               spanlist = []
44: (4)               ind = 0
45: (4)               while True:
46: (8)                 m = routine_start_re.search(astr, ind)
47: (8)                 if m is None:
48: (12)                   break
49: (8)                 start = m.start()
50: (8)                 if function_start_re.match(astr, start, m.end()):
51: (12)                   while True:
52: (16)                     i = astr.rfind('\n', ind, start)
53: (16)                     if i== -1:
54: (20)                       break
```

```

54: (16)                      start = i
55: (16)                      if astr[i:i+7]!='\n      $':
56: (20)                          break
57: (8)                      start += 1
58: (8)                      m = routine_end_re.search(astr, m.end())
59: (8)                      ind = end = m and m.end()-1 or len(astr)
60: (8)                      spanlist.append((start, end))
61: (4)                      return spanlist
62: (0) template_re = re.compile(r"<\s*(\w[\w\d]*)\s*>")
63: (0) named_re = re.compile(r"<\s*(\w[\w\d]*)\s*=\s*(.*?)\s*>")
64: (0) list_re = re.compile(r"<\s*(.*?)\s*>")
65: (0) def find_repl_patterns(astr):
66: (4)     reps = named_re.findall(astr)
67: (4)     names = {}
68: (4)     for rep in reps:
69: (8)         name = rep[0].strip() or unique_key(names)
70: (8)         repl = rep[1].replace(r'\,', '@comma@')
71: (8)         thelist = conv(repl)
72: (8)         names[name] = thelist
73: (4)     return names
74: (0) def find_and_remove_repl_patterns(astr):
75: (4)     names = find_repl_patterns(astr)
76: (4)     astr = re.subn(named_re, '', astr)[0]
77: (4)     return astr, names
78: (0) item_re = re.compile(r"\A\\(?P<index>\d+)\Z")
79: (0) def conv(astr):
80: (4)     b = astr.split(',')
81: (4)     l = [x.strip() for x in b]
82: (4)     for i in range(len(l)):
83: (8)         m = item_re.match(l[i])
84: (8)         if m:
85: (12)             j = int(m.group('index'))
86: (12)             l[i] = l[j]
87: (4)     return ','.join(l)
88: (0) def unique_key(adict):
89: (4)     """ Obtain a unique key given a dictionary."""
90: (4)     allkeys = list(adict.keys())
91: (4)     done = False
92: (4)     n = 1
93: (4)     while not done:
94: (8)         newkey = '__%s' % (n)
95: (8)         if newkey in allkeys:
96: (12)             n += 1
97: (8)         else:
98: (12)             done = True
99: (4)     return newkey
100: (0) template_name_re = re.compile(r'\A\s*(\w[\w\d]*)\s*\Z')
101: (0) def expand_sub(substr, names):
102: (4)     substr = substr.replace(r'\>', '@rightarrow@')
103: (4)     substr = substr.replace(r'\<', '@leftarrow@')
104: (4)     lnames = find_repl_patterns(substr)
105: (4)     substr = named_re.sub(r"<\1>", substr) # get rid of definition templates
106: (4)     def listrepl(mobj):
107: (8)         thelist = conv(mobj.group(1).replace(r'\,', '@comma@'))
108: (8)         if template_name_re.match(thelist):
109: (12)             return "<%s>" % (thelist)
110: (8)         name = None
111: (8)         for key in lnames.keys(): # see if list is already in dictionary
112: (12)             if lnames[key] == thelist:
113: (16)                 name = key
114: (8)         if name is None: # this list is not in the dictionary yet
115: (12)             name = unique_key(lnames)
116: (12)             lnames[name] = thelist
117: (8)         return "<%s>" % name
118: (4)     substr = list_re.sub(listrepl, substr) # convert all lists to named
119: (4)     templates
120: (4)     numsubs = None
121: (4)     base_rule = None
122: (4)     rules = {}

```

```

122: (4)             for r in template_re.findall(substr):
123: (8)                 if r not in rules:
124: (12)                     thelist = lnames.get(r, names.get(r, None))
125: (12)                     if thelist is None:
126: (16)                         raise ValueError('No replicates found for <%s> % (r)')
127: (12)                     if r not in names and not thelist.startswith('_'):
128: (16)                         names[r] = thelist
129: (12)                     rule = [i.replace('@comma@', ',') for i in thelist.split(',')]
130: (12)                     num = len(rule)
131: (12)                     if numsubs is None:
132: (16)                         numsubs = num
133: (16)                         rules[r] = rule
134: (16)                         base_rule = r
135: (12)                     elif num == numsubs:
136: (16)                         rules[r] = rule
137: (12)                     else:
138: (16)                         print("Mismatch in number of replacements (base <{}={}>) "
139: (22)                             "for <{}={}>. Ignoring.".format(base_rule,
', '.join(rules[base_rule])), r, thelist))
140: (4)                     if not rules:
141: (8)                         return substr
142: (4)             def namerepl(mobj):
143: (8)                 name = mobj.group(1)
144: (8)                 return rules.get(name, (k+1)*[name])[k]
145: (4)             newstr = ''
146: (4)             for k in range(numsubs):
147: (8)                 newstr += template_re.sub(namerepl, substr) + '\n\n'
148: (4)             newstr = newstr.replace('@rightarrow@', '>')
149: (4)             newstr = newstr.replace('@leftarrow@', '<')
150: (4)             return newstr
151: (0)             def process_str(allstr):
152: (4)                 newstr = allstr
153: (4)                 writestr = ''
154: (4)                 struct = parse_structure(newstr)
155: (4)                 oldend = 0
156: (4)                 names = {}
157: (4)                 names.update(_special_names)
158: (4)                 for sub in struct:
159: (8)                     cleanedstr, defs =
find_and_remove_repl_patterns(newstr[oldend:sub[0]])
160: (8)                     writestr += cleanedstr
161: (8)                     names.update(defs)
162: (8)                     writestr += expand_sub(newstr[sub[0]:sub[1]], names)
163: (8)                     oldend = sub[1]
164: (4)                     writestr += newstr[oldend:]
165: (4)                     return writestr
166: (0)             include_src_re = re.compile(r"(\n|\A)\s*include\s*['"](?P<name>
[\w\d./\\]+\.src)[['"]", re.I)
167: (0)             def resolve_includes(source):
168: (4)                 d = os.path.dirname(source)
169: (4)                 with open(source) as fid:
170: (8)                     lines = []
171: (8)                     for line in fid:
172: (12)                         m = include_src_re.match(line)
173: (12)                         if m:
174: (16)                             fn = m.group('name')
175: (16)                             if not os.path.isabs(fn):
176: (20)                                 fn = os.path.join(d, fn)
177: (16)                             if os.path.isfile(fn):
178: (20)                                 lines.extend(resolve_includes(fn))
179: (16)                             else:
180: (20)                                 lines.append(line)
181: (12)                         else:
182: (16)                             lines.append(line)
183: (4)             return lines
184: (0)             def process_file(source):
185: (4)                 lines = resolve_includes(source)
186: (4)                 return process_str(''.join(lines))
187: (0)             _special_names = find_repl_patterns('')

```

```

188: (0)          <_c=s,d,c,z>
189: (0)          <_t=real,double precision,complex,double complex>
190: (0)          <prefix=s,d,c,z>
191: (0)          <ftype=real,double precision,complex,double complex>
192: (0)          <cotype=float,double,complex_float,complex_double>
193: (0)          <ftypereal=real,double precision,\0,\1>
194: (0)          <ctypereal=float,double,\0,\1>
195: (0)          ''')
-----
```

## File 158 - \_\_init\_\_.py:

```

1: (0)          """Fortran to Python Interface Generator.
2: (0)          Copyright 1999 -- 2011 Pearu Peterson all rights reserved.
3: (0)          Copyright 2011 -- present NumPy Developers.
4: (0)          Permission to use, modify, and distribute this software is given under the
terms
5: (0)          of the NumPy License.
6: (0)          NO WARRANTY IS EXPRESSED OR IMPLIED. USE AT YOUR OWN RISK.
7: (0)
8: (0)          """
9: (0)          __all__ = ['run_main', 'compile', 'get_include']
10: (0)         import sys
11: (0)         import subprocess
12: (0)         import os
13: (0)         import warnings
14: (0)         from numpy.exceptions import VisibleDeprecationWarning
15: (0)         from . import f2py2e
16: (0)         from . import diagnose
17: (0)         run_main = f2py2e.run_main
18: (0)         main = f2py2e.main
19: (0)         def compile(source,
20: (12)            modulename='untitled',
21: (12)            extra_args='',
22: (12)            verbose=True,
23: (12)            source_fn=None,
24: (12)            extension='.f',
25: (12)            full_output=False
25: (11)        ):
26: (4)
27: (4)          """
27: (4)          Build extension module from a Fortran 77 source string with f2py.
28: (4)          Parameters
29: (4)          -----
30: (4)          source : str or bytes
31: (8)              Fortran source of module / subroutine to compile
32: (8)              .. versionchanged:: 1.16.0
33: (11)                  Accept str as well as bytes
34: (4)          modulename : str, optional
35: (8)              The name of the compiled python module
36: (4)          extra_args : str or list, optional
37: (8)              Additional parameters passed to f2py
38: (8)              .. versionchanged:: 1.16.0
39: (12)                  A list of args may also be provided.
40: (4)          verbose : bool, optional
41: (8)              Print f2py output to screen
42: (4)          source_fn : str, optional
43: (8)              Name of the file where the fortran source is written.
44: (8)              The default is to use a temporary file with the extension
45: (8)                  provided by the ``extension`` parameter
46: (4)          extension : ``{'.f', '.f90'}``, optional
47: (8)              Filename extension if `source_fn` is not provided.
48: (8)              The extension tells which fortran standard is used.
49: (8)              The default is ``.f``, which implies F77 standard.
50: (8)              .. versionadded:: 1.11.0
51: (4)          full_output : bool, optional
52: (8)              If True, return a `subprocess.CompletedProcess` containing
53: (8)                  the stdout and stderr of the compile process, instead of just
54: (8)                  the status code.
55: (8)              .. versionadded:: 1.20.0

```

```

56: (4)          Returns
57: (4)          -----
58: (4)          result : int or `subprocess.CompletedProcess`
59: (8)          0 on success, or a `subprocess.CompletedProcess` if
60: (8)          ``full_output=True``
61: (4)          Examples
62: (4)          -----
63: (4)          .. literalinclude:: ../../source/f2py/code/results/compile_session.dat
64: (8)          :language: python
65: (4)          """
66: (4)          import tempfile
67: (4)          import shlex
68: (4)          if source_fn is None:
69: (8)              f, fname = tempfile.mkstemp(suffix=extension)
70: (8)              os.close(f)
71: (4)          else:
72: (8)              fname = source_fn
73: (4)          if not isinstance(source, str):
74: (8)              source = str(source, 'utf-8')
75: (4)          try:
76: (8)              with open(fname, 'w') as f:
77: (12)                  f.write(source)
78: (8)              args = ['-c', '-m', modulename, f.name]
79: (8)              if isinstance(extra_args, str):
80: (12)                  is_posix = (os.name == 'posix')
81: (12)                  extra_args = shlex.split(extra_args, posix=is_posix)
82: (8)              args.extend(extra_args)
83: (8)              c = [sys.executable,
84: (13)                  '-c',
85: (13)                  'import numpy.f2py as f2py2e;f2py2e.main()'] + args
86: (8)              try:
87: (12)                  cp = subprocess.run(c, capture_output=True)
88: (8)              except OSError:
89: (12)                  cp = subprocess.CompletedProcess(c, 127, stdout=b'', stderr=b'')
90: (8)              else:
91: (12)                  if verbose:
92: (16)                      print(cp.stdout.decode())
93: (4)          finally:
94: (8)              if source_fn is None:
95: (12)                  os.remove(fname)
96: (4)          if full_output:
97: (8)              return cp
98: (4)          else:
99: (8)              return cp.returncode
100: (0)         def get_include():
101: (4)             """
102: (4)                 Return the directory that contains the ``fortranobject.c`` and ``.h``
103: (4)                 files.
104: (8)                 .. note::
105: (8)                     This function is not needed when building an extension with
106: (8)                     `numpy.distutils` directly from ``.f`` and/or ``.pyf`` files
107: (4)                     in one go.
108: (4)                     Python extension modules built with f2py-generated code need to use
109: (4)                     ``fortranobject.c`` as a source file, and include the ``fortranobject.h``
110: (4)                     header. This function can be used to obtain the directory containing
111: (4)                     both of these files.
112: (4)                     Returns
113: (4)                     -----
114: (8)                     include_path : str
115: (8)                         Absolute path to the directory containing ``fortranobject.c`` and
116: (4)                         ``fortranobject.h``.
117: (4)                     Notes
118: (4)                     -----
119: (4)                     .. versionadded:: 1.21.1
120: (4)                     Unless the build system you are using has specific support for f2py,
121: (4)                     building a Python extension using a ``.pyf`` signature file is a two-step
122: (4)                     process. For a module ``mymod``:
123: (6)                         * Step 1: run ``python -m numpy.f2py mymod.pyf --quiet``. This
123: (6)                             generates ``_mymodmodule.c`` and (if needed)

```

```

124: (6)          ``_fblas-f2pywrappers.f`` files next to ``mymod.pyf``.
125: (4)          * Step 2: build your Python extension module. This requires the
126: (6)          following source files:
127: (6)          * ``_mymodmodule.c``
128: (6)          * ``_mymod-f2pywrappers.f`` (if it was generated in Step 1)
129: (6)          * ``fortranobject.c``
130: (4)          See Also
131: (4)          -----
132: (4)          numpy.get_include : function that returns the numpy include directory
133: (4)          """
134: (4)          return os.path.join(os.path.dirname(__file__), 'src')
135: (0)          def __getattr__(attr):
136: (4)              if attr == "test":
137: (8)                  from numpy._pytesttester import PytestTester
138: (8)                  test = PytestTester(__name__)
139: (8)                  return test
140: (4)              else:
141: (8)                  raise AttributeError("module {!r} has no attribute "
142: (30)                                "{}!".format(__name__, attr))
143: (0)          def __dir__():
144: (4)              return list(globals().keys() | {"test"})

```

-----

## File 159 - \_\_main\_\_.py:

```

1: (0)          from numpy.f2py.f2py2e import main
2: (0)          main()

```

-----

## File 160 - \_\_version\_\_.py:

```

1: (0)          from numpy.version import version

```

-----

## File 161 - test\_abstract\_interface.py:

```

1: (0)          from pathlib import Path
2: (0)          import pytest
3: (0)          import textwrap
4: (0)          from . import util
5: (0)          from numpy.f2py import crackfortran
6: (0)          from numpy.testing import IS_WASM
7: (0)          @pytest.mark.skipif(IS_WASM, reason="Cannot start subprocess")
8: (0)          class TestAbstractInterface(util.F2PyTest):
9: (4)              sources = [util.getpath("tests", "src", "abstract_interface", "foo.f90")]
10: (4)             skip = ["add1", "add2"]
11: (4)             def test_abstract_interface(self):
12: (8)                 assert self.module.ops_module.foo(3, 5) == (8, 13)
13: (4)             def test_parse_abstract_interface(self):
14: (8)                 fpath = util.getpath("tests", "src", "abstract_interface",
15: (29)                               "gh18403_mod.f90")
16: (8)                 mod = crackfortran.crackfortran([str(fpath)])
17: (8)                 assert len(mod) == 1
18: (8)                 assert len(mod[0]["body"]) == 1
19: (8)                 assert mod[0]["body"][0]["block"] == "abstract interface"

```

-----

## File 162 - test\_array\_from\_pyobj.py:

```

1: (0)          import os
2: (0)          import sys
3: (0)          import copy
4: (0)          import platform
5: (0)          import pytest
6: (0)          import numpy as np

```

```

7: (0)    from numpy.testing import assert_, assert_equal
8: (0)    from numpy.core.multiarray import typeinfo as _typeinfo
9: (0)    from . import util
10: (0)   wrap = None
11: (0)   _ti = _typeinfo['STRING']
12: (0)   typeinfo = dict(
13: (4)     CHARACTER=type(_ti)((‘c’, _ti.num, 8, _ti.alignment, _ti.type)),
14: (4)     **_typeinfo)
15: (0)   def setup_module():
16: (4)     """
17: (4)       Build the required testing extension module
18: (4)     """
19: (4)       global wrap
20: (4)       if not util.has_c_compiler():
21: (8)         pytest.skip("No C compiler available")
22: (4)       if wrap is None:
23: (8)         config_code = """
24: (8)           config.add_extension('test_array_from_pyobj_ext',
25: (29)               sources=['wrapmodule.c', 'fortranobject.c'],
26: (29)               define_macros=[])

27: (8)         """
28: (8)         d = os.path.dirname(__file__)
29: (8)         src = [
30: (12)           util.getpath("tests", "src", "array_from_pyobj", "wrapmodule.c"),
31: (12)           util.getpath("src", "fortranobject.c"),
32: (12)           util.getpath("src", "fortranobject.h"),
33: (8)         ]
34: (8)         wrap = util.build_module_distutils(src, config_code,
35: (43)             "test_array_from_pyobj_ext")
36: (0)   def flags_info(arr):
37: (4)     flags = wrap.array_attrs(arr)[6]
38: (4)     return flags2names(flags)
39: (0)   def flags2names(flags):
40: (4)     info = []
41: (4)     for flagname in [
42: (12)       "CONTIGUOUS",
43: (12)       "FORTRAN",
44: (12)       "OWNDATA",
45: (12)       "ENSURECOPY",
46: (12)       "ENSUREARRAY",
47: (12)       "ALIGNED",
48: (12)       "NOTSWAPPED",
49: (12)       "WRITEABLE",
50: (12)       "WRITEBACKIFCOPY",
51: (12)       "UPDATEIFCOPY",
52: (12)       "BEHAVED",
53: (12)       "BEHAVED_RO",
54: (12)       "CARRAY",
55: (12)       "FARRAY",
56: (4)     ]:
57: (8)       if abs(flags) & getattr(wrap, flagname, 0):
58: (12)         info.append(flagname)
59: (4)     return info
60: (0)   class Intent:
61: (4)     def __init__(self, intent_list=[]):
62: (8)       self.intent_list = intent_list[:]
63: (8)       flags = 0
64: (8)       for i in intent_list:
65: (12)         if i == "optional":
66: (16)           flags |= wrap.F2PY_OPTIONAL
67: (12)         else:
68: (16)           flags |= getattr(wrap, "F2PY_INTENT_" + i.upper())
69: (8)       self.flags = flags
70: (4)     def __getattr__(self, name):
71: (8)       name = name.lower()
72: (8)       if name == "in__":
73: (12)         name = "in"
74: (8)       return self.__class__(self.intent_list + [name])
75: (4)     def __str__(self):

```

```

76: (8)             return "intent(%s)" % (",".join(self.intent_list))
77: (4)         def __repr__(self):
78: (8)             return "Intent(%r)" % (self.intent_list)
79: (4)         def is_intent(self, *names):
80: (8)             for name in names:
81: (12)                 if name not in self.intent_list:
82: (16)                     return False
83: (8)             return True
84: (4)         def is_intent_exact(self, *names):
85: (8)             return len(self.intent_list) == len(names) and self.is_intent(*names)
86: (0)     intent = Intent()
87: (0)     _type_names = [
88: (4)         "BOOL",
89: (4)         "BYTE",
90: (4)         "UBYTE",
91: (4)         "SHORT",
92: (4)         "USHORT",
93: (4)         "INT",
94: (4)         "UINT",
95: (4)         "LONG",
96: (4)         "ULONG",
97: (4)         "LONGLONG",
98: (4)         "ULLONG",
99: (4)         "FLOAT",
100: (4)        "DOUBLE",
101: (4)        "CFLOAT",
102: (4)        "STRING1",
103: (4)        "STRING5",
104: (4)        "CHARACTER",
105: (0)
106: (0)    _cast_dict = {"BOOL": ["BOOL"]}
107: (0)    _cast_dict["BYTE"] = _cast_dict["BOOL"] + ["BYTE"]
108: (0)    _cast_dict["UBYTE"] = _cast_dict["BOOL"] + ["UBYTE"]
109: (0)    _cast_dict["BYTE"] = ["BYTE"]
110: (0)    _cast_dict["UBYTE"] = ["UBYTE"]
111: (0)    _cast_dict["SHORT"] = _cast_dict["BYTE"] + ["UBYTE", "SHORT"]
112: (0)    _cast_dict["USHORT"] = _cast_dict["UBYTE"] + ["BYTE", "USHORT"]
113: (0)    _cast_dict["INT"] = _cast_dict["SHORT"] + ["USHORT", "INT"]
114: (0)    _cast_dict["UINT"] = _cast_dict["USHORT"] + ["SHORT", "UINT"]
115: (0)    _cast_dict["LONG"] = _cast_dict["INT"] + ["LONG"]
116: (0)    _cast_dict["ULONG"] = _cast_dict["UINT"] + ["ULONG"]
117: (0)    _cast_dict["LONGLONG"] = _cast_dict["LONG"] + ["LONGLONG"]
118: (0)    _cast_dict["ULLONG"] = _cast_dict["ULONG"] + ["ULLONG"]
119: (0)    _cast_dict["FLOAT"] = _cast_dict["SHORT"] + ["USHORT", "FLOAT"]
120: (0)    _cast_dict["DOUBLE"] = _cast_dict["INT"] + ["UINT", "FLOAT", "DOUBLE"]
121: (0)    _cast_dict["CFLOAT"] = _cast_dict["FLOAT"] + ["CFLOAT"]
122: (0)    _cast_dict['STRING1'] = ['STRING1']
123: (0)    _cast_dict['STRING5'] = ['STRING5']
124: (0)    _cast_dict['CHARACTER'] = ['CHARACTER']
125: (0)    if ((np.intp().dtype.itemsize != 4 or np.clongdouble().dtype.alignment <= 8)
126: (8)        and sys.platform != "win32"
127: (8)        and (platform.system(), platform.processor()) != ("Darwin", "arm")):
128: (4)        _type_names.extend(["LONGDOUBLE", "CDOUBLE", "CLONGDOUBLE"])
129: (4)        _cast_dict["LONGDOUBLE"] = _cast_dict["LONG"] + [
130: (8)            "ULONG",
131: (8)            "FLOAT",
132: (8)            "DOUBLE",
133: (8)            "LONGDOUBLE",
134: (4)
135: (4)            _cast_dict["CLONGDOUBLE"] = _cast_dict["LONGDOUBLE"] + [
136: (8)                "CFLOAT",
137: (8)                "CDOUBLE",
138: (8)                "CLONGDOUBLE",
139: (4)
140: (4)                _cast_dict["CDOUBLE"] = _cast_dict["DOUBLE"] + ["CFLOAT", "CDOUBLE"]
141: (0)
142: (4)            _type_cache = {}
143: (4)            def __new__(cls, name):
144: (8)                if isinstance(name, np.dtype):

```

```

145: (12)           dtype0 = name
146: (12)           name = None
147: (12)           for n, i in typeinfo.items():
148: (16)             if not isinstance(i, type) and dtype0.type is i.type:
149: (20)               name = n
150: (20)               break
151: (8)           obj = cls._type_cache.get(name.upper(), None)
152: (8)           if obj is not None:
153: (12)             return obj
154: (8)           obj = object.__new__(cls)
155: (8)           obj.__init__(name)
156: (8)           cls._type_cache[name.upper()] = obj
157: (8)           return obj
158: (4)           def __init__(self, name):
159: (8)             self.NAME = name.upper()
160: (8)             if self.NAME == 'CHARACTER':
161: (12)               info = typeinfo[self.NAME]
162: (12)               self.type_num = getattr(wrap, 'NPY_STRING')
163: (12)               self.elsize = 1
164: (12)               self.dtype = np.dtype('c')
165: (8)             elif self.NAME.startswith('STRING'):
166: (12)               info = typeinfo[self.NAME[:6]]
167: (12)               self.type_num = getattr(wrap, 'NPY_STRING')
168: (12)               self.elsize = int(self.NAME[6:] or 0)
169: (12)               self.dtype = np.dtype(f'S{self.elsize}')
170: (8)             else:
171: (12)               info = typeinfo[self.NAME]
172: (12)               self.type_num = getattr(wrap, 'NPY_' + self.NAME)
173: (12)               self.elsize = info.bits // 8
174: (12)               self.dtype = np.dtype(info.type)
175: (8)             assert self.type_num == info.num
176: (8)             self.type = info.type
177: (8)             self.dtypechar = info.char
178: (4)           def __repr__(self):
179: (8)             return f"Type({self.NAME})|type_num={self.type_num},"
180: (16)             f" dtype={self.dtype},"
181: (16)             f" type={self.type}, elsize={self.elsize},"
182: (16)             f" dtypechar={self.dtypechar})"
183: (4)           def cast_types(self):
184: (8)             return [self.__class__(_m) for _m in _cast_dict[self.NAME]]
185: (4)           def all_types(self):
186: (8)             return [self.__class__(_m) for _m in _type_names]
187: (4)           def smaller_types(self):
188: (8)             bits = typeinfo[self.NAME].alignment
189: (8)             types = []
190: (8)             for name in _type_names:
191: (12)               if typeinfo[name].alignment < bits:
192: (16)                 types.append(Type(name))
193: (8)             return types
194: (4)           def equal_types(self):
195: (8)             bits = typeinfo[self.NAME].alignment
196: (8)             types = []
197: (8)             for name in _type_names:
198: (12)               if name == self.NAME:
199: (16)                 continue
200: (12)               if typeinfo[name].alignment == bits:
201: (16)                 types.append(Type(name))
202: (8)             return types
203: (4)           def larger_types(self):
204: (8)             bits = typeinfo[self.NAME].alignment
205: (8)             types = []
206: (8)             for name in _type_names:
207: (12)               if typeinfo[name].alignment > bits:
208: (16)                 types.append(Type(name))
209: (8)             return types
210: (0)           class Array:
211: (4)             def __repr__(self):
212: (8)               return (f'Array({self.type}, {self.dims}, {self.intent},'
213: (16)                 f' {self.obj})|arr={self.arr}')
```

```

214: (4)
215: (8)
216: (8)
217: (8)
218: (8)
219: (8)
220: (8)
221: (29)
222: (29)
223: (8)
224: (8)
225: (8)
226: (12)
227: (16)
228: (16)
229: (16)
230: (16)
231: (12)
232: (16)
233: (16)
234: (16)
235: (16)
236: (8)
237: (12)
238: (12)
239: (12)
240: (8)
241: (12)
242: (12)
243: (8)
244: (12)
245: (16)
246: (16)
247: (12)
248: (12)
249: (8)
250: (8)
251: (8)
252: (8)
253: (12)
254: (16)
255: (16)
256: (16)
257: (12)
258: (16)
259: (16)
260: (16)
261: (8)
262: (8)
263: (8)
264: (12)
265: (16)
266: (16)
267: (16)
268: (16)
269: (12)
270: (8)
271: (12)
272: (12)
273: (8)
274: (12)
275: (12)
276: (12)
277: (12)
278: (12)
279: (8)
280: (8)
281: (12)
282: (8)

        def __init__(self, typ, dims, intent, obj):
            self.type = typ
            self.dims = dims
            self.intent = intent
            self.obj_copy = copy.deepcopy(obj)
            self.obj = obj
            self.arr = wrap.call(typ.type_num,
                                typ.elsize,
                                dims, intent.flags, obj)
            assert isinstance(self.arr, np.ndarray)
            self.arr_attr = wrap.array_attrs(self.arr)
            if len(dims) > 1:
                if self.intent.is_intent("c"):
                    assert (intent.flags & wrap.F2PY_INTENT_C)
                    assert not self.arr.flags["FORTRAN"]
                    assert self.arr.flags["CONTIGUOUS"]
                    assert (not self.arr_attr[6] & wrap.FORTAN)
                else:
                    assert (not intent.flags & wrap.F2PY_INTENT_C)
                    assert self.arr.flags["FORTRAN"]
                    assert not self.arr.flags["CONTIGUOUS"]
                    assert (self.arr_attr[6] & wrap.FORTAN)
            if obj is None:
                self.pyarr = None
                self.pyarr_attr = None
                return
            if intent.is_intent("cache"):
                assert isinstance(obj, np.ndarray), repr(type(obj))
                self.pyarr = np.array(obj).reshape(*dims).copy()
            else:
                self.pyarr = np.array(
                    np.array(obj, dtype=typ.dtypechar).reshape(*dims),
                    order=self.intent.is_intent("c") and "C" or "F",
                )
                assert self.pyarr.dtype == typ
            self.pyarr.setflags(write=self.arr.flags["WRITEABLE"])
            assert self.pyarr.flags["OWNDATA"], (obj, intent)
            self.pyarr_attr = wrap.array_attrs(self.pyarr)
            if len(dims) > 1:
                if self.intent.is_intent("c"):
                    assert not self.pyarr.flags["FORTRAN"]
                    assert self.pyarr.flags["CONTIGUOUS"]
                    assert (not self.pyarr_attr[6] & wrap.FORTAN)
                else:
                    assert self.pyarr.flags["FORTRAN"]
                    assert not self.pyarr.flags["CONTIGUOUS"]
                    assert (self.pyarr_attr[6] & wrap.FORTAN)
            assert self.arr_attr[1] == self.pyarr_attr[1] # nd
            assert self.arr_attr[2] == self.pyarr_attr[2] # dimensions
            if self.arr_attr[1] <= 1:
                assert self.arr_attr[3] == self.pyarr_attr[3], repr((
                    self.arr_attr[3],
                    self.pyarr_attr[3],
                    self.arr.tobytes(),
                    self.pyarr.tobytes(),
                )) # strides
            assert self.arr_attr[5][-2:] == self.pyarr_attr[5][-2:], repr((
                self.arr_attr[5], self.pyarr_attr[5]
            )) # descr
            assert self.arr_attr[6] == self.pyarr_attr[6], repr((
                self.arr_attr[6],
                self.pyarr_attr[6],
                flags2names(0 * self.arr_attr[6] - self.pyarr_attr[6]),
                flags2names(self.arr_attr[6]),
                intent,
            )) # flags
            if intent.is_intent("cache"):
                assert self.arr_attr[5][3] >= self.type.elsize
            else:

```

```

283: (12)             assert self.arr_attr[5][3] == self.type.elsize
284: (12)             assert (self.arr_equal(self.pyarr, self.arr))
285: (8)              if isinstance(self.obj, np.ndarray):
286: (12)                  if typ.elsize == Type(obj.dtype).elsize:
287: (16)                      if not intent.is_intent("copy") and self.arr_attr[1] <= 1:
288: (20)                          assert self.has_shared_memory()
289: (4)               def arr_equal(self, arr1, arr2):
290: (8)                   if arr1.shape != arr2.shape:
291: (12)                       return False
292: (8)                   return (arr1 == arr2).all()
293: (4)               def __str__(self):
294: (8)                   return str(self.arr)
295: (4)               def has_shared_memory(self):
296: (8)                   """Check that created array shares data with input array."""
297: (8)                   if self.obj is self.arr:
298: (12)                       return True
299: (8)                   if not isinstance(self.obj, np.ndarray):
300: (12)                       return False
301: (8)                   obj_attr = wrap.array_attrs(self.obj)
302: (8)                   return obj_attr[0] == self.arr_attr[0]
303: (0)               class TestIntent:
304: (4)                   def test_in_out(self):
305: (8)                       assert str(intent.in_.out) == "intent(in,out)"
306: (8)                       assert intent.in_.c.is_intent("c")
307: (8)                       assert not intent.in_.c.is_intent_exact("c")
308: (8)                       assert intent.in_.c.is_intent_exact("c", "in")
309: (8)                       assert intent.in_.c.is_intent_exact("in", "c")
310: (8)                       assert not intent.in_.is_intent("c")
311: (0)               class TestSharedMemory:
312: (4)                   @pytest.fixture(autouse=True, scope="class", params=_type_names)
313: (4)                   def setup_type(self, request):
314: (8)                       request.cls.type = Type(request.param)
315: (8)                       request.cls.array = lambda self, dims, intent, obj: Array(
316: (12)                           Type(request.param), dims, intent, obj)
317: (4)                   @property
318: (4)                   def num2seq(self):
319: (8)                       if self.type.NAME.startswith('STRING'):
320: (12)                           elsize = self.type.elsize
321: (12)                           return ['1' * elsize, '2' * elsize]
322: (8)                           return [1, 2]
323: (4)                   @property
324: (4)                   def num23seq(self):
325: (8)                       if self.type.NAME.startswith('STRING'):
326: (12)                           elsize = self.type.elsize
327: (12)                           return [['1' * elsize, '2' * elsize, '3' * elsize],
328: (20)                               ['4' * elsize, '5' * elsize, '6' * elsize]]
329: (8)                           return [[1, 2, 3], [4, 5, 6]]
330: (4)                   def test_in_from_2seq(self):
331: (8)                       a = self.array([2], intent.in_, self.num2seq)
332: (8)                       assert not a.has_shared_memory()
333: (4)                   def test_in_from_2casttype(self):
334: (8)                       for t in self.type.cast_types():
335: (12)                           obj = np.array(self.num2seq, dtype=t.dtype)
336: (12)                           a = self.array([len(self.num2seq)], intent.in_, obj)
337: (12)                           if t.elsize == self.type.elsize:
338: (16)                               assert a.has_shared_memory(), repr((self.type.dtype, t.dtype))
339: (12)                           else:
340: (16)                               assert not a.has_shared_memory()
341: (4)                   @pytest.mark.parametrize("write", ["w", "ro"])
342: (4)                   @pytest.mark.parametrize("order", ["C", "F"])
343: (4)                   @pytest.mark.parametrize("inp", ["2seq", "23seq"])
344: (4)                   def test_in_nocopy(self, write, order, inp):
345: (8)                       """Test if intent(in) array can be passed without copies"""
346: (8)                       seq = getattr(self, "num" + inp)
347: (8)                       obj = np.array(seq, dtype=self.type.dtype, order=order)
348: (8)                       obj.setflags(write=(write == 'w'))
349: (8)                       a = self.array(obj.shape,
350: (23)                           ((order == 'C' and intent.in_.c) or intent.in_), obj)
351: (8)                       assert a.has_shared_memory()

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

352: (4)
353: (8)
354: (8)
355: (8)
356: (8)
357: (12)
358: (8)
359: (12)
360: (20)
361: (16)
362: (8)
363: (12)
364: (4)
365: (8)
366: (8)
367: (8)
368: (8)
369: (8)
370: (8)
371: (8)
372: (12)
373: (8)
374: (12)
375: (20)
376: (16)
377: (8)
378: (12)
379: (16)
380: (4)
381: (8)
382: (8)
383: (8)
384: (8)
385: (4)
386: (8)
387: (12)
388: (12)
389: (12)
390: (4)
391: (8)
392: (12)
393: (12)
394: (8)
395: (4)
396: (8)
397: (12)
398: (12)
399: (16)
400: (12)
401: (4)
402: (8)
403: (12)
404: (12)
405: (16)
406: (12)
407: (16)
408: (12)
409: (16)
410: (4)
411: (8)
412: (12)
413: (12)
414: (16)
obj)
415: (12)
416: (16)
417: (12)
418: (16)
419: (4)

def test_inout_2seq(self):
    obj = np.array(self.num2seq, dtype=self.type.dtype)
    a = self.array([len(self.num2seq)], intent.inout, obj)
    assert a.has_shared_memory()
    try:
        a = self.array([2], intent.in_.inout, self.num2seq)
    except TypeError as msg:
        if not str(msg).startswith(
            "failed to initialize intent(inout|inplace|cache) array"):
            raise
    else:
        raise SystemError("intent(inout) should have failed on sequence")
def test_f_inout_23seq(self):
    obj = np.array(self.num23seq, dtype=self.type.dtype, order="F")
    shape = (len(self.num23seq), len(self.num23seq[0]))
    a = self.array(shape, intent.in_.inout, obj)
    assert a.has_shared_memory()
    obj = np.array(self.num23seq, dtype=self.type.dtype, order="C")
    shape = (len(self.num23seq), len(self.num23seq[0]))
    try:
        a = self.array(shape, intent.in_.inout, obj)
    except ValueError as msg:
        if not str(msg).startswith(
            "failed to initialize intent(inout) array"):
            raise
    else:
        raise SystemError(
            "intent(inout) should have failed on improper array")
def test_c_inout_23seq(self):
    obj = np.array(self.num23seq, dtype=self.type.dtype)
    shape = (len(self.num23seq), len(self.num23seq[0]))
    a = self.array(shape, intent.in_.c.inout, obj)
    assert a.has_shared_memory()
def test_in_copy_from_2casttype(self):
    for t in self.type.cast_types():
        obj = np.array(self.num2seq, dtype=t.dtype)
        a = self.array([len(self.num2seq)], intent.in_.copy, obj)
        assert not a.has_shared_memory()
def test_c_in_from_23seq(self):
    a = self.array(
        [len(self.num23seq), len(self.num23seq[0])], intent.in_,
        self.num23seq)
    assert not a.has_shared_memory()
def test_in_from_23casttype(self):
    for t in self.type.cast_types():
        obj = np.array(self.num23seq, dtype=t.dtype)
        a = self.array(
            [len(self.num23seq), len(self.num23seq[0])], intent.in_, obj)
        assert not a.has_shared_memory()
def test_f_in_from_23casttype(self):
    for t in self.type.cast_types():
        obj = np.array(self.num23seq, dtype=t.dtype, order="F")
        a = self.array(
            [len(self.num23seq), len(self.num23seq[0])], intent.in_, obj)
        if t.elsize == self.type.elsize:
            assert a.has_shared_memory()
        else:
            assert not a.has_shared_memory()
def test_c_in_from_23casttype(self):
    for t in self.type.cast_types():
        obj = np.array(self.num23seq, dtype=t.dtype)
        a = self.array(
            [len(self.num23seq), len(self.num23seq[0])], intent.in_.c,
            obj)
        if t.elsize == self.type.elsize:
            assert a.has_shared_memory()
        else:
            assert not a.has_shared_memory()
def test_f_copy_in_from_23casttype(self):

```

```

420: (8)
421: (12)
422: (12)
423: (16)
424: (16)
425: (12)
426: (4)
427: (8)
428: (12)
429: (12)
430: (16)
intent.in_.c.copy,
431: (16)
432: (12)
433: (4)
434: (8)
435: (12)
436: (16)
437: (12)
438: (12)
439: (12)
440: (12)
441: (12)
442: (12)
443: (12)
444: (12)
445: (12)
446: (12)
447: (12)
448: (12)
449: (16)
450: (12)
451: (16)
452: (24)
453: (20)
454: (12)
455: (16)
456: (20)
array")
457: (4)
458: (8)
459: (12)
460: (16)
461: (12)
462: (16)
463: (12)
464: (12)
465: (12)
try:
    a = self.array(shape, intent.in_.cache, obj[:::-1])
except ValueError as msg:
    if not str(msg).startswith(
        "failed to initialize intent(cache) array"):
        raise
    else:
        raise SystemError(
            "intent(cache) should have failed on multisegmented
array")
466: (16)
467: (12)
468: (16)
469: (24)
470: (20)
471: (12)
472: (16)
473: (20)
474: (4)
475: (8)
476: (8)
477: (8)
478: (8)
479: (8)
480: (8)
481: (8)
482: (8)
483: (12)
484: (8)
485: (12)
486: (20)
for t in self.type.cast_types():
    obj = np.array(self.num2seq, dtype=t.dtype, order="F")
    a = self.array(
        [len(self.num2seq), len(self.num2seq[0])], intent.in_.copy,
        obj)
    assert not a.has_shared_memory()
def test_c_copy_in_from_2casttype(self):
    for t in self.type.cast_types():
        obj = np.array(self.num2seq, dtype=t.dtype)
        a = self.array(
            [len(self.num2seq), len(self.num2seq[0])],
            obj)
        assert not a.has_shared_memory()
def test_in_cache_from_2casttype(self):
    for t in self.type.all_types():
        if t.elsize != self.type.elsize:
            continue
        obj = np.array(self.num2seq, dtype=t.dtype)
        shape = (len(self.num2seq), )
        a = self.array(shape, intent.in_.cache, obj)
        assert a.has_shared_memory()
        a = self.array(shape, intent.in_.cache, obj)
        assert a.has_shared_memory()
        obj = np.array(self.num2seq, dtype=t.dtype, order="F")
        a = self.array(shape, intent.in_.cache, obj)
        assert a.has_shared_memory()
        a = self.array(shape, intent.in_.cache, obj)
        assert a.has_shared_memory()
        try:
            a = self.array(shape, intent.in_.cache, obj[:::-1])
        except ValueError as msg:
            if not str(msg).startswith(
                "failed to initialize intent(cache) array"):
                raise
            else:
                raise SystemError(
                    "intent(cache) should have failed on multisegmented
array")
def test_in_cache_from_2casttype_failure(self):
    for t in self.type.all_types():
        if t.NAME == 'STRING':
            continue
        if t.elsize >= self.type.elsize:
            continue
        obj = np.array(self.num2seq, dtype=t.dtype)
        shape = (len(self.num2seq), )
        try:
            self.array(shape, intent.in_.cache, obj) # Should succeed
        except ValueError as msg:
            if not str(msg).startswith(
                "failed to initialize intent(cache) array"):
                raise
            else:
                raise SystemError(
                    "intent(cache) should have failed on smaller array")
def test_cache_hidden(self):
    shape = (2, )
    a = self.array(shape, intent.cache.hide, None)
    assert a.arr.shape == shape
    shape = (2, 3)
    a = self.array(shape, intent.cache.hide, None)
    assert a.arr.shape == shape
    shape = (-1, 3)
    try:
        a = self.array(shape, intent.cache.hide, None)
    except ValueError as msg:
        if not str(msg).startswith(
            "failed to create intent(cache|hide)|optional array"):

```

```

487: (16)           raise
488: (8)            else:
489: (12)           raise SystemError(
490: (16)             "intent(cache) should have failed on undefined dimensions")
491: (4)            def test_hidden(self):
492: (8)              shape = (2, )
493: (8)              a = self.array(shape, intent.hide, None)
494: (8)              assert a.arr.shape == shape
495: (8)              assert a.arr_equal(a.arr, np.zeros(shape, dtype=self.type.dtype))
496: (8)              shape = (2, 3)
497: (8)              a = self.array(shape, intent.hide, None)
498: (8)              assert a.arr.shape == shape
499: (8)              assert a.arr_equal(a.arr, np.zeros(shape, dtype=self.type.dtype))
500: (8)              assert a.arr.flags["FORTRAN"] and not a.arr.flags["CONTIGUOUS"]
501: (8)              shape = (2, 3)
502: (8)              a = self.array(shape, intent.c.hide, None)
503: (8)              assert a.arr.shape == shape
504: (8)              assert a.arr_equal(a.arr, np.zeros(shape, dtype=self.type.dtype))
505: (8)              assert not a.arr.flags["FORTRAN"] and a.arr.flags["CONTIGUOUS"]
506: (8)              shape = (-1, 3)
507: (8)            try:
508: (12)              a = self.array(shape, intent.hide, None)
509: (8)            except ValueError as msg:
510: (12)              if not str(msg).startswith(
511: (20)                "failed to create intent(cache|hide)|optional array"):
512: (16)                raise
513: (8)            else:
514: (12)              raise SystemError(
515: (16)                "intent(hide) should have failed on undefined dimensions")
516: (4)            def test_optional_none(self):
517: (8)              shape = (2, )
518: (8)              a = self.array(shape, intent.optional, None)
519: (8)              assert a.arr.shape == shape
520: (8)              assert a.arr_equal(a.arr, np.zeros(shape, dtype=self.type.dtype))
521: (8)              shape = (2, 3)
522: (8)              a = self.array(shape, intent.optional, None)
523: (8)              assert a.arr.shape == shape
524: (8)              assert a.arr_equal(a.arr, np.zeros(shape, dtype=self.type.dtype))
525: (8)              assert a.arr.flags["FORTRAN"] and not a.arr.flags["CONTIGUOUS"]
526: (8)              shape = (2, 3)
527: (8)              a = self.array(shape, intent.c.optional, None)
528: (8)              assert a.arr.shape == shape
529: (8)              assert a.arr_equal(a.arr, np.zeros(shape, dtype=self.type.dtype))
530: (8)              assert not a.arr.flags["FORTRAN"] and a.arr.flags["CONTIGUOUS"]
531: (4)            def test_optional_from_2seq(self):
532: (8)              obj = self.num2seq
533: (8)              shape = (len(obj), )
534: (8)              a = self.array(shape, intent.optional, obj)
535: (8)              assert a.arr.shape == shape
536: (8)              assert not a.has_shared_memory()
537: (4)            def test_optional_from_23seq(self):
538: (8)              obj = self.num23seq
539: (8)              shape = (len(obj), len(obj[0]))
540: (8)              a = self.array(shape, intent.optional, obj)
541: (8)              assert a.arr.shape == shape
542: (8)              assert not a.has_shared_memory()
543: (8)              a = self.array(shape, intent.optional.c, obj)
544: (8)              assert a.arr.shape == shape
545: (8)              assert not a.has_shared_memory()
546: (4)            def test_inplace(self):
547: (8)              obj = np.array(self.num23seq, dtype=self.type.dtype)
548: (8)              assert not obj.flags["FORTRAN"] and obj.flags["CONTIGUOUS"]
549: (8)              shape = obj.shape
550: (8)              a = self.array(shape, intent.inplace, obj)
551: (8)              assert obj[1][2] == a.arr[1][2], repr((obj, a.arr))
552: (8)              a.arr[1][2] = 54
553: (8)              assert obj[1][2] == a.arr[1][2] == np.array(54, dtype=self.type.dtype)
554: (8)              assert a.arr is obj
555: (8)              assert obj.flags["FORTRAN"] # obj attributes are changed inplace!

```

```

556: (8)             assert not obj.flags["CONTIGUOUS"]
557: (4)             def test_inplace_from_casttype(self):
558: (8)                 for t in self.type.cast_types():
559: (12)                     if t is self.type:
560: (16)                         continue
561: (12)                     obj = np.array(self.num23seq, dtype=t.dtype)
562: (12)                     assert obj.dtype.type == t.type
563: (12)                     assert obj.dtype.type is not self.type.type
564: (12)                     assert not obj.flags["FORTRAN"] and obj.flags["CONTIGUOUS"]
565: (12)                     shape = obj.shape
566: (12)                     a = self.array(shape, intent.inplace, obj)
567: (12)                     assert obj[1][2] == a.arr[1][2], repr((obj, a.arr))
568: (12)                     a.arr[1][2] = 54
569: (12)                     assert obj[1][2] == a.arr[1][2] == np.array(54,
570: (56)                                     dtype=self.type.dtype)
571: (12)                     assert a.arr is obj
572: (12)                     assert obj.flags["FORTRAN"] # obj attributes changed inplace!
573: (12)                     assert not obj.flags["CONTIGUOUS"]
574: (12)                     assert obj.dtype.type is self.type.type # obj changed inplace!

```

---

File 163 - test\_assumed\_shape.py:

```

1: (0)             import os
2: (0)             import pytest
3: (0)             import tempfile
4: (0)             from . import util
5: (0)             class TestAssumedShapeSumExample(util.F2PyTest):
6: (4)                 sources = [
7: (8)                     util.getpath("tests", "src", "assumed_shape", "foo_free.f90"),
8: (8)                     util.getpath("tests", "src", "assumed_shape", "foo_use.f90"),
9: (8)                     util.getpath("tests", "src", "assumed_shape", "precision.f90"),
10: (8)                    util.getpath("tests", "src", "assumed_shape", "foo_mod.f90"),
11: (8)                    util.getpath("tests", "src", "assumed_shape", ".f2py_f2cmap"),
12: (4)                 ]
13: (4)             @pytest.mark.slow
14: (4)             def test_all(self):
15: (8)                 r = self.module.fsum([1, 2])
16: (8)                 assert r == 3
17: (8)                 r = self.module.sum([1, 2])
18: (8)                 assert r == 3
19: (8)                 r = self.module.sum_with_use([1, 2])
20: (8)                 assert r == 3
21: (8)                 r = self.module.mod.sum([1, 2])
22: (8)                 assert r == 3
23: (8)                 r = self.module.mod.fsum([1, 2])
24: (8)                 assert r == 3
25: (0)             class TestF2cmapOption(TestAssumedShapeSumExample):
26: (4)                 def setup_method(self):
27: (8)                     self.sources = list(self.sources)
28: (8)                     f2cmap_src = self.sources.pop(-1)
29: (8)                     self.f2cmap_file = tempfile.NamedTemporaryFile(delete=False)
30: (8)                     with open(f2cmap_src, "rb") as f:
31: (12)                         self.f2cmap_file.write(f.read())
32: (8)                     self.f2cmap_file.close()
33: (8)                     self.sources.append(self.f2cmap_file.name)
34: (8)                     self.options = ["--f2cmap", self.f2cmap_file.name]
35: (8)                     super().setup_method()
36: (4)                 def teardown_method(self):
37: (8)                     os.unlink(self.f2cmap_file.name)

```

---

File 164 - test\_callback.py:

```

1: (0)             import math
2: (0)             import textwrap
3: (0)             import sys

```

```

4: (0) import pytest
5: (0) import threading
6: (0) import traceback
7: (0) import time
8: (0) import numpy as np
9: (0) from numpy.testing import IS_PYPY
10: (0) from . import util
11: (0) class TestF77Callback(util.F2PyTest):
12: (4)     sources = [util.getpath("tests", "src", "callback", "foo.f")]
13: (4)     @pytest.mark.parametrize("name", "t,t2".split(","))
14: (4)     def test_all(self, name):
15: (8)         self.check_function(name)
16: (4)         @pytest.mark.xfail(IS_PYPY,
17: (23)             reason="PyPy cannot modify tp_doc after PyType_Ready")
18: (4)     def test_docstring(self):
19: (8)         expected = textwrap.dedent("""\
20: (8)             a = t(fun,[fun_extra_args])
21: (8)             Wrapper for ``t``.
22: (8)             Parameters
23: (8)             -----
24: (8)             fun : call-back function
25: (8)             Other Parameters
26: (8)             -----
27: (8)             fun_extra_args : input tuple, optional
28: (12)                 Default: ()
29: (8)             Returns
30: (8)             -----
31: (8)             a : int
32: (8)             Notes
33: (8)             -----
34: (8)             Call-back functions::
35: (12)                 def fun(): return a
36: (12)                 Return objects:
37: (16)                     a : int
38: (8)                     """
39: (8)             assert self.module.t.__doc__ == expected
40: (4)     def check_function(self, name):
41: (8)         t = getattr(self.module, name)
42: (8)         r = t(lambda: 4)
43: (8)         assert r == 4
44: (8)         r = t(lambda a: 5, fun_extra_args=(6, ))
45: (8)         assert r == 5
46: (8)         r = t(lambda a: a, fun_extra_args=(6, ))
47: (8)         assert r == 6
48: (8)         r = t(lambda a: 5 + a, fun_extra_args=(7, ))
49: (8)         assert r == 12
50: (8)         r = t(lambda a: math.degrees(a), fun_extra_args=(math.pi, ))
51: (8)         assert r == 180
52: (8)         r = t(math.degrees, fun_extra_args=(math.pi, ))
53: (8)         assert r == 180
54: (8)         r = t(self.module.func, fun_extra_args=(6, ))
55: (8)         assert r == 17
56: (8)         r = t(self.module.func0)
57: (8)         assert r == 11
58: (8)         r = t(self.module.func0._cpointer)
59: (8)         assert r == 11
60: (8)         class A:
61: (12)             def __call__(self):
62: (16)                 return 7
63: (12)             def mth(self):
64: (16)                 return 9
65: (8)                 a = A()
66: (8)                 r = t(a)
67: (8)                 assert r == 7
68: (8)                 r = t(a.mth)
69: (8)                 assert r == 9
70: (4)                 @pytest.mark.skipif(sys.platform == 'win32',
71: (24)                     reason='Fails with MinGW64 Gfortran (Issue #9673)')
72: (4)                 def test_string_callback(self):

```

```

73: (8)           def callback(code):
74: (12)          if code == "r":
75: (16)          return 0
76: (12)          else:
77: (16)          return 1
78: (8)          f = getattr(self.module, "string_callback")
79: (8)          r = f(callback)
80: (8)          assert r == 0
81: (4) @pytest.mark.skipif(sys.platform == 'win32',
82: (24)             reason='Fails with MinGW64 Gfortran (Issue #9673)')
83: (4) def test_string_callback_array(self):
84: (8)     cu1 = np.zeros((1, ), "S8")
85: (8)     cu2 = np.zeros((1, 8), "c")
86: (8)     cu3 = np.array([""], "S8")
87: (8)     def callback(cu, lencu):
88: (12)         if cu.shape != (lencu,):
89: (16)             return 1
90: (12)         if cu.dtype != "S8":
91: (16)             return 2
92: (12)         if not np.all(cu == b ""):
93: (16)             return 3
94: (12)         return 0
95: (8)     f = getattr(self.module, "string_callback_array")
96: (8)     for cu in [cu1, cu2, cu3]:
97: (12)         res = f(callback, cu, cu.size)
98: (12)         assert res == 0
99: (4) def test_threadsafety(self):
100: (8)    errors = []
101: (8)    def cb():
102: (12)        time.sleep(1e-3)
103: (12)        r = self.module.t(lambda: 123)
104: (12)        assert r == 123
105: (12)        return 42
106: (8)    def runner(name):
107: (12)        try:
108: (16)            for j in range(50):
109: (20)                r = self.module.t(cb)
110: (20)                assert r == 42
111: (20)                self.check_function(name)
112: (12)        except Exception:
113: (16)            errors.append(traceback.format_exc())
114: (8)    threads = [
115: (12)        threading.Thread(target=runner, args=(arg, ))
116: (12)        for arg in ("t", "t2") for n in range(20)
117: (8)    ]
118: (8)    for t in threads:
119: (12)        t.start()
120: (8)    for t in threads:
121: (12)        t.join()
122: (8)    errors = "\n\n".join(errors)
123: (8)    if errors:
124: (12)        raise AssertionError(errors)
125: (4) def test_hidden_callback(self):
126: (8)    try:
127: (12)        self.module.hidden_callback(2)
128: (8)    except Exception as msg:
129: (12)        assert str(msg).startswith("Callback global_f not defined")
130: (8)    try:
131: (12)        self.module.hidden_callback2(2)
132: (8)    except Exception as msg:
133: (12)        assert str(msg).startswith("cb: Callback global_f not defined")
134: (8)    self.module.global_f = lambda x: x + 1
135: (8)    r = self.module.hidden_callback(2)
136: (8)    assert r == 3
137: (8)    self.module.global_f = lambda x: x + 2
138: (8)    r = self.module.hidden_callback(2)
139: (8)    assert r == 4
140: (8)    del self.module.global_f
141: (8)    try:

```

```

142: (12)             self.module.hidden_callback(2)
143: (8)          except Exception as msg:
144: (12)              assert str(msg).startswith("Callback global_f not defined")
145: (8)              self.module.global_f = lambda x=0: x + 3
146: (8)              r = self.module.hidden_callback(2)
147: (8)              assert r == 5
148: (8)              r = self.module.hidden_callback2(2)
149: (8)              assert r == 3
150: (0)      class TestF77CallbackPythonTLS(TestF77Callback):
151: (4)          """
152: (4)              Callback tests using Python thread-local storage instead of
153: (4)              compiler-provided
154: (4)          """
155: (4)          options = ["-DF2PY_USE_PYTHON_TLS"]
156: (0)      class TestF90Callback(util.F2PyTest):
157: (4)          sources = [util.getpath("tests", "src", "callback", "gh17797.f90")]
158: (4)          def test_gh17797(self):
159: (8)              def incr(x):
160: (12)                  return x + 123
161: (8)              y = np.array([1, 2, 3], dtype=np.int64)
162: (8)              r = self.module.gh17797(incr, y)
163: (8)              assert r == 123 + 1 + 2 + 3
164: (0)      class TestGH18335(util.F2PyTest):
165: (4)          """
166: (4)              The reproduction of the reported issue requires specific input that
167: (4)              extensions may break the issue conditions, so the reproducer is
168: (4)              implemented as a separate test class. Do not extend this test with
169: (4)              other tests!
169: (4)          """
170: (4)          sources = [util.getpath("tests", "src", "callback", "gh18335.f90")]
171: (4)          def test_gh18335(self):
172: (8)              def foo(x):
173: (12)                  x[0] += 1
174: (8)                  r = self.module.gh18335(foo)
175: (8)                  assert r == 123 + 1
176: (0)      class TestGH25211(util.F2PyTest):
177: (4)          sources = [util.getpath("tests", "src", "callback", "gh25211.f"),
178: (15)              util.getpath("tests", "src", "callback", "gh25211.pyf")]
179: (4)          module_name = "callback2"
180: (4)          def test_gh18335(self):
181: (8)              def bar(x):
182: (12)                  return x*x
183: (8)                  res = self.module.foo(bar)
184: (8)                  assert res == 110

```

-----

## File 165 - test\_block\_docstring.py:

```

1: (0)          import sys
2: (0)          import pytest
3: (0)          from . import util
4: (0)          from numpy.testing import IS_PYPY
5: (0)          class TestBlockDocString(util.F2PyTest):
6: (4)              sources = [util.getpath("tests", "src", "block_docstring", "foo.f")]
7: (4)              @pytest.mark.skipif(sys.platform == "win32",
8: (24)                  reason="Fails with MinGW64 Gfortran (Issue #9673)")
9: (4)              @pytest.mark.xfail(IS_PYPY,
10: (23)                  reason="PyPy cannot modify tp_doc after PyType_Ready")
11: (4)              def test_block_docstring(self):
12: (8)                  expected = "bar : 'i'-array(2,3)\n"
13: (8)                  assert self.module.block.__doc__ == expected

```

-----

## File 166 - test\_common.py:

```

1: (0)          import os
2: (0)          import sys
3: (0)          import pytest

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4: (0) import numpy as np
5: (0) from . import util
6: (0) class TestCommonBlock(util.F2PyTest):
7: (4)     sources = [util.getpath("tests", "src", "common", "block.f")]
8: (4)     @pytest.mark.skipif(sys.platform == "win32",
9: (24)                     reason="Fails with MinGW64 Gfortran (Issue #9673)")
10: (4)    def test_common_block(self):
11: (8)        self.module.initcb()
12: (8)        assert self.module.block.long_bn == np.array(1.0, dtype=np.float64)
13: (8)        assert self.module.block.string_bn == np.array("2", dtype="|S1")
14: (8)        assert self.module.block.ok == np.array(3, dtype=np.int32)
15: (0) class TestCommonWithUse(util.F2PyTest):
16: (4)     sources = [util.getpath("tests", "src", "common", "gh19161.f90")]
17: (4)     @pytest.mark.skipif(sys.platform == "win32",
18: (24)                     reason="Fails with MinGW64 Gfortran (Issue #9673)")
19: (4)    def test_common_gh19161(self):
20: (8)        assert self.module.data.x == 0

```

---

## File 167 - test\_character.py:

```

1: (0) import pytest
2: (0) import textwrap
3: (0) from numpy.testing import assert_array_equal, assert_equal, assert_raises
4: (0) import numpy as np
5: (0) from numpy.f2py.tests import util
6: (0) class TestCharacterString(util.F2PyTest):
7: (4)     suffix = '.f90'
8: (4)     fprefix = 'test_character_string'
9: (4)     length_list = ['1', '3', 'star']
10: (4)    code = ''
11: (4)    for length in length_list:
12: (8)        fsuffix = length
13: (8)        clength = dict(star='(*)').get(length, length)
14: (8)        code += textwrap.dedent(f"""
15: (8)            subroutine {fprefix}_input_{fsuffix}(c, o, n)
16: (10)                character*{clength}, intent(in) :: c
17: (10)                integer n
18: (10)                !f2py integer, depend(c), intent(hide) :: n = slen(c)
19: (10)                integer*1, dimension(n) :: o
20: (10)                !f2py intent(out) o
21: (10)                o = transfer(c, o)
22: (8)            end subroutine {fprefix}_input_{fsuffix}
23: (8)            subroutine {fprefix}_output_{fsuffix}(c, o, n)
24: (10)                character*{clength}, intent(out) :: c
25: (10)                integer n
26: (10)                integer*1, dimension(n), intent(in) :: o
27: (10)                !f2py integer, depend(o), intent(hide) :: n = len(o)
28: (10)                c = transfer(o, c)
29: (8)            end subroutine {fprefix}_output_{fsuffix}
30: (8)            subroutine {fprefix}_array_input_{fsuffix}(c, o, m, n)
31: (10)                integer m, i, n
32: (10)                character*{clength}, intent(in), dimension(m) :: c
33: (10)                !f2py integer, depend(c), intent(hide) :: m = len(c)
34: (10)                !f2py integer, depend(c), intent(hide) :: n = f2py_itemsize(c)
35: (10)                integer*1, dimension(m, n), intent(out) :: o
36: (10)                do i=1,m
37: (12)                    o(i, :) = transfer(c(i), o(i, :))
38: (10)                end do
39: (8)            end subroutine {fprefix}_array_input_{fsuffix}
40: (8)            subroutine {fprefix}_array_output_{fsuffix}(c, o, m, n)
41: (10)                character*{clength}, intent(out), dimension(m) :: c
42: (10)                integer n
43: (10)                integer*1, dimension(m, n), intent(in) :: o
44: (10)                !f2py character(f2py_len=n) :: c
45: (10)                !f2py integer, depend(o), intent(hide) :: m = len(o)
46: (10)                !f2py integer, depend(o), intent(hide) :: n = shape(o, 1)
47: (10)                do i=1,m

```

```

48: (12)           c(i) = transfer(o(i, :), c(i))
49: (10)         end do
50: (8)       end subroutine {fprefix}_array_output_{fsuffix}
51: (8)       subroutine {fprefix}_2d_array_input_{fsuffix}(c, o, m1, m2, n)
52: (10)         integer m1, m2, i, j, n
53: (10)         character*{clength}, intent(in), dimension(m1, m2) :: c
54: (10)         !f2py integer, depend(c), intent(hide) :: m1 = len(c)
55: (10)         !f2py integer, depend(c), intent(hide) :: m2 = shape(c, 1)
56: (10)         !f2py integer, depend(c), intent(hide) :: n = f2py_itemsize(c)
57: (10)         integer*1, dimension(m1, m2, n), intent(out) :: o
58: (10)         do i=1,m1
59: (12)           do j=1,m2
60: (14)             o(i, j, :) = transfer(c(i, j), o(i, j, :))
61: (12)           end do
62: (10)         end do
63: (8)       end subroutine {fprefix}_2d_array_input_{fsuffix}
64: (8)     """
65: (4)   @pytest.mark.parametrize("length", length_list)
66: (4)   def test_input(self, length):
67: (8)     fsuffix = {'(*)': 'star'}.get(length, length)
68: (8)     f = getattr(self.module, self.fprefix + '_input_' + fsuffix)
69: (8)     a = {'1': 'a', '3': 'abc', 'star': 'abcde' * 3}[length]
70: (8)     assert_array_equal(f(a), np.array(list(map(ord, a))), dtype='u1'))
71: (4)   @pytest.mark.parametrize("length", length_list[:-1])
72: (4)   def test_output(self, length):
73: (8)     fsuffix = length
74: (8)     f = getattr(self.module, self.fprefix + '_output_' + fsuffix)
75: (8)     a = {'1': 'a', '3': 'abc'}[length]
76: (8)     assert_array_equal(f(np.array(list(map(ord, a))), dtype='u1')),
77: (27)           a.encode())
78: (4)   @pytest.mark.parametrize("length", length_list)
79: (4)   def test_array_input(self, length):
80: (8)     fsuffix = length
81: (8)     f = getattr(self.module, self.fprefix + '_array_input_' + fsuffix)
82: (8)     a = np.array([{'1': 'a', '3': 'abc', 'star': 'abcde' * 3}[length],
83: (22)           {'1': 'A', '3': 'ABC', 'star': 'ABCDE' * 3}[length],
84: (22)           ], dtype='S')
85: (8)     expected = np.array([[c for c in s] for s in a], dtype='u1')
86: (8)     assert_array_equal(f(a), expected)
87: (4)   @pytest.mark.parametrize("length", length_list)
88: (4)   def test_array_output(self, length):
89: (8)     fsuffix = length
90: (8)     f = getattr(self.module, self.fprefix + '_array_output_' + fsuffix)
91: (8)     expected = np.array(
92: (12)       [{'1': 'a', '3': 'abc', 'star': 'abcde' * 3}[length],
93: (13)         {'1': 'A', '3': 'ABC', 'star': 'ABCDE' * 3}[length]], dtype='S')
94: (8)     a = np.array([[c for c in s] for s in expected], dtype='u1')
95: (8)     assert_array_equal(f(a), expected)
96: (4)   @pytest.mark.parametrize("length", length_list)
97: (4)   def test_2d_array_input(self, length):
98: (8)     fsuffix = length
99: (8)     f = getattr(self.module, self.fprefix + '_2d_array_input_' + fsuffix)
100: (8)    a = np.array([[[{'1': 'a', '3': 'abc', 'star': 'abcde' * 3}[length],
101: (23)          {'1': 'A', '3': 'ABC', 'star': 'ABCDE' * 3}[length]],
102: (22)            [{}'1': 'f', '3': 'fg', 'star': 'fghij' * 3}[length],
103: (23)              {'1': 'F', '3': 'FGH', 'star': 'FGHIJ' * 3}[length]]],
104: (21)            dtype='S')
105: (8)    expected = np.array([[[[c for c in item] for item in row] for row in
106: (28)                  a],
107: (8)                      dtype='u1', order='F')
108: (0)        assert_array_equal(f(a), expected)
109: (4)      class TestCharacter(util.F2PyTest):
110: (4)        suffix = '.f90'
111: (4)        fprefix = 'test_character'
112: (4)        code = textwrap.dedent("""
113: (7)          subroutine {fprefix}_input(c, o)
114: (10)            character, intent(in) :: c
115: (10)            integer*1 o
116: (10)            !f2py intent(out) o

```

```

116: (10)          o = transfer(c, o)
117: (7)           end subroutine {fprefix}_input
118: (7)           subroutine {fprefix}_output(c, o)
119: (10)             character :: c
120: (10)             integer*1, intent(in) :: o
121: (10)             !f2py intent(out) c
122: (10)             c = transfer(o, c)
123: (7)           end subroutine {fprefix}_output
124: (7)           subroutine {fprefix}_input_output(c, o)
125: (10)             character, intent(in) :: c
126: (10)             character o
127: (10)             !f2py intent(out) o
128: (10)             o = c
129: (7)           end subroutine {fprefix}_input_output
130: (7)           subroutine {fprefix}_inout(c, n)
131: (10)             character :: c, n
132: (10)             !f2py intent(in) n
133: (10)             !f2py intent(inout) c
134: (10)             c = n
135: (7)           end subroutine {fprefix}_inout
136: (7)           function {fprefix}_return(o) result (c)
137: (10)             character :: c
138: (10)             character, intent(in) :: o
139: (10)             c = transfer(o, c)
140: (7)           end function {fprefix}_return
141: (7)           subroutine {fprefix}_array_input(c, o)
142: (10)             character, intent(in) :: c(3)
143: (10)             integer*1 o(3)
144: (10)             !f2py intent(out) o
145: (10)             integer i
146: (10)             do i=1,3
147: (12)               o(i) = transfer(c(i), o(i))
148: (10)             end do
149: (7)           end subroutine {fprefix}_array_input
150: (7)           subroutine {fprefix}_2d_array_input(c, o)
151: (10)             character, intent(in) :: c(2, 3)
152: (10)             integer*1 o(2, 3)
153: (10)             !f2py intent(out) o
154: (10)             integer i, j
155: (10)             do i=1,2
156: (12)               do j=1,3
157: (14)                 o(i, j) = transfer(c(i, j), o(i, j))
158: (12)               end do
159: (10)             end do
160: (7)           end subroutine {fprefix}_2d_array_input
161: (7)           subroutine {fprefix}_array_output(c, o)
162: (10)             character :: c(3)
163: (10)             integer*1, intent(in) :: o(3)
164: (10)             !f2py intent(out) c
165: (10)             do i=1,3
166: (12)               c(i) = transfer(o(i), c(i))
167: (10)             end do
168: (7)           end subroutine {fprefix}_array_output
169: (7)           subroutine {fprefix}_array_inout(c, n)
170: (10)             character :: c(3), n(3)
171: (10)             !f2py intent(in) n(3)
172: (10)             !f2py intent(inout) c(3)
173: (10)             do i=1,3
174: (12)               c(i) = n(i)
175: (10)             end do
176: (7)           end subroutine {fprefix}_array_inout
177: (7)           subroutine {fprefix}_2d_array_inout(c, n)
178: (10)             character :: c(2, 3), n(2, 3)
179: (10)             !f2py intent(in) n(2, 3)
180: (10)             !f2py intent(inout) c(2, 3)
181: (10)             integer i, j
182: (10)             do i=1,2
183: (12)               do j=1,3
184: (14)                 c(i, j) = n(i, j)

```

```

185: (12)           end do
186: (10)          end do
187: (7)           end subroutine {fprefix}_2d_array_inout
188: (7)           function {fprefix}_array_return(o) result (c)
189: (10)          character, dimension(3) :: c
190: (10)          character, intent(in) :: o(3)
191: (10)          do i=1,3
192: (12)            c(i) = o(i)
193: (10)          end do
194: (7)           end function {fprefix}_array_return
195: (7)           function {fprefix}_optional(o) result (c)
196: (10)          character, intent(in) :: o
197: (10)          !f2py character o = "a"
198: (10)          character :: c
199: (10)          c = o
200: (7)           end function {fprefix}_optional
201: (4)           """
202: (4)           @pytest.mark.parametrize("dtype", ['c', 'S1'])
203: (4)           def test_input(self, dtype):
204: (8)             f = getattr(self.module, self.fprefix + '_input')
205: (8)             assert_equal(f(np.array('a', dtype=dtype)), ord('a'))
206: (8)             assert_equal(f(np.array(b'a', dtype=dtype)), ord('a'))
207: (8)             assert_equal(f(np.array(['a'], dtype=dtype)), ord('a'))
208: (8)             assert_equal(f(np.array('abc', dtype=dtype)), ord('a'))
209: (8)             assert_equal(f(np.array([[a]]), dtype=dtype)), ord('a'))
210: (4)           def test_input_varia(self):
211: (8)             f = getattr(self.module, self.fprefix + '_input')
212: (8)             assert_equal(f('a'), ord('a'))
213: (8)             assert_equal(f(b'a'), ord(b'a'))
214: (8)             assert_equal(f(''), 0)
215: (8)             assert_equal(f(b''), 0)
216: (8)             assert_equal(f(b'\0'), 0)
217: (8)             assert_equal(f('ab'), ord('a'))
218: (8)             assert_equal(f(b'ab'), ord('a'))
219: (8)             assert_equal(f(['a']), ord('a'))
220: (8)             assert_equal(f(np.array(b'a')), ord('a'))
221: (8)             assert_equal(f(np.array([b'a])), ord('a'))
222: (8)             a = np.array('a')
223: (8)             assert_equal(f(a), ord('a'))
224: (8)             a = np.array(['a'])
225: (8)             assert_equal(f(a), ord('a'))
226: (8)             try:
227: (12)               f([])
228: (8)             except IndexError as msg:
229: (12)               if not str(msg).endswith(' got 0-list'):
230: (16)                 raise
231: (8)             else:
232: (12)               raise SystemError(f'{f.__name__} should have failed on empty
list')
233: (8)
234: (12)
235: (8)
236: (12)
237: (16)
238: (8)
239: (12)             raise SystemError(f'{f.__name__} should have failed on int value')
240: (4)           @pytest.mark.parametrize("dtype", ['c', 'S1', 'U1'])
241: (4)           def test_array_input(self, dtype):
242: (8)             f = getattr(self.module, self.fprefix + '_array_input')
243: (8)             assert_array_equal(f(np.array(['a', 'b', 'c'], dtype=dtype)),
244: (27)                           np.array(list(map(ord, 'abc'))), dtype='i1')
245: (8)             assert_array_equal(f(np.array([b'a', b'b', b'c'], dtype=dtype)),
246: (27)                           np.array(list(map(ord, 'abc'))), dtype='i1')
247: (4)           def test_array_input_varia(self):
248: (8)             f = getattr(self.module, self.fprefix + '_array_input')
249: (8)             assert_array_equal(f(['a', 'b', 'c']),
250: (27)                           np.array(list(map(ord, 'abc'))), dtype='i1')
251: (8)             assert_array_equal(f([b'a', b'b', b'c']),
252: (27)                           np.array(list(map(ord, 'abc'))), dtype='i1'))

```

```

253: (8)
254: (12)
255: (8)
256: (12)
257: (20)
258: (16)
259: (8)
260: (12)
261: (16)
262: (4)
263: (4)
264: (8)
265: (8)
266: (22)
267: (8)
268: (8)
269: (4)
270: (8)
271: (8)
272: (8)
273: (4)
274: (8)
275: (8)
276: (27)
277: (4)
278: (8)
279: (8)
280: (8)
281: (8)
282: (4)
283: (4)
284: (8)
285: (8)
286: (8)
287: (8)
288: (8)
289: (8)
290: (8)
291: (8)
292: (8)
293: (4)
294: (8)
295: (8)
296: (8)
297: (8)
298: (8)
299: (8)
300: (8)
301: (8)
302: (12)
303: (8)
304: (12)
305: (16)
306: (8)
307: (12)
308: (4)
309: (4)
310: (8)
311: (8)
312: (8)
313: (8)
314: (8)
315: (8)
316: (8)
317: (8)
318: (8)
319: (8)
320: (8)
321: (8)

        try:
            f(['a', 'b', 'c', 'd'])
        except ValueError as msg:
            if not str(msg).endswith(
                'th dimension must be fixed to 3 but got 4'):
                raise
            else:
                raise SystemError(
                    f'{f.__name__} should have failed on wrong input')
    @pytest.mark.parametrize("dtype", ['c', 'S1', 'U1'])
    def test_2d_array_input(self, dtype):
        f = getattr(self.module, self.fprefix + '_2d_array_input')
        a = np.array([[['a', 'b', 'c'],
                      ['d', 'e', 'f']]],
                     dtype=dtype, order='F')
        expected = a.view(np.uint32 if dtype == 'U1' else np.uint8)
        assert_array_equal(f(a), expected)
    def test_output(self):
        f = getattr(self.module, self.fprefix + '_output')
        assert_equal(f(ord(b'a')), b'a')
        assert_equal(f(0), b'\0')
    def test_array_output(self):
        f = getattr(self.module, self.fprefix + '_array_output')
        assert_array_equal(f(list(map(ord, 'abc'))),
                           np.array(list('abc'), dtype='S1'))
    def test_input_output(self):
        f = getattr(self.module, self.fprefix + '_input_output')
        assert_equal(f(b'a'), b'a')
        assert_equal(f('a'), b'a')
        assert_equal(f(''), b'\0')
    @pytest.mark.parametrize("dtype", ['c', 'S1'])
    def test_inout(self, dtype):
        f = getattr(self.module, self.fprefix + '_inout')
        a = np.array(list('abc'), dtype=dtype)
        f(a, 'A')
        assert_array_equal(a, np.array(list('Abc'), dtype=a.dtype))
        f(a[1:], 'B')
        assert_array_equal(a, np.array(list('ABc'), dtype=a.dtype))
        a = np.array(['abc'], dtype=dtype)
        f(a, 'A')
        assert_array_equal(a, np.array(['Abc'], dtype=a.dtype))
    def test_inout_varia(self):
        f = getattr(self.module, self.fprefix + '_inout')
        a = np.array('abc', dtype='S3')
        f(a, 'A')
        assert_array_equal(a, np.array('Abc', dtype=a.dtype))
        a = np.array(['abc'], dtype='S3')
        f(a, 'A')
        assert_array_equal(a, np.array(['Abc'], dtype=a.dtype))
        try:
            f('abc', 'A')
        except ValueError as msg:
            if not str(msg).endswith(' got 3-str'):
                raise
            else:
                raise SystemError(f'{f.__name__} should have failed on str value')
    @pytest.mark.parametrize("dtype", ['c', 'S1'])
    def test_array_inout(self, dtype):
        f = getattr(self.module, self.fprefix + '_array_inout')
        n = np.array(['A', 'B', 'C'], dtype=dtype, order='F')
        a = np.array(['a', 'b', 'c'], dtype=dtype, order='F')
        f(a, n)
        assert_array_equal(a, n)
        a = np.array(['a', 'b', 'c', 'd'], dtype=dtype)
        f(a[1:], n)
        assert_array_equal(a, np.array(['a', 'A', 'B', 'C'], dtype=dtype))
        a = np.array([['a', 'b', 'c']], dtype=dtype, order='F')
        f(a, n)
        assert_array_equal(a, np.array([['A', 'B', 'C']], dtype=dtype))
        a = np.array(['a', 'b', 'c', 'd'], dtype=dtype, order='F')

```

```

322: (8)
323: (12)
324: (8)
325: (12)
326: (20)
327: (16)
328: (8)
329: (12)
330: (16)
331: (4)
332: (4)
333: (8)
334: (8)
335: (22)
336: (21)
337: (8)
338: (22)
339: (21)
340: (8)
341: (8)
342: (4)
343: (8)
344: (8)
345: (4)
346: (4)
347: (8)
348: (8)
349: (8)
350: (4)
351: (8)
352: (8)
353: (8)
354: (0)
355: (4)
356: (4)
357: (4)
358: (7)
359: (9)
360: (9)
361: (9)
362: (9)
363: (9)
364: (11)
365: (9)
366: (7)
367: (7)
368: (9)
369: (9)
370: (9)
371: (9)
372: (9)
373: (7)
374: (7)
375: (9)
376: (9)
377: (9)
378: (9)
379: (9)
380: (9)
381: (9)
382: (7)
383: (7)
384: (9)
385: (9)
386: (9)
387: (9)
388: (11)
389: (11)
390: (13)

        try:
            f(a, n)
        except ValueError as msg:
            if not str(msg).endswith(
                'th dimension must be fixed to 3 but got 4'):
                raise
            else:
                raise SystemError(
                    f'{f.__name__} should have failed on wrong input')
    @pytest.mark.parametrize("dtype", ['c', 'S1'])
    def test_2d_array_inout(self, dtype):
        f = getattr(self.module, self.fprefix + '_2d_array_inout')
        n = np.array([[['A', 'B', 'C'],
                      ['D', 'E', 'F']],
                      dtype=dtype, order='F')
        a = np.array([[['a', 'b', 'c'],
                      ['d', 'e', 'f']],
                      dtype=dtype, order='F)')
        f(a, n)
        assert_array_equal(a, n)
    def test_return(self):
        f = getattr(self.module, self.fprefix + '_return')
        assert_equal(f('a'), b'a')
    @pytest.mark.skip('fortran function returning array segfaults')
    def test_array_return(self):
        f = getattr(self.module, self.fprefix + '_array_return')
        a = np.array(list('abc'), dtype='S1')
        assert_array_equal(f(a), a)
    def test_optional(self):
        f = getattr(self.module, self.fprefix + '_optional')
        assert_equal(f(), b"a")
        assert_equal(f(b'B'), b"B")
    class TestMiscCharacter(util.F2PyTest):
        suffix = '.f90'
        fprefix = 'test_misc_character'
        code = textwrap.dedent("""
            subroutine {fprefix}_gh18684(x, y, m)
                character(len=5), dimension(m), intent(in) :: x
                character*5, dimension(m), intent(out) :: y
                integer i, m
                !f2py integer, intent(hide), depend(x) :: m = f2py_len(x)
                do i=1,m
                    y(i) = x(i)
                end do
            end subroutine {fprefix}_gh18684
            subroutine {fprefix}_gh6308(x, i)
                integer i
                !f2py check(i>=0 && i<12) i
                character*5 name, x
                common name(12)
                name(i + 1) = x
            end subroutine {fprefix}_gh6308
            subroutine {fprefix}_gh4519(x)
                character(len=*), intent(in) :: x(:)
                !f2py intent(out) x
                integer :: i
                ! Uncomment for debug printing:
                !do i=1, size(x)
                !    print*, "x(",i,")=", x(i)
                !end do
            end subroutine {fprefix}_gh4519
            pure function {fprefix}_gh3425(x) result (y)
                character(len=*), intent(in) :: x
                character(len=len(x)) :: y
                integer :: i
                do i = 1, len(x)
                    j = iachar(x(i:i))
                    if (j>=iachar("a") .and. j<=iachar("z")) then
                        y(i:i) = achar(j-32)
                    end if
                end do
            end function {fprefix}_gh3425
        """))

```

```

391: (11)
392: (13)
393: (11)
394: (9)
395: (7)
396: (7)
397: (9)
398: (9)
399: (9)
400: (9)
401: (9)
402: (9)
403: (9)
404: (9)
405: (9)
406: (11)
407: (9)
408: (11)
409: (9)
410: (9)
411: (7)
412: (7)
413: (9)
414: (9)
415: (9)
416: (9)
z
417: (9)
418: (9)
419: (9)
420: (9)
421: (10)
422: (11)
423: (9)
424: (11)
425: (9)
426: (9)
427: (7)
428: (4)
429: (4)
430: (8)
431: (8)
432: (8)
433: (8)
434: (4)
435: (8)
436: (8)
437: (8)
438: (8)
439: (8)
440: (8)
441: (8)
442: (4)
443: (8)
444: (8)
445: (16)
446: (16)
447: (16)
448: (17)
449: (16)
450: (17)
451: (16)
452: (12)
453: (12)
454: (16)
455: (4)
456: (8)
457: (8)
458: (8)

            else
                y(i:i) = x(i:i)
            endif
        end do
    end function {fprefix}_gh3425
subroutine {fprefix}_character_bc_new(x, y, z)
    character, intent(in) :: x
    character, intent(out) :: y
    !f2py character, depend(x) :: y = x
    !f2py character, dimension((x=='a'?1:2)), depend(x), intent(out) :: z
    character, dimension(*) :: z
    !f2py character, optional, check(x == 'a' || x == 'b') :: x = 'a'
    !f2py callstatement (*f2py_func)(&x, &y, z)
    !f2py callprotoargument character*, character*, character*
    if (y.eq.x) then
        y = x
    else
        y = 'e'
    endif
    z(1) = 'c'
end subroutine {fprefix}_character_bc_new
subroutine {fprefix}_character_bc_old(x, y, z)
    character, intent(in) :: x
    character, intent(out) :: y
    !f2py character, depend(x) :: y = x[0]
    !f2py character, dimension((*x=='a'?1:2)), depend(x), intent(out) :: z
    character, dimension(*) :: z
    !f2py character, optional, check(*x == 'a' || x[0] == 'b') :: x = 'a'
    !f2py callstatement (*f2py_func)(x, y, z)
    !f2py callprotoargument char*, char*, char*
    if (y.eq.x) then
        y = x
    else
        y = 'e'
    endif
    z(1) = 'c'
end subroutine {fprefix}_character_bc_old
"""
def test_gh18684(self):
    f = getattr(self.module, self.fprefix + '_gh18684')
    x = np.array(["abcde", "fghij"], dtype='S5')
    y = f(x)
    assert_array_equal(x, y)
def test_gh6308(self):
    f = getattr(self.module, self.fprefix + '_gh6308')
    assert_equal(self.module._BLNK_.name.dtype, np.dtype('S5'))
    assert_equal(len(self.module._BLNK_.name), 12)
    f("abcde", 0)
    assert_equal(self.module._BLNK_.name[0], b"abcde")
    f("12345", 5)
    assert_equal(self.module._BLNK_.name[5], b"12345")
def test_gh4519(self):
    f = getattr(self.module, self.fprefix + '_gh4519')
    for x, expected in [
        ('a', dict(shape=(), dtype=np.dtype('S1'))),
        ('text', dict(shape=(), dtype=np.dtype('S4'))),
        (np.array(['1', '2', '3']), dict(shape='S1'),
         dict(shape=(3,), dtype=np.dtype('S1'))),
        (['1', '2', '34'], dict(shape=(3,), dtype=np.dtype('S2'))),
        ([' ', ''], dict(shape=(2,), dtype=np.dtype('S1')))]:
        r = f(x)
        for k, v in expected.items():
            assert_equal(getattr(r, k), v)
def test_gh3425(self):
    f = getattr(self.module, self.fprefix + '_gh3425')
    assert_equal(f('abC'), b'ABC')
    assert_equal(f(''), b'')
```

```

459: (8)             assert_equal(f('abC12d'), b'ABC12D')
460: (4)             @pytest.mark.parametrize("state", ['new', 'old'])
461: (4)             def test_character_bc(self, state):
462: (8)                 f = getattr(self.module, self.fprefix + '_character_bc_' + state)
463: (8)                 c, a = f()
464: (8)                 assert_equal(c, b'a')
465: (8)                 assert_equal(len(a), 1)
466: (8)                 c, a = f(b'b')
467: (8)                 assert_equal(c, b'b')
468: (8)                 assert_equal(len(a), 2)
469: (8)                 assert_raises(Exception, lambda: f(b'c'))
470: (0)             class TestStringScalarArr(util.F2PyTest):
471: (4)                 sources = [util.getpath("tests", "src", "string", "scalar_string.f90")]
472: (4)                 def test_char(self):
473: (8)                     for out in (self.module.string_test.string,
474: (20)                         self.module.string_test.string77):
475: (12)                         expected = ()
476: (12)                         assert out.shape == expected
477: (12)                         expected = '|S8'
478: (12)                         assert out.dtype == expected
479: (4)                 def test_char_arr(self):
480: (8)                     for out in (self.module.string_test.strarr,
481: (20)                         self.module.string_test.strarr77):
482: (12)                         expected = (5,7)
483: (12)                         assert out.shape == expected
484: (12)                         expected = '|S12'
485: (12)                         assert out.dtype == expected
486: (0)             class TestStringAssumedLength(util.F2PyTest):
487: (4)                 sources = [util.getpath("tests", "src", "string", "gh24008.f")]
488: (4)                 def test_gh24008(self):
489: (8)                     self.module.greet("joe", "bob")
490: (0)             class TestStringOptionalInOut(util.F2PyTest):
491: (4)                 sources = [util.getpath("tests", "src", "string", "gh24662.f90")]
492: (4)                 def test_gh24662(self):
493: (8)                     self.module.string_inout_optional()
494: (8)                     a = np.array('hi', dtype='S32')
495: (8)                     self.module.string_inout_optional(a)
496: (8)                     assert "output string" in a.tobytes().decode()
497: (8)                     with pytest.raises(Exception):
498: (12)                         aa = "Hi"
499: (12)                         self.module.string_inout_optional(aa)
500: (0)             @pytest.mark.slow
501: (0)             class TestNewCharHandling(util.F2PyTest):
502: (4)                 sources = [
503: (8)                     util.getpath("tests", "src", "string", "gh25286.pyf"),
504: (8)                     util.getpath("tests", "src", "string", "gh25286.f90")
505: (4)                 ]
506: (4)                 module_name = "_char_handling_test"
507: (4)                 def test_gh25286(self):
508: (8)                     info = self.module.charint('T')
509: (8)                     assert info == 2
510: (0)             @pytest.mark.slow
511: (0)             class TestBCCCharHandling(util.F2PyTest):
512: (4)                 sources = [
513: (8)                     util.getpath("tests", "src", "string", "gh25286_bc.pyf"),
514: (8)                     util.getpath("tests", "src", "string", "gh25286.f90")
515: (4)                 ]
516: (4)                 module_name = "_char_handling_test"
517: (4)                 def test_gh25286(self):
518: (8)                     info = self.module.charint('T')
519: (8)                     assert info == 2

```

-----  
File 168 - test\_crackfortran.py:

```

1: (0)             import importlib
2: (0)             import codecs
3: (0)             import time

```

```

4: (0) import unicodedata
5: (0) import pytest
6: (0) import numpy as np
7: (0) from numpy.f2py.crackfortran import markinnernspaces, nameargspattern
8: (0) from . import util
9: (0) from numpy.f2py import crackfortran
10: (0) import textwrap
11: (0) import contextlib
12: (0) import io
13: (0) class TestNoSpace(util.F2PyTest):
14: (4)     sources = [util.getpath("tests", "src", "crackfortran", "gh15035.f")]
15: (4)     def test_module(self):
16: (8)         k = np.array([1, 2, 3], dtype=np.float64)
17: (8)         w = np.array([1, 2, 3], dtype=np.float64)
18: (8)         self.module.subb(k)
19: (8)         assert np.allclose(k, w + 1)
20: (8)         self.module.subc([w, k])
21: (8)         assert np.allclose(k, w + 1)
22: (8)         assert self.module.t0("23") == b"2"
23: (0) class TestPublicPrivate:
24: (4)     def test_defaultPrivate(self):
25: (8)         fpath = util.getpath("tests", "src", "crackfortran", "privatemod.f90")
26: (8)         mod = crackfortran.crackfortran([str(fpath)])
27: (8)         assert len(mod) == 1
28: (8)         mod = mod[0]
29: (8)         assert "private" in mod["vars"]["a"]["attrspec"]
30: (8)         assert "public" not in mod["vars"]["a"]["attrspec"]
31: (8)         assert "private" in mod["vars"]["b"]["attrspec"]
32: (8)         assert "public" not in mod["vars"]["b"]["attrspec"]
33: (8)         assert "private" not in mod["vars"]["seta"]["attrspec"]
34: (8)         assert "public" in mod["vars"]["seta"]["attrspec"]
35: (4)     def test_defaultPublic(self, tmp_path):
36: (8)         fpath = util.getpath("tests", "src", "crackfortran", "publicmod.f90")
37: (8)         mod = crackfortran.crackfortran([str(fpath)])
38: (8)         assert len(mod) == 1
39: (8)         mod = mod[0]
40: (8)         assert "private" in mod["vars"]["a"]["attrspec"]
41: (8)         assert "public" not in mod["vars"]["a"]["attrspec"]
42: (8)         assert "private" not in mod["vars"]["seta"]["attrspec"]
43: (8)         assert "public" in mod["vars"]["seta"]["attrspec"]
44: (4)     def test_access_type(self, tmp_path):
45: (8)         fpath = util.getpath("tests", "src", "crackfortran", "accesstype.f90")
46: (8)         mod = crackfortran.crackfortran([str(fpath)])
47: (8)         assert len(mod) == 1
48: (8)         tt = mod[0]['vars']
49: (8)         assert set(tt['a']['attrspec']) == {'private', 'bind(c)'}
50: (8)         assert set(tt['b_']['attrspec']) == {'public', 'bind(c)'}
51: (8)         assert set(tt['c']['attrspec']) == {'public'}
52: (4)     def test_nowrap_private_procedures(self, tmp_path):
53: (8)         fpath = util.getpath("tests", "src", "crackfortran", "gh23879.f90")
54: (8)         mod = crackfortran.crackfortran([str(fpath)])
55: (8)         assert len(mod) == 1
56: (8)         pyf = crackfortran.crack2fortran(mod)
57: (8)         assert 'bar' not in pyf
58: (0) class TestModuleProcedure():
59: (4)     def test_moduleOperators(self, tmp_path):
60: (8)         fpath = util.getpath("tests", "src", "crackfortran", "operators.f90")
61: (8)         mod = crackfortran.crackfortran([str(fpath)])
62: (8)         assert len(mod) == 1
63: (8)         mod = mod[0]
64: (8)         assert "body" in mod and len(mod["body"]) == 9
65: (8)         assert mod["body"][1]["name"] == "operator(.item.)"
66: (8)         assert "implementedby" in mod["body"][1]
67: (8)         assert mod["body"][1]["implementedby"] == \
68: (12)             ["item_int", "item_real"]
69: (8)         assert mod["body"][2]["name"] == "operator(==)"
70: (8)         assert "implementedby" in mod["body"][2]
71: (8)         assert mod["body"][2]["implementedby"] == ["items_are_equal"]
72: (8)         assert mod["body"][3]["name"] == "assignment(-)"

```

```

73: (8)             assert "implementedby" in mod["body"][3]
74: (8)             assert mod["body"][3]["implementedby"] == \
75: (12)                 ["get_int", "get_real"]
76: (4)             def test_notPublicPrivate(self, tmp_path):
77: (8)                 fpath = util.getpath("tests", "src", "crackfortran", "pubprivmod.f90")
78: (8)                 mod = crackfortran.crackfortran([str(fpath)])
79: (8)                 assert len(mod) == 1
80: (8)                 mod = mod[0]
81: (8)                 assert mod['vars']['a']['atrspec'] == ['private', ]
82: (8)                 assert mod['vars']['b']['atrspec'] == ['public', ]
83: (8)                 assert mod['vars']['seta']['atrspec'] == ['public', ]
84: (0)             class TestExternal(util.F2PyTest):
85: (4)                 sources = [util.getpath("tests", "src", "crackfortran", "gh17859.f")]
86: (4)                 def test_external_as_statement(self):
87: (8)                     def incr(x):
88: (12)                         return x + 123
89: (8)                     r = self.module.external_as_statement(incr)
90: (8)                     assert r == 123
91: (4)                     def test_external_as_attribute(self):
92: (8)                         def incr(x):
93: (12)                             return x + 123
94: (8)                             r = self.module.external_as_attribute(incr)
95: (8)                             assert r == 123
96: (0)             class TestCrackFortran(util.F2PyTest):
97: (4)                 sources = [util.getpath("tests", "src", "crackfortran", "gh2848.f90")]
98: (4)                 def test_gh2848(self):
99: (8)                     r = self.module.gh2848(1, 2)
100: (8)                    assert r == (1, 2)
101: (0)             class TestMarkinnerspaces:
102: (4)                 def test_do_not_touch_normal_spaces(self):
103: (8)                     test_list = ["a ", " a", "a b c", "'abcdefg'hij'"]
104: (8)                     for i in test_list:
105: (12)                         assert markinnerspaces(i) == i
106: (4)                 def test_one_relevant_space(self):
107: (8)                     assert markinnerspaces("a 'b c' \\' \\'") == "a 'b@_@c' \\' \\'"
108: (8)                     assert markinnerspaces(r'a "b c" \" \"') == r'a "b@_@c" \" \"'
109: (4)                 def test_ignore_inner_quotes(self):
110: (8)                     assert markinnerspaces("a 'b c\" \" d' e") == "a 'b@_@c\" @_@\" @_@d' e"
111: (8)                     assert markinnerspaces("a \"b c' ' d\" e") == "a \"b@_@c' @_@' @_@d\" e"
112: (4)                 def test_multiple_relevant_spaces(self):
113: (8)                     assert markinnerspaces("a 'b c' 'd e'") == "a 'b@_@c' 'd@_@e'"
114: (8)                     assert markinnerspaces(r'a "b c" "d e'") == r'a "b@_@c" "d@_@e"'
115: (0)             class TestDimSpec(util.F2PyTest):
116: (4)                 """This test suite tests various expressions that are used as dimension
117: (4)                 specifications.
118: (4)                 There exists two usage cases where analyzing dimensions
119: (4)                 specifications are important.
120: (4)                 In the first case, the size of output arrays must be defined based
121: (4)                 on the inputs to a Fortran function. Because Fortran supports
122: (4)                 arbitrary bases for indexing, for instance, `arr(lower:upper)`
123: (4)                 f2py has to evaluate an expression `upper - lower + 1` where
124: (4)                 `lower` and `upper` are arbitrary expressions of input parameters.
125: (4)                 The evaluation is performed in C, so f2py has to translate Fortran
126: (4)                 expressions to valid C expressions (an alternative approach is
127: (4)                 that a developer specifies the corresponding C expressions in a
128: (4)                 .pyf file).
129: (4)                 In the second case, when user provides an input array with a given
130: (4)                 size but some hidden parameters used in dimensions specifications
131: (4)                 need to be determined based on the input array size. This is a
132: (4)                 harder problem because f2py has to solve the inverse problem: find
133: (4)                 a parameter `p` such that `upper(p) - lower(p) + 1` equals to the
134: (4)                 size of input array. In the case when this equation cannot be
135: (4)                 solved (e.g. because the input array size is wrong), raise an
136: (4)                 error before calling the Fortran function (that otherwise would
137: (4)                 likely crash Python process when the size of input arrays is
138: (4)                 wrong). f2py currently supports this case only when the equation
139: (4)                 is linear with respect to unknown parameter.
140: (4)                 """
141: (4)             suffix = ".f90"

```

```

142: (4)
143: (6)
144: (8)
145: (8)
146: (8)
147: (8)
148: (6)
149: (6)
150: (8)
151: (8)
152: (8)
153: (8)
154: (6)
155: (4)
156: (4)
157: (8)
158: (8)
159: (4)
160: (4)
161: (4)
162: (4)
163: (4)
164: (8)
dimspec.split(',')])
165: (8)
166: (12)
167: (12)
168: (12)
169: (8)
170: (4)
171: (4)
172: (8)
173: (8)
174: (8)
175: (12)
176: (12)
177: (4)
178: (4)
179: (8)
180: (8)
181: (8)
182: (8)
183: (12)
184: (12)
185: (16)
186: (12)
187: (16)
188: (12)
189: (12)
190: (0)
class TestModuleDeclaration:
191: (4)
def test_dependencies(self, tmp_path):
192: (8)
    fpath = util.getpath("tests", "src", "crackfortran", "foo_deps.f90")
193: (8)
    mod = crackfortran.crackfortran([str(fpath)])
194: (8)
    assert len(mod) == 1
195: (8)
    assert mod[0]["vars"]["abar"]["="] == "bar('abar')"
196: (0)
class TestEval(util.F2PyTest):
197: (4)
def test_eval_scalar(self):
198: (8)
    eval_scalar = crackfortran._eval_scalar
199: (8)
    assert eval_scalar('123', {}) == '123'
200: (8)
    assert eval_scalar('12 + 3', {}) == '15'
201: (8)
    assert eval_scalar('a + b', dict(a=1, b=2)) == '3'
202: (8)
    assert eval_scalar('"123"', {}) == "'123'"
203: (0)
class TestFortranReader(util.F2PyTest):
204: (4)
@pytest.mark.parametrize("encoding",
205: (29)
                        ['ascii', 'utf-8', 'utf-16', 'utf-32'])
206: (4)
def test_input_encoding(self, tmp_path, encoding):
207: (8)
    f_path = tmp_path / f"input_with_{encoding}_encoding.f90"
208: (8)
    with f_path.open('w', encoding=encoding) as ff:
209: (12)
        ff.write("""

```

```

210: (21)                     subroutine foo()
211: (21)                     end subroutine foo
212: (21)
213: (8)                      mod = crackfortran.crackfortran([str(f_path)])
214: (8)                      assert mod[0]['name'] == 'foo'
215: (0)                      class TestUnicodeComment(util.F2PyTest):
216: (4)                        sources = [util.getpath("tests", "src", "crackfortran",
217: (4)                          "unicode_comment.f90")]
218: (4)                          @pytest.mark.skipif(
219: (8)                            importlib.util.find_spec("charset_normalizer") is None),
220: (8)                            reason="test requires charset_normalizer which is not installed",
221: (4)                          )
222: (8)                        def test_encoding_comment(self):
223: (0)                          self.module.foo(3)
224: (4)                          class TestNameArgsPatternBacktracking:
225: (8)                            @pytest.mark.parametrize(
226: (8)                              ['adversary'],
227: (12)                             [
228: (12)                               ('@)@bind@(('@, ),
229: (12)                               ('@)@bind
230: (8)                                 '@(@', ),
231: (4)                               ('@)@bind foo bar baz@('@, )
232: (4)                             ]
233: (4)                           )
234: (8)                           def test_nameargspattern_backtracking(self, adversary):
235: (8)                             '''address ReDOS vulnerability:
236: (8)                             https://github.com/numpy/numpy/issues/23338'''
237: (8)                             trials_per_batch = 12
238: (8)                             batches_per_regex = 4
239: (12)                            start_reps, end_reps = 15, 25
240: (12)                            for ii in range(start_reps, end_reps):
241: (16)                              repeated_adversary = adversary * ii
242: (16)                              for _ in range(batches_per_regex):
243: (20)                                times = []
244: (20)                                for _ in range(trials_per_batch):
245: (20)                                  t0 = time.perf_counter()
246: (16)                                  mtch = nameargspattern.search(repeated_adversary)
247: (12)                                  times.append(time.perf_counter() - t0)
248: (12)                                  assert np.median(times) < 0.2
249: (12)                                  assert not mtch
250: (0)                                  good_version_of_adversary = repeated_adversary + '@)@'
251: (4)                                  assert nameargspattern.search(good_version_of_adversary)
252: (4)                           class TestFunctionReturn(util.F2PyTest):
253: (8)                             sources = [util.getpath("tests", "src", "crackfortran", "gh23598.f90")]
254: (0)                             def test_function_retttype(self):
255: (4)                               assert self.module.intproduct(3, 4) == 12
256: (0)                           class TestFortranGroupCounters(util.F2PyTest):
257: (4)                             def test_end_if_comment(self):
258: (8)                               fpath = util.getpath("tests", "src", "crackfortran", "gh23533.f")
259: (8)                               try:
260: (12)                                 crackfortran.crackfortran([str(fpath)])
261: (0)                               except Exception as exc:
262: (4)                                 assert False, f"'crackfortran.crackfortran' raised an exception
263: {exc}"
264: (0)                           class TestF77CommonBlockReader():
265: (4)                             def test_gh22648(self, tmp_path):
266: (8)                               fpath = util.getpath("tests", "src", "crackfortran", "gh22648.pyf")
267: (8)                               with contextlib.redirect_stdout(io.StringIO()) as stdout_f2py:
268: (12)                                 mod = crackfortran.crackfortran([str(fpath)])
269: (8)                                 assert "Mismatch" not in stdout_f2py.getvalue()

```

-----  
File 169 - test\_compile\_function.py:

```

1: (0)                     """See https://github.com/numpy/numpy/pull/11937.
2: (0)                     """
3: (0)                     import sys
4: (0)                     import os
5: (0)                     import uuid

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

6: (0)         from importlib import import_module
7: (0)         import pytest
8: (0)         import numpy.f2py
9: (0)         from . import util
10: (0)        def setup_module():
11: (4)          if not util.has_c_compiler():
12: (8)            pytest.skip("Needs C compiler")
13: (4)          if not util.has_f77_compiler():
14: (8)            pytest.skip("Needs FORTRAN 77 compiler")
15: (0)        @pytest.mark.parametrize("extra_args",
16: (25)                  [[>--noopt", "--debug"], "--noopt --debug", ""])
17: (0)        @pytest.mark.leaks_references(reason="Imported module seems never deleted.")
18: (0)        def test_f2py_init_compile(extra_args):
19: (4)          fsource = """
20: (8)              integer function foo()
21: (8)              foo = 10 + 5
22: (8)              return
23: (8)              end
24: (4)
25: (4)          moddir = util.get_module_dir()
26: (4)          modname = util.get_temp_module_name()
27: (4)          cwd = os.getcwd()
28: (4)          target = os.path.join(moddir, str(uuid.uuid4()) + ".f")
29: (4)          for source_fn in [target, None]:
30: (8)              with util.switchdir(moddir):
31: (12)                ret_val = numpy.f2py.compile(fsource,
32: (41)                                modulename=modname,
33: (41)                                extra_args=extra_args,
34: (41)                                source_fn=source_fn)
35: (12)                assert ret_val == 0
36: (4)          if sys.platform != "win32":
37: (8)              return_check = import_module(modname)
38: (8)              calc_result = return_check.foo()
39: (8)              assert calc_result == 15
40: (8)              del sys.modules[modname]
41: (0)          def test_f2py_init_compile_failure():
42: (4)              ret_val = numpy.f2py.compile(b"invalid")
43: (4)              assert ret_val == 1
44: (0)          def test_f2py_init_compile_bad_cmd():
45: (4)              try:
46: (8)                  temp = sys.executable
47: (8)                  sys.executable = "does not exist"
48: (8)                  ret_val = numpy.f2py.compile(b"invalid")
49: (8)                  assert ret_val == 127
50: (4)                  finally:
51: (8)                      sys.executable = temp
52: (0)          @pytest.mark.parametrize(
53: (4)              "fsource",
54: (4)              [
55: (8)                  "program test_f2py\nend program test_f2py",
56: (8)                  b"program test_f2py\nend program test_f2py",
57: (4)              ],
58: (0)
59: (0)          )
60: (4)          def test_compile_from_strings(tmpdir, fsource):
61: (8)              with util.switchdir(tmpdir):
62: (37)                ret_val = numpy.f2py.compile(fsource,
63: (37)                                modulename="test_compile_from_strings",
64: (8)                                extension=".f90")
64: (8)                assert ret_val == 0

```

---

File 170 - test\_data.py:

```

1: (0)          import os
2: (0)          import pytest
3: (0)          import numpy as np
4: (0)          from . import util
5: (0)          from numpy.f2py.crackfortran import crackfortran

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

6: (0)         class TestData(util.F2PyTest):
7: (4)             sources = [util.getpath("tests", "src", "crackfortran", "data_stmts.f90")]
8: (4)             def test_data_stmts(self):
9: (8)                 assert self.module.complexdat.i == 2
10: (8)                assert self.module.complexdat.j == 3
11: (8)                assert self.module.complexdat.x == 1.5
12: (8)                assert self.module.complexdat.y == 2.0
13: (8)                assert self.module.complexdat.pi ==
3.1415926535897932384626433832795028841971693993751058209749445923078164062
14: (8)                    assert self.module.complexdat.medium_ref_index == np.array(1.+0.j)
15: (8)                    assert np.all(self.module.complexdat.z == np.array([3.5, 7.0]))
16: (8)                    assert np.all(self.module.complexdat.my_array == np.array([-3.+4.j]))
17: (8)                    assert np.all(self.module.complexdat.my_real_array == np.array([1., 2., 3.]))
18: (8)                    assert np.all(self.module.complexdat.ref_index_one == np.array([13.0 + 21.0j]))
19: (8)                    assert np.all(self.module.complexdat.ref_index_two == np.array([-30.0 + 43.0j]))
20: (4)            def test_crackedlines(self):
21: (8)                mod = crackfortran(self.sources)
22: (8)                assert mod[0]['vars']['x']['='] == '1.5'
23: (8)                assert mod[0]['vars']['y']['='] == '2.0'
24: (8)                assert mod[0]['vars']['pi']['='] ==
'3.1415926535897932384626433832795028841971693993751058209749445923078164062d0'
25: (8)                    assert mod[0]['vars']['my_real_array']['='] == '(/1.0d0, 2.0d0, 3.0d0)'
26: (8)                    assert mod[0]['vars']['ref_index_one']['='] == '(13.0d0, 21.0d0)'
27: (8)                    assert mod[0]['vars']['ref_index_two']['='] == '(-30.0d0, 43.0d0)'
28: (8)                    assert mod[0]['vars']['my_array']['='] == '(/(1.0d0, 2.0d0), (-3.0d0, 4.0d0))'
29: (8)                    assert mod[0]['vars']['z']['='] == '(/3.5, 7.0)'
30: (0)            class TestDataF77(util.F2PyTest):
31: (4)                sources = [util.getpath("tests", "src", "crackfortran", "data_common.f")]
32: (4)                def test_data_stmts(self):
33: (8)                    assert self.module.mycom.mydata == 0
34: (4)                def test_crackedlines(self):
35: (8)                    mod = crackfortran(str(self.sources[0]))
36: (8)                    print(mod[0]['vars'])
37: (8)                    assert mod[0]['vars']['mydata']['='] == '0'
38: (0)            class TestDataMultiplierF77(util.F2PyTest):
39: (4)                sources = [util.getpath("tests", "src", "crackfortran",
"data_multiplier.f")]
40: (4)                def test_data_stmts(self):
41: (8)                    assert self.module.mycom.ivar1 == 3
42: (8)                    assert self.module.mycom.ivar2 == 3
43: (8)                    assert self.module.mycom.ivar3 == 2
44: (8)                    assert self.module.mycom.ivar4 == 2
45: (8)                    assert self.module.mycom.evar5 == 0
46: (0)            class TestDataWithCommentsF77(util.F2PyTest):
47: (4)                sources = [util.getpath("tests", "src", "crackfortran",
"data_with_comments.f")]
48: (4)                def test_data_stmts(self):
49: (8)                    assert len(self.module.mycom.mytab) == 3
50: (8)                    assert self.module.mycom.mytab[0] == 0
51: (8)                    assert self.module.mycom.mytab[1] == 4
52: (8)                    assert self.module.mycom.mytab[2] == 0

```

---

## File 171 - test\_docs.py:

```

1: (0)         import os
2: (0)         import pytest
3: (0)         import numpy as np
4: (0)         from numpy.testing import assert_array_equal, assert_equal
5: (0)         from . import util
6: (0)         def get_docdir():
7: (4)             return os.path.abspath(os.path.join(

```

```

8: (8)             os.path.dirname(__file__),
9: (8)             '...', '...', '...',
10: (8)             'doc', 'source', 'f2py', 'code'))
11: (0)         pytestmark = pytest.mark.skipif(
12: (4)             not os.path.isdir(get_docdir()),
13: (4)             reason='Could not find f2py documentation sources'
14: (12)             f' ({get_docdir()} does not exists)')
15: (0)         def _path(*a):
16: (4)             return os.path.join(*((get_docdir(),) + a))
17: (0)         class TestDocAdvanced(util.F2PyTest):
18: (4)             sources = [_path('asterisk1.f90'), _path('asterisk2.f90'),
19: (15)                 _path('ftype.f')]
20: (4)             def test_asterisk1(self):
21: (8)                 foo = getattr(self.module, 'foo1')
22: (8)                 assert_equal(foo(), b'123456789A12')
23: (4)             def test_asterisk2(self):
24: (8)                 foo = getattr(self.module, 'foo2')
25: (8)                 assert_equal(foo(2), b'12')
26: (8)                 assert_equal(foo(12), b'123456789A12')
27: (8)                 assert_equal(foo(24), b'123456789A123456789B')
28: (4)             def test_ftype(self):
29: (8)                 ftype = self.module
30: (8)                 ftype.foo()
31: (8)                 assert_equal(ftype.data.a, 0)
32: (8)                 ftype.data.a = 3
33: (8)                 ftype.data.x = [1, 2, 3]
34: (8)                 assert_equal(ftype.data.a, 3)
35: (8)                 assert_array_equal(ftype.data.x,
36: (27)                     np.array([1, 2, 3], dtype=np.float32))
37: (8)                 ftype.data.x[1] = 45
38: (8)                 assert_array_equal(ftype.data.x,
39: (27)                     np.array([1, 45, 3], dtype=np.float32))

```

-----  
File 172 - test\_f2cmap.py:

```

1: (0)             from . import util
2: (0)             import numpy as np
3: (0)             class TestF2Cmap(util.F2PyTest):
4: (4)                 sources = [
5: (8)                     util.getpath("tests", "src", "f2cmap", "isoFortranEnvMap.f90"),
6: (8)                     util.getpath("tests", "src", "f2cmap", ".f2py_f2cmap")
7: (4)                 ]
8: (4)                 def test_long_long_map(self):
9: (8)                     inp = np.ones(3)
10: (8)                    out = self.module.func1(inp)
11: (8)                    exp_out = 3
12: (8)                    assert out == exp_out

```

-----  
File 173 - test\_f2py2e.py:

```

1: (0)             import textwrap, re, sys, subprocess, shlex
2: (0)             from pathlib import Path
3: (0)             from collections import namedtuple
4: (0)             import platform
5: (0)             import pytest
6: (0)             from . import util
7: (0)             from numpy.f2py.f2py2e import main as f2pycli
8: (0)             PPaths = namedtuple("PPaths", "finp, f90inp, pyf, wrap77, wrap90, cmodf")
9: (0)             def get_io_paths(fname_inp, mname="untitled"):
10: (4)                 """Takes in a temporary file for testing and returns the expected output
and input paths
11: (4)                 Here expected output is essentially one of any of the possible generated
files.
12: (4)                 ..note::
13: (4)                     Since this does not actually run f2py, none of these are guaranteed

```

```

to
15: (9)           exist, and module names are typically incorrect
16: (4)
17: (4)
18: (4)
19: (16)          Parameters
20: (4)           -----
21: (4)           fname_inp : str
22: (4)             The input filename
23: (4)           mname : str, optional
24: (4)             The name of the module, untitled by default
25: (4)
26: (4)
27: (4)           Returns
28: (4)           -----
29: (4)           genp : NamedTuple PPaths
30: (4)             The possible paths which are generated, not all of which exist
31: (4)
32: (4)
33: (4)
34: (4)
35: (4)
36: (0) @pytest.fixture(scope="session")
37: (0) def hello_world_f90(tmpdir_factory):
38: (4)     """Generates a single f90 file for testing"""
39: (4)     fdat = util.getpath("tests", "src", "cli", "hiworld.f90").read_text()
40: (4)     fn = tmpdir_factory.getbasetemp() / "hello.f90"
41: (4)     fn.write_text(fdat, encoding="ascii")
42: (4)     return fn
43: (0)
44: (0) @pytest.fixture(scope="session")
45: (4) def gh23598_warn(tmpdir_factory):
46: (4)     """F90 file for testing warnings in gh23598"""
47: (4)     fdat = util.getpath("tests", "src", "crackfortran",
48: (4)         "gh23598Warn.f90").read_text()
49: (4)     fn = tmpdir_factory.getbasetemp() / "gh23598Warn.f90"
50: (4)     fn.write_text(fdat, encoding="ascii")
51: (4)     return fn
52: (0) @pytest.fixture(scope="session")
53: (0) def gh22819_cli(tmpdir_factory):
54: (4)     """F90 file for testing disallowed CLI arguments in ghff819"""
55: (4)     fdat = util.getpath("tests", "src", "cli", "gh_22819.pyf").read_text()
56: (4)     fn = tmpdir_factory.getbasetemp() / "gh_22819.pyf"
57: (4)     fn.write_text(fdat, encoding="ascii")
58: (0) @pytest.fixture(scope="session")
59: (0) def hello_world_f77(tmpdir_factory):
60: (4)     """Generates a single f77 file for testing"""
61: (4)     fdat = util.getpath("tests", "src", "cli", "hi77.f").read_text()
62: (4)     fn = tmpdir_factory.getbasetemp() / "hello.f"
63: (4)     fn.write_text(fdat, encoding="ascii")
64: (0)
65: (0) @pytest.fixture(scope="session")
66: (4) def retreal_f77(tmpdir_factory):
67: (4)     """Generates a single f77 file for testing"""
68: (4)     fdat = util.getpath("tests", "src", "return_real", "foo77.f").read_text()
69: (4)     fn = tmpdir_factory.getbasetemp() / "foo.f"
70: (4)     fn.write_text(fdat, encoding="ascii")
71: (0)
72: (0) @pytest.fixture(scope="session")
73: (0) def f2cmap_f90(tmpdir_factory):
74: (4)     """Generates a single f90 file for testing"""
75: (4)     fdat = util.getpath("tests", "src", "f2cmap",
76: (4)         "isoFortranEnvMap.f90").read_text()
77: (4)     f2cmap = util.getpath("tests", "src", "f2cmap",
78: (4)         ".f2py_f2cmap").read_text()
79: (4)     fn = tmpdir_factory.getbasetemp() / "f2cmap.f90"
80: (4)     fmap = tmpdir_factory.getbasetemp() / "mapfile"
81: (4)     fn.write_text(fdat, encoding="ascii")
82: (4)     fmap.write_text(f2cmap, encoding="ascii")

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

80: (4)           return fn
81: (0)           def test_gh22819_cli(capfd, gh22819_cli, monkeypatch):
82: (4)             """Check that module names are handled correctly
83: (4)             gh-22819
84: (4)             Essentially, the -m name cannot be used to import the module, so the
85: (4)             module
86: (4)             named in the .pyf needs to be used instead
87: (4)             CLI :: -m and a .pyf file
88: (4)             """
89: (4)             ipath = Path(gh22819_cli)
90: (4)             monkeypatch setattr(sys, "argv", f"f2py -m blah {ipath}'.split())
91: (8)             with util.switchdir(ipath.parent):
92: (8)               f2pycli()
93: (8)               gen_paths = [item.name for item in ipath.parent.rglob("*") if
94: (8)                 assert "blahmodule.c" not in gen_paths # shouldn't be generated
95: (8)                 assert "blah-f2pywrappers.f" not in gen_paths
96: (8)                 assert "test_22819-f2pywrappers.f" in gen_paths
97: (8)                 assert "test_22819module.c" in gen_paths
98: (8)                 assert "Ignoring blah"
99: (0)             def test_gh22819_many_pyf(capfd, gh22819_cli, monkeypatch):
100: (4)               """Only one .pyf file allowed
101: (4)               gh-22819
102: (4)               CLI :: .pyf files
103: (4)               """
104: (4)               ipath = Path(gh22819_cli)
105: (4)               monkeypatch setattr(sys, "argv", f"f2py -m blah {ipath}
hello.pyf'.split())
106: (8)               with util.switchdir(ipath.parent):
107: (12)                 with pytest.raises(ValueError, match="Only one .pyf file per call"):
108: (0)                   f2pycli()
109: (4)             def test_gh23598_warn(capfd, gh23598_warn, monkeypatch):
110: (4)               foutl = get_io_paths(gh23598_warn, mname="test")
111: (4)               ipath = foutl.f90inp
112: (8)               monkeypatch setattr(
113: (8)                 sys, "argv",
114: (4)                 f'f2py {ipath} -m test'.split())
115: (8)               with util.switchdir(ipath.parent):
116: (8)                 f2pycli() # Generate files
117: (8)                 wrapper = foutl.wrap90.read_text()
118: (0)                 assert "intproductf2pywrap, intpr" not in wrapper
119: (4)             def test_gen_pyf(capfd, hello_world_f90, monkeypatch):
120: (4)               """Ensures that a signature file is generated via the CLI
121: (4)               CLI :: -h
122: (4)               """
123: (4)               ipath = Path(hello_world_f90)
124: (4)               opath = Path(hello_world_f90).stem + ".pyf"
125: (4)               monkeypatch setattr(sys, "argv", f'f2py -h {opath} {ipath}'.split())
126: (8)               with util.switchdir(ipath.parent):
127: (8)                 f2pycli() # Generate wrappers
128: (8)                 out, _ = capfd.readouterr()
129: (8)                 assert "Saving signatures to file" in out
130: (0)                 assert Path(f'{opath)').exists()
131: (4)             def test_gen_pyf_stdout(capfd, hello_world_f90, monkeypatch):
132: (4)               """Ensures that a signature file can be dumped to stdout
133: (4)               CLI :: -h
134: (4)               """
135: (4)               ipath = Path(hello_world_f90)
136: (4)               monkeypatch setattr(sys, "argv", f'f2py -h stdout {ipath}'.split())
137: (8)               with util.switchdir(ipath.parent):
138: (8)                 f2pycli()
139: (8)                 out, _ = capfd.readouterr()
140: (8)                 assert "Saving signatures to file" in out
141: (0)                 assert "function hi() ! in " in out
142: (4)             def test_gen_pyf_no_overwrite(capfd, hello_world_f90, monkeypatch):
143: (4)               """Ensures that the CLI refuses to overwrite signature files
144: (4)               CLI :: -h without --overwrite-signature
145: (4)               """

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

146: (4)         monkeypatch setattr(sys, "argv", f'f2py -h faker.pyf {ipath}'.split())
147: (4)         with util.switchdir(ipath.parent):
148: (8)             Path("faker.pyf").write_text("Fake news", encoding="ascii")
149: (8)             with pytest.raises(SystemExit):
150: (12)                 f2pycli() # Refuse to overwrite
151: (12)                 _, err = capfd.readouterr()
152: (12)                 assert "Use --overwrite-signature to overwrite" in err
153: (0)         @pytest.mark.skipif((platform.system() != 'Linux') or (sys.version_info <= (3,
12)), reason='Compiler and 3.12 required')
154: (20)
155: (0)     def testUntitled_cli(capfd, hello_world_f90, monkeypatch):
156: (4)         """Check that modules are named correctly
157: (4)         CLI :: defaults
158: (4)         """
159: (4)         ipath = Path(hello_world_f90)
160: (4)         monkeypatch setattr(sys, "argv", f"f2py --backend meson -c
{ipath}.split()")
161: (4)         with util.switchdir(ipath.parent):
162: (8)             f2pycli()
163: (8)             out, _ = capfd.readouterr()
164: (8)             assert "untitledmodule.c" in out
165: (0)         @pytest.mark.skipif((platform.system() != 'Linux') or (sys.version_info <= (3,
12)), reason='Compiler and 3.12 required')
166: (0)     def testNo_py312_distutils_fcompiler(capfd, hello_world_f90, monkeypatch):
167: (4)         """Check that no distutils imports are performed on 3.12
168: (4)         CLI :: --fcompiler --help-link --backend distutils
169: (4)         """
170: (4)         MNAME = "hi"
171: (4)         fout1 = get_io_paths(hello_world_f90, mname=MNAME)
172: (4)         ipath = fout1.f90inp
173: (4)         monkeypatch setattr(
174: (8)             sys, "argv", f"f2py {ipath} -c --fcompiler=gfortran -m
{MNAME}.split()"
175: (4)
176: (4)         with util.switchdir(ipath.parent):
177: (8)             f2pycli()
178: (8)             out, _ = capfd.readouterr()
179: (8)             assert "--fcompiler cannot be used with meson" in out
180: (4)         monkeypatch setattr(
181: (8)             sys, "argv", f"f2py --help-link".split()
182: (4)
183: (4)         with util.switchdir(ipath.parent):
184: (8)             f2pycli()
185: (8)             out, _ = capfd.readouterr()
186: (8)             assert "Use --dep for meson builds" in out
187: (4)             MNAME = "hi2" # Needs to be different for a new -c
188: (4)             monkeypatch setattr(
189: (8)                 sys, "argv", f"f2py {ipath} -c -m {MNAME} --backend distutils".split()
190: (4)
191: (4)         with util.switchdir(ipath.parent):
192: (8)             f2pycli()
193: (8)             out, _ = capfd.readouterr()
194: (8)             assert "Cannot use distutils backend with Python>=3.12" in out
195: (0)         @pytest.mark.xfail
196: (0)     def testF2py_skip(capfd, retreal_f77, monkeypatch):
197: (4)         """Tests that functions can be skipped
198: (4)         CLI :: skip
199: (4)         """
200: (4)         fout1 = get_io_paths(retreal_f77, mname="test")
201: (4)         ipath = fout1.finp
202: (4)         toskip = "t0 t4 t8 sd s8 s4"
203: (4)         remaining = "td s0"
204: (4)         monkeypatch setattr(
205: (8)             sys, "argv",
206: (8)             f'f2py {ipath} -m test skip: {toskip}'.split())
207: (4)         with util.switchdir(ipath.parent):
208: (8)             f2pycli()
209: (8)             out, err = capfd.readouterr()
210: (8)             for skey in toskip.split():

```

```

211: (12)                     assert (
212: (16)                         f'buildmodule: Could not found the body of interfaced routine
213: (16) " {skey} ". Skipping.'
214: (8)                         in err)
215: (12)                     for rkey in remaining.split():
216: (0)                         assert f'Constructing wrapper function "{rkey}"' in out
217: (4)                     """Test that functions can be kept by only:
218: (4)                         CLI :: only:
219: (4)                         """
220: (4)                     foutl = get_io_paths(retreal_f77, mname="test")
221: (4)                     ipath = foutl.finp
222: (4)                     toskip = "t0 t4 t8 sd s8 s4"
223: (4)                     tokeep = "td s0"
224: (4)                     monkeypatch setattr(
225: (8)                         sys, "argv",
226: (8)                         f'f2py {ipath} -m test only: {tokeep}'.split())
227: (4)                     with util.switchdir(ipath.parent):
228: (8)                         f2pycli()
229: (8)                     out, err = capfd.readouterr()
230: (8)                     for skey in toskip.split():
231: (12)                         assert (
232: (16)                             f'buildmodule: Could not find the body of interfaced routine " {skey} ". Skipping.'
233: (16)                         in err)
234: (8)                     for rkey in tokeep.split():
235: (12)                         assert f'Constructing wrapper function "{rkey}"' in out
236: (0)                     def test_file_processing_switch(capfd, hello_world_f90, retreal_f77,
237: (32)                                     monkeypatch):
238: (4)                         """Tests that it is possible to return to file processing mode
239: (4)                         CLI :: :
240: (4)                         BUG: numpy-gh #20520
241: (4)                         """
242: (4)                     foutl = get_io_paths(retreal_f77, mname="test")
243: (4)                     ipath = foutl.finp
244: (4)                     toskip = "t0 t4 t8 sd s8 s4"
245: (4)                     ipath2 = Path(hello_world_f90)
246: (4)                     tokeep = "td s0 hi" # hi is in ipath2
247: (4)                     mname = "blah"
248: (4)                     monkeypatch setattr(
249: (8)                         sys,
250: (8)                         "argv",
251: (8)                         f'f2py {ipath} -m {mname} only: {tokeep} : {ipath2}'.split(
252: (8)                             ),
253: (4)                         )
254: (4)                     with util.switchdir(ipath.parent):
255: (8)                         f2pycli()
256: (8)                     out, err = capfd.readouterr()
257: (8)                     for skey in toskip.split():
258: (12)                         assert (
259: (16)                             f'buildmodule: Could not find the body of interfaced routine " {skey} ". Skipping.'
260: (16)                         in err)
261: (8)                     for rkey in tokeep.split():
262: (12)                         assert f'Constructing wrapper function "{rkey}"' in out
263: (0)                     def test_mod_gen_f77(capfd, hello_world_f90, monkeypatch):
264: (4)                         """Checks the generation of files based on a module name
265: (4)                         CLI :: -m
266: (4)                         """
267: (4)                         MNAME = "hi"
268: (4)                         foutl = get_io_paths(hello_world_f90, mname=MNAME)
269: (4)                         ipath = foutl.f90inp
270: (4)                         monkeypatch setattr(sys, "argv", f'f2py {ipath} -m {MNAME}'.split())
271: (4)                         with util.switchdir(ipath.parent):
272: (8)                             f2pycli()
273: (4)                             assert Path.exists(foutl.cmodf)
274: (4)                             assert Path.exists(foutl.wrap77)
275: (0)                     def test_mod_gen_gh25263(capfd, hello_world_f77, monkeypatch):
276: (4)                         """Check that pyf files are correctly generated with module structure

```

```

277: (4)           CLI :: -m <name> -h pyf_file
278: (4)           BUG: numpy-gh #20520
279: (4)           """
280: (4)           MNAME = "hi"
281: (4)           foutl = get_io_paths(hello_world_f77, mname=MNAME)
282: (4)           ipath = foutl.finp
283: (4)           monkeypatch setattr(sys, "argv", f'f2py {ipath} -m {MNAME} -h
hi.pyf'.split())
284: (4)           with util.switchdir(ipath.parent):
285: (8)             f2pycli()
286: (8)             with Path('hi.pyf').open() as hipyf:
287: (12)               pyfdat = hipyf.read()
288: (12)               assert "python module hi" in pyfdat
289: (0)           def test_lower_cmod(capfd, hello_world_f77, monkeypatch):
290: (4)             """Lowers cases by flag or when -h is present
291: (4)             CLI :: --[no-]lower
292: (4)             """
293: (4)             foutl = get_io_paths(hello_world_f77, mname="test")
294: (4)             ipath = foutl.finp
295: (4)             capshi = re.compile(r"HI\(\)")
296: (4)             capslo = re.compile(r"hi\(\)")
297: (4)             monkeypatch setattr(sys, "argv",
298: (4)               f'f2py {ipath} -m test --lower'.split())
299: (8)             with util.switchdir(ipath.parent):
300: (8)               f2pycli()
301: (8)               out, _ = capfd.readouterr()
302: (8)               assert capslo.search(out) is not None
303: (8)               assert capshi.search(out) is None
304: (24)             monkeypatch setattr(sys, "argv",
305: (4)               f'f2py {ipath} -m test --no-lower'.split())
306: (4)             with util.switchdir(ipath.parent):
307: (8)               f2pycli()
308: (8)               out, _ = capfd.readouterr()
309: (8)               assert capslo.search(out) is None
309: (8)               assert capshi.search(out) is not None
310: (0)           def test_lower_sig(capfd, hello_world_f77, monkeypatch):
311: (4)             """Lowers cases in signature files by flag or when -h is present
312: (4)             CLI :: --[no-]lower -h
313: (4)             """
314: (4)             foutl = get_io_paths(hello_world_f77, mname="test")
315: (4)             ipath = foutl.finp
316: (4)             capshi = re.compile(r"Block: HI")
317: (4)             capslo = re.compile(r"Block: hi")
318: (4)             monkeypatch setattr(
319: (8)               sys,
320: (8)               "argv",
321: (8)               f'f2py {ipath} -h {foutl.pyf} -m test --overwrite-signature'.split(),
322: (4)             )
323: (4)             with util.switchdir(ipath.parent):
324: (8)               f2pycli()
325: (8)               out, _ = capfd.readouterr()
326: (8)               assert capslo.search(out) is not None
327: (8)               assert capshi.search(out) is None
328: (4)             monkeypatch setattr(
329: (8)               sys,
330: (8)               "argv",
331: (8)               f'f2py {ipath} -h {foutl.pyf} -m test --overwrite-signature --no-
lower'
332: (8)               .split(),
333: (4)             )
334: (4)             with util.switchdir(ipath.parent):
335: (8)               f2pycli()
336: (8)               out, _ = capfd.readouterr()
337: (8)               assert capslo.search(out) is None
338: (8)               assert capshi.search(out) is not None
339: (0)           def test_build_dir(capfd, hello_world_f90, monkeypatch):
340: (4)             """Ensures that the build directory can be specified
341: (4)             CLI :: --build-dir
342: (4)             """
343: (4)             ipath = Path(hello_world_f90)

```

```

344: (4) mname = "blah"
345: (4) odir = "tttmp"
346: (4) monkeypatch setattr(sys, "argv",
347: (24) f'f2py -m {mname} {ipath} --build-dir {odir}'.split())
348: (4) with util.switchdir(ipath.parent):
349: (8)     f2pycli()
350: (8)     out, _ = capfd.readouterr()
351: (8)     assert f"Wrote C/API module \\"{mname}\\\" in out"
352: (0) def test_overwrite(capfd, hello_world_f90, monkeypatch):
353: (4)     """Ensures that the build directory can be specified
354: (4)     CLI :: --overwrite-signature
355: (4)
356: (4)     ipath = Path(hello_world_f90)
357: (4)     monkeypatch setattr(
358: (8)         sys, "argv",
359: (8)         f'f2py -h faker.pyf {ipath} --overwrite-signature'.split())
360: (4)     with util.switchdir(ipath.parent):
361: (8)         Path("faker.pyf").write_text("Fake news", encoding="ascii")
362: (8)         f2pycli()
363: (8)         out, _ = capfd.readouterr()
364: (8)         assert "Saving signatures to file" in out
365: (0) def test_latexdoc(capfd, hello_world_f90, monkeypatch):
366: (4)     """Ensures that TeX documentation is written out
367: (4)     CLI :: --latex-doc
368: (4)
369: (4)     ipath = Path(hello_world_f90)
370: (4)     mname = "blah"
371: (4)     monkeypatch setattr(sys, "argv",
372: (24)         f'f2py -m {mname} {ipath} --latex-doc'.split())
373: (4)     with util.switchdir(ipath.parent):
374: (8)         f2pycli()
375: (8)         out, _ = capfd.readouterr()
376: (8)         assert "Documentation is saved to file" in out
377: (8)         with Path(f"{mname}module.tex").open() as otex:
378: (12)             assert "\\documentclass" in otex.read()
379: (0) def test_nolatexdoc(capfd, hello_world_f90, monkeypatch):
380: (4)     """Ensures that TeX documentation is written out
381: (4)     CLI :: --no-latex-doc
382: (4)
383: (4)     ipath = Path(hello_world_f90)
384: (4)     mname = "blah"
385: (4)     monkeypatch setattr(sys, "argv",
386: (24)         f'f2py -m {mname} {ipath} --no-latex-doc'.split())
387: (4)     with util.switchdir(ipath.parent):
388: (8)         f2pycli()
389: (8)         out, _ = capfd.readouterr()
390: (8)         assert "Documentation is saved to file" not in out
391: (0) def test_shortlatex(capfd, hello_world_f90, monkeypatch):
392: (4)     """Ensures that truncated documentation is written out
393: (4)     TODO: Test to ensure this has no effect without --latex-doc
394: (4)     CLI :: --latex-doc --short-latex
395: (4)
396: (4)     ipath = Path(hello_world_f90)
397: (4)     mname = "blah"
398: (4)     monkeypatch setattr(
399: (8)         sys,
400: (8)         "argv",
401: (8)         f'f2py -m {mname} {ipath} --latex-doc --short-latex'.split(),
402: (4)     )
403: (4)     with util.switchdir(ipath.parent):
404: (8)         f2pycli()
405: (8)         out, _ = capfd.readouterr()
406: (8)         assert "Documentation is saved to file" in out
407: (8)         with Path(f"./{mname}module.tex").open() as otex:
408: (12)             assert "\\documentclass" not in otex.read()
409: (0) def test_restdoc(capfd, hello_world_f90, monkeypatch):
410: (4)     """Ensures that RST documentation is written out
411: (4)     CLI :: --rest-doc
412: (4)

```

```

413: (4)             ipath = Path(hello_world_f90)
414: (4)             mname = "blah"
415: (4)             monkeypatch setattr(sys, "argv",
416: (24)                 f'f2py -m {mname} {ipath} --rest-doc'.split())
417: (4)             with util.switchdir(ipath.parent):
418: (8)                 f2pycli()
419: (8)                 out, _ = capfd.readouterr()
420: (8)                 assert "ReST Documentation is saved to file" in out
421: (8)                 with Path(f"./{mname}module.rest").open() as orst:
422: (12)                   assert r".. -*- rest -*-" in orst.read()
423: (0)             def test_norestexdoc(capfd, hello_world_f90, monkeypatch):
424: (4)                 """Ensures that TeX documentation is written out
425: (4)                 CLI :: --no-rest-doc
426: (4)                 """
427: (4)                 ipath = Path(hello_world_f90)
428: (4)                 mname = "blah"
429: (4)                 monkeypatch setattr(sys, "argv",
430: (24)                     f'f2py -m {mname} {ipath} --no-rest-doc'.split())
431: (4)                 with util.switchdir(ipath.parent):
432: (8)                     f2pycli()
433: (8)                     out, _ = capfd.readouterr()
434: (8)                     assert "ReST Documentation is saved to file" not in out
435: (0)             def test_debugcapi(capfd, hello_world_f90, monkeypatch):
436: (4)                 """Ensures that debugging wrappers are written
437: (4)                 CLI :: --debug-capi
438: (4)                 """
439: (4)                 ipath = Path(hello_world_f90)
440: (4)                 mname = "blah"
441: (4)                 monkeypatch setattr(sys, "argv",
442: (24)                     f'f2py -m {mname} {ipath} --debug-capi'.split())
443: (4)                 with util.switchdir(ipath.parent):
444: (8)                     f2pycli()
445: (8)                     with Path(f"./{mname}module.c").open() as ocmod:
446: (12)                       assert r"#define DEBUGCFUNCS" in ocmod.read()
447: (0)             @pytest.mark.xfail(reason="Consistently fails on CI.")
448: (0)             def test_debugcapi_bld(hello_world_f90, monkeypatch):
449: (4)                 """Ensures that debugging wrappers work
450: (4)                 CLI :: --debug-capi -c
451: (4)                 """
452: (4)                 ipath = Path(hello_world_f90)
453: (4)                 mname = "blah"
454: (4)                 monkeypatch setattr(sys, "argv",
455: (24)                     f'f2py -m {mname} {ipath} -c --debug-capi'.split())
456: (4)                 with util.switchdir(ipath.parent):
457: (8)                     f2pycli()
458: (8)                     cmd_run = shlex.split("python3 -c \"import blah; blah.hi()\"")
459: (8)                     rout = subprocess.run(cmd_run, capture_output=True, encoding='UTF-8')
460: (8)                     eout = 'Hello World\n'
461: (8)                     eerr = textwrap.dedent("""
462: (0)             debug-capi:Python C/API function blah.hi()
463: (0)             debug-capi:float hi=:output,hidden,scalar
464: (0)             debug-capi:hi=0
465: (0)             debug-capi:Fortran subroutine `f2pywraphi(&hi)'
466: (0)             debug-capi:hi=0
467: (0)             debug-capi:Building return value.
468: (0)             debug-capi:Python C/API function blah.hi: successful.
469: (0)             debug-capi:Freeing memory.
470: (8)             """
471: (8)             assert rout.stdout == eout
472: (8)             assert rout.stderr == eerr
473: (0)             def test_wrapfunc_def(capfd, hello_world_f90, monkeypatch):
474: (4)                 """Ensures that fortran subroutine wrappers for F77 are included by
475: (4)                 default
476: (4)                 CLI :: --[no]-wrap-functions
477: (4)                 """
478: (4)                 ipath = Path(hello_world_f90)
479: (4)                 mname = "blah"
480: (4)                 monkeypatch setattr(sys, "argv", f'f2py -m {mname} {ipath}'.split())
481: (4)                 with util.switchdir(ipath.parent):

```

```

481: (8)           f2pycli()
482: (4)           out, _ = capfd.readouterr()
483: (4)           assert r"Fortran 77 wrappers are saved to" in out
484: (4)           monkeypatch setattr(sys, "argv",
485: (24)             f'f2py -m {mname} {ipath} --wrap-functions'.split())
486: (4)           with util.switchdir(ipath.parent):
487: (8)             f2pycli()
488: (8)             out, _ = capfd.readouterr()
489: (8)             assert r"Fortran 77 wrappers are saved to" in out
490: (0)           def test_nowrapfunc(capfd, hello_world_f90, monkeypatch):
491: (4)             """Ensures that fortran subroutine wrappers for F77 can be disabled
492: (4)             CLI :: --no-wrap-functions
493: (4)
494: (4)             ipath = Path(hello_world_f90)
495: (4)             mname = "blah"
496: (4)             monkeypatch setattr(sys, "argv",
497: (24)               f'f2py -m {mname} {ipath} --no-wrap-
functions'.split())
498: (4)           with util.switchdir(ipath.parent):
499: (8)             f2pycli()
500: (8)             out, _ = capfd.readouterr()
501: (8)             assert r"Fortran 77 wrappers are saved to" not in out
502: (0)           def test_inclheader(capfd, hello_world_f90, monkeypatch):
503: (4)             """Add to the include directories
504: (4)             CLI :: -include
505: (4)             TODO: Document this in the help string
506: (4)
507: (4)             ipath = Path(hello_world_f90)
508: (4)             mname = "blah"
509: (4)             monkeypatch setattr(
510: (8)               sys,
511: (8)               "argv",
512: (8)               f'f2py -m {mname} {ipath} -include<stdbool.h> -include<stdio.h> '.
513: (8)               split(),
514: (4)             )
515: (4)           with util.switchdir(ipath.parent):
516: (8)             f2pycli()
517: (8)             with Path(f"./{mname}module.c").open() as ocmod:
518: (12)               ocmr = ocmod.read()
519: (12)               assert "#include <stdbool.h>" in ocmr
520: (12)               assert "#include <stdio.h>" in ocmr
521: (0)           def test_inclpath():
522: (4)             """Add to the include directories
523: (4)             CLI :: --include-paths
524: (4)
525: (4)             pass
526: (0)           def test_hlink():
527: (4)             """Add to the include directories
528: (4)             CLI :: --help-link
529: (4)
530: (4)             pass
531: (0)           def test_f2cmap(capfd, f2cmap_f90, monkeypatch):
532: (4)             """Check that Fortran-to-Python KIND specs can be passed
533: (4)             CLI :: --f2cmap
534: (4)
535: (4)             ipath = Path(f2cmap_f90)
536: (4)             monkeypatch setattr(sys, "argv", f'f2py -m blah {ipath} --f2cmap
mapfile'.split())
537: (4)           with util.switchdir(ipath.parent):
538: (8)             f2pycli()
539: (8)             out, _ = capfd.readouterr()
540: (8)             assert "Reading f2cmap from 'mapfile' ..." in out
541: (8)             assert "Mapping \"real(kind=real32)\" to \"float\"" in out
542: (8)             assert "Mapping \"real(kind=real64)\" to \"double\"" in out
543: (8)             assert "Mapping \"integer(kind=int64)\" to \"long_long\"" in out
544: (8)             assert "Successfully applied user defined f2cmap changes" in out
545: (0)           def test_quiet(capfd, hello_world_f90, monkeypatch):
546: (4)             """Reduce verbosity
547: (4)             CLI :: --quiet

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

548: (4)
549: (4)
550: (4)
551: (4)
552: (8)
553: (8)
554: (8)
555: (0)
556: (4)
557: (4)
558: (4)
559: (4)
560: (4)
verbose'.split())
561: (4)
562: (8)
563: (8)
564: (8)
565: (0)
566: (4)
567: (4)
568: (4)
569: (4)
570: (4)
571: (8)
572: (8)
573: (8)
574: (8)
575: (0)
576: (0)
577: (4)
578: (4)
579: (4)
580: (4)
581: (4)
582: (4)
583: (8)
584: (8)
585: (8)
586: (8)
587: (8)
588: (0)
589: (4)
590: (4)
591: (4)
592: (4)
593: (0)
594: (4)
595: (4)
596: (4)
597: (4)
598: (0)
599: (4)
600: (4)
601: (4)
602: (4)
603: (0)
604: (4)
605: (4)
606: (4)
607: (4)
608: (0)
609: (4)
610: (4)
611: (4)
612: (4)
613: (0)
614: (4)
615: (4)

        """
        ipath = Path(hello_world_f90)
        monkeypatch setattr(sys, "argv", f'f2py -m blah {ipath} --quiet'.split())
        with util.switchdir(ipath.parent):
            f2pycli()
            out, _ = capfd.readouterr()
            assert len(out) == 0
    def test_verbose(capfd, hello_world_f90, monkeypatch):
        """Increase verbosity
        CLI :: --verbose
        """
        ipath = Path(hello_world_f90)
        monkeypatch setattr(sys, "argv", f'f2py -m blah {ipath} --
verbose'.split())
        with util.switchdir(ipath.parent):
            f2pycli()
            out, _ = capfd.readouterr()
            assert "analyzeline" in out
    def test_version(capfd, monkeypatch):
        """Ensure version
        CLI :: -v
        """
        monkeypatch setattr(sys, "argv", 'f2py -v'.split())
        with pytest.raises(SystemExit):
            f2pycli()
            out, _ = capfd.readouterr()
            import numpy as np
            assert np.__version__ == out.strip()
@ pytest.mark.xfail(reason="Consistently fails on CI.")
def test_npdistop(hello_world_f90, monkeypatch):
    """
    CLI :: -c
    """
    ipath = Path(hello_world_f90)
    monkeypatch setattr(sys, "argv", f'f2py -m blah {ipath} -c'.split())
    with util.switchdir(ipath.parent):
        f2pycli()
        cmd_run = shlex.split("python -c \"import blah; blah.hi()\"")
        rout = subprocess.run(cmd_run, capture_output=True, encoding='UTF-8')
        eout = ' Hello World\n'
        assert rout.stdout == eout
    def test_npd_fcompiler():
        """
        CLI :: -c --fcompiler
        """
        pass
    def test_npd_compiler():
        """
        CLI :: -c --compiler
        """
        pass
    def test_npd_help_fcompiler():
        """
        CLI :: -c --help-fcompiler
        """
        pass
    def test_npd_f77exec():
        """
        CLI :: -c --f77exec
        """
        pass
    def test_npd_f90exec():
        """
        CLI :: -c --f90exec
        """
        pass
    def test_npd_f77flags():
        """
        CLI :: -c --f77flags
        """

```

```

616: (4)          """
617: (4)          pass
618: (0)          def test_npd_f90flags():
619: (4)          """
620: (4)          CLI :: -c --f90flags
621: (4)          """
622: (4)          pass
623: (0)          def test_npd_opt():
624: (4)          """
625: (4)          CLI :: -c --opt
626: (4)          """
627: (4)          pass
628: (0)          def test_npd_arch():
629: (4)          """
630: (4)          CLI :: -c --arch
631: (4)          """
632: (4)          pass
633: (0)          def test_npd_noopt():
634: (4)          """
635: (4)          CLI :: -c --noopt
636: (4)          """
637: (4)          pass
638: (0)          def test_npd_noarch():
639: (4)          """
640: (4)          CLI :: -c --noarch
641: (4)          """
642: (4)          pass
643: (0)          def test_npd_debug():
644: (4)          """
645: (4)          CLI :: -c --debug
646: (4)          """
647: (4)          pass
648: (0)          def test_npd_link_auto():
649: (4)          """
650: (4)          CLI :: -c --link-<resource>
651: (4)          """
652: (4)          pass
653: (0)          def test_npd_lib():
654: (4)          """
655: (4)          CLI :: -c -L/path/to/lib/ -l<libname>
656: (4)          """
657: (4)          pass
658: (0)          def test_npd_define():
659: (4)          """
660: (4)          CLI :: -D<define>
661: (4)          """
662: (4)          pass
663: (0)          def test_npd_undefine():
664: (4)          """
665: (4)          CLI :: -U<name>
666: (4)          """
667: (4)          pass
668: (0)          def test_npd_incl():
669: (4)          """
670: (4)          CLI :: -I/path/to/include/
671: (4)          """
672: (4)          pass
673: (0)          def test_npd_linker():
674: (4)          """
675: (4)          CLI :: <filename>.o <filename>.so <filename>.a
676: (4)          """
677: (4)          pass

```

---

File 174 - test\_isoc.py:

```

1: (0)          from . import util
2: (0)          import numpy as np

```

```

3: (0)         import pytest
4: (0)         from numpy.testing import assert_allclose
5: (0)         class TestISOC(util.F2PyTest):
6: (4)             sources = [
7: (8)                 util.getpath("tests", "src", "isocINTRIN", "isoCtests.f90"),
8: (4)             ]
9: (4)             def test_c_double(self):
10: (8)                 out = self.module.coddity.c_add(1, 2)
11: (8)                 exp_out = 3
12: (8)                 assert out == exp_out
13: (4)             def test_bindc_function(self):
14: (8)                 out = self.module.coddity.wat(1, 20)
15: (8)                 exp_out = 8
16: (8)                 assert out == exp_out
17: (4)             def test_bindc_kinds(self):
18: (8)                 out = self.module.coddity.c_add_int64(1, 20)
19: (8)                 exp_out = 21
20: (8)                 assert out == exp_out
21: (4)             def test_bindc_add_arr(self):
22: (8)                 a = np.array([1,2,3])
23: (8)                 b = np.array([1,2,3])
24: (8)                 out = self.module.coddity.add_arr(a, b)
25: (8)                 exp_out = a*2
26: (8)                 assert_allclose(out, exp_out)
27: (0)             def test_process_f2cmap_dict():
28: (4)                 from numpy.f2py.auxfuncs import process_f2cmap_dict
29: (4)                 f2cmap_all = {"integer": {"8": "rubbish_type"}}
30: (4)                 new_map = {"INTEGER": {"4": "int"}}
31: (4)                 c2py_map = {"int": "int", "rubbish_type": "long"}
32: (4)                 exp_map, exp_maptyp = ({"integer": {"8": "rubbish_type"}, "4": "int"}}, 
["int"])
33: (4)                 res_map, res_maptyp = process_f2cmap_dict(f2cmap_all, new_map, c2py_map)
34: (4)                 assert res_map == exp_map
35: (4)                 assert res_maptyp == exp_maptyp
-----
```

## File 175 - test\_kind.py:

```

1: (0)         import os
2: (0)         import pytest
3: (0)         import platform
4: (0)         from numpy.f2py.crackfortran import (
5: (4)             _selected_int_kind_func as selected_int_kind,
6: (4)             _selected_real_kind_func as selected_real_kind,
7: (0)         )
8: (0)         from . import util
9: (0)         class TestKind(util.F2PyTest):
10: (4)             sources = [util.getpath("tests", "src", "kind", "foo.f90")]
11: (4)             def test_int(self):
12: (8)                 """Test `int` kind_func for integers up to 10**40."""
13: (8)                 selectedintkind = self.module.selectedintkind
14: (8)                 for i in range(40):
15: (12)                     assert selectedintkind(i) == selected_int_kind(
16: (16)                         i
17: (12)                     ), f"selectedintkind({i}): expected {selected_int_kind(i)!r} but
got {selectedintkind(i)!r}"
18: (4)             def test_real(self):
19: (8)                 """
20: (8)                     Test (processor-dependent) `real` kind_func for real numbers
21: (8)                     of up to 31 digits precision (extended/quadruple).
22: (8)                 """
23: (8)                 selectedrealkind = self.module.selectedrealkind
24: (8)                 for i in range(32):
25: (12)                     assert selectedrealkind(i) == selected_real_kind(
26: (16)                         i
27: (12)                     ), f"selectedrealkind({i}): expected {selected_real_kind(i)!r} but
got {selectedrealkind(i)!r}"
28: (4)                     @pytest.mark.xfail(platform.machine().lower().startswith("ppc"),
```

```

29: (23)                     reason="Some PowerPC may not support full IEEE 754
precision")
30: (4)             def test_quad_precision(self):
31: (8)                 """
32: (8)                     Test kind_func for quadruple precision [`real(16)`] of 32+ digits .
33: (8)                 """
34: (8)                     selectedrealkind = self.module.selectedrealkind
35: (8)                     for i in range(32, 40):
36: (12)                         assert selectedrealkind(i) == selected_real_kind(
37: (16)                             i
38: (12)                         ), f"selectedrealkind({i}): expected {selected_real_kind(i)!r} but
got {selectedrealkind(i)!r}"
-----
```

## File 176 - test\_mixed.py:

```

1: (0)             import os
2: (0)             import textwrap
3: (0)             import pytest
4: (0)             from numpy.testing import IS_PYPY
5: (0)             from . import util
6: (0)             class TestMixed(util.F2PyTest):
7: (4)                 sources = [
8: (8)                     util.getpath("tests", "src", "mixed", "foo.f"),
9: (8)                     util.getpath("tests", "src", "mixed", "foo_fixed.f90"),
10: (8)                    util.getpath("tests", "src", "mixed", "foo_free.f90"),
11: (4)
12: (4)                 def test_all(self):
13: (8)                     assert self.module.bar11() == 11
14: (8)                     assert self.module.foo_fixed.bar12() == 12
15: (8)                     assert self.module.foo_free.bar13() == 13
16: (4)                     @pytest.mark.xfail(IS_PYPY,
17: (23)                         reason="PyPy cannot modify tp_doc after PyType_Ready")
18: (4)                 def test_docstring(self):
19: (8)                     expected = textwrap.dedent("""\
20: (8)                         a = bar11()
21: (8)                         Wrapper for ``bar11``.
22: (8)                         Returns
23: (8)                         -----
24: (8)                         a : int
25: (8)                         """)
26: (8)                     assert self.module.bar11.__doc__ == expected
-----
```

## File 177 - test\_module\_doc.py:

```

1: (0)             import os
2: (0)             import sys
3: (0)             import pytest
4: (0)             import textwrap
5: (0)             from . import util
6: (0)             from numpy.testing import IS_PYPY
7: (0)             class TestModuleDocString(util.F2PyTest):
8: (4)                 sources = [
9: (8)                     util.getpath("tests", "src", "module_data",
10: (21)                         "module_data_docstring.f90")
11: (4)
12: (4)                     @pytest.mark.skipif(sys.platform == "win32",
13: (24)                         reason="Fails with MinGW64 Gfortran (Issue #9673)")
14: (4)                     @pytest.mark.xfail(IS_PYPY,
15: (23)                         reason="PyPy cannot modify tp_doc after PyType_Ready")
16: (4)                 def test_module_docstring(self):
17: (8)                     assert self.module.mod.__doc__ == textwrap.dedent("""\
18: (21)                         i : 'i'-scalar
19: (21)                         x : 'i'-array(4)
20: (21)                         a : 'f'-array(2,3)
21: (21)                         b : 'f'-array(-1,-1), not allocated\x00
-----
```

```
22: (21)          foo()\n23: (21)          Wrapper for ``foo``.\n\n""")
```

---

File 178 - test\_parameter.py:

```
1: (0)          import os\n2: (0)          import pytest\n3: (0)          import numpy as np\n4: (0)          from . import util\n5: (0)          class TestParameters(util.F2PyTest):\n6: (4)              sources = [\n7: (8)                  util.getpath("tests", "src", "parameter", "constant_real.f90"),\n8: (8)                  util.getpath("tests", "src", "parameter", "constant_integer.f90"),\n9: (8)                  util.getpath("tests", "src", "parameter", "constant_both.f90"),\n10: (8)                 util.getpath("tests", "src", "parameter", "constant_compound.f90"),\n11: (8)                 util.getpath("tests", "src", "parameter",\n"constant_non_compound.f90"),\n12: (4)             ]\n13: (4)             @pytest.mark.slow\n14: (4)             def test_constant_real_single(self):\n15: (8)                 x = np.arange(6, dtype=np.float32)[::2]\n16: (8)                 pytest.raises(ValueError, self.module.foo_single, x)\n17: (8)                 x = np.arange(3, dtype=np.float32)\n18: (8)                 self.module.foo_single(x)\n19: (8)                 assert np.allclose(x, [0 + 1 + 2 * 3, 1, 2])\n20: (4)             @pytest.mark.slow\n21: (4)             def test_constant_real_double(self):\n22: (8)                 x = np.arange(6, dtype=np.float64)[::2]\n23: (8)                 pytest.raises(ValueError, self.module.foo_double, x)\n24: (8)                 x = np.arange(3, dtype=np.float64)\n25: (8)                 self.module.foo_double(x)\n26: (8)                 assert np.allclose(x, [0 + 1 + 2 * 3, 1, 2])\n27: (4)             @pytest.mark.slow\n28: (4)             def test_constant_compound_int(self):\n29: (8)                 x = np.arange(6, dtype=np.int32)[::2]\n30: (8)                 pytest.raises(ValueError, self.module.foo_compound_int, x)\n31: (8)                 x = np.arange(3, dtype=np.int32)\n32: (8)                 self.module.foo_compound_int(x)\n33: (8)                 assert np.allclose(x, [0 + 1 + 2 * 6, 1, 2])\n34: (4)             @pytest.mark.slow\n35: (4)             def test_constant_non_compound_int(self):\n36: (8)                 x = np.arange(4, dtype=np.int32)\n37: (8)                 self.module.foo_non_compound_int(x)\n38: (8)                 assert np.allclose(x, [0 + 1 + 2 + 3 * 4, 1, 2, 3])\n39: (4)             @pytest.mark.slow\n40: (4)             def test_constant_integer_int(self):\n41: (8)                 x = np.arange(6, dtype=np.int32)[::2]\n42: (8)                 pytest.raises(ValueError, self.module.foo_int, x)\n43: (8)                 x = np.arange(3, dtype=np.int32)\n44: (8)                 self.module.foo_int(x)\n45: (8)                 assert np.allclose(x, [0 + 1 + 2 * 3, 1, 2])\n46: (4)             @pytest.mark.slow\n47: (4)             def test_constant_integer_long(self):\n48: (8)                 x = np.arange(6, dtype=np.int64)[::2]\n49: (8)                 pytest.raises(ValueError, self.module.foo_long, x)\n50: (8)                 x = np.arange(3, dtype=np.int64)\n51: (8)                 self.module.foo_long(x)\n52: (8)                 assert np.allclose(x, [0 + 1 + 2 * 3, 1, 2])\n53: (4)             @pytest.mark.slow\n54: (4)             def test_constant_both(self):\n55: (8)                 x = np.arange(6, dtype=np.float64)[::2]\n56: (8)                 pytest.raises(ValueError, self.module.foo, x)\n57: (8)                 x = np.arange(3, dtype=np.float64)\n58: (8)                 self.module.foo(x)\n59: (8)                 assert np.allclose(x, [0 + 1 * 3 * 3 + 2 * 3 * 3, 1 * 3, 2 * 3])\n60: (4)             @pytest.mark.slow\n61: (4)             def test_constant_no(self):
```

```

62: (8)          x = np.arange(6, dtype=np.float64)[::2]
63: (8)          pytest.raises(ValueError, self.module.foo_no, x)
64: (8)          x = np.arange(3, dtype=np.float64)
65: (8)          self.module.foo_no(x)
66: (8)          assert np.allclose(x, [0 + 1 * 3 * 3 + 2 * 3 * 3, 1 * 3, 2 * 3])
67: (4)          @pytest.mark.slow
68: (4)          def test_constant_sum(self):
69: (8)              x = np.arange(6, dtype=np.float64)[::2]
70: (8)              pytest.raises(ValueError, self.module.foo_sum, x)
71: (8)              x = np.arange(3, dtype=np.float64)
72: (8)              self.module.foo_sum(x)
73: (8)              assert np.allclose(x, [0 + 1 * 3 * 3 + 2 * 3 * 3, 1 * 3, 2 * 3])
-----
```

## File 179 - test\_pyf\_src.py:

```

1: (0)          from numpy.f2py._src_pyf import process_str
2: (0)          from numpy.testing import assert_equal
3: (0)          pyf_src = """
4: (0)          python module foo
5: (4)          <_rd=real,double precision>
6: (4)          interface
7: (8)              subroutine <s,d>foosub(tol)
8: (12)                  <_rd>, intent(in,out) :: tol
9: (8)              end subroutine <s,d>foosub
10: (4)          end interface
11: (0)          end python module foo
12: (0)          """
13: (0)          expected_pyf = """
14: (0)          python module foo
15: (4)          interface
16: (8)              subroutine sfoosub(tol)
17: (12)                  real, intent(in,out) :: tol
18: (8)              end subroutine sfoosub
19: (8)              subroutine dfoosub(tol)
20: (12)                  double precision, intent(in,out) :: tol
21: (8)              end subroutine dfoosub
22: (4)          end interface
23: (0)          end python module foo
24: (0)          """
25: (0)          def normalize_whitespace(s):
26: (4)          """
27: (4)              Remove leading and trailing whitespace, and convert internal
28: (4)              stretches of whitespace to a single space.
29: (4)          """
30: (4)          return ' '.join(s.split())
31: (0)          def test_from_template():
32: (4)              """Regression test for gh-10712."""
33: (4)              pyf = process_str(pyf_src)
34: (4)              normalized_pyf = normalize_whitespace(pyf)
35: (4)              normalized_expected_pyf = normalize_whitespace(expected_pyf)
36: (4)              assert_equal(normalized_pyf, normalized_expected_pyf)
-----
```

## File 180 - test\_regression.py:

```

1: (0)          import os
2: (0)          import pytest
3: (0)          import numpy as np
4: (0)          from . import util
5: (0)          class TestIntentInOut(util.F2PyTest):
6: (4)              sources = [util.getpath("tests", "src", "regression", "inout.f90")]
7: (4)              @pytest.mark.slow
8: (4)              def test_inout(self):
9: (8)                  x = np.arange(6, dtype=np.float32)[::2]
10: (8)                  pytest.raises(ValueError, self.module.foo, x)
11: (8)                  x = np.arange(3, dtype=np.float32)
-----
```

```

12: (8)             self.module.foo(x)
13: (8)             assert np.allclose(x, [3, 1, 2])
14: (0)         class TestNegativeBounds(util.F2PyTest):
15: (4)             sources = [util.getpath("tests", "src", "negative_bounds",
16: (4) "issue_20853.f90")]
17: (4)             @pytest.mark.slow
18: (8)             def test_negbound(self):
19: (8)                 xvec = np.arange(12)
20: (8)                 xlow = -6
21: (8)                 xhigh = 4
22: (12)                 def ubound(xl, xh):
23: (8)                     return xh - xl + 1
24: (24)                 rval = self.module.foo(is_=xlow, ie_=xhigh,
25: (8)                             arr=xvec[:ubound(xlow, xhigh)])
26: (8)                 expval = np.arange(11, dtype = np.float32)
27: (8)                 assert np.allclose(rval, expval)
27: (0)         class TestNumpyVersionAttribute(util.F2PyTest):
28: (4)             sources = [util.getpath("tests", "src", "regression", "inout.f90")]
29: (4)             @pytest.mark.slow
30: (4)             def test_numpy_version_attribute(self):
31: (8)                 assert hasattr(self.module, "__f2py_numpy_version__")
32: (8)                 assert isinstance(self.module.__f2py_numpy_version__, str)
33: (8)                 assert np.__version__ == self.module.__f2py_numpy_version__
34: (0)             def test_include_path():
35: (4)                 incdir = np.f2py.get_include()
36: (4)                 fnames_in_dir = os.listdir(incdir)
37: (4)                 for fname in ("fortranobject.c", "fortranobject.h"):
38: (8)                     assert fname in fnames_in_dir
39: (0)         class TestModuleAndSubroutine(util.F2PyTest):
40: (4)             module_name = "example"
41: (4)             sources = [util.getpath("tests", "src", "regression", "gh25337",
42: (8) "data.f90"),
43: (15)                         util.getpath("tests", "src", "regression", "gh25337",
44: (4) "use_data.f90")]
45: (4)             @pytest.mark.slow
46: (8)             def test_gh25337(self):
47: (8)                 self.module.data.set_shift(3)
48: (8)                 assert "data" in dir(self.module)

```

---

**File 181 - test\_quoted\_character.py:**

```

1: (0)             """See https://github.com/numpy/numpy/pull/10676.
2: (0)             """
3: (0)             import sys
4: (0)             import pytest
5: (0)             from . import util
6: (0)             class TestQuotedCharacter(util.F2PyTest):
7: (4)                 sources = [util.getpath("tests", "src", "quoted_character", "foo.f")]
8: (4)                 @pytest.mark.skipif(sys.platform == "win32",
9: (24)                               reason="Fails with MinGW64 Gfortran (Issue #9673)")
10: (4)                 def test_quoted_character(self):
11: (8)                     assert self.module.foo() == (b'''', b'''', b";", b"!", b"(", b")")

```

---

**File 182 - test\_return\_character.py:**

```

1: (0)             import pytest
2: (0)             from numpy import array
3: (0)             from . import util
4: (0)             import platform
5: (0)             IS_S390X = platform.machine() == "s390x"
6: (0)             class TestReturnCharacter(util.F2PyTest):
7: (4)                 def check_function(self, t, tname):
8: (8)                     if tname in ["t0", "t1", "s0", "s1"]:
9: (12)                         assert t("23") == b"2"
10: (12)                        r = t("ab")

```

```

11: (12)             assert r == b"a"
12: (12)             r = t(array("ab"))
13: (12)             assert r == b"a"
14: (12)             r = t(array(77, "u1"))
15: (12)             assert r == b"M"
16: (8)              elif tname in ["ts", "ss"]:
17: (12)                  assert t(23) == b"23"
18: (12)                  assert t("123456789abcdef") == b"123456789a"
19: (8)              elif tname in ["t5", "s5"]:
20: (12)                  assert t(23) == b"23"
21: (12)                  assert t("ab") == b"ab"
22: (12)                  assert t("123456789abcdef") == b"12345"
23: (8)              else:
24: (12)                  raise NotImplementedError
25: (0)  class TestFReturnCharacter(TestReturnCharacter):
26: (4)      sources = [
27: (8)          util.getpath("tests", "src", "return_character", "foo77.f"),
28: (8)          util.getpath("tests", "src", "return_character", "foo90.f90"),
29: (4)      ]
30: (4)      @pytest.mark.xfail(IS_S390X, reason="callback returns ' '")
31: (4)      @pytest.mark.parametrize("name", "t0,t1,t5,s0,s1,s5,ss".split(","))
32: (4)      def test_all_f77(self, name):
33: (8)          self.check_function(getattr(self.module, name), name)
34: (4)      @pytest.mark.xfail(IS_S390X, reason="callback returns ' '")
35: (4)      @pytest.mark.parametrize("name", "t0,t1,t5,ts,s0,s1,s5,ss".split(","))
36: (4)      def test_all_f90(self, name):
37: (8)          self.check_function(getattr(self.module.f90_return_char, name), name)
-----
```

## File 183 - test\_return\_complex.py:

```

1: (0)      import pytest
2: (0)      from numpy import array
3: (0)      from . import util
4: (0)  class TestReturnComplex(util.F2PyTest):
5: (4)      def check_function(self, t, tname):
6: (8)          if tname in ["t0", "t8", "s0", "s8"]:
7: (12)              err = 1e-5
8: (8)          else:
9: (12)              err = 0.0
10: (8)          assert abs(t(234j) - 234.0j) <= err
11: (8)          assert abs(t(234.6) - 234.6) <= err
12: (8)          assert abs(t(234) - 234.0) <= err
13: (8)          assert abs(t(234.6 + 3j) - (234.6 + 3j)) <= err
14: (8)          assert abs(t(-234) + 234.0) <= err
15: (8)          assert abs(t([234]) - 234.0) <= err
16: (8)          assert abs(t((234, )) - 234.0) <= err
17: (8)          assert abs(t(array(234)) - 234.0) <= err
18: (8)          assert abs(t(array(23 + 4j, "F")) - (23 + 4j)) <= err
19: (8)          assert abs(t(array([234])) - 234.0) <= err
20: (8)          assert abs(t(array([[234]])) - 234.0) <= err
21: (8)          assert abs(t(array([234]).astype("b")) + 22.0) <= err
22: (8)          assert abs(t(array([234], "h")) - 234.0) <= err
23: (8)          assert abs(t(array([234], "i")) - 234.0) <= err
24: (8)          assert abs(t(array([234], "l")) - 234.0) <= err
25: (8)          assert abs(t(array([234], "q")) - 234.0) <= err
26: (8)          assert abs(t(array([234], "f")) - 234.0) <= err
27: (8)          assert abs(t(array([234], "d")) - 234.0) <= err
28: (8)          assert abs(t(array([234 + 3j], "F")) - (234 + 3j)) <= err
29: (8)          assert abs(t(array([234], "D")) - 234.0) <= err
30: (8)          pytest.raises(TypeError, t, "abc")
31: (8)          pytest.raises(IndexError, t, [])
32: (8)          pytest.raises(IndexError, t, ())
33: (8)          pytest.raises(TypeError, t, t)
34: (8)          pytest.raises(TypeError, t, {})
35: (8)
36: (12)          try:
37: (12)              r = t(10**400)
37: (12)              assert repr(r) in ["(inf+0j)", "(Infinity+0j)"]
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

38: (8)           except OverflowError:
39: (12)         pass
40: (0)       class TestFReturnComplex(TestReturnComplex):
41: (4)         sources = [
42: (8)           util.getpath("tests", "src", "return_complex", "foo77.f"),
43: (8)           util.getpath("tests", "src", "return_complex", "foo90.f90"),
44: (4)         ]
45: (4)     @pytest.mark.parametrize("name", "t0,t8,t16,td,s0,s8,s16,sd".split(","))
46: (4)     def test_all_f77(self, name):
47: (8)       self.check_function(getattr(self.module, name), name)
48: (4)     @pytest.mark.parametrize("name", "t0,t8,t16,td,s0,s8,s16,sd".split(","))
49: (4)     def test_all_f90(self, name):
50: (8)       self.check_function(getattr(self.module.f90_return_complex, name),
51: (28)             name)

```

---

## File 184 - test\_return\_integer.py:

```

1: (0)           import pytest
2: (0)           from numpy import array
3: (0)           from . import util
4: (0)       class TestReturnInteger(util.F2PyTest):
5: (4)         def check_function(self, t, tname):
6: (8)           assert t(123) == 123
7: (8)           assert t(123.6) == 123
8: (8)           assert t("123") == 123
9: (8)           assert t(-123) == -123
10: (8)          assert t([123]) == 123
11: (8)          assert t((123, )) == 123
12: (8)          assert t(array(123)) == 123
13: (8)          assert t(array(123, "b")) == 123
14: (8)          assert t(array(123, "h")) == 123
15: (8)          assert t(array(123, "i")) == 123
16: (8)          assert t(array(123, "l")) == 123
17: (8)          assert t(array(123, "B")) == 123
18: (8)          assert t(array(123, "f")) == 123
19: (8)          assert t(array(123, "d")) == 123
20: (8)          pytest.raises(ValueError, t, "abc")
21: (8)          pytest.raises(IndexError, t, [])
22: (8)          pytest.raises(IndexError, t, ())
23: (8)          pytest.raises(Exception, t, t)
24: (8)          pytest.raises(Exception, t, {})
25: (8)          if tname in ["t8", "s8"]:
26: (12)            pytest.raises(OverflowError, t, 10000000000000000000000000000000)
27: (12)            pytest.raises(OverflowError, t, 1000000001111111111111.23)
28: (0)       class TestFReturnInteger(TestReturnInteger):
29: (4)         sources = [
30: (8)           util.getpath("tests", "src", "return_integer", "foo77.f"),
31: (8)           util.getpath("tests", "src", "return_integer", "foo90.f90"),
32: (4)         ]
33: (4)     @pytest.mark.parametrize("name",
34: (29)           "t0,t1,t2,t4,t8,s0,s1,s2,s4,s8".split(","))
35: (4)     def test_all_f77(self, name):
36: (8)       self.check_function(getattr(self.module, name), name)
37: (4)     @pytest.mark.parametrize("name",
38: (29)           "t0,t1,t2,t4,t8,s0,s1,s2,s4,s8".split(","))
39: (4)     def test_all_f90(self, name):
40: (8)       self.check_function(getattr(self.module.f90_return_integer, name),
41: (28)             name)

```

---

## File 185 - test\_return\_logical.py:

```

1: (0)           import pytest
2: (0)           from numpy import array
3: (0)           from . import util
4: (0)       class TestReturnLogical(util.F2PyTest):

```

```

5: (4)             def check_function(self, t):
6: (8)                 assert t(True) == 1
7: (8)                 assert t(False) == 0
8: (8)                 assert t(0) == 0
9: (8)                 assert t(None) == 0
10: (8)                assert t(0.0) == 0
11: (8)                assert t(0j) == 0
12: (8)                assert t(1j) == 1
13: (8)                assert t(234) == 1
14: (8)                assert t(234.6) == 1
15: (8)                assert t(234.6 + 3j) == 1
16: (8)                assert t("234") == 1
17: (8)                assert t("aaa") == 1
18: (8)                assert t("") == 0
19: (8)                assert t([]) == 0
20: (8)                assert t(() == 0
21: (8)                assert t({}) == 0
22: (8)                assert t(t) == 1
23: (8)                assert t(-234) == 1
24: (8)                assert t(10**100) == 1
25: (8)                assert t([234]) == 1
26: (8)                assert t((234, )) == 1
27: (8)                assert t(array(234)) == 1
28: (8)                assert t(array([234])) == 1
29: (8)                assert t(array([[234]])) == 1
30: (8)                assert t(array([127], "b")) == 1
31: (8)                assert t(array([234], "h")) == 1
32: (8)                assert t(array([234], "i")) == 1
33: (8)                assert t(array([234], "l")) == 1
34: (8)                assert t(array([234], "f")) == 1
35: (8)                assert t(array([234], "d")) == 1
36: (8)                assert t(array([234 + 3j], "F")) == 1
37: (8)                assert t(array([234], "D")) == 1
38: (8)                assert t(array(0)) == 0
39: (8)                assert t(array([0])) == 0
40: (8)                assert t(array([[0]])) == 0
41: (8)                assert t(array([0j])) == 0
42: (8)                assert t(array([1])) == 1
43: (8)                pytest.raises(ValueError, t, array([0, 0]))
44: (0)            class TestFReturnLogical(TestReturnLogical):
45: (4)                sources = [
46: (8)                    util.getpath("tests", "src", "return_logical", "foo77.f"),
47: (8)                    util.getpath("tests", "src", "return_logical", "foo90.f90"),
48: (4)                ]
49: (4)                @pytest.mark.slow
50: (4)                @pytest.mark.parametrize("name", "t0,t1,t2,t4,s0,s1,s2,s4".split(","))
51: (4)                def test_all_f77(self, name):
52: (8)                    self.check_function(getattr(self.module, name))
53: (4)                @pytest.mark.slow
54: (4)                @pytest.mark.parametrize("name",
55: (29)                            "t0,t1,t2,t4,t8,s0,s1,s2,s4,s8".split(","))
56: (4)                def test_all_f90(self, name):
57: (8)                    self.check_function(getattr(self.module.f90_return_logical, name))

```

---

## File 186 - test\_return\_real.py:

```

1: (0)            import platform
2: (0)            import pytest
3: (0)            import numpy as np
4: (0)            from numpy import array
5: (0)            from . import util
6: (0)            class TestReturnReal(util.F2PyTest):
7: (4)                def check_function(self, t, tname):
8: (8)                    if tname in ["t0", "t4", "s0", "s4"]:
9: (12)                        err = 1e-5
10: (8)                    else:
11: (12)                        err = 0.0

```

```

12: (8)             assert abs(t(234) - 234.0) <= err
13: (8)             assert abs(t(234.6) - 234.6) <= err
14: (8)             assert abs(t("234") - 234) <= err
15: (8)             assert abs(t("234.6") - 234.6) <= err
16: (8)             assert abs(t(-234) + 234) <= err
17: (8)             assert abs(t([234]) - 234) <= err
18: (8)             assert abs(t((234, )) - 234.0) <= err
19: (8)             assert abs(t(array(234)) - 234.0) <= err
20: (8)             assert abs(t(array(234).astype("b")) + 22) <= err
21: (8)             assert abs(t(array(234, "h")) - 234.0) <= err
22: (8)             assert abs(t(array(234, "i")) - 234.0) <= err
23: (8)             assert abs(t(array(234, "l")) - 234.0) <= err
24: (8)             assert abs(t(array(234, "B")) - 234.0) <= err
25: (8)             assert abs(t(array(234, "f")) - 234.0) <= err
26: (8)             assert abs(t(array(234, "d")) - 234.0) <= err
27: (8)             if tname in ["t0", "t4", "s0", "s4"]:
28: (12)                 assert t(1e200) == t(1e300) # inf
29: (8)             pytest.raises(ValueError, t, "abc")
30: (8)             pytest.raises(IndexError, t, [])
31: (8)             pytest.raises(IndexError, t, ())
32: (8)             pytest.raises(Exception, t, t)
33: (8)             pytest.raises(Exception, t, {})
34: (8)             try:
35: (12)                 r = t(10**400)
36: (12)                 assert repr(r) in ["inf", "Infinity"]
37: (8)             except OverflowError:
38: (12)                 pass
39: (0)             @pytest.mark.skipif(
40: (4)                 platform.system() == "Darwin",
41: (4)                 reason="Prone to error when run with numpy/f2py/tests on mac os, "
42: (4)                 "but not when run in isolation",
43: (0)             )
44: (0)             @pytest.mark.skipif(
45: (4)                 np.dtype(np.intp).itemsize < 8,
46: (4)                 reason="32-bit builds are buggy"
47: (0)             )
48: (0)             class TestCReturnReal(TestReturnReal):
49: (4)                 suffix = ".pyf"
50: (4)                 module_name = "c_ext_return_real"
51: (4)                 code = """
52: (0)             python module c_ext_return_real
53: (0)             usercode \
54: (0)                 float t4(float value) { return value; }
55: (0)                 void s4(float *t4, float value) { *t4 = value; }
56: (0)                 double t8(double value) { return value; }
57: (0)                 void s8(double *t8, double value) { *t8 = value; }
58: (0)             \
59: (0)             interface
60: (2)                 function t4(value)
61: (4)                     real*4 intent(c) :: t4,value
62: (2)                 end
63: (2)                 function t8(value)
64: (4)                     real*8 intent(c) :: t8,value
65: (2)                 end
66: (2)                 subroutine s4(t4,value)
67: (4)                     intent(c) s4
68: (4)                     real*4 intent(out) :: t4
69: (4)                     real*4 intent(c) :: value
70: (2)                 end
71: (2)                 subroutine s8(t8,value)
72: (4)                     intent(c) s8
73: (4)                     real*8 intent(out) :: t8
74: (4)                     real*8 intent(c) :: value
75: (2)                 end
76: (0)             end interface
77: (0)             end python module c_ext_return_real
78: (4)             """
79: (4)             @pytest.mark.parametrize("name", "t4,t8,s4,s8".split(","))
80: (4)             def test_all(self, name):

```

```

81: (8)             self.check_function(getattr(self.module, name), name)
82: (0)
83: (4)         class TestFReturnReal(TestReturnReal):
84: (8)             sources = [
85: (8)                 util.getpath("tests", "src", "return_real", "foo77.f"),
86: (4)                 util.getpath("tests", "src", "return_real", "foo90.f90"),
87: (4)             ]
88: (4)             @pytest.mark.parametrize("name", "t0,t4,t8,td,s0,s4,s8,sd".split(","))
89: (8)             def test_all_f77(self, name):
90: (4)                 self.check_function(getattr(self.module, name), name)
91: (4)             @pytest.mark.parametrize("name", "t0,t4,t8,td,s0,s4,s8,sd".split(","))
92: (4)             def test_all_f90(self, name):
93: (8)                 self.check_function(getattr(self.module.f90_return_real, name), name)

-----

```

## File 187 - test\_semicolon\_split.py:

```

1: (0)         import platform
2: (0)         import pytest
3: (0)         import numpy as np
4: (0)         from . import util
5: (0)         @pytest.mark.skipif(
6: (4)             platform.system() == "Darwin",
7: (4)             reason="Prone to error when run with numpy/f2py/tests on mac os, "
8: (4)             "but not when run in isolation",
9: (0)         )
10: (0)         @pytest.mark.skipif(
11: (4)             np.dtype(np.intp).itemsize < 8,
12: (4)             reason="32-bit builds are buggy"
13: (0)         )
14: (0)         class TestMultiline(util.F2PyTest):
15: (4)             suffix = ".pyf"
16: (4)             module_name = "multiline"
17: (4)             code = f"""
18: (0)             python module {module_name}
19: (4)                 usercode """
20: (0)                 void foo(int* x) {{{
21: (4)                     char dummy = ';';
22: (4)                     *x = 42;
23: (0)                 }}}
24: (0)             """
25: (4)             interface
26: (8)                 subroutine foo(x)
27: (12)                     intent(c) foo
28: (12)                     integer intent(out) :: x
29: (8)                 end subroutine foo
30: (4)             end interface
31: (0)         end python module {module_name}
32: (4)             """
33: (4)             def test_multiline(self):
34: (8)                 assert self.module.foo() == 42
35: (0)             @pytest.mark.skipif(
36: (4)                 platform.system() == "Darwin",
37: (4)                 reason="Prone to error when run with numpy/f2py/tests on mac os, "
38: (4)                 "but not when run in isolation",
39: (0)             )
40: (0)             @pytest.mark.skipif(
41: (4)                 np.dtype(np.intp).itemsize < 8,
42: (4)                 reason="32-bit builds are buggy"
43: (0)             )
44: (0)             class TestCallstatement(util.F2PyTest):
45: (4)                 suffix = ".pyf"
46: (4)                 module_name = "callstatement"
47: (4)                 code = f"""
48: (0)                 python module {module_name}
49: (4)                     usercode """
50: (0)                     void foo(int* x) {{{
51: (0)                         }}}
52: (0)                     """

```

```

53: (4)           interface
54: (8)             subroutine foo(x)
55: (12)               intent(c) foo
56: (12)               integer intent(out) :: x
57: (12)               callprotoargument int*
58: (12)               callstatement {{ &
59: (16)                 ; &
60: (16)                 x = 42; &
61: (12)               }
62: (8)             end subroutine foo
63: (4)           end interface
64: (0)         end python module {module_name}
65: (4)         """
66: (4)         def test_callstatement(self):
67: (8)           assert self.module.foo() == 42

```

-----

File 188 - test\_size.py:

```

1: (0)         import os
2: (0)         import pytest
3: (0)         import numpy as np
4: (0)         from . import util
5: (0)         class TestSizeSumExample(util.F2PyTest):
6: (4)           sources = [util.getpath("tests", "src", "size", "foo.f90")]
7: (4)           @pytest.mark.slow
8: (4)           def test_all(self):
9: (8)             r = self.module.foo([[[]]])
10: (8)            assert r == [0]
11: (8)            r = self.module.foo([[1, 2]])
12: (8)            assert r == [3]
13: (8)            r = self.module.foo([[1, 2], [3, 4]])
14: (8)            assert np.allclose(r, [3, 7])
15: (8)            r = self.module.foo([[1, 2], [3, 4], [5, 6]])
16: (8)            assert np.allclose(r, [3, 7, 11])
17: (4)           @pytest.mark.slow
18: (4)           def test_transpose(self):
19: (8)             r = self.module.trans([[[]]])
20: (8)             assert np.allclose(r.T, np.array([]))
21: (8)             r = self.module.trans([[1, 2]])
22: (8)             assert np.allclose(r, [[1.], [2.]])
23: (8)             r = self.module.trans([[1, 2, 3], [4, 5, 6]])
24: (8)             assert np.allclose(r, [[1, 4], [2, 5], [3, 6]])
25: (4)           @pytest.mark.slow
26: (4)           def test_flatten(self):
27: (8)             r = self.module.flatten([[[]]])
28: (8)             assert np.allclose(r, [])
29: (8)             r = self.module.flatten([[1, 2]])
30: (8)             assert np.allclose(r, [1, 2])
31: (8)             r = self.module.flatten([[1, 2, 3], [4, 5, 6]])
32: (8)             assert np.allclose(r, [1, 2, 3, 4, 5, 6])

```

-----

File 189 - test\_string.py:

```

1: (0)         import os
2: (0)         import pytest
3: (0)         import textwrap
4: (0)         import numpy as np
5: (0)         from . import util
6: (0)         class TestString(util.F2PyTest):
7: (4)           sources = [util.getpath("tests", "src", "string", "char.f90")]
8: (4)           @pytest.mark.slow
9: (4)           def test_char(self):
10: (8)             strings = np.array(["ab", "cd", "ef"], dtype="c").T
11: (8)             inp, out = self.module.char_test.change_strings(
12: (12)               strings, strings.shape[1])

```

```

13: (8)             assert inp == pytest.approx(strings)
14: (8)             expected = strings.copy()
15: (8)             expected[1, :] = "AAA"
16: (8)             assert out == pytest.approx(expected)
17: (0)         class TestDocStringArguments(util.F2PyTest):
18: (4)             sources = [util.getpath("tests", "src", "string", "string.f")]
19: (4)             def test_example(self):
20: (8)                 a = np.array(b"123\0\0")
21: (8)                 b = np.array(b"123\0\0")
22: (8)                 c = np.array(b"123")
23: (8)                 d = np.array(b"123")
24: (8)                 self.module.foo(a, b, c, d)
25: (8)                 assert a.tobytes() == b"123\0\0"
26: (8)                 assert b.tobytes() == b"B23\0\0"
27: (8)                 assert c.tobytes() == b"123"
28: (8)                 assert d.tobytes() == b"D23"
29: (0)         class TestFixedString(util.F2PyTest):
30: (4)             sources = [util.getpath("tests", "src", "string", "fixed_string.f90")]
31: (4)             @staticmethod
32: (4)             def _sint(s, start=0, end=None):
33: (8)                 """Return the content of a string buffer as integer value.
34: (8)                 For example:
35: (10)                 _sint('1234') -> 4321
36: (10)                 _sint('123A') -> 17321
37: (8)
38: (8)
39: (12)                 """
40: (8)
41: (12)                 if isinstance(s, np.ndarray):
42: (8)                     s = s.tobytes()
43: (8)                 elif isinstance(s, str):
44: (12)                     s = s.encode()
45: (8)                 assert isinstance(s, bytes)
46: (8)                 if end is None:
47: (12)                     end = len(s)
48: (8)
49: (4)                 i = 0
50: (8)                 for j in range(start, min(end, len(s))): i += s[j] * 10**j
51: (8)                 return i
52: (4)             def _get_input(self, intent="in"):
53: (8)                 if intent in ["in"]:
54: (12)                     yield ""
55: (12)                     yield "1"
56: (12)                     yield "1234"
57: (12)                     yield "12345"
58: (12)                     yield b""
59: (12)                     yield b"\0"
60: (12)                     yield b"1"
61: (12)                     yield b"\01"
62: (8)                     yield b"1\0"
63: (8)                     yield b"1234"
64: (8)                     yield np.ndarray(()), np.bytes_, buffer=b"" # array(b'', dtype='|S0')
65: (8)                     yield np.array(b"") # array(b'', dtype='|S1')
66: (8)                     yield np.array(b"\0")
67: (8)                     yield np.array(b"1")
68: (8)                     yield np.array(b"1\0")
69: (8)                     yield np.array(b"\01")
70: (8)                     yield np.array(b"1234")
71: (8)                     yield np.array(b"123\0")
72: (8)                     yield np.array(b"12345")
73: (4)             def test_intent_in(self):
74: (8)                 for s in self._get_input():
75: (12)                     r = self.module.test_in_bytes4(s)
76: (12)                     expected = self._sint(s, end=4)
77: (12)                     assert r == expected, s
78: (4)             def test_intent_inout(self):
79: (8)                 for s in self._get_input(intent="inout"):
80: (12)                     rest = self._sint(s, start=4)
81: (12)                     r = self.module.test_inout_bytes4(s)
82: (12)                     expected = self._sint(s, end=4)
83: (12)                     assert r == expected

```

82: (12)

```
assert rest == self._sint(s, start=4)
```

-----  
File 190 - test\_symbolic.py:

```

1: (0)          import pytest
2: (0)          from numpy.f2py.symbolic import (
3: (4)            Expr,
4: (4)            Op,
5: (4)            ArithOp,
6: (4)            Language,
7: (4)            as_symbol,
8: (4)            as_number,
9: (4)            as_string,
10: (4)           as_array,
11: (4)           as_complex,
12: (4)           as_terms,
13: (4)           as_factors,
14: (4)           eliminate_quotes,
15: (4)           insert_quotes,
16: (4)           fromstring,
17: (4)           as_expr,
18: (4)           as_apply,
19: (4)           as_numer_denom,
20: (4)           as_ternary,
21: (4)           as_ref,
22: (4)           as_deref,
23: (4)           normalize,
24: (4)           as_eq,
25: (4)           as_ne,
26: (4)           as_lt,
27: (4)           as_gt,
28: (4)           as_le,
29: (4)           as_ge,
30: (0)
31: (0)          ) from . import util
32: (0)          class TestSymbolic(util.F2PyTest):
33: (4)            def test_eliminate_quotes(self):
34: (8)              def worker(s):
35: (12)                  r, d = eliminate_quotes(s)
36: (12)                  s1 = insert_quotes(r, d)
37: (12)                  assert s1 == s
38: (8)                  for kind in [ "", "mykind_"]:
39: (12)                      worker(kind + "'1234' // 'ABCD' ")
40: (12)                      worker(kind + "'1234' // ' + kind + 'ABCD' ")
41: (12)                      worker(kind + "\\"1234\\' // 'ABCD' ")
42: (12)                      worker(kind + "'1234' // ' + kind + "'ABCD' ")
43: (12)                      worker(kind + "'1\\\"2\\'AB\\'34' ")
44: (12)                      worker("a = " + kind + "'1\\\"2\\'AB\\'34' ")
45: (4)              def test_sanity(self):
46: (8)                  x = as_symbol("x")
47: (8)                  y = as_symbol("y")
48: (8)                  z = as_symbol("z")
49: (8)                  assert x.op == Op.SYMBOL
50: (8)                  assert repr(x) == "Expr(Op.SYMBOL, 'x')"
51: (8)                  assert x == x
52: (8)                  assert x != y
53: (8)                  assert hash(x) is not None
54: (8)                  n = as_number(123)
55: (8)                  m = as_number(456)
56: (8)                  assert n.op == Op.INTEGER
57: (8)                  assert repr(n) == "Expr(Op.INTEGER, (123, 4))"
58: (8)                  assert n == n
59: (8)                  assert n != m
60: (8)                  assert hash(n) is not None
61: (8)                  fn = as_number(12.3)
62: (8)                  fm = as_number(45.6)
63: (8)                  assert fn.op == Op.REAL

```

```

64: (8) assert repr(fn) == "Expr(Op.REAL, (12.3, 4))"
65: (8) assert fn == fn
66: (8) assert fn != fm
67: (8) assert hash(fn) is not None
68: (8) c = as_complex(1, 2)
69: (8) c2 = as_complex(3, 4)
70: (8) assert c.op == Op.COMPLEX
71: (8) assert repr(c) == ("Expr(Op.COMPLEX, (Expr(Op.INTEGER, (1, 4)),"
72: (27) " Expr(Op.INTEGER, (2, 4))))")
73: (8) assert c == c
74: (8) assert c != c2
75: (8) assert hash(c) is not None
76: (8) s = as_string("'123'")
77: (8) s2 = as_string('"ABC"')
78: (8) assert s.op == Op.STRING
79: (8) assert repr(s) == "Expr(Op.STRING, (\''123'\", 1))", repr(s)
80: (8) assert s == s
81: (8) assert s != s2
82: (8) a = as_array((n, m))
83: (8) b = as_array((n, ))
84: (8) assert a.op == Op.ARRAY
85: (8) assert repr(a) == ("Expr(Op.ARRAY, (Expr(Op.INTEGER, (123, 4)),"
86: (27) " Expr(Op.INTEGER, (456, 4))))")
87: (8) assert a == a
88: (8) assert a != b
89: (8) t = as_terms(x)
90: (8) u = as_terms(y)
91: (8) assert t.op == Op.TERMS
92: (8) assert repr(t) == "Expr(Op.TERMS, {Expr(Op.SYMBOL, 'x'): 1})"
93: (8) assert t == t
94: (8) assert t != u
95: (8) assert hash(t) is not None
96: (8) v = as_factors(x)
97: (8) w = as_factors(y)
98: (8) assert v.op == Op.FACTORS
99: (8) assert repr(v) == "Expr(Op.FACTORS, {Expr(Op.SYMBOL, 'x'): 1})"
100: (8) assert v == v
101: (8) assert w != v
102: (8) assert hash(v) is not None
103: (8) t = as_ternary(x, y, z)
104: (8) u = as_ternary(x, z, y)
105: (8) assert t.op == Op.TERNARY
106: (8) assert t == t
107: (8) assert t != u
108: (8) assert hash(t) is not None
109: (8) e = as_eq(x, y)
110: (8) f = as_lt(x, y)
111: (8) assert e.op == Op.RELATIONAL
112: (8) assert e == e
113: (8) assert e != f
114: (8) assert hash(e) is not None
115: (4) def test_tostring_fortran(self):
116: (8) x = as_symbol("x")
117: (8) y = as_symbol("y")
118: (8) z = as_symbol("z")
119: (8) n = as_number(123)
120: (8) m = as_number(456)
121: (8) a = as_array((n, m))
122: (8) c = as_complex(n, m)
123: (8) assert str(x) == "x"
124: (8) assert str(n) == "123"
125: (8) assert str(a) == "[123, 456]"
126: (8) assert str(c) == "(123, 456)"
127: (8) assert str(Expr(Op.TERMS, {x: 1})) == "x"
128: (8) assert str(Expr(Op.TERMS, {x: 2})) == "2 * x"
129: (8) assert str(Expr(Op.TERMS, {x: -1})) == "-x"
130: (8) assert str(Expr(Op.TERMS, {x: -2})) == "-2 * x"
131: (8) assert str(Expr(Op.TERMS, {x: 1, y: 1})) == "x + y"
132: (8) assert str(Expr(Op.TERMS, {x: -1, y: -1})) == "-x - y"

```

```

133: (8) assert str(Expr(Op.TERMS, {x: 2, y: 3})) == "2 * x + 3 * y"
134: (8) assert str(Expr(Op.TERMS, {x: -2, y: 3})) == "-2 * x + 3 * y"
135: (8) assert str(Expr(Op.TERMS, {x: 2, y: -3})) == "2 * x - 3 * y"
136: (8) assert str(Expr(Op.FACTORS, {x: 1})) == "x"
137: (8) assert str(Expr(Op.FACTORS, {x: 2})) == "x ** 2"
138: (8) assert str(Expr(Op.FACTORS, {x: -1})) == "x ** -1"
139: (8) assert str(Expr(Op.FACTORS, {x: -2})) == "x ** -2"
140: (8) assert str(Expr(Op.FACTORS, {x: 1, y: 1})) == "x * y"
141: (8) assert str(Expr(Op.FACTORS, {x: 2, y: 3})) == "x ** 2 * y ** 3"
142: (8) v = Expr(Op.FACTORS, {x: 2, Expr(Op.TERMS, {x: 1, y: 1}): 3})
143: (8) assert str(v) == "x ** 2 * (x + y) ** 3", str(v)
144: (8) v = Expr(Op.FACTORS, {x: 2, Expr(Op.FACTORS, {x: 1, y: 1}): 3})
145: (8) assert str(v) == "x ** 2 * (x * y) ** 3", str(v)
146: (8) assert str(Expr(Op.APPLY, ("f", (), {}))) == "f()"
147: (8) assert str(Expr(Op.APPLY, ("f", (x, ), {}))) == "f(x)"
148: (8) assert str(Expr(Op.APPLY, ("f", (x, y), {}))) == "f(x, y)"
149: (8) assert str(Expr(Op.INDEXING, ("f", x))) == "f[x]"
150: (8) assert str(as_ternary(x, y, z)) == "merge(y, z, x)"
151: (8) assert str(as_eq(x, y)) == "x .eq. y"
152: (8) assert str(as_ne(x, y)) == "x .ne. y"
153: (8) assert str(as_lt(x, y)) == "x .lt. y"
154: (8) assert str(as_le(x, y)) == "x .le. y"
155: (8) assert str(as_gt(x, y)) == "x .gt. y"
156: (8) assert str(as_ge(x, y)) == "x .ge. y"
157: (4) def test_tostring_c(self):
158: (8) language = Language.C
159: (8) x = as_symbol("x")
160: (8) y = as_symbol("y")
161: (8) z = as_symbol("z")
162: (8) n = as_number(123)
163: (8) assert Expr(Op.FACTORS, {x: 2}).tostring(language=language) == "x * x"
164: (8) assert (Expr(Op.FACTORS, {
165: (12)     x + y: 2
166: (8) }).tostring(language=language) == "(x + y) * (x + y)")
167: (8) assert Expr(Op.FACTORS, {
168: (12)     x: 12
169: (8) }).tostring(language=language) == "pow(x, 12)"
170: (8) assert as_apply(ArithOp.DIV, x,
171: (24)             y).tostring(language=language) == "x / y"
172: (8) assert (as_apply(ArithOp.DIV, x,
173: (25)             x + y).tostring(language=language) == "x / (x + y)")
174: (8) assert (as_apply(ArithOp.DIV, x - y, x +
175: (25)                 y).tostring(language=language) == "(x - y) / (x +
y"))
176: (8) assert (x + (x - y) / (x + y) +
177: (16)     n).tostring(language=language) == "123 + x + (x - y) / (x +
y)"
178: (8) assert as_ternary(x, y, z).tostring(language=language) == "(x?y:z)"
179: (8) assert as_eq(x, y).tostring(language=language) == "x == y"
180: (8) assert as_ne(x, y).tostring(language=language) == "x != y"
181: (8) assert as_lt(x, y).tostring(language=language) == "x < y"
182: (8) assert as_le(x, y).tostring(language=language) == "x <= y"
183: (8) assert as_gt(x, y).tostring(language=language) == "x > y"
184: (8) assert as_ge(x, y).tostring(language=language) == "x >= y"
185: (4) def test_operations(self):
186: (8) x = as_symbol("x")
187: (8) y = as_symbol("y")
188: (8) z = as_symbol("z")
189: (8) assert x + x == Expr(Op.TERMS, {x: 2})
190: (8) assert x - x == Expr(Op.INTEGER, (0, 4))
191: (8) assert x + y == Expr(Op.TERMS, {x: 1, y: 1})
192: (8) assert x - y == Expr(Op.TERMS, {x: 1, y: -1})
193: (8) assert x * x == Expr(Op.FACTORS, {x: 2})
194: (8) assert x * y == Expr(Op.FACTORS, {x: 1, y: 1})
195: (8) assert +x == x
196: (8) assert -x == Expr(Op.TERMS, {x: -1}), repr(-x)
197: (8) assert 2 * x == Expr(Op.TERMS, {x: 2})
198: (8) assert 2 + x == Expr(Op.TERMS, {x: 1, as_number(1): 2})
199: (8) assert 2 * x + 3 * y == Expr(Op.TERMS, {x: 2, y: 3})

```

```

200: (8) assert (x + y) * 2 == Expr(Op.TERMS, {x: 2, y: 2})
201: (8) assert x**2 == Expr(Op.FACTORS, {x: 2})
202: (8) assert (x + y)**2 == Expr(
203: (12) Op.TERMS,
204: (12) {
205: (16) Expr(Op.FACTORS, {x: 2}): 1,
206: (16) Expr(Op.FACTORS, {y: 2}): 1,
207: (16) Expr(Op.FACTORS, {
208: (20) x: 1,
209: (20) y: 1
210: (16) }): 2,
211: (12) },
212: (8)
213: (8) )
214: (8) assert (x + y) * x == x**2 + x * y
215: (8) assert (x + y)**2 == x**2 + 2 * x * y + y**2
216: (8) assert (x + y)**2 + (x - y)**2 == 2 * x**2 + 2 * y**2
217: (8) assert (x + y) * z == x * z + y * z
218: (8) assert z * (x + y) == x * z + y * z
219: (8) assert (x / 2) == as_apply(ArithOp.DIV, x, as_number(2))
220: (8) assert (2 * x / 2) == x
221: (8) assert (3 * x / 2) == as_apply(ArithOp.DIV, 3 * x, as_number(2))
222: (8) assert (4 * x / 2) == 2 * x
223: (8) assert (5 * x / 2) == as_apply(ArithOp.DIV, 5 * x, as_number(2))
224: (8) assert (6 * x / 2) == 3 * x
225: (8) assert ((3 * 5) * x / 6) == as_apply(ArithOp.DIV, 5 * x, as_number(2))
226: (12) assert (30 * x**2 * y**4 / (24 * x**3 * y**3)) == as_apply(
227: (8) ArithOp.DIV, 5 * y, 4 * x)
228: (46) assert ((15 * x / 6) / 5) == as_apply(ArithOp.DIV, x,
229: (8) as_number(2)), (15 * x / 6) / 5
230: (8) assert (x / (5 / x)) == as_apply(ArithOp.DIV, x**2, as_number(5))
231: (8) assert (x / 2.0) == Expr(Op.TERMS, {x: 0.5})
232: (8) s = as_string(''ABC'')
233: (8) t = as_string(''123'')
234: (8) assert s // t == Expr(Op.STRING, (''ABC123'', 1))
235: (8) assert s // x == Expr(Op.CONCAT, (s, x))
236: (8) assert x // s == Expr(Op.CONCAT, (x, s))
237: (8) c = as_complex(1.0, 2.0)
238: (8) assert -c == as_complex(-1.0, -2.0)
239: (8) assert c + c == as_expr((1 + 2j) * 2)
240: (4) assert c * c == as_expr((1 + 2j)**2)
241: (8) def test_substitute(self):
242: (8) x = as_symbol("x")
243: (8) y = as_symbol("y")
244: (8) z = as_symbol("z")
245: (8) a = as_array((x, y))
246: (8) assert x.substitute({x: y}) == y
247: (8) assert (x + y).substitute({x: z}) == y + z
248: (8) assert (x * y).substitute({x: z}) == y * z
249: (8) assert (x**4).substitute({x: z}) == z**4
250: (8) assert (x / y).substitute({x: z}) == z / y
251: (8) assert x.substitute({x: y + z}) == y + z
252: (8) assert a.substitute({x: y + z}) == as_array((y + z, y))
253: (26) assert as_ternary(x, y,
254: (8) z).substitute({x: y + z}) == as_ternary(y + z, y, z)
255: (4) assert as_eq(x, y).substitute({x: y + z}) == as_eq(y + z, y)
256: (8) def test_fromstring(self):
257: (8) x = as_symbol("x")
258: (8) y = as_symbol("y")
259: (8) z = as_symbol("z")
260: (8) f = as_symbol("f")
261: (8) s = as_string(''ABC'')
262: (8) t = as_string(''123'')
263: (8) a = as_array((x, y))
264: (8) assert fromstring("x") == x
265: (8) assert fromstring("+ x") == x
266: (8) assert fromstring("- x") == -x
267: (8) assert fromstring("x + y") == x + y
268: (8) assert fromstring("x + 1") == x + 1
269: (8) assert fromstring("x * y") == x * y

```

```

269: (8) assert fromstring("x * 2") == x * 2
270: (8) assert fromstring("x / y") == x / y
271: (8) assert fromstring("x ** 2", language=Language.Python) == x**2
272: (8) assert fromstring("x ** 2 ** 3", language=Language.Python) == x**2**3
273: (8) assert fromstring("(x + y) * z") == (x + y) * z
274: (8) assert fromstring("f(x)") == f(x)
275: (8) assert fromstring("f(x,y)") == f(x, y)
276: (8) assert fromstring("f[x]") == f[x]
277: (8) assert fromstring("f[x][y]") == f[x][y]
278: (8) assert fromstring('"ABC"') == s
279: (8) assert (normalize(
280: (12)     fromstring('"ABC" // "123" ',
281: (23)                 language=Language.Fortran)) == s // t)
282: (8) assert fromstring('f("ABC")') == f(s)
283: (8) assert fromstring('MYSTRKIND_"ABC"') == as_string('"ABC"',
284: (8) "MYSTRKIND")
285: (8) assert fromstring("(x, y)") == a, fromstring("(x, y)")
286: (8) assert fromstring("((x+y)*z/)") == as_array(((x + y) * z, ))
287: (8) assert fromstring("123") == as_number(123)
288: (8) assert fromstring("123_2") == as_number(123, 2)
289: (8) assert fromstring("123_myintkind") == as_number(123, "myintkind")
290: (8) assert fromstring("123.0") == as_number(123.0, 4)
291: (8) assert fromstring("123.0_4") == as_number(123.0, 4)
292: (8) assert fromstring("123.0_8") == as_number(123.0, 8)
293: (8) assert fromstring("123.0e0") == as_number(123.0, 4)
294: (8) assert fromstring("123.0d0") == as_number(123.0, 8)
295: (8) assert fromstring("123d0") == as_number(123.0, 8)
296: (8) assert fromstring("123e-0") == as_number(123.0, 4)
297: (8) assert fromstring("123d+0") == as_number(123.0, 8)
298: (8) assert fromstring("123.0_myrealmkind") == as_number(123.0,
299: (8) "myrealmkind")
300: (8) assert fromstring("3E4") == as_number(30000.0, 4)
301: (8) assert fromstring("(1, 2)") == as_complex(1, 2)
302: (53) assert fromstring("(1e2, PI)") == as_complex(as_number(100.0),
303: (8)                                     as_symbol("PI"))
304: (8) assert fromstring("[1, 2]") == as_array((as_number(1), as_number(2)))
305: (55) assert fromstring("POINT(x, y=1)") == as_apply(as_symbol("POINT"),
306: (55)   x,
307: (8)   y=as_number(1))
308: (12) assert fromstring(
309: (16)     'PERSON(name="John", age=50, shape=(/34, 23/))') == as_apply(
310: (16)         as_symbol("PERSON"),
311: (16)         name=as_string('"John"'),
312: (16)         age=as_number(50),
313: (12)         shape=as_array((as_number(34), as_number(23))),
314: (8)     )
315: (8) assert fromstring("x?y:z") == as_ternary(x, y, z)
316: (8) assert fromstring("*x") == as_deref(x)
317: (8) assert fromstring("**x") == as_deref(as_deref(x))
318: (8) assert fromstring("&x") == as_ref(x)
319: (8) assert fromstring("(x) * (y)") == as_deref(x) * as_deref(y)
320: (8) assert fromstring("(x) * y") == as_deref(x) * as_deref(y)
321: (8) assert fromstring("*x * y") == as_deref(x) * as_deref(y)
322: (8) assert fromstring("*x**y") == as_deref(x) * as_deref(y)
323: (8) assert fromstring("x == y") == as_eq(x, y)
324: (8) assert fromstring("x != y") == as_ne(x, y)
325: (8) assert fromstring("x < y") == as_lt(x, y)
326: (8) assert fromstring("x > y") == as_gt(x, y)
327: (8) assert fromstring("x <= y") == as_le(x, y)
328: (8) assert fromstring("x >= y") == as_ge(x, y)
329: (8) assert fromstring("x .eq. y", language=Language.Fortran) == as_eq(x,
330: (8) y)
331: (8) assert fromstring("x .ne. y", language=Language.Fortran) == as_ne(x,
332: (8) y)
333: (8) assert fromstring("x .lt. y", language=Language.Fortran) == as_lt(x,
334: (8) y)
335: (8) assert fromstring("x .gt. y", language=Language.Fortran) == as_gt(x,
336: (8) y)

```

```

332: (8)
y)
333: (8)
y)
334: (4)
335: (8)
336: (8)
337: (8)
338: (8)
339: (8)
340: (12)
341: (16)
342: (8)
343: (8)
344: (8)
345: (8)
346: (8)
347: (8)
348: (8)
349: (8)
350: (8)
351: (16)
352: (57)
353: (8)
354: (8)
355: (8)
356: (8)
357: (12)
358: (16)
359: (16)
360: (16)
361: (20)
362: (12)
363: (16)
364: (8)
365: (8)
366: (8)
367: (8)
368: (12)
369: (16)
370: (8)
371: (8)
372: (8)
373: (8)
374: (12)
375: (16)
376: (12)
377: (16)
378: (8)
379: (8)
380: (8)
381: (4)
382: (8)
383: (8)
384: (8)
385: (8)
386: (8)
387: (8)
388: (8)
389: (8)
390: (8)
391: (8)
392: (8)
393: (8)
394: (8)
395: (8)
396: (8)
397: (4)
398: (8)

    assert fromstring("x .le. y", language=Language.Fortran) == as_le(x,
    assert fromstring("x .ge. y", language=Language.Fortran) == as_ge(x,
def test_traverse(self):
    x = as_symbol("x")
    y = as_symbol("y")
    z = as_symbol("z")
    f = as_symbol("f")
    def replace_visit(s, r=z):
        if s == x:
            return r
        assert x.traverse(replace_visit) == z
        assert y.traverse(replace_visit) == y
        assert z.traverse(replace_visit) == z
        assert (f(y)).traverse(replace_visit) == f(y)
        assert (f(x)).traverse(replace_visit) == f(z)
        assert (f[y]).traverse(replace_visit) == f[y]
        assert (f[z]).traverse(replace_visit) == f[z]
        assert (x + y + z).traverse(replace_visit) == (2 * z + y)
        assert (x +
               f(y, x - z)).traverse(replace_visit) == (z +
   f(y, as_number(0)))
    assert as_eq(x, y).traverse(replace_visit) == as_eq(z, y)
    function_symbols = set()
    symbols = set()
    def collect_symbols(s):
        if s.op is Op.APPLY:
            oper = s.data[0]
            function_symbols.add(oper)
            if oper in symbols:
                symbols.remove(oper)
        elif s.op is Op.SYMBOL and s not in function_symbols:
            symbols.add(s)
    (x + f(y, x - z)).traverse(collect_symbols)
    assert function_symbols == {f}
    assert symbols == {x, y, z}
    def collect_symbols2(expr, symbols):
        if expr.op is Op.SYMBOL:
            symbols.add(expr)
    symbols = set()
    (x + f(y, x - z)).traverse(collect_symbols2, symbols)
    assert symbols == {x, y, z, f}
    def collect_symbols3(expr, symbols):
        if expr.op is Op.APPLY:
            return expr
        if expr.op is Op.SYMBOL:
            symbols.add(expr)
    symbols = set()
    (x + f(y, x - z)).traverse(collect_symbols3, symbols)
    assert symbols == {x}
def test_linear_solve(self):
    x = as_symbol("x")
    y = as_symbol("y")
    z = as_symbol("z")
    assert x.linear_solve(x) == (as_number(1), as_number(0))
    assert (x + 1).linear_solve(x) == (as_number(1), as_number(1))
    assert (2 * x).linear_solve(x) == (as_number(2), as_number(0))
    assert (2 * x + 3).linear_solve(x) == (as_number(2), as_number(3))
    assert as_number(3).linear_solve(x) == (as_number(0), as_number(3))
    assert y.linear_solve(x) == (as_number(0), y)
    assert (y * z).linear_solve(x) == (as_number(0), y * z)
    assert (x + y).linear_solve(x) == (as_number(1), y)
    assert (z * x + y).linear_solve(x) == (z, y)
    assert ((z + y) * x + y).linear_solve(x) == (z + y, y)
    assert (z * y * x + y).linear_solve(x) == (z * y, y)
    pytest.raises(RuntimeError, lambda: (x * x).linear_solve(x))
def test_as_numer_denom(self):
    x = as_symbol("x")

```

```

399: (8)             y = as_symbol("y")
400: (8)             n = as_number(123)
401: (8)             assert as_numer_denom(x) == (x, as_number(1))
402: (8)             assert as_numer_denom(x / n) == (x, n)
403: (8)             assert as_numer_denom(n / x) == (n, x)
404: (8)             assert as_numer_denom(x / y) == (x, y)
405: (8)             assert as_numer_denom(x * y) == (x * y, as_number(1))
406: (8)             assert as_numer_denom(n + x / y) == (x + n * y, y)
407: (8)             assert as_numer_denom(n + x / (y - x / n)) == (y * n**2, y * n - x)
408: (4)             def test_polynomial_atoms(self):
409: (8)                 x = as_symbol("x")
410: (8)                 y = as_symbol("y")
411: (8)                 n = as_number(123)
412: (8)                 assert x.polynomial_atoms() == {x}
413: (8)                 assert n.polynomial_atoms() == set()
414: (8)                 assert (y[x]).polynomial_atoms() == {y[x]}
415: (8)                 assert (y(x)).polynomial_atoms() == {y(x)}
416: (8)                 assert (y(x) + x).polynomial_atoms() == {y(x), x}
417: (8)                 assert (y(x) * x[y]).polynomial_atoms() == {y(x), x[y]}
418: (8)                 assert (y(x)**x).polynomial_atoms() == {y(x)}
```

-----

## File 191 - test\_value\_attrspec.py:

```

1: (0)             import os
2: (0)             import pytest
3: (0)             from . import util
4: (0)             class TestValueAttr(util.F2PyTest):
5: (4)                 sources = [util.getpath("tests", "src", "value_attrspec", "gh21665.f90")]
6: (4)                 def test_long_long_map(self):
7: (8)                     inp = 2
8: (8)                     out = self.module.fortfuncs.square(inp)
9: (8)                     exp_out = 4
10: (8)                    assert out == exp_out
```

-----

## File 192 - util.py:

```

1: (0)             """
2: (0)             Utility functions for
3: (0)             - building and importing modules on test time, using a temporary location
4: (0)             - detecting if compilers are present
5: (0)             - determining paths to tests
6: (0)             """
7: (0)             import glob
8: (0)             import os
9: (0)             import sys
10: (0)            import subprocess
11: (0)            import tempfile
12: (0)            import shutil
13: (0)            import atexit
14: (0)            import textwrap
15: (0)            import re
16: (0)            import pytest
17: (0)            import contextlib
18: (0)            import numpy
19: (0)            from pathlib import Path
20: (0)            from numpy.compat import asstr
21: (0)            from numpy._utils import asunicode
22: (0)            from numpy.testing import temppath, IS_WASM
23: (0)            from importlib import import_module
24: (0)            _module_dir = None
25: (0)            _module_num = 5403
26: (0)            if sys.platform == "cygwin":
27: (4)                NUMPY_INSTALL_ROOT = Path(__file__).parent.parent.parent
28: (4)                _module_list = list(NUMPY_INSTALL_ROOT.glob("**/*.dll"))
29: (0)            def _cleanup():
```

```

30: (4)
31: (4)
32: (8)
33: (12)
34: (8)
35: (12)
36: (8)
37: (12)
38: (8)
39: (12)
40: (8)
41: (0)
42: (4)
43: (4)
44: (8)
45: (8)
46: (8)
47: (12)
48: (4)
49: (0)
50: (4)
51: (4)
52: (4)
53: (4)
54: (4)
55: (8)
56: (4)
57: (0)
58: (4)
59: (4)
60: (8)
61: (8)
62: (12)
63: (16)
64: (12)
65: (16)
66: (16)
67: (8)
68: (8)
69: (12)
70: (8)
71: (4)
72: (4)
73: (0)
74: (0)
    module_name=None):
75: (4)
76: (4)
77: (4)
78: (4)
    numpy.f2py.main()
79: (4)
80: (4)
81: (4)
82: (4)
83: (8)
84: (12)
85: (8)
86: (8)
87: (8)
88: (8)
89: (8)
90: (12)
91: (4)
92: (4)
93: (8)
94: (4)
95: (4)
96: (8)

        global _module_dir
        if _module_dir is not None:
            try:
                sys.path.remove(_module_dir)
            except ValueError:
                pass
            try:
                shutil.rmtree(_module_dir)
            except OSError:
                pass
            _module_dir = None
    def get_module_dir():
        global _module_dir
        if _module_dir is None:
            _module_dir = tempfile.mkdtemp()
            atexit.register(_cleanup)
            if _module_dir not in sys.path:
                sys.path.insert(0, _module_dir)
    return _module_dir
def get_temp_module_name():
    global _module_num
    get_module_dir()
    name = "_test_ext_module_%d" % _module_num
    _module_num += 1
    if name in sys.modules:
        raise RuntimeError("Temporary module name already in use.")
    return name
def _memoize(func):
    memo = {}
    def wrapper(*a, **kw):
        key = repr((a, kw))
        if key not in memo:
            try:
                memo[key] = func(*a, **kw)
            except Exception as e:
                memo[key] = e
                raise
        ret = memo[key]
        if isinstance(ret, Exception):
            raise ret
        return ret
    wrapper.__name__ = func.__name__
    return wrapper
 @_memoize
def build_module(source_files, options=[], skip=[], only=[],
                 module_name=None):
    """
    Compile and import a f2py module, built from the given files.
    """
    code = f"import sys; sys.path = {sys.path!r}; import numpy.f2py;\n"
    d = get_module_dir()
    dst_sources = []
    f2py_sources = []
    for fn in source_files:
        if not os.path.isfile(fn):
            raise RuntimeError("%s is not a file" % fn)
        dst = os.path.join(d, os.path.basename(fn))
        shutil.copyfile(fn, dst)
        dst_sources.append(dst)
        base, ext = os.path.splitext(dst)
        if ext in (".f90", ".f", ".c", ".pyf"):
            f2py_sources.append(dst)
    assert f2py_sources
    if module_name is None:
        module_name = get_temp_module_name()
    f2py_opts = ["-c", "-m", module_name] + options + f2py_sources
    if skip:
        f2py_opts += ["skip:""] + skip

```

```

97: (4)
98: (8)
99: (4)
100: (4)
101: (8)
102: (8)
103: (8)
104: (29)
105: (29)
106: (8)
107: (8)
108: (12)
109: (31)
110: (4)
111: (8)
112: (8)
113: (12)
114: (4)
115: (8)
116: (12)
117: (8)
118: (8)
119: (12)
120: (12)
121: (8)
122: (4)
123: (0)
124: (0)
125: (15)
126: (15)
127: (15)
128: (15)
129: (15)
130: (4)
131: (4)
132: (4)
133: (4)
134: (8)
135: (4)
136: (8)
137: (12)
138: (8)
139: (28)
140: (28)
141: (28)
142: (28)
143: (0)
144: (0)
145: (4)
146: (4)
147: (8)
148: (4)
149: (4)
150: (8)
151: (4)
152: (8)
153: (8)
154: (8)
155: (8)
156: (12)
157: (12)
158: (12)
159: (12)
160: (8)
161: (8)
162: (8)
163: (8)
164: (8)
165: (42)

    if only:
        f2py_opts += ["only:"]
        cwd = os.getcwd()
        try:
            os.chdir(d)
            cmd = [sys.executable, "-c", code] + f2py_opts
            p = subprocess.Popen(cmd,
                                stdout=subprocess.PIPE,
                                stderr=subprocess.STDOUT)
            out, err = p.communicate()
            if p.returncode != 0:
                raise RuntimeError("Running f2py failed: %s\n%s" %
                                   (cmd[4:], asunicode(out)))
        finally:
            os.chdir(cwd)
            for fn in dst_sources:
                os.unlink(fn)
    if sys.platform == "cygwin":
        _module_list.extend(
            glob.glob(os.path.join(d, "{:s}*".format(module_name))))
    )
    subprocess.check_call(
        ["/usr/bin/rebase", "--database", "--oblivious", "--verbose"]
        + _module_list
    )
    return import_module(module_name)
@memoize
def build_code(source_code,
               options=[],
               skip=[],
               only=[],
               suffix=None,
               module_name=None):
    """
    Compile and import Fortran code using f2py.
    """
    if suffix is None:
        suffix = ".f"
    with tempPath(suffix=suffix) as path:
        with open(path, "w") as f:
            f.write(source_code)
    return build_module([path],
                       options=options,
                       skip=skip,
                       only=only,
                       module_name=module_name)

    _compiler_status = None
def _get_compiler_status():
    global _compiler_status
    if _compiler_status is not None:
        return _compiler_status
    _compiler_status = (False, False, False)
    if IS_WASM:
        return _compiler_status
    code = textwrap.dedent(f"""
        import os
        import sys
        sys.path = {repr(sys.path)}
        def configuration(parent_name='', top_path=None):
            global config
            from numpy.distutils.misc_util import Configuration
            config = Configuration('', parent_name, top_path)
            return config
        from numpy.distutils.core import setup
        setup(configuration=configuration)
        config_cmd = config.get_config_cmd()
        have_c = config_cmd.try_compile('void foo() {}')
        print('COMPILERS:%d,%d,%d' % (have_c,
                                       config.have_f77c(),


```

```

166: (42)                                     config.have_f90c()))
167: (8)          sys.exit(99)
168: (8)
169: (4)          """
170: (4)          code = code % dict(syspath=repr(sys.path))
171: (4)          tmpdir = tempfile.mkdtemp()
172: (8)          try:
173: (8)              script = os.path.join(tmpdir, "setup.py")
174: (12)             with open(script, "w") as f:
175: (8)                 f.write(code)
176: (8)             cmd = [sys.executable, "setup.py", "config"]
177: (29)             p = subprocess.Popen(cmd,
178: (29)                             stdout=subprocess.PIPE,
179: (29)                             stderr=subprocess.STDOUT,
180: (8)                             cwd=tmpdir)
181: (4)             out, err = p.communicate()
182: (8)         finally:
183: (4)             shutil.rmtree(tmpdir)
184: (4)         m = re.search(br"COMPILERS:(\d+),(\d+),(\d+)", out)
185: (8)         if m:
186: (12)             _compiler_status = (
187: (12)                 bool(int(m.group(1))),
188: (12)                 bool(int(m.group(2))),
189: (8)                 bool(int(m.group(3))),
190: (4)             )
191: (0)             return _compiler_status
192: (4)         def has_c_compiler():
193: (0)             return _get_compiler_status()[0]
194: (4)         def has_f77_compiler():
195: (0)             return _get_compiler_status()[1]
196: (4)         def has_f90_compiler():
197: (0)             return _get_compiler_status()[2]
198: (0)         @_memoize
199: (4)         def build_module_distutils(source_files, config_code, module_name, **kw):
200: (4)             """
201: (4)             Build a module via distutils and import it.
202: (4)             """
203: (4)             d = get_module_dir()
204: (4)             dst_sources = []
205: (8)             for fn in source_files:
206: (12)                 if not os.path.isfile(fn):
207: (8)                     raise RuntimeError("%s is not a file" % fn)
208: (8)                 dst = os.path.join(d, os.path.basename(fn))
209: (8)                 shutil.copyfile(fn, dst)
210: (8)                 dst_sources.append(dst)
211: (4)             config_code = textwrap.dedent(config_code).replace("\n", "\n      ")
212: (0)             code = fn"""
213: (0)             import os
214: (0)             import sys
215: (0)             sys.path = {repr(sys.path)}
216: (4)             def configuration(parent_name='', top_path=None):
217: (4)                 from numpy.distutils.misc_util import Configuration
218: (4)                 config = Configuration('', parent_name, top_path)
219: (4)                 {config_code}
220: (0)             return config
221: (4)             if __name__ == "__main__":
222: (4)                 from numpy.distutils.core import setup
223: (4)                 setup(configuration=configuration)
224: (4)                 """
225: (4)                 script = os.path.join(d, get_temp_module_name() + ".py")
226: (4)                 dst_sources.append(script)
227: (8)                 with open(script, "wb") as f:
228: (4)                     f.write(code.encode('latin1'))
229: (4)                     cwd = os.getcwd()
230: (8)                     try:
231: (8)                         os.chdir(d)
232: (8)                         cmd = [sys.executable, script, "build_ext", "-i"]
233: (29)                         p = subprocess.Popen(cmd,
234: (29)                             stdout=subprocess.PIPE,
235: (29)                             stderr=subprocess.STDOUT)

```

```

235: (8)                     out, err = p.communicate()
236: (8)                     if p.returncode != 0:
237: (12)                         raise RuntimeError("Running distutils build failed: %s\n%s" %
238: (31)   (cmd[4:], asstr(out)))
239: (4)                     finally:
240: (8)                         os.chdir(cwd)
241: (8)                         for fn in dst_sources:
242: (12)                             os.unlink(fn)
243: (4)                     __import__(module_name)
244: (4)                     return sys.modules[module_name]
245: (0) class F2PyTest:
246: (4)     code = None
247: (4)     sources = None
248: (4)     options = []
249: (4)     skip = []
250: (4)     only = []
251: (4)     suffix = ".f"
252: (4)     module = None
253: (4)     @property
254: (4)     def module_name(self):
255: (8)         cls = type(self)
256: (8)         return f'_{cls.__module__.rsplit(".",1)[-1]}'
257: (4)     def setup_method(self):
258: (8)         if sys.platform == "win32":
259: (12)             pytest.skip("Fails with MinGW64 Gfortran (Issue #9673)")
260: (8)         if self.module is not None:
261: (12)             return
262: (8)         if not has_c_compiler():
263: (12)             pytest.skip("No C compiler available")
264: (8)         codes = []
265: (8)         if self.sources:
266: (12)             codes.extend(self.sources)
267: (8)         if self.code is not None:
268: (12)             codes.append(self.suffix)
269: (8)         needs_f77 = False
270: (8)         needs_f90 = False
271: (8)         needs_pyf = False
272: (8)         for fn in codes:
273: (12)             if str(fn).endswith(".f"):
274: (16)                 needs_f77 = True
275: (12)             elif str(fn).endswith(".f90"):
276: (16)                 needs_f90 = True
277: (12)             elif str(fn).endswith(".pyf"):
278: (16)                 needs_pyf = True
279: (8)         if needs_f77 and not has_f77_compiler():
280: (12)             pytest.skip("No Fortran 77 compiler available")
281: (8)         if needs_f90 and not has_f90_compiler():
282: (12)             pytest.skip("No Fortran 90 compiler available")
283: (8)         if needs_pyf and not (has_f90_compiler() or has_f77_compiler()):
284: (12)             pytest.skip("No Fortran compiler available")
285: (8)         if self.code is not None:
286: (12)             self.module = build_code(
287: (16)                 self.code,
288: (16)                 options=self.options,
289: (16)                 skip=self.skip,
290: (16)                 only=self.only,
291: (16)                 suffix=self.suffix,
292: (16)                 module_name=self.module_name,
293: (12)             )
294: (8)         if self.sources is not None:
295: (12)             self.module = build_module(
296: (16)                 self.sources,
297: (16)                 options=self.options,
298: (16)                 skip=self.skip,
299: (16)                 only=self.only,
300: (16)                 module_name=self.module_name,
301: (12)             )
302: (0)     def getpath(*a):

```

```

303: (4)         d = Path(numpy.f2py.__file__).parent.resolve()
304: (4)         return d.joinpath(*a)
305: (0) @contextlib.contextmanager
306: (0) def switchdir(path):
307: (4)     curpath = Path.cwd()
308: (4)     os.chdir(path)
309: (4)     try:
310: (8)         yield
311: (4)     finally:
312: (8)         os.chdir(curpath)

```

-----

File 193 - \_\_init\_\_.py:

```
1: (0)
```

-----

File 194 - \_backend.py:

```

1: (0)         from __future__ import annotations
2: (0)         from abc import ABC, abstractmethod
3: (0) class Backend(ABC):
4: (4)     def __init__(
5: (8)             self,
6: (8)             modulename,
7: (8)             sources,
8: (8)             extra_objects,
9: (8)             build_dir,
10: (8)            include_dirs,
11: (8)            library_dirs,
12: (8)            libraries,
13: (8)            define_macros,
14: (8)            undef_macros,
15: (8)            f2py_flags,
16: (8)            sysinfo_flags,
17: (8)            fc_flags,
18: (8)            flib_flags,
19: (8)            setup_flags,
20: (8)            remove_build_dir,
21: (8)            extra_dat,
22: (4)        ):
23: (8)             self.modulename = modulename
24: (8)             self.sources = sources
25: (8)             self.extra_objects = extra_objects
26: (8)             self.build_dir = build_dir
27: (8)             self.include_dirs = include_dirs
28: (8)             self.library_dirs = library_dirs
29: (8)             self.libraries = libraries
30: (8)             self.define_macros = define_macros
31: (8)             self.undef_macros = undef_macros
32: (8)             self.f2py_flags = f2py_flags
33: (8)             self.sysinfo_flags = sysinfo_flags
34: (8)             self.fc_flags = fc_flags
35: (8)             self.flib_flags = flib_flags
36: (8)             self.setup_flags = setup_flags
37: (8)             self.remove_build_dir = remove_build_dir
38: (8)             self.extra_dat = extra_dat
39: (4)     @abstractmethod
40: (4)     def compile(self) -> None:
41: (8)             """Compile the wrapper."""
42: (8)             pass

```

-----

File 195 - \_distutils.py:

```
1: (0)         from ._backend import Backend
```

```

2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (0)
8: (0)
9: (0)
10: (0)
11: (4)
12: (8)
13: (12)
14: (12)
15: (12)
16: (12)
17: (12)
18: (8)
19: (8)
20: (4)
21: (8)
22: (8)
23: (12)
24: (8)
25: (12)
26: (12)
27: (12)
28: (12)
29: (12)
30: (12)
31: (12)
32: (12)
33: (12)
34: (8)
35: (8)
36: (12)
37: (16)
38: (16)
39: (20)
40: (24)
41: (24)
42: (20)
43: (16)
44: (8)
45: (8)
46: (8)
47: (12)
48: (16)
49: (16)
50: (16)
51: (16)
52: (16)
53: (16)
54: (16)
55: (16)
56: (12)
57: (8)
58: (8)
59: (12)
60: (8)
61: (12)
62: (8)
63: (8)
64: (12)
65: (12)

-----
```

File 196 - \_meson.py:

```

1: (0)         from __future__ import annotations
2: (0)         import os
3: (0)         import errno
4: (0)         import shutil
5: (0)         import subprocess
6: (0)         import sys
7: (0)         from pathlib import Path
8: (0)         from .backend import Backend
9: (0)         from string import Template
10: (0)        from itertools import chain
11: (0)        import warnings
12: (0)        class MesonTemplate:
13: (4)            """Template meson build file generation class."""
14: (4)            def __init__(
15: (8)                self,
16: (8)                modulename: str,
17: (8)                sources: list[Path],
18: (8)                deps: list[str],
19: (8)                libraries: list[str],
20: (8)                library_dirs: list[Path],
21: (8)                include_dirs: list[Path],
22: (8)                object_files: list[Path],
23: (8)                linker_args: list[str],
24: (8)                c_args: list[str],
25: (8)                build_type: str,
26: (8)                python_exe: str,
27: (4)            ):
28: (8)                self.modulename = modulename
29: (8)                self.build_template_path = (
30: (12)                    Path(__file__).parent.absolute() / "meson.build.template"
31: (8)
32: (8)                self.sources = sources
33: (8)                self.deps = deps
34: (8)                self.libraries = libraries
35: (8)                self.library_dirs = library_dirs
36: (8)                if include_dirs is not None:
37: (12)                    self.include_dirs = include_dirs
38: (8)                else:
39: (12)                    self.include_dirs = []
40: (8)                self.substitutions = {}
41: (8)                self.objects = object_files
42: (8)                self.pipeline = [
43: (12)                    self.initialize_template,
44: (12)                    self.sources_substitution,
45: (12)                    self.deps_substitution,
46: (12)                    self.include_substitution,
47: (12)                    self.libraries_substitution,
48: (8)
49: (8)                self.build_type = build_type
50: (8)                self.python_exe = python_exe
51: (4)            def meson_build_template(self) -> str:
52: (8)                if not self.build_template_path.is_file():
53: (12)                    raise FileNotFoundError(
54: (16)                        errno.ENOENT,
55: (16)                        "Meson build template"
56: (16)                        f" {self.build_template_path.absolute()}"
57: (16)                        " does not exist.",
58: (12)
59: (8)                    )
60: (4)                    return self.build_template_path.read_text()
61: (8)            def initialize_template(self) -> None:
62: (8)                self.substitutions["modulename"] = self.modulename
63: (8)                self.substitutions["buildtype"] = self.build_type
64: (8)                self.substitutions["python"] = self.python_exe
65: (8)            def sources_substitution(self) -> None:
66: (8)                indent = " " * 21
67: (12)                self.substitutions["source_list"] = f",\n{indent}".join(
68: (8)                    [f"{indent}{source}" for source in self.sources]
69: (8)
69: (4)            def deps_substitution(self) -> None:

```

```

70: (8)           indent = " " * 21
71: (8)           self.substitutions["dep_list"] = f",\n{indent}".join(
72: (12)             [f"{indent}dependency('{dep}')" for dep in self.deps]
73: (8)
74: (4)           def libraries_substitution(self) -> None:
75: (8)               self.substitutions["lib_dir_declarations"] = "\n".join(
76: (12)                 [
77: (16)                   f"lib_dir_{i} = declare_dependency(link_args : ['-L{lib_dir}'])"
78: (16)
79: (12)
80: (8)
81: (8)
82: (12)
83: (16)           self.substitutions["lib_declarations"] = "\n".join(
84: (16)             [
85: (12)               f"{lib} = declare_dependency(link_args : ['-l{lib}'])"
86: (8)                 for lib in self.libraries
87: (8)
88: (8)
89: (12)           indent = " " * 21
90: (8)           self.substitutions["lib_list"] = f"\n{indent}".join(
91: (8)             [f"{indent}{lib},," for lib in self.libraries]
92: (12)
93: (8)
94: (4)           def include_substitution(self) -> None:
95: (8)               indent = " " * 21
96: (8)               self.substitutions["inc_list"] = f",\n{indent}".join(
97: (12)                 [f"{indent}'{inc}'" for inc in self.include_dirs]
98: (8)
99: (4)           def generate_meson_build(self):
100: (8)             for node in self.pipeline:
101: (12)               node()
102: (8)             template = Template(self.meson_build_template())
103: (8)             return template.substitute(self.substitutions)
104: (0)           class MesonBackend(Backend):
105: (4)             def __init__(self, *args, **kwargs):
106: (8)               super().__init__(*args, **kwargs)
107: (8)               self.dependencies = self.extra_dat.get("dependencies", [])
108: (8)               self.meson_build_dir = "bkdir"
109: (8)               self.build_type =
110: (12)                 "debug" if any("debug" in flag for flag in self.fc_flags) else
"release"
111: (8)
112: (4)             def _move_exec_to_root(self, build_dir: Path):
113: (8)               walk_dir = Path(build_dir) / self.meson_build_dir
114: (8)               path_objects = chain(
115: (12)                 walk_dir.glob(f"{self.modulename}*.so"),
116: (12)                 walk_dir.glob(f"{self.modulename}*.pyd"),
117: (8)
118: (8)
119: (12)
120: (12)
121: (16)
122: (12)
123: (12)
124: (4)             def write_meson_build(self, build_dir: Path) -> None:
125: (8)               """Writes the meson build file at specified location"""
126: (8)               meson_template = MesonTemplate(
127: (12)                 self.modulename,
128: (12)                 self.sources,
129: (12)                 self.dependencies,
130: (12)                 self.libraries,
131: (12)                 self.library_dirs,
132: (12)                 self.include_dirs,
133: (12)                 self.extra_objects,
134: (12)                 self.flib_flags,
135: (12)                 self.fc_flags,
136: (12)                 self.build_type,

```

```

137: (12)                     sys.executable,
138: (8)                      )
139: (8)                      src = meson_template.generate_meson_build()
140: (8)                      Path(build_dir).mkdir(parents=True, exist_ok=True)
141: (8)                      meson_build_file = Path(build_dir) / "meson.build"
142: (8)                      meson_build_file.write_text(src)
143: (8)                      return meson_build_file
144: (4)                      def _run_subprocess_command(self, command, cwd):
145: (8)                        subprocess.run(command, cwd=cwd, check=True)
146: (4)                      def run_meson(self, build_dir: Path):
147: (8)                        setup_command = ["meson", "setup", self.meson_build_dir]
148: (8)                        self._run_subprocess_command(setup_command, build_dir)
149: (8)                        compile_command = ["meson", "compile", "-C", self.meson_build_dir]
150: (8)                        self._run_subprocess_command(compile_command, build_dir)
151: (4)                      def compile(self) -> None:
152: (8)                        self.sources = _prepare_sources(self.modulename, self.sources,
self.build_dir)
153: (8)                        self.write_meson_build(self.build_dir)
154: (8)                        self.run_meson(self.build_dir)
155: (8)                        self._move_exec_to_root(self.build_dir)
156: (0)                      def _prepare_sources(mname, sources, bdir):
157: (4)                        extended_sources = sources.copy()
158: (4)                        Path(bdir).mkdir(parents=True, exist_ok=True)
159: (4)                        for source in sources:
160: (8)                          if Path(source).exists() and Path(source).is_file():
161: (12)                            shutil.copy(source, bdir)
162: (4)                        generated_sources = [
163: (8)                          Path(f"{mname}module.c"),
164: (8)                          Path(f"{mname}-f2pywrappers2.f90"),
165: (8)                          Path(f"{mname}-f2pywrappers.f"),
166: (4)                        ]
167: (4)                        bdir = Path(bdir)
168: (4)                        for generated_source in generated_sources:
169: (8)                          if generated_source.exists():
170: (12)                            shutil.copy(generated_source, bdir / generated_source.name)
171: (12)                            extended_sources.append(generated_source.name)
172: (12)                            generated_source.unlink()
173: (4)                        extended_sources = [
174: (8)                          Path(source).name
175: (8)                          for source in extended_sources
176: (8)                          if not Path(source).suffix == ".pyf"
177: (4)                        ]
178: (4)                        return extended_sources

```

---

## File 197 - helper.py:

```

1: (0)      """
2: (0)      Discrete Fourier Transforms - helper.py
3: (0)      """
4: (0)      from numpy.core import integer, empty, arange, asarray, roll
5: (0)      from numpy.core.overrides import array_function_dispatch, set_module
6: (0)      __all__ = ['fftshift', 'ifftshift', 'fftfreq', 'rfftfreq']
7: (0)      integer_types = (int, integer)
8: (0)      def _fftshift_dispatcher(x, axes=None):
9: (4)        return (x,)
10: (0)      @array_function_dispatch(_fftshift_dispatcher, module='numpy.fft')
11: (0)      def fftshift(x, axes=None):
12: (4)        """
13: (4)          Shift the zero-frequency component to the center of the spectrum.
14: (4)          This function swaps half-spaces for all axes listed (defaults to all).
15: (4)          Note that ``y[0]`` is the Nyquist component only if ``len(x)`` is even.
16: (4)          Parameters
17: (4)          -----
18: (4)          x : array_like
19: (8)            Input array.
20: (4)          axes : int or shape tuple, optional
21: (8)            Axes over which to shift. Default is None, which shifts all axes.

```

```

22: (4)          Returns
23: (4)          -----
24: (4)          y : ndarray
25: (8)          The shifted array.
26: (4)          See Also
27: (4)          -----
28: (4)          ifftshift : The inverse of `fftshift`.
29: (4)          Examples
30: (4)          -----
31: (4)          >>> freqs = np.fft.fftfreq(10, 0.1)
32: (4)          >>> freqs
33: (4)          array([ 0.,  1.,  2., ..., -3., -2., -1.])
34: (4)          >>> np.fft.fftshift(freqs)
35: (4)          array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
36: (4)          Shift the zero-frequency component only along the second axis:
37: (4)          >>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
38: (4)          >>> freqs
39: (4)          array([[ 0.,  1.,  2.],
40: (11)           [ 3.,  4., -4.],
41: (11)           [-3., -2., -1.]])
42: (4)          >>> np.fft.fftshift(freqs, axes=(1,))
43: (4)          array([[ 2.,  0.,  1.],
44: (11)           [-4.,  3.,  4.],
45: (11)           [-1., -3., -2.]])
46: (4)          """
47: (4)          x = asarray(x)
48: (4)          if axes is None:
49: (8)              axes = tuple(range(x.ndim))
50: (8)              shift = [dim // 2 for dim in x.shape]
51: (4)          elif isinstance(axes, integer_types):
52: (8)              shift = x.shape[axes] // 2
53: (4)          else:
54: (8)              shift = [x.shape[ax] // 2 for ax in axes]
55: (4)          return roll(x, shift, axes)
56: (0)          @array_function_dispatch(_fftshift_dispatcher, module='numpy.fft')
57: (0)          def ifftshift(x, axes=None):
58: (4)          """
59: (4)          The inverse of `fftshift`. Although identical for even-length `x`, the
60: (4)          functions differ by one sample for odd-length `x`.
61: (4)          Parameters
62: (4)          -----
63: (4)          x : array_like
64: (8)              Input array.
65: (4)          axes : int or shape tuple, optional
66: (8)              Axes over which to calculate. Defaults to None, which shifts all
axes.
67: (4)          Returns
68: (4)          -----
69: (4)          y : ndarray
70: (8)          The shifted array.
71: (4)          See Also
72: (4)          -----
73: (4)          fftshift : Shift zero-frequency component to the center of the spectrum.
74: (4)          Examples
75: (4)          -----
76: (4)          >>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
77: (4)          >>> freqs
78: (4)          array([[ 0.,  1.,  2.],
79: (11)           [ 3.,  4., -4.],
80: (11)           [-3., -2., -1.]])
81: (4)          >>> np.fft.ifftshift(np.fft.fftshift(freqs))
82: (4)          array([[ 0.,  1.,  2.],
83: (11)           [ 3.,  4., -4.],
84: (11)           [-3., -2., -1.]])
85: (4)          """
86: (4)          x = asarray(x)
87: (4)          if axes is None:
88: (8)              axes = tuple(range(x.ndim))
89: (8)              shift = [-(dim // 2) for dim in x.shape]

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

90: (4)
91: (8)
92: (4)
93: (8)
94: (4)
95: (0)
96: (0)
97: (4)
98: (4)
99: (4)
100: (4)
101: (4)
cycles/second.
102: (4)
103: (6)
104: (6)
105: (4)
106: (4)
107: (4)
108: (8)
109: (4)
110: (8)
111: (4)
112: (4)
113: (4)
114: (8)
115: (4)
116: (4)
117: (4)
118: (4)
119: (4)
120: (4)
121: (4)
122: (4)
123: (4)
124: (4)
125: (4)
126: (8)
127: (4)
128: (4)
129: (4)
130: (4)
131: (4)
132: (4)
133: (4)
134: (4)
135: (0)
136: (0)
137: (4)
138: (4)
139: (4)
140: (4)
141: (4)
142: (4)
cycles/second.
143: (4)
144: (6)
145: (6)
146: (4)
147: (4)
148: (4)
149: (4)
150: (4)
151: (8)
152: (4)
153: (8)
154: (4)
155: (4)
156: (4)

    elif isinstance(axes, integer_types):
        shift = -(x.shape[axes] // 2)
    else:
        shift = [-(x.shape[ax] // 2) for ax in axes]
    return roll(x, shift, axes)
@set_module('numpy.fft')
def fftfreq(n, d=1.0):
    """
    Return the Discrete Fourier Transform sample frequencies.
    The returned float array `f` contains the frequency bin centers in cycles
    per unit of the sample spacing (with zero at the start). For instance, if
    the sample spacing is in seconds, then the frequency unit is

    Given a window length `n` and a sample spacing `d`::
        f = [0, 1, ..., n/2-1, -n/2, ..., -1] / (d*n)    if n is even
        f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n)    if n is odd
    Parameters
    -----
    n : int
        Window length.
    d : scalar, optional
        Sample spacing (inverse of the sampling rate). Defaults to 1.
    Returns
    -----
    f : ndarray
        Array of length `n` containing the sample frequencies.
    Examples
    -----
    >>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
    >>> fourier = np.fft.fft(signal)
    >>> n = signal.size
    >>> timestep = 0.1
    >>> freq = np.fft.fftfreq(n, d=timestep)
    >>> freq
    array([ 0. ,  1.25,  2.5 , ..., -3.75, -2.5 , -1.25])
    """
    if not isinstance(n, integer_types):
        raise ValueError("n should be an integer")
    val = 1.0 / (n * d)
    results = empty(n, int)
    N = (n-1)//2 + 1
    p1 = arange(0, N, dtype=int)
    results[:N] = p1
    p2 = arange(-(N//2), 0, dtype=int)
    results[N:] = p2
    return results * val
@set_module('numpy.fft')
def rfftfreq(n, d=1.0):
    """
    Return the Discrete Fourier Transform sample frequencies
    (for usage with rfft, irfft).
    The returned float array `f` contains the frequency bin centers in cycles
    per unit of the sample spacing (with zero at the start). For instance, if
    the sample spacing is in seconds, then the frequency unit is

    Given a window length `n` and a sample spacing `d`::
        f = [0, 1, ..., n/2-1, n/2] / (d*n)    if n is even
        f = [0, 1, ..., (n-1)/2-1, (n-1)/2] / (d*n)    if n is odd
    Unlike `fftfreq` (but like `scipy.fftpack.rfftfreq`)
    the Nyquist frequency component is considered to be positive.
    Parameters
    -----
    n : int
        Window length.
    d : scalar, optional
        Sample spacing (inverse of the sampling rate). Defaults to 1.
    Returns
    -----
    f : ndarray

```

```

157: (8)          Array of length ``n//2 + 1`` containing the sample frequencies.
158: (4)          Examples
159: (4)
160: (4)          -----
161: (4)          >>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5, -3, 4], dtype=float)
162: (4)          >>> fourier = np.fft.rfft(signal)
163: (4)          >>> n = signal.size
164: (4)          >>> sample_rate = 100
165: (4)          >>> freq = np.fft.fftfreq(n, d=1./sample_rate)
166: (4)          >>> freq
167: (4)          array([ 0., 10., 20., ..., -30., -20., -10.])
168: (4)          >>> freq = np.fft.rfftfreq(n, d=1./sample_rate)
169: (4)          >>> freq
170: (4)          array([ 0., 10., 20., 30., 40., 50.])
171: (4)          """
172: (8)          if not isinstance(n, integer_types):
173: (4)              raise ValueError("n should be an integer")
174: (4)          val = 1.0/(n*d)
175: (4)          N = n//2 + 1
176: (4)          results = arange(0, N, dtype=int)
177: (4)          return results * val
-----
```

File 198 - `__init__.py`:

```

1: (0)          def f2py_build_generator(name):
2: (4)          if name == "meson":
3: (8)              from .meson import MesonBackend
4: (8)              return MesonBackend
5: (4)          elif name == "distutils":
6: (8)              from .distutils import DistutilsBackend
7: (8)              return DistutilsBackend
8: (4)          else:
9: (8)              raise ValueError(f"Unknown backend: {name}")
-----
```

File 199 - `_pocketfft.py`:

```

1: (0)          """
2: (0)          Discrete Fourier Transforms
3: (0)          Routines in this module:
4: (0)          fft(a, n=None, axis=-1, norm="backward")
5: (0)          ifft(a, n=None, axis=-1, norm="backward")
6: (0)          rfft(a, n=None, axis=-1, norm="backward")
7: (0)          irfft(a, n=None, axis=-1, norm="backward")
8: (0)          hfft(a, n=None, axis=-1, norm="backward")
9: (0)          ihfft(a, n=None, axis=-1, norm="backward")
10: (0)          fftn(a, s=None, axes=None, norm="backward")
11: (0)          ifftn(a, s=None, axes=None, norm="backward")
12: (0)          rfftn(a, s=None, axes=None, norm="backward")
13: (0)          irfftn(a, s=None, axes=None, norm="backward")
14: (0)          fft2(a, s=None, axes=(-2,-1), norm="backward")
15: (0)          ifft2(a, s=None, axes=(-2, -1), norm="backward")
16: (0)          rfft2(a, s=None, axes=(-2, -1), norm="backward")
17: (0)          irfft2(a, s=None, axes=(-2, -1), norm="backward")
18: (0)          i = inverse transform
19: (0)          r = transform of purely real data
20: (0)          h = Hermite transform
21: (0)          n = n-dimensional transform
22: (0)          2 = 2-dimensional transform
23: (0)          (Note: 2D routines are just nD routines with different default
24: (0)          behavior.)
25: (0)          """
26: (0)          __all__ = ['fft', 'ifft', 'rfft', 'irfft', 'hfft', 'ihfft', 'rfftn',
27: (11)          'irfftn', 'rfft2', 'irfft2', 'fft2', 'ifft2', 'fftn', 'ifftn']
28: (0)          import functools
29: (0)          from numpy.core import asarray, zeros, swapaxes, conjugate, take, sqrt
30: (0)          from . import _pocketfft_internal as pfi
-----
```

```

31: (0)
32: (0)
33: (0)
34: (4)
35: (0)
36: (4)
37: (4)
38: (8)
39: (4)
40: (4)
41: (8)
42: (8)
43: (8)
44: (12)
45: (12)
46: (8)
47: (12)
48: (12)
49: (12)
50: (12)
51: (12)
52: (4)
53: (8)
54: (4)
55: (8)
56: (8)
57: (8)
58: (4)
59: (0)
60: (4)
61: (8)
specified.")
62: (4)
63: (8)
64: (4)
65: (8)
66: (4)
67: (8)
68: (4)
69: (21)
70: (0)
71: (4)
72: (8)
specified.")
73: (4)
74: (8)
75: (4)
76: (8)
77: (4)
78: (8)
79: (4)
80: (21)
81: (0)
82: (23)
83: (0)
84: (4)
85: (8)
86: (4)
87: (8)
88: (25)
89: (0)
90: (4)
91: (0)
92: (0)
93: (4)
94: (4)
95: (4)
96: (4)
97: (4)

from numpy.core.multiarray import normalize_axis_index
from numpy.core import overrides
array_function_dispatch = functools.partial(
    overrides.array_function_dispatch, module='numpy.fft')
def _raw_fft(a, n, axis, is_real, is_forward, inv_norm):
    axis = normalize_axis_index(axis, a.ndim)
    if n is None:
        n = a.shape[axis]
    fct = 1/inv_norm
    if a.shape[axis] != n:
        s = list(a.shape)
        index = [slice(None)]*len(s)
        if s[axis] > n:
            index[axis] = slice(0, n)
        a = a[tuple(index)]
    else:
        index[axis] = slice(0, s[axis])
        s[axis] = n
        z = zeros(s, a.dtype.char)
        z[tuple(index)] = a
        a = z
    if axis == a.ndim-1:
        r = pfi.execute(a, is_real, is_forward, fct)
    else:
        a = swapaxes(a, axis, -1)
        r = pfi.execute(a, is_real, is_forward, fct)
        r = swapaxes(r, axis, -1)
    return r
def _get_forward_norm(n, norm):
    if n < 1:
        raise ValueError(f"Invalid number of FFT data points ({n})"
specified.")
    if norm is None or norm == "backward":
        return 1
    elif norm == "ortho":
        return sqrt(n)
    elif norm == "forward":
        return n
    raise ValueError(f'Invalid norm value {norm}; should be "backward", '
                  '"ortho" or "forward".')
def _get_backward_norm(n, norm):
    if n < 1:
        raise ValueError(f"Invalid number of FFT data points ({n})"
specified.")
    if norm is None or norm == "backward":
        return n
    elif norm == "ortho":
        return sqrt(n)
    elif norm == "forward":
        return 1
    raise ValueError(f'Invalid norm value {norm}; should be "backward", '
                  '"ortho" or "forward".')
_SWAP_DIRECTION_MAP = {"backward": "forward", None: "forward",
                      "ortho": "ortho", "forward": "backward"}
def _swap_direction(norm):
    try:
        return _SWAP_DIRECTION_MAP[norm]
    except KeyError:
        raise ValueError(f'Invalid norm value {norm}; should be "backward", '
                      '"ortho" or "forward".') from None
def _fft_dispatcher(a, n=None, axis=None, norm=None):
    return (a,)
@array_function_dispatch(_fft_dispatcher)
def fft(a, n=None, axis=-1, norm=None):
    """
    Compute the one-dimensional discrete Fourier Transform.
    This function computes the one-dimensional *n*-point discrete Fourier
    Transform (DFT) with the efficient Fast Fourier Transform (FFT)
    algorithm [CT].

```

```

98: (4)                               Parameters
99: (4)                               -----
100: (4)                             a : array_like
101: (8)                             Input array, can be complex.
102: (4)                             n : int, optional
103: (8)                             Length of the transformed axis of the output.
104: (8)                             If `n` is smaller than the length of the input, the input is cropped.
105: (8)                             If it is larger, the input is padded with zeros. If `n` is not given,
106: (8)                             the length of the input along the axis specified by `axis` is used.
107: (4)                             axis : int, optional
108: (8)                             Axis over which to compute the FFT. If not given, the last axis is
109: (8)                             used.
110: (4)                             norm : {"backward", "ortho", "forward"}, optional
111: (8)                             .. versionadded:: 1.10.0
112: (8)                             Normalization mode (see `numpy.fft`). Default is "backward".
113: (8)                             Indicates which direction of the forward/backward pair of transforms
114: (8)                             is scaled and with what normalization factor.
115: (8)                             .. versionadded:: 1.20.0
116: (12)                            The "backward", "forward" values were added.
117: (4)                               Returns
118: (4)                               -----
119: (4)                             out : complex ndarray
120: (8)                             The truncated or zero-padded input, transformed along the axis
121: (8)                             indicated by `axis`, or the last one if `axis` is not specified.
122: (4)                               Raises
123: (4)                               -----
124: (4)                             IndexError
125: (8)                             If `axis` is not a valid axis of `a`.
126: (4)                               See Also
127: (4)                               -----
128: (4)                             numpy.fft : for definition of the DFT and conventions used.
129: (4)                             ifft : The inverse of `fft`.
130: (4)                             fft2 : The two-dimensional FFT.
131: (4)                             fftn : The *n*-dimensional FFT.
132: (4)                             rfftn : The *n*-dimensional FFT of real input.
133: (4)                             fftfreq : Frequency bins for given FFT parameters.
134: (4)                               Notes
135: (4)                               -----
136: (4)                             FFT (Fast Fourier Transform) refers to a way the discrete Fourier
137: (4)                             Transform (DFT) can be calculated efficiently, by using symmetries in the
138: (4)                             calculated terms. The symmetry is highest when `n` is a power of 2, and
139: (4)                             the transform is therefore most efficient for these sizes.
140: (4)                             The DFT is defined, with the conventions used in this implementation, in
141: (4)                             the documentation for the `numpy.fft` module.
142: (4)                               References
143: (4)                               -----
144: (4)                             .. [CT] Cooley, James W., and John W. Tukey, 1965, "An algorithm for the
145: (12)                            machine calculation of complex Fourier series," *Math. Comput.*  

146: (12)                            19: 297-301.
147: (4)                               Examples
148: (4)                               -----
149: (4)                             >>> np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
150: (4)                             array([-2.33486982e-16+1.14423775e-17j,  8.00000000e+00-1.25557246e-15j,
151: (12)                            2.33486982e-16+2.33486982e-16j,  0.00000000e+00+1.22464680e-16j,
152: (11)                            -1.14423775e-17+2.33486982e-16j,  0.00000000e+00+5.20784380e-16j,
153: (12)                            1.14423775e-17+1.14423775e-17j,  0.00000000e+00+1.22464680e-16j])
154: (4)                             In this example, real input has an FFT which is Hermitian, i.e., symmetric
155: (4)                             in the real part and anti-symmetric in the imaginary part, as described in
156: (4)                             the `numpy.fft` documentation:
157: (4)                             >>> import matplotlib.pyplot as plt
158: (4)                             >>> t = np.arange(256)
159: (4)                             >>> sp = np.fft.fft(np.sin(t))
160: (4)                             >>> freq = np.fft.fftfreq(t.shape[-1])
161: (4)                             >>> plt.plot(freq, sp.real, freq, sp.imag)
162: (4)                             [<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D
object at 0x...>]
163: (4)                             >>> plt.show()
164: (4)                             """
165: (4)                             a = asarray(a)

```

```

166: (4)           if n is None:
167: (8)             n = a.shape[axis]
168: (4)             inv_norm = _get_forward_norm(n, norm)
169: (4)             output = _raw_fft(a, n, axis, False, True, inv_norm)
170: (4)             return output
171: (0) @array_function_dispatch(_fft_dispatcher)
172: (0) def ifft(a, n=None, axis=-1, norm=None):
173: (4)     """
174: (4)         Compute the one-dimensional inverse discrete Fourier Transform.
175: (4)         This function computes the inverse of the one-dimensional *n*-point
176: (4)         discrete Fourier transform computed by `fft`. In other words,
177: (4)         ``ifft(fft(a)) == a`` to within numerical accuracy.
178: (4)         For a general description of the algorithm and definitions,
179: (4)         see `numpy.fft`.
180: (4)         The input should be ordered in the same way as is returned by `fft`,
181: (4)         i.e.,
182: (4)         * ``a[0]`` should contain the zero frequency term,
183: (4)         * ``a[1:n//2]`` should contain the positive-frequency terms,
184: (4)         * ``a[n//2 + 1:]`` should contain the negative-frequency terms, in
185: (6)         increasing order starting from the most negative frequency.
186: (4)         For an even number of input points, ``A[n//2]`` represents the sum of
187: (4)         the values at the positive and negative Nyquist frequencies, as the two
188: (4)         are aliased together. See `numpy.fft` for details.
189: (4)     Parameters
190: (4)     -----
191: (4)     a : array_like
192: (8)       Input array, can be complex.
193: (4)     n : int, optional
194: (8)       Length of the transformed axis of the output.
195: (8)       If `n` is smaller than the length of the input, the input is cropped.
196: (8)       If it is larger, the input is padded with zeros. If `n` is not given,
197: (8)       the length of the input along the axis specified by `axis` is used.
198: (8)       See notes about padding issues.
199: (4)     axis : int, optional
200: (8)       Axis over which to compute the inverse DFT. If not given, the last
201: (8)       axis is used.
202: (4)     norm : {"backward", "ortho", "forward"}, optional
203: (8)       .. versionadded:: 1.10.0
204: (8)       Normalization mode (see `numpy.fft`). Default is "backward".
205: (8)       Indicates which direction of the forward/backward pair of transforms
206: (8)       is scaled and with what normalization factor.
207: (8)       .. versionadded:: 1.20.0
208: (12)      The "backward", "forward" values were added.
209: (4)     Returns
210: (4)     -----
211: (4)     out : complex ndarray
212: (8)       The truncated or zero-padded input, transformed along the axis
213: (8)       indicated by `axis`, or the last one if `axis` is not specified.
214: (4)     Raises
215: (4)     -----
216: (4)     IndexError
217: (8)       If `axis` is not a valid axis of `a`.
218: (4)     See Also
219: (4)     -----
220: (4)     numpy.fft : An introduction, with definitions and general explanations.
221: (4)     fft : The one-dimensional (forward) FFT, of which `ifft` is the inverse
222: (4)     ifft2 : The two-dimensional inverse FFT.
223: (4)     ifftn : The n-dimensional inverse FFT.
224: (4)     Notes
225: (4)     -----
226: (4)     If the input parameter `n` is larger than the size of the input, the input
227: (4)     is padded by appending zeros at the end. Even though this is the common
228: (4)     approach, it might lead to surprising results. If a different padding is
229: (4)     desired, it must be performed before calling `ifft`.
230: (4)     Examples
231: (4)     -----
232: (4)     >>> np.fft.ifft([0, 4, 0, 0])
233: (4)     array([ 1.+0.j,  0.+1.j, -1.+0.j,  0.-1.j]) # may vary
234: (4)     Create and plot a band-limited signal with random phases:

```

```

235: (4)          >>> import matplotlib.pyplot as plt
236: (4)          >>> t = np.arange(400)
237: (4)          >>> n = np.zeros((400,), dtype=complex)
238: (4)          >>> n[40:60] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20,)))
239: (4)          >>> s = np.fft.ifft(n)
240: (4)          >>> plt.plot(t, s.real, label='real')
241: (4)          [<matplotlib.lines.Line2D object at ...>]
242: (4)          >>> plt.plot(t, s.imag, '--', label='imaginary')
243: (4)          [<matplotlib.lines.Line2D object at ...>]
244: (4)          >>> plt.legend()
245: (4)          <matplotlib.legend.Legend object at ...>
246: (4)          >>> plt.show()
247: (4)          """
248: (4)          a = asarray(a)
249: (4)          if n is None:
250: (8)              n = a.shape[axis]
251: (4)          inv_norm = _get_backward_norm(n, norm)
252: (4)          output = _raw_fft(a, n, axis, False, False, inv_norm)
253: (4)          return output
254: (0)          @array_function_dispatch(_fft_dispatcher)
255: (0)          def rfft(a, n=None, axis=-1, norm=None):
256: (4)          """
257: (4)          Compute the one-dimensional discrete Fourier Transform for real input.
258: (4)          This function computes the one-dimensional *n*-point discrete Fourier
259: (4)          Transform (DFT) of a real-valued array by means of an efficient algorithm
260: (4)          called the Fast Fourier Transform (FFT).
261: (4)          Parameters
262: (4)          -----
263: (4)          a : array_like
264: (8)              Input array
265: (4)          n : int, optional
266: (8)              Number of points along transformation axis in the input to use.
267: (8)              If `n` is smaller than the length of the input, the input is cropped.
268: (8)              If it is larger, the input is padded with zeros. If `n` is not given,
269: (8)              the length of the input along the axis specified by `axis` is used.
270: (4)          axis : int, optional
271: (8)              Axis over which to compute the FFT. If not given, the last axis is
272: (8)              used.
273: (4)          norm : {"backward", "ortho", "forward"}, optional
274: (8)              .. versionadded:: 1.10.0
275: (8)              Normalization mode (see `numpy.fft`). Default is "backward".
276: (8)              Indicates which direction of the forward/backward pair of transforms
277: (8)              is scaled and with what normalization factor.
278: (8)              .. versionadded:: 1.20.0
279: (12)             The "backward", "forward" values were added.
280: (4)          Returns
281: (4)          -----
282: (4)          out : complex ndarray
283: (8)              The truncated or zero-padded input, transformed along the axis
284: (8)              indicated by `axis`, or the last one if `axis` is not specified.
285: (8)              If `n` is even, the length of the transformed axis is `` $(n/2)+1$ ``.
286: (8)              If `n` is odd, the length is `` $(n+1)/2$ ``.
287: (4)          Raises
288: (4)          -----
289: (4)          IndexError
290: (8)              If `axis` is not a valid axis of `a`.
291: (4)          See Also
292: (4)          -----
293: (4)          numpy.fft : For definition of the DFT and conventions used.
294: (4)          irfft : The inverse of `rfft`.
295: (4)          fft : The one-dimensional FFT of general (complex) input.
296: (4)          fftn : The *n*-dimensional FFT.
297: (4)          rfftn : The *n*-dimensional FFT of real input.
298: (4)          Notes
299: (4)          -----
300: (4)          When the DFT is computed for purely real input, the output is
301: (4)          Hermitian-symmetric, i.e. the negative frequency terms are just the
complex
302: (4)          conjugates of the corresponding positive-frequency terms, and the

```

```

303: (4) negative-frequency terms are therefore redundant. This function does not
304: (4) compute the negative frequency terms, and the length of the transformed
305: (4) axis of the output is therefore ``n//2 + 1``.
306: (4) When ``A = rfft(a)`` and fs is the sampling frequency, ``A[0]`` contains
307: (4) the zero-frequency term  $0*fs$ , which is real due to Hermitian symmetry.
308: (4) If `n` is even, ``A[-1]`` contains the term representing both positive
309: (4) and negative Nyquist frequency ( $+fs/2$  and  $-fs/2$ ), and must also be purely
310: (4) real. If `n` is odd, there is no term at  $fs/2$ ; ``A[-1]`` contains
311: (4) the largest positive frequency ( $fs/2*(n-1)/n$ ), and is complex in the
312: (4) general case.
313: (4) If the input `a` contains an imaginary part, it is silently discarded.
314: (4) Examples
315: (4) -----
316: (4) >>> np.fft.fft([0, 1, 0, 0])
317: (4) array([ 1.+0.j, 0.-1.j, -1.+0.j, 0.+1.j]) # may vary
318: (4) >>> np.fft.rfft([0, 1, 0, 0])
319: (4) array([ 1.+0.j, 0.-1.j, -1.+0.j]) # may vary
320: (4) Notice how the final element of the `fft` output is the complex conjugate
321: (4) of the second element, for real input. For `rfft`, this symmetry is
322: (4) exploited to compute only the non-negative frequency terms.
323: (4) """
324: (4) a = asarray(a)
325: (4) if n is None:
326: (8)     n = a.shape[axis]
327: (4)     inv_norm = _get_forward_norm(n, norm)
328: (4)     output = _raw_fft(a, n, axis, True, True, inv_norm)
329: (4)     return output
330: (0) @array_function_dispatch(_fft_dispatcher)
331: (0) def irfft(a, n=None, axis=-1, norm=None):
332: (4) """
333: (4)     Computes the inverse of `rfft`.
334: (4)     This function computes the inverse of the one-dimensional *n*-point
335: (4)     discrete Fourier Transform of real input computed by `rfft`.
336: (4)     In other words, ``irfft(rfft(a), len(a)) == a`` to within numerical
337: (4)     accuracy. (See Notes below for why ``len(a)`` is necessary here.)
338: (4)     The input is expected to be in the form returned by `rfft`, i.e. the
339: (4)     real zero-frequency term followed by the complex positive frequency terms
340: (4)     in order of increasing frequency. Since the discrete Fourier Transform of
341: (4)     real input is Hermitian-symmetric, the negative frequency terms are taken
342: (4)     to be the complex conjugates of the corresponding positive frequency
terms.
343: (4) Parameters
344: (4) -----
345: (4) a : array_like
346: (8)     The input array.
347: (4) n : int, optional
348: (8)     Length of the transformed axis of the output.
349: (8)     For `n` output points, ``n//2+1`` input points are necessary. If the
350: (8)     input is longer than this, it is cropped. If it is shorter than this,
351: (8)     it is padded with zeros. If `n` is not given, it is taken to be
352: (8)     ``2*(m-1)`` where ``m`` is the length of the input along the axis
353: (8)     specified by `axis`.
354: (4) axis : int, optional
355: (8)     Axis over which to compute the inverse FFT. If not given, the last
356: (8)     axis is used.
357: (4) norm : {"backward", "ortho", "forward"}, optional
358: (8)     .. versionadded:: 1.10.0
359: (8)     Normalization mode (see `numpy.fft`). Default is "backward".
360: (8)     Indicates which direction of the forward/backward pair of transforms
361: (8)     is scaled and with what normalization factor.
362: (8)     .. versionadded:: 1.20.0
363: (12)     The "backward", "forward" values were added.
364: (4) Returns
365: (4) -----
366: (4) out : ndarray
367: (8)     The truncated or zero-padded input, transformed along the axis
368: (8)     indicated by `axis`, or the last one if `axis` is not specified.
369: (8)     The length of the transformed axis is `n`, or, if `n` is not given,
370: (8)     ``2*(m-1)`` where ``m`` is the length of the transformed axis of the

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

371: (8)           input. To get an odd number of output points, `n` must be specified.
372: (4)           Raises
373: (4)           -----
374: (4)           IndexError
375: (8)           If `axis` is not a valid axis of `a`.
376: (4)           See Also
377: (4)           -----
378: (4)           numpy.fft : For definition of the DFT and conventions used.
379: (4)           rfft : The one-dimensional FFT of real input, of which `irfft` is inverse.
380: (4)           fft : The one-dimensional FFT.
381: (4)           irfft2 : The inverse of the two-dimensional FFT of real input.
382: (4)           irfftn : The inverse of the *n*-dimensional FFT of real input.
383: (4)           Notes
384: (4)           -----
385: (4)           Returns the real valued `n`-point inverse discrete Fourier transform
386: (4)           of `a`, where `a` contains the non-negative frequency terms of a
387: (4)           Hermitian-symmetric sequence. `n` is the length of the result, not the
388: (4)           input.
389: (4)           If you specify an `n` such that `a` must be zero-padded or truncated, the
390: (4)           extra/removed values will be added/removed at high frequencies. One can
391: (4)           thus resample a series to `m` points via Fourier interpolation by:
392: (4)           ``a_resamp = irfft(rfft(a), m)``.
393: (4)           The correct interpretation of the hermitian input depends on the length of
394: (4)           the original data, as given by `n`. This is because each input shape could
395: (4)           correspond to either an odd or even length signal. By default, `irfft`
396: (4)           assumes an even output length which puts the last entry at the Nyquist
397: (4)           frequency; aliasing with its symmetric counterpart. By Hermitian symmetry,
398: (4)           the value is thus treated as purely real. To avoid losing information, the
399: (4)           correct length of the real input **must** be given.
400: (4)           Examples
401: (4)           -----
402: (4)           >>> np.fft.ifft([1, -1j, -1, 1j])
403: (4)           array([0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j]) # may vary
404: (4)           >>> np.fft.irfft([1, -1j, -1])
405: (4)           array([0., 1., 0., 0.])
406: (4)           Notice how the last term in the input to the ordinary `ifft` is the
407: (4)           complex conjugate of the second term, and the output has zero imaginary
408: (4)           part everywhere. When calling `irfft`, the negative frequencies are not
409: (4)           specified, and the output array is purely real.
410: (4)           """
411: (4)           a = asarray(a)
412: (4)           if n is None:
413: (8)               n = (a.shape[axis] - 1) * 2
414: (4)           inv_norm = _get_backward_norm(n, norm)
415: (4)           output = _raw_fft(a, n, axis, True, False, inv_norm)
416: (4)           return output
417: (0)           @array_function_dispatch(_fft_dispatcher)
418: (0)           def hfft(a, n=None, axis=-1, norm=None):
419: (4)               """
420: (4)               Compute the FFT of a signal that has Hermitian symmetry, i.e., a real
421: (4)               spectrum.
422: (4)               Parameters
423: (4)               -----
424: (4)               a : array_like
425: (8)                   The input array.
426: (4)               n : int, optional
427: (8)                   Length of the transformed axis of the output. For `n` output
428: (8)                   points, ``n//2 + 1`` input points are necessary. If the input is
429: (8)                   longer than this, it is cropped. If it is shorter than this, it is
430: (8)                   padded with zeros. If `n` is not given, it is taken to be ``2*(m-1)``
431: (8)                   where ``m`` is the length of the input along the axis specified by
432: (8)                   `axis`.
433: (4)               axis : int, optional
434: (8)                   Axis over which to compute the FFT. If not given, the last
435: (8)                   axis is used.
436: (4)               norm : {"backward", "ortho", "forward"}, optional
437: (8)                   .. versionadded:: 1.10.0
438: (8)                   Normalization mode (see `numpy.fft`). Default is "backward".
439: (8)                   Indicates which direction of the forward/backward pair of transforms

```

```

440: (8)           is scaled and with what normalization factor.
441: (8)           .. versionadded:: 1.20.0
442: (12)          The "backward", "forward" values were added.
443: (4)          Returns
444: (4)
445: (4)          out : ndarray
446: (8)          The truncated or zero-padded input, transformed along the axis
447: (8)          indicated by `axis`, or the last one if `axis` is not specified.
448: (8)          The length of the transformed axis is `n`, or, if `n` is not given,
449: (8)          ``2*m - 2`` where ``m`` is the length of the transformed axis of
450: (8)          the input. To get an odd number of output points, `n` must be
451: (8)          specified, for instance as ``2*m - 1`` in the typical case,
452: (4)          Raises
453: (4)
454: (4)          IndexError
455: (8)          If `axis` is not a valid axis of `a`.
456: (4)          See also
457: (4)
458: (4)          rfft : Compute the one-dimensional FFT for real input.
459: (4)          ihfft : The inverse of `hfft`.
460: (4)          Notes
461: (4)
462: (4)          `hfft`/`ihfft` are a pair analogous to `rfft`/`irfft`, but for the
463: (4)          opposite case: here the signal has Hermitian symmetry in the time
464: (4)          domain and is real in the frequency domain. So here it's `hfft` for
465: (4)          which you must supply the length of the result if it is to be odd.
466: (4)          * even: ``ihfft(hfft(a, 2*len(a) - 2)) == a``, within roundoff error,
467: (4)          * odd: ``ihfft(hfft(a, 2*len(a) - 1)) == a``, within roundoff error.
468: (4)          The correct interpretation of the hermitian input depends on the length of
469: (4)          the original data, as given by `n`. This is because each input shape could
470: (4)          correspond to either an odd or even length signal. By default, `hfft`
471: (4)          assumes an even output length which puts the last entry at the Nyquist
472: (4)          frequency; aliasing with its symmetric counterpart. By Hermitian symmetry,
473: (4)          the value is thus treated as purely real. To avoid losing information, the
474: (4)          shape of the full signal **must** be given.
475: (4)          Examples
476: (4)
477: (4)          >>> signal = np.array([1, 2, 3, 4, 3, 2])
478: (4)          >>> np.fft.fft(signal)
479: (4)          array([15.+0.j, -4.+0.j,  0.+0.j, -1.-0.j,  0.+0.j, -4.+0.j]) # may
vary
480: (4)          >>> np.fft.hfft(signal[:4]) # Input first half of signal
481: (4)          array([15., -4.,  0., -1.,  0., -4.])
482: (4)          >>> np.fft.hfft(signal, 6) # Input entire signal and truncate
483: (4)          array([15., -4.,  0., -1.,  0., -4.])
484: (4)          >>> signal = np.array([[1, 1.j], [-1.j, 2]])
485: (4)          >>> np.conj(signal.T) - signal # check Hermitian symmetry
486: (4)          array([[ 0.-0.j, -0.+0.j], # may vary
487: (11)             [ 0.+0.j,  0.-0.j]])
488: (4)          >>> freq_spectrum = np.fft.hfft(signal)
489: (4)          >>> freq_spectrum
490: (4)          array([[ 1.,  1.],
491: (11)             [ 2., -2.]])
492: (4)
493: (4)          """
494: (4)          a = asarray(a)
495: (8)          if n is None:
496: (4)              n = (a.shape[axis] - 1) * 2
497: (4)              new_norm = _swap_direction(norm)
498: (4)              output = irfft(conjugate(a), n, axis, norm=new_norm)
499: (0)              return output
500: (0)          @array_function_dispatch(_fft_dispatcher)
501: (4)          def ihfft(a, n=None, axis=-1, norm=None):
502: (4)              """
503: (4)              Compute the inverse FFT of a signal that has Hermitian symmetry.
504: (4)              Parameters
505: (4)
506: (8)              a : array_like
507: (4)                  Input array.
n : int, optional

```

```

508: (8) Length of the inverse FFT, the number of points along
509: (8) transformation axis in the input to use. If `n` is smaller than
510: (8) the length of the input, the input is cropped. If it is larger,
511: (8) the input is padded with zeros. If `n` is not given, the length of
512: (8) the input along the axis specified by `axis` is used.
513: (4) axis : int, optional
514: (8) Axis over which to compute the inverse FFT. If not given, the last
515: (8) axis is used.
516: (4) norm : {"backward", "ortho", "forward"}, optional
517: (8) .. versionadded:: 1.10.0
518: (8) Normalization mode (see `numpy.fft`). Default is "backward".
519: (8) Indicates which direction of the forward/backward pair of transforms
520: (8) is scaled and with what normalization factor.
521: (8) .. versionadded:: 1.20.0
522: (12) The "backward", "forward" values were added.
523: (4) Returns
524: (4) -----
525: (4) out : complex ndarray
526: (8) The truncated or zero-padded input, transformed along the axis
527: (8) indicated by `axis`, or the last one if `axis` is not specified.
528: (8) The length of the transformed axis is ``n//2 + 1``.
529: (4) See also
530: (4) -----
531: (4) hfft, irfft
532: (4) Notes
533: (4) -----
534: (4) `hfft`/`ihfft` are a pair analogous to `rfft`/`irfft`, but for the
535: (4) opposite case: here the signal has Hermitian symmetry in the time
536: (4) domain and is real in the frequency domain. So here it's `hfft` for
537: (4) which you must supply the length of the result if it is to be odd:
538: (4) * even: ``ihfft(hfft(a, 2*len(a) - 2)) == a``, within roundoff error,
539: (4) * odd: ``ihfft(hfft(a, 2*len(a) - 1)) == a``, within roundoff error.
540: (4) Examples
541: (4) -----
542: (4) >>> spectrum = np.array([ 15, -4, 0, -1, 0, -4])
543: (4) >>> np.fft.ifft(spectrum)
544: (4) array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 3.+0.j, 2.+0.j]) # may vary
545: (4) >>> np.fft.ihfft(spectrum)
546: (4) array([ 1.-0.j, 2.-0.j, 3.-0.j, 4.-0.j]) # may vary
547: (4) """
548: (4) a = asarray(a)
549: (4) if n is None:
550: (8)     n = a.shape[axis]
551: (4) new_norm = _swap_direction(norm)
552: (4) output = conjugate(rfft(a, n, axis, norm=new_norm))
553: (4) return output
554: (0) def _cook_nd_args(a, s=None, axes=None, invreal=0):
555: (4)     if s is None:
556: (8)         shapeless = 1
557: (8)         if axes is None:
558: (12)             s = list(a.shape)
559: (8)         else:
560: (12)             s = take(a.shape, axes)
561: (4)     else:
562: (8)         shapeless = 0
563: (4)     s = list(s)
564: (4)     if axes is None:
565: (8)         axes = list(range(-len(s), 0))
566: (4)     if len(s) != len(axes):
567: (8)         raise ValueError("Shape and axes have different lengths.")
568: (4)     if invreal and shapeless:
569: (8)         s[-1] = (a.shape[axes[-1]] - 1) * 2
570: (4)     return s, axes
571: (0) def _raw_fftn(a, s=None, axes=None, function=fft, norm=None):
572: (4)     a = asarray(a)
573: (4)     s, axes = _cook_nd_args(a, s, axes)
574: (4)     itl = list(range(len(axes)))
575: (4)     itl.reverse()
576: (4)     for ii in itl:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

577: (8)           a = function(a, n=s[ii], axis=axes[ii], norm=norm)
578: (4)           return a
579: (0)           def _fftn_dispatcher(a, s=None, axes=None, norm=None):
580: (4)             return (a,)
581: (0)           @array_function_dispatch(_fftn_dispatcher)
582: (0)           def fftn(a, s=None, axes=None, norm=None):
583: (4)             """
584: (4)               Compute the N-dimensional discrete Fourier Transform.
585: (4)               This function computes the *N*-dimensional discrete Fourier Transform over
586: (4)               any number of axes in an *M*-dimensional array by means of the Fast
587: (4)               Fourier
588: (4)               Transform (FFT).
589: (4)               Parameters
590: (4)               -----
591: (8)                 a : array_like
592: (4)                   Input array, can be complex.
593: (8)                 s : sequence of ints, optional
594: (8)                   Shape (length of each transformed axis) of the output
595: (8)                   (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.).
596: (8)                   This corresponds to ``n`` for ``fft(x, n)``.
597: (8)                   Along any axis, if the given shape is smaller than that of the input,
598: (8)                   the input is cropped. If it is larger, the input is padded with
599: (8)                   zeros.
600: (4)                 if `s` is not given, the shape of the input along the axes specified
601: (4)                   by `axes` is used.
602: (4)                 axes : sequence of ints, optional
603: (8)                   Axes over which to compute the FFT. If not given, the last ``len(s)``
604: (8)                   axes are used, or all axes if `s` is also not specified.
605: (8)                   Repeated indices in `axes` means that the transform over that axis is
606: (4)                   performed multiple times.
607: (8)                 norm : {"backward", "ortho", "forward"}, optional
608: (8)                   .. versionadded:: 1.10.0
609: (8)                   Normalization mode (see `numpy.fft`). Default is "backward".
610: (8)                   Indicates which direction of the forward/backward pair of transforms
611: (12)                  is scaled and with what normalization factor.
612: (4)                   .. versionadded:: 1.20.0
613: (4)                   The "backward", "forward" values were added.
614: (4)               Returns
615: (4)               -----
616: (4)                 out : complex ndarray
617: (8)                   The truncated or zero-padded input, transformed along the axes
618: (8)                   indicated by `axes`, or by a combination of `s` and `a`,
619: (8)                   as explained in the parameters section above.
620: (4)               Raises
621: (4)               -----
622: (4)                 ValueError
623: (8)                   If `s` and `axes` have different length.
624: (4)                 IndexError
625: (8)                   If an element of `axes` is larger than than the number of axes of `a`.
626: (4)               See Also
627: (4)               -----
628: (4)                 numpy.fft : Overall view of discrete Fourier transforms, with definitions
629: (8)                   and conventions used.
630: (4)                 ifftn : The inverse of `fftn`, the inverse *n*-dimensional FFT.
631: (4)                 fft : The one-dimensional FFT, with definitions and conventions used.
632: (4)                 rfftn : The *n*-dimensional FFT of real input.
633: (4)                 fft2 : The two-dimensional FFT.
634: (4)                 fftshift : Shifts zero-frequency terms to centre of array
635: (4)               Notes
636: (4)               -----
637: (4)                 The output, analogously to `fft`, contains the term for zero frequency in
638: (4)                 the low-order corner of all axes, the positive frequency terms in the
639: (4)                 first half of all axes, the term for the Nyquist frequency in the middle
640: (4)                 of all axes and the negative frequency terms in the second half of all
641: (4)                 axes, in order of decreasingly negative frequency.
642: (4)                 See `numpy.fft` for details, definitions and conventions used.
643: (4)               Examples
644: (4)               -----
645: (4)                 >>> a = np.mgrid[:3, :3, :3][0]

```

```

644: (4)
645: (4)
646: (12)
647: (12)
648: (11)
649: (12)
650: (12)
651: (11)
652: (12)
653: (12)
654: (4)
655: (4)
656: (12)
657: (11)
658: (12)
659: (4)
660: (4)
661: (4)
662: (4)
663: (4)
664: (4)
665: (4)
666: (4)
667: (4)
668: (4)
669: (0)
670: (0)
671: (4)
672: (4)
673: (4)
674: (4)
675: (4)
676: (4)
677: (4)
`numpy.fft`.

678: (4)
679: (4)
680: (4)
681: (4)
682: (4)
683: (4)
684: (4)
685: (4)
686: (4)
687: (8)
688: (4)
689: (8)
690: (8)
691: (8)
692: (8)
693: (8)
zeros.

694: (8)
695: (8)
696: (4)
697: (8)
``len(s)``
698: (8)
699: (8)
700: (8)
701: (4)
702: (8)
703: (8)
704: (8)
705: (8)
706: (8)
707: (12)
708: (4)
709: (4)

    >>> np.fft.fftn(a, axes=(1, 2))
    array([[[[ 0.+0.j,  0.+0.j,  0.+0.j], # may vary
              [ 0.+0.j,  0.+0.j,  0.+0.j],
              [ 0.+0.j,  0.+0.j,  0.+0.j]],
             [[ 9.+0.j,  0.+0.j,  0.+0.j],
              [ 0.+0.j,  0.+0.j,  0.+0.j],
              [ 0.+0.j,  0.+0.j,  0.+0.j]],
             [[18.+0.j,  0.+0.j,  0.+0.j],
              [ 0.+0.j,  0.+0.j,  0.+0.j],
              [ 0.+0.j,  0.+0.j,  0.+0.j]]])
    >>> np.fft.fftn(a, (2, 2), axes=(0, 1))
    array([[[[ 2.+0.j,  2.+0.j,  2.+0.j], # may vary
              [ 0.+0.j,  0.+0.j,  0.+0.j],
              [-2.+0.j, -2.+0.j, -2.+0.j],
              [ 0.+0.j,  0.+0.j,  0.+0.j]]])
    >>> import matplotlib.pyplot as plt
    >>> [X, Y] = np.meshgrid(2 * np.pi * np.arange(200) / 12,
                           ...                                2 * np.pi * np.arange(200) / 34)
    >>> S = np.sin(X) + np.cos(Y) + np.random.uniform(0, 1, X.shape)
    >>> FS = np.fft.fftn(S)
    >>> plt.imshow(np.log(np.abs(np.fft.fftshift(FS))**2))
    <matplotlib.image.AxesImage object at 0x...>
    >>> plt.show()
    """
    return _raw_fftn(a, s, axes, fft, norm)
@array_function_dispatch(_fftn_dispatcher)
def ifftn(a, s=None, axes=None, norm=None):
    """
    Compute the N-dimensional inverse discrete Fourier Transform.
    This function computes the inverse of the N-dimensional discrete
    Fourier Transform over any number of axes in an M-dimensional array by
    means of the Fast Fourier Transform (FFT). In other words,
    ``ifftn(fftn(a)) == a`` to within numerical accuracy.
    For a description of the definitions and conventions used, see
    `numpy.fft`.

    The input, analogously to `ifft`, should be ordered in the same way as is
    returned by `fftn`, i.e. it should have the term for zero frequency
    in all axes in the low-order corner, the positive frequency terms in the
    first half of all axes, the term for the Nyquist frequency in the middle
    of all axes and the negative frequency terms in the second half of all
    axes, in order of decreasingly negative frequency.

    Parameters
    -----
    a : array_like
        Input array, can be complex.
    s : sequence of ints, optional
        Shape (length of each transformed axis) of the output
        (``s[0]`` refers to axis 0, ``s[1]`` to axis 1, etc.).
        This corresponds to ``n`` for ``ifft(x, n)``.
        Along any axis, if the given shape is smaller than that of the input,
        the input is cropped. If it is larger, the input is padded with
        zeros.
    axes : sequence of ints, optional
        Axes over which to compute the IFFT. If not given, the last
        axes are used, or all axes if `s` is also not specified.
        Repeated indices in `axes` means that the inverse transform over that
        axis is performed multiple times.
    norm : {"backward", "ortho", "forward"}, optional
        .. versionadded:: 1.10.0
        Normalization mode (see `numpy.fft`). Default is "backward".
        Indicates which direction of the forward/backward pair of transforms
        is scaled and with what normalization factor.
        .. versionadded:: 1.20.0
        The "backward", "forward" values were added.

    Returns
    -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

710: (4)
711: (8)         out : complex ndarray
712: (8)             The truncated or zero-padded input, transformed along the axes
713: (8)             indicated by `axes`, or by a combination of `s` or `a`,
714: (4)             as explained in the parameters section above.
715: (4)         Raises
716: (4)         -----
717: (8)             ValueError
718: (4)                 If `s` and `axes` have different length.
719: (8)             IndexError
720: (4)                 If an element of `axes` is larger than than the number of axes of `a`.
721: (4)         See Also
722: (4)         -----
723: (9)             numpy.fft : Overall view of discrete Fourier transforms, with definitions
724: (4)             and conventions used.
725: (4)             fftn : The forward *n*-dimensional FFT, of which `ifftn` is the inverse.
726: (4)             ifft : The one-dimensional inverse FFT.
727: (4)             ifft2 : The two-dimensional inverse FFT.
728: (8)             ifftshift : Undoes `fftshift`, shifts zero-frequency terms to beginning
729: (4)             of array.
730: (4)         Notes
731: (4)         -----
732: (4)             See `numpy.fft` for definitions and conventions used.
733: (4)             Zero-padding, analogously with `ifft`, is performed by appending zeros to
734: (4)             the input along the specified dimension. Although this is the common
735: (4)             approach, it might lead to surprising results. If another form of zero
736: (4)             padding is desired, it must be performed before `ifftn` is called.
737: (4)         Examples
738: (4)         -----
739: (4)             >>> a = np.eye(4)
740: (4)             >>> np.fft.ifftn(np.fft.fftn(a, axes=(0,)), axes=(1,))
741: (11)            array([[1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j], # may vary
742: (11)                         [0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
743: (11)                         [0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
744: (4)                         [0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j]])
745: (4)             Create and plot an image with band-limited frequency content:
746: (4)             >>> import matplotlib.pyplot as plt
747: (4)             >>> n = np.zeros((200,200), dtype=complex)
748: (4)             >>> n[60:80, 20:40] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20, 20)))
749: (4)             >>> im = np.fft.ifftn(n).real
750: (4)             >>> plt.imshow(im)
751: (4)             <matplotlib.image.AxesImage object at 0x...>
752: (4)             >>> plt.show()
753: (4)             """
754: (0)             return _raw_fftn(a, s, axes, ifft, norm)
755: (0)         @array_function_dispatch(_fftn_dispatcher)
756: (4)         def fft2(a, s=None, axes=(-2, -1), norm=None):
757: (4)             """
758: (4)                 Compute the 2-dimensional discrete Fourier Transform.
759: (4)                 This function computes the *n*-dimensional discrete Fourier Transform
760: (4)                 over any axes in an *M*-dimensional array by means of the
761: (4)                 Fast Fourier Transform (FFT). By default, the transform is computed over
762: (4)                 the last two axes of the input array, i.e., a 2-dimensional FFT.
763: (4)             Parameters
764: (4)             -----
765: (8)                 a : array_like
766: (4)                     Input array, can be complex
767: (8)                 s : sequence of ints, optional
768: (8)                     Shape (length of each transformed axis) of the output
769: (8)                     (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.).
770: (8)                     This corresponds to `n` for `fft(x, n)`.
771: (8)                     Along each axis, if the given shape is smaller than that of the input,
772: (8)                     the input is cropped. If it is larger, the input is padded with
773: (8)                     zeros.
774: (8)                     if `s` is not given, the shape of the input along the axes specified
775: (8)                     by `axes` is used.
776: (4)                 axes : sequence of ints, optional
777: (8)                     Axes over which to compute the FFT. If not given, the last two
778: (8)                     axes are used. A repeated index in `axes` means the transform over
779: (8)                     that axis is performed multiple times. A one-element sequence means

```

```

778: (8)           that a one-dimensional FFT is performed.
779: (4)           norm : {"backward", "ortho", "forward"}, optional
780: (8)             .. versionadded:: 1.10.0
781: (8)             Normalization mode (see `numpy.fft`). Default is "backward".
782: (8)             Indicates which direction of the forward/backward pair of transforms
783: (8)             is scaled and with what normalization factor.
784: (8)             .. versionadded:: 1.20.0
785: (12)            The "backward", "forward" values were added.
786: (4)           Returns
787: (4)           -----
788: (4)           out : complex ndarray
789: (8)             The truncated or zero-padded input, transformed along the axes
790: (8)             indicated by `axes`, or the last two axes if `axes` is not given.
791: (4)           Raises
792: (4)           -----
793: (4)           ValueError
794: (8)             If `s` and `axes` have different length, or `axes` not given and
795: (8)             ``len(s) != 2``.
796: (4)           IndexError
797: (8)             If an element of `axes` is larger than than the number of axes of `a`.
798: (4)           See Also
799: (4)           -----
800: (4)           numpy.fft : Overall view of discrete Fourier transforms, with definitions
801: (9)             and conventions used.
802: (4)           ifft2 : The inverse two-dimensional FFT.
803: (4)           fft : The one-dimensional FFT.
804: (4)           fftn : The *n*-dimensional FFT.
805: (4)           fftshift : Shifts zero-frequency terms to the center of the array.
806: (8)             For two-dimensional input, swaps first and third quadrants, and second
807: (8)             and fourth quadrants.
808: (4)           Notes
809: (4)           -----
810: (4)             `fft2` is just `fftn` with a different default for `axes`.
811: (4)             The output, analogously to `fft`, contains the term for zero frequency in
812: (4)             the low-order corner of the transformed axes, the positive frequency terms
813: (4)             in the first half of these axes, the term for the Nyquist frequency in the
814: (4)             middle of the axes and the negative frequency terms in the second half of
815: (4)             the axes, in order of decreasingly negative frequency.
816: (4)             See `fftn` for details and a plotting example, and `numpy.fft` for
817: (4)             definitions and conventions used.
818: (4)           Examples
819: (4)           -----
820: (4)           >>> a = np.mgrid[:5, :5][0]
821: (4)           >>> np.fft.fft2(a)
822: (4)           array([[ 50. +0.j      ,  0. +0.j      ,  0. +0.j      , # may
vary
823: (14)             0. +0.j      ,  0. +0.j      ],,
824: (11)             [-12.5+17.20477401j,  0. +0.j      ,  0. +0.j      ],
825: (14)             0. +0.j      ,  0. +0.j      ],,
826: (11)             [-12.5 +4.0614962j ,  0. +0.j      ,  0. +0.j      ],
827: (14)             0. +0.j      ,  0. +0.j      ],,
828: (11)             [-12.5 -4.0614962j ,  0. +0.j      ,  0. +0.j      ],
829: (14)             0. +0.j      ,  0. +0.j      ],,
830: (11)             [-12.5-17.20477401j,  0. +0.j      ,  0. +0.j      ],
831: (14)             0. +0.j      ,  0. +0.j      ]])
832: (4)           """
833: (4)             return _raw_fftn(a, s, axes, fft, norm)
834: (0)           @array_function_dispatch(_fftn_dispatcher)
835: (0)           def ifft2(a, s=None, axes=(-2, -1), norm=None):
836: (4)             """
837: (4)               Compute the 2-dimensional inverse discrete Fourier Transform.
838: (4)               This function computes the inverse of the 2-dimensional discrete Fourier
839: (4)               Transform over any number of axes in an M-dimensional array by means of
840: (4)               the Fast Fourier Transform (FFT). In other words, ``ifft2(fft2(a)) == a``
841: (4)               to within numerical accuracy. By default, the inverse transform is
842: (4)               computed over the last two axes of the input array.
843: (4)               The input, analogously to `ifft`, should be ordered in the same way as is
844: (4)               returned by `fft2`, i.e. it should have the term for zero frequency
845: (4)               in the low-order corner of the two axes, the positive frequency terms in

```

```

846: (4)          the first half of these axes, the term for the Nyquist frequency in the
847: (4)          middle of the axes and the negative frequency terms in the second half of
848: (4)          both axes, in order of decreasingly negative frequency.
849: (4)          Parameters
850: (4)          -----
851: (4)          a : array_like
852: (8)          Input array, can be complex.
853: (4)          s : sequence of ints, optional
854: (8)          Shape (length of each axis) of the output (` `s[0]` ` refers to axis 0,
855: (8)          ` `s[1]` ` to axis 1, etc.). This corresponds to ` `n` for ``ifft(x,
n)` `.
856: (8)          Along each axis, if the given shape is smaller than that of the input,
857: (8)          the input is cropped. If it is larger, the input is padded with
zeros.
858: (8)          if ` `s` is not given, the shape of the input along the axes specified
859: (8)          by ` `axes` is used. See notes for issue on ` `ifft` zero padding.
860: (4)          axes : sequence of ints, optional
861: (8)          Axes over which to compute the FFT. If not given, the last two
862: (8)          axes are used. A repeated index in ` `axes` means the transform over
863: (8)          that axis is performed multiple times. A one-element sequence means
864: (8)          that a one-dimensional FFT is performed.
865: (4)          norm : {"backward", "ortho", "forward"}, optional
866: (8)          .. versionadded:: 1.10.0
867: (8)          Normalization mode (see ` `numpy.fft` ). Default is "backward".
868: (8)          Indicates which direction of the forward/backward pair of transforms
869: (8)          is scaled and with what normalization factor.
870: (8)          .. versionadded:: 1.20.0
871: (12)         The "backward", "forward" values were added.
872: (4)          Returns
873: (4)          -----
874: (4)          out : complex ndarray
875: (8)          The truncated or zero-padded input, transformed along the axes
876: (8)          indicated by ` `axes` , or the last two axes if ` `axes` is not given.
877: (4)          Raises
878: (4)          -----
879: (4)          ValueError
880: (8)          If ` `s` and ` `axes` have different length, or ` `axes` not given and
881: (8)          ` `len(s) != 2` ` .
882: (4)          IndexError
883: (8)          If an element of ` `axes` is larger than than the number of axes of ` `a` .
884: (4)          See Also
885: (4)          -----
886: (4)          numpy.fft : Overall view of discrete Fourier transforms, with definitions
887: (9)          and conventions used.
888: (4)          fft2 : The forward 2-dimensional FFT, of which ` `ifft2` is the inverse.
889: (4)          ifftn : The inverse of the *n*-dimensional FFT.
890: (4)          fft : The one-dimensional FFT.
891: (4)          ifft : The one-dimensional inverse FFT.
892: (4)          Notes
893: (4)          -----
894: (4)          ` `ifft2` is just ` `ifftn` with a different default for ` `axes` .
895: (4)          See ` `ifftn` for details and a plotting example, and ` `numpy.fft` for
896: (4)          definition and conventions used.
897: (4)          Zero-padding, analogously with ` `ifft` , is performed by appending zeros to
898: (4)          the input along the specified dimension. Although this is the common
899: (4)          approach, it might lead to surprising results. If another form of zero
900: (4)          padding is desired, it must be performed before ` `ifft2` is called.
901: (4)          Examples
902: (4)          -----
903: (4)          >>> a = 4 * np.eye(4)
904: (4)          >>> np.fft.ifft2(a)
905: (4)          array([[1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j], # may vary
906: (11)             [0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j],
907: (11)             [0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
908: (11)             [0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j]])
909: (4)          """
910: (4)          return _raw_fftnd(a, s, axes, ifft, norm)
911: (0)          @array_function_dispatch(_fftn_dispatcher)
912: (0)          def rfftn(a, s=None, axes=None, norm=None):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

913: (4)
914: (4)      """
915: (4)      Compute the N-dimensional discrete Fourier Transform for real input.
916: (4)      This function computes the N-dimensional discrete Fourier Transform over
917: (4)      any number of axes in an M-dimensional real array by means of the Fast
918: (4)      Fourier Transform (FFT). By default, all axes are transformed, with the
919: (4)      real transform performed over the last axis, while the remaining
920: (4)      transforms are complex.
921: (4)
922: (4)      Parameters
923: (8)      -----
924: (4)      a : array_like
925: (8)          Input array, taken to be real.
926: (8)      s : sequence of ints, optional
927: (8)          Shape (length along each transformed axis) to use from the input.
928: (8)          (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.).
929: (8)          The final element of `s` corresponds to `n` for ``rfft(x, n)``,
930: (8)          while for the remaining axes, it corresponds to `n` for ``fft(x, n)``.
931: (8)          Along any axis, if the given shape is smaller than that of the input,
932: (8)          the input is cropped. If it is larger, the input is padded with
933: (4)          zeros.
934: (8)      if `s` is not given, the shape of the input along the axes specified
935: (8)          by `axes` is used.
936: (4)      axes : sequence of ints, optional
937: (8)          Axes over which to compute the FFT. If not given, the last ``len(s)``
938: (8)          axes are used, or all axes if `s` is also not specified.
939: (8)      norm : {"backward", "ortho", "forward"}, optional
940: (8)          .. versionadded:: 1.10.0
941: (8)          Normalization mode (see `numpy.fft`). Default is "backward".
942: (12)          Indicates which direction of the forward/backward pair of transforms
943: (4)          is scaled and with what normalization factor.
944: (4)          .. versionadded:: 1.20.0
945: (4)          The "backward", "forward" values were added.
946: (4)      Returns
947: (4)      -----
948: (4)      out : complex ndarray
949: (8)          The truncated or zero-padded input, transformed along the axes
950: (8)          indicated by `axes`, or by a combination of `s` and `a`,
951: (8)          as explained in the parameters section above.
952: (8)          The length of the last axis transformed will be ``s[-1]//2+1```,
953: (8)          while the remaining transformed axes will have lengths according to
954: (8)          `s`, or unchanged from the input.
955: (4)      Raises
956: (4)      -----
957: (4)      ValueError
958: (8)          If `s` and `axes` have different length.
959: (4)      IndexError
960: (8)          If an element of `axes` is larger than than the number of axes of `a`.
961: (4)      See Also
962: (4)      -----
963: (4)      irfftn : The inverse of `rfftn`, i.e. the inverse of the n-dimensional FFT
964: (4)          of real input.
965: (4)      fft : The one-dimensional FFT, with definitions and conventions used.
966: (4)      rfft : The one-dimensional FFT of real input.
967: (4)      fftn : The n-dimensional FFT.
968: (4)      rfft2 : The two-dimensional FFT of real input.
969: (4)      Notes
970: (4)      -----
971: (4)      The transform for real input is performed over the last transformation
972: (4)          axis, as by `rfft`, then the transform over the remaining axes is
973: (4)          performed as by `fftn`. The order of the output is as for `rfft` for the
974: (4)          final transformation axis, and as for `fftn` for the remaining
975: (4)          transformation axes.
976: (4)      See `fft` for details, definitions and conventions used.
977: (4)      Examples
978: (4)      -----
979: (12)      >>> a = np.ones((2, 2, 2))
980: (11)      >>> np.fft.rfftn(a)

```

```

981: (12)           [0.+0.j,  0.+0.j]]])
982: (4)             >>> np.fft.rfftn(a, axes=(2, 0))
983: (4)             array([[ [4.+0.j,  0.+0.j], # may vary
984: (12)                 [4.+0.j,  0.+0.j]],
985: (11)                 [[0.+0.j,  0.+0.j],
986: (12)                   [0.+0.j,  0.+0.j]]])
987: (4)             """
988: (4)             a = asarray(a)
989: (4)             s, axes = _cook_nd_args(a, s, axes)
990: (4)             a = rfft(a, s[-1], axes[-1], norm)
991: (4)             for ii in range(len(axes)-1):
992: (8)                 a = fft(a, s[ii], axes[ii], norm)
993: (4)             return a
994: (0) @array_function_dispatch(_fftn_dispatcher)
995: (0) def rfft2(a, s=None, axes=(-2, -1), norm=None):
996: (4)             """
997: (4)             Compute the 2-dimensional FFT of a real array.
998: (4)             Parameters
999: (4)             -----
1000: (4)             a : array
1001: (8)                 Input array, taken to be real.
1002: (4)             s : sequence of ints, optional
1003: (8)                 Shape of the FFT.
1004: (4)             axes : sequence of ints, optional
1005: (8)                 Axes over which to compute the FFT.
1006: (4)             norm : {"backward", "ortho", "forward"}, optional
1007: (8)                 .. versionadded:: 1.10.0
1008: (8)                 Normalization mode (see `numpy.fft`). Default is "backward".
1009: (8)                 Indicates which direction of the forward/backward pair of transforms
1010: (8)                 is scaled and with what normalization factor.
1011: (8)                 .. versionadded:: 1.20.0
1012: (12)                 The "backward", "forward" values were added.
1013: (4)             Returns
1014: (4)             -----
1015: (4)             out : ndarray
1016: (8)                 The result of the real 2-D FFT.
1017: (4)             See Also
1018: (4)             -----
1019: (4)             rfftn : Compute the N-dimensional discrete Fourier Transform for real
1020: (12)               input.
1021: (4)             Notes
1022: (4)             -----
1023: (4)             This is really just `rfftn` with different default behavior.
1024: (4)             For more details see `rfftn`.
1025: (4)             Examples
1026: (4)             -----
1027: (4)             >>> a = np.mgrid[:5, :5][0]
1028: (4)             >>> np.fft.rfft2(a)
1029: (4)             array([[ 50. +0.j      ,  0. +0.j      ,  0. +0.j      ],
1030: (11)                [-12.5+17.20477401j,  0. +0.j      ,  0. +0.j      ],
1031: (11)                [-12.5 +4.0614962j ,  0. +0.j      ,  0. +0.j      ],
1032: (11)                [-12.5 -4.0614962j ,  0. +0.j      ,  0. +0.j      ],
1033: (11)                [-12.5-17.20477401j,  0. +0.j      ,  0. +0.j      ]])
1034: (4)             """
1035: (4)             return rfftn(a, s, axes, norm)
1036: (0) @array_function_dispatch(_fftn_dispatcher)
1037: (0) def irfftn(a, s=None, axes=None, norm=None):
1038: (4)             """
1039: (4)             Computes the inverse of `rfftn`.
1040: (4)             This function computes the inverse of the N-dimensional discrete
1041: (4)             Fourier Transform for real input over any number of axes in an
1042: (4)             M-dimensional array by means of the Fast Fourier Transform (FFT). In
1043: (4)             other words, ``irfftn(rfftn(a), a.shape) == a`` to within numerical
1044: (4)             accuracy. (The ``a.shape`` is necessary like ``len(a)`` is for `irfft`,
1045: (4)             and for the same reason.)
1046: (4)             The input should be ordered in the same way as is returned by `rfftn`,
1047: (4)             i.e. as for `irfft` for the final transformation axis, and as for `ifftn`
1048: (4)             along all the other axes.
1049: (4)             Parameters

```

```

1050: (4)
1051: (4)
1052: (8)
1053: (4)
1054: (8)
1055: (8)
1056: (8)
1057: (8)
1058: (8)
1059: (8)
1060: (8)
1061: (8)
1062: (8)
1063: (4)
1064: (8)
1065: (8)
1066: (8)
1067: (8)
1068: (4)
1069: (8)
1070: (8)
1071: (8)
1072: (8)
1073: (8)
1074: (12)
1075: (4)
1076: (4)
1077: (4)
1078: (8)
1079: (8)
1080: (8)
1081: (8)
1082: (8)
the
1083: (8)
length
1084: (8)
1085: (8)
1086: (8)
1087: (4)
1088: (4)
1089: (4)
1090: (8)
1091: (4)
1092: (8)
1093: (4)
1094: (4)
1095: (4)
1096: (12)
1097: (4)
1098: (4)
1099: (4)
1100: (4)
1101: (4)
1102: (4)
1103: (4)
1104: (4)
1105: (4)
1106: (4)
1107: (4)
1108: (4)
1109: (4)
1110: (4)
1111: (4)
1112: (4)
1113: (4)
1114: (4)
1115: (4)
1116: (4)

      -----
      a : array_like
          Input array.
      s : sequence of ints, optional
          Shape (length of each transformed axis) of the output
          (` `s[0]` ` refers to axis 0, ``s[1]`` to axis 1, etc.). ` `s` is also the
          number of input points used along this axis, except for the last axis,
          where ``s[-1]//2+1`` points of the input are used.
          Along any axis, if the shape indicated by ` `s` is smaller than that of
          the input, the input is cropped. If it is larger, the input is padded
          with zeros. If ` `s` is not given, the shape of the input along the axes
          specified by axes is used. Except for the last axis which is taken to
          be ``2*(m-1)`` where ``m`` is the length of the input along that axis.
      axes : sequence of ints, optional
          Axes over which to compute the inverse FFT. If not given, the last
          ` `len(s)` ` axes are used, or all axes if ` `s` is also not specified.
          Repeated indices in ` `axes` means that the inverse transform over that
          axis is performed multiple times.
      norm : {"backward", "ortho", "forward"}, optional
          .. versionadded:: 1.10.0
          Normalization mode (see ` `numpy.fft` ). Default is "backward".
          Indicates which direction of the forward/backward pair of transforms
          is scaled and with what normalization factor.
          .. versionadded:: 1.20.0
              The "backward", "forward" values were added.

      Returns
      -----
      out : ndarray
          The truncated or zero-padded input, transformed along the axes
          indicated by ` `axes` , or by a combination of ` `s` or ` `a` ,
          as explained in the parameters section above.
          The length of each transformed axis is as given by the corresponding
          element of ` `s` , or the length of the input in every axis except for
          last one if ` `s` is not given. In the final transformed axis the
          of the output when ` `s` is not given is ``2*(m-1)`` where ``m`` is the
          length of the final transformed axis of the input. To get an odd
          number of output points in the final axis, ` `s` must be specified.

      Raises
      -----
      ValueError
          If ` `s` and ` `axes` have different length.
      IndexError
          If an element of ` `axes` is larger than than the number of axes of ` `a` .
      See Also
      -----
      rfftn : The forward n-dimensional FFT of real input,
              of which ` `ifftn` is the inverse.
      fft : The one-dimensional FFT, with definitions and conventions used.
      irfft : The inverse of the one-dimensional FFT of real input.
      irfft2 : The inverse of the two-dimensional FFT of real input.

      Notes
      -----
      See ` `fft` for definitions and conventions used.
      See ` `rfft` for definitions and conventions used for real input.
      The correct interpretation of the hermitian input depends on the shape of
      the original data, as given by ` `s` . This is because each input shape could
      correspond to either an odd or even length signal. By default, ` `irfftn`'
      assumes an even output length which puts the last entry at the Nyquist
      frequency; aliasing with its symmetric counterpart. When performing the
      final complex to real transform, the last value is thus treated as purely
      real. To avoid losing information, the correct shape of the real input
      **must** be given.

      Examples
      -----
      >>> a = np.zeros((3, 2, 2))
      >>> a[0, 0, 0] = 3 * 2 * 2
      >>> np.fft.irfftn(a)

```

```

1117: (4)         array([[[1.,  1.],
1118: (12)           [1.,  1.]],
1119: (11)           [[1.,  1.],
1120: (12)           [1.,  1.]],
1121: (11)           [[1.,  1.],
1122: (12)           [1.,  1.]]])
1123: (4)
1124: (4)         """
1125: (4)         a = asarray(a)
1126: (4)         s, axes = _cook_nd_args(a, s, axes, invreal=1)
1127: (8)         for ii in range(len(axes)-1):
1128: (4)             a = ifft(a, s[ii], axes[ii], norm)
1129: (4)         a = irfft(a, s[-1], axes[-1], norm)
1130: (0)         return a
1131: (0) @array_function_dispatch(_fftn_dispatcher)
1132: (4) def irfft2(a, s=None, axes=(-2, -1), norm=None):
1133: (4)         """
1134: (4)         Computes the inverse of `rfft2`.
1135: (4)         Parameters
1136: (4)         -----
1137: (8)         a : array_like
1138: (4)             The input array
1139: (8)         s : sequence of ints, optional
1140: (4)             Shape of the real output to the inverse FFT.
1141: (8)         axes : sequence of ints, optional
1142: (8)             The axes over which to compute the inverse fft.
1143: (8)             Default is the last two axes.
1144: (8)         norm : {"backward", "ortho", "forward"}, optional
1145: (8)             .. versionadded:: 1.10.0
1146: (8)             Normalization mode (see `numpy.fft`). Default is "backward".
1147: (8)             Indicates which direction of the forward/backward pair of transforms
1148: (8)             is scaled and with what normalization factor.
1149: (12)             .. versionadded:: 1.20.0
1150: (4)             The "backward", "forward" values were added.
1151: (4)         Returns
1152: (4)         -----
1153: (8)         out : ndarray
1154: (4)             The result of the inverse real 2-D FFT.
1155: (4)         See Also
1156: (4)         -----
1157: (12)         rfft2 : The forward two-dimensional FFT of real input,
1158: (4)             of which `irfft2` is the inverse.
1159: (4)         rfft : The one-dimensional FFT for real input.
1160: (4)         irfft : The inverse of the one-dimensional FFT of real input.
1161: (4)         irfftn : Compute the inverse of the N-dimensional FFT of real input.
1162: (4)         Notes
1163: (4)         -----
1164: (4)         This is really `irfftn` with different defaults.
1165: (4)         For more details see `irfftn`.
1166: (4)         Examples
1167: (4)         -----
1168: (4)         >>> a = np.mgrid[:5, :5][0]
1169: (4)         >>> A = np.fft.rfft2(a)
1170: (4)         >>> np.fft.irfft2(A, s=a.shape)
1171: (11)         array([[0.,  0.,  0.,  0.,  0.],
1172: (11)             [1.,  1.,  1.,  1.,  1.],
1173: (11)             [2.,  2.,  2.,  2.,  2.],
1174: (11)             [3.,  3.,  3.,  3.,  3.],
1175: (4)             [4.,  4.,  4.,  4.,  4.]])
1176: (4)         """
1177: (4)         return irfftn(a, s, axes, norm)

```

-----  
File 200 - \_\_init\_\_.py:

```

1: (0)         """
2: (0)         Discrete Fourier Transform (:mod:`numpy.fft`)
3: (0) =====
4: (0)         .. currentmodule:: numpy.fft

```

```

5: (0) The SciPy module `scipy.fft` is a more comprehensive superset
6: (0) of ``numpy.fft``, which includes only a basic set of routines.
7: (0) Standard FFTs
8: (0)
9: (0) .. autosummary::
10: (3)   :toctree: generated/
11: (3)   fft      Discrete Fourier transform.
12: (3)   ifft     Inverse discrete Fourier transform.
13: (3)   fft2    Discrete Fourier transform in two dimensions.
14: (3)   ifft2   Inverse discrete Fourier transform in two dimensions.
15: (3)   fftn    Discrete Fourier transform in N-dimensions.
16: (3)   ifftn   Inverse discrete Fourier transform in N dimensions.
17: (0) Real FFTs
18: (0)
19: (0) .. autosummary::
20: (3)   :toctree: generated/
21: (3)   rfft     Real discrete Fourier transform.
22: (3)   irfft   Inverse real discrete Fourier transform.
23: (3)   rfft2   Real discrete Fourier transform in two dimensions.
24: (3)   irfft2  Inverse real discrete Fourier transform in two dimensions.
25: (3)   rfftn   Real discrete Fourier transform in N dimensions.
26: (3)   irfftn  Inverse real discrete Fourier transform in N dimensions.
27: (0) Hermitian FFTs
28: (0)
29: (0) .. autosummary::
30: (3)   :toctree: generated/
31: (3)   hfft     Hermitian discrete Fourier transform.
32: (3)   ihfft   Inverse Hermitian discrete Fourier transform.
33: (0) Helper routines
34: (0)
35: (0) .. autosummary::
36: (3)   :toctree: generated/
37: (3)   fftfreq  Discrete Fourier Transform sample frequencies.
38: (3)   rfftfreq DFT sample frequencies (for usage with rfft, irfft).
39: (3)   fftshift  Shift zero-frequency component to center of spectrum.
40: (3)   ifftshift Inverse of fftshift.
41: (0) Background information
42: (0)
43: (0) Fourier analysis is fundamentally a method for expressing a function as a
44: (0) sum of periodic components, and for recovering the function from those
45: (0) components. When both the function and its Fourier transform are
46: (0) replaced with discretized counterparts, it is called the discrete Fourier
47: (0) transform (DFT). The DFT has become a mainstay of numerical computing in
48: (0) part because of a very fast algorithm for computing it, called the Fast
49: (0) Fourier Transform (FFT), which was known to Gauss (1805) and was brought
50: (0) to light in its current form by Cooley and Tukey [CT]_. Press et al. [NR]_
51: (0) provide an accessible introduction to Fourier analysis and its
52: (0) applications.
53: (0) Because the discrete Fourier transform separates its input into
54: (0) components that contribute at discrete frequencies, it has a great number
55: (0) of applications in digital signal processing, e.g., for filtering, and in
56: (0) this context the discretized input to the transform is customarily
57: (0) referred to as a *signal*, which exists in the *time domain*. The output
58: (0) is called a *spectrum* or *transform* and exists in the *frequency
59: (0) domain*.
60: (0) Implementation details
61: (0)
62: (0) There are many ways to define the DFT, varying in the sign of the
63: (0) exponent, normalization, etc. In this implementation, the DFT is defined
64: (0) as
65: (0) .. math::
66: (3)   A_k = \sum_{m=0}^{n-1} a_m \exp\left(-2\pi i \frac{mk}{n}\right)
67: (3)   \quad k = 0, \dots, n-1.
68: (0) The DFT is in general defined for complex inputs and outputs, and a
69: (0) single-frequency component at linear frequency :math:`f` is
70: (0) represented by a complex exponential
71: (0) :math:`a_m = \exp\left(2\pi i f m \Delta t\right)`, where :math:`\Delta t`_
72: (0) is the sampling interval.
73: (0) The values in the result follow so-called "standard" order: If ``A =
```

74: (0)  
 75: (0)  
 76: (0)  
 77: (0)  
 78: (0)  
 79: (0)  
 80: (0)  
 81: (0)  
 82: (0)  
 83: (0)  
 84: (0)  
 85: (0)  
 86: (0)  
 87: (0)  
 88: (0)  
 89: (0)  
 90: (0)  
 91: (0)  
 92: (3)  

$$\left. n \right\rangle \right\}$$
  
 93: (3)  
 94: (0)  
 95: (0)  
 96: (0)  
 97: (0)  
 98: (0)  
 99: (0)  
 100: (0)  
 101: (0)  
 102: (0)  
 103: (0)  
 104: (0)  
 105: (0)  
 106: (0)  
 107: (0)  
 108: (0)  
 109: (0)  
 110: (0)  
 111: (0)  
 112: (0)  
 113: (0)  
 114: (0)  
 115: (0)  
 116: (0)  
 117: (0)  
 118: (0)  
 119: (0)  
 120: (0)  
 121: (0)  
 122: (0)  
 123: (0)  
 124: (0)  
 125: (0)  
 126: (0)  
 127: (0)  
 128: (0)  
 129: (0)  
 130: (0)  
 131: (0)  
 132: (0)  
 133: (0)  
 134: (0)  
 135: (0)  
 136: (0)  
 137: (0)  
 138: (0)  
 139: (0)  
 140: (3)  
 141: (3)

fft(a, n)``, then ``A[0]`` contains the zero-frequency term (the sum of the signal), which is always purely real for real inputs. Then ``A[1:n/2]`` contains the positive-frequency terms, and ``A[n/2+1:]`` contains the negative-frequency terms, in order of decreasingly negative frequency. For an even number of input points, ``A[n/2]`` represents both positive and negative Nyquist frequency, and is also purely real for real input. For an odd number of input points, ``A[(n-1)/2]`` contains the largest positive frequency, while ``A[(n+1)/2]`` contains the largest negative frequency. The routine ``np.fft.freq(n)`` returns an array giving the frequencies of corresponding elements in the output. The routine ``np.fft.fftshift(A)`` shifts transforms and their frequencies to put the zero-frequency components in the middle, and ``np.fft.ifftshift(A)`` undoes that shift.

When the input `a` is a time-domain signal and ``A = fft(a)``, ``np.abs(A)`` is its amplitude spectrum and ``np.abs(A)\*\*2`` is its power spectrum. The phase spectrum is obtained by ``np.angle(A)``.

The inverse DFT is defined as

```
.. math::  

    a_m = \frac{1}{n} \sum_{k=0}^{n-1} A_k \exp(-j \frac{2\pi}{n} k m)
```

It differs from the forward transform by the sign of the exponential argument and the default normalization by :math:`1/n`.

Type Promotion

-----

numpy.fft` promotes ``float32`` and ``complex64`` arrays to ``float64`` and ``complex128`` arrays respectively. For an FFT implementation that does not promote input arrays, see `scipy.fftpack`.

Normalization

-----

The argument ``norm`` indicates which direction of the pair of direct/inverse transforms is scaled and with what normalization factor.

The default normalization (``"backward"``) has the direct (forward) transforms unscaled and the inverse (backward) transforms scaled by :math:`1/n`. It is possible to obtain unitary transforms by setting the keyword argument ``norm`` to ``"ortho"`` so that both direct and inverse transforms are scaled by :math:`1/\sqrt{n}`. Finally, setting the keyword argument ``norm`` to ``"forward"`` has the direct transforms scaled by :math:`1/n` and the inverse transforms unscaled (i.e. exactly opposite to the default ``"backward"``). ``None`` is an alias of the default option ``"backward"`` for backward compatibility.

Real and Hermitian transforms

-----

When the input is purely real, its transform is Hermitian, i.e., the component at frequency :math:f\_k` is the complex conjugate of the component at frequency :math:-f\_k`, which means that for real inputs there is no information in the negative frequency components that is not already available from the positive frequency components.

The family of `rfft` functions is designed to operate on real inputs, and exploits this symmetry by computing only the positive frequency components, up to and including the Nyquist frequency. Thus, ``n`` input points produce ``n/2+1`` complex output points. The inverses of this family assumes the same symmetry of its input, and for an output of ``n`` points uses ``n/2+1`` input points. Correspondingly, when the spectrum is purely real, the signal is Hermitian. The `hfft` family of functions exploits this symmetry by using ``n/2+1`` complex points in the input (time) domain for ``n`` real points in the frequency domain.

In higher dimensions, FFTs are used, e.g., for image analysis and filtering. The computational efficiency of the FFT means that it can also be a faster way to compute large convolutions, using the property that a convolution in the time domain is equivalent to a point-by-point multiplication in the frequency domain.

Higher dimensions

-----

In two dimensions, the DFT is defined as

```
.. math::  

    A_{kl} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} a_{mn} \exp(-j \frac{2\pi}{M} k m + j \frac{2\pi}{N} l n)
```

```

N}\right)\right\}
142: (3)          \quad\quad k = 0, \ldots, M-1;\quad l = 0, \ldots, N-1,
143: (0)          which extends in the obvious way to higher dimensions, and the inverses
144: (0)          in higher dimensions also extend in the same way.
145: (0)          References
146: (0)          -----
147: (0)          .. [CT] Cooley, James W., and John W. Tukey, 1965, "An algorithm for the
148: (8)          machine calculation of complex Fourier series," *Math. Comput.*  

149: (8)          19: 297-301.
150: (0)          .. [NR] Press, W., Teukolsky, S., Vetterline, W.T., and Flannery, B.P.,
151: (8)          2007, *Numerical Recipes: The Art of Scientific Computing*, ch.
152: (8)          12-13. Cambridge Univ. Press, Cambridge, UK.
153: (0)          Examples
154: (0)          -----
155: (0)          For examples, see the various functions.
156: (0)          """
157: (0)          from . import _pocketfft, helper
158: (0)          from ._pocketfft import *
159: (0)          from .helper import *
160: (0)          __all__ = _pocketfft.__all__.copy()
161: (0)          __all__ += helper.__all__
162: (0)          from numpy._pytesttester import PytestTester
163: (0)          test = PytestTester(__name__)
164: (0)          del PytestTester
-----
```

## File 201 - test\_helper.py:

```

1: (0)          """Test functions for fftpack.helper module
2: (0)          Copied from fftpack.helper by Pearu Peterson, October 2005
3: (0)          """
4: (0)          import numpy as np
5: (0)          from numpy.testing import assert_array_almost_equal
6: (0)          from numpy import fft, pi
7: (0)          class TestFFTShift:
8: (4)          def test_definition(self):
9: (8)              x = [0, 1, 2, 3, 4, -4, -3, -2, -1]
10: (8)             y = [-4, -3, -2, -1, 0, 1, 2, 3, 4]
11: (8)             assert_array_almost_equal(fft.fftshift(x), y)
12: (8)             assert_array_almost_equal(fft.ifftshift(y), x)
13: (8)             x = [0, 1, 2, 3, 4, -5, -4, -3, -2, -1]
14: (8)             y = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
15: (8)             assert_array_almost_equal(fft.fftshift(x), y)
16: (8)             assert_array_almost_equal(fft.ifftshift(y), x)
17: (4)          def test_inverse(self):
18: (8)              for n in [1, 4, 9, 100, 211]:
19: (12)                  x = np.random.random((n,))
20: (12)                  assert_array_almost_equal(fft.ifftshift(fft.fftshift(x)), x)
21: (4)          def test_axes_keyword(self):
22: (8)              freqs = [[0, 1, 2], [3, 4, -4], [-3, -2, -1]]
23: (8)              shifted = [[-1, -3, -2], [2, 0, 1], [-4, 3, 4]]
24: (8)              assert_array_almost_equal(fft.fftshift(freqs, axes=(0, 1)), shifted)
25: (8)              assert_array_almost_equal(fft.fftshift(freqs, axes=0),
26: (34)                                fft.fftshift(freqs, axes=(0,)))
27: (8)              assert_array_almost_equal(fft.ifftshift(shifted, axes=(0, 1)), freqs)
28: (8)              assert_array_almost_equal(fft.ifftshift(shifted, axes=0),
29: (34)                                fft.ifftshift(shifted, axes=(0,)))
30: (8)              assert_array_almost_equal(fft.fftshift(freqs), shifted)
31: (8)              assert_array_almost_equal(fft.ifftshift(shifted), freqs)
32: (4)          def test_uneven_dims(self):
33: (8)              """ Test 2D input, which has uneven dimension sizes """
34: (8)              freqs = [
35: (12)                  [0, 1],
36: (12)                  [2, 3],
37: (12)                  [4, 5]
38: (8)              ]
39: (8)              shift_dim0 = [
40: (12)                  [4, 5],
```

```

41: (12)          [0, 1],
42: (12)          [2, 3]
43: (8)
44: (8)
45: (8)
46: (8)
47: (8)
48: (8)
49: (12)
50: (12)
51: (12)
52: (8)
53: (8)
54: (8)
55: (8)
56: (12)
57: (12)
58: (12)
59: (8)
60: (8)
shift_dim_both)
61: (8)          assert_array_almost_equal(fft.fftshift(freqs, axes=0), shift_dim0)
freqs)
62: (8)          assert_array_almost_equal(fft.ifftshift(shift_dim0, axes=0), freqs)
shift_dim_both)
63: (8)          assert_array_almost_equal(fft.fftshift(freqs, axes=1), shift_dim1)
freqs)
64: (8)          assert_array_almost_equal(fft.ifftshift(shift_dim1, axes=1), freqs)
shift_dim_both)
65: (8)          assert_array_almost_equal(fft.fftshift(freqs, axes=[0, 1]), shift_dim_both)
freqs)
66: (8)          assert_array_almost_equal(fft.ifftshift(shift_dim_both, axes=[0, 1]), freqs)
67: (8)
68: (4)          def test_equal_to_original(self):
69: (8)          """ Test that the new (>v1.15) implementation (see #10073) is equal
to the original (<=v1.14) """
70: (8)          from numpy.core import asarray, concatenate, arange, take
71: (8)          def original_fftshift(x, axes=None):
72: (12)          """ How fftshift was implemented in v1.14 """
73: (12)          tmp = asarray(x)
74: (12)          ndim = tmp.ndim
75: (12)          if axes is None:
76: (16)          axes = list(range(ndim))
77: (12)          elif isinstance(axes, int):
78: (16)          axes = (axes,)
79: (12)          y = tmp
80: (12)          for k in axes:
81: (16)          n = tmp.shape[k]
82: (16)          p2 = (n + 1) // 2
83: (16)          mylist = concatenate((arange(p2, n), arange(p2)))
84: (16)          y = take(y, mylist, k)
85: (12)          return y
86: (8)          def original_ifftshift(x, axes=None):
87: (12)          """ How ifftshift was implemented in v1.14 """
88: (12)          tmp = asarray(x)
89: (12)          ndim = tmp.ndim
90: (12)          if axes is None:
91: (16)          axes = list(range(ndim))
92: (12)          elif isinstance(axes, int):
93: (16)          axes = (axes,)
94: (12)          y = tmp
95: (12)          for k in axes:
96: (16)          n = tmp.shape[k]
97: (16)          p2 = n - (n + 1) // 2
98: (16)          mylist = concatenate((arange(p2, n), arange(p2)))
99: (16)          y = take(y, mylist, k)
100: (12)         return y
101: (8)         for i in range(16):
102: (12)             for j in range(16):

```

```

103: (16)                         for axes_keyword in [0, 1, None, (0,), (0, 1)]:
104: (20)                           inp = np.random.rand(i, j)
105: (20)                           assert_array_almost_equal(fft.fftshift(inp, axes_keyword),
106: (46)   original_fftshift(inp,
axes_keyword))
107: (20)                               assert_array_almost_equal(fft.ifftshift(inp,
108: (46)   original_ifftshift(inp,
axes_keyword))
109: (0)                                class TestFFTFreq:
110: (4)                                  def test_definition(self):
111: (8)                                    x = [0, 1, 2, 3, 4, -4, -3, -2, -1]
112: (8)                                    assert_array_almost_equal(9*fft.fftfreq(9), x)
113: (8)                                    assert_array_almost_equal(9*pi*fft.fftfreq(9, pi), x)
114: (8)                                    x = [0, 1, 2, 3, 4, -5, -4, -3, -2, -1]
115: (8)                                    assert_array_almost_equal(10*fft.fftfreq(10), x)
116: (8)                                    assert_array_almost_equal(10*pi*fft.fftfreq(10, pi), x)
117: (0)                                class TestRFFTFreq:
118: (4)                                  def test_definition(self):
119: (8)                                    x = [0, 1, 2, 3, 4]
120: (8)                                    assert_array_almost_equal(9*rfft.rfftfreq(9), x)
121: (8)                                    assert_array_almost_equal(9*pi*rfft.rfftfreq(9, pi), x)
122: (8)                                    x = [0, 1, 2, 3, 4, 5]
123: (8)                                    assert_array_almost_equal(10*rfft.rfftfreq(10), x)
124: (8)                                    assert_array_almost_equal(10*pi*rfft.rfftfreq(10, pi), x)
125: (0)                                class TestIRFFTN:
126: (4)                                  def test_not_last_axis_success(self):
127: (8)                                    ar, ai = np.random.random((2, 16, 8, 32))
128: (8)                                    a = ar + 1j*ai
129: (8)                                    axes = (-2,)
130: (8)                                    fft.irfftn(a, axes=axes)
-----
```

## File 202 - \_\_init\_\_.py:

1: (0)

## File 203 - test\_pocketfft.py:

```

1: (0)          import numpy as np
2: (0)          import pytest
3: (0)          from numpy.random import random
4: (0)          from numpy.testing import (
5: (8)            assert_array_equal, assert_raises, assert_allclose, IS_WASM
6: (8)          )
7: (0)          import threading
8: (0)          import queue
9: (0)          def fft1(x):
10: (4)            L = len(x)
11: (4)            phase = -2j * np.pi * (np.arange(L) / L)
12: (4)            phase = np.arange(L).reshape(-1, 1) * phase
13: (4)            return np.sum(x*np.exp(phase), axis=1)
14: (0)          class TestFFTShift:
15: (4)            def test_fft_n(self):
16: (8)              assert_raises(ValueError, np.fft.fft, [1, 2, 3], 0)
17: (0)          class TestFFT1D:
18: (4)            def test_identity(self):
19: (8)              maxlen = 512
20: (8)              x = random(maxlen) + 1j*random(maxlen)
21: (8)              xr = random(maxlen)
22: (8)              for i in range(1, maxlen):
23: (12)                assert_allclose(np.fft.ifft(np.fft.fft(x[0:i])), x[0:i],
24: (28)                  atol=1e-12)
25: (12)                assert_allclose(np.fft.irfft(np.fft.rfft(xr[0:i])), i),
26: (28)                  xr[0:i], atol=1e-12)
27: (4)            def test_fft(self):
```

```

28: (8)           x = random(30) + 1j*random(30)
29: (8)           assert_allclose(fft1(x), np.fft.fft(x), atol=1e-6)
30: (8)           assert_allclose(fft1(x), np.fft.fft(x, norm="backward"), atol=1e-6)
31: (8)           assert_allclose(fft1(x) / np.sqrt(30),
32: (24)                 np.fft.fft(x, norm="ortho"), atol=1e-6)
33: (8)           assert_allclose(fft1(x) / 30.,
34: (24)                 np.fft.fft(x, norm="forward"), atol=1e-6)
35: (4) @pytest.mark.parametrize('norm', (None, 'backward', 'ortho', 'forward'))
36: (4) def test_ifft(self, norm):
37: (8)     x = random(30) + 1j*random(30)
38: (8)     assert_allclose(
39: (12)         x, np.fft.ifft(np.fft.fft(x, norm=norm), norm=norm),
40: (12)         atol=1e-6)
41: (8)     with pytest.raises(ValueError,
42: (27)             match='Invalid number of FFT data points'):
43: (12)         np.fft.ifft([], norm=norm)
44: (4) def test_fft2(self):
45: (8)     x = random((30, 20)) + 1j*random((30, 20))
46: (8)     assert_allclose(np.fft.fft(np.fft.fft(x, axis=1), axis=0),
47: (24)                 np.fft.fft2(x), atol=1e-6)
48: (8)     assert_allclose(np.fft.fft2(x),
49: (24)                 np.fft.fft2(x, norm="backward"), atol=1e-6)
50: (8)     assert_allclose(np.fft.fft2(x) / np.sqrt(30 * 20),
51: (24)                 np.fft.fft2(x, norm="ortho"), atol=1e-6)
52: (8)     assert_allclose(np.fft.fft2(x) / (30. * 20.),
53: (24)                 np.fft.fft2(x, norm="forward"), atol=1e-6)
54: (4) def test_ifft2(self):
55: (8)     x = random((30, 20)) + 1j*random((30, 20))
56: (8)     assert_allclose(np.fft.ifft(np.fft.ifft(x, axis=1), axis=0),
57: (24)                 np.fft.ifft2(x), atol=1e-6)
58: (8)     assert_allclose(np.fft.ifft2(x),
59: (24)                 np.fft.ifft2(x, norm="backward"), atol=1e-6)
60: (8)     assert_allclose(np.fft.ifft2(x) * np.sqrt(30 * 20),
61: (24)                 np.fft.ifft2(x, norm="ortho"), atol=1e-6)
62: (8)     assert_allclose(np.fft.ifft2(x) * (30. * 20.),
63: (24)                 np.fft.ifft2(x, norm="forward"), atol=1e-6)
64: (4) def test_fftn(self):
65: (8)     x = random((30, 20, 10)) + 1j*random((30, 20, 10))
66: (8)     assert_allclose(
67: (12)         np.fft.fft(np.fft.fft(x, axis=2), axis=1), axis=0),
68: (12)         np.fft.fftn(x), atol=1e-6)
69: (8)     assert_allclose(np.fft.fftn(x),
70: (24)                 np.fft.fftn(x, norm="backward"), atol=1e-6)
71: (8)     assert_allclose(np.fft.fftn(x) / np.sqrt(30 * 20 * 10),
72: (24)                 np.fft.fftn(x, norm="ortho"), atol=1e-6)
73: (8)     assert_allclose(np.fft.fftn(x) / (30. * 20. * 10.),
74: (24)                 np.fft.fftn(x, norm="forward"), atol=1e-6)
75: (4) def test_iffftn(self):
76: (8)     x = random((30, 20, 10)) + 1j*random((30, 20, 10))
77: (8)     assert_allclose(
78: (12)         np.fft.ifft(np.fft.ifft(np.fft.ifft(x, axis=2), axis=1), axis=0),
79: (12)         np.fft.iffftn(x), atol=1e-6)
80: (8)     assert_allclose(np.fft.iffftn(x),
81: (24)                 np.fft.iffftn(x, norm="backward"), atol=1e-6)
82: (8)     assert_allclose(np.fft.iffftn(x) * np.sqrt(30 * 20 * 10),
83: (24)                 np.fft.iffftn(x, norm="ortho"), atol=1e-6)
84: (8)     assert_allclose(np.fft.iffftn(x) * (30. * 20. * 10.),
85: (24)                 np.fft.iffftn(x, norm="forward"), atol=1e-6)
86: (4) def test_rfft(self):
87: (8)     x = random(30)
88: (8)     for n in [x.size, 2*x.size]:
89: (12)         for norm in [None, 'backward', 'ortho', 'forward']:
90: (16)             assert_allclose(
91: (20)                 np.fft.fft(x, n=n, norm=norm)[:(n//2 + 1)],
92: (20)                 np.fft.rfft(x, n=n, norm=norm), atol=1e-6)
93: (12)             assert_allclose(
94: (16)                 np.fft.rfft(x, n=n),
95: (16)                 np.fft.rfft(x, n=n, norm="backward"), atol=1e-6)
96: (12)             assert_allclose(

```

```

97: (16) np.fft.rfft(x, n=n) / np.sqrt(n),
98: (16) np.fft.rfft(x, n=n, norm="ortho"), atol=1e-6)
99: (12) assert_allclose(
100: (16) np.fft.rfft(x, n=n) / n,
101: (16) np.fft.rfft(x, n=n, norm="forward"), atol=1e-6)
102: (4) def test_irfft(self):
103: (8) x = random(30)
104: (8) assert_allclose(x, np.fft.irfft(np.fft.rfft(x)), atol=1e-6)
105: (8) assert_allclose(x, np.fft.irfft(np.fft.rfft(x, norm="backward")),
106: (24) norm="backward"), atol=1e-6)
107: (8) assert_allclose(x, np.fft.irfft(np.fft.rfft(x, norm="ortho")),
108: (24) norm="ortho"), atol=1e-6)
109: (8) assert_allclose(x, np.fft.irfft(np.fft.rfft(x, norm="forward")),
110: (24) norm="forward"), atol=1e-6)
111: (4) def test_rfft2(self):
112: (8) x = random((30, 20))
113: (8) assert_allclose(np.fft.fft2(x)[:, :11], np.fft.rfft2(x), atol=1e-6)
114: (8) assert_allclose(np.fft.rfft2(x),
115: (24) np.fft.rfft2(x, norm="backward"), atol=1e-6)
116: (8) assert_allclose(np.fft.rfft2(x) / np.sqrt(30 * 20),
117: (24) np.fft.rfft2(x, norm="ortho"), atol=1e-6)
118: (8) assert_allclose(np.fft.rfft2(x) / (30. * 20.),
119: (24) np.fft.rfft2(x, norm="forward"), atol=1e-6)
120: (4) def test_irfft2(self):
121: (8) x = random((30, 20))
122: (8) assert_allclose(x, np.fft.irfft2(np.fft.rfft2(x)), atol=1e-6)
123: (8) assert_allclose(x, np.fft.irfft2(np.fft.rfft2(x, norm="backward")),
124: (24) norm="backward"), atol=1e-6)
125: (8) assert_allclose(x, np.fft.irfft2(np.fft.rfft2(x, norm="ortho")),
126: (24) norm="ortho"), atol=1e-6)
127: (8) assert_allclose(x, np.fft.irfft2(np.fft.rfft2(x, norm="forward")),
128: (24) norm="forward"), atol=1e-6)
129: (4) def test_rfftn(self):
130: (8) x = random((30, 20, 10))
131: (8) assert_allclose(np.fft.fftn(x)[:, :, :6], np.fft.rfftn(x), atol=1e-6)
132: (8) assert_allclose(np.fft.rfftn(x),
133: (24) np.fft.rfftn(x, norm="backward"), atol=1e-6)
134: (8) assert_allclose(np.fft.rfftn(x) / np.sqrt(30 * 20 * 10),
135: (24) np.fft.rfftn(x, norm="ortho"), atol=1e-6)
136: (8) assert_allclose(np.fft.rfftn(x) / (30. * 20. * 10.),
137: (24) np.fft.rfftn(x, norm="forward"), atol=1e-6)
138: (4) def test_irfftn(self):
139: (8) x = random((30, 20, 10))
140: (8) assert_allclose(x, np.fft.irfftn(np.fft.rfftn(x)), atol=1e-6)
141: (8) assert_allclose(x, np.fft.irfftn(np.fft.rfftn(x, norm="backward")),
142: (24) norm="backward"), atol=1e-6)
143: (8) assert_allclose(x, np.fft.irfftn(np.fft.rfftn(x, norm="ortho")),
144: (24) norm="ortho"), atol=1e-6)
145: (8) assert_allclose(x, np.fft.irfftn(np.fft.rfftn(x, norm="forward")),
146: (24) norm="forward"), atol=1e-6)
147: (4) def test_hfft(self):
148: (8) x = random(14) + 1j*random(14)
149: (8) x_herm = np.concatenate((random(1), x, random(1)))
150: (8) x = np.concatenate((x_herm, x[::-1].conj()))
151: (8) assert_allclose(np.fft.fft(x), np.fft.hfft(x_herm), atol=1e-6)
152: (8) assert_allclose(np.fft.hfft(x_herm),
153: (24) np.fft.hfft(x_herm, norm="backward"), atol=1e-6)
154: (8) assert_allclose(np.fft.hfft(x_herm) / np.sqrt(30),
155: (24) np.fft.hfft(x_herm, norm="ortho"), atol=1e-6)
156: (8) assert_allclose(np.fft.hfft(x_herm) / 30.,
157: (24) np.fft.hfft(x_herm, norm="forward"), atol=1e-6)
158: (4) def test_ihfft(self):
159: (8) x = random(14) + 1j*random(14)
160: (8) x_herm = np.concatenate((random(1), x, random(1)))
161: (8) x = np.concatenate((x_herm, x[::-1].conj()))
162: (8) assert_allclose(x_herm, np.fft.ihfft(np.fft.hfft(x_herm)), atol=1e-6)
163: (8) assert_allclose(x_herm, np.fft.ihfft(np.fft.hfft(x_herm,
164: (24) norm="backward"), norm="backward"), atol=1e-6)
165: (8) assert_allclose(x_herm, np.fft.ihfft(np.fft.hfft(x_herm,
```

```

166: (24)                         norm="ortho"), norm="ortho"), atol=1e-6)
167: (8)                          assert_allclose(x_herm, np.fft.ihfft(np.fft.hfft(x_herm,
168: (24)   norm="forward"), norm="forward"), atol=1e-6)
169: (4) @pytest.mark.parametrize("op", [np.fft.fftn, np.fft.ifftn,
170: (36)   np.fft.rfftn, np.fft.irfftn])
171: (4) def test_axes(self, op):
172: (8)     x = random((30, 20, 10))
173: (8)     axes = [(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1,
0)]
174: (8)     for a in axes:
175: (12)         op_tr = op(np.transpose(x, a))
176: (12)         tr_op = np.transpose(op(x, axes=a), a)
177: (12)         assert_allclose(op_tr, tr_op, atol=1e-6)
178: (4) def test_all_1d_norm_preserving(self):
179: (8)     x = random(30)
180: (8)     x_norm = np.linalg.norm(x)
181: (8)     n = x.size * 2
182: (8)     func_pairs = [(np.fft.fft, np.fft.ifft),
183: (22)                     (np.fft.rfft, np.fft.irfft),
184: (22)                     (np.fft.ihfft, np.fft.hfft),
185: (22)                     ]
186: (8)     for forw, back in func_pairs:
187: (12)         for n in [x.size, 2*x.size]:
188: (16)             for norm in [None, 'backward', 'ortho', 'forward']:
189: (20)                 tmp = forw(x, n=n, norm=norm)
190: (20)                 tmp = back(tmp, n=n, norm=norm)
191: (20)                 assert_allclose(x_norm,
192: (36)                               np.linalg.norm(tmp), atol=1e-6)
193: (4) @pytest.mark.parametrize("dtype", [np.half, np.single, np.double,
194: (39)   np.longdouble])
195: (4) def test_dtypes(self, dtype):
196: (8)     x = random(30).astype(dtype)
197: (8)     assert_allclose(np.fft.ifft(np.fft.fft(x)), x, atol=1e-6)
198: (8)     assert_allclose(np.fft.irfft(np.fft.rfft(x)), x, atol=1e-6)
199: (0) @pytest.mark.parametrize(
200: (8)     "dtype",
201: (8)     [np.float32, np.float64, np.complex64, np.complex128])
202: (0) @pytest.mark.parametrize("order", ["F", 'non-contiguous'])
203: (0) @pytest.mark.parametrize(
204: (8)     "fft",
205: (8)     [np.fft.fft, np.fft.fft2, np.fft.fftn,
206: (9)         np.fft.ifft, np.fft.ifft2, np.fft.ifftn])
207: (0) def test_fft_with_order(dtype, order, fft):
208: (4)     rng = np.random.RandomState(42)
209: (4)     X = rng.rand(8, 7, 13).astype(dtype, copy=False)
210: (4)     tol = 8.0 * np.sqrt(np.log2(X.size)) * np.finfo(X.dtype).eps
211: (4)     if order == 'F':
212: (8)         Y = np.asfortranarray(X)
213: (4)     else:
214: (8)         Y = X[::-1]
215: (8)         X = np.ascontiguousarray(X[::-1])
216: (4)     if fft.__name__.endswith('fft'):
217: (8)         for axis in range(3):
218: (12)             X_res = fft(X, axis=axis)
219: (12)             Y_res = fft(Y, axis=axis)
220: (12)             assert_allclose(X_res, Y_res, atol=_tol, rtol=_tol)
221: (4)     elif fft.__name__.endswith(('fft2', 'fftn')):
222: (8)         axes = [(0, 1), (1, 2), (0, 2)]
223: (8)         if fft.__name__.endswith('fftn'):
224: (12)             axes.extend([(0,), (1,), (2,), None])
225: (8)         for ax in axes:
226: (12)             X_res = fft(X, axes=ax)
227: (12)             Y_res = fft(Y, axes=ax)
228: (12)             assert_allclose(X_res, Y_res, atol=_tol, rtol=_tol)
229: (4)     else:
230: (8)         raise ValueError()
231: (0) @pytest.mark.skipif(IS_WASM, reason="Cannot start thread")
232: (0) class TestFFTThreadSafe:
233: (4)     threads = 16

```

```

234: (4)           input_shape = (800, 200)
235: (4)           def _test_mtsame(self, func, *args):
236: (8)             def worker(args, q):
237: (12)               q.put(func(*args))
238: (8)               q = queue.Queue()
239: (8)               expected = func(*args)
240: (8)               t = [threading.Thread(target=worker, args=(args, q))
241: (13)                 for i in range(self.threads)]
242: (8)               [x.start() for x in t]
243: (8)               [x.join() for x in t]
244: (8)               for i in range(self.threads):
245: (12)                 assert_array_equal(q.get(timeout=5), expected,
246: (16)                   'Function returned wrong value in multithreaded context')
247: (4)           def test_fft(self):
248: (8)             a = np.ones(self.input_shape) * 1+0j
249: (8)             self._test_mtsame(np.fft.fft, a)
250: (4)           def test_ifft(self):
251: (8)             a = np.ones(self.input_shape) * 1+0j
252: (8)             self._test_mtsame(np.fft.ifft, a)
253: (4)           def test_rfft(self):
254: (8)             a = np.ones(self.input_shape)
255: (8)             self._test_mtsame(np.fft.rfft, a)
256: (4)           def test_irfft(self):
257: (8)             a = np.ones(self.input_shape) * 1+0j
258: (8)             self._test_mtsame(np.fft.irfft, a)

```

-----

## File 204 - arraypad.py:

```

1: (0)           """
2: (0)             The arraypad module contains a group of functions to pad values onto the edges
3: (0)             of an n-dimensional array.
4: (0)           """
5: (0)           import numpy as np
6: (0)           from numpy.core.overrides import array_function_dispatch
7: (0)           from numpy.lib.index_tricks import ndindex
8: (0)           __all__ = ['pad']
9: (0)           def _round_if_needed(arr, dtype):
10: (4)             """
11: (4)               Rounds arr inplace if destination dtype is integer.
12: (4)               Parameters
13: (4)               -----
14: (4)               arr : ndarray
15: (8)                 Input array.
16: (4)               dtype : dtype
17: (8)                 The dtype of the destination array.
18: (4)               """
19: (4)               if np.issubdtype(dtype, np.integer):
20: (8)                 arr.round(out=arr)
21: (0)           def _slice_at_axis(sl, axis):
22: (4)             """
23: (4)               Construct tuple of slices to slice an array in the given dimension.
24: (4)               Parameters
25: (4)               -----
26: (4)               sl : slice
27: (8)                 The slice for the given dimension.
28: (4)               axis : int
29: (8)                 The axis to which `sl` is applied. All other dimensions are left
30: (8)                 "unsliced".
31: (4)               Returns
32: (4)               -----
33: (4)               sl : tuple of slices
34: (8)                 A tuple with slices matching `shape` in length.
35: (4)               Examples
36: (4)               -----
37: (4)               >>> _slice_at_axis(slice(None, 3, -1), 1)
38: (4)               (slice(None, None, None), slice(None, 3, -1), (...,))
39: (4)               """

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

40: (4)           return (slice(None),) * axis + (sl,) + (...,)
41: (0)           def _view_roi(array, original_area_slice, axis):
42: (4)             """
43: (4)               Get a view of the current region of interest during iterative padding.
44: (4)               When padding multiple dimensions iteratively corner values are
45: (4)               unnecessarily overwritten multiple times. This function reduces the
46: (4)               working area for the first dimensions so that corners are excluded.
47: (4)               Parameters
48: (4)                 -----
49: (4)               array : ndarray
50: (8)                 The array with the region of interest.
51: (4)               original_area_slice : tuple of slices
52: (8)                 Denotes the area with original values of the unpadded array.
53: (4)               axis : int
54: (8)                 The currently padded dimension assuming that `axis` is padded before
55: (8)                 `axis` + 1.
56: (4)               Returns
57: (4)                 -----
58: (4)               roi : ndarray
59: (8)                 The region of interest of the original `array`.
60: (4)               """
61: (4)               axis += 1
62: (4)               sl = (slice(None),) * axis + original_area_slice[axis:]
63: (4)               return array[sl]
64: (0)           def _pad_simple(array, pad_width, fill_value=None):
65: (4)             """
66: (4)               Pad array on all sides with either a single value or undefined values.
67: (4)               Parameters
68: (4)                 -----
69: (4)               array : ndarray
70: (8)                 Array to grow.
71: (4)               pad_width : sequence of tuple[int, int]
72: (8)                 Pad width on both sides for each dimension in `arr`.
73: (4)               fill_value : scalar, optional
74: (8)                 If provided the padded area is filled with this value, otherwise
75: (8)                 the pad area left undefined.
76: (4)               Returns
77: (4)                 -----
78: (4)               padded : ndarray
79: (8)                 The padded array with the same dtype as `array`. Its order will default
80: (8)                 to C-style if `array` is not F-contiguous.
81: (4)               original_area_slice : tuple
82: (8)                 A tuple of slices pointing to the area of the original array.
83: (4)               """
84: (4)               new_shape = tuple(
85: (8)                 left + size + right
86: (8)                 for size, (left, right) in zip(array.shape, pad_width)
87: (4)               )
88: (4)               order = 'F' if array.flags.fnc else 'C' # Fortran and not also C-order
89: (4)               padded = np.empty(new_shape, dtype=array.dtype, order=order)
90: (4)               if fill_value is not None:
91: (8)                 padded.fill(fill_value)
92: (4)               original_area_slice = tuple(
93: (8)                 slice(left, left + size)
94: (8)                 for size, (left, right) in zip(array.shape, pad_width)
95: (4)               )
96: (4)               padded[original_area_slice] = array
97: (4)               return padded, original_area_slice
98: (0)           def _set_pad_area(padded, axis, width_pair, value_pair):
99: (4)             """
100: (4)               Set empty-padded area in given dimension.
101: (4)               Parameters
102: (4)                 -----
103: (4)               padded : ndarray
104: (8)                 Array with the pad area which is modified inplace.
105: (4)               axis : int
106: (8)                 Dimension with the pad area to set.
107: (4)               width_pair : (int, int)
108: (8)                 Pair of widths that mark the pad area on both sides in the given

```

```

109: (8)           dimension.
110: (4)           value_pair : tuple of scalars or ndarrays
111: (8)           Values inserted into the pad area on each side. It must match or be
112: (8)           broadcastable to the shape of `arr`.
113: (4)
114: (4)           left_slice = _slice_at_axis(slice(None, width_pair[0]), axis)
115: (4)           padded[left_slice] = value_pair[0]
116: (4)           right_slice = _slice_at_axis(
117: (8)             slice(padded.shape[axis] - width_pair[1], None), axis)
118: (4)           padded[right_slice] = value_pair[1]
119: (0)           def _get_edges(padded, axis, width_pair):
120: (4)
121: (4)             """Retrieve edge values from empty-padded array in given dimension.
122: (4)             Parameters
123: (4)             -----
124: (4)               padded : ndarray
125: (8)                 Empty-padded array.
126: (4)               axis : int
127: (8)                 Dimension in which the edges are considered.
128: (4)               width_pair : (int, int)
129: (8)                 Pair of widths that mark the pad area on both sides in the given
130: (8)                 dimension.
131: (4)             Returns
132: (4)
133: (4)             left_edge, right_edge : ndarray
134: (8)               Edge values of the valid area in `padded` in the given dimension. Its
135: (8)               shape will always match `padded` except for the dimension given by
136: (8)               `axis` which will have a length of 1.
137: (4)
138: (4)             left_index = width_pair[0]
139: (4)             left_slice = _slice_at_axis(slice(left_index, left_index + 1), axis)
140: (4)             left_edge = padded[left_slice]
141: (4)             right_index = padded.shape[axis] - width_pair[1]
142: (4)             right_slice = _slice_at_axis(slice(right_index - 1, right_index), axis)
143: (4)             right_edge = padded[right_slice]
144: (4)             return left_edge, right_edge
145: (0)           def _get_linear_ramps(padded, axis, width_pair, end_value_pair):
146: (4)
147: (4)             """Construct linear ramps for empty-padded array in given dimension.
148: (4)             Parameters
149: (4)
150: (4)               padded : ndarray
151: (8)                 Empty-padded array.
152: (4)               axis : int
153: (8)                 Dimension in which the ramps are constructed.
154: (4)               width_pair : (int, int)
155: (8)                 Pair of widths that mark the pad area on both sides in the given
156: (8)                 dimension.
157: (4)               end_value_pair : (scalar, scalar)
158: (8)                 End values for the linear ramps which form the edge of the fully
padded
159: (8)                 array. These values are included in the linear ramps.
160: (4)             Returns
161: (4)
162: (4)             left_ramp, right_ramp : ndarray
163: (8)               Linear ramps to set on both sides of `padded`.
164: (4)
165: (4)             edge_pair = _get_edges(padded, axis, width_pair)
166: (4)             left_ramp, right_ramp = (
167: (8)               np.linspace(
168: (12)                 start=end_value,
169: (12)                 stop=edge.squeeze(axis), # Dimension is replaced by linspace
170: (12)                 num=width,
171: (12)                 endpoint=False,
172: (12)                 dtype=padded.dtype,
173: (12)                 axis=axis
174: (8)               )
175: (8)               for end_value, edge, width in zip(
176: (12)                 end_value_pair, edge_pair, width_pair

```

```

177: (8) )
178: (4) )
179: (4) right_ramp = right_ramp[_slice_at_axis(slice(None, None, -1), axis)]
180: (4) return left_ramp, right_ramp
181: (0) def _get_stats(padded, axis, width_pair, length_pair, stat_func):
182: (4) """
183: (4) Calculate statistic for the empty-padded array in given dimension.
184: (4) Parameters
185: (4) -----
186: (4) padded : ndarray
187: (8)     Empty-padded array.
188: (4) axis : int
189: (8)     Dimension in which the statistic is calculated.
190: (4) width_pair : (int, int)
191: (8)     Pair of widths that mark the pad area on both sides in the given
192: (8)     dimension.
193: (4) length_pair : 2-element sequence of None or int
194: (8)     Gives the number of values in valid area from each side that is
195: (8)     taken into account when calculating the statistic. If None the entire
196: (8)     valid area in `padded` is considered.
197: (4) stat_func : function
198: (8)     Function to compute statistic. The expected signature is
199: (8)     ``stat_func(x: ndarray, axis: int, keepdims: bool) -> ndarray``.
200: (4) Returns
201: (4) -----
202: (4) left_stat, right_stat : ndarray
203: (8)     Calculated statistic for both sides of `padded`.
204: (4) """
205: (4) left_index = width_pair[0]
206: (4) right_index = padded.shape[axis] - width_pair[1]
207: (4) max_length = right_index - left_index
208: (4) left_length, right_length = length_pair
209: (4) if left_length is None or max_length < left_length:
210: (8)     left_length = max_length
211: (4) if right_length is None or max_length < right_length:
212: (8)     right_length = max_length
213: (4) if (left_length == 0 or right_length == 0) \
214: (12)     and stat_func in {np.amax, np.amin}:
215: (8)     raise ValueError("stat_length of 0 yields no value for padding")
216: (4) left_slice = _slice_at_axis(
217: (8)     slice(left_index, left_index + left_length), axis)
218: (4) left_chunk = padded[left_slice]
219: (4) left_stat = stat_func(left_chunk, axis=axis, keepdims=True)
220: (4) _round_if_needed(left_stat, padded.dtype)
221: (4) if left_length == right_length == max_length:
222: (8)     return left_stat, left_stat
223: (4) right_slice = _slice_at_axis(
224: (8)     slice(right_index - right_length, right_index), axis)
225: (4) right_chunk = padded[right_slice]
226: (4) right_stat = stat_func(right_chunk, axis=axis, keepdims=True)
227: (4) _round_if_needed(right_stat, padded.dtype)
228: (4) return left_stat, right_stat
229: (0) def _set_reflect_both(padded, axis, width_pair, method, include_edge=False):
230: (4) """
231: (4) Pad `axis` of `arr` with reflection.
232: (4) Parameters
233: (4) -----
234: (4) padded : ndarray
235: (8)     Input array of arbitrary shape.
236: (4) axis : int
237: (8)     Axis along which to pad `arr`.
238: (4) width_pair : (int, int)
239: (8)     Pair of widths that mark the pad area on both sides in the given
240: (8)     dimension.
241: (4) method : str
242: (8)     Controls method of reflection; options are 'even' or 'odd'.
243: (4) include_edge : bool
244: (8)     If true, edge value is included in reflection, otherwise the edge
245: (8)     value forms the symmetric axis to the reflection.

```

```

246: (4)             Returns
247: (4)
248: (4)
249: (8)             pad_amt : tuple of ints, length 2
250: (8)             New index positions of padding to do along the `axis`. If these are
251: (4)             both 0, padding is done in this dimension.
252: (4)
253: (4)
254: (4)
255: (8)
256: (4)
257: (8)
258: (8)
259: (4)
260: (8)
261: (8)
262: (8)
263: (8)
264: (8)
265: (8)
266: (12)
267: (12)
268: (8)
269: (8)
270: (8)
271: (8)
272: (8)
273: (4)
274: (8)
275: (8)
276: (8)
277: (8)
278: (8)
279: (8)
280: (12)
281: (16)
282: (12)
283: (8)
284: (8)
285: (8)
286: (8)
287: (8)
288: (4)
289: (0)
290: (4)
291: (4)
292: (4)
293: (4)
294: (4)
295: (8)
296: (4)
297: (8)
298: (4)
299: (8)
300: (8)
301: (4)
302: (8)
303: (4)
304: (4)
305: (4)
306: (8)
307: (8)
308: (4)
309: (4)
310: (4)
311: (4)
312: (4)
313: (4)
314: (4)

        -----
        pad_amt : tuple of ints, length 2
        New index positions of padding to do along the `axis`. If these are
        both 0, padding is done in this dimension.
        """
        left_pad, right_pad = width_pair
        old_length = padded.shape[axis] - right_pad - left_pad
        if include_edge:
            edge_offset = 1
        else:
            edge_offset = 0 # Edge is not included, no need to offset pad amount
            old_length -= 1 # but must be omitted from the chunk
        if left_pad > 0:
            chunk_length = min(old_length, left_pad)
            stop = left_pad - edge_offset
            start = stop + chunk_length
            left_slice = _slice_at_axis(slice(start, stop, -1), axis)
            left_chunk = padded[left_slice]
            if method == "odd":
                edge_slice = _slice_at_axis(slice(left_pad, left_pad + 1), axis)
                left_chunk = 2 * padded[edge_slice] - left_chunk
            start = left_pad - chunk_length
            stop = left_pad
            pad_area = _slice_at_axis(slice(start, stop), axis)
            padded[pad_area] = left_chunk
            left_pad -= chunk_length
        if right_pad > 0:
            chunk_length = min(old_length, right_pad)
            start = -right_pad + edge_offset - 2
            stop = start - chunk_length
            right_slice = _slice_at_axis(slice(start, stop, -1), axis)
            right_chunk = padded[right_slice]
            if method == "odd":
                edge_slice = _slice_at_axis(
                    slice(-right_pad - 1, -right_pad), axis)
                right_chunk = 2 * padded[edge_slice] - right_chunk
            start = padded.shape[axis] - right_pad
            stop = start + chunk_length
            pad_area = _slice_at_axis(slice(start, stop), axis)
            padded[pad_area] = right_chunk
            right_pad -= chunk_length
        return left_pad, right_pad
def _set_wrap_both(padded, axis, width_pair, original_period):
    """
        Pad `axis` of `arr` with wrapped values.
        Parameters
        -----
        padded : ndarray
            Input array of arbitrary shape.
        axis : int
            Axis along which to pad `arr`.
        width_pair : (int, int)
            Pair of widths that mark the pad area on both sides in the given
            dimension.
        original_period : int
            Original length of data on `axis` of `arr`.
        Returns
        -----
        pad_amt : tuple of ints, length 2
        New index positions of padding to do along the `axis`. If these are
        both 0, padding is done in this dimension.
        """
        left_pad, right_pad = width_pair
        period = padded.shape[axis] - right_pad - left_pad
        period = period // original_period * original_period
        new_left_pad = 0
        new_right_pad = 0
        if left_pad > 0:

```

```

315: (8)             slice_end = left_pad + period
316: (8)             slice_start = slice_end - min(period, left_pad)
317: (8)             right_slice = _slice_at_axis(slice(slice_start, slice_end), axis)
318: (8)             right_chunk = padded[right_slice]
319: (8)             if left_pad > period:
320: (12)                 pad_area = _slice_at_axis(slice(left_pad - period, left_pad),
axis)
321: (12)                     new_left_pad = left_pad - period
322: (8)             else:
323: (12)                 pad_area = _slice_at_axis(slice(None, left_pad), axis)
324: (8)                 padded[pad_area] = right_chunk
325: (4)             if right_pad > 0:
326: (8)                 slice_start = -right_pad - period
327: (8)                 slice_end = slice_start + min(period, right_pad)
328: (8)                 left_slice = _slice_at_axis(slice(slice_start, slice_end), axis)
329: (8)                 left_chunk = padded[left_slice]
330: (8)                 if right_pad > period:
331: (12)                     pad_area = _slice_at_axis(
332: (16)                         slice(-right_pad, -right_pad + period), axis)
333: (12)                     new_right_pad = right_pad - period
334: (8)                 else:
335: (12)                     pad_area = _slice_at_axis(slice(-right_pad, None), axis)
336: (8)                     padded[pad_area] = left_chunk
337: (4)             return new_left_pad, new_right_pad
338: (0)         def _as_pairs(x, ndim, as_index=False):
339: (4)             """
340: (4)                 Broadcast `x` to an array with the shape (`ndim`, 2).
341: (4)                 A helper function for `pad` that prepares and validates arguments like
342: (4)                 `pad_width` for iteration in pairs.
343: (4)                 Parameters
344: (4)                 -----
345: (4)                 x : {None, scalar, array-like}
346: (8)                     The object to broadcast to the shape (`ndim`, 2).
347: (4)                 ndim : int
348: (8)                     Number of pairs the broadcasted `x` will have.
349: (4)                 as_index : bool, optional
350: (8)                     If `x` is not None, try to round each element of `x` to an integer
351: (8)                     (dtype `np.intp`) and ensure every element is positive.
352: (4)             Returns
353: (4)             -----
354: (4)             pairs : nested iterables, shape (`ndim`, 2)
355: (8)                 The broadcasted version of `x`.
356: (4)             Raises
357: (4)             -----
358: (4)             ValueError
359: (8)                 If `as_index` is True and `x` contains negative elements.
360: (8)                 Or if `x` is not broadcastable to the shape (`ndim`, 2).
361: (4)             """
362: (4)             if x is None:
363: (8)                 return ((None, None),) * ndim
364: (4)             x = np.array(x)
365: (4)             if as_index:
366: (8)                 x = np.round(x).astype(np.intp, copy=False)
367: (4)             if x.ndim < 3:
368: (8)                 if x.size == 1:
369: (12)                     x = x.ravel() # Ensure x[0] works for x.ndim == 0, 1, 2
370: (12)                     if as_index and x < 0:
371: (16)                         raise ValueError("index can't contain negative values")
372: (12)                     return ((x[0], x[0]),) * ndim
373: (8)                 if x.size == 2 and x.shape != (2, 1):
374: (12)                     x = x.ravel() # Ensure x[0], x[1] works
375: (12)                     if as_index and (x[0] < 0 or x[1] < 0):
376: (16)                         raise ValueError("index can't contain negative values")
377: (12)                     return ((x[0], x[1]),) * ndim
378: (4)                 if as_index and x.min() < 0:
379: (8)                     raise ValueError("index can't contain negative values")
380: (4)                 return np.broadcast_to(x, (ndim, 2)).tolist()
381: (0)             def _pad_dispatcher(array, pad_width, mode=None, **kwargs):
382: (4)                 return (array,)
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

383: (0)
384: (0)
385: (4)
386: (4)
387: (4)
388: (4)
389: (4)
390: (8)
391: (4)
392: (8)
393: (8)
394: (8)
395: (8)
396: (8)
397: (8)
398: (8)
399: (4)
400: (8)
401: (8)
402: (12)
403: (8)
404: (12)
405: (8)
406: (12)
407: (12)
408: (8)
409: (12)
410: (12)
411: (8)
412: (12)
413: (12)
414: (8)
415: (12)
416: (12)
417: (8)
418: (12)
419: (12)
420: (8)
421: (12)
422: (12)
423: (12)
424: (8)
425: (12)
426: (12)
427: (8)
428: (12)
429: (12)
430: (12)
431: (8)
432: (12)
433: (12)
434: (8)
435: (12)
436: (4)
437: (8)
438: (8)
439: (8)
440: (8)
441: (8)
442: (8)
443: (8)
444: (8)
445: (8)
446: (4)
447: (8)
448: (8)
449: (8)
constants
450: (8)

```

`@array_function_dispatch(_pad_dispatcher, module='numpy')`

`def pad(array, pad_width, mode='constant', **kwargs):`

`"""`

Pad an array.

Parameters

-----

array : array\_like of rank N

The array to pad.

pad\_width : {sequence, array\_like, int}

Number of values padded to the edges of each axis.

`((before\_1, after\_1), ... (before\_N, after\_N))` unique pad widths for each axis.

`((before, after)` or `((before, after),)` yields same before and after pad for each axis.

`((pad,)` or `int` is a shortcut for before = after = pad width for all axes.

mode : str or function, optional

One of the following string values or a user supplied function.

'constant' (default)

Pads with a constant value.

'edge'

Pads with the edge values of array.

'linear\_ramp'

Pads with the linear ramp between end\_value and the array edge value.

'maximum'

Pads with the maximum value of all or part of the vector along each axis.

'mean'

Pads with the mean value of all or part of the vector along each axis.

'median'

Pads with the median value of all or part of the vector along each axis.

'minimum'

Pads with the minimum value of all or part of the vector along each axis.

'reflect'

Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.

'symmetric'

Pads with the reflection of the vector mirrored along the edge of the array.

'wrap'

Pads with the wrap of the vector along the axis.

The first values are used to pad the end and the end values are used to pad the beginning.

'empty'

Pads with undefined values.

.. versionadded:: 1.17

<function>

Padding function, see Notes.

stat\_length : sequence or int, optional

Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value.

`((before\_1, after\_1), ... (before\_N, after\_N))` unique statistic lengths for each axis.

`((before, after)` or `((before, after),)` yields same before and after statistic lengths for each axis.

`((stat\_length,)` or `int` is a shortcut for `before = after = statistic` length for all axes.

Default is `None`, to use the entire axis.

constant\_values : sequence or scalar, optional

Used in 'constant'. The values to set the padded values for each axis.

`((before\_1, after\_1), ... (before\_N, after\_N))` unique pad for each axis.

```

451: (8) ````(before, after)```` or ````((before, after),)```` yields same before
452: (8) and after constants for each axis.
453: (8) ````(constant,)```` or ````constant```` is a shortcut for
454: (8) ````before = after = constant```` for all axes.
455: (8) Default is 0.
456: (4) end_values : sequence or scalar, optional
457: (8) Used in 'linear_ramp'. The values used for the ending value of the
458: (8) linear_ramp and that will form the edge of the padded array.
459: (8) ````((before_1, after_1), ... (before_N, after_N))```` unique end values
460: (8) for each axis.
461: (8) ````(before, after)```` or ````((before, after),)```` yields same before
462: (8) and after end values for each axis.
463: (8) ````(constant,)```` or ````constant```` is a shortcut for
464: (8) ````before = after = constant```` for all axes.
465: (8) Default is 0.
466: (4) reflect_type : {'even', 'odd'}, optional
467: (8) Used in 'reflect', and 'symmetric'. The 'even' style is the
468: (8) default with an unaltered reflection around the edge value. For
469: (8) the 'odd' style, the extended part of the array is created by
470: (8) subtracting the reflected values from two times the edge value.
471: (4) Returns
472: (4)
473: (4) pad : ndarray
474: (8) Padded array of rank equal to `array` with shape increased
475: (8) according to `pad_width`.
476: (4) Notes
477: (4)
478: (4) .. versionadded:: 1.7.0
479: (4) For an array with rank greater than 1, some of the padding of later
480: (4) axes is calculated from padding of previous axes. This is easiest to
481: (4) think about with a rank 2 array where the corners of the padded array
482: (4) are calculated by using padded values from the first axis.
483: (4) The padding function, if used, should modify a rank 1 array in-place. It
484: (4) has the following signature::
485: (8)     padding_func(vector, iaxis_pad_width, iaxis, kwargs)
486: (4) where
487: (8)     vector : ndarray
488: (12)    A rank 1 array already padded with zeros. Padded values are
489: (12)    vector[:iaxis_pad_width[0]] and vector[-iaxis_pad_width[1]:].
490: (8)     iaxis_pad_width : tuple
491: (12)        A 2-tuple of ints, iaxis_pad_width[0] represents the number of
492: (12)        values padded at the beginning of vector where
493: (12)        iaxis_pad_width[1] represents the number of values padded at
494: (12)        the end of vector.
495: (8)     iaxis : int
496: (12)        The axis currently being calculated.
497: (8)     kwargs : dict
498: (12)        Any keyword arguments the function requires.
499: (4) Examples
500: (4)
501: (4) >>> a = [1, 2, 3, 4, 5]
502: (4) >>> np.pad(a, (2, 3), 'constant', constant_values=(4, 6))
503: (4) array([4, 4, 1, ..., 6, 6, 6])
504: (4) >>> np.pad(a, (2, 3), 'edge')
505: (4) array([1, 1, 1, ..., 5, 5, 5])
506: (4) >>> np.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
507: (4) array([ 5,  3,  1,  2,  3,  4,  5,  2, -1, -4])
508: (4) >>> np.pad(a, (2,), 'maximum')
509: (4) array([5, 5, 1, 2, 3, 4, 5, 5, 5])
510: (4) >>> np.pad(a, (2,), 'mean')
511: (4) array([3, 3, 1, 2, 3, 4, 5, 3, 3])
512: (4) >>> np.pad(a, (2,), 'median')
513: (4) array([3, 3, 1, 2, 3, 4, 5, 3, 3])
514: (4) >>> a = [[1, 2], [3, 4]]
515: (4) >>> np.pad(a, ((3, 2), (2, 3)), 'minimum')
516: (4) array([[1, 1, 1, 2, 1, 1, 1],
517: (11)           [1, 1, 1, 2, 1, 1, 1],
518: (11)           [1, 1, 1, 2, 1, 1, 1],
519: (11)           [1, 1, 1, 2, 1, 1, 1],

```

```

520: (11)
521: (11)
522: (11)
523: (4)
524: (4)
525: (4)
526: (4)
527: (4)
528: (4)
529: (4)
530: (4)
531: (4)
532: (4)
533: (4)
534: (4)
535: (4)
536: (4)
537: (4)
538: (4)
539: (4)
540: (4)
541: (4)
542: (11)
543: (11)
544: (11)
545: (11)
546: (11)
547: (4)
548: (4)
549: (11)
550: (11)
551: (11)
552: (11)
553: (11)
554: (4)
555: (4)
556: (4)
557: (4)
558: (8)
559: (4)
560: (4)
561: (8)
562: (8)
563: (8)
564: (12)
565: (12)
566: (12)
567: (12)
568: (16)
569: (8)
570: (4)
571: (8)
572: (8)
573: (8)
574: (8)
575: (8)
576: (8)
577: (8)
578: (8)
579: (8)
580: (4)
581: (4)
582: (8)
583: (4)
584: (8)
585: (4)
586: (8)
587: (25)
588: (4)

[3, 3, 3, 4, 3, 3, 3],
[1, 1, 1, 2, 1, 1, 1],
[1, 1, 1, 2, 1, 1, 1])
>>> a = [1, 2, 3, 4, 5]
>>> np.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])
>>> np.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1, 0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> np.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])
>>> np.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])
>>> np.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
>>> def pad_with(vector, pad_width, iaxis, kwargs):
...     pad_value = kwargs.get('padder', 10)
...     vector[:pad_width[0]] = pad_value
...     vector[-pad_width[1]:] = pad_value
>>> a = np.arange(6)
>>> a = a.reshape((2, 3))
>>> np.pad(a, 2, pad_with)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 0, 1, 2, 10, 10],
       [10, 10, 3, 4, 5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])
>>> np.pad(a, 2, pad_with, padder=100)
array([[100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 0, 1, 2, 100, 100],
       [100, 100, 3, 4, 5, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100]])
"""
array = np.asarray(array)
pad_width = np.asarray(pad_width)
if not pad_width.dtype.kind == 'i':
    raise TypeError(`pad_width` must be of integral type.')
pad_width = _as_pairs(pad_width, array.ndim, as_index=True)
if callable(mode):
    function = mode
    padded, _ = _pad_simple(array, pad_width, fill_value=0)
    for axis in range(padded.ndim):
        view = np.moveaxis(padded, axis, -1)
        inds = ndindex(view.shape[:-1])
        inds = (ind + (Ellipsis,) for ind in inds)
        for ind in inds:
            function(view[ind], pad_width[axis], axis, kwargs)
    return padded
allowed_kwargs = {
    'empty': [], 'edge': [], 'wrap': [],
    'constant': ['constant_values'],
    'linear_ramp': ['end_values'],
    'maximum': ['stat_length'],
    'mean': ['stat_length'],
    'median': ['stat_length'],
    'minimum': ['stat_length'],
    'reflect': ['reflect_type'],
    'symmetric': ['reflect_type'],
}
try:
    unsupported_kwargs = set(kwargs) - set(allowed_kwargs[mode])
except KeyError:
    raise ValueError("mode '{}' is not supported".format(mode)) from None
if unsupported_kwargs:
    raise ValueError("unsupported keyword arguments for mode '{}': {}"
                     .format(mode, unsupported_kwargs))
stat_functions = {"maximum": npamax, "minimum": npamin,

```

```

589: (22)                         "mean": np.mean, "median": np.median}
590: (4)                          padded, original_area_slice = _pad_simple(array, pad_width)
591: (4)                          axes = range(padded.ndim)
592: (4)                          if mode == "constant":
593: (8)                            values = kwargs.get("constant_values", 0)
594: (8)                            values = _as_pairs(values, padded.ndim)
595: (8)                            for axis, width_pair, value_pair in zip(axes, pad_width, values):
596: (12)                              roi = _view_roi(padded, original_area_slice, axis)
597: (12)                              _set_pad_area(roi, axis, width_pair, value_pair)
598: (4)                          elif mode == "empty":
599: (8)                            pass # Do nothing as _pad_simple already returned the correct result
600: (4)                          elif array.size == 0:
601: (8)                            for axis, width_pair in zip(axes, pad_width):
602: (12)                              if array.shape[axis] == 0 and any(width_pair):
603: (16)                                raise ValueError(
604: (20)                                  "can't extend empty axis {} using modes other than "
605: (20)                                  "'constant' or 'empty'".format(axis)
606: (16)                                )
607: (4)                          elif mode == "edge":
608: (8)                            for axis, width_pair in zip(axes, pad_width):
609: (12)                              roi = _view_roi(padded, original_area_slice, axis)
610: (12)                              edge_pair = _get_edges(roi, axis, width_pair)
611: (12)                              _set_pad_area(roi, axis, width_pair, edge_pair)
612: (4)                          elif mode == "linear_ramp":
613: (8)                            end_values = kwargs.get("end_values", 0)
614: (8)                            end_values = _as_pairs(end_values, padded.ndim)
615: (8)                            for axis, width_pair, value_pair in zip(axes, pad_width, end_values):
616: (12)                              roi = _view_roi(padded, original_area_slice, axis)
617: (12)                              ramp_pair = _get_linear_ramps(roi, axis, width_pair, value_pair)
618: (12)                              _set_pad_area(roi, axis, width_pair, ramp_pair)
619: (4)                          elif mode in stat_functions:
620: (8)                            func = stat_functions[mode]
621: (8)                            length = kwargs.get("stat_length", None)
622: (8)                            length = _as_pairs(length, padded.ndim, as_index=True)
623: (8)                            for axis, width_pair, length_pair in zip(axes, pad_width, length):
624: (12)                              roi = _view_roi(padded, original_area_slice, axis)
625: (12)                              stat_pair = _get_stats(roi, axis, width_pair, length_pair, func)
626: (12)                              _set_pad_area(roi, axis, width_pair, stat_pair)
627: (4)                          elif mode in {"reflect", "symmetric"}:
628: (8)                            method = kwargs.get("reflect_type", "even")
629: (8)                            include_edge = True if mode == "symmetric" else False
630: (8)                            for axis, (left_index, right_index) in zip(axes, pad_width):
631: (12)                              if array.shape[axis] == 1 and (left_index > 0 or right_index > 0):
632: (16)                                edge_pair = _get_edges(padded, axis, (left_index,
right_index))
633: (16)
634: (20)                                _set_pad_area(
635: (16)                                  padded, axis, (left_index, right_index), edge_pair)
636: (16)                                continue
637: (12)                                roi = _view_roi(padded, original_area_slice, axis)
638: (16)                                while left_index > 0 or right_index > 0:
639: (20)                                  left_index, right_index = _set_reflect_both(
640: (20)                                      roi, axis, (left_index, right_index),
641: (16)                                      method, include_edge
642: (4)                                )
643: (4)                          elif mode == "wrap":
644: (8)                            for axis, (left_index, right_index) in zip(axes, pad_width):
645: (12)                              roi = _view_roi(padded, original_area_slice, axis)
646: (12)                              original_period = padded.shape[axis] - right_index - left_index
647: (16)                              while left_index > 0 or right_index > 0:
648: (20)                                left_index, right_index = _set_wrap_both(
649: (20)                                      roi, axis, (left_index, right_index), original_period)
649: (4)                          return padded

```

-----  
File 205 - arraysetops.py:

```

1: (0)      """
2: (0)      Set operations for arrays based on sorting.

```

```

3: (0)          Notes
4: (0)          -----
5: (0)          For floating point arrays, inaccurate results may appear due to usual round-
off
6: (0)          and floating point comparison issues.
7: (0)          Speed could be gained in some operations by an implementation of
8: (0)          `numpy.sort`, that can provide directly the permutation vectors, thus avoiding
9: (0)          calls to `numpy.argsort`.
10: (0)         Original author: Robert Cimrman
11: (0)         """
12: (0)         import functools
13: (0)         import numpy as np
14: (0)         from numpy.core import overrides
15: (0)         array_function_dispatch = functools.partial(
16: (4)             overrides.array_function_dispatch, module='numpy')
17: (0)         __all__ = [
18: (4)             'ediff1d', 'intersect1d', 'setxor1d', 'union1d', 'setdiff1d', 'unique',
19: (4)             'in1d', 'isin'
20: (4)         ]
21: (0)         def _ediff1d_dispatcher(ary, to_end=None, to_begin=None):
22: (4)             return (ary, to_end, to_begin)
23: (0)         @array_function_dispatch(_ediff1d_dispatcher)
24: (0)         def ediff1d(ary, to_end=None, to_begin=None):
25: (4)             """
26: (4)             The differences between consecutive elements of an array.
27: (4)             Parameters
28: (4)             -----
29: (4)             ary : array_like
30: (8)                 If necessary, will be flattened before the differences are taken.
31: (4)             to_end : array_like, optional
32: (8)                 Number(s) to append at the end of the returned differences.
33: (4)             to_begin : array_like, optional
34: (8)                 Number(s) to prepend at the beginning of the returned differences.
35: (4)             Returns
36: (4)             -----
37: (4)             ediff1d : ndarray
38: (8)                 The differences. Loosely, this is ``ary.flat[1:] - ary.flat[:-1]``.
39: (4)             See Also
40: (4)             -----
41: (4)             diff, gradient
42: (4)             Notes
43: (4)             -----
44: (4)             When applied to masked arrays, this function drops the mask information
45: (4)             if the `to_begin` and/or `to_end` parameters are used.
46: (4)             Examples
47: (4)             -----
48: (4)             >>> x = np.array([1, 2, 4, 7, 0])
49: (4)             >>> np.ediff1d(x)
50: (4)             array([ 1,  2,  3, -7])
51: (4)             >>> np.ediff1d(x, to_begin=-99, to_end=np.array([88, 99]))
52: (4)             array([-99,  1,  2, ..., -7,  88,  99])
53: (4)             The returned array is always 1D.
54: (4)             >>> y = [[1, 2, 4], [1, 6, 24]]
55: (4)             >>> np.ediff1d(y)
56: (4)             array([ 1,  2, -3,  5, 18])
57: (4)             """
58: (4)             ary = np.asanyarray(ary).ravel()
59: (4)             dtype_req = ary.dtype
60: (4)             if to_begin is None and to_end is None:
61: (8)                 return ary[1:] - ary[:-1]
62: (4)             if to_begin is None:
63: (8)                 l_begin = 0
64: (4)             else:
65: (8)                 to_begin = np.asanyarray(to_begin)
66: (8)                 if not np.can_cast(to_begin, dtype_req, casting="same_kind"):
67: (12)                     raise TypeError("dtype of `to_begin` must be compatible "
68: (28)                         "with input `ary` under the `same_kind` rule.")
69: (8)                 to_begin = to_begin.ravel()
70: (8)                 l_begin = len(to_begin)

```

```

71: (4)           if to_end is None:
72: (8)             l_end = 0
73: (4)           else:
74: (8)             to_end = np.asarray(to_end)
75: (8)             if not np.can_cast(to_end, dtype_req, casting="same_kind"):
76: (12)               raise TypeError("dtype of `to_end` must be compatible "
77: (28)                 "with input `ary` under the `same_kind` rule.")
78: (8)             to_end = to_end.ravel()
79: (8)             l_end = len(to_end)
80: (4)             l_diff = max(len(ary) - 1, 0)
81: (4)             result = np.empty(l_diff + l_begin + l_end, dtype=ary.dtype)
82: (4)             result = ary.__array_wrap__(result)
83: (4)             if l_begin > 0:
84: (8)               result[:l_begin] = to_begin
85: (4)             if l_end > 0:
86: (8)               result[l_begin + l_diff:] = to_end
87: (4)             np.subtract(ary[1:], ary[:-1], result[l_begin:l_begin + l_diff])
88: (4)             return result
89: (0)           def _unpack_tuple(x):
90: (4)             """ Unpacks one-element tuples for use as return values """
91: (4)             if len(x) == 1:
92: (8)               return x[0]
93: (4)             else:
94: (8)               return x
95: (0)           def _unique_dispatcher(ar, return_index=None, return_inverse=None,
96: (23)                         return_counts=None, axis=None, *, equal_nan=None):
97: (4)             return (ar,)
98: (0)           @array_function_dispatch(_unique_dispatcher)
99: (0)           def unique(ar, return_index=False, return_inverse=False,
100: (11)             return_counts=False, axis=None, *, equal_nan=True):
101: (4)             """
102: (4)             Find the unique elements of an array.
103: (4)             Returns the sorted unique elements of an array. There are three optional
104: (4)             outputs in addition to the unique elements:
105: (4)             * the indices of the input array that give the unique values
106: (4)             * the indices of the unique array that reconstruct the input array
107: (4)             * the number of times each unique value comes up in the input array
108: (4)             Parameters
109: (4)             -----
110: (4)             ar : array_like
111: (8)               Input array. Unless `axis` is specified, this will be flattened if it
112: (8)                 is not already 1-D.
113: (4)             return_index : bool, optional
114: (8)               If True, also return the indices of `ar` (along the specified axis,
115: (8)                 if provided, or in the flattened array) that result in the unique
array.
116: (4)             return_inverse : bool, optional
117: (8)               If True, also return the indices of the unique array (for the
specified
118: (8)                 axis, if provided) that can be used to reconstruct `ar`.
119: (4)             return_counts : bool, optional
120: (8)               If True, also return the number of times each unique item appears
in `ar`.
121: (8)             axis : int or None, optional
122: (4)               The axis to operate on. If None, `ar` will be flattened. If an
integer,
123: (8)                 the subarrays indexed by the given axis will be flattened and treated
124: (8)                   as the elements of a 1-D array with the dimension of the given axis,
125: (8)                   see the notes for more details. Object arrays or structured arrays
126: (8)                   that contain objects are not supported if the `axis` kwarg is used.
The
128: (8)               default is None.
129: (8)               .. versionadded:: 1.13.0
130: (4)             equal_nan : bool, optional
131: (8)               If True, collapses multiple NaN values in the return array into one.
132: (8)               .. versionadded:: 1.24
133: (4)             Returns
134: (4)             -----
135: (4)             unique : ndarray

```

```

136: (8)           The sorted unique values.
137: (4)           unique_indices : ndarray, optional
138: (8)           The indices of the first occurrences of the unique values in the
139: (8)           original array. Only provided if `return_index` is True.
140: (4)           unique_inverse : ndarray, optional
141: (8)           The indices to reconstruct the original array from the
142: (8)           unique array. Only provided if `return_inverse` is True.
143: (4)           unique_counts : ndarray, optional
144: (8)           The number of times each of the unique values comes up in the
145: (8)           original array. Only provided if `return_counts` is True.
146: (8)           .. versionadded:: 1.9.0
147: (4)           See Also
148: (4)           -----
149: (4)           numpy.lib.arraysetops : Module with a number of other functions for
150: (28)          performing set operations on arrays.
151: (4)           repeat : Repeat elements of an array.
152: (4)           Notes
153: (4)           -----
154: (4)           When an axis is specified the subarrays indexed by the axis are sorted.
155: (4)           This is done by making the specified axis the first dimension of the array
156: (4)           (move the axis to the first dimension to keep the order of the other axes)
157: (4)           and then flattening the subarrays in C order. The flattened subarrays are
158: (4)           then viewed as a structured type with each element given a label, with the
159: (4)           effect that we end up with a 1-D array of structured types that can be
160: (4)           treated in the same way as any other 1-D array. The result is that the
161: (4)           flattened subarrays are sorted in lexicographic order starting with the
162: (4)           first element.
163: (4)           .. versionchanged: NumPy 1.21
164: (8)           If nan values are in the input array, a single nan is put
165: (8)           to the end of the sorted unique values.
166: (8)           Also for complex arrays all NaN values are considered equivalent
167: (8)           (no matter whether the NaN is in the real or imaginary part).
168: (8)           As the representant for the returned array the smallest one in the
169: (8)           lexicographical order is chosen - see np.sort for how the
lexicographical
170: (8)           order is defined for complex arrays.
171: (4)           Examples
172: (4)           -----
173: (4)           >>> np.unique([1, 1, 2, 2, 3, 3])
174: (4)           array([1, 2, 3])
175: (4)           >>> a = np.array([[1, 1], [2, 3]])
176: (4)           >>> np.unique(a)
177: (4)           array([1, 2, 3])
178: (4)           Return the unique rows of a 2D array
179: (4)           >>> a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
180: (4)           >>> np.unique(a, axis=0)
181: (4)           array([[1, 0, 0], [2, 3, 4]])
182: (4)           Return the indices of the original array that give the unique values:
183: (4)           >>> a = np.array(['a', 'b', 'b', 'c', 'a'])
184: (4)           >>> u, indices = np.unique(a, return_index=True)
185: (4)           >>> u
186: (4)           array(['a', 'b', 'c'], dtype='<U1')
187: (4)           >>> indices
188: (4)           array([0, 1, 3])
189: (4)           >>> a[indices]
190: (4)           array(['a', 'b', 'c'], dtype='<U1')
191: (4)           Reconstruct the input array from the unique values and inverse:
192: (4)           >>> a = np.array([1, 2, 6, 4, 2, 3, 2])
193: (4)           >>> u, indices = np.unique(a, return_inverse=True)
194: (4)           >>> u
195: (4)           array([1, 2, 3, 4, 6])
196: (4)           >>> indices
197: (4)           array([0, 1, 4, 3, 1, 2, 1])
198: (4)           >>> u[indices]
199: (4)           array([1, 2, 6, 4, 2, 3, 2])
200: (4)           Reconstruct the input values from the unique values and counts:
201: (4)           >>> a = np.array([1, 2, 6, 4, 2, 3, 2])
202: (4)           >>> values, counts = np.unique(a, return_counts=True)
203: (4)           >>> values

```

```

204: (4)                                array([1, 2, 3, 4, 6])
205: (4)                                >>> counts
206: (4)                                array([1, 3, 1, 1, 1])
207: (4)                                >>> np.repeat(values, counts)
208: (4)                                array([1, 2, 2, 2, 3, 4, 6])    # original order not preserved
209: (4)                                """
210: (4)                                ar = np.asarray(ar)
211: (4)                                if axis is None:
212: (8)                                    ret = _unique1d(ar, return_index, return_inverse, return_counts,
213: (24)   equal_nan=equal_nan)
214: (8)                                    return _unpack_tuple(ret)
215: (4)                                try:
216: (8)                                    ar = np.moveaxis(ar, axis, 0)
217: (4)                                except np.AxisError:
218: (8)                                    raise np.AxisError(axis, ar.ndim) from None
219: (4)                                orig_shape, orig_dtype = ar.shape, ar.dtype
220: (4)                                ar = ar.reshape(orig_shape[0], np.prod(orig_shape[1:], dtype=np.intp))
221: (4)                                ar = np.ascontiguousarray(ar)
222: (4)                                dtype = [('{f}{i}'.format(i=i), ar.dtype) for i in range(ar.shape[1])]
223: (4)                                try:
224: (8)                                    if ar.shape[1] > 0:
225: (12)  consolidated = ar.view(dtype)
226: (8)                                    else:
227: (12)  consolidated = np.empty(len(ar), dtype=dtype)
228: (4)                                except TypeError as e:
229: (8)                                    msg = 'The axis argument to unique is not supported for dtype {dt}''.
230: (8)                                    raise TypeError(msg.format(dt=ar.dtype)) from e
231: (4)                                def reshape_uniq(uniq):
232: (8)                                    n = len(uniq)
233: (8)                                    uniq = uniq.view(orig_dtype)
234: (8)                                    uniq = uniq.reshape(n, *orig_shape[1:])
235: (8)                                    uniq = np.moveaxis(uniq, 0, axis)
236: (8)                                    return uniq
237: (4)                                output = _unique1d(consolidated, return_index,
238: (23)                                     return_inverse, return_counts, equal_nan=equal_nan)
239: (4)                                output = (reshape_uniq(output[0]),) + output[1:]
240: (4)                                return _unpack_tuple(output)
241: (0)                                def _unique1d(ar, return_index=False, return_inverse=False,
242: (14)                                     return_counts=False, *, equal_nan=True):
243: (4)                                """
244: (4)                                    Find the unique elements of an array, ignoring shape.
245: (4)                                """
246: (4)                                ar = np.asarray(ar).flatten()
247: (4)                                optional_indices = return_index or return_inverse
248: (4)                                if optional_indices:
249: (8)                                    perm = ar.argsort(kind='mergesort' if return_index else 'quicksort')
250: (8)                                    aux = ar[perm]
251: (4)                                else:
252: (8)                                    ar.sort()
253: (8)                                    aux = ar
254: (4)                                mask = np.empty(aux.shape, dtype=np.bool_)
255: (4)                                mask[:1] = True
256: (4)                                if (equal_nan and aux.shape[0] > 0 and aux.dtype.kind in "cfmM" and
257: (12)                                     np.isnan(aux[-1])):
258: (8)                                    if aux.dtype.kind == "c": # for complex all NaNs are considered
259: (12)  aux_firstnan = np.searchsorted(np.isnan(aux), True, side='left')
260: (8)                                    else:
261: (12)  aux_firstnan = np.searchsorted(aux, aux[-1], side='left')
262: (8)                                    if aux_firstnan > 0:
263: (12)  mask[1:aux_firstnan] = (
264: (16)  aux[1:aux_firstnan] != aux[:aux_firstnan - 1])
265: (8)  mask[aux_firstnan] = True
266: (8)  mask[aux_firstnan + 1:] = False
267: (4)                                    else:
268: (8)  mask[1:] = aux[1:] != aux[:-1]
269: (4)                                    ret = (aux[mask],)
270: (4)                                    if return_index:
271: (8)  ret += (perm[mask],)

```

```

272: (4)             if return_inverse:
273: (8)                 imask = np.cumsum(mask) - 1
274: (8)                 inv_idx = np.empty(mask.shape, dtype=np.intp)
275: (8)                 inv_idx[perm] = imask
276: (8)                 ret += (inv_idx,)
277: (4)             if return_counts:
278: (8)                 idx = np.concatenate(np.nonzero(mask) + ([mask.size],))
279: (8)                 ret += (np.diff(idx),)
280: (4)             return ret
281: (0)         def _intersect1d_dispatcher(
282: (8)             ar1, ar2, assume_unique=None, return_indices=None):
283: (4)             return (ar1, ar2)
284: (0)     @array_function_dispatch(_intersect1d_dispatcher)
285: (0)     def intersect1d(ar1, ar2, assume_unique=False, return_indices=False):
286: (4)             """
287: (4)             Find the intersection of two arrays.
288: (4)             Return the sorted, unique values that are in both of the input arrays.
289: (4)             Parameters
290: (4)             -----
291: (4)             ar1, ar2 : array_like
292: (8)                 Input arrays. Will be flattened if not already 1D.
293: (4)             assume_unique : bool
294: (8)                 If True, the input arrays are both assumed to be unique, which
295: (8)                 can speed up the calculation. If True but ``ar1`` or ``ar2`` are not
296: (8)                 unique, incorrect results and out-of-bounds indices could result.
297: (8)                 Default is False.
298: (4)             return_indices : bool
299: (8)                 If True, the indices which correspond to the intersection of the two
300: (8)                 arrays are returned. The first instance of a value is used if there
are
301: (8)                 multiple. Default is False.
302: (8)                 .. versionadded:: 1.15.0
303: (4)             Returns
304: (4)             -----
305: (4)             intersect1d : ndarray
306: (8)                 Sorted 1D array of common and unique elements.
307: (4)             comm1 : ndarray
308: (8)                 The indices of the first occurrences of the common values in `ar1`.
309: (8)                 Only provided if `return_indices` is True.
310: (4)             comm2 : ndarray
311: (8)                 The indices of the first occurrences of the common values in `ar2`.
312: (8)                 Only provided if `return_indices` is True.
313: (4)             See Also
314: (4)             -----
315: (4)             numpy.lib.arraysetops : Module with a number of other functions for
316: (28)                 performing set operations on arrays.
317: (4)             Examples
318: (4)             -----
319: (4)             >>> np.intersect1d([1, 3, 4, 3], [3, 1, 2, 1])
320: (4)                 array([1, 3])
321: (4)                 To intersect more than two arrays, use functools.reduce:
322: (4)                 >>> from functools import reduce
323: (4)                 >>> reduce(np.intersect1d, ([1, 3, 4, 3], [3, 1, 2, 1], [6, 3, 4, 2]))
324: (4)                 array([3])
325: (4)                 To return the indices of the values common to the input arrays
326: (4)                 along with the intersected values:
327: (4)                 >>> x = np.array([1, 1, 2, 3, 4])
328: (4)                 >>> y = np.array([2, 1, 4, 6])
329: (4)                 >>> xy, x_ind, y_ind = np.intersect1d(x, y, return_indices=True)
330: (4)                 >>> x_ind, y_ind
331: (4)                 (array([0, 2, 4]), array([1, 0, 2]))
332: (4)                 >>> xy, x[x_ind], y[y_ind]
333: (4)                 (array([1, 2, 4]), array([1, 2, 4]), array([1, 2, 4]))
334: (4)                 """
335: (4)                 ar1 = np.asarray(ar1)
336: (4)                 ar2 = np.asarray(ar2)
337: (4)                 if not assume_unique:
338: (8)                     if return_indices:
339: (12)                         ar1, ind1 = np.unique(ar1, return_index=True)

```

```

340: (12)                                ar2, ind2 = unique(ar2, return_index=True)
341: (8)                                 else:
342: (12)                               ar1 = unique(ar1)
343: (12)                               ar2 = unique(ar2)
344: (4)                                 else:
345: (8)                               ar1 = ar1.ravel()
346: (8)                               ar2 = ar2.ravel()
347: (4)                               aux = np.concatenate((ar1, ar2))
348: (4)                               if return_indices:
349: (8)                                 aux_sort_indices = np.argsort(aux, kind='mergesort')
350: (8)                                 aux = aux[aux_sort_indices]
351: (4)                               else:
352: (8)                                 aux.sort()
353: (4)                               mask = aux[1:] == aux[:-1]
354: (4)                               int1d = aux[:-1][mask]
355: (4)                               if return_indices:
356: (8)                                 ar1_indices = aux_sort_indices[:-1][mask]
357: (8)                                 ar2_indices = aux_sort_indices[1:][mask] - ar1.size
358: (8)                                 if not assume_unique:
359: (12)                                   ar1_indices = ind1[ar1_indices]
360: (12)                                   ar2_indices = ind2[ar2_indices]
361: (8)                               return int1d, ar1_indices, ar2_indices
362: (4)                               else:
363: (8)                                 return int1d
364: (0)                               def _setxor1d_dispatcher(ar1, ar2, assume_unique=None):
365: (4)                                 return (ar1, ar2)
366: (0)                               @array_function_dispatch(_setxor1d_dispatcher)
367: (0)                               def setxor1d(ar1, ar2, assume_unique=False):
368: (4)                                 """
369: (4)                                 Find the set exclusive-or of two arrays.
370: (4)                                 Return the sorted, unique values that are in only one (not both) of the
371: (4)                                 input arrays.
372: (4)                                 Parameters
373: (4)                                 -----
374: (4)                                 ar1, ar2 : array_like
375: (8)                                   Input arrays.
376: (4)                                 assume_unique : bool
377: (8)                                   If True, the input arrays are both assumed to be unique, which
378: (8)                                   can speed up the calculation. Default is False.
379: (4)                                 Returns
380: (4)                                 -----
381: (4)                                 setxor1d : ndarray
382: (8)                                   Sorted 1D array of unique values that are in only one of the input
383: (8)                                   arrays.
384: (4)                                 Examples
385: (4)                                 -----
386: (4)                                 >>> a = np.array([1, 2, 3, 2, 4])
387: (4)                                 >>> b = np.array([2, 3, 5, 7, 5])
388: (4)                                 >>> np.setxor1d(a,b)
389: (4)                                 array([1, 4, 5, 7])
390: (4)                                 """
391: (4)                                 if not assume_unique:
392: (8)                                   ar1 = unique(ar1)
393: (8)                                   ar2 = unique(ar2)
394: (4)                                   aux = np.concatenate((ar1, ar2))
395: (4)                                   if aux.size == 0:
396: (8)                                     return aux
397: (4)                                   aux.sort()
398: (4)                                   flag = np.concatenate(([True], aux[1:] != aux[:-1], [True]))
399: (4)                                   return aux[flag[1:] & flag[:-1]]
400: (0)                               def _in1d_dispatcher(ar1, ar2, assume_unique=None, invert=False, *, kind=None):
401: (21)                                 return (ar1, ar2)
402: (4)                               @array_function_dispatch(_in1d_dispatcher)
403: (0)                               def in1d(ar1, ar2, assume_unique=False, invert=False, *, kind=None):
404: (0)                                 """
405: (4)                                 Test whether each element of a 1-D array is also present in a second
406: (4)                                 array.
407: (4)                                 Returns a boolean array the same length as `ar1` that is True

```

```

408: (4) where an element of `ar1` is in `ar2` and False otherwise.
409: (4) We recommend using :func:`isin` instead of `in1d` for new code.
410: (4) Parameters
411: (4)
412: (4) ar1 : (M,) array_like
413: (8) Input array.
414: (4) ar2 : array_like
415: (8) The values against which to test each value of `ar1`.
416: (4) assume_unique : bool, optional
417: (8) If True, the input arrays are both assumed to be unique, which
418: (8) can speed up the calculation. Default is False.
419: (4) invert : bool, optional
420: (8) If True, the values in the returned array are inverted (that is,
421: (8) False where an element of `ar1` is in `ar2` and True otherwise).
422: (8) Default is False. ``np.in1d(a, b, invert=True)`` is equivalent
423: (8) to (but is faster than) ``np.invert(np.in1d(a, b))``.
424: (4) kind : {None, 'sort', 'table'}, optional
425: (8) The algorithm to use. This will not affect the final result,
426: (8) but will affect the speed and memory use. The default, None,
427: (8) will select automatically based on memory considerations.
428: (8)
429: (10) * If 'sort', will use a mergesort-based approach. This will have
430: (10) a memory usage of roughly 6 times the sum of the sizes of
431: (8) `ar1` and `ar2`, not accounting for size of dtypes.
432: (10) * If 'table', will use a lookup table approach similar
433: (10) to a counting sort. This is only available for boolean and
434: (10) integer arrays. This will have a memory usage of the
435: (10) size of `ar1` plus the max-min value of `ar2`. `assume_unique`
436: (8) has no effect when the 'table' option is used.
437: (10) * If None, will automatically choose 'table' if
438: (10) the required memory allocation is less than or equal to
439: (10) 6 times the sum of the sizes of `ar1` and `ar2`,
440: (10) otherwise will use 'sort'. This is done to not use
441: (10) a large amount of memory by default, even though
442: (10) 'table' may be faster in most cases. If 'table' is chosen,
443: (8) `assume_unique` will have no effect.
444: (4) .. versionadded:: 1.8.0
445: (4) Returns
446: (4)
447: (8) in1d : (M,) ndarray, bool
448: (4) The values `ar1[in1d]` are in `ar2`.
449: (4) See Also
450: (4) isin : Version of this function that preserves the
451: (28) shape of ar1.
452: (4) numpy.lib.arraysetops : Module with a number of other functions for
453: (28) performing set operations on arrays.
454: (4) Notes
455: (4)
456: (4) `in1d` can be considered as an element-wise function version of the
457: (4) python keyword `in`, for 1-D sequences. ``in1d(a, b)`` is roughly
458: (4) equivalent to ``np.array([item in b for item in a])``.
459: (4) However, this idea fails if `ar2` is a set, or similar (non-sequence)
460: (4) container: As ``ar2`` is converted to an array, in those cases
461: (4) ``asarray(ar2)`` is an object array rather than the expected array of
462: (4) contained values.
463: (4) Using ``kind='table'`` tends to be faster than `kind='sort'` if the
464: (4) following relationship is true:
465: (4) ``log10(len(ar2)) > (log10(max(ar2)-min(ar2)) - 2.27) / 0.927``,
466: (4) but may use greater memory. The default value for `kind` will
467: (4) be automatically selected based only on memory usage, so one may
468: (4) manually set ``kind='table'`` if memory constraints can be relaxed.
469: (4) .. versionadded:: 1.4.0
470: (4) Examples
471: (4)
472: (4) >>> test = np.array([0, 1, 2, 5, 0])
473: (4) >>> states = [0, 2]
474: (4) >>> mask = np.in1d(test, states)
475: (4) >>> mask
476: (4) array([ True, False,  True, False,  True])

```

```

477: (4)
478: (4)
479: (4)
480: (4)
481: (4)
482: (4)
483: (4)
484: (4)
485: (4)
486: (4)
487: (4)
488: (8)
489: (4)
490: (8)
491: (12)
492: (4)
493: (4)
494: (4)
495: (8)
496: (12)
497: (16)
498: (12)
499: (16)
500: (8)
501: (12)
502: (8)
503: (12)
504: (8)
505: (8)
506: (8)
507: (8)
508: (8)
509: (8)
510: (12)
511: (12)
512: (12)
513: (12)
514: (12)
515: (16)
516: (16)
517: (12)
518: (8)
519: (12)
520: (12)
521: (8)
522: (12)
523: (16)
524: (12)
525: (16)
526: (12)
527: (16)
528: (16)
529: (12)
530: (16)
531: (16)
532: (12)
533: (12)
534: (56)
535: (12)
536: (8)
537: (12)
538: (16)
539: (16)
540: (16)
541: (16)
542: (12)
543: (4)
544: (8)
545: (12)

        >>> test[mask]
        array([0, 2, 0])
        >>> mask = np.in1d(test, states, invert=True)
        >>> mask
        array([False, True, False, True, False])
        >>> test[mask]
        array([1, 5])
        """
        ar1 = np.asarray(ar1).ravel()
        ar2 = np.asarray(ar2).ravel()
        if ar2.dtype == object:
            ar2 = ar2.reshape(-1, 1)
        if kind not in {None, 'sort', 'table'}:
            raise ValueError(
                f"Invalid kind: '{kind}'. Please use None, 'sort' or 'table'.")
        is_int_arrays = all(ar.dtype.kind in ("u", "i", "b") for ar in (ar1, ar2))
        use_table_method = is_int_arrays and kind in {None, 'table'}
        if use_table_method:
            if ar2.size == 0:
                if invert:
                    return np.ones_like(ar1, dtype=bool)
                else:
                    return np.zeros_like(ar1, dtype=bool)
            if ar1.dtype == bool:
                ar1 = ar1.astype(np.uint8)
            if ar2.dtype == bool:
                ar2 = ar2.astype(np.uint8)
            ar2_min = np.min(ar2)
            ar2_max = np.max(ar2)
            ar2_range = int(ar2_max) - int(ar2_min)
            below_memory_constraint = ar2_range <= 6 * (ar1.size + ar2.size)
            range_safe_from_overflow = ar2_range <= np.iinfo(ar2.dtype).max
            if ar1.size > 0:
                ar1_min = np.min(ar1)
                ar1_max = np.max(ar1)
                ar1_upper = min(int(ar1_max), int(ar2_max))
                ar1_lower = max(int(ar1_min), int(ar2_min))
                range_safe_from_overflow &= all((
                    ar1_upper - int(ar2_min) <= np.iinfo(ar1.dtype).max,
                    ar1_lower - int(ar2_min) >= np.iinfo(ar1.dtype).min
                ))
            if (
                range_safe_from_overflow and
                (below_memory_constraint or kind == 'table')
            ):
                if invert:
                    outgoing_array = np.ones_like(ar1, dtype=bool)
                else:
                    outgoing_array = np.zeros_like(ar1, dtype=bool)
                if invert:
                    isin_helper_ar = np.ones(ar2_range + 1, dtype=bool)
                    isin_helper_ar[ar2 - ar2_min] = 0
                else:
                    isin_helper_ar = np.zeros(ar2_range + 1, dtype=bool)
                    isin_helper_ar[ar2 - ar2_min] = 1
                basic_mask = (ar1 <= ar2_max) & (ar1 >= ar2_min)
                outgoing_array[basic_mask] = isin_helper_ar[ar1[basic_mask] -
  ar2_min]
            return outgoing_array
        elif kind == 'table': # not range_safe_from_overflow
            raise RuntimeError(
                "You have specified kind='table', "
                "but the range of values in `ar2` or `ar1` exceed the "
                "maximum integer of the datatype. "
                "Please set `kind` to None or 'sort'."
            )
        elif kind == 'table':
            raise ValueError(
                "The 'table' method is only "

```

```

546: (12)                     "supported for boolean or integer arrays. "
547: (12)                     "Please select 'sort' or None for kind."
548: (8)
549: (4)
550: (4)
551: (8)
552: (12)
553: (12)
554: (16)
555: (8)
556: (12)
557: (12)
558: (16)
559: (8)
560: (4)
561: (8)                     if len(ar2) < 10 * len(ar1) ** 0.145 or contains_object:
562: (8)                     if invert:
563: (12)                     mask = np.ones(len(ar1), dtype=bool)
564: (12)                     for a in ar2:
565: (16)                     mask &= (ar1 != a)
566: (8)
567: (4)                     else:
568: (8)                     mask = np.zeros(len(ar1), dtype=bool)
569: (12)                     for a in ar2:
570: (12)                     mask |= (ar1 == a)
571: (8)
572: (4)
573: (4)
574: (8)
575: (4)
576: (8)
577: (0)                     return mask
578: (21)                     if not assume_unique:
579: (4)                     ar1, rev_idx = np.unique(ar1, return_inverse=True)
580: (8)                     ar2 = np.unique(ar2)
581: (4)                     ar = np.concatenate((ar1, ar2))
582: (4)                     order = ar.argsort(kind='mergesort')
583: (4)                     sar = ar[order]
584: (4)                     if invert:
585: (8)                     bool_ar = (sar[1:] != sar[:-1])
586: (4)                     else:
587: (8)                     bool_ar = (sar[1:] == sar[:-1])
588: (4)                     flag = np.concatenate((bool_ar, [invert]))
589: (8)
590: (4)
591: (8)
592: (4)
593: (8)
594: (4)
595: (4)
596: (8)
597: (8)
598: (4)
599: (8)
600: (8)
601: (8)
602: (8)
603: (4)
604: (8)
605: (8)
606: (8)
607: (8)
608: (10)
609: (10)
610: (8)
611: (10)
612: (10)
613: (10)
614: (10)

def _isin_dispatcher(element, test_elements, assume_unique=None, invert=None,
                     *, kind=None):
    return (element, test_elements)
@array_function_dispatch(_isin_dispatcher)
def isin(element, test_elements, assume_unique=False, invert=False, *,
         kind=None):
    """
    Calculates ``element in test_elements``, broadcasting over `element` only.
    Returns a boolean array of the same shape as `element` that is True
    where an element of `element` is in `test_elements` and False otherwise.

    Parameters
    -----
    element : array_like
        Input array.
    test_elements : array_like
        The values against which to test each value of `element`.
        This argument is flattened if it is an array or array_like.
        See notes for behavior with non-array-like parameters.
    assume_unique : bool, optional
        If True, the input arrays are both assumed to be unique, which
        can speed up the calculation. Default is False.
    invert : bool, optional
        If True, the values in the returned array are inverted, as if
        calculating `element not in test_elements`. Default is False.
        ``np.isin(a, b, invert=True)`` is equivalent to (but faster
        than) ``np.invert(np.isin(a, b))``.
    kind : {None, 'sort', 'table'}, optional
        The algorithm to use. This will not affect the final result,
        but will affect the speed and memory use. The default, None,
        will select automatically based on memory considerations.
        * If 'sort', will use a mergesort-based approach. This will have
          a memory usage of roughly 6 times the sum of the sizes of
          `ar1` and `ar2`, not accounting for size of dtypes.
        * If 'table', will use a lookup table approach similar
          to a counting sort. This is only available for boolean and
          integer arrays. This will have a memory usage of the
          size of `ar1` plus the max-min value of `ar2`. `assume_unique`
          has no effect when the 'table' option is used.
    """

```

```

615: (8) * If None, will automatically choose 'table' if
616: (10) the required memory allocation is less than or equal to
617: (10) 6 times the sum of the sizes of `ar1` and `ar2`,
618: (10) otherwise will use 'sort'. This is done to not use
619: (10) a large amount of memory by default, even though
620: (10) 'table' may be faster in most cases. If 'table' is chosen,
621: (10) `assume_unique` will have no effect.
622: (4) Returns
623: (4)
624: (4) -----
625: (8) isin : ndarray, bool
626: (8) Has the same shape as `element`. The values `element[isin]`
627: (4) are in `test_elements`.
628: (4) See Also
629: (4) -----
630: (4) in1d : Flattened version of this function.
631: (28) numpy.lib.arraysetops : Module with a number of other functions for
632: (4) performing set operations on arrays.
633: (4) Notes
634: (4) -----
635: (4) `isin` is an element-wise function version of the python keyword `in`.
636: (4) ``isin(a, b)`` is roughly equivalent to
637: (4) ``np.array([item in b for item in a])`` if `a` and `b` are 1-D sequences.
638: (4) `element` and `test_elements` are converted to arrays if they are not
639: (4) already. If `test_elements` is a set (or other non-sequence collection)
640: (4) it will be converted to an object array with one element, rather than an
641: (4) array of the values contained in `test_elements`. This is a consequence
642: (4) of the `array` constructor's way of handling non-sequence collections.
643: (4) Converting the set to a list usually gives the desired behavior.
644: (4) Using ``kind='table'`` tends to be faster than `kind='sort'` if the
645: (4) following relationship is true:
646: (4) ``log10(len(ar2)) > (log10(max(ar2)-min(ar2)) - 2.27) / 0.927``,
647: (4) but may use greater memory. The default value for `kind` will
648: (4) be automatically selected based only on memory usage, so one may
649: (4) manually set ``kind='table'`` if memory constraints can be relaxed.
650: (4) .. versionadded:: 1.13.0
651: (4) Examples
652: (4) -----
653: (4) >>> element = 2*np.arange(4).reshape((2, 2))
654: (4) >>> element
655: (11) array([[0, 2],
656: (4) [4, 6]])
657: (4) >>> test_elements = [1, 2, 4, 8]
658: (4) >>> mask = np.isin(element, test_elements)
659: (4) >>> mask
660: (11) array([[False,  True],
661: (4) [ True, False]])
662: (4) >>> element[mask]
663: (4) array([2, 4])
664: (4) The indices of the matched values can be obtained with `nonzero`:
665: (4) >>> np.nonzero(mask)
666: (4) (array([0, 1]), array([1, 0]))
667: (4) The test can also be inverted:
668: (4) >>> mask = np.isin(element, test_elements, invert=True)
669: (4) >>> mask
670: (11) array([[ True, False],
671: (4) [False,  True]])
672: (4) >>> element[mask]
673: (4) array([0, 6])
674: (4) Because of how `array` handles sets, the following does not
675: (4) work as expected:
676: (4) >>> test_set = {1, 2, 4, 8}
677: (4) >>> np.isin(element, test_set)
678: (11) array([[False, False],
679: (4) [False, False]])
680: (4) Casting the set to a list gives the expected result:
681: (4) >>> np.isin(element, list(test_set))
682: (11) array([[False,  True],
683: (4) [ True, False]])
"""

```

```

684: (4)           element = np.asarray(element)
685: (4)           return in1d(element, test_elements, assume_unique=assume_unique,
686: (16)             invert=invert, kind=kind).reshape(element.shape)
687: (0)           def _union1d_dispatcher(ar1, ar2):
688: (4)             return (ar1, ar2)
689: (0)           @array_function_dispatch(_union1d_dispatcher)
690: (0)           def union1d(ar1, ar2):
691: (4)             """
692: (4)             Find the union of two arrays.
693: (4)             Return the unique, sorted array of values that are in either of the two
694: (4)             input arrays.
695: (4)             Parameters
696: (4)               -----
697: (4)               ar1, ar2 : array_like
698: (8)                 Input arrays. They are flattened if they are not already 1D.
699: (4)             Returns
699: (4)               -----
700: (4)               union1d : ndarray
701: (8)                 Unique, sorted union of the input arrays.
702: (4)             See Also
703: (4)               -----
704: (4)               numpy.lib.arraysetops : Module with a number of other functions for
705: (4)               performing set operations on arrays.
706: (28)
707: (4)             Examples
708: (4)               -----
709: (4)               >>> np.union1d([-1, 0, 1], [-2, 0, 2])
710: (4)               array([-2, -1, 0, 1, 2])
711: (4)               To find the union of more than two arrays, use functools.reduce:
712: (4)               >>> from functools import reduce
713: (4)               >>> reduce(np.union1d, ([1, 3, 4, 3], [3, 1, 2, 1], [6, 3, 4, 2]))
714: (4)               array([1, 2, 3, 4, 6])
715: (4)               """
716: (4)               return unique(np.concatenate((ar1, ar2), axis=None))
717: (0)           def _setdiff1d_dispatcher(ar1, ar2, assume_unique=None):
718: (4)             return (ar1, ar2)
719: (0)           @array_function_dispatch(_setdiff1d_dispatcher)
720: (0)           def setdiff1d(ar1, ar2, assume_unique=False):
721: (4)             """
722: (4)             Find the set difference of two arrays.
723: (4)             Return the unique values in `ar1` that are not in `ar2`.
724: (4)             Parameters
725: (4)               -----
726: (4)               ar1 : array_like
727: (8)                 Input array.
728: (4)               ar2 : array_like
729: (8)                 Input comparison array.
730: (4)               assume_unique : bool
731: (8)                 If True, the input arrays are both assumed to be unique, which
732: (8)                   can speed up the calculation. Default is False.
733: (4)             Returns
734: (4)               -----
735: (4)               setdiff1d : ndarray
736: (8)                 1D array of values in `ar1` that are not in `ar2`. The result
737: (8)                   is sorted when `assume_unique=False`, but otherwise only sorted
738: (8)                     if the input is sorted.
739: (4)             See Also
740: (4)               -----
741: (4)               numpy.lib.arraysetops : Module with a number of other functions for
742: (28)                   performing set operations on arrays.
743: (4)             Examples
744: (4)               -----
745: (4)               >>> a = np.array([1, 2, 3, 2, 4, 1])
746: (4)               >>> b = np.array([3, 4, 5, 6])
747: (4)               >>> np.setdiff1d(a, b)
748: (4)                 array([1, 2])
749: (4)               """
750: (4)               if assume_unique:
751: (8)                 ar1 = np.asarray(ar1).ravel()
752: (4)

```

```

753: (8)             ar1 = unique(ar1)
754: (8)             ar2 = unique(ar2)
755: (4)             return ar1[in1d(ar1, ar2, assume_unique=True, invert=True)]
-----
```

## File 206 - arrayterator.py:

```

1: (0)             """
2: (0)             A buffered iterator for big arrays.
3: (0)             This module solves the problem of iterating over a big file-based array
4: (0)             without having to read it into memory. The `Arrayterator` class wraps
5: (0)             an array object, and when iterated it will return sub-arrays with at most
6: (0)             a user-specified number of elements.
7: (0)             """
8: (0)             from operator import mul
9: (0)             from functools import reduce
10: (0)            __all__ = ['Arrayterator']
11: (0)            class Arrayterator:
12: (4)             """
13: (4)             Buffered iterator for big arrays.
14: (4)             `Arrayterator` creates a buffered iterator for reading big arrays in small
15: (4)             contiguous blocks. The class is useful for objects stored in the
16: (4)             file system. It allows iteration over the object *without* reading
17: (4)             everything in memory; instead, small blocks are read and iterated over.
18: (4)             `Arrayterator` can be used with any object that supports multidimensional
19: (4)             slices. This includes NumPy arrays, but also variables from
20: (4)             Scientific.IO.NetCDF or pynetcdf for example.
21: (4)             Parameters
22: (4)             -----
23: (4)             var : array_like
24: (8)             The object to iterate over.
25: (4)             buf_size : int, optional
26: (8)             The buffer size. If `buf_size` is supplied, the maximum amount of
27: (8)             data that will be read into memory is `buf_size` elements.
28: (8)             Default is None, which will read as many element as possible
29: (8)             into memory.
30: (4)             Attributes
31: (4)             -----
32: (4)             var
33: (4)             buf_size
34: (4)             start
35: (4)             stop
36: (4)             step
37: (4)             shape
38: (4)             flat
39: (4)             See Also
40: (4)             -----
41: (4)             ndenumerate : Multidimensional array iterator.
42: (4)             flatiter : Flat array iterator.
43: (4)             memmap : Create a memory-map to an array stored in a binary file on disk.
44: (4)             Notes
45: (4)             -----
46: (4)             The algorithm works by first finding a "running dimension", along which
47: (4)             the blocks will be extracted. Given an array of dimensions
48: (4)             `` $d_1, d_2, \dots, d_n$ `` , e.g. if `buf_size` is smaller than `` $d_1$ `` , the
49: (4)             first dimension will be used. If, on the other hand,
50: (4)             `` $d_1 < buf\_size < d_1 \cdot d_2$ `` the second dimension will be used, and so on.
51: (4)             Blocks are extracted along this dimension, and when the last block is
52: (4)             returned the process continues from the next dimension, until all
53: (4)             elements have been read.
54: (4)             Examples
55: (4)             -----
56: (4)             >>> a = np.arange(3 * 4 * 5 * 6).reshape(3, 4, 5, 6)
57: (4)             >>> a_itor = np.lib.Arrayterator(a, 2)
58: (4)             >>> a_itor.shape
59: (4)             (3, 4, 5, 6)
60: (4)             Now we can iterate over ``a_itor`` , and it will return arrays of size
61: (4)             two. Since `buf_size` was smaller than any dimension, the first
```

```

62: (4) dimension will be iterated over first:
63: (4) >>> for subarr in a_itor:
64: (4) ...     if not subarr.all():
65: (4) ...         print(subarr, subarr.shape) # doctest: +SKIP
66: (4) >>> # [[[0 1]]]] (1, 1, 1, 2)
67: (4)
68: (4) """
69: (8) def __init__(self, var, buf_size=None):
70: (8)     self.var = var
71: (8)     self.buf_size = buf_size
72: (8)     self.start = [0 for dim in var.shape]
73: (8)     self.stop = [dim for dim in var.shape]
74: (8)     self.step = [1 for dim in var.shape]
75: (8)     self._getattribute_()
76: (4)     return getattr(self.var, attr)
77: (8)     self._getitem_(index):
78: (8) """
79: (8)     Return a new arrayterator.
80: (8)     """
81: (12)     if not isinstance(index, tuple):
82: (8)         index = (index,)
83: (8)     fixed = []
84: (8)     length, dims = len(index), self.ndim
85: (12)     for slice_ in index:
86: (16)         if slice_ is Ellipsis:
87: (16)             fixed.extend([slice(None)] * (dims-length+1))
88: (12)             length = len(fixed)
89: (16)         elif isinstance(slice_, int):
90: (12)             fixed.append(slice(slice_, slice_+1, 1))
91: (16)         else:
92: (8)             fixed.append(slice_)
93: (8)     index = tuple(fixed)
94: (12)     if len(index) < dims:
95: (8)         index += (slice(None),) * (dims-len(index))
96: (8)     out = self.__class__(self.var, self.buf_size)
97: (16)     for i, (start, stop, step, slice_) in enumerate(
98: (12)         zip(self.start, self.stop, self.step, index)):
99: (12)         out.start[i] = start + (slice_.start or 0)
100: (12)         out.step[i] = step * (slice_.step or 1)
101: (12)         out.stop[i] = start + (slice_.stop or stop-step)
102: (12)         out.stop[i] = min(stop, out.stop[i])
103: (8)     return out
104: (4)     def __array__(self):
105: (8) """
106: (8)     Return corresponding data.
107: (8)     """
108: (16)     slice_ = tuple(slice(*t) for t in zip(
109: (16)         self.start, self.stop, self.step))
110: (8)     return self.var[slice_]
111: (4)     @property
112: (8)     def flat(self):
113: (8) """
114: (8)     A 1-D flat iterator for Arrayterator objects.
115: (8)     This iterator returns elements of the array to be iterated over in
116: (8)     `Arrayterator` one by one. It is similar to `flatiter`.
117: (8)     See Also
118: (8)     -----
119: (8)     Arrayterator
120: (8)     flatiter
121: (8)     Examples
122: (8)     -----
123: (8)     >>> a = np.arange(3 * 4 * 5 * 6).reshape(3, 4, 5, 6)
124: (8)     >>> a_itor = np.lib.Arrayterator(a, 2)
125: (8)     >>> for subarr in a_itor.flat:
126: (8)         if not subarr:
127: (8)             print(subarr, type(subarr))
128: (8)             ...
129: (8)             0 <class 'numpy.int64'>
130: (8)             """
130: (8)             for block in self:

```

```

131: (12)             yield from block.flat
132: (4)             @property
133: (4)             def shape(self):
134: (8)                 """
135: (8)                     The shape of the array to be iterated over.
136: (8)                     For an example, see `Arrayterator`.
137: (8)                 """
138: (8)                     return tuple(((stop-start-1)//step+1) for start, stop, step in
139: (16)                         zip(self.start, self.stop, self.step))
140: (4)             def __iter__(self):
141: (8)                 if [dim for dim in self.shape if dim <= 0]:
142: (12)                     return
143: (8)                 start = self.start[:]
144: (8)                 stop = self.stop[:]
145: (8)                 step = self.step[:]
146: (8)                 ndims = self.var.ndim
147: (8)                 while True:
148: (12)                     count = self.buf_size or reduce(mul, self.shape)
149: (12)                     rundim = 0
150: (12)                     for i in range(ndims-1, -1, -1):
151: (16)                         if count == 0:
152: (20)                             stop[i] = start[i]+1
153: (16)                         elif count <= self.shape[i]:
154: (20)                             stop[i] = start[i] + count*step[i]
155: (20)                             rundim = i
156: (16)                         else:
157: (20)                             stop[i] = self.stop[i]
158: (16)                             stop[i] = min(self.stop[i], stop[i])
159: (16)                             count = count//self.shape[i]
160: (12)                     slice_ = tuple(slice(*t) for t in zip(start, stop, step))
161: (12)                     yield self.var[slice_]
162: (12)                     start[rundim] = stop[rundim] # start where we stopped
163: (12)                     for i in range(ndims-1, 0, -1):
164: (16)                         if start[i] >= self.stop[i]:
165: (20)                             start[i] = self.start[i]
166: (20)                             start[i-1] += self.step[i-1]
167: (12)                     if start[0] >= self.stop[0]:
168: (16)                         return

```

---

## File 207 - format.py:

```

1: (0)             """
2: (0)             Binary serialization
3: (0)             NPY format
4: (0)             =====
5: (0)             A simple format for saving numpy arrays to disk with the full
6: (0)             information about them.
7: (0)             The ``.npy`` format is the standard binary file format in NumPy for
8: (0)             persisting a *single* arbitrary NumPy array on disk. The format stores all
9: (0)             of the shape and dtype information necessary to reconstruct the array
10: (0)            correctly even on another machine with a different architecture.
11: (0)            The format is designed to be as simple as possible while achieving
12: (0)            its limited goals.
13: (0)            The ``.npz`` format is the standard format for persisting *multiple* NumPy
14: (0)            arrays on disk. A ``.npz`` file is a zip file containing multiple ``.npy``
15: (0)            files, one for each array.
16: (0)            Capabilities
17: (0)            -----
18: (0)            - Can represent all NumPy arrays including nested record arrays and
19: (2)            object arrays.
20: (0)            - Represents the data in its native binary form.
21: (0)            - Supports Fortran-contiguous arrays directly.
22: (0)            - Stores all of the necessary information to reconstruct the array
23: (2)            including shape and dtype on a machine of a different
24: (2)            architecture. Both little-endian and big-endian arrays are
25: (2)            supported, and a file with little-endian numbers will yield
26: (2)            a little-endian array on any machine reading the file. The

```

27: (2) types are described in terms of their actual sizes. For example,  
 28: (2) if a machine with a 64-bit C "long int" writes out an array with  
 29: (2) "long ints", a reading machine with 32-bit C "long ints" will yield  
 30: (2) an array with 64-bit integers.  
 31: (0) - Is straightforward to reverse engineer. Datasets often live longer than  
 32: (2) the programs that created them. A competent developer should be  
 33: (2) able to create a solution in their preferred programming language to  
 34: (2) read most ``.npy`` files that they have been given without much  
 35: (2) documentation.  
 36: (0) - Allows memory-mapping of the data. See `open\_memmap`.  
 37: (0) - Can be read from a filelike stream object instead of an actual file.  
 38: (0) - Stores object arrays, i.e. arrays containing elements that are arbitrary  
 39: (2) Python objects. Files with object arrays are not to be mmapable, but  
 40: (2) can be read and written to disk.

41: (0) Limitations  
 42: (0) -----  
 43: (0) - Arbitrary subclasses of numpy.ndarray are not completely preserved.  
 44: (2) Subclasses will be accepted for writing, but only the array data will  
 45: (2) be written out. A regular numpy.ndarray object will be created  
 46: (2) upon reading the file.

47: (0) .. warning::  
 48: (2) Due to limitations in the interpretation of structured dtypes, dtypes  
 49: (2) with fields with empty names will have the names replaced by 'f0', 'f1',  
 50: (2) etc. Such arrays will not round-trip through the format entirely  
 51: (2) accurately. The data is intact; only the field names will differ. We are  
 52: (2) working on a fix for this. This fix will not require a change in the  
 53: (2) file format. The arrays with such structures can still be saved and  
 54: (2) restored, and the correct dtype may be restored by using the  
 55: (2) ``loadedarray.view(correct\_dtype)`` method.

56: (0) File extensions  
 57: (0) -----  
 58: (0) We recommend using the ``.npy`` and ``.npz`` extensions for files saved  
 59: (0) in this format. This is by no means a requirement; applications may wish  
 60: (0) to use these file formats but use an extension specific to the  
 61: (0) application. In the absence of an obvious alternative, however,  
 62: (0) we suggest using ``.npy`` and ``.npz``.

63: (0) Version numbering  
 64: (0) -----  
 65: (0) The version numbering of these formats is independent of NumPy version  
 66: (0) numbering. If the format is upgraded, the code in `numpy.io` will still  
 67: (0) be able to read and write Version 1.0 files.

68: (0) Format Version 1.0  
 69: (0) -----  
 70: (0) The first 6 bytes are a magic string: exactly ``\\x93NUMPY``.  
 71: (0) The next 1 byte is an unsigned byte: the major version number of the file  
 72: (0) format, e.g. ``\\x01``.  
 73: (0) The next 1 byte is an unsigned byte: the minor version number of the file  
 74: (0) format, e.g. ``\\x00``. Note: the version of the file format is not tied  
 75: (0) to the version of the numpy package.  
 76: (0) The next 2 bytes form a little-endian unsigned short int: the length of  
 77: (0) the header data HEADER\_LEN.  
 78: (0) The next HEADER\_LEN bytes form the header data describing the array's  
 79: (0) format. It is an ASCII string which contains a Python literal expression  
 80: (0) of a dictionary. It is terminated by a newline (``\\n``) and padded with  
 81: (0) spaces (``\\x20``) to make the total of  
 82: (0) ``len(magic string) + 2 + len(length) + HEADER\_LEN`` be evenly divisible  
 83: (0) by 64 for alignment purposes.  
 84: (0) The dictionary contains three keys:  
 85: (4)    "descr" : dtype.descr  
 86: (6)     An object that can be passed as an argument to the `numpy.dtype`  
 87: (6)     constructor to create the array's dtype.  
 88: (4)    "fortran\_order" : bool  
 89: (6)     Whether the array data is Fortran-contiguous or not. Since  
 90: (6)     Fortran-contiguous arrays are a common form of non-C-contiguity,  
 91: (6)     we allow them to be written directly to disk for efficiency.  
 92: (4)    "shape" : tuple of int  
 93: (6)     The shape of the array.  
 94: (0) For repeatability and readability, the dictionary keys are sorted in  
 95: (0) alphabetic order. This is for convenience only. A writer SHOULD implement

```

96: (0) this if possible. A reader MUST NOT depend on this.
97: (0) Following the header comes the array data. If the dtype contains Python
98: (0) objects (i.e. ``dtype.hasobject is True``), then the data is a Python
99: (0) pickle of the array. Otherwise the data is the contiguous (either C-
100: (0) or Fortran-, depending on ``fortran_order``) bytes of the array.
101: (0) Consumers can figure out the number of bytes by multiplying the number
102: (0) of elements given by the shape (noting that ``shape=()`` means there is
103: (0) 1 element) by ``dtype.itemsize``.
104: (0) Format Version 2.0
105: (0) -----
106: (0) The version 1.0 format only allowed the array header to have a total size of
107: (0) 65535 bytes. This can be exceeded by structured arrays with a large number of
108: (0) columns. The version 2.0 format extends the header size to 4 GiB.
109: (0) `numpy.save` will automatically save in 2.0 format if the data requires it,
110: (0) else it will always use the more compatible 1.0 format.
111: (0) The description of the fourth element of the header therefore has become:
112: (0) "The next 4 bytes form a little-endian unsigned int: the length of the header
113: (0) data HEADER_LEN."
114: (0) Format Version 3.0
115: (0) -----
116: (0) This version replaces the ASCII string (which in practice was latin1) with
117: (0) a utf8-encoded string, so supports structured types with any unicode field
118: (0) names.
119: (0) Notes
120: (0) -----
121: (0) The ``.npy`` format, including motivation for creating it and a comparison of
122: (0) alternatives, is described in the
123: (0) :doc:`"npy-format" NEP <neps:nep-0001-npy-format>`, however details have
124: (0) evolved with time and this document is more current.
125: (0)
126: (0) """
127: (0) import numpy
128: (0) import warnings
129: (0) from numpy.lib.utils import safe_eval, drop_metadata
130: (4) from numpy.compat import (
131: (4)     isfileobj, os_fspath, pickle
132: (0) )
133: (0) __all__ = []
134: (0) EXPECTED_KEYS = {'descr', 'fortran_order', 'shape'}
135: (0) MAGIC_PREFIX = b'\x93NUMPY'
136: (0) MAGIC_LEN = len(MAGIC_PREFIX) + 2
137: (0) ARRAY_ALIGN = 64 # plausible values are powers of 2 between 16 and 4096
138: (0) BUFFER_SIZE = 2**18 # size of buffer for reading npz files in bytes
139: (0) GROWTH_AXIS_MAX_DIGITS = 21 # = len(str(8*2**64-1)) hypothetical int1 dtype
140: (4) _header_size_info = {
141: (4)     (1, 0): ('<H', 'latin1'),
142: (4)     (2, 0): ('<I', 'latin1'),
143: (0)     (3, 0): ('<I', 'utf8'),
144: (0) }
145: (0) _MAX_HEADER_SIZE = 10000
146: (4) def _check_version(version):
147: (8)     if version not in [(1, 0), (2, 0), (3, 0), None]:
148: (8)         msg = "we only support format version (1,0), (2,0), and (3,0), not %s"
149: (0)         raise ValueError(msg % (version,))
150: (4) def magic(major, minor):
151: (4)     """ Return the magic string for the given file format version.
152: (4)     Parameters
153: (4)     -----
154: (4)     major : int in [0, 255]
155: (4)     minor : int in [0, 255]
156: (4)     Returns
157: (4)     -----
158: (4)     magic : str
159: (4)     Raises
159: (4)     -----
160: (4)     ValueError if the version cannot be formatted.
161: (4)     """
162: (4)     if major < 0 or major > 255:
163: (8)         raise ValueError("major version must be 0 <= major < 256")
164: (4)     if minor < 0 or minor > 255:

```

```

165: (8)           raise ValueError("minor version must be 0 <= minor < 256")
166: (4)           return MAGIC_PREFIX + bytes([major, minor])
167: (0)
168: (4)           """ Read the magic string to get the version of the file format.
169: (4)           Parameters
170: (4)           -----
171: (4)           fp : filelike object
172: (4)           Returns
173: (4)           -----
174: (4)           major : int
175: (4)           minor : int
176: (4)           """
177: (4)           magic_str = _read_bytes(fp, MAGIC_LEN, "magic string")
178: (4)           if magic_str[:-2] != MAGIC_PREFIX:
179: (8)               msg = "the magic string is not correct; expected %r, got %r"
180: (8)               raise ValueError(msg % (MAGIC_PREFIX, magic_str[:-2]))
181: (4)           major, minor = magic_str[-2:]
182: (4)           return major, minor
183: (0)
184: (4)           def dtype_to_descr(dtype):
185: (4)               """
186: (4)               Get a serializable descriptor from the dtype.
187: (4)               The .descr attribute of a dtype object cannot be round-tripped through
188: (4)               the dtype() constructor. Simple types, like dtype('float32'), have
189: (4)               a descr which looks like a record array with one field with '' as
190: (4)               a name. The dtype() constructor interprets this as a request to give
191: (4)               a default name. Instead, we construct descriptor that can be passed to
192: (4)               dtype().
193: (4)               Parameters
194: (4)               -----
195: (4)               dtype : dtype
196: (4)                   The dtype of the array that will be written to disk.
197: (4)               Returns
198: (4)               -----
199: (4)               descr : object
200: (8)                   An object that can be passed to `numpy.dtype()` in order to
201: (4)                   replicate the input dtype.
202: (4)               """
203: (4)               new_dtype = drop_metadata(dtype)
204: (4)               if new_dtype is not dtype:
205: (8)                   warnings.warn("metadata on a dtype is not saved to an npy/npz. "
206: (22)                       "Use another format (such as pickle) to store it.", UserWarning, stacklevel=2)
207: (4)               if dtype.names is not None:
208: (8)                   return dtype.descr
209: (4)               else:
210: (8)                   return dtype.str
211: (0)
212: (4)           def descr_to_dtype(descr):
213: (4)               """
214: (4)               Returns a dtype based off the given description.
215: (4)               This is essentially the reverse of `dtype_to_descr()`. It will remove
216: (4)               the valueless padding fields created by, i.e. simple fields like
217: (4)               dtype('float32'), and then convert the description to its corresponding
218: (4)               dtype.
219: (4)               Parameters
220: (4)               -----
221: (4)               descr : object
222: (8)                   The object retrieved by dtype.descr. Can be passed to
223: (8)                   `numpy.dtype()` in order to replicate the input dtype.
224: (4)               Returns
225: (4)               -----
226: (4)               dtype : dtype
227: (8)                   The dtype constructed by the description.
228: (4)               """
229: (8)               if isinstance(descr, str):
230: (8)                   return numpy.dtype(descr)
231: (4)               elif isinstance(descr, tuple):
232: (8)                   dt = descr_to_dtype(descr[0])
233: (8)                   return numpy.dtype((dt, descr[1]))
233: (4)               titles = []

```

```

234: (4)             names = []
235: (4)             formats = []
236: (4)             offsets = []
237: (4)             offset = 0
238: (4)             for field in descr:
239: (8)                 if len(field) == 2:
240: (12)                     name, descr_str = field
241: (12)                     dt = descr_to_dtype(descr_str)
242: (8)                 else:
243: (12)                     name, descr_str, shape = field
244: (12)                     dt = numpy.dtype((descr_to_dtype(descr_str), shape))
245: (8)             is_pad = (name == '' and dt.type is numpy.void and dt.names is None)
246: (8)             if not is_pad:
247: (12)                 title, name = name if isinstance(name, tuple) else (None, name)
248: (12)                 titles.append(title)
249: (12)                 names.append(name)
250: (12)                 formats.append(dt)
251: (12)                 offsets.append(offset)
252: (8)                 offset += dt.itemsize
253: (4)             return numpy.dtype({'names': names, 'formats': formats, 'titles': titles,
254: (24)                           'offsets': offsets, 'itemsize': offset})
255: (0)             def header_data_from_array_1_0(array):
256: (4)                 """ Get the dictionary of header metadata from a numpy.ndarray.
257: (4)                 Parameters
258: (4)                 -----
259: (4)                 array : numpy.ndarray
260: (4)                 Returns
261: (4)                 -----
262: (4)                 d : dict
263: (8)                     This has the appropriate entries for writing its string representation
264: (8)                     to the header of the file.
265: (4) """
266: (4)             d = {'shape': array.shape}
267: (4)             if array.flags.c_contiguous:
268: (8)                 d['fortran_order'] = False
269: (4)             elif array.flags.f_contiguous:
270: (8)                 d['fortran_order'] = True
271: (4)             else:
272: (8)                 d['fortran_order'] = False
273: (4)             d['descr'] = dtype_to_descr(array.dtype)
274: (4)             return d
275: (0)             def _wrap_header(header, version):
276: (4) """
277: (4)                 Takes a stringified header, and attaches the prefix and padding to it
278: (4) """
279: (4)             import struct
280: (4)             assert version is not None
281: (4)             fmt, encoding = _header_size_info[version]
282: (4)             header = header.encode(encoding)
283: (4)             hlen = len(header) + 1
284: (4)             padlen = ARRAY_ALIGN - ((MAGIC_LEN + struct.calcsize(fmt) + hlen) %
ARRAY_ALIGN)
285: (4)
286: (8)             try:
287: (4)                 header_prefix = magic(*version) + struct.pack(fmt, hlen + padlen)
288: (8)             except struct.error:
289: (8)                 msg = "Header length {} too big for version={}".format(hlen, version)
290: (8)                 raise ValueError(msg) from None
291: (4)             return header_prefix + header + b'*' * padlen + b'\n'
292: (0)             def _wrap_header_guess_version(header):
293: (4) """
294: (4)                 Like `_wrap_header`, but chooses an appropriate version given the contents
295: (4) """
296: (8)             try:
297: (4)                 return _wrap_header(header, (1, 0))
298: (4)             except ValueError:
299: (4)                 pass
300: (8)             try:
301: (4)                 ret = _wrap_header(header, (2, 0))
302: (4)             except UnicodeEncodeError:

```

```

302: (8)           pass
303: (4)           else:
304: (8)             warnings.warn("Stored array in format 2.0. It can only be"
305: (22)               "read by NumPy >= 1.9", UserWarning, stacklevel=2)
306: (8)             return ret
307: (4)             header = _wrap_header(header, (3, 0))
308: (4)             warnings.warn("Stored array in format 3.0. It can only be "
309: (18)               "read by NumPy >= 1.17", UserWarning, stacklevel=2)
310: (4)             return header
311: (0)           def _write_array_header(fp, d, version=None):
312: (4)             """ Write the header for an array and returns the version used
313: (4)             Parameters
314: (4)               -----
315: (4)               fp : filelike object
316: (4)               d : dict
317: (8)                 This has the appropriate entries for writing its string representation
318: (8)                 to the header of the file.
319: (4)               version : tuple or None
320: (8)                 None means use oldest that works. Providing an explicit version will
321: (8)                 raise a ValueError if the format does not allow saving this data.
322: (8)                 Default: None
323: (4)               """
324: (4)               header = ["{"]
325: (4)               for key, value in sorted(d.items()):
326: (8)                 header.append("%s: %s, " % (key, repr(value)))
327: (4)               header.append("}")
328: (4)               header = "".join(header)
329: (4)               shape = d['shape']
330: (4)               header += " " * ((GROWTH_AXIS_MAX_DIGITS - len(repr(
331: (8)                 shape[-1 if d['fortran_order'] else 0]
332: (4)               ))) if len(shape) > 0 else 0)
333: (4)               if version is None:
334: (8)                 header = _wrap_header_guess_version(header)
335: (4)               else:
336: (8)                 header = _wrap_header(header, version)
337: (4)               fp.write(header)
338: (0)           def write_array_header_1_0(fp, d):
339: (4)             """ Write the header for an array using the 1.0 format.
340: (4)             Parameters
341: (4)               -----
342: (4)               fp : filelike object
343: (4)               d : dict
344: (8)                 This has the appropriate entries for writing its string
345: (8)                 representation to the header of the file.
346: (4)               """
347: (4)               _write_array_header(fp, d, (1, 0))
348: (0)           def write_array_header_2_0(fp, d):
349: (4)             """ Write the header for an array using the 2.0 format.
350: (8)               The 2.0 format allows storing very large structured arrays.
351: (4)               .. versionadded:: 1.9.0
352: (4)             Parameters
353: (4)               -----
354: (4)               fp : filelike object
355: (4)               d : dict
356: (8)                 This has the appropriate entries for writing its string
357: (8)                 representation to the header of the file.
358: (4)               """
359: (4)               _write_array_header(fp, d, (2, 0))
360: (0)           def read_array_header_1_0(fp, max_header_size=_MAX_HEADER_SIZE):
361: (4)             """
362: (4)               Read an array header from a filelike object using the 1.0 file format
363: (4)               version.
364: (4)               This will leave the file object located just after the header.
365: (4)             Parameters
366: (4)               -----
367: (4)               fp : filelike object
368: (8)                 A file object or something with a `read()` method like a file.
369: (4)             Returns
369: (4)               -----

```

```

371: (4)           shape : tuple of int
372: (8)             The shape of the array.
373: (4)           fortran_order : bool
374: (8)             The array data will be written out directly if it is either
375: (8)               C-contiguous or Fortran-contiguous. Otherwise, it will be made
376: (8)               contiguous before writing it out.
377: (4)           dtype : dtype
378: (8)             The dtype of the file's data.
379: (4)           max_header_size : int, optional
380: (8)             Maximum allowed size of the header. Large headers may not be safe
381: (8)               to load securely and thus require explicitly passing a larger value.
382: (8)               See :py:func:`ast.literal_eval()` for details.
383: (4)           Raises
384: (4)             -----
385: (4)           ValueError
386: (8)             If the data is invalid.
387: (4)             """
388: (4)           return _read_array_header(
389: (12)             fp, version=(1, 0), max_header_size=max_header_size)
390: (0)           def read_array_header_2_0(fp, max_header_size=_MAX_HEADER_SIZE):
391: (4)             """
392: (4)             Read an array header from a filelike object using the 2.0 file format
393: (4)               version.
394: (4)             This will leave the file object located just after the header.
395: (4)               .. versionadded:: 1.9.0
396: (4)           Parameters
397: (4)             -----
398: (4)           fp : filelike object
399: (8)             A file object or something with a `read()` method like a file.
400: (4)           max_header_size : int, optional
401: (8)             Maximum allowed size of the header. Large headers may not be safe
402: (8)               to load securely and thus require explicitly passing a larger value.
403: (8)               See :py:func:`ast.literal_eval()` for details.
404: (4)           Returns
405: (4)             -----
406: (4)           shape : tuple of int
407: (8)             The shape of the array.
408: (4)           fortran_order : bool
409: (8)             The array data will be written out directly if it is either
410: (8)               C-contiguous or Fortran-contiguous. Otherwise, it will be made
411: (8)               contiguous before writing it out.
412: (4)           dtype : dtype
413: (8)             The dtype of the file's data.
414: (4)           Raises
415: (4)             -----
416: (4)           ValueError
417: (8)             If the data is invalid.
418: (4)             """
419: (4)           return _read_array_header(
420: (12)             fp, version=(2, 0), max_header_size=max_header_size)
421: (0)           def _filter_header(s):
422: (4)             """Clean up 'L' in npz header ints.
423: (4)             Cleans up the 'L' in strings representing integers. Needed to allow npz
424: (4)               headers produced in Python2 to be read in Python3.
425: (4)           Parameters
426: (4)             -----
427: (4)           s : string
428: (8)             Npy file header.
429: (4)           Returns
429: (4)             -----
430: (4)           header : str
431: (4)             Cleaned up header.
432: (8)             """
433: (4)           import tokenize
434: (4)           from io import StringIO
435: (4)           tokens = []
436: (4)           last_token_was_number = False
437: (4)           for token in tokenize.generate_tokens(StringIO(s).readline):
438: (4)             token_type = token[0]
439: (8)

```

```

440: (8)             token_string = token[1]
441: (8)             if (last_token_was_number and
442: (16)                 token_type == tokenize.NAME and
443: (16)                 token_string == "L"):
444: (12)                 continue
445: (8)             else:
446: (12)                 tokens.append(token)
447: (8)             last_token_was_number = (token_type == tokenize.NUMBER)
448: (4)             return tokenize.untokenize(tokens)
449: (0)         def _read_array_header(fp, version, max_header_size=_MAX_HEADER_SIZE):
450: (4)             """
451: (4)             see read_array_header_1_0
452: (4)             """
453: (4)             import struct
454: (4)             hinfo = _header_size_info.get(version)
455: (4)             if hinfo is None:
456: (8)                 raise ValueError("Invalid version {!r}".format(version))
457: (4)             hlength_type, encoding = hinfo
458: (4)             hlength_str = _read_bytes(fp, struct.calcsize(hlength_type), "array header
length")
459: (4)             header_length = struct.unpack(hlength_type, hlength_str)[0]
460: (4)             header = _read_bytes(fp, header_length, "array header")
461: (4)             header = header.decode(encoding)
462: (4)             if len(header) > max_header_size:
463: (8)                 raise ValueError(
464: (12)                     f"Header info length ({len(header)}) is large and may not be safe
"
465: (12)                     "to load securely.\n"
466: (12)                     "To allow loading, adjust `max_header_size` or fully trust "
467: (12)                     "the `.npy` file using `allow_pickle=True`.\n"
468: (12)                     "For safety against large resource use or crashes, sandboxing "
469: (12)                     "may be necessary.")
470: (4)             try:
471: (8)                 d = safe_eval(header)
472: (4)             except SyntaxError as e:
473: (8)                 if version <= (2, 0):
474: (12)                     header = _filter_header(header)
475: (12)                     try:
476: (16)                         d = safe_eval(header)
477: (12)                     except SyntaxError as e2:
478: (16)                         msg = "Cannot parse header: {!r}"
479: (16)                         raise ValueError(msg.format(header)) from e2
480: (12)                     else:
481: (16)                         warnings.warn(
482: (20)                             "Reading `.npy` or `.npz` file required additional "
483: (20)                             "header parsing as it was created on Python 2. Save the "
484: (20)                             "file again to speed up loading and avoid this warning.",
485: (20)                             UserWarning, stacklevel=4)
486: (8)                     else:
487: (12)                         msg = "Cannot parse header: {!r}"
488: (12)                         raise ValueError(msg.format(header)) from e
489: (4)             if not isinstance(d, dict):
490: (8)                 msg = "Header is not a dictionary: {!r}"
491: (8)                 raise ValueError(msg.format(d))
492: (4)             if EXPECTED_KEYS != d.keys():
493: (8)                 keys = sorted(d.keys())
494: (8)                 msg = "Header does not contain the correct keys: {!r}"
495: (8)                 raise ValueError(msg.format(keys))
496: (4)             if (not isinstance(d['shape'], tuple) or
497: (12)                 not all(isinstance(x, int) for x in d['shape'])):
498: (8)                 msg = "shape is not valid: {!r}"
499: (8)                 raise ValueError(msg.format(d['shape']))
500: (4)             if not isinstance(d['fortran_order'], bool):
501: (8)                 msg = "fortran_order is not a valid bool: {!r}"
502: (8)                 raise ValueError(msg.format(d['fortran_order']))
503: (4)             try:
504: (8)                 dtype = descr_to_dtype(d['descr'])
505: (4)             except TypeError as e:
506: (8)                 msg = "descr is not a valid dtype descriptor: {!r}"

```

```

507: (8)             raise ValueError(msg.format(d['descr'])) from e
508: (4)             return d['shape'], d['fortran_order'], dtype
509: (0)         def write_array(fp, array, version=None, allow_pickle=True,
pickle_kwarg=None):
510: (4)             """
511: (4)             Write an array to an NPY file, including a header.
512: (4)             If the array is neither C-contiguous nor Fortran-contiguous AND the
513: (4)             file_like object is not a real file object, this function will have to
514: (4)             copy data in memory.
515: (4)             Parameters
516: (4)             -----
517: (4)             fp : file_like object
518: (8)                 An open, writable file object, or similar object with a
519: (8)                 ```.write()`` method.
520: (4)             array : ndarray
521: (8)                 The array to write to disk.
522: (4)             version : (int, int) or None, optional
523: (8)                 The version number of the format. None means use the oldest
524: (8)                 supported version that is able to store the data. Default: None
525: (4)             allow_pickle : bool, optional
526: (8)                 Whether to allow writing pickled data. Default: True
527: (4)             pickle_kwarg : dict, optional
528: (8)                 Additional keyword arguments to pass to pickle.dump, excluding
529: (8)                 'protocol'. These are only useful when pickling objects in object
530: (8)                 arrays on Python 3 to Python 2 compatible format.
531: (4)             Raises
532: (4)             -----
533: (4)             ValueError
534: (8)                 If the array cannot be persisted. This includes the case of
535: (8)                 allow_pickle=False and array being an object array.
536: (4)             Various other errors
537: (8)                 If the array contains Python objects as part of its dtype, the
538: (8)                 process of pickling them may raise various errors if the objects
539: (8)                 are not picklable.
540: (4)             """
541: (4)             _check_version(version)
542: (4)             _write_array_header(fp, header_data_from_array_1_0(array), version)
543: (4)             if array.itemsize == 0:
544: (8)                 buffersize = 0
545: (4)             else:
546: (8)                 buffersize = max(16 * 1024 ** 2 // array.itemsize, 1)
547: (4)             if array.dtype.hasobject:
548: (8)                 if not allow_pickle:
549: (12)                     raise ValueError("Object arrays cannot be saved when "
550: (29)                         "allow_pickle=False")
551: (8)                 if pickle_kwarg is None:
552: (12)                     pickle_kwarg = {}
553: (8)                 pickle.dump(array, fp, protocol=3, **pickle_kwarg)
554: (4)             elif array.flags.f_contiguous and not array.flags.c_contiguous:
555: (8)                 if isfileobj(fp):
556: (12)                     array.T.tofile(fp)
557: (8)                 else:
558: (12)                     for chunk in numpy.nditer(
559: (20)                         array, flags=['external_loop', 'buffered', 'zerosize_ok'],
560: (20)                         buffersize=buffersize, order='F'):
561: (16)                         fp.write(chunk.tobytes('C'))
562: (4)             else:
563: (8)                 if isfileobj(fp):
564: (12)                     array.tofile(fp)
565: (8)                 else:
566: (12)                     for chunk in numpy.nditer(
567: (20)                         array, flags=['external_loop', 'buffered', 'zerosize_ok'],
568: (20)                         buffersize=buffersize, order='C'):
569: (16)                         fp.write(chunk.tobytes('C'))
570: (0)         def read_array(fp, allow_pickle=False, pickle_kwarg=None, *,
571: (15)                         max_header_size=_MAX_HEADER_SIZE):
572: (4)             """
573: (4)             Read an array from an NPY file.
574: (4)             Parameters

```

```

575: (4)
576: (4)
577: (8)
578: (8)
579: (4)
580: (8)
581: (8)
582: (12)
583: (4)
584: (8)
585: (8)
586: (8)
587: (4)
588: (8)
589: (8)
590: (8)
591: (8)
592: (8)
593: (4)
594: (4)
595: (4)
596: (8)
597: (4)
598: (4)
599: (4)
600: (8)
601: (8)
602: (4)
603: (4)
604: (8)
605: (4)
606: (4)
607: (4)
608: (12)
609: (4)
610: (8)
611: (4)
612: (8)
613: (4)
614: (8)
615: (12)
616: (29)
617: (8)
618: (12)
619: (8)
620: (12)
621: (8)
622: (12)
623: (31)
624: (31)
625: (4)
626: (8)
627: (12)
628: (8)
629: (12)
630: (12)
631: (16)
dtype.itemsize)
632: (16)
633: (20)
634: (20)
635: (20)
636: (20)
637: (61)
638: (8)
639: (12)
640: (12)
641: (8)

    -----
    fp : file_like object
        If this is not a real file object, then this may take extra memory
        and time.
    allow_pickle : bool, optional
        Whether to allow writing pickled data. Default: False
        .. versionchanged:: 1.16.3
            Made default False in response to CVE-2019-6446.
    pickle_kwds : dict
        Additional keyword arguments to pass to pickle.load. These are only
        useful when loading object arrays saved on Python 2 when using
        Python 3.
    max_header_size : int, optional
        Maximum allowed size of the header. Large headers may not be safe
        to load securely and thus require explicitly passing a larger value.
        See :py:func:`ast.literal_eval()` for details.
        This option is ignored when `allow_pickle` is passed. In that case
        the file is by definition trusted and the limit is unnecessary.

Returns
-----
array : ndarray
    The array from the data on disk.

Raises
-----
ValueError
    If the data is invalid, or allow_pickle=False and the file contains
    an object array.

"""
if allow_pickle:
    max_header_size = 2**64
version = read_magic(fp)
_check_version(version)
shape, fortran_order, dtype = _read_array_header(
    fp, version, max_header_size=max_header_size)
if len(shape) == 0:
    count = 1
else:
    count = numpy.multiply.reduce(shape, dtype=numpy.int64)
if dtype.hasobject:
    if not allow_pickle:
        raise ValueError("Object arrays cannot be loaded when "
                         "allow_pickle=False")
    if pickle_kwds is None:
        pickle_kwds = {}
try:
    array = pickle.load(fp, **pickle_kwds)
except UnicodeError as err:
    raise UnicodeError("Unpickling a python object failed: %r\n"
                       "You may need to pass the encoding= option "
                       "to numpy.load" % (err,)) from err
else:
    if isfileobj(fp):
        array = numpy.fromfile(fp, dtype=dtype, count=count)
    else:
        array = numpy.ndarray(count, dtype=dtype)
        if dtype.itemsize > 0:
            max_read_count = BUFFER_SIZE // min(BUFFER_SIZE,
  dtype.itemsize)
            for i in range(0, count, max_read_count):
                read_count = min(max_read_count, count - i)
                read_size = int(read_count * dtype.itemsize)
                data = _read_bytes(fp, read_size, "array data")
                array[i:i+read_count] = numpy.frombuffer(data,
  count=read_count)
        if fortran_order:
            array.shape = shape[::-1]
            array = array.transpose()
        else:

```

```

642: (12)           array.shape = shape
643: (4)           return array
644: (0)
645: (16)          def open_memmap(filename, mode='r+', dtype=None, shape=None,
646: (16)                      fortran_order=False, version=None, *,
647: (16)                      max_header_size=_MAX_HEADER_SIZE):
648: (4)                      """
649: (4)                      Open a .npy file as a memory-mapped array.
650: (4)                      This may be used to read an existing file or create a new one.
651: (4)                      Parameters
652: (4)                      -----
653: (4)                      filename : str or path-like
654: (8)                          The name of the file on disk. This may *not* be a file-like
655: (8)                          object.
656: (4)                      mode : str, optional
657: (8)                          The mode in which to open the file; the default is 'r+'. In
658: (8)                          addition to the standard file modes, 'c' is also accepted to mean
659: (8)                          "copy on write." See `memmap` for the available mode strings.
660: (4)                      dtype : data-type, optional
661: (8)                          The data type of the array if we are creating a new file in "write"
662: (8)                          mode, if not, `dtype` is ignored. The default value is None, which
663: (8)                          results in a data-type of `float64`.
664: (4)                      shape : tuple of int
665: (8)                          The shape of the array if we are creating a new file in "write"
666: (8)                          mode, in which case this parameter is required. Otherwise, this
667: (8)                          parameter is ignored and is thus optional.
668: (4)                      fortran_order : bool, optional
669: (8)                          Whether the array should be Fortran-contiguous (True) or
670: (8)                          C-contiguous (False, the default) if we are creating a new file in
671: (8)                          "write" mode.
672: (4)                      version : tuple of int (major, minor) or None
673: (8)                          If the mode is a "write" mode, then this is the version of the file
674: (8)                          format used to create the file. None means use the oldest
675: (8)                          supported version that is able to store the data. Default: None
676: (4)                      max_header_size : int, optional
677: (8)                          Maximum allowed size of the header. Large headers may not be safe
678: (8)                          to load securely and thus require explicitly passing a larger value.
679: (8)                          See :py:func:`ast.literal_eval()` for details.
680: (4)                      Returns
681: (4)                      -----
682: (4)                      marray : memmap
683: (8)                          The memory-mapped array.
684: (4)                      Raises
685: (4)                      -----
686: (4)                      ValueError
687: (8)                          If the data or the mode is invalid.
688: (4)                      OSError
689: (8)                          If the file is not found or cannot be opened correctly.
690: (4)                      See Also
691: (4)                      -----
692: (4)                      numpy.memmap
693: (4)                      """
694: (8)                      if isfileobj(filename):
695: (25)                          raise ValueError("Filename must be a string or a path-like object."
696: (4)                                      "Memmap cannot use existing file handles.")
697: (8)                      if 'w' in mode:
698: (8)                          _check_version(version)
699: (8)                          dtype = numpy.dtype(dtype)
700: (12)                          if dtype.hasobject:
701: (12)                              msg = "Array can't be memory-mapped: Python objects in dtype."
702: (8)                              raise ValueError(msg)
703: (12)                          d = dict(
704: (12)                              descr=dtype_to_descr(dtype),
705: (12)                              fortran_order=fortran_order,
706: (8)                              shape=shape,
707: (8)                          )
708: (12)                          with open(os_fspath(filename), mode+'b') as fp:
709: (12)                              _write_array_header(fp, d, version)
710: (4)                              offset = fp.tell()
711: (4)                      else:
```

```

711: (8)             with open(os_fspath(filename), 'rb') as fp:
712: (12)             version = read_magic(fp)
713: (12)             _check_version(version)
714: (12)             shape, fortran_order, dtype = _read_array_header(
715: (20)                 fp, version, max_header_size=max_header_size)
716: (12)             if dtype.hasobject:
717: (16)                 msg = "Array can't be memory-mapped: Python objects in dtype."
718: (16)                 raise ValueError(msg)
719: (12)             offset = fp.tell()
720: (4)             if fortran_order:
721: (8)                 order = 'F'
722: (4)             else:
723: (8)                 order = 'C'
724: (4)             if mode == 'w+':
725: (8)                 mode = 'r+'
726: (4)             marray = numpy.memmap(filename, dtype=dtype, shape=shape, order=order,
727: (8)                 mode=mode, offset=offset)
728: (4)             return marray
729: (0)         def _read_bytes(fp, size, error_template="ran out of data"):
730: (4)             """
731: (4)             Read from file-like object until size bytes are read.
732: (4)             Raises ValueError if not EOF is encountered before size bytes are read.
733: (4)             Non-blocking objects only supported if they derive from io objects.
734: (4)             Required as e.g. ZipExtFile in python 2.6 can return less data than
735: (4)             requested.
736: (4)             """
737: (4)             data = bytes()
738: (4)             while True:
739: (8)                 try:
740: (12)                     r = fp.read(size - len(data))
741: (12)                     data += r
742: (12)                     if len(r) == 0 or len(data) == size:
743: (16)                         break
744: (8)                     except BlockingIOError:
745: (12)                         pass
746: (4)                     if len(data) != size:
747: (8)                         msg = "EOF: reading %s, expected %d bytes got %d"
748: (8)                         raise ValueError(msg % (error_template, size, len(data)))
749: (4)                 else:
750: (8)                     return data

```

-----

## File 208 - function\_base.py:

```

1: (0)             import collections.abc
2: (0)             import functools
3: (0)             import re
4: (0)             import sys
5: (0)             import warnings
6: (0)             from ..utils import set_module
7: (0)             import numpy as np
8: (0)             import numpy.core.numeric as _nx
9: (0)             from numpy.core import transpose
10: (0)            from numpy.core.numeric import (
11: (4)                ones, zeros_like, arange, concatenate, array, asarray, asanyarray, empty,
12: (4)                ndarray, take, dot, where, intp, integer, isscalar, absolute
13: (4)                )
14: (0)            from numpy.core.umath import (
15: (4)                pi, add, arctan2, frompyfunc, cos, less_equal, sqrt, sin,
16: (4)                mod, exp, not_equal, subtract
17: (4)                )
18: (0)            from numpy.core.fromnumeric import (
19: (4)                ravel, nonzero, partition, mean, any, sum
20: (4)                )
21: (0)            from numpy.core.numerictypes import typecodes
22: (0)            from numpy.core import overrides
23: (0)            from numpy.core.function_base import add_newdoc
24: (0)            from numpy.lib.twodim_base import diag

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

25: (0)
26: (4)     from numpy.core.multiarray import (
27: (4)         _place, add_docstring, bincount, normalize_axis_index, _monotonicity,
28: (4)         interp as compiled_interp, interp_complex as compiled_interp_complex
29: (0)     )
30: (0)     from numpy.core.umath import _add_newdoc_ufunc as add_newdoc_ufunc
31: (0)     import builtins
32: (0)     from numpy.lib.histograms import histogram, histogramdd  # noqa: F401
33: (4)     array_function_dispatch = functools.partial(
34: (4)         overrides.array_function_dispatch, module='numpy')
35: (0)     __all__ = [
36: (4)         'select', 'piecewise', 'trim_zeros', 'copy', 'iterable', 'percentile',
37: (4)         'diff', 'gradient', 'angle', 'unwrap', 'sort_complex', 'disp', 'flip',
38: (4)         'rot90', 'extract', 'place', 'vectorize', 'asarray_chkfinite', 'average',
39: (4)         'bincount', 'digitize', 'cov', 'corrcoef',
40: (4)         'msort', 'median', 'sinc', 'hamming', 'hanning', 'bartlett',
41: (4)         'blackman', 'kaiser', 'trapz', 'i0', 'add_newdoc', 'add_docstring',
42: (4)         'meshgrid', 'delete', 'insert', 'append', 'interp', 'add_newdoc_ufunc',
43: (4)         'quantile'
44: (0)     ]
45: (4)     _QuantileMethods = dict(
46: (8)         inverted_cdf=dict(
47: (8)             get_virtual_index=lambda n, quantiles: _inverted_cdf(n, quantiles),
48: (8)             fix_gamma=lambda gamma, _: gamma, # should never be called
49: (4)         ),
50: (8)         averaged_inverted_cdf=dict(
51: (8)             get_virtual_index=lambda n, quantiles: (n * quantiles) - 1,
52: (12)             fix_gamma=lambda gamma, _: _get_gamma_mask(
53: (12)                 shape=gamma.shape,
54: (12)                 default_value=1.,
55: (12)                 conditioned_value=0.5,
56: (12)                 where=gamma == 0),
57: (4)         ),
58: (8)         closest_observation=dict(
59: (8)             get_virtual_index=lambda n, quantiles: _closest_observation(n,
60: (8)                 quantiles),
61: (8)                 fix_gamma=lambda gamma, _: gamma, # should never be called
62: (4)         ),
63: (8)         interpolated_inverted_cdf=dict(
64: (8)             get_virtual_index=lambda n, quantiles:
65: (8)                 _compute_virtual_index(n, quantiles, 0, 1),
66: (8)                 fix_gamma=lambda gamma, _: gamma,
67: (4)         ),
68: (8)         hazen=dict(
69: (8)             get_virtual_index=lambda n, quantiles:
70: (8)                 _compute_virtual_index(n, quantiles, 0.5, 0.5),
71: (8)                 fix_gamma=lambda gamma, _: gamma,
72: (4)         ),
73: (8)         weibull=dict(
74: (8)             get_virtual_index=lambda n, quantiles:
75: (8)                 _compute_virtual_index(n, quantiles, 0, 0),
76: (8)                 fix_gamma=lambda gamma, _: gamma,
77: (4)         ),
78: (8)         linear=dict(
79: (8)             get_virtual_index=lambda n, quantiles: (n - 1) * quantiles,
80: (8)             fix_gamma=lambda gamma, _: gamma,
81: (4)         ),
82: (8)         median_unbiased=dict(
83: (8)             get_virtual_index=lambda n, quantiles:
84: (8)                 _compute_virtual_index(n, quantiles, 1 / 3.0, 1 / 3.0),
85: (8)                 fix_gamma=lambda gamma, _: gamma,
86: (4)         ),
87: (8)         normal_unbiased=dict(
88: (8)             get_virtual_index=lambda n, quantiles:
89: (8)                 _compute_virtual_index(n, quantiles, 3 / 8.0, 3 / 8.0),
90: (8)                 fix_gamma=lambda gamma, _: gamma,
91: (4)         ),
92: (8)         lower=dict(
92: (8)             get_virtual_index=lambda n, quantiles: np.floor(

```

```

93: (12)           (n - 1) * quantiles).astype(np.intp),
94: (8)             fix_gamma=lambda gamma, _: gamma,
95: (4)
96: (4)
97: (8)             higher=dict(
98: (12)               get_virtual_index=lambda n, quantiles: np.ceil(
99: (8)                 (n - 1) * quantiles).astype(np.intp),
100: (4)                   fix_gamma=lambda gamma, _: gamma,
101: (4)
102: (8)                   midpoint=dict(
103: (16)                     get_virtual_index=lambda n, quantiles: 0.5 * (
104: (16)                       np.floor((n - 1) * quantiles)
105: (8)                         + np.ceil((n - 1) * quantiles)),
106: (12)                         fix_gamma=lambda gamma, index: _get_gamma_mask(
107: (12)                           shape=gamma.shape,
108: (12)                           default_value=0.5,
109: (12)                           conditioned_value=0.,
110: (4)                             where=index % 1 == 0),
111: (4)
112: (8)                           nearest=dict(
113: (12)                             get_virtual_index=lambda n, quantiles: np.around(
114: (8)                               (n - 1) * quantiles).astype(np.intp),
115: (4)                                 fix_gamma=lambda gamma, _: gamma,
116: (0)
117: (4)
118: (0) @array_function_dispatch(_rot90_dispatcher)
119: (0) def rot90(m, k=1, axes=(0, 1)):
120: (4) """
121: (4)     Rotate an array by 90 degrees in the plane specified by axes.
122: (4)     Rotation direction is from the first towards the second axis.
123: (4)     This means for a 2D array with the default `k` and `axes`, the
124: (4)     rotation will be counterclockwise.
125: (4) Parameters
126: (4) -----
127: (4) m : array_like
128: (8)     Array of two or more dimensions.
129: (4) k : integer
130: (8)     Number of times the array is rotated by 90 degrees.
131: (4) axes : (2,) array_like
132: (8)     The array is rotated in the plane defined by the axes.
133: (8)     Axes must be different.
134: (8)     .. versionadded:: 1.12.0
135: (4) Returns
136: (4) -----
137: (4) y : ndarray
138: (8)     A rotated view of `m`.
139: (4) See Also
140: (4) -----
141: (4) flip : Reverse the order of elements in an array along the given axis.
142: (4) fliplr : Flip an array horizontally.
143: (4) flipud : Flip an array vertically.
144: (4) Notes
145: (4) -----
146: (4) ```rot90(m, k=1, axes=(1,0))``` is the reverse of
147: (4) ```rot90(m, k=1, axes=(0,1))``` .
148: (4) ```rot90(m, k=1, axes=(1,0))``` is equivalent to
149: (4) ```rot90(m, k=-1, axes=(0,1))``` .
150: (4) Examples
151: (4) -----
152: (4) >>> m = np.array([[1,2],[3,4]], int)
153: (4) >>> m
154: (4) array([[1, 2],
155: (11)           [3, 4]])
156: (4)
157: (4) >>> np.rot90(m)
158: (4) array([[2, 4],
159: (11)           [1, 3]])
160: (4) >>> np.rot90(m, 2)
161: (4) array([[4, 3],
161: (11)           [2, 1]])

```

```

162: (4)
163: (4)
164: (4)
165: (12)
166: (11)
167: (12)
168: (4)
169: (4)
170: (4)
171: (8)
172: (4)
173: (4)
174: (8)
175: (4)
176: (8)
177: (8)
178: (12)
179: (4)
180: (4)
181: (8)
182: (4)
183: (8)
184: (4)
185: (4)
186: (48)
187: (4)
188: (8)
189: (4)
190: (8)
191: (0)
192: (4)
193: (0)
194: (0)
195: (4)
196: (4)
197: (4)
198: (4)
199: (4)
200: (4)
201: (4)
202: (8)
203: (4)
204: (9)
205: (9)
206: (9)
207: (9)
208: (9)
209: (9)
210: (12)
211: (4)
212: (4)
213: (4)
214: (8)
215: (8)
216: (4)
217: (4)
218: (4)
219: (4)
220: (4)
221: (4)
222: (4)
223: (4)
224: (4)
225: (4)
226: (4)
227: (4)
228: (4)
229: (4)
230: (4)

    >>> m = np.arange(8).reshape((2,2,2))
    >>> np.rot90(m, 1, (1,2))
    array([[[1, 3],
             [0, 2]],
           [[5, 7],
             [4, 6]]])
    """
axes = tuple(axes)
if len(axes) != 2:
    raise ValueError("len(axes) must be 2.")
m = asanyarray(m)
if axes[0] == axes[1] or absolute(axes[0] - axes[1]) == m.ndim:
    raise ValueError("Axes must be different.")
if (axes[0] >= m.ndim or axes[0] < -m.ndim
    or axes[1] >= m.ndim or axes[1] < -m.ndim):
    raise ValueError("Axes={} out of range for array of ndim={}."
                     .format(axes, m.ndim))
k %= 4
if k == 0:
    return m[:]
if k == 2:
    return flip(flip(m, axes[0]), axes[1])
axes_list = arange(0, m.ndim)
(axes_list[axes[0]], axes_list[axes[1]]) = (axes_list[axes[1]],
   axes_list[axes[0]])
if k == 1:
    return transpose(flip(m, axes[1]), axes_list)
else:
    return flip(transpose(m, axes_list), axes[1])
def _flip_dispatcher(m, axis=None):
    return (m,)
@array_function_dispatch(_flip_dispatcher)
def flip(m, axis=None):
    """
    Reverse the order of elements in an array along the given axis.
    The shape of the array is preserved, but the elements are reordered.
    .. versionadded:: 1.12.0
    Parameters
    -----
    m : array_like
        Input array.
    axis : None or int or tuple of ints, optional
        Axis or axes along which to flip over. The default,
        axis=None, will flip over all of the axes of the input array.
        If axis is negative it counts from the last to the first axis.
        If axis is a tuple of ints, flipping is performed on all of the axes
        specified in the tuple.
        .. versionchanged:: 1.15.0
        None and tuples of axes are supported
    Returns
    -----
    out : array_like
        A view of `m` with the entries of axis reversed. Since a view is
        returned, this operation is done in constant time.
    See Also
    -----
    flipud : Flip an array vertically (axis=0).
    fliplr : Flip an array horizontally (axis=1).
    Notes
    -----
    flip(m, 0) is equivalent to flipud(m).
    flip(m, 1) is equivalent to fliplr(m).
    flip(m, n) corresponds to ``m[...,:-1,...]`` with ``::-1`` at position n.
    flip(m) corresponds to ``m[::-1,:,:-1,...,:-1]`` with ``::-1`` at all
    positions.
    flip(m, (0, 1)) corresponds to ``m[::-1,:,:-1,...]`` with ``::-1`` at
    position 0 and position 1.
    Examples
    -----

```

```

231: (4)
232: (4)
233: (4)
234: (12)
235: (11)
236: (12)
237: (4)
238: (4)
239: (12)
240: (11)
241: (12)
242: (4)
243: (4)
244: (12)
245: (11)
246: (12)
247: (4)
248: (4)
249: (12)
250: (11)
251: (12)
252: (4)
253: (4)
254: (12)
255: (11)
256: (12)
257: (4)
258: (4)
259: (4)
260: (4)
261: (4)
262: (8)
263: (4)
264: (8)
265: (4)
266: (8)
267: (8)
268: (8)
269: (12)
270: (8)
271: (4)
272: (0)
273: (0)
274: (4)
275: (4)
276: (4)
277: (4)
278: (4)
279: (6)
280: (4)
281: (4)
282: (4)
283: (6)
284: (6)
285: (4)
286: (4)
287: (4)
288: (4)
289: (4)
290: (4)
291: (4)
292: (4)
293: (4)
294: (4)
295: (4)
296: (8)
297: (8)
298: (8)
299: (8)

        >>> A = np.arange(8).reshape((2,2,2))
        >>> A
        array([[[0, 1],
                  [2, 3]],
                  [[4, 5],
                  [6, 7]]])
        >>> np.flip(A, 0)
        array([[[4, 5],
                  [6, 7]],
                  [[0, 1],
                  [2, 3]]])
        >>> np.flip(A, 1)
        array([[[2, 3],
                  [0, 1]],
                  [[6, 7],
                  [4, 5]]])
        >>> np.flip(A)
        array([[[7, 6],
                  [5, 4]],
                  [[3, 2],
                  [1, 0]]])
        >>> np.flip(A, (0, 2))
        array([[[5, 4],
                  [7, 6]],
                  [[1, 0],
                  [3, 2]]])
        >>> A = np.random.randn(3,4,5)
        >>> np.all(np.flip(A,2) == A[:, :, ::-1,...])
        True
        """
        if not hasattr(m, 'ndim'):
            m = asarray(m)
        if axis is None:
            indexer = (np.s_[::-1],) * m.ndim
        else:
            axis = _nx.normalize_axis_tuple(axis, m.ndim)
            indexer = [np.s_[:] * m.ndim
                       for ax in axis:
                           indexer[ax] = np.s_[::-1]]
            indexer = tuple(indexer)
        return m[indexer]
    @set_module('numpy')
    def iterable(y):
        """
        Check whether or not an object can be iterated over.
        Parameters
        -----
        y : object
            Input object.
        Returns
        -----
        b : bool
            Return ``True`` if the object has an iterator method or is a
            sequence and ``False`` otherwise.
        Examples
        -----
        >>> np.iterable([1, 2, 3])
        True
        >>> np.iterable(2)
        False
        Notes
        -----
        In most cases, the results of ``np.iterable(obj)`` are consistent with
        ```isinstance(obj, collections.abc.Iterable)```. One notable exception is
        the treatment of 0-dimensional arrays::
            >>> from collections.abc import Iterable
            >>> a = np.array(1.0) # 0-dimensional numpy array
            >>> isinstance(a, Iterable)
            True

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

300: (8)          >>> np.iterable(a)
301: (8)          False
302: (4)          """
303: (4)          try:
304: (8)            iter(y)
305: (4)          except TypeError:
306: (8)            return False
307: (4)          return True
308: (0)          def _average_dispatcher(a, axis=None, weights=None, returned=None, *,
309: (24)            keepdims=None):
310: (4)          return (a, weights)
311: (0)          @array_function_dispatch(_average_dispatcher)
312: (0)          def average(a, axis=None, weights=None, returned=False, *,
313: (12)            keepdims=np._NoValue):
314: (4)          """
315: (4)          Compute the weighted average along the specified axis.
316: (4)          Parameters
317: (4)          -----
318: (4)          a : array_like
319: (8)          Array containing data to be averaged. If `a` is not an array, a
320: (8)          conversion is attempted.
321: (4)          axis : None or int or tuple of ints, optional
322: (8)          Axis or axes along which to average `a`. The default,
323: (8)          axis=None, will average over all of the elements of the input array.
324: (8)          If axis is negative it counts from the last to the first axis.
325: (8)          .. versionadded:: 1.7.0
326: (8)          If axis is a tuple of ints, averaging is performed on all of the axes
327: (8)          specified in the tuple instead of a single axis or all the axes as
328: (8)          before.
329: (4)          weights : array_like, optional
330: (8)          An array of weights associated with the values in `a`. Each value in
331: (8)          `a` contributes to the average according to its associated weight.
332: (8)          The weights array can either be 1-D (in which case its length must be
333: (8)          the size of `a` along the given axis) or of the same shape as `a`.
334: (8)          If `weights=None`, then all data in `a` are assumed to have a
335: (8)          weight equal to one. The 1-D calculation is::
336: (12)          avg = sum(a * weights) / sum(weights)
337: (8)          The only constraint on `weights` is that `sum(weights)` must not be 0.
338: (4)          returned : bool, optional
339: (8)          Default is `False`. If `True`, the tuple (`average`, `sum_of_weights`)
340: (8)          is returned, otherwise only the average is returned.
341: (8)          If `weights=None`, `sum_of_weights` is equivalent to the number of
342: (8)          elements over which the average is taken.
343: (4)          keepdims : bool, optional
344: (8)          If this is set to True, the axes which are reduced are left
345: (8)          in the result as dimensions with size one. With this option,
346: (8)          the result will broadcast correctly against the original `a`.
347: (8)          *Note:* `keepdims` will not work with instances of `numpy.matrix`
348: (8)          or other classes whose methods do not support `keepdims`.
349: (8)          .. versionadded:: 1.23.0
350: (4)          Returns
351: (4)          -----
352: (4)          retval, [sum_of_weights] : array_type or double
353: (8)          Return the average along the specified axis. When `returned` is
`True`,
354: (8)          return a tuple with the average as the first element and the sum
355: (8)          of the weights as the second element. `sum_of_weights` is of the
356: (8)          same type as `retval`. The result dtype follows a general pattern.
357: (8)          If `weights` is None, the result dtype will be that of `a` , or
`float64` .
358: (8)          if `a` is integral. Otherwise, if `weights` is not None and `a` is
non-
359: (8)          integral, the result type will be the type of lowest precision capable
of
360: (8)          representing values of both `a` and `weights`. If `a` happens to be
361: (8)          integral, the previous rules still applies but the result dtype will
362: (8)          at least be ``float64``.
363: (4)          Raises
364: (4)          -----

```

```

365: (4) ZeroDivisionError
366: (8)     When all weights along axis are zero. See `numpy.ma.average` for a
367: (8)     version robust to this type of error.
368: (4) TypeError
369: (8)     When the length of 1D `weights` is not the same as the shape of `a`
370: (8)     along axis.
371: (4) See Also
372: (4) -----
373: (4) mean
374: (4) ma.average : average for masked arrays -- useful if your data contains
375: (17)         "missing" values
376: (4) numpy.result_type : Returns the type that results from applying the
377: (24)         numpy type promotion rules to the arguments.
378: (4) Examples
379: (4) -----
380: (4) >>> data = np.arange(1, 5)
381: (4) >>> data
382: (4) array([1, 2, 3, 4])
383: (4) >>> np.average(data)
384: (4) 2.5
385: (4) >>> np.average(np.arange(1, 11), weights=np.arange(10, 0, -1))
386: (4) 4.0
387: (4) >>> data = np.arange(6).reshape((3, 2))
388: (4) >>> data
389: (4) array([[0, 1],
390: (11)             [2, 3],
391: (11)             [4, 5]])
392: (4) >>> np.average(data, axis=1, weights=[1./4, 3./4])
393: (4) array([0.75, 2.75, 4.75])
394: (4) >>> np.average(data, weights=[1./4, 3./4])
395: (4) Traceback (most recent call last):
396: (8) ...
397: (4) TypeError: Axis must be specified when shapes of a and weights differ.
398: (4) >>> a = np.ones(5, dtype=np.float128)
399: (4) >>> w = np.ones(5, dtype=np.complex64)
400: (4) >>> avg = np.average(a, weights=w)
401: (4) >>> print(avg.dtype)
402: (4) complex256
403: (4) With ``keepdims=True``, the following result has shape (3, 1).
404: (4) >>> np.average(data, axis=1, keepdims=True)
405: (4) array([[0.5],
406: (11)             [2.5],
407: (11)             [4.5]])
408: (4) """
409: (4) a = np.asanyarray(a)
410: (4) if keepdims is np._NoValue:
411: (8)     keepdims_kw = {}
412: (4) else:
413: (8)     keepdims_kw = {'keepdims': keepdims}
414: (4) if weights is None:
415: (8)     avg = a.mean(axis, **keepdims_kw)
416: (8)     avg_as_array = np.asanyarray(avg)
417: (8)     scl = avg_as_array.dtype.type(a.size/avg_as_array.size)
418: (4) else:
419: (8)     wgt = np.asanyarray(weights)
420: (8)     if issubclass(a.dtype.type, (np.integer, np.bool_)):
421: (12)         result_dtype = np.result_type(a.dtype, wgt.dtype, 'f8')
422: (8)     else:
423: (12)         result_dtype = np.result_type(a.dtype, wgt.dtype)
424: (8)     if a.shape != wgt.shape:
425: (12)         if axis is None:
426: (16)             raise TypeError(
427: (20)                 "Axis must be specified when shapes of a and weights "
428: (20)                 "differ.")
429: (12)         if wgt.ndim != 1:
430: (16)             raise TypeError(
431: (20)                 "1D weights expected when shapes of a and weights "
differ.")
432: (12)         if wgt.shape[0] != a.shape[axis]:

```

```

433: (16)                     raise ValueError(
434: (20)                         "Length of weights not compatible with specified axis.")
435: (12)                         wgt = np.broadcast_to(wgt, (a.ndim-1)*(1,) + wgt.shape)
436: (12)                         wgt = wgt.swapaxes(-1, axis)
437: (8)                         scl = wgt.sum(axis=axis, dtype=result_dtype, **keepdims_kw)
438: (8)                         if np.any(scl == 0.0):
439: (12)                             raise ZeroDivisionError(
440: (16)                                 "Weights sum to zero, can't be normalized")
441: (8)                         avg = avg_as_array = np.multiply(a, wgt,
442: (26)                                         dtype=result_dtype).sum(axis, **keepdims_kw) / scl
443: (4)                         if returned:
444: (8)                             if scl.shape != avg_as_array.shape:
445: (12)                                 scl = np.broadcast_to(scl, avg_as_array.shape).copy()
446: (8)                                 return avg, scl
447: (4)                         else:
448: (8)                             return avg
449: (0) @set_module('numpy')
450: (0) def asarray_chkfinite(a, dtype=None, order=None):
451: (4)     """Convert the input to an array, checking for NaNs or Infs.
452: (4)     Parameters
453: (4)     -----
454: (4)     a : array_like
455: (8)         Input data, in any form that can be converted to an array. This
456: (8)         includes lists, lists of tuples, tuples, tuples of tuples, tuples
457: (8)         of lists and ndarrays. Success requires no NaNs or Infs.
458: (4)     dtype : data-type, optional
459: (8)         By default, the data-type is inferred from the input data.
460: (4)     order : {'C', 'F', 'A', 'K'}, optional
461: (8)         Memory layout. 'A' and 'K' depend on the order of input array a.
462: (8)         'C' row-major (C-style),
463: (8)         'F' column-major (Fortran-style) memory representation.
464: (8)         'A' (any) means 'F' if `a` is Fortran contiguous, 'C' otherwise
465: (8)         'K' (keep) preserve input order
466: (8)         Defaults to 'C'.
467: (4)     Returns
468: (4)     -----
469: (4)     out : ndarray
470: (8)         Array interpretation of `a`. No copy is performed if the input
471: (8)         is already an ndarray. If `a` is a subclass of ndarray, a base
472: (8)         class ndarray is returned.
473: (4)     Raises
474: (4)     -----
475: (4)     ValueError
476: (8)         Raises ValueError if `a` contains NaN (Not a Number) or Inf
477: (8)         (Infinity).
478: (4)     See Also
479: (4)     -----
480: (4)     asarray : Create an array.
481: (4)     asanyarray : Similar function which passes through subclasses.
482: (4)     ascontiguousarray : Convert input to a contiguous array.
483: (4)     asfarray : Convert input to a floating point ndarray.
484: (21)     asfortranarray : Convert input to an ndarray with column-major
485: (4)         memory order.
486: (4)     fromiter : Create an array from an iterator.
487: (19)     fromfunction : Construct an array by executing a function on grid
488: (4)         positions.
489: (4)     Examples
490: (4)     -----
491: (4)     Convert a list into an array. If all elements are finite
492: (4)         ``asarray_chkfinite`` is identical to ``asarray``.
493: (4)         >>> a = [1, 2]
494: (4)         >>> np.asarray_chkfinite(a, dtype=float)
495: (4)         array([1., 2.])
496: (4)         Raises ValueError if array_like contains Nans or Infs.
497: (4)         >>> a = [1, 2, np.inf]
498: (4)         >>> try:
499: (4)             ...     np.asarray_chkfinite(a)
500: (4)             ... except ValueError:

```

```

501: (4) ...
502: (4) ValueError
503: (4)
504: (4) """
505: (4)     a = asarray(a, dtype=dtype, order=order)
506: (8)     if a.dtype.char in typecodes['AllFloat'] and not np.isfinite(a).all():
507: (12)         raise ValueError(
508: (4)             "array must not contain infs or NaNs")
509: (0)     return a
510: (4) def _piecewise_dispatcher(x, condlist, funclist, *args, **kw):
511: (4)     yield x
512: (8)     if np.iterable(condlist):
513: (0)         yield from condlist
514: (0) @array_function_dispatch(_piecewise_dispatcher)
515: (4) def piecewise(x, condlist, funclist, *args, **kw):
516: (4) """
517: (4)     Evaluate a piecewise-defined function.
518: (4)     Given a set of conditions and corresponding functions, evaluate each
519: (4)     function on the input data wherever its condition is true.
520: (4)     Parameters
521: (4)     -----
522: (8)     x : ndarray or scalar
523: (4)         The input domain.
524: (8)     condlist : list of bool arrays or bool scalars
525: (8)         Each boolean array corresponds to a function in `funclist`. Wherever
526: (8)         `condlist[i]` is True, `funclist[i](x)` is used as the output value.
527: (8)         Each boolean array in `condlist` selects a piece of `x`,
528: (8)         and should therefore be of the same shape as `x`.
529: (8)         The length of `condlist` must correspond to that of `funclist`.
530: (8)         If one extra function is given, i.e. if
531: (8)             ``len(funclist) == len(condlist) + 1``,
532: (8)             then that extra function
533: (8)             is the default value, used wherever all conditions are false.
534: (8)     funclist : list of callables, f(x,*args,**kw), or scalars
535: (8)         Each function is evaluated over `x` wherever its corresponding
536: (8)         condition is True. It should take a 1d array as input and give an 1d
537: (8)         array or a scalar value as output. If, instead of a callable,
538: (4)             a scalar is provided then a constant function (``lambda x: scalar``)
539: (8)         assumed.
540: (8)     args : tuple, optional
541: (8)         Any further arguments given to `piecewise` are passed to the functions
542: (8)         upon execution, i.e., if called ``piecewise(..., ..., 1, 'a')``,
543: (8)         then each function is called as ``f(x, 1, 'a')``.
544: (8)     kw : dict, optional
545: (8)         Keyword arguments used in calling `piecewise` are passed to the
546: (8)         functions upon execution, i.e., if called
547: (4)             ``piecewise(..., ..., alpha=1)``,
548: (4)             then each function is called as
549: (4)                 ``f(x, alpha=1)``.
550: (8)     Returns
551: (8)     -----
552: (8)     out : ndarray
553: (8)         The output is the same shape and type as x and is found by
554: (8)         calling the functions in `funclist` on the appropriate portions of
555: (8)         `x`,
556: (8)         as defined by the boolean arrays in `condlist`. Portions not covered
557: (8)         by any condition have a default value of 0.
558: (4)     See Also
559: (4)     -----
560: (4)     choose, select, where
561: (4)     Notes
562: (4)     -----
563: (12)     This is similar to choose or select, except that functions are
564: (12)     evaluated on elements of `x` that satisfy the corresponding condition from
565: (6)     `condlist`.
566: (12)     The result is::
567: (12)         |--
568: (12)             |funclist[0](x[condlist[0]])
569: (12)             |out = |funclist[1](x[condlist[1]])
570: (12)             |...
571: (12)             |funclist[n2](x[condlist[n2]])

```

```

568: (12)          | --
569: (4)          Examples
570: (4)          -----
571: (4)          Define the sigma function, which is -1 for ``x < 0`` and +1 for ``x >
0``.
572: (4)          >>> x = np.linspace(-2.5, 2.5, 6)
573: (4)          >>> np.piecewise(x, [x < 0, x >= 0], [-1, 1])
574: (4)          array([-1., -1., -1., 1., 1., 1.])
575: (4)          Define the absolute value, which is ``-x`` for ``x < 0`` and ``x`` for
576: (4)          ``x >= 0``.
577: (4)          >>> np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x])
578: (4)          array([2.5, 1.5, 0.5, 0.5, 1.5, 2.5])
579: (4)          Apply the same function to a scalar value.
580: (4)          >>> y = -2
581: (4)          >>> np.piecewise(y, [y < 0, y >= 0], [lambda x: -x, lambda x: x])
582: (4)          array(2)
583: (4)          """
584: (4)          x = asanyarray(x)
585: (4)          n2 = len(funclist)
586: (4)          if isscalar(condlist) or (
587: (12)              not isinstance(condlist[0], (list, ndarray)) and x.ndim != 0):
588: (8)              condlist = [condlist]
589: (4)          condlist = asarray(condlist, dtype=bool)
590: (4)          n = len(condlist)
591: (4)          if n == n2 - 1: # compute the "otherwise" condition.
592: (8)              condelse = ~np.any(condlist, axis=0, keepdims=True)
593: (8)              condlist = np.concatenate([condlist, condelse], axis=0)
594: (8)              n += 1
595: (4)          elif n != n2:
596: (8)              raise ValueError(
597: (12)                  "with {} condition(s), either {} or {} functions are expected"
598: (12)                  .format(n, n, n+1)
599: (8)              )
600: (4)          y = zeros_like(x)
601: (4)          for cond, func in zip(condlist, funclist):
602: (8)              if not isinstance(func, collections.abc.Callable):
603: (12)                  y[cond] = func
604: (8)              else:
605: (12)                  vals = x[cond]
606: (12)                  if vals.size > 0:
607: (16)                      y[cond] = func(vals, *args, **kw)
608: (4)          return y
609: (0)          def _select_dispatcher(condlist, choicelist, default=None):
610: (4)              yield from condlist
611: (4)              yield from choicelist
612: (0)          @array_function_dispatch(_select_dispatcher)
613: (0)          def select(condlist, choicelist, default=0):
614: (4)              """
615: (4)              Return an array drawn from elements in choicelist, depending on
conditions.
616: (4)          Parameters
617: (4)          -----
618: (4)          condlist : list of bool ndarrays
619: (8)              The list of conditions which determine from which array in
`choicelist`
620: (8)              the output elements are taken. When multiple conditions are satisfied,
621: (8)              the first one encountered in `condlist` is used.
622: (4)          choicelist : list of ndarrays
623: (8)              The list of arrays from which the output elements are taken. It has
624: (8)              to be of the same length as `condlist`.
625: (4)          default : scalar, optional
626: (8)              The element inserted in `output` when all conditions evaluate to
False.
627: (4)          Returns
628: (4)          -----
629: (4)          output : ndarray
630: (8)              The output at position m is the m-th element of the array in
631: (8)              `choicelist` where the m-th element of the corresponding array in
632: (8)              `condlist` is True.

```

```

633: (4)          See Also
634: (4)          -----
635: (4)          where : Return elements from one of two arrays depending on condition.
636: (4)          take, choose, compress, diag, diagonal
637: (4)          Examples
638: (4)          -----
639: (4)          >>> x = np.arange(6)
640: (4)          >>> condlist = [x<3, x>3]
641: (4)          >>> choicelist = [x, x**2]
642: (4)          >>> np.select(condlist, choicelist, 42)
643: (4)          array([ 0,  1,  2, 42, 16, 25])
644: (4)          >>> condlist = [x<=4, x>3]
645: (4)          >>> choicelist = [x, x**2]
646: (4)          >>> np.select(condlist, choicelist, 55)
647: (4)          array([ 0,  1,  2,  3,  4, 25])
648: (4)          """
649: (4)          if len(condlist) != len(choicelist):
650: (8)              raise ValueError(
651: (12)                  'list of cases must be same length as list of conditions')
652: (4)          if len(condlist) == 0:
653: (8)              raise ValueError("select with an empty condition list is not
possible")
654: (4)          choicelist = [np.asarray(choice) for choice in choicelist]
655: (4)          try:
656: (8)              intermediate_dtype = np.result_type(*choicelist)
657: (4)          except TypeError as e:
658: (8)              msg = f'Choicelist elements do not have a common dtype: {e}'
659: (8)              raise TypeError(msg) from None
660: (4)          default_array = np.asarray(default)
661: (4)          choicelist.append(default_array)
662: (4)          try:
663: (8)              dtype = np.result_type(intermediate_dtype, default_array)
664: (4)          except TypeError as e:
665: (8)              msg = f'Choicelists and default value do not have a common dtype: {e}'
666: (8)              raise TypeError(msg) from None
667: (4)          condlist = np.broadcast_arrays(*condlist)
668: (4)          choicelist = np.broadcast_arrays(*choicelist)
669: (4)          for i, cond in enumerate(condlist):
670: (8)              if cond.dtype.type is not np.bool_:
671: (12)                  raise TypeError(
672: (16)                      'invalid entry {} in condlist: should be boolean
ndarray'.format(i))
673: (4)          if choicelist[0].ndim == 0:
674: (8)              result_shape = condlist[0].shape
675: (4)          else:
676: (8)              result_shape = np.broadcast_arrays(condlist[0], choicelist[0])
677: (4)          result = np.full(result_shape, choicelist[-1], dtype)
678: (4)          choicelist = choicelist[-2::-1]
679: (4)          condlist = condlist[::-1]
680: (4)          for choice, cond in zip(choicelist, condlist):
681: (8)              np.copyto(result, choice, where=cond)
682: (4)          return result
683: (0)          def _copy_dispatcher(a, order=None, subok=None):
684: (4)              return (a,)
685: (0)          @array_function_dispatch(_copy_dispatcher)
686: (0)          def copy(a, order='K', subok=False):
687: (4)              """
688: (4)              Return an array copy of the given object.
689: (4)              Parameters
690: (4)              -----
691: (4)              a : array_like
692: (8)                  Input data.
693: (4)              order : {'C', 'F', 'A', 'K'}, optional
694: (8)                  Controls the memory layout of the copy. 'C' means C-order,
695: (8)                  'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
696: (8)                  'C' otherwise. 'K' means match the layout of `a` as closely
697: (8)                  as possible. (Note that this function and :meth:`ndarray.copy` are
very

```

```

698: (8)           similar, but have different default values for their order-
699: (8)           arguments.)
700: (4)           subok : bool, optional
701: (8)           If True, then sub-classes will be passed-through, otherwise the
702: (8)           returned array will be forced to be a base-class array (defaults to
False).
703: (8)           .. versionadded:: 1.19.0
704: (4)           Returns
705: (4)           -----
706: (4)           arr : ndarray
707: (8)           Array interpretation of `a`.
708: (4)           See Also
709: (4)           -----
710: (4)           ndarray.copy : Preferred method for creating an array copy
711: (4)           Notes
712: (4)           -----
713: (4)           This is equivalent to:
714: (4)           >>> np.array(a, copy=True) #doctest: +SKIP
715: (4)           Examples
716: (4)           -----
717: (4)           Create an array x, with a reference y and a copy z:
718: (4)           >>> x = np.array([1, 2, 3])
719: (4)           >>> y = x
720: (4)           >>> z = np.copy(x)
721: (4)           Note that, when we modify x, y changes, but not z:
722: (4)           >>> x[0] = 10
723: (4)           >>> x[0] == y[0]
724: (4)           True
725: (4)           >>> x[0] == z[0]
726: (4)           False
727: (4)           Note that, np.copy clears previously set WRITEABLE=False flag.
728: (4)           >>> a = np.array([1, 2, 3])
729: (4)           >>> a.flags["WRITEABLE"] = False
730: (4)           >>> b = np.copy(a)
731: (4)           >>> b.flags["WRITEABLE"]
732: (4)           True
733: (4)           >>> b[0] = 3
734: (4)           >>> b
735: (4)           array([3, 2, 3])
736: (4)           Note that np.copy is a shallow copy and will not copy object
737: (4)           elements within arrays. This is mainly important for arrays
738: (4)           containing Python objects. The new array will contain the
739: (4)           same object which may lead to surprises if that object can
740: (4)           be modified (is mutable):
741: (4)           >>> a = np.array([1, 'm', [2, 3, 4]], dtype=object)
742: (4)           >>> b = np.copy(a)
743: (4)           >>> b[2][0] = 10
744: (4)           >>> a
745: (4)           array([1, 'm', list([10, 3, 4])], dtype=object)
746: (4)           To ensure all elements within an ``object`` array are copied,
747: (4)           use `copy.deepcopy`:
748: (4)           >>> import copy
749: (4)           >>> a = np.array([1, 'm', [2, 3, 4]], dtype=object)
750: (4)           >>> c = copy.deepcopy(a)
751: (4)           >>> c[2][0] = 10
752: (4)           >>> c
753: (4)           array([1, 'm', list([10, 3, 4])], dtype=object)
754: (4)           >>> a
755: (4)           array([1, 'm', list([2, 3, 4])], dtype=object)
756: (4)           """
757: (4)           return array(a, order=order, subok=subok, copy=True)
758: (0)           def _gradient_dispatcher(f, *varargs, axis=None, edge_order=None):
759: (4)               yield f
760: (4)               yield from varargs
761: (0)               @array_function_dispatch(_gradient_dispatcher)
762: (0)               def gradient(f, *varargs, axis=None, edge_order=1):
763: (4)                   """
764: (4)                   Return the gradient of an N-dimensional array.
765: (4)                   The gradient is computed using second order accurate central differences

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

766: (4)           in the interior points and either first or second order accurate one-sides
767: (4)           (forward or backwards) differences at the boundaries.
768: (4)           The returned gradient hence has the same shape as the input array.
769: (4)           Parameters
770: (4)           -----
771: (4)           f : array_like
772: (8)             An N-dimensional array containing samples of a scalar function.
773: (4)           varargs : list of scalar or array, optional
774: (8)             Spacing between f values. Default unitary spacing for all dimensions.
775: (8)             Spacing can be specified using:
776: (8)               1. single scalar to specify a sample distance for all dimensions.
777: (8)               2. N scalars to specify a constant sample distance for each dimension.
778: (11)                 i.e. `dx`, `dy`, `dz`, ...
779: (8)               3. N arrays to specify the coordinates of the values along each
780: (11)                 dimension of F. The length of the array must match the size of
781: (11)                 the corresponding dimension
782: (8)               4. Any combination of N scalars/arrays with the meaning of 2. and 3.
783: (8)             If `axis` is given, the number of varargs must equal the number of
784: axes.
785: (8)             Default: 1.
786: (4)           edge_order : {1, 2}, optional
787: (8)             Gradient is calculated using N-th order accurate differences
788: (8)             at the boundaries. Default: 1.
789: (8)               .. versionadded:: 1.9.1
790: (4)           axis : None or int or tuple of ints, optional
791: (8)             Gradient is calculated only along the given axis or axes
792: axes
793: (8)             The default (axis = None) is to calculate the gradient for all the
794: of the input array. axis may be negative, in which case it counts from
795: the last to the first axis.
796: (8)               .. versionadded:: 1.11.0
797: (4)           Returns
798: (8)           -----
799: (4)           gradient : ndarray or list of ndarray
800: (8)             A list of ndarrays (or a single ndarray if there is only one
801: dimension)
802: (4)             corresponding to the derivatives of f with respect to each dimension.
803: (4)             Each derivative has the same shape as f.
804: (4)           Examples
805: (4)           -----
806: (4)           >>> f = np.array([1, 2, 4, 7, 11, 16], dtype=float)
807: (4)           >>> np.gradient(f)
808: (4)             array([1. , 1.5, 2.5, 3.5, 4.5, 5. ])
809: coordinates
810: (4)             Spacing can be also specified with an array that represents the
811: of the values F along the dimensions.
812: (4)             For instance a uniform spacing:
813: (4)             >>> x = np.arange(f.size)
814: (4)             >>> np.gradient(f, x)
815: (4)             array([1. , 1.5, 2.5, 3.5, 4.5, 5. ])
816: (4)             Or a non uniform one:
817: (4)             >>> x = np.array([0., 1., 1.5, 3.5, 4., 6.], dtype=float)
818: (4)             >>> np.gradient(f, x)
819: (4)             array([1. , 3. , 3.5, 6.7, 6.9, 2.5])
820: (4)             For two dimensional arrays, the return will be two arrays ordered by
821: axis. In this example the first array stands for the gradient in
822: rows and the second one in columns direction:
823: (4)             >>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]]), dtype=float))
824: (4)             [array([[ 2.,  2., -1.],
825: (11)                   [ 2.,  2., -1.]]), array([[1. , 2.5, 4. ],
826: (11)                   [1. , 1. , 1. ]]])
827: (4)             In this example the spacing is also specified:
828: (4)             uniform for axis=0 and non uniform for axis=1
829: (4)             >>> dx = 2.
830: (4)             >>> y = [1., 1.5, 3.5]
831: (4)             >>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]]), dtype=float), dx, y)
832: (4)             [array([[ 1. ,  1. , -0.5],
```

```

831: (11) [ 1. , 1. , -0.5]], array([[2. , 2. , 2. ],
832: (11) [2. , 1.7, 0.5]])
833: (4) It is possible to specify how boundaries are treated using `edge_order`:
834: (4) >>> x = np.array([0, 1, 2, 3, 4])
835: (4) >>> f = x**2
836: (4) >>> np.gradient(f, edge_order=1)
837: (4) array([1., 2., 4., 6., 7.])
838: (4) >>> np.gradient(f, edge_order=2)
839: (4) array([0., 2., 4., 6., 8.])
840: (4) The `axis` keyword can be used to specify a subset of axes of which the
841: (4) gradient is calculated
842: (4) >>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]]), dtype=float), axis=0)
843: (4) array([[ 2.,  2., -1.],
844: (11) [ 2.,  2., -1.]])
845: (4) Notes
846: (4) -----
847: (4) Assuming that :math:`f \in C^3` (i.e., :math:`f` has at least 3
848: (4) continuous
849: (4) derivatives) and let :math:`h_{\cdot i}` be a non-homogeneous stepsize, we
gradient minimize the "consistency error" :math:`\|\eta_i\|` between the true
850: (4) and its estimate from a linear combination of the neighboring grid-points:
851: (4) .. math::
852: (8) 
$$\|\eta_i\| = f_{\cdot i}^{\prime \prime}(x_i) - \left[ \alpha f(x_i) + \beta f(x_i + h_d) + \gamma f(x_i - h_s) \right]$$

853: (20)
854: (28)
855: (28)
856: (20)
857: (4) By substituting :math:`f(x_i + h_d)` and :math:`f(x_i - h_s)` with their Taylor series expansion, this translates into solving
858: (4) the following the linear system:
859: (4) .. math::
860: (4) .. math::
861: (8) 
$$\begin{array}{l} \left( \alpha + \beta + \gamma \right) h_d = 0 \\ \beta h_d - \gamma h_s = 1 \\ \beta h_d^2 + \gamma h_s^2 = 0 \end{array}$$

862: (12)
863: (16)
864: (16)
865: (16)
866: (12)
867: (8)
868: (4) The resulting approximation of :math:`f_{\cdot i}^{\prime \prime}(1)` is the following:
869: (4) .. math::
870: (8) .. math::
871: (12) 
$$\hat{f}_{\cdot i}^{\prime \prime}(1) = \frac{h_s^2 f(x_i + h_d) + (h_d^2 - h_s^2) f(x_i) - h_d^2 f(x_i - h_s)}{2 h_d^2}$$

872: (16)
873: (16)
874: (16)
875: (16)
876: (12)
877: (32)
878: (32)
879: (4) It is worth noting that if :math:`h_s = h_d` (i.e., data are evenly spaced)
880: (4) we find the standard second order approximation:
881: (4) .. math::
882: (4) .. math::
883: (8) .. math::
884: (12) 
$$\hat{f}_{\cdot i}^{\prime \prime}(1) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + \frac{h^2}{12} \mathcal{O}(h^2)$$

885: (12)
886: (4) With a similar procedure the forward/backward approximations used for
887: (4) boundaries can be derived.
888: (4) References
889: (4) -----
890: (4) .. [1] Quarteroni A., Sacco R., Saleri F. (2007) Numerical Mathematics
891: (12) (Texts in Applied Mathematics). New York: Springer.
892: (4)
893: (12)
894: (4)
895: (12)
896: (12)
897: (12) .. [2] Durran D. R. (1999) Numerical Methods for Wave Equations
898: (4) in Geophysical Fluid Dynamics. New York: Springer.
899: (4) .. [3] Fornberg B. (1988) Generation of Finite Difference Formulas on
900: (12) Arbitrarily Spaced Grids,
901: (12) Mathematics of Computation 51, no. 184 : 699-706.
902: (12) `PDF <http://www.ams.org/journals/mcom/1988-51-184/>
```

```

898: (12)                               S0025-5718-1988-0935077-0/S0025-5718-1988-0935077-0.pdf>`_.
899: (4)
900: (4)
901: (4)
902: (4)
903: (8)
904: (4)
905: (8)
906: (4)
907: (4)
908: (4)
909: (8)
910: (4)
911: (8)
912: (4)
913: (8)
914: (8)
915: (12)
916: (12)
917: (16)
918: (12)
919: (16)
920: (12)
921: (16)
922: (33)
923: (12)
924: (16)
925: (12)
926: (12)
927: (16)
928: (12)
929: (4)
930: (8)
931: (4)
932: (8)
933: (4)
934: (4)
935: (4)
936: (4)
937: (4)
938: (4)
939: (4)
940: (8)
941: (8)
942: (4)
943: (8)
944: (4)
945: (8)
946: (4)
947: (8)
948: (12)
949: (8)
950: (4)
951: (8)
952: (12)
953: (16)
954: (16)
955: (8)
956: (8)
957: (8)
958: (8)
959: (8)
960: (8)
961: (8)
962: (12)
ax_dx)
963: (8)
964: (12)
965: (12)

         """
f = np.asarray(f)
N = f.ndim # number of dimensions
if axis is None:
    axes = tuple(range(N))
else:
    axes = _nx.normalize_axis_tuple(axis, N)
len_axes = len(axes)
n = len(varargs)
if n == 0:
    dx = [1.0] * len_axes
elif n == 1 and np.ndim(varargs[0]) == 0:
    dx = varargs * len_axes
elif n == len_axes:
    dx = list(varargs)
    for i, distances in enumerate(dx):
        distances = np.asarray(distances)
        if distances.ndim == 0:
            continue
        elif distances.ndim != 1:
            raise ValueError("distances must be either scalars or 1d")
        if len(distances) != f.shape[axes[i]]:
            raise ValueError("when 1d, distances must match "
                            "the length of the corresponding dimension")
        if np.issubdtype(distances.dtype, np.integer):
            distances = distances.astype(np.float64)
        diffx = np.diff(distances)
        if (diffx == diffx[0]).all():
            diffx = diffx[0]
        dx[i] = diffx
else:
    raise TypeError("invalid number of arguments")
if edge_order > 2:
    raise ValueError("'edge_order' greater than 2 not supported")
outvals = []
slice1 = [slice(None)]*N
slice2 = [slice(None)]*N
slice3 = [slice(None)]*N
slice4 = [slice(None)]*N
otype = f.dtype
if otype.type is np.datetime64:
    otype = np.dtype(otype.name.replace('datetime', 'timedelta'))
    f = f.view(otype)
elif otype.type is np.timedelta64:
    pass
elif np.issubdtype(otype, np.inexact):
    pass
else:
    if np.issubdtype(otype, np.integer):
        f = f.astype(np.float64)
    otype = np.float64
for axis, ax_dx in zip(axes, dx):
    if f.shape[axis] < edge_order + 1:
        raise ValueError(
            "Shape of array too small to calculate a numerical gradient, "
            "at least (edge_order + 1) elements are required.")
    out = np.empty_like(f, dtype=otype)
    uniform_spacing = np.ndim(ax_dx) == 0
    slice1[axis] = slice(1, -1)
    slice2[axis] = slice(None, -2)
    slice3[axis] = slice(1, -1)
    slice4[axis] = slice(2, None)
    if uniform_spacing:
        out[tuple(slice1)] = (f[tuple(slice4)] - f[tuple(slice2)]) / (2. *
ax_dx)
    else:
        dx1 = ax_dx[0:-1]
        dx2 = ax_dx[1:]

```

```

966: (12)           a = -(dx2)/(dx1 * (dx1 + dx2))
967: (12)           b = (dx2 - dx1) / (dx1 * dx2)
968: (12)           c = dx1 / (dx2 * (dx1 + dx2))
969: (12)           shape = np.ones(N, dtype=int)
970: (12)           shape[axis] = -1
971: (12)           a.shape = b.shape = c.shape = shape
972: (12)           out[tuple(slice1)] = a * f[tuple(slice2)] + b * f[tuple(slice3)] +
c * f[tuple(slice4)]
973: (8)            if edge_order == 1:
974: (12)              slice1[axis] = 0
975: (12)              slice2[axis] = 1
976: (12)              slice3[axis] = 0
977: (12)              dx_0 = ax_dx if uniform_spacing else ax_dx[0]
978: (12)              out[tuple(slice1)] = (f[tuple(slice2)] - f[tuple(slice3)]) / dx_0
979: (12)              slice1[axis] = -1
980: (12)              slice2[axis] = -1
981: (12)              slice3[axis] = -2
982: (12)              dx_n = ax_dx if uniform_spacing else ax_dx[-1]
983: (12)              out[tuple(slice1)] = (f[tuple(slice2)] - f[tuple(slice3)]) / dx_n
984: (8)            else:
985: (12)              slice1[axis] = 0
986: (12)              slice2[axis] = 0
987: (12)              slice3[axis] = 1
988: (12)              slice4[axis] = 2
989: (12)              if uniform_spacing:
990: (16)                  a = -1.5 / ax_dx
991: (16)                  b = 2. / ax_dx
992: (16)                  c = -0.5 / ax_dx
993: (12)              else:
994: (16)                  dx1 = ax_dx[0]
995: (16)                  dx2 = ax_dx[1]
996: (16)                  a = -(2. * dx1 + dx2)/(dx1 * (dx1 + dx2))
997: (16)                  b = (dx1 + dx2) / (dx1 * dx2)
998: (16)                  c = - dx1 / (dx2 * (dx1 + dx2))
999: (12)          out[tuple(slice1)] = a * f[tuple(slice2)] + b * f[tuple(slice3)] +
c * f[tuple(slice4)]
1000: (12)         slice1[axis] = -1
1001: (12)         slice2[axis] = -3
1002: (12)         slice3[axis] = -2
1003: (12)         slice4[axis] = -1
1004: (12)         if uniform_spacing:
1005: (16)             a = 0.5 / ax_dx
1006: (16)             b = -2. / ax_dx
1007: (16)             c = 1.5 / ax_dx
1008: (12)         else:
1009: (16)             dx1 = ax_dx[-2]
1010: (16)             dx2 = ax_dx[-1]
1011: (16)             a = (dx2) / (dx1 * (dx1 + dx2))
1012: (16)             b = - (dx2 + dx1) / (dx1 * dx2)
1013: (16)             c = (2. * dx2 + dx1) / (dx2 * (dx1 + dx2))
1014: (12)         out[tuple(slice1)] = a * f[tuple(slice2)] + b * f[tuple(slice3)] +
c * f[tuple(slice4)]
1015: (8)          outvals.append(out)
1016: (8)          slice1[axis] = slice(None)
1017: (8)          slice2[axis] = slice(None)
1018: (8)          slice3[axis] = slice(None)
1019: (8)          slice4[axis] = slice(None)
1020: (4)          if len_axes == 1:
1021: (8)              return outvals[0]
1022: (4)          elif np._using_numpy2_behavior():
1023: (8)              return tuple(outvals)
1024: (4)          else:
1025: (8)              return outvals
1026: (0)          def _diff_dispatcher(a, n=None, axis=None, prepend=None, append=None):
1027: (4)              return (a, prepend, append)
1028: (0)          @array_function_dispatch(_diff_dispatcher)
1029: (0)          def diff(a, n=1, axis=-1, prepend=np._NoValue, append=np._NoValue):
1030: (4)              """
1031: (4)              Calculate the n-th discrete difference along the given axis.

```

```

1032: (4)          The first difference is given by ``out[i] = a[i+1] - a[i]`` along
1033: (4)          the given axis, higher differences are calculated by using `diff`
1034: (4)          recursively.
1035: (4)          Parameters
1036: (4)          -----
1037: (4)          a : array_like
1038: (8)            Input array
1039: (4)          n : int, optional
1040: (8)            The number of times values are differenced. If zero, the input
1041: (8)            is returned as-is.
1042: (4)          axis : int, optional
1043: (8)            The axis along which the difference is taken, default is the
1044: (8)            last axis.
1045: (4)          prepend, append : array_like, optional
1046: (8)            Values to prepend or append to `a` along axis prior to
1047: (8)            performing the difference. Scalar values are expanded to
1048: (8)            arrays with length 1 in the direction of axis and the shape
1049: (8)            of the input array in along all other axes. Otherwise the
1050: (8)            dimension and shape must match `a` except along axis.
1051: (8)            .. versionadded:: 1.16.0
1052: (4)          Returns
1053: (4)          -----
1054: (4)          diff : ndarray
1055: (8)            The n-th differences. The shape of the output is the same as `a`
1056: (8)            except along `axis` where the dimension is smaller by `n`. The
1057: (8)            type of the output is the same as the type of the difference
1058: (8)            between any two elements of `a`. This is the same as the type of
1059: (8)            `a` in most cases. A notable exception is `datetime64`, which
1060: (8)            results in a `timedelta64` output array.
1061: (4)          See Also
1062: (4)          -----
1063: (4)          gradient, ediff1d, cumsum
1064: (4)          Notes
1065: (4)          -----
1066: (4)            Type is preserved for boolean arrays, so the result will contain
1067: (4)            `False` when consecutive elements are the same and `True` when they
1068: (4)            differ.
1069: (4)            For unsigned integer arrays, the results will also be unsigned. This
1070: (4)            should not be surprising, as the result is consistent with
1071: (4)            calculating the difference directly:
1072: (4)            >>> u8_arr = np.array([1, 0], dtype=np.uint8)
1073: (4)            >>> np.diff(u8_arr)
1074: (4)            array([255], dtype=uint8)
1075: (4)            >>> u8_arr[1,...] - u8_arr[0,...]
1076: (4)            255
1077: (4)            If this is not desirable, then the array should be cast to a larger
1078: (4)            integer type first:
1079: (4)            >>> i16_arr = u8_arr.astype(np.int16)
1080: (4)            >>> np.diff(i16_arr)
1081: (4)            array([-1], dtype=int16)
1082: (4)          Examples
1083: (4)          -----
1084: (4)            >>> x = np.array([1, 2, 4, 7, 0])
1085: (4)            >>> np.diff(x)
1086: (4)            array([ 1,  2,  3, -7])
1087: (4)            >>> np.diff(x, n=2)
1088: (4)            array([ 1,  1, -10])
1089: (4)            >>> x = np.array([[1, 3, 6, 10], [0, 5, 6, 8]])
1090: (4)            >>> np.diff(x)
1091: (4)            array([[2, 3, 4],
1092: (11)              [5, 1, 2]])
1093: (4)            >>> np.diff(x, axis=0)
1094: (4)            array([[-1,  2,  0, -2]])
1095: (4)            >>> x = np.arange('1066-10-13', '1066-10-16', dtype=np.datetime64)
1096: (4)            >>> np.diff(x)
1097: (4)            array([1, 1], dtype='timedelta64[D]')
1098: (4)            """
1099: (4)            if n == 0:
1100: (8)              return a

```

```

1101: (4)             if n < 0:
1102: (8)               raise ValueError(
1103: (12)                 "order must be non-negative but got " + repr(n))
1104: (4)               a = asanyarray(a)
1105: (4)               nd = a.ndim
1106: (4)               if nd == 0:
1107: (8)                 raise ValueError("diff requires input that is at least one
dimensional")
1108: (4)               axis = normalize_axis_index(axis, nd)
1109: (4)               combined = []
1110: (4)               if prepend is not np._NoValue:
1111: (8)                 prepend = np.asanyarray(prepend)
1112: (8)                 if prepend.ndim == 0:
1113: (12)                   shape = list(a.shape)
1114: (12)                   shape[axis] = 1
1115: (12)                   prepend = np.broadcast_to(prepend, tuple(shape))
1116: (8)                   combined.append(prepend)
1117: (4)               combined.append(a)
1118: (4)               if append is not np._NoValue:
1119: (8)                 append = np.asanyarray(append)
1120: (8)                 if append.ndim == 0:
1121: (12)                   shape = list(a.shape)
1122: (12)                   shape[axis] = 1
1123: (12)                   append = np.broadcast_to(append, tuple(shape))
1124: (8)                   combined.append(append)
1125: (4)               if len(combined) > 1:
1126: (8)                 a = np.concatenate(combined, axis)
1127: (4)                 slice1 = [slice(None)] * nd
1128: (4)                 slice2 = [slice(None)] * nd
1129: (4)                 slice1[axis] = slice(1, None)
1130: (4)                 slice2[axis] = slice(None, -1)
1131: (4)                 slice1 = tuple(slice1)
1132: (4)                 slice2 = tuple(slice2)
1133: (4)                 op = not_equal if a.dtype == np.bool_ else subtract
1134: (4)                 for _ in range(n):
1135: (8)                   a = op(a[slice1], a[slice2])
1136: (4)               return a
1137: (0)             def _interp_dispatcher(x, xp, fp, left=None, right=None, period=None):
1138: (4)               return (x, xp, fp)
1139: (0)             @array_function_dispatch(_interp_dispatcher)
1140: (0)             def interp(x, xp, fp, left=None, right=None, period=None):
1141: (4)               """
1142: (4)               One-dimensional linear interpolation for monotonically increasing sample
points.
1143: (4)               Returns the one-dimensional piecewise linear interpolant to a function
1144: (4)               with given discrete data points (`xp`, `fp`), evaluated at `x`.
1145: (4)               Parameters
1146: (4)               -----
1147: (4)               x : array_like
1148: (8)                 The x-coordinates at which to evaluate the interpolated values.
1149: (4)               xp : 1-D sequence of floats
1150: (8)                 The x-coordinates of the data points, must be increasing if argument
1151: (8)                 `period` is not specified. Otherwise, `xp` is internally sorted after
1152: (8)                 normalizing the periodic boundaries with ``xp = xp % period``.
1153: (4)               fp : 1-D sequence of float or complex
1154: (8)                 The y-coordinates of the data points, same length as `xp`.
1155: (4)               left : optional float or complex corresponding to fp
1156: (8)                 Value to return for `x < xp[0]`, default is `fp[0]`.
1157: (4)               right : optional float or complex corresponding to fp
1158: (8)                 Value to return for `x > xp[-1]`, default is `fp[-1]`.
1159: (4)               period : None or float, optional
1160: (8)                 A period for the x-coordinates. This parameter allows the proper
1161: (8)                 interpolation of angular x-coordinates. Parameters `left` and `right`
1162: (8)                 are ignored if `period` is specified.
1163: (8)                 .. versionadded:: 1.10.0
1164: (4)               Returns
1165: (4)               -----
1166: (4)               y : float or complex (corresponding to fp) or ndarray
1167: (8)                 The interpolated values, same shape as `x`.

```

```

1168: (4)             Raises
1169: (4)             -----
1170: (4)             ValueError
1171: (8)               If `xp` and `fp` have different length
1172: (8)               If `xp` or `fp` are not 1-D sequences
1173: (8)               If `period == 0`
1174: (4)             See Also
1175: (4)             -----
1176: (4)               scipy.interpolate
1177: (4)             Warnings
1178: (4)             -----
1179: (4)               The x-coordinate sequence is expected to be increasing, but this is not
1180: (4)               explicitly enforced. However, if the sequence `xp` is non-increasing,
1181: (4)               interpolation results are meaningless.
1182: (4)               Note that, since NaN is unsortable, `xp` also cannot contain NaNs.
1183: (4)               A simple check for `xp` being strictly increasing is::
1184: (8)                 np.all(np.diff(xp) > 0)
1185: (4)             Examples
1186: (4)             -----
1187: (4)               >>> xp = [1, 2, 3]
1188: (4)               >>> fp = [3, 2, 0]
1189: (4)               >>> np.interp(2.5, xp, fp)
1190: (4)               1.0
1191: (4)               >>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
1192: (4)               array([3. , 3. , 2.5 , 0.56, 0. ])
1193: (4)               >>> UNDEF = -99.0
1194: (4)               >>> np.interp(3.14, xp, fp, right=UNDEF)
1195: (4)               -99.0
1196: (4)             Plot an interpolant to the sine function:
1197: (4)               >>> x = np.linspace(0, 2*np.pi, 10)
1198: (4)               >>> y = np.sin(x)
1199: (4)               >>> xvals = np.linspace(0, 2*np.pi, 50)
1200: (4)               >>> yinterp = np.interp(xvals, x, y)
1201: (4)               >>> import matplotlib.pyplot as plt
1202: (4)               >>> plt.plot(x, y, 'o')
1203: (4)               [<matplotlib.lines.Line2D object at 0x...>]
1204: (4)               >>> plt.plot(xvals, yinterp, '-x')
1205: (4)               [<matplotlib.lines.Line2D object at 0x...>]
1206: (4)               >>> plt.show()
1207: (4)             Interpolation with periodic x-coordinates:
1208: (4)               >>> x = [-180, -170, -185, 185, -10, -5, 0, 365]
1209: (4)               >>> xp = [190, -190, 350, -350]
1210: (4)               >>> fp = [5, 10, 3, 4]
1211: (4)               >>> np.interp(x, xp, fp, period=360)
1212: (4)               array([7.5 , 5. , 8.75, 6.25, 3. , 3.25, 3.5 , 3.75])
1213: (4)             Complex interpolation:
1214: (4)               >>> x = [1.5, 4.0]
1215: (4)               >>> xp = [2, 3, 5]
1216: (4)               >>> fp = [1.0j, 0, 2+3j]
1217: (4)               >>> np.interp(x, xp, fp)
1218: (4)               array([0.+1.j, 1.+1.5j])
1219: (4)               """
1220: (4)               fp = np.asarray(fp)
1221: (4)               if np.iscomplexobj(fp):
1222: (8)                 interp_func = compiled_interp_complex
1223: (8)                 input_dtype = np.complex128
1224: (4)               else:
1225: (8)                 interp_func = compiled_interp
1226: (8)                 input_dtype = np.float64
1227: (4)               if period is not None:
1228: (8)                 if period == 0:
1229: (12)                   raise ValueError("period must be a non-zero value")
1230: (8)                 period = abs(period)
1231: (8)                 left = None
1232: (8)                 right = None
1233: (8)                 x = np.asarray(x, dtype=np.float64)
1234: (8)                 xp = np.asarray(xp, dtype=np.float64)
1235: (8)                 fp = np.asarray(fp, dtype=input_dtype)
1236: (8)                 if xp.ndim != 1 or fp.ndim != 1:

```

```

1237: (12)             raise ValueError("Data points must be 1-D sequences")
1238: (8)              if xp.shape[0] != fp.shape[0]:
1239: (12)                  raise ValueError("fp and xp are not of the same length")
1240: (8)              x = x % period
1241: (8)              xp = xp % period
1242: (8)              asort_xp = np.argsort(xp)
1243: (8)              xp = xp[asort_xp]
1244: (8)              fp = fp[asort_xp]
1245: (8)              xp = np.concatenate((xp[-1:]-period, xp, xp[0:1]+period))
1246: (8)              fp = np.concatenate((fp[-1:], fp, fp[0:1]))
1247: (4)              return interp_func(x, xp, fp, left, right)
1248: (0) def _angle_dispatcher(z, deg=None):
1249: (4)     return (z,)
1250: (0) @array_function_dispatch(_angle_dispatcher)
1251: (0) def angle(z, deg=False):
1252: (4)     """
1253: (4)     Return the angle of the complex argument.
1254: (4)     Parameters
1255: (4)     -----
1256: (4)     z : array_like
1257: (8)         A complex number or sequence of complex numbers.
1258: (4)     deg : bool, optional
1259: (8)         Return angle in degrees if True, radians if False (default).
1260: (4)     Returns
1261: (4)     -----
1262: (4)     angle : ndarray or scalar
1263: (8)         The counterclockwise angle from the positive real axis on the complex
1264: (8)         plane in the range ``(-pi, pi)``, with dtype as numpy.float64.
1265: (8)         .. versionchanged:: 1.16.0
1266: (12)             This function works on subclasses of ndarray like `ma.array`.
1267: (4)     See Also
1268: (4)     -----
1269: (4)     arctan2
1270: (4)     absolute
1271: (4)     Notes
1272: (4)     -----
1273: (4)     Although the angle of the complex number 0 is undefined,
``numpy.angle(0)``
1274: (4)     returns the value 0.
1275: (4)     Examples
1276: (4)     -----
1277: (4)     >>> np.angle([1.0, 1.0j, 1+1j])           # in radians
1278: (4)     array([ 0. ,  1.57079633,  0.78539816]) # may vary
1279: (4)     >>> np.angle(1+1j, deg=True)          # in degrees
1280: (4)     45.0
1281: (4)     """
1282: (4)     z = asanyarray(z)
1283: (4)     if issubclass(z.dtype.type, _nx.complexfloating):
1284: (8)         zimag = z.imag
1285: (8)         zreal = z.real
1286: (4)     else:
1287: (8)         zimag = 0
1288: (8)         zreal = z
1289: (4)     a = arctan2(zimag, zreal)
1290: (4)     if deg:
1291: (8)         a *= 180/pi
1292: (4)     return a
1293: (0) def _unwrap_dispatcher(p, discont=None, axis=None, *, period=None):
1294: (4)     return (p,)
1295: (0) @array_function_dispatch(_unwrap_dispatcher)
1296: (0) def unwrap(p, discont=None, axis=-1, *, period=2*pi):
1297: (4)     """
1298: (4)     Unwrap by taking the complement of large deltas with respect to the
1299: (4)     period.
1300: (4)     This unwraps a signal `p` by changing elements which have an absolute
1301: (4)     difference from their predecessor of more than ``max(discont, period/2)``
1302: (4)     to their `period`-complementary values.
1303: (4)     For the default case where `period` is :math:`2\pi` and `discont` is
:math:`\pi`, this unwraps a radian phase `p` such that adjacent

```

```

differences
1304: (4)     are never greater than :math:`\pi` by adding :math:`2k\pi` for some
1305: (4)     integer :math:`k`.
1306: (4)     Parameters
1307: (4)     -----
1308: (4)     p : array_like
1309: (8)       Input array.
1310: (4)     discontinuity : float, optional
1311: (8)       Maximum discontinuity between values, default is ``period/2``.
1312: (8)       Values below ``period/2`` are treated as if they were ``period/2``.
1313: (8)       To have an effect different from the default, `discont` should be
1314: (8)       larger than ``period/2``.
1315: (4)     axis : int, optional
1316: (8)       Axis along which unwrap will operate, default is the last axis.
1317: (4)     period : float, optional
1318: (8)       Size of the range over which the input wraps. By default, it is
1319: (8)       ``2 pi``.
1320: (8)       .. versionadded:: 1.21.0
1321: (4)     Returns
1322: (4)     -----
1323: (4)     out : ndarray
1324: (8)       Output array.
1325: (4)     See Also
1326: (4)     -----
1327: (4)     rad2deg, deg2rad
1328: (4)     Notes
1329: (4)     -----
1330: (4)     If the discontinuity in `p` is smaller than ``period/2``,
1331: (4)     but larger than `discont`, no unwrapping is done because taking
1332: (4)     the complement would only make the discontinuity larger.
1333: (4)     Examples
1334: (4)     -----
1335: (4)     >>> phase = np.linspace(0, np.pi, num=5)
1336: (4)     >>> phase[3:] += np.pi
1337: (4)     >>> phase
1338: (4)     array([ 0.          ,  0.78539816,  1.57079633,  5.49778714,  6.28318531]) #
may vary
1339: (4)     >>> np.unwrap(phase)
1340: (4)     array([ 0.          ,  0.78539816,  1.57079633, -0.78539816,  0.          ]) #
may vary
1341: (4)     >>> np.unwrap([0, 1, 2, -1, 0], period=4)
1342: (4)     array([0, 1, 2, 3, 4])
1343: (4)     >>> np.unwrap([ 1, 2, 3, 4, 5, 6, 1, 2, 3], period=6)
1344: (4)     array([1, 2, 3, 4, 5, 6, 7, 8, 9])
1345: (4)     >>> np.unwrap([2, 3, 4, 5, 2, 3, 4, 5], period=4)
1346: (4)     array([2, 3, 4, 5, 6, 7, 8, 9])
1347: (4)     >>> phase_deg = np.mod(np.linspace(0, 720, 19), 360) - 180
1348: (4)     >>> np.unwrap(phase_deg, period=360)
1349: (4)     array([-180., -140., -100., -60., -20.,  20.,  60.,  100.,  140.,
1350: (12)      180.,  220.,  260.,  300.,  340.,  380.,  420.,  460.,  500.,
1351: (12)      540.])
1352: (4)     """
1353: (4)     p = asarray(p)
1354: (4)     nd = p.ndim
1355: (4)     dd = diff(p, axis=axis)
1356: (4)     if discont is None:
1357: (8)         discont = period/2
1358: (4)     slice1 = [slice(None, None)]*nd      # full slices
1359: (4)     slice1[axis] = slice(1, None)
1360: (4)     slice1 = tuple(slice1)
1361: (4)     dtype = np.result_type(dd, period)
1362: (4)     if _nx.issubdtype(dtype, _nx.integer):
1363: (8)         interval_high, rem = divmod(period, 2)
1364: (8)         boundary_ambiguous = rem == 0
1365: (4)     else:
1366: (8)         interval_high = period / 2
1367: (8)         boundary_ambiguous = True
1368: (4)     interval_low = -interval_high
1369: (4)     ddmod = mod(dd - interval_low, period) + interval_low

```

```

1370: (4)         if boundary_ambiguous:
1371: (8)             _nx.copyto(ddmod, interval_high,
1372: (19)                 where=(ddmod == interval_low) & (dd > 0))
1373: (4)             ph_correct = ddmod - dd
1374: (4)             _nx.copyto(ph_correct, 0, where=abs(dd) < discontinuity)
1375: (4)             up = array(p, copy=True, dtype=dtype)
1376: (4)             up[slice1] = p[slice1] + ph_correct.cumsum(axis)
1377: (4)             return up
1378: (0)         def _sort_complex(a):
1379: (4)             return (a,)
1380: (0)         @array_function_dispatch(_sort_complex)
1381: (0)         def sort_complex(a):
1382: (4)             """
1383: (4)             Sort a complex array using the real part first, then the imaginary part.
1384: (4)             Parameters
1385: (4)             -----
1386: (4)             a : array_like
1387: (8)                 Input array
1388: (4)             Returns
1389: (4)             -----
1390: (4)             out : complex ndarray
1391: (8)                 Always returns a sorted complex array.
1392: (4)             Examples
1393: (4)             -----
1394: (4)             >>> np.sort_complex([5, 3, 6, 2, 1])
1395: (4)             array([1.+0.j, 2.+0.j, 3.+0.j, 5.+0.j, 6.+0.j])
1396: (4)             >>> np.sort_complex([1 + 2j, 2 - 1j, 3 - 2j, 3 - 3j, 3 + 5j])
1397: (4)             array([1.+2.j, 2.-1.j, 3.-3.j, 3.-2.j, 3.+5.j])
1398: (4)             """
1399: (4)             b = array(a, copy=True)
1400: (4)             b.sort()
1401: (4)             if not issubclass(b.dtype.type, _nx.complexfloating):
1402: (8)                 if b.dtype.char in 'bhBH':
1403: (12)                     return b.astype('F')
1404: (8)                 elif b.dtype.char == 'g':
1405: (12)                     return b.astype('G')
1406: (8)                 else:
1407: (12)                     return b.astype('D')
1408: (4)             else:
1409: (8)                 return b
1410: (0)         def _trim_zerosfilt, trim=None):
1411: (4)             return (filt,)
1412: (0)         @array_function_dispatch(_trim_zeros)
1413: (0)         def trim_zerosfilt, trim='fb'):
1414: (4)             """
1415: (4)             Trim the leading and/or trailing zeros from a 1-D array or sequence.
1416: (4)             Parameters
1417: (4)             -----
1418: (4)             filt : 1-D array or sequence
1419: (8)                 Input array.
1420: (4)             trim : str, optional
1421: (8)                 A string with 'f' representing trim from front and 'b' to trim from
1422: (8)                 back. Default is 'fb', trim zeros from both front and back of the
1423: (8)                 array.
1424: (4)             Returns
1425: (4)             -----
1426: (4)             trimmed : 1-D array or sequence
1427: (8)                 The result of trimming the input. The input data type is preserved.
1428: (4)             Examples
1429: (4)             -----
1430: (4)             >>> a = np.array((0, 0, 0, 1, 2, 3, 0, 2, 1, 0))
1431: (4)             >>> np.trim_zeros(a)
1432: (4)             array([1, 2, 3, 0, 2, 1])
1433: (4)             >>> np.trim_zeros(a, 'b')
1434: (4)             array([0, 0, 0, ..., 0, 2, 1])
1435: (4)             The input data type is preserved, list/tuple in means list/tuple out.
1436: (4)             >>> np.trim_zeros([0, 1, 2, 0])
1437: (4)             [1, 2]
1438: (4)             """

```

```

1439: (4)         first = 0
1440: (4)         trim = trim.upper()
1441: (4)         if 'F' in trim:
1442: (8)             for i in filt:
1443: (12)                 if i != 0.:
1444: (16)                     break
1445: (12)                 else:
1446: (16)                     first = first + 1
1447: (4)         last = len(filt)
1448: (4)         if 'B' in trim:
1449: (8)             for i in filt[::-1]:
1450: (12)                 if i != 0.:
1451: (16)                     break
1452: (12)                 else:
1453: (16)                     last = last - 1
1454: (4)         return filt[first:last]
1455: (0)         def _extract_dispatcher(condition, arr):
1456: (4)             return (condition, arr)
1457: (0)         @array_function_dispatch(_extract_dispatcher)
1458: (0)         def extract(condition, arr):
1459: (4)             """
1460: (4)             Return the elements of an array that satisfy some condition.
1461: (4)             This is equivalent to ``np.compress(ravel(condition), ravel(arr))``. If
1462: (4)             `condition` is boolean ``np.extract`` is equivalent to ``arr[condition]``.
1463: (4)             Note that `place` does the exact opposite of `extract`.
1464: (4)             Parameters
1465: (4)             -----
1466: (4)             condition : array_like
1467: (8)                 An array whose nonzero or True entries indicate the elements of `arr`
1468: (8)                 to extract.
1469: (4)             arr : array_like
1470: (8)                 Input array of the same size as `condition`.
1471: (4)             Returns
1472: (4)             -----
1473: (4)             extract : ndarray
1474: (8)                 Rank 1 array of values from `arr` where `condition` is True.
1475: (4)             See Also
1476: (4)             -----
1477: (4)             take, put, copyto, compress, place
1478: (4)             Examples
1479: (4)             -----
1480: (4)             >>> arr = np.arange(12).reshape((3, 4))
1481: (4)             >>> arr
1482: (4)             array([[ 0,  1,  2,  3],
1483: (11)                 [ 4,  5,  6,  7],
1484: (11)                 [ 8,  9, 10, 11]])
1485: (4)             >>> condition = np.mod(arr, 3)==0
1486: (4)             >>> condition
1487: (4)             array([[ True, False, False,  True],
1488: (11)                 [False, False,  True, False],
1489: (11)                 [False,  True, False, False]])
1490: (4)             >>> np.extract(condition, arr)
1491: (4)             array([0, 3, 6, 9])
1492: (4)             If `condition` is boolean:
1493: (4)             >>> arr[condition]
1494: (4)             array([0, 3, 6, 9])
1495: (4)             """
1496: (4)             return _nx.take(ravel(arr), nonzero(ravel(condition))[0])
1497: (0)         def _place_dispatcher(arr, mask, vals):
1498: (4)             return (arr, mask, vals)
1499: (0)         @array_function_dispatch(_place_dispatcher)
1500: (0)         def place(arr, mask, vals):
1501: (4)             """
1502: (4)             Change elements of an array based on conditional and input values.
1503: (4)             Similar to ``np.copyto(arr, vals, where=mask)``, the difference is that
1504: (4)             `place` uses the first N elements of `vals`, where N is the number of
1505: (4)             True values in `mask`, while `copyto` uses the elements where `mask`
1506: (4)             is True.
1507: (4)             Note that `extract` does the exact opposite of `place`.

```

```

1508: (4)             Parameters
1509: (4)             -----
1510: (4)             arr : ndarray
1511: (8)             Array to put data into.
1512: (4)             mask : array_like
1513: (8)             Boolean mask array. Must have the same size as `a`.
1514: (4)             vals : 1-D sequence
1515: (8)             Values to put into `a`. Only the first N elements are used, where
1516: (8)             N is the number of True values in `mask`. If `vals` is smaller
1517: (8)             than N, it will be repeated, and if elements of `a` are to be masked,
1518: (8)             this sequence must be non-empty.
1519: (4)             See Also
1520: (4)             -----
1521: (4)             copyto, put, take, extract
1522: (4)             Examples
1523: (4)             -----
1524: (4)             >>> arr = np.arange(6).reshape(2, 3)
1525: (4)             >>> np.place(arr, arr>2, [44, 55])
1526: (4)             >>> arr
1527: (4)             array([[ 0,  1,  2],
1528: (11)                [44, 55, 44]])
1529: (4)             """
1530: (4)             return _place(arr, mask, vals)
1531: (0)             def disp(mesg, device=None, linefeed=True):
1532: (4)             """
1533: (4)             Display a message on a device.
1534: (4)             Parameters
1535: (4)             -----
1536: (4)             mesg : str
1537: (8)             Message to display.
1538: (4)             device : object
1539: (8)             Device to write message. If None, defaults to ``sys.stdout`` which is
1540: (8)             very similar to ``print``. `device` needs to have ``write()`` and
1541: (8)             ``flush()`` methods.
1542: (4)             linefeed : bool, optional
1543: (8)             Option whether to print a line feed or not. Defaults to True.
1544: (4)             Raises
1545: (4)             -----
1546: (4)             AttributeError
1547: (8)             If `device` does not have a ``write()`` or ``flush()`` method.
1548: (4)             Examples
1549: (4)             -----
1550: (4)             Besides ``sys.stdout``, a file-like object can also be used as it has
1551: (4)             both required methods:
1552: (4)             >>> from io import StringIO
1553: (4)             >>> buf = StringIO()
1554: (4)             >>> np.disp(u'"Display" in a file', device=buf)
1555: (4)             >>> buf.getvalue()
1556: (4)             '"Display" in a file\n'
1557: (4)             """
1558: (4)             if device is None:
1559: (8)                 device = sys.stdout
1560: (4)             if linefeed:
1561: (8)                 device.write('%s\n' % mesg)
1562: (4)             else:
1563: (8)                 device.write('%s' % mesg)
1564: (4)             device.flush()
1565: (4)             return
1566: (0)             _DIMENSION_NAME = r'\w+'
1567: (0)             _CORE_DIMENSION_LIST = '(?:{0:}({:},{0:})*?)'.format(_DIMENSION_NAME)
1568: (0)             _ARGUMENT = r'\({}\)'.format(_CORE_DIMENSION_LIST)
1569: (0)             _ARGUMENT_LIST = '{0:}({:},{0:})*'.format(_ARGUMENT)
1570: (0)             _SIGNATURE = '^{0:}->{0:$}'.format(_ARGUMENT_LIST)
1571: (0)             def _parse_gufunc_signature(signature):
1572: (4)             """
1573: (4)             Parse string signatures for a generalized universal function.
1574: (4)             Arguments
1575: (4)             -----
1576: (4)             signature : string

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1577: (8)           Generalized universal function signature, e.g., ``(m,n),(n,p)->(m,p)``
1578: (8)           for ``np.matmul``.
1579: (4)           Returns
1580: (4)
1581: (4)           Tuple of input and output core dimensions parsed from the signature, each
1582: (4)           of the form List[Tuple[str, ...]].
1583: (4)
1584: (4)           signature = re.sub(r'\s+', '', signature)
1585: (4)           if not re.match(_SIGNATURE, signature):
1586: (8)               raise ValueError(
1587: (12)                   'not a valid gufunc signature: {}'.format(signature))
1588: (4)           return tuple([tuple(re.findall(_DIMENSION_NAME, arg))
1589: (18)                           for arg in re.findall(_ARGUMENT, arg_list)])
1590: (17)                           for arg_list in signature.split('->')])
1591: (0)           def _update_dim_sizes(dim_sizes, arg, core_dims):
1592: (4)
1593: (4)           """Incrementally check and update core dimension sizes for a single argument.
1594: (4)           Arguments
1595: (4)
1596: (4)           dim_sizes : Dict[str, int]
1597: (8)               Sizes of existing core dimensions. Will be updated in-place.
1598: (4)           arg : ndarray
1599: (8)               Argument to examine.
1600: (4)           core_dims : Tuple[str, ...]
1601: (8)               Core dimensions for this argument.
1602: (4)
1603: (4)
1604: (8)           if not core_dims:
1605: (4)               return
1606: (4)           num_core_dims = len(core_dims)
1607: (8)           if arg.ndim < num_core_dims:
1608: (12)               raise ValueError(
1609: (12)                   '%d-dimensional argument does not have enough '
1610: (12)                   'dimensions for all core dimensions %r'
1611: (4)                   % (arg.ndim, core_dims))
1612: (4)           core_shape = arg.shape[-num_core_dims:]
1613: (8)           for dim, size in zip(core_dims, core_shape):
1614: (12)               if dim in dim_sizes:
1615: (16)                   if size != dim_sizes[dim]:
1616: (20)                       raise ValueError(
1617: (20)                           'inconsistent size for core dimension %r: %r vs %r'
1618: (8)                           % (dim, size, dim_sizes[dim]))
1619: (12)                   else:
1620: (0)                       dim_sizes[dim] = size
def _parse_input_dimensions(args, input_core_dims):
1621: (4)
1622: (4)           """Parse broadcast and core dimensions for vectorize with a signature.
1623: (4)           Arguments
1624: (4)
1625: (4)           args : Tuple[ndarray, ...]
1626: (8)               Tuple of input arguments to examine.
1627: (4)           input_core_dims : List[Tuple[str, ...]]
1628: (8)               List of core dimensions corresponding to each input.
1629: (4)           Returns
1630: (4)
1631: (4)           broadcast_shape : Tuple[int, ...]
1632: (8)               Common shape to broadcast all non-core dimensions to.
1633: (4)           dim_sizes : Dict[str, int]
1634: (8)               Common sizes for named core dimensions.
1635: (4)
1636: (4)           broadcast_args = []
1637: (4)           dim_sizes = {}
1638: (4)           for arg, core_dims in zip(args, input_core_dims):
1639: (8)               _update_dim_sizes(dim_sizes, arg, core_dims)
1640: (8)               ndim = arg.ndim - len(core_dims)
1641: (8)               dummy_array = np.lib.stride_tricks.as_strided(0, arg.shape[:ndim])
1642: (8)               broadcast_args.append(dummy_array)
1643: (4)               broadcast_shape = np.lib.stride_tricks._broadcast_shape(*broadcast_args)
1644: (4)               return broadcast_shape, dim_sizes
1645: (0)           def _calculate_shapes(broadcast_shape, dim_sizes, list_of_core_dims):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1646: (4)     """Helper for calculating broadcast shapes with core dimensions."""
1647: (4)     return [broadcast_shape + tuple(dim_sizes[dim] for dim in core_dims)
1648: (12)         for core_dims in list_of_core_dims]
1649: (0) def _create_arrays(broadcast_shape, dim_sizes, list_of_core_dims, dtypes,
1650: (19)             results=None):
1651: (4)     """Helper for creating output arrays in vectorize."""
1652: (4)     shapes = _calculate_shapes(broadcast_shape, dim_sizes, list_of_core_dims)
1653: (4)     if dtypes is None:
1654: (8)         dtypes = [None] * len(shapes)
1655: (4)     if results is None:
1656: (8)         arrays = tuple(np.empty(shape=shape, dtype=dtype)
1657: (23)             for shape, dtype in zip(shapes, dtypes))
1658: (4)     else:
1659: (8)         arrays = tuple(np.empty_like(result, shape=shape, dtype=dtype)
1660: (23)             for result, shape, dtype
1661: (23)                 in zip(results, shapes, dtypes))
1662: (4)     return arrays
1663: (0) @set_module('numpy')
1664: (0) class vectorize:
1665: (4)     """
1666: (4)     vectorize(pyfunc=np._NoValue, otypes=None, doc=None, excluded=None,
1667: (4)         cache=False, signature=None)
1668: (4)     Returns an object that acts like pyfunc, but takes arrays as input.
1669: (4)     Define a vectorized function which takes a nested sequence of objects or
1670: (4)     numpy arrays as inputs and returns a single numpy array or a tuple of
1671: (4)     arrays. The vectorized function evaluates `pyfunc` over successive tuples
1672: (4)     of the input arrays like the python map function, except it uses the
1673: (4)     broadcasting rules of numpy.
1674: (4)     The data type of the output of `vectorized` is determined by calling
1675: (4)     the function with the first element of the input. This can be avoided
1676: (4)     by specifying the `otypes` argument.
1677: (4)     Parameters
1678: (4)     -----
1679: (4)     pyfunc : callable, optional
1680: (8)         A python function or method.
1681: (8)         Can be omitted to produce a decorator with keyword arguments.
1682: (4)     otypes : str or list of dtypes, optional
1683: (8)         The output data type. It must be specified as either a string of
1684: (8)         typecode characters or a list of data type specifiers. There should
1685: (8)         be one data type specifier for each output.
1686: (4)     doc : str, optional
1687: (8)         The docstring for the function. If None, the docstring will be the
1688: (8)         ``pyfunc.__doc__``.
1689: (4)     excluded : set, optional
1690: (8)         Set of strings or integers representing the positional or keyword
1691: (8)         arguments for which the function will not be vectorized. These will
be
1692: (8)         passed directly to `pyfunc` unmodified.
1693: (8)         .. versionadded:: 1.7.0
1694: (4)     cache : bool, optional
1695: (8)         If `True`, then cache the first function call that determines the
number
1696: (8)         of outputs if `otypes` is not provided.
1697: (8)         .. versionadded:: 1.7.0
1698: (4)     signature : string, optional
1699: (8)         Generalized universal function signature, e.g., ``(m,n),(n)->(m)`` for
1700: (8)         vectorized matrix-vector multiplication. If provided, ``pyfunc`` will
1701: (8)         be called with (and expected to return) arrays with shapes given by
the
1702: (8)         size of corresponding core dimensions. By default, ``pyfunc`` is
1703: (8)         assumed to take scalars as input and output.
1704: (8)         .. versionadded:: 1.12.0
1705: (4)     Returns
1706: (4)     -----
1707: (4)     out : callable
1708: (8)         A vectorized function if ``pyfunc`` was provided,
1709: (8)         a decorator otherwise.
1710: (4)     See Also

```

```

1711: (4) -----
1712: (4) from numpy import frompyfunc : Takes an arbitrary Python function and returns a ufunc
1713: (4) Notes -----
1714: (4) The `vectorize` function is provided primarily for convenience, not for
1715: (4) performance. The implementation is essentially a for loop.
1716: (4) If `otypes` is not specified, then a call to the function with the
1717: (4) first argument will be used to determine the number of outputs. The
1718: (4) results of this call will be cached if `cache` is `True` to prevent
1719: (4) calling the function twice. However, to implement the cache, the
1720: (4) original function must be wrapped which will slow down subsequent
1721: (4) calls, so only do this if your function is expensive.
1722: (4) The new keyword argument interface and `excluded` argument support
1723: (4) further degrades performance.
1724: (4) References -----
1725: (4) .. [1] :doc:`/reference/c-api/generalized-ufuncs`
1726: (4) Examples -----
1727: (4) >>> def myfunc(a, b):
1728: (4) ...     "Return a-b if a>b, otherwise return a+b"
1729: (4) ...     if a > b:
1730: (4) ...         return a - b
1731: (4) ...     else:
1732: (4) ...         return a + b
1733: (4) >>> vfunc = np.vectorize(myfunc)
1734: (4) >>> vfunc([1, 2, 3, 4], 2)
1735: (4) array([3, 4, 1, 2])
1736: (4) The docstring is taken from the input function to `vectorize` unless it
1737: (4) is specified:
1738: (4) >>> vfunc.__doc__
1739: (4) 'Return a-b if a>b, otherwise return a+b'
1740: (4) >>> vfunc = np.vectorize(myfunc, doc='Vectorized `myfunc`')
1741: (4) >>> vfunc.__doc__
1742: (4) 'Vectorized `myfunc`'
1743: (4) The output type is determined by evaluating the first element of the
1744: (4) input,
1745: (4) unless it is specified:
1746: (4) >>> out = vfunc([1, 2, 3, 4], 2)
1747: (4) >>> type(out[0])
1748: (4) <class 'numpy.int64'>
1749: (4) >>> vfunc = np.vectorize(myfunc, otypes=[float])
1750: (4) >>> out = vfunc([1, 2, 3, 4], 2)
1751: (4) >>> type(out[0])
1752: (4) <class 'numpy.float64'>
1753: (4) The `excluded` argument can be used to prevent vectorizing over certain
1754: (4) arguments. This can be useful for array-like arguments of a fixed length
1755: (4) such as the coefficients for a polynomial as in `polyval`:
1756: (4) >>> def mypolyval(p, x):
1757: (4) ...     _p = list(p)
1758: (4) ...     res = _p.pop(0)
1759: (4) ...     while _p:
1760: (4) ...         res = res*x + _p.pop(0)
1761: (4) ...     return res
1762: (4) >>> vpolyval = np.vectorize(mypolyval, excluded=['p'])
1763: (4) >>> vpolyval(p=[1, 2, 3], x=[0, 1])
1764: (4) array([3, 6])
1765: (4) Positional arguments may also be excluded by specifying their position:
1766: (4) >>> vpolyval.excluded.add(0)
1767: (4) >>> vpolyval([1, 2, 3], x=[0, 1])
1768: (4) array([3, 6])
1769: (4) The `signature` argument allows for vectorizing functions that act on
1770: (4) non-scalar arrays of fixed length. For example, you can use it for a
1771: (4) vectorized calculation of Pearson correlation coefficient and its p-value:
1772: (4) >>> import scipy.stats
1773: (4) >>> pearsonr = np.vectorize(scipy.stats.pearsonr,
1774: (4) ...             signature='(n),(n)->(),()')
1775: (4) >>> pearsonr([[0, 1, 2, 3]], [[1, 2, 3, 4], [4, 3, 2, 1]])
1776: (4) (array([ 1., -1.]), array([ 0.,  0.]))
1777: (4)
1778: (4)

```

```

1779: (4)          Or for a vectorized convolution:
1780: (4)          >>> convolve = np.vectorize(np.convolve, signature='(n),(m)->(k)')
1781: (4)          >>> convolve(np.eye(4), [1, 2, 1])
1782: (4)          array([[1.,  2.,  1.,  0.,  0.,  0.],
1783: (11)             [0.,  1.,  2.,  1.,  0.,  0.],
1784: (11)             [0.,  0.,  1.,  2.,  1.,  0.],
1785: (11)             [0.,  0.,  0.,  1.,  2.,  1.]])
1786: (4)          Decorator syntax is supported. The decorator can be called as
1787: (4)          a function to provide keyword arguments.
1788: (4)          >>>@np.vectorize
1789: (4)          ...def identity(x):
1790: (4)          ...    return x
1791: (4)
1792: (4)
1793: (4)
1794: (4)
1795: (4)
1796: (4)
1797: (4)
1798: (4)
1799: (4)
1800: (4)
1801: (4)
1802: (17)
1803: (8)
1804: (12)
1805: (12)
1806: (12)
1807: (8)
1808: (8)
1809: (8)
1810: (8)
1811: (12)
1812: (8)
1813: (8)
1814: (8)
1815: (8)
1816: (12)
1817: (8)
1818: (12)
1819: (8)
1820: (12)
1821: (16)
1822: (20)
1823: (8)
1824: (12)
1825: (8)
1826: (12)
1827: (8)
1828: (8)
1829: (12)
1830: (8)
1831: (8)
1832: (12)
1833: (8)
1834: (12)
1835: (4)
1836: (8)
1837: (8)
1838: (8)
1839: (12)
1840: (8)
1841: (12)
1842: (4)
1843: (8)
1844: (8)
1845: (8)
1846: (8)
1847: (8)

        def __init__(self, pyfunc=np._NoValue, otypes=None, doc=None,
                     excluded=None, cache=False, signature=None):
            if (pyfunc != np._NoValue) and (not callable(pyfunc)):
                part1 = "When used as a decorator, "
                part2 = "only accepts keyword arguments."
                raise TypeError(part1 + part2)
            self.pyfunc = pyfunc
            self.cache = cache
            self.signature = signature
            if pyfunc != np._NoValue and hasattr(pyfunc, '__name__'):
                self.__name__ = pyfunc.__name__
            self._ufunc = {} # Caching to improve default performance
            self._doc = None
            self.__doc__ = doc
            if doc is None and hasattr(pyfunc, '__doc__'):
                self.__doc__ = pyfunc.__doc__
            else:
                self._doc = doc
            if isinstance(otypes, str):
                for char in otypes:
                    if char not in typecodes['All']:
                        raise ValueError("Invalid otype specified: %s" % (char,))
            elif iterable(otypes):
                otypes = ''.join([_nx.dtype(x).char for x in otypes])
            elif otypes is not None:
                raise ValueError("Invalid otype specification")
            self.otypes = otypes
            if excluded is None:
                excluded = set()
            self.excluded = set(excluded)
            if signature is not None:
                self._in_and_out_core_dims = _parse_gufunc_signature(signature)
            else:
                self._in_and_out_core_dims = None
        def __init_stage_2(self, pyfunc, *args, **kwargs):
            self.__name__ = pyfunc.__name__
            self.pyfunc = pyfunc
            if self._doc is None:
                self.__doc__ = pyfunc.__doc__
            else:
                self.__doc__ = self._doc
        def __call_as_normal(self, *args, **kwargs):
            """
                Return arrays with the results of `pyfunc` broadcast (vectorized) over
                `args` and `kwargs` not in `excluded`.
            """
            excluded = self.excluded

```

```

1848: (8)             if not kwargs and not excluded:
1849: (12)             func = self.pyfunc
1850: (12)             vargs = args
1851: (8)         else:
1852: (12)             nargs = len(args)
1853: (12)             names = [_n for _n in kwargs if _n not in excluded]
1854: (12)             inds = [_i for _i in range(nargs) if _i not in excluded]
1855: (12)             the_args = list(args)
1856: (12)             def func(*vargs):
1857: (16)                 for _n, _i in enumerate(inds):
1858: (20)                     the_args[_i] = vargs[_n]
1859: (16)                     kwargs.update(zip(names, vargs[len(inds):])))
1860: (16)                     return self.pyfunc(*the_args, **kwargs)
1861: (12)                     vargs = [args[_i] for _i in inds]
1862: (12)                     vargs.extend([kwargs[_n] for _n in names])
1863: (8)             return self._vectorize_call(func=func, args=vargs)
1864: (4)         def __call__(self, *args, **kwargs):
1865: (8)             if self.pyfunc is np._NoValue:
1866: (12)                 self._init_stage_2(*args, **kwargs)
1867: (12)                 return self
1868: (8)             return self._call_as_normal(*args, **kwargs)
1869: (4)         def _get_ufunc_and_otypes(self, func, args):
1870: (8)             """Return (ufunc, otypes)."""
1871: (8)             if not args:
1872: (12)                 raise ValueError('args can not be empty')
1873: (8)             if self.otypes is not None:
1874: (12)                 otypes = self.otypes
1875: (12)                 nin = len(args)
1876: (12)                 nout = len(self.otypes)
1877: (12)                 if func is not self.pyfunc or nin not in self._ufunc:
1878: (16)                     ufunc = frompyfunc(func, nin, nout)
1879: (12)                 else:
1880: (16)                     ufunc = None # We'll get it from self._ufunc
1881: (12)                 if func is self.pyfunc:
1882: (16)                     ufunc = self._ufunc.setdefault(nin, ufunc)
1883: (8)             else:
1884: (12)                 args = [asarray(arg) for arg in args]
1885: (12)                 if builtins.any(arg.size == 0 for arg in args):
1886: (16)                     raise ValueError('cannot call `vectorize` on size 0 inputs '
1887: (33)                         'unless `otypes` is set')
1888: (12)                 inputs = [arg.flat[0] for arg in args]
1889: (12)                 outputs = func(*inputs)
1890: (12)                 if self.cache:
1891: (16)                     _cache = [outputs]
1892: (16)                     def __func(*vargs):
1893: (20)                         if _cache:
1894: (24)                             return _cache.pop()
1895: (20)                         else:
1896: (24)                             return func(*vargs)
1897: (12)                     else:
1898: (16)                         __func = func
1899: (12)                         if isinstance(outputs, tuple):
1900: (16)                             nout = len(outputs)
1901: (12)                         else:
1902: (16)                             nout = 1
1903: (16)                             outputs = (outputs,)
1904: (12)                             otypes = ''.join([asarray(outputs[_k]).dtype.char
1905: (30)                                 for _k in range(nout)])
1906: (12)                             ufunc = frompyfunc(__func, len(args), nout)
1907: (8)                         return ufunc, otypes
1908: (4)         def _vectorize_call(self, func, args):
1909: (8)             """Vectorized call to `func` over positional `args`."""
1910: (8)             if self.signature is not None:
1911: (12)                 res = self._vectorize_call_with_signature(func, args)
1912: (8)             elif not args:
1913: (12)                 res = func()
1914: (8)             else:
1915: (12)                 ufunc, otypes = self._get_ufunc_and_otypes(func=func, args=args)
1916: (12)                 inputs = [asanyarray(a, dtype=object) for a in args]

```

```

1917: (12)             outputs = ufunc(*inputs)
1918: (12)             if ufunc.nout == 1:
1919: (16)                 res = asanyarray(outputs, dtype=otypes[0])
1920: (12)             else:
1921: (16)                 res = tuple([asanyarray(x, dtype=t)
1922: (29)                               for x, t in zip(outputs, otypes)])
1923: (8)             return res
1924: (4)             def _vectorize_call_with_signature(self, func, args):
1925: (8)                 """Vectorized call over positional arguments with a signature."""
1926: (8)                 input_core_dims, output_core_dims = self._in_and_out_core_dims
1927: (8)                 if len(args) != len(input_core_dims):
1928: (12)                     raise TypeError('wrong number of positional arguments: '
1929: (28)                         'expected %r, got %r'
1930: (28)                         % (len(input_core_dims), len(args)))
1931: (8)                 args = tuple(asanyarray(arg) for arg in args)
1932: (8)                 broadcast_shape, dim_sizes = _parse_input_dimensions(
1933: (12)                     args, input_core_dims)
1934: (8)                 input_shapes = _calculate_shapes(broadcast_shape, dim_sizes,
1935: (41)                     input_core_dims)
1936: (8)                 args = [np.broadcast_to(arg, shape, subok=True)
1937: (16)                   for arg, shape in zip(args, input_shapes)] 
1938: (8)                 outputs = None
1939: (8)                 otypes = self.otypes
1940: (8)                 nout = len(output_core_dims)
1941: (8)                 for index in np.ndindex(*broadcast_shape):
1942: (12)                     results = func(*(arg[index] for arg in args))
1943: (12)                     n_results = len(results) if isinstance(results, tuple) else 1
1944: (12)                     if nout != n_results:
1945: (16)                         raise ValueError(
1946: (20)                             'wrong number of outputs from pyfunc: expected %r, got %r'
1947: (20)                             % (nout, n_results))
1948: (12)                     if nout == 1:
1949: (16)                         results = (results,)
1950: (12)                     if outputs is None:
1951: (16)                         for result, core_dims in zip(results, output_core_dims):
1952: (20)                             _update_dim_sizes(dim_sizes, result, core_dims)
1953: (16)                         outputs = _create_arrays(broadcast_shape, dim_sizes,
1954: (41)                             output_core_dims, otypes, results)
1955: (12)                         for output, result in zip(outputs, results):
1956: (16)                             output[index] = result
1957: (8)                     if outputs is None:
1958: (12)                         if otypes is None:
1959: (16)                             raise ValueError('cannot call `vectorize` on size 0 inputs '
1960: (33)                               'unless `otypes` is set')
1961: (12)                         if builtins.any(dim not in dim_sizes
1962: (28)                           for dims in output_core_dims
1963: (28)                             for dim in dims):
1964: (16)                             raise ValueError('cannot call `vectorize` with a signature '
1965: (33)                               'including new output dimensions on size 0 '
1966: (33)                               'inputs')
1967: (12)                         outputs = _create_arrays(broadcast_shape, dim_sizes,
1968: (37)                             output_core_dims, otypes)
1969: (8)                         return outputs[0] if nout == 1 else outputs
1970: (0)             def _cov_dispatcher(m, y=None, rowvar=None, bias=None, ddof=None,
1971: (20)                             fweights=None, aweights=None, *, dtype=None):
1972: (4)                 return (m, y, fweights, aweights)
1973: (0)             @array_function_dispatch(_cov_dispatcher)
1974: (0)             def cov(m, y=None, rowvar=True, bias=False, ddof=None, fweights=None,
1975: (8)                             aweights=None, *, dtype=None):
1976: (4)                             """
1977: (4)                             Estimate a covariance matrix, given data and weights.
1978: (4)                             Covariance indicates the level to which two variables vary together.
1979: (4)                             If we examine N-dimensional samples, :math:`X = [x_1, x_2, \dots x_N]^T` ,
1980: (4)                             then the covariance matrix element :math:`C_{ij}` is the covariance of
1981: (4)                             :math:`x_i` and :math:`x_j` . The element :math:`C_{ii}` is the variance
1982: (4)                             of :math:`x_i` .
1983: (4)                             See the notes for an outline of the algorithm.
1984: (4)                             Parameters
1985: (4)                             -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1986: (4)
1987: (8) m : array_like
1988: (8) A 1-D or 2-D array containing multiple variables and observations.
1989: (8) Each row of `m` represents a variable, and each column a single
1990: (4) observation of all those variables. Also see `rowvar` below.
1991: (8) y : array_like, optional
1992: (8) An additional set of variables and observations. `y` has the same form
1993: (4) as that of `m`.
1994: (8) rowvar : bool, optional
1995: (8) If `rowvar` is True (default), then each row represents a
relationship variable, with observations in the columns. Otherwise, the
1996: (8) is transposed: each column represents a variable, while the rows
1997: (8) contain observations.
1998: (4) bias : bool, optional
1999: (8) Default normalization (False) is by `` $(N - 1)$ ``, where `` $N$ `` is the
2000: (8) number of observations given (unbiased estimate). If `bias` is True,
2001: (8) then normalization is by `` $N$ ``. These values can be overridden by
using
2002: (8) the keyword ``ddof`` in numpy versions  $\geq 1.5$ .
2003: (4) ddof : int, optional
2004: (8) If not ``None`` the default value implied by `bias` is overridden.
2005: (8) Note that ``ddof=1`` will return the unbiased estimate, even if both
2006: (8) `fweights` and `aweights` are specified, and ``ddof=0`` will return
2007: (8) the simple average. See the notes for the details. The default value
2008: (8) is ``None``.
2009: (8) .. versionadded:: 1.5
2010: (4) fweights : array_like, int, optional
2011: (8) 1-D array of integer frequency weights; the number of times each
2012: (8) observation vector should be repeated.
2013: (8) .. versionadded:: 1.10
2014: (4) aweights : array_like, optional
2015: (8) 1-D array of observation vector weights. These relative weights are
2016: (8) typically large for observations considered "important" and smaller
for
2017: (8) observations considered less "important". If ``ddof=0`` the array of
2018: (8) weights can be used to assign probabilities to observation vectors.
2019: (8) .. versionadded:: 1.10
2020: (4) dtype : data-type, optional
2021: (8) Data-type of the result. By default, the return data-type will have
2022: (8) at least `numpy.float64` precision.
2023: (8) .. versionadded:: 1.20
2024: (4) Returns
2025: (4) -----
2026: (4) out : ndarray
2027: (8) The covariance matrix of the variables.
2028: (4) See Also
2029: (4) -----
2030: (4) corrcoef : Normalized covariance matrix
2031: (4) Notes
2032: (4) -----
2033: (4) Assume that the observations are in the columns of the observation
2034: (4) array `m` and let ``f = fweights`` and ``a = aweights`` for brevity. The
2035: (4) steps to compute the weighted covariance are as follows::
2036: (8) >>> m = np.arange(10, dtype=np.float64)
2037: (8) >>> f = np.arange(10) * 2
2038: (8) >>> a = np.arange(10) ** 2.
2039: (8) >>> ddof = 1
2040: (8) >>> w = f * a
2041: (8) >>> v1 = np.sum(w)
2042: (8) >>> v2 = np.sum(w * a)
2043: (8) >>> m -= np.sum(m * w, axis=None, keepdims=True) / v1
2044: (8) >>> cov = np.dot(m * w, m.T) * v1 / (v1**2 - ddof * v2)
2045: (4) Note that when ``a == 1``, the normalization factor
2046: (4) ``v1 / (v1**2 - ddof * v2)`` goes over to ``1 / (np.sum(f) - ddof)``
2047: (4) as it should.
2048: (4) Examples
2049: (4) -----
2050: (4) Consider two variables, :math:`x_0` and :math:`x_1`, which
2051: (4) correlate perfectly, but in opposite directions:
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

2052: (4)
2053: (4)
2054: (4)
2055: (11)
2056: (4)
2057: (4)
2058: (4)
2059: (4)
2060: (11)
2061: (4)
2062: (4)
2063: (4)
2064: (4)
2065: (4)
2066: (4)
2067: (4)
2068: (4)
2069: (11)
2070: (4)
2071: (4)
2072: (11)
2073: (4)
2074: (4)
2075: (4)
2076: (4)
2077: (8)
2078: (12)
2079: (4)
2080: (4)
2081: (8)
2082: (4)
2083: (8)
2084: (8)
2085: (12)
2086: (4)
2087: (8)
2088: (12)
2089: (8)
2090: (12)
2091: (4)
2092: (4)
2093: (8)
2094: (4)
2095: (8)
2096: (4)
2097: (8)
2098: (8)
2099: (12)
2100: (8)
2101: (4)
2102: (8)
2103: (12)
2104: (8)
2105: (12)
2106: (4)
2107: (4)
2108: (8)
2109: (8)
2110: (12)
2111: (16)
2112: (8)
2113: (12)
2114: (16)
2115: (8)
2116: (12)
2117: (16)
2118: (8)
2119: (12)
2120: (16)

    >>> x = np.array([[0, 2], [1, 1], [2, 0]]).T
    >>> x
    array([[0, 1, 2],
           [2, 1, 0]])
    Note how :math:`x_0` increases while :math:`x_1` decreases. The covariance
    matrix shows this clearly:
    >>> np.cov(x)
    array([[ 1., -1.],
           [-1.,  1.]])
    Note that element :math:`C_{0,1}`, which shows the correlation between
    :math:`x_0` and :math:`x_1`, is negative.
    Further, note how `x` and `y` are combined:
    >>> x = [-2.1, -1, 4.3]
    >>> y = [3, 1.1, 0.12]
    >>> X = np.stack((x, y), axis=0)
    >>> np.cov(X)
    array([[11.71      , -4.286      ], # may vary
           [-4.286      ,  2.144133]])]
    >>> np.cov(x, y)
    array([[11.71      , -4.286      ], # may vary
           [-4.286      ,  2.144133]])]
    >>> np.cov(x)
    array(11.71)
    """
    if ddof is not None and ddof != int(ddof):
        raise ValueError(
            "ddof must be integer")
    m = np.asarray(m)
    if m.ndim > 2:
        raise ValueError("m has more than 2 dimensions")
    if y is not None:
        y = np.asarray(y)
        if y.ndim > 2:
            raise ValueError("y has more than 2 dimensions")
    if dtype is None:
        if y is None:
            dtype = np.result_type(m, np.float64)
        else:
            dtype = np.result_type(m, y, np.float64)
    X = array(m, ndmin=2, dtype=dtype)
    if not rowvar and X.shape[0] != 1:
        X = X.T
    if X.shape[0] == 0:
        return np.array([]).reshape(0, 0)
    if y is not None:
        y = array(y, copy=False, ndmin=2, dtype=dtype)
        if not rowvar and y.shape[0] != 1:
            y = y.T
        X = np.concatenate((X, y), axis=0)
    if ddof is None:
        if bias == 0:
            ddof = 1
        else:
            ddof = 0
    w = None
    if fweights is not None:
        fweights = np.asarray(fweights, dtype=float)
        if not np.all(fweights == np.around(fweights)):
            raise TypeError(
                "fweights must be integer")
        if fweights.ndim > 1:
            raise RuntimeError(
                "cannot handle multidimensional fweights")
        if fweights.shape[0] != X.shape[1]:
            raise RuntimeError(
                "incompatible numbers of samples and fweights")
        if any(fweights < 0):
            raise ValueError(
                "fweights cannot be negative")

```

```

2121: (8)             w = fweights
2122: (4)             if aweights is not None:
2123: (8)                 aweights = np.asarray(aweights, dtype=float)
2124: (8)                 if aweights.ndim > 1:
2125: (12)                     raise RuntimeError(
2126: (16)                         "cannot handle multidimensional aweights")
2127: (8)                     if aweights.shape[0] != X.shape[1]:
2128: (12)                         raise RuntimeError(
2129: (16)                             "incompatible numbers of samples and aweights")
2130: (8)                     if any(aweights < 0):
2131: (12)                         raise ValueError(
2132: (16)                             "aweights cannot be negative")
2133: (8)                     if w is None:
2134: (12)                         w = aweights
2135: (8)                     else:
2136: (12)                         w *= aweights
2137: (4)             avg, w_sum = average(X, axis=1, weights=w, returned=True)
2138: (4)             w_sum = w_sum[0]
2139: (4)             if w is None:
2140: (8)                 fact = X.shape[1] - ddof
2141: (4)             elif ddof == 0:
2142: (8)                 fact = w_sum
2143: (4)             elif aweights is None:
2144: (8)                 fact = w_sum - ddof
2145: (4)             else:
2146: (8)                 fact = w_sum - ddof*sum(w*aweights)/w_sum
2147: (4)             if fact <= 0:
2148: (8)                 warnings.warn("Degrees of freedom <= 0 for slice",
2149: (22)                               RuntimeWarning, stacklevel=2)
2150: (8)                 fact = 0.0
2151: (4)             X -= avg[:, None]
2152: (4)             if w is None:
2153: (8)                 X_T = X.T
2154: (4)             else:
2155: (8)                 X_T = (X*w).T
2156: (4)             c = dot(X, X_T.conj())
2157: (4)             c *= np.true_divide(1, fact)
2158: (4)             return c.squeeze()
2159: (0)             def _corrcoef_dispatcher(x, y=None, rowvar=None, bias=None, ddof=None,
2160: (25)                           *, dtype=None):
2161: (4)                 return (x, y)
2162: (0)             @array_function_dispatch(_corrcoef_dispatcher)
2163: (0)             def corrcoef(x, y=None, rowvar=True, bias=np._NoValue, ddof=np._NoValue, *,
2164: (13)                           dtype=None):
2165: (4)                 """
2166: (4)                     Return Pearson product-moment correlation coefficients.
2167: (4)                     Please refer to the documentation for `cov` for more detail. The
2168: (4)                     relationship between the correlation coefficient matrix, `R`, and the
2169: (4)                     covariance matrix, `C`, is
2170: (4)                     .. math:: R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} C_{jj}}}
2171: (4)                     The values of `R` are between -1 and 1, inclusive.
2172: (4)                     Parameters
2173: (4)                     -----
2174: (4)                     x : array_like
2175: (8)                         A 1-D or 2-D array containing multiple variables and observations.
2176: (8)                         Each row of `x` represents a variable, and each column a single
2177: (8)                         observation of all those variables. Also see `rowvar` below.
2178: (4)                     y : array_like, optional
2179: (8)                         An additional set of variables and observations. `y` has the same
2180: (8)                         shape as `x`.
2181: (4)                     rowvar : bool, optional
2182: (8)                         If `rowvar` is True (default), then each row represents a
2183: (8)                         variable, with observations in the columns. Otherwise, the
2184: (8)                         is transposed: each column represents a variable, while the rows
2185: (8)                         contain observations.
2186: (4)                     bias : _NoValue, optional
2187: (8)                         Has no effect, do not use.
2188: (8)                         .. deprecated:: 1.10.0

```

```

2189: (4)           ddof : _NoValue, optional
2190: (8)             Has no effect, do not use.
2191: (8)               .. deprecated:: 1.10.0
2192: (4)             dtype : data-type, optional
2193: (8)               Data-type of the result. By default, the return data-type will have
2194: (8)               at least `numpy.float64` precision.
2195: (8)               .. versionadded:: 1.20
2196: (4)             Returns
2197: (4)             -----
2198: (4)             R : ndarray
2199: (8)               The correlation coefficient matrix of the variables.
2200: (4)             See Also
2201: (4)             -----
2202: (4)             cov : Covariance matrix
2203: (4)             Notes
2204: (4)             -----
2205: (4)             Due to floating point rounding the resulting array may not be Hermitian,
2206: (4)             the diagonal elements may not be 1, and the elements may not satisfy the
2207: (4)             inequality  $\text{abs}(a) \leq 1$ . The real and imaginary parts are clipped to the
2208: (4)             interval [-1, 1] in an attempt to improve on that situation but is not
2209: (4)             much help in the complex case.
2210: (4)             This function accepts but discards arguments `bias` and `ddof`. This is
2211: (4)             for backwards compatibility with previous versions of this function.

These
2212: (4)             arguments had no effect on the return values of the function and can be
2213: (4)             safely ignored in this and previous versions of numpy.

Examples
2215: (4)             -----
2216: (4)             In this example we generate two random arrays, ``xarr`` and ``yarr``, and
2217: (4)             compute the row-wise and column-wise Pearson correlation coefficients,
2218: (4)             ``R``. Since ``rowvar`` is true by default, we first find the row-wise
2219: (4)             Pearson correlation coefficients between the variables of ``xarr``.
2220: (4)             >>> import numpy as np
2221: (4)             >>> rng = np.random.default_rng(seed=42)
2222: (4)             >>> xarr = rng.random((3, 3))
2223: (4)             >>> xarr
2224: (4)             array([[0.77395605, 0.43887844, 0.85859792],
2225: (11)                 [0.69736803, 0.09417735, 0.97562235],
2226: (11)                 [0.7611397 , 0.78606431, 0.12811363]])
2227: (4)             >>> R1 = np.corrcoef(xarr)
2228: (4)             >>> R1
2229: (4)             array([[ 1.          ,  0.99256089, -0.68080986],
2230: (11)                 [ 0.99256089,  1.          , -0.76492172],
2231: (11)                 [-0.68080986, -0.76492172,  1.          ]])
2232: (4)             If we add another set of variables and observations ``yarr``, we can
2233: (4)             compute the row-wise Pearson correlation coefficients between the
2234: (4)             variables in ``xarr`` and ``yarr``.
2235: (4)             >>> yarr = rng.random((3, 3))
2236: (4)             >>> yarr
2237: (4)             array([[0.45038594, 0.37079802, 0.92676499],
2238: (11)                 [0.64386512, 0.82276161, 0.4434142 ],
2239: (11)                 [0.22723872, 0.55458479, 0.06381726]])
2240: (4)             >>> R2 = np.corrcoef(xarr, yarr)
2241: (4)             >>> R2
2242: (4)             array([[ 1.          ,  0.99256089, -0.68080986,  0.75008178, -0.934284 ,
2243: (12)                 -0.99004057],
2244: (11)                 [ 0.99256089,  1.          , -0.76492172,  0.82502011, -0.97074098,
2245: (12)                 -0.99981569],
2246: (11)                 [-0.68080986, -0.76492172,  1.          , -0.99507202,  0.89721355,
2247: (13)                 0.77714685],
2248: (11)                 [ 0.75008178,  0.82502011, -0.99507202,  1.          , -0.93657855,
2249: (12)                 -0.83571711],
2250: (11)                 [-0.934284 , -0.97074098,  0.89721355, -0.93657855,  1.          ,
2251: (13)                 0.97517215],
2252: (11)                 [-0.99004057, -0.99981569,  0.77714685, -0.83571711,  0.97517215,
2253: (13)                 1.          ]])
2254: (4)             Finally if we use the option ``rowvar=False``, the columns are now
2255: (4)             being treated as the variables and we will find the column-wise Pearson
2256: (4)             correlation coefficients between variables in ``xarr`` and ``yarr``.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2257: (4)          >>> R3 = np.corrcoef(xarr, yarr, rowvar=False)
2258: (4)          >>> R3
2259: (4)          array([[ 1.          ,  0.77598074, -0.47458546, -0.75078643, -0.9665554 ,
2260: (13)            0.22423734],
2261: (11)           [ 0.77598074,  1.          , -0.92346708, -0.99923895, -0.58826587,
2262: (12)             -0.44069024],
2263: (11)           [-0.47458546, -0.92346708,  1.          ,  0.93773029,  0.23297648,
2264: (13)             0.75137473],
2265: (11)           [-0.75078643, -0.99923895,  0.93773029,  1.          ,  0.55627469,
2266: (13)             0.47536961],
2267: (11)           [-0.9665554 , -0.58826587,  0.23297648,  0.55627469,  1.          ,
2268: (12)             -0.46666491],
2269: (11)           [ 0.22423734, -0.44069024,  0.75137473,  0.47536961, -0.46666491,
2270: (13)             1.          ]])
2271: (4)          """
2272: (4)          if bias is not np._NoValue or ddof is not np._NoValue:
2273: (8)              warnings.warn('bias and ddof have no effect and are deprecated',
2274: (22)                  DeprecationWarning, stacklevel=2)
2275: (4)          c = cov(x, y, rowvar, dtype=dtype)
2276: (4)          try:
2277: (8)              d = diag(c)
2278: (4)          except ValueError:
2279: (8)              return c / c
2280: (4)          stddev = sqrt(d.real)
2281: (4)          c /= stddev[:, None]
2282: (4)          c /= stddev[None, :]
2283: (4)          np.clip(c.real, -1, 1, out=c.real)
2284: (4)          if np.iscomplexobj(c):
2285: (8)              np.clip(c.imag, -1, 1, out=c.imag)
2286: (4)          return c
2287: (0)          @set_module('numpy')
2288: (0)          def blackman(M):
2289: (4)          """
2290: (4)          Return the Blackman window.
2291: (4)          The Blackman window is a taper formed by using the first three
2292: (4)          terms of a summation of cosines. It was designed to have close to the
2293: (4)          minimal leakage possible. It is close to optimal, only slightly worse
2294: (4)          than a Kaiser window.
2295: (4)          Parameters
2296: (4)          -----
2297: (4)          M : int
2298: (8)              Number of points in the output window. If zero or less, an empty
2299: (8)              array is returned.
2300: (4)          Returns
2301: (4)          -----
2302: (4)          out : ndarray
2303: (8)              The window, with the maximum value normalized to one (the value one
2304: (8)              appears only if the number of samples is odd).
2305: (4)          See Also
2306: (4)          -----
2307: (4)          bartlett, hamming, hanning, kaiser
2308: (4)          Notes
2309: (4)          -----
2310: (4)          The Blackman window is defined as
2311: (4)          .. math:: w(n) = 0.42 - 0.5 \cos(2\pi n/M) + 0.08 \cos(4\pi n/M)
2312: (4)          Most references to the Blackman window come from the signal processing
2313: (4)          literature, where it is used as one of many windowing functions for
2314: (4)          smoothing values. It is also known as an apodization (which means
2315: (4)          "removing the foot", i.e. smoothing discontinuities at the beginning
2316: (4)          and end of the sampled signal) or tapering function. It is known as a
2317: (4)          "near optimal" tapering function, almost as good (by some measures)
2318: (4)          as the kaiser window.
2319: (4)          References
2320: (4)          -----
2321: (4)          Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra,
2322: (4)          Dover Publications, New York.
2323: (4)          Oppenheim, A.V., and R.W. Schafer. Discrete-Time Signal Processing.
2324: (4)          Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.
2325: (4)          Examples

```

```

2326: (4)
2327: (4)
2328: (4)
2329: (4)
2330: (12)
2331: (12)
2332: (12)
2333: (4)
2334: (4)
2335: (4)
2336: (4)
2337: (4)
2338: (4)
2339: (4)
2340: (4)
2341: (4)
2342: (4)
2343: (4)
2344: (4)
2345: (4)
2346: (4)
2347: (4)
2348: (4)
2349: (4)
2350: (4)
2351: (4)
2352: (4)
2353: (4)
2354: (4)
2355: (4)
2356: (4)
2357: (4)
2358: (4)
2359: (4)
2360: (4)
2361: (4)
2362: (4)
2363: (4)
2364: (4)
2365: (4)
2366: (4)
2367: (4)
2368: (8)
2369: (4)
2370: (8)
2371: (4)
2372: (4)
2373: (0)
2374: (0)
2375: (4)
2376: (4)
2377: (4)
2378: (4)
2379: (4)
2380: (4)
2381: (4)
2382: (4)
2383: (4)
2384: (8)
2385: (8)
2386: (4)
2387: (4)
2388: (4)
2389: (8)
2390: (8)
2391: (8)
2392: (4)
2393: (4)
2394: (4)

-----  

>>> import matplotlib.pyplot as plt  

>>> np.blackman(12)  

array([-1.38777878e-17,  3.26064346e-02,  1.59903635e-01, # may vary  

       4.14397981e-01,  7.36045180e-01,  9.67046769e-01,  

       9.67046769e-01,  7.36045180e-01,  4.14397981e-01,  

       1.59903635e-01,  3.26064346e-02, -1.38777878e-17])  

Plot the window and the frequency response:  

>>> from numpy.fft import fft, fftshift  

>>> window = np.blackman(51)  

>>> plt.plot(window)  

[<matplotlib.lines.Line2D object at 0x...>]  

>>> plt.title("Blackman window")  

Text(0.5, 1.0, 'Blackman window')  

>>> plt.ylabel("Amplitude")  

Text(0, 0.5, 'Amplitude')  

>>> plt.xlabel("Sample")  

Text(0.5, 0, 'Sample')  

>>> plt.show()  

>>> plt.figure()  

<Figure size 640x480 with 0 Axes>  

>>> A = fft(window, 2048) / 25.5  

>>> mag = np.abs(fftshift(A))  

>>> freq = np.linspace(-0.5, 0.5, len(A))  

>>> with np.errstate(divide='ignore', invalid='ignore'):  

...     response = 20 * np.log10(mag)  

...  

>>> response = np.clip(response, -100, 100)  

>>> plt.plot(freq, response)  

[<matplotlib.lines.Line2D object at 0x...>]  

>>> plt.title("Frequency response of Blackman window")  

Text(0.5, 1.0, 'Frequency response of Blackman window')  

>>> plt.ylabel("Magnitude [dB]")  

Text(0, 0.5, 'Magnitude [dB]')  

>>> plt.xlabel("Normalized frequency [cycles per sample]")  

Text(0.5, 0, 'Normalized frequency [cycles per sample]')  

>>> _ = plt.axis('tight')  

>>> plt.show()  

"""  

values = np.array([0.0, M])  

M = values[1]  

if M < 1:  

    return array([], dtype=values.dtype)  

if M == 1:  

    return ones(1, dtype=values.dtype)  

n = arange(1-M, M, 2)  

return 0.42 + 0.5*cos(pi*n/(M-1)) + 0.08*cos(2.0*pi*n/(M-1))  

@set_module('numpy')
def bartlett(M):
"""
Return the Bartlett window.

The Bartlett window is very similar to a triangular window, except
that the end points are at zero. It is often used in signal
processing for tapering a signal, without generating too much
ripple in the frequency domain.

Parameters
-----
M : int
    Number of points in the output window. If zero or less, an
    empty array is returned.

Returns
-----
out : array
    The triangular window, with the maximum value normalized to one
    (the value one appears only if the number of samples is odd), with
    the first and last samples equal to zero.

See Also
-----
blackman, hamming, hanning, kaiser
"""

-----  


```

```

2395: (4)          Notes
2396: (4)
2397: (4)
2398: (4)
2399: (14)         -----
2400: (14)         The Bartlett window is defined as
2401: (4)         .. math:: w(n) = \frac{2}{M-1} \left( \frac{M-1}{2} - \left| n - \frac{M-1}{2} \right| \right)
2402: (4)         Most references to the Bartlett window come from the signal processing
2403: (4)         literature, where it is used as one of many windowing functions for
2404: (4)         smoothing values. Note that convolution with this window produces linear
2405: (4)         interpolation. It is also known as an apodization (which means "removing
2406: (4)         the foot", i.e. smoothing discontinuities at the beginning and end of the
2407: (4)         sampled signal) or tapering function. The Fourier transform of the
2408: (4)         Bartlett window is the product of two sinc functions. Note the excellent
2409: (4)         discussion in Kanasewich [2]_.
2410: (4)         References
2411: (4)         -----
2412: (11)         .. [1] M.S. Bartlett, "Periodogram Analysis and Continuous Spectra",
2413: (4)             Biometrika 37, 1-16, 1950.
2414: (11)         .. [2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics",
2415: (4)             The University of Alberta Press, 1975, pp. 109-110.
2416: (11)         .. [3] A.V. Oppenheim and R.W. Schafer, "Discrete-Time Signal
2417: (4)             Processing", Prentice-Hall, 1999, pp. 468-471.
2418: (11)         .. [4] Wikipedia, "Window function",
2419: (4)             https://en.wikipedia.org/wiki/Window\_function
2420: (11)         .. [5] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling,
2421: (4)             "Numerical Recipes", Cambridge University Press, 1986, page 429.
2422: (4)         Examples
2423: (4)         -----
2424: (4)         >>> import matplotlib.pyplot as plt
2425: (4)         array([ 0.           ,  0.18181818,  0.36363636,  0.54545455,  0.72727273, #
may vary
2426: (12)           0.90909091,  0.90909091,  0.72727273,  0.54545455,  0.36363636,
2427: (12)           0.18181818,  0.           ])
2428: (4)         Plot the window and its frequency response (requires SciPy and
matplotlib):
2429: (4)         >>> from numpy.fft import fft, fftshift
2430: (4)         >>> window = np.bartlett(51)
2431: (4)         >>> plt.plot(window)
2432: (4)           [<matplotlib.lines.Line2D object at 0x...>]
2433: (4)         >>> plt.title("Bartlett window")
2434: (4)           Text(0.5, 1.0, 'Bartlett window')
2435: (4)         >>> plt.ylabel("Amplitude")
2436: (4)           Text(0, 0.5, 'Amplitude')
2437: (4)         >>> plt.xlabel("Sample")
2438: (4)           Text(0.5, 0, 'Sample')
2439: (4)         >>> plt.show()
2440: (4)         >>> plt.figure()
2441: (4)           <Figure size 640x480 with 0 Axes>
2442: (4)         >>> A = fft(window, 2048) / 25.5
2443: (4)         >>> mag = np.abs(fftshift(A))
2444: (4)         >>> freq = np.linspace(-0.5, 0.5, len(A))
2445: (4)         >>> with np.errstate(divide='ignore', invalid='ignore'):
2446: (4)             ...     response = 20 * np.log10(mag)
2447: (4)             ...
2448: (4)             >>> response = np.clip(response, -100, 100)
2449: (4)             >>> plt.plot(freq, response)
2450: (4)               [<matplotlib.lines.Line2D object at 0x...>]
2451: (4)             >>> plt.title("Frequency response of Bartlett window")
2452: (4)               Text(0.5, 1.0, 'Frequency response of Bartlett window')
2453: (4)             >>> plt.ylabel("Magnitude [dB]")
2454: (4)               Text(0, 0.5, 'Magnitude [dB]')
2455: (4)             >>> plt.xlabel("Normalized frequency [cycles per sample]")
2456: (4)               Text(0.5, 0, 'Normalized frequency [cycles per sample]')
2457: (4)             >>> _ = plt.axis('tight')
2458: (4)             >>> plt.show()
2459: (4)             """
2460: (4)             values = np.array([0.0, M])
2461: (4)             M = values[1]

```

```

2462: (4)         if M < 1:
2463: (8)             return array([], dtype=values.dtype)
2464: (4)         if M == 1:
2465: (8)             return ones(1, dtype=values.dtype)
2466: (4)         n = arange(1-M, M, 2)
2467: (4)         return where(less_equal(n, 0), 1 + n/(M-1), 1 - n/(M-1))
2468: (0) @set_module('numpy')
2469: (0) def hanning(M):
2470: (4)     """
2471: (4)         Return the Hanning window.
2472: (4)         The Hanning window is a taper formed by using a weighted cosine.
2473: (4)         Parameters
2474: (4)         -----
2475: (4)         M : int
2476: (8)             Number of points in the output window. If zero or less, an
2477: (8)             empty array is returned.
2478: (4)         Returns
2479: (4)         -----
2480: (4)         out : ndarray, shape(M,)
2481: (8)             The window, with the maximum value normalized to one (the value
2482: (8)             one appears only if `M` is odd).
2483: (4)         See Also
2484: (4)         -----
2485: (4)         bartlett, blackman, hamming, kaiser
2486: (4)         Notes
2487: (4)         -----
2488: (4)         The Hanning window is defined as
2489: (4)         .. math:: w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right)
2490: (15)             \quad 0 \leq n \leq M-1
2491: (4)         The Hanning was named for Julius von Hann, an Austrian meteorologist.
2492: (4)         It is also known as the Cosine Bell. Some authors prefer that it be
2493: (4)         called a Hann window, to help avoid confusion with the very similar
2494: (4)         Hamming window.
2495: (4)         Most references to the Hanning window come from the signal processing
2496: (4)         literature, where it is used as one of many windowing functions for
2497: (4)         smoothing values. It is also known as an apodization (which means
2498: (4)         "removing the foot", i.e. smoothing discontinuities at the beginning
2499: (4)         and end of the sampled signal) or tapering function.
2500: (4)         References
2501: (4)         -----
2502: (4)         .. [1] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power
2503: (11)             spectra, Dover Publications, New York.
2504: (4)         .. [2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics",
2505: (11)             The University of Alberta Press, 1975, pp. 106-108.
2506: (4)         .. [3] Wikipedia, "Window function",
2507: (11)             https://en.wikipedia.org/wiki/Window\_function
2508: (4)         .. [4] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling,
2509: (11)             "Numerical Recipes", Cambridge University Press, 1986, page 425.
2510: (4)         Examples
2511: (4)         -----
2512: (4)         >>> np.hanning(12)
2513: (4)         array([0.          , 0.07937323, 0.29229249, 0.57115742, 0.82743037,
2514: (11)             0.97974649, 0.97974649, 0.82743037, 0.57115742, 0.29229249,
2515: (11)             0.07937323, 0.          ])
2516: (4)         Plot the window and its frequency response:
2517: (4)         >>> import matplotlib.pyplot as plt
2518: (4)         >>> from numpy.fft import fft, fftshift
2519: (4)         >>> window = np.hanning(51)
2520: (4)         >>> plt.plot(window)
2521: (4)         [

```

```

2531: (4)          >>> A = fft(window, 2048) / 25.5
2532: (4)          >>> mag = np.abs(fftshift(A))
2533: (4)          >>> freq = np.linspace(-0.5, 0.5, len(A))
2534: (4)          >>> with np.errstate(divide='ignore', invalid='ignore'):
2535: (4)              ...      response = 20 * np.log10(mag)
2536: (4)              ...
2537: (4)          >>> response = np.clip(response, -100, 100)
2538: (4)          >>> plt.plot(freq, response)
2539: (4)          [<matplotlib.lines.Line2D object at 0x...>]
2540: (4)          >>> plt.title("Frequency response of the Hann window")
2541: (4)          Text(0.5, 1.0, 'Frequency response of the Hann window')
2542: (4)          >>> plt.ylabel("Magnitude [dB]")
2543: (4)          Text(0, 0.5, 'Magnitude [dB]')
2544: (4)          >>> plt.xlabel("Normalized frequency [cycles per sample]")
2545: (4)          Text(0.5, 0, 'Normalized frequency [cycles per sample]')
2546: (4)          >>> plt.axis('tight')
2547: (4)          ...
2548: (4)          >>> plt.show()
2549: (4)          """
2550: (4)          values = np.array([0.0, M])
2551: (4)          M = values[1]
2552: (4)          if M < 1:
2553: (8)              return array([], dtype=values.dtype)
2554: (4)          if M == 1:
2555: (8)              return ones(1, dtype=values.dtype)
2556: (4)          n = arange(1-M, M, 2)
2557: (4)          return 0.5 + 0.5*cos(pi*n/(M-1))
2558: (0)          @set_module('numpy')
2559: (0)          def hamming(M):
2560: (4)          """
2561: (4)          Return the Hamming window.
2562: (4)          The Hamming window is a taper formed by using a weighted cosine.
2563: (4)          Parameters
2564: (4)          -----
2565: (4)          M : int
2566: (8)              Number of points in the output window. If zero or less, an
2567: (8)              empty array is returned.
2568: (4)          Returns
2569: (4)          -----
2570: (4)          out : ndarray
2571: (8)              The window, with the maximum value normalized to one (the value
2572: (8)              one appears only if the number of samples is odd).
2573: (4)          See Also
2574: (4)          -----
2575: (4)          bartlett, blackman, hanning, kaiser
2576: (4)          Notes
2577: (4)          -----
2578: (4)          The Hamming window is defined as
2579: (4)          .. math:: w(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{M-1}\right)
2580: (15)          \quad \quad \quad 0 \leq n \leq M-1
2581: (4)          The Hamming was named for R. W. Hamming, an associate of J. W. Tukey
2582: (4)          and is described in Blackman and Tukey. It was recommended for
2583: (4)          smoothing the truncated autocovariance function in the time domain.
2584: (4)          Most references to the Hamming window come from the signal processing
2585: (4)          literature, where it is used as one of many windowing functions for
2586: (4)          smoothing values. It is also known as an apodization (which means
2587: (4)          "removing the foot", i.e. smoothing discontinuities at the beginning
2588: (4)          and end of the sampled signal) or tapering function.
2589: (4)          References
2590: (4)          -----
2591: (4)          .. [1] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power
2592: (11)          spectra, Dover Publications, New York.
2593: (4)          .. [2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The
2594: (11)          University of Alberta Press, 1975, pp. 109-110.
2595: (4)          .. [3] Wikipedia, "Window function",
2596: (11)          https://en.wikipedia.org/wiki/Window_function
2597: (4)          .. [4] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling,
2598: (11)          "Numerical Recipes", Cambridge University Press, 1986, page 425.
2599: (4)          Examples

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2600: (4)
2601: (4)
2602: (4)
may vary
2603: (12)
2604: (12)
2605: (4)
2606: (4)
2607: (4)
2608: (4)
2609: (4)
2610: (4)
[<matplotlib.lines.Line2D object at 0x...>]
2611: (4)
2612: (4)
2613: (4)
2614: (4)
2615: (4)
2616: (4)
2617: (4)
2618: (4)
2619: (4)
2620: (4)
2621: (4)
2622: (4)
2623: (4)
2624: (4)
2625: (4)
2626: (4)
[<matplotlib.lines.Line2D object at 0x...>]
2627: (4)
2628: (4)
2629: (4)
2630: (4)
2631: (4)
2632: (4)
2633: (4)
2634: (4)
2635: (4)
2636: (4)
2637: (4)
2638: (4)
2639: (4)
2640: (8)
2641: (4)
2642: (8)
2643: (4)
2644: (4)
2645: (0)
_i0A = [
-4.41534164647933937950E-18,
3.33079451882223809783E-17,
-2.43127984654795469359E-16,
1.71539128555513303061E-15,
-1.16853328779934516808E-14,
7.67618549860493561688E-14,
-4.85644678311192946090E-13,
2.95505266312963983461E-12,
-1.72682629144155570723E-11,
9.67580903537323691224E-11,
-5.18979560163526290666E-10,
2.65982372468238665035E-9,
-1.30002500998624804212E-8,
6.04699502254191894932E-8,
-2.67079385394061173391E-7,
1.11738753912010371815E-6,
-4.41673835845875056359E-6,
1.64484480707288970893E-5,
-5.75419501008210370398E-5,
1.88502885095841655729E-4,
-5.76375574538582365885E-4,
1.63947561694133579842E-3,

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2668: (4)           -4.32430999505057594430E-3,
2669: (4)           1.05464603945949983183E-2,
2670: (4)           -2.37374148058994688156E-2,
2671: (4)           4.93052842396707084878E-2,
2672: (4)           -9.49010970480476444210E-2,
2673: (4)           1.71620901522208775349E-1,
2674: (4)           -3.04682672343198398683E-1,
2675: (4)           6.76795274409476084995E-1
2676: (4)           ]
2677: (0)           _i0B = [
2678: (4)             -7.23318048787475395456E-18,
2679: (4)             -4.83050448594418207126E-18,
2680: (4)             4.46562142029675999901E-17,
2681: (4)             3.46122286769746109310E-17,
2682: (4)             -2.82762398051658348494E-16,
2683: (4)             -3.42548561967721913462E-16,
2684: (4)             1.77256013305652638360E-15,
2685: (4)             3.81168066935262242075E-15,
2686: (4)             -9.55484669882830764870E-15,
2687: (4)             -4.15056934728722208663E-14,
2688: (4)             1.54008621752140982691E-14,
2689: (4)             3.85277838274214270114E-13,
2690: (4)             7.18012445138366623367E-13,
2691: (4)             -1.79417853150680611778E-12,
2692: (4)             -1.32158118404477131188E-11,
2693: (4)             -3.14991652796324136454E-11,
2694: (4)             1.18891471078464383424E-11,
2695: (4)             4.94060238822496958910E-10,
2696: (4)             3.39623202570838634515E-9,
2697: (4)             2.26666899049817806459E-8,
2698: (4)             2.04891858946906374183E-7,
2699: (4)             2.89137052083475648297E-6,
2700: (4)             6.88975834691682398426E-5,
2701: (4)             3.36911647825569408990E-3,
2702: (4)             8.04490411014108831608E-1
2703: (4)           ]
2704: (0)           def _chbevl(x, vals):
2705: (4)             b0 = vals[0]
2706: (4)             b1 = 0.0
2707: (4)             for i in range(1, len(vals)):
2708: (8)               b2 = b1
2709: (8)               b1 = b0
2710: (8)               b0 = x*b1 - b2 + vals[i]
2711: (4)             return 0.5*(b0 - b2)
2712: (0)           def _i0_1(x):
2713: (4)             return exp(x) * _chbevl(x/2.0-2, _i0A)
2714: (0)           def _i0_2(x):
2715: (4)             return exp(x) * _chbevl(32.0/x - 2.0, _i0B) / sqrt(x)
2716: (0)           def _i0_dispatcher(x):
2717: (4)             return (x,)
2718: (0)           @array_function_dispatch(_i0_dispatcher)
2719: (0)           def i0(x):
2720: (4)             """
2721: (4)               Modified Bessel function of the first kind, order 0.
2722: (4)               Usually denoted :math:`I_0`.
2723: (4)               Parameters
2724: (4)               -----
2725: (4)               x : array_like of float
2726: (8)                 Argument of the Bessel function.
2727: (4)               Returns
2728: (4)               -----
2729: (4)               out : ndarray, shape = x.shape, dtype = float
2730: (8)                 The modified Bessel function evaluated at each of the elements of `x`.
2731: (4)               See Also
2732: (4)               -----
2733: (4)               scipy.special.i0, scipy.special.iv, scipy.special.ive
2734: (4)               Notes
2735: (4)               -----
2736: (4)               The scipy implementation is recommended over this function: it is a

```

proper ufunc written in C, and more than an order of magnitude faster. We use the algorithm published by Clenshaw [1] and referenced by Abramowitz and Stegun [2], for which the function domain is partitioned into the two intervals  $[0,8]$  and  $(8,\infty)$ , and Chebyshev polynomial expansions are employed in each interval. Relative error on the domain  $[0,30]$  using IEEE arithmetic is documented [3] as having a peak of  $5.8e-16$  with an rms of  $1.4e-16$  ( $n = 30000$ ).

References

-----

- .. [1] C. W. Clenshaw, "Chebyshev series for mathematical functions", in \*National Physical Laboratory Mathematical Tables\*, vol. 5, London: Her Majesty's Stationery Office, 1962.
- .. [2] M. Abramowitz and I. A. Stegun, \*Handbook of Mathematical Functions\*, 10th printing, New York: Dover, 1964, pp. 379.  
[https://personal.math.ubc.ca/~cbm/aands/page\\_379.htm](https://personal.math.ubc.ca/~cbm/aands/page_379.htm)
- .. [3] <https://metacpan.org/pod/distribution/Math-Cephes/lib/Math/Cephes.pod#i0:-Modified-Bessel-function-of-order-zero>

Examples

-----

```
>>> np.i0(0.)
array(1.0)
>>> np.i0([0, 1, 2, 3])
array([1.          , 1.26606588, 2.2795853 , 4.88079259])
```

"""

```
x = np.asarray(x)
if x.dtype.kind == 'c':
    raise TypeError("i0 not supported for complex values")
if x.dtype.kind != 'f':
    x = x.astype(float)
x = np.abs(x)
return piecewise(x, [x <= 8.0], [_i0_1, _i0_2])
```

@set\_module('numpy')

```
def kaiser(M, beta):
    """
    Return the Kaiser window.

    The Kaiser window is a taper formed by using a Bessel function.

    Parameters
    -----
    M : int
        Number of points in the output window. If zero or less, an
        empty array is returned.
    beta : float
        Shape parameter for window.

    Returns
    -----
    out : array
        The window, with the maximum value normalized to one (the value
        one appears only if the number of samples is odd).
    See Also
    -----
    bartlett, blackman, hamming, hanning
    Notes
    -----
    The Kaiser window is defined as
    .. math:: w(n) = I_0 \left( \beta \sqrt{1 - \frac{4n^2}{(M-1)^2}} \right) / I_0(\beta)
    with
    .. math:: \quad -\frac{M-1}{2} \leq n \leq \frac{M-1}{2},
    where :math:`I_0` is the modified zeroth-order Bessel function.
    The Kaiser was named for Jim Kaiser, who discovered a simple
    approximation to the DPSS window based on Bessel functions. The Kaiser
    window is a very good approximation to the Digital Prolate Spheroidal
    Sequence, or Slepian window, which is the transform which maximizes the
    energy in the main lobe of the window relative to total energy.
    The Kaiser can approximate many other windows by varying the beta
    parameter.
```

=====

```
beta Window shape
```

=====

```

2805: (4)          0      Rectangular
2806: (4)          5      Similar to a Hamming
2807: (4)          6      Similar to a Hanning
2808: (4)          8.6    Similar to a Blackman
2809: (4)          ===== =====
2810: (4)          A beta value of 14 is probably a good starting point. Note that as beta
2811: (4)          gets large, the window narrows, and so the number of samples needs to be
2812: (4)          large enough to sample the increasingly narrow spike, otherwise NaNs will
2813: (4)          get returned.
2814: (4)          Most references to the Kaiser window come from the signal processing
2815: (4)          literature, where it is used as one of many windowing functions for
2816: (4)          smoothing values. It is also known as an apodization (which means
2817: (4)          "removing the foot", i.e. smoothing discontinuities at the beginning
2818: (4)          and end of the sampled signal) or tapering function.
2819: (4)          References
2820: (4)          -----
2821: (4)          .. [1] J. F. Kaiser, "Digital Filters" - Ch 7 in "Systems analysis by
2822: (11)          digital computer", Editors: F.F. Kuo and J.F. Kaiser, p 218-285.
2823: (11)          John Wiley and Sons, New York, (1966).
2824: (4)          .. [2] E.R. Kanasewich, "Time Sequence Analysis in Geophysics", The
2825: (11)          University of Alberta Press, 1975, pp. 177-178.
2826: (4)          .. [3] Wikipedia, "Window function",
2827: (11)          https://en.wikipedia.org/wiki/Window_function
2828: (4)          Examples
2829: (4)          -----
2830: (4)          >>> import matplotlib.pyplot as plt
2831: (4)          >>> np.kaiser(12, 14)
2832: (5)          array([7.72686684e-06, 3.46009194e-03, 4.65200189e-02, # may vary
2833: (12)          2.29737120e-01, 5.99885316e-01, 9.45674898e-01,
2834: (12)          9.45674898e-01, 5.99885316e-01, 2.29737120e-01,
2835: (12)          4.65200189e-02, 3.46009194e-03, 7.72686684e-06])
2836: (4)          Plot the window and the frequency response:
2837: (4)          >>> from numpy.fft import fft, fftshift
2838: (4)          >>> window = np.kaiser(51, 14)
2839: (4)          >>> plt.plot(window)
2840: (4)          [<matplotlib.lines.Line2D object at 0x...>]
2841: (4)          >>> plt.title("Kaiser window")
2842: (4)          Text(0.5, 1.0, 'Kaiser window')
2843: (4)          >>> plt.ylabel("Amplitude")
2844: (4)          Text(0, 0.5, 'Amplitude')
2845: (4)          >>> plt.xlabel("Sample")
2846: (4)          Text(0.5, 0, 'Sample')
2847: (4)          >>> plt.show()
2848: (4)          >>> plt.figure()
2849: (4)          <Figure size 640x480 with 0 Axes>
2850: (4)          >>> A = fft(window, 2048) / 25.5
2851: (4)          >>> mag = np.abs(fftshift(A))
2852: (4)          >>> freq = np.linspace(-0.5, 0.5, len(A))
2853: (4)          >>> response = 20 * np.log10(mag)
2854: (4)          >>> response = np.clip(response, -100, 100)
2855: (4)          >>> plt.plot(freq, response)
2856: (4)          [<matplotlib.lines.Line2D object at 0x...>]
2857: (4)          >>> plt.title("Frequency response of Kaiser window")
2858: (4)          Text(0.5, 1.0, 'Frequency response of Kaiser window')
2859: (4)          >>> plt.ylabel("Magnitude [dB]")
2860: (4)          Text(0, 0.5, 'Magnitude [dB]')
2861: (4)          >>> plt.xlabel("Normalized frequency [cycles per sample]")
2862: (4)          Text(0.5, 0, 'Normalized frequency [cycles per sample]')
2863: (4)          >>> plt.axis('tight')
2864: (4)          (-0.5, 0.5, -100.0, ...) # may vary
2865: (4)          >>> plt.show()
2866: (4)          """
2867: (4)          values = np.array([0.0, M, beta])
2868: (4)          M = values[1]
2869: (4)          beta = values[2]
2870: (4)          if M == 1:
2871: (8)              return np.ones(1, dtype=values.dtype)
2872: (4)          n = arange(0, M)
2873: (4)          alpha = (M-1)/2.0

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2874: (4)             return i0(beta * sqrt(1-((n-alpha)/alpha)**2.0))/i0(beta)
2875: (0)             def _sinc_dispatcher(x):
2876: (4)                 return (x,)
2877: (0)             @array_function_dispatch(_sinc_dispatcher)
2878: (0)             def sinc(x):
2879: (4)                 """
2880: (4)                 Return the normalized sinc function.
2881: (4)                 The sinc function is equal to :math:`\sin(\pi x)/(\pi x)` for any argument
2882: (4)                 :math:`x \neq 0` . ``sinc(0)`` takes the limit value 1, making ``sinc`` not
2883: (4)                 only everywhere continuous but also infinitely differentiable.
2884: (4)                 .. note::
2885: (8)                     Note the normalization factor of ``pi`` used in the definition.
2886: (8)                     This is the most commonly used definition in signal processing.
2887: (8)                     Use ``sinc(x / np.pi)`` to obtain the unnormalized sinc function
2888: (8)                     :math:`\sin(x)/x` that is more common in mathematics.
2889: (4)             Parameters
2890: (4)             -----
2891: (4)             x : ndarray
2892: (8)                 Array (possibly multi-dimensional) of values for which to calculate
2893: (8)                 ``sinc(x)``.
2894: (4)             Returns
2895: (4)             -----
2896: (4)             out : ndarray
2897: (8)                 ``sinc(x)``, which has the same shape as the input.
2898: (4)             Notes
2899: (4)             -----
2900: (4)                 The name sinc is short for "sine cardinal" or "sinus cardinalis".
2901: (4)                 The sinc function is used in various signal processing applications,
2902: (4)                 including in anti-aliasing, in the construction of a Lanczos resampling
2903: (4)                 filter, and in interpolation.
2904: (4)                 For bandlimited interpolation of discrete-time signals, the ideal
2905: (4)                 interpolation kernel is proportional to the sinc function.
2906: (4)             References
2907: (4)             -----
2908: (4)                 .. [1] Weisstein, Eric W. "Sinc Function." From MathWorld--A Wolfram Web
2909: (11)                   Resource. http://mathworld.wolfram.com/SincFunction.html
2910: (4)                 .. [2] Wikipedia, "Sinc function",
2911: (11)                   https://en.wikipedia.org/wiki/Sinc\_function
2912: (4)             Examples
2913: (4)             -----
2914: (4)                 >>> import matplotlib.pyplot as plt
2915: (4)                 >>> x = np.linspace(-4, 4, 41)
2916: (4)                 >>> np.sinc(x)
2917: (5)                 array([-3.89804309e-17, -4.92362781e-02, -8.40918587e-02, # may vary
2918: (12)                   -8.90384387e-02, -5.84680802e-02, 3.89804309e-17,
2919: (12)                   6.68206631e-02, 1.16434881e-01, 1.26137788e-01,
2920: (12)                   8.50444803e-02, -3.89804309e-17, -1.03943254e-01,
2921: (12)                   -1.89206682e-01, -2.16236208e-01, -1.55914881e-01,
2922: (12)                   3.89804309e-17, 2.33872321e-01, 5.04551152e-01,
2923: (12)                   7.56826729e-01, 9.35489284e-01, 1.00000000e+00,
2924: (12)                   9.35489284e-01, 7.56826729e-01, 5.04551152e-01,
2925: (12)                   2.33872321e-01, 3.89804309e-17, -1.55914881e-01,
2926: (11)                   -2.16236208e-01, -1.89206682e-01, -1.03943254e-01,
2927: (11)                   -3.89804309e-17, 8.50444803e-02, 1.26137788e-01,
2928: (12)                   1.16434881e-01, 6.68206631e-02, 3.89804309e-17,
2929: (12)                   -5.84680802e-02, -8.90384387e-02, -8.40918587e-02,
2930: (12)                   -4.92362781e-02, -3.89804309e-17])
2931: (4)                 >>> plt.plot(x, np.sinc(x))
2932: (4)                 [

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2943: (4)             return sin(y)/y
2944: (0)             def _msort_dispatcher(a):
2945: (4)                 return (a,)
2946: (0)             @array_function_dispatch(_msort_dispatcher)
2947: (0)             def msort(a):
2948: (4)                 """
2949: (4)                     Return a copy of an array sorted along the first axis.
2950: (4)                     .. deprecated:: 1.24
2951: (7)                         msort is deprecated, use ``np.sort(a, axis=0)`` instead.
2952: (4)             Parameters
2953: (4)                 -----
2954: (4)                 a : array_like
2955: (8)                     Array to be sorted.
2956: (4)             Returns
2957: (4)                 -----
2958: (4)                 sorted_array : ndarray
2959: (8)                     Array of the same type and shape as `a`.
2960: (4)             See Also
2961: (4)                 -----
2962: (4)                 sort
2963: (4)             Notes
2964: (4)                 -----
2965: (4)                 ``np.msort(a)`` is equivalent to ``np.sort(a, axis=0)``.
2966: (4)             Examples
2967: (4)                 -----
2968: (4)                 >>> a = np.array([[1, 4], [3, 1]])
2969: (4)                 >>> np.msort(a) # sort along the first axis
2970: (4)                 array([[1, 1],
2971: (11)                     [3, 4]])
2972: (4)                 """
2973: (4)                 warnings.warn(
2974: (8)                     "'msort is deprecated, use np.sort(a, axis=0) instead'",
2975: (8)                     DeprecationWarning,
2976: (8)                     stacklevel=2,
2977: (4)                 )
2978: (4)                 b = array(a, subok=True, copy=True)
2979: (4)                 b.sort(0)
2980: (4)                 return b
2981: (0)             def _ureduce(a, func, keepdims=False, **kwargs):
2982: (4)                 """
2983: (4)                     Internal Function.
2984: (4)                     Call `func` with `a` as first argument swapping the axes to use extended
2985: (4)                     axis on functions that don't support it natively.
2986: (4)                     Returns result and a.shape with axis dims set to 1.
2987: (4)                     Parameters
2988: (4)                         -----
2989: (4)                         a : array_like
2990: (8)                             Input array or object that can be converted to an array.
2991: (4)                         func : callable
2992: (8)                             Reduction function capable of receiving a single axis argument.
2993: (8)                             It is called with `a` as first argument followed by `kwargs`.
2994: (4)                         kwargs : keyword arguments
2995: (8)                             additional keyword arguments to pass to `func`.
2996: (4)             Returns
2997: (4)                 -----
2998: (4)                 result : tuple
2999: (8)                     Result of func(a, **kwargs) and a.shape with axis dims set to 1
3000: (8)                     which can be used to reshape the result to the same shape a ufunc with
3001: (8)                     keepdims=True would produce.
3002: (4)                 """
3003: (4)                 a = np.asarray(a)
3004: (4)                 axis = kwargs.get('axis', None)
3005: (4)                 out = kwargs.get('out', None)
3006: (4)                 if keepdims is np._NoValue:
3007: (8)                     keepdims = False
3008: (4)                 nd = a.ndim
3009: (4)                 if axis is not None:
3010: (8)                     axis = _nx.normalize_axis_tuple(axis, nd)
3011: (8)                     if keepdims:

```

```

3012: (12)             if out is not None:
3013: (16)                 index_out = tuple(
3014: (20)                     0 if i in axis else slice(None) for i in range(nd))
3015: (16)                     kkwargs['out'] = out[Ellipsis, ] + index_out]
3016: (8)             if len(axis) == 1:
3017: (12)                 kkwargs['axis'] = axis[0]
3018: (8)             else:
3019: (12)                 keep = set(range(nd)) - set(axis)
3020: (12)                 nkeep = len(keep)
3021: (12)                 for i, s in enumerate(sorted(keep)):
3022: (16)                     a = a.swapaxes(i, s)
3023: (12)                     a = a.reshape(a.shape[:nkeep] + (-1,))
3024: (12)                     kkwargs['axis'] = -1
3025: (4)             else:
3026: (8)                 if keepdims:
3027: (12)                     if out is not None:
3028: (16)                         index_out = (0, ) * nd
3029: (16)                         kkwargs['out'] = out[Ellipsis, ] + index_out]
3030: (4)             r = func(a, **kwargs)
3031: (4)             if out is not None:
3032: (8)                 return out
3033: (4)             if keepdims:
3034: (8)                 if axis is None:
3035: (12)                     index_r = (np.newaxis, ) * nd
3036: (8)                 else:
3037: (12)                     index_r = tuple(
3038: (16)                         np.newaxis if i in axis else slice(None)
3039: (16)                         for i in range(nd))
3040: (8)                     r = r[Ellipsis, ] + index_r]
3041: (4)             return r
3042: (0)             def _median_dispatcher(
3043: (8)                 a, axis=None, out=None, overwrite_input=None, keepdims=None):
3044: (4)                 return (a, out)
3045: (0)             @array_function_dispatch(_median_dispatcher)
3046: (0)             def median(a, axis=None, out=None, overwrite_input=False, keepdims=False):
3047: (4)                 """
3048: (4)                 Compute the median along the specified axis.
3049: (4)                 Returns the median of the array elements.
3050: (4)                 Parameters
3051: (4)                 -----
3052: (4)                 a : array_like
3053: (8)                     Input array or object that can be converted to an array.
3054: (4)                 axis : {int, sequence of int, None}, optional
3055: (8)                     Axis or axes along which the medians are computed. The default
3056: (8)                     is to compute the median along a flattened version of the array.
3057: (8)                     A sequence of axes is supported since version 1.9.0.
3058: (4)                 out : ndarray, optional
3059: (8)                     Alternative output array in which to place the result. It must
3060: (8)                     have the same shape and buffer length as the expected output,
3061: (8)                     but the type (of the output) will be cast if necessary.
3062: (4)                 overwrite_input : bool, optional
3063: (7)                     If True, then allow use of memory of input array `a` for
3064: (7)                     calculations. The input array will be modified by the call to
3065: (7)                     `median`. This will save memory when you do not need to preserve
3066: (7)                     the contents of the input array. Treat the input as undefined,
3067: (7)                     but it will probably be fully or partially sorted. Default is
3068: (7)                     False. If `overwrite_input` is ``True`` and `a` is not already an
3069: (7)                     `ndarray`, an error will be raised.
3070: (4)                 keepdims : bool, optional
3071: (8)                     If this is set to True, the axes which are reduced are left
3072: (8)                     in the result as dimensions with size one. With this option,
3073: (8)                     the result will broadcast correctly against the original `arr`.
3074: (8)                     .. versionadded:: 1.9.0
3075: (4)             Returns
3076: (4)             -----
3077: (4)             median : ndarray
3078: (8)                 A new array holding the result. If the input contains integers
3079: (8)                 or floats smaller than ``float64``, then the output data-type is
3080: (8)                 ``np.float64``. Otherwise, the data-type of the output is the

```

```

3081: (8)           same as that of the input. If `out` is specified, that array is
3082: (8)           returned instead.
3083: (4)           See Also
3084: (4)           -----
3085: (4)           mean, percentile
3086: (4)           Notes
3087: (4)           -----
3088: (4)           Given a vector ``V`` of length ``N``, the median of ``V`` is the
3089: (4)           middle value of a sorted copy of ``V``, ``V_sorted`` - i
3090: (4)           e., ``V_sorted[(N-1)/2]``, when ``N`` is odd, and the average of the
3091: (4)           two middle values of ``V_sorted`` when ``N`` is even.
3092: (4)           Examples
3093: (4)           -----
3094: (4)           >>> a = np.array([[10, 7, 4], [3, 2, 1]])
3095: (4)           >>> a
3096: (4)           array([[10, 7, 4],
3097: (11)             [ 3, 2, 1]])
3098: (4)           >>> np.median(a)
3099: (4)           3.5
3100: (4)           >>> np.median(a, axis=0)
3101: (4)           array([6.5, 4.5, 2.5])
3102: (4)           >>> np.median(a, axis=1)
3103: (4)           array([7., 2.])
3104: (4)           >>> m = np.median(a, axis=0)
3105: (4)           >>> out = np.zeros_like(m)
3106: (4)           >>> np.median(a, axis=0, out=out)
3107: (4)           array([6.5, 4.5, 2.5])
3108: (4)           >>> m
3109: (4)           array([6.5, 4.5, 2.5])
3110: (4)           >>> b = a.copy()
3111: (4)           >>> np.median(b, axis=1, overwrite_input=True)
3112: (4)           array([7., 2.])
3113: (4)           >>> assert not np.all(a==b)
3114: (4)           >>> b = a.copy()
3115: (4)           >>> np.median(b, axis=None, overwrite_input=True)
3116: (4)           3.5
3117: (4)           >>> assert not np.all(a==b)
3118: (4)           """
3119: (4)           return _ureduce(a, func=_median, keepdims=keepdims, axis=axis, out=out,
3120: (20)             overwrite_input=overwrite_input)
3121: (0)           def _median(a, axis=None, out=None, overwrite_input=False):
3122: (4)             a = np.asarray(a)
3123: (4)             if axis is None:
3124: (8)               sz = a.size
3125: (4)             else:
3126: (8)               sz = a.shape[axis]
3127: (4)             if sz % 2 == 0:
3128: (8)               szh = sz // 2
3129: (8)               kth = [szh - 1, szh]
3130: (4)             else:
3131: (8)               kth = [(sz - 1) // 2]
3132: (4)             supports_nans = np.issubdtype(a.dtype, np.inexact) or a.dtype.kind in 'Mm'
3133: (4)             if supports_nans:
3134: (8)               kth.append(-1)
3135: (4)             if overwrite_input:
3136: (8)               if axis is None:
3137: (12)                 part = a.ravel()
3138: (12)                 part.partition(kth)
3139: (8)               else:
3140: (12)                 a.partition(kth, axis=axis)
3141: (12)                 part = a
3142: (4)             else:
3143: (8)               part = partition(a, kth, axis=axis)
3144: (4)             if part.shape == ():
3145: (8)               return part.item()
3146: (4)             if axis is None:
3147: (8)               axis = 0
3148: (4)             indexer = [slice(None)] * part.ndim
3149: (4)             index = part.shape[axis] // 2

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3150: (4)
3151: (8)
3152: (4)
3153: (8)
3154: (4)
3155: (4)
3156: (4)
3157: (8)
3158: (4)
3159: (0)
3160: (27)
3161: (4)
3162: (0)
3163: (0)
3164: (15)
3165: (15)
3166: (15)
3167: (15)
3168: (15)
3169: (15)
3170: (15)
3171: (15)
3172: (4)
3173: (4)
3174: (4)
3175: (4)
3176: (4)
3177: (4)
3178: (8)
3179: (4)
3180: (8)
3181: (8)
3182: (4)
3183: (8)
3184: (8)
3185: (8)
3186: (8)
3187: (12)
3188: (4)
3189: (8)
3190: (8)
3191: (8)
3192: (4)
3193: (8)
3194: (8)
3195: (8)
3196: (4)
3197: (8)
3198: (8)
3199: (8)
3200: (8)
3201: (8)
3202: (8)
3203: (8)
3204: (8)
3205: (8)
3206: (8)
3207: (8)
3208: (8)
3209: (8)
3210: (8)
3211: (8)
option:
3212: (8)
3213: (8)
3214: (8)
3215: (8)
3216: (8)
3217: (12)

    if part.shape[axis] % 2 == 1:
        indexer[axis] = slice(index, index+1)
    else:
        indexer[axis] = slice(index-1, index+1)
    indexer = tuple(indexer)
    rout = mean(part[indexer], axis=axis, out=out)
    if supports_nans and sz > 0:
        rout = np.lib.utils._median_nancheck(part, rout, axis)
    return rout
def _percentile_dispatcher(a, q, axis=None, out=None, overwrite_input=None,
                           method=None, keepdims=None, *, interpolation=None):
    return (a, q, out)
@array_function_dispatch(_percentile_dispatcher)
def percentile(a,
               q,
               axis=None,
               out=None,
               overwrite_input=False,
               method="linear",
               keepdims=False,
               *,
               interpolation=None):
    """
    Compute the q-th percentile of the data along the specified axis.
    Returns the q-th percentile(s) of the array elements.
    Parameters
    -----
    a : array_like of real numbers
        Input array or object that can be converted to an array.
    q : array_like of float
        Percentage or sequence of percentages for the percentiles to compute.
        Values must be between 0 and 100 inclusive.
    axis : {int, tuple of int, None}, optional
        Axis or axes along which the percentiles are computed. The
        default is to compute the percentile(s) along a flattened
        version of the array.
        .. versionchanged:: 1.9.0
            A tuple of axes is supported
    out : ndarray, optional
        Alternative output array in which to place the result. It must
        have the same shape and buffer length as the expected output,
        but the type (of the output) will be cast if necessary.
    overwrite_input : bool, optional
        If True, then allow the input array `a` to be modified by intermediate
        calculations, to save memory. In this case, the contents of the input
        `a` after this function completes is undefined.
    method : str, optional
        This parameter specifies the method to use for estimating the
        percentile. There are many different methods, some unique to NumPy.
        See the notes for explanation. The options sorted by their R type
        as summarized in the H&F paper [1]_ are:
        1. 'inverted_cdf'
        2. 'averaged_inverted_cdf'
        3. 'closest_observation'
        4. 'interpolated_inverted_cdf'
        5. 'hazen'
        6. 'weibull'
        7. 'linear' (default)
        8. 'median_unbiased'
        9. 'normal_unbiased'
        The first three methods are discontinuous. NumPy further defines the
        following discontinuous variations of the default 'linear' (7.).
        *
        * 'lower'
        * 'higher',
        * 'midpoint'
        * 'nearest'
        .. versionchanged:: 1.22.0
            This argument was previously called "interpolation" and only

```

```

3218: (12) offered the "linear" default and last four options.
3219: (4) keepdims : bool, optional
3220: (8) If this is set to True, the axes which are reduced are left in
3221: (8) the result as dimensions with size one. With this option, the
3222: (8) result will broadcast correctly against the original array `a`.
3223: (8) .. versionadded:: 1.9.0
3224: (4) interpolation : str, optional
3225: (8) Deprecated name for the method keyword argument.
3226: (8) .. deprecated:: 1.22.0
3227: (4) Returns
3228: (4)
3229: (4) -----
3230: (8) percentile : scalar or ndarray
3231: (8) If `q` is a single percentile and `axis=None`, then the result
3232: (8) is a scalar. If multiple percentiles are given, first axis of
3233: (8) the result corresponds to the percentiles. The other axes are
3234: (8) the axes that remain after the reduction of `a`. If the input
3235: (8) contains integers or floats smaller than ``float64``, the output
3236: (8) data-type is ``float64``. Otherwise, the output data-type is the
3237: (8) same as that of the input. If `out` is specified, that array is
3238: (4) returned instead.
3239: (4) See Also
3240: (4) -----
3241: (4) mean
3242: (4) median : equivalent to ``percentile(..., 50)``
3243: (4) nanpercentile
3244: (4) quantile : equivalent to percentile, except q in the range [0, 1].
3245: (4) Notes
3246: (4) -----
3247: (4) Given a vector ``V`` of length ``n``, the q-th percentile of ``V`` is
3248: (4) the value ``q/100`` of the way from the minimum to the maximum in a
3249: (4) sorted copy of ``V``. The values and distances of the two nearest
3250: (4) neighbors as well as the `method` parameter will determine the
3251: (4) percentile if the normalized ranking does not match the location of
3252: (4) ``q`` exactly. This function is the same as the median if ``q=50``, the
3253: (4) same as the minimum if ``q=0`` and the same as the maximum if
3254: (4) ``q=100``.
3255: (4) The optional `method` parameter specifies the method to use when the
3256: (4) desired percentile lies between two indexes ``i`` and ``j = i + 1``.
3257: (4) In that case, we first determine ``i + g``, a virtual index that lies
3258: (4) between ``i`` and ``j``, where ``i`` is the floor and ``g`` is the
3259: (4) fractional part of the index. The final result is, then, an interpolation
3260: (4) of ``a[i]`` and ``a[j]`` based on ``g``. During the computation of ``g``,
3261: (4) ``i`` and ``j`` are modified using correction constants ``alpha`` and
3262: (4) ``beta`` whose choices depend on the ``method`` used. Finally, note that
3263: (4) since Python uses 0-based indexing, the code subtracts another 1 from the
3264: (4) index internally.
3265: (4) The following formula determines the virtual index ``i + g``, the location
3266: (4) of the percentile in the sorted sample:
3267: (8) .. math::
3268: (8)     i + g = (q / 100) * (n - alpha - beta + 1) + alpha
3269: (4) The different methods then work as follows
3270: (8) inverted_cdf:
3271: (8)     method 1 of H&F [1]_.
3272: (8)     This method gives discontinuous results:
3273: (8)     * if g > 0 ; then take j
3274: (8)     * if g = 0 ; then take i
3275: (8) averaged_inverted_cdf:
3276: (8)     method 2 of H&F [1]_.
3277: (8)     This method give discontinuous results:
3278: (8)     * if g > 0 ; then take j
3279: (4)     * if g = 0 ; then average between bounds
3280: (8) closest_observation:
3281: (8)     method 3 of H&F [1]_.
3282: (8)     This method give discontinuous results:
3283: (8)     * if g > 0 ; then take j
3284: (8)     * if g = 0 and index is odd ; then take j
3285: (4)     * if g = 0 and index is even ; then take i
3286: (8) interpolated_inverted_cdf:
3287: (8)     method 4 of H&F [1]_.

```

```

3287: (8)           This method give continuous results using:
3288: (8)           * alpha = 0
3289: (8)           * beta = 1
3290: (4)          hazen:
3291: (8)            method 5 of H&F [1]..
3292: (8)            This method give continuous results using:
3293: (8)            * alpha = 1/2
3294: (8)            * beta = 1/2
3295: (4)          weibull:
3296: (8)            method 6 of H&F [1]..
3297: (8)            This method give continuous results using:
3298: (8)            * alpha = 0
3299: (8)            * beta = 0
3300: (4)          linear:
3301: (8)            method 7 of H&F [1]..
3302: (8)            This method give continuous results using:
3303: (8)            * alpha = 1
3304: (8)            * beta = 1
3305: (4)          median_unbiased:
3306: (8)            method 8 of H&F [1]..
3307: (8)            This method is probably the best method if the sample
3308: (8)            distribution function is unknown (see reference).
3309: (8)            This method give continuous results using:
3310: (8)            * alpha = 1/3
3311: (8)            * beta = 1/3
3312: (4)          normal_unbiased:
3313: (8)            method 9 of H&F [1]..
3314: (8)            This method is probably the best method if the sample
3315: (8)            distribution function is known to be normal.
3316: (8)            This method give continuous results using:
3317: (8)            * alpha = 3/8
3318: (8)            * beta = 3/8
3319: (4)          lower:
3320: (8)            NumPy method kept for backwards compatibility.
3321: (8)            Takes ``i`` as the interpolation point.
3322: (4)          higher:
3323: (8)            NumPy method kept for backwards compatibility.
3324: (8)            Takes ``j`` as the interpolation point.
3325: (4)          nearest:
3326: (8)            NumPy method kept for backwards compatibility.
3327: (8)            Takes ``i`` or ``j``, whichever is nearest.
3328: (4)          midpoint:
3329: (8)            NumPy method kept for backwards compatibility.
3330: (8)            Uses ``(i + j) / 2``.
3331: (4)          Examples
3332: (4)
3333: (4)          -----
3334: (4)          >>> a = np.array([[10, 7, 4], [3, 2, 1]])
3335: (4)          >>> a
3336: (11)         array([[10, 7, 4],
3337: (4)                  [ 3, 2, 1]])
3338: (4)          >>> np.percentile(a, 50)
3339: (4)          3.5
3340: (4)          >>> np.percentile(a, 50, axis=0)
3341: (4)          array([6.5, 4.5, 2.5])
3342: (4)          >>> np.percentile(a, 50, axis=1)
3343: (4)          array([7., 2.])
3344: (4)          >>> np.percentile(a, 50, axis=1, keepdims=True)
3345: (11)         array([[7.],
3346: (4)                  [2.]])
3347: (4)          >>> m = np.percentile(a, 50, axis=0)
3348: (4)          >>> out = np.zeros_like(m)
3349: (4)          >>> np.percentile(a, 50, axis=0, out=out)
3350: (4)          array([6.5, 4.5, 2.5])
3351: (4)          >>> m
3352: (4)          array([6.5, 4.5, 2.5])
3353: (4)          >>> b = a.copy()
3354: (4)          >>> np.percentile(b, 50, axis=1, overwrite_input=True)
3355: (4)          array([7., 2.])
3356: (4)          >>> assert not np.all(a == b)

```

```

3356: (4)          The different methods can be visualized graphically:
3357: (4)          ..
3358: (8)          plot::
3359: (8)          import matplotlib.pyplot as plt
3360: (8)          a = np.arange(4)
3361: (8)          p = np.linspace(0, 100, 6001)
3362: (8)          ax = plt.gca()
3363: (12)         lines = [
3364: (12)             ('linear', '-', 'C0'),
3365: (12)             ('inverted_cdf', ':', 'C1'),
3366: (12)             ('averaged_inverted_cdf', '-.', 'C1'),
3367: (12)             ('closest_observation', ':', 'C2'),
3368: (12)             ('interpolated_inverted_cdf', '--', 'C1'),
3369: (12)             ('hazen', '--', 'C3'),
3370: (12)             ('weibull', '-.', 'C4'),
3371: (12)             ('median_unbiased', '--', 'C5'),
3372: (12)             ('normal_unbiased', '-.', 'C6'),
3373: (8)         ]
3374: (12)         for method, style, color in lines:
3375: (16)             ax.plot(
3376: (16)                 p, np.percentile(a, p, method=method),
3377: (8)                 label=method, linestyle=style, color=color)
3378: (12)         ax.set(
3379: (12)             title='Percentiles for different methods and data: ' + str(a),
3380: (12)             xlabel='Percentile',
3381: (12)             ylabel='Estimated percentile value',
3382: (8)             yticks=a)
3383: (8)         ax.legend(bbox_to_anchor=(1.03, 1))
3384: (8)         plt.tight_layout()
3385: (8)         plt.show()
3386: (4)         References
3387: (4)         -----
3388: (7)         .. [1] R. J. Hyndman and Y. Fan,
3389: (7)             "Sample quantiles in statistical packages,"
3390: (7)             The American Statistician, 50(4), pp. 361-365, 1996
3391: (4)         """
3392: (8)         if interpolation is not None:
3393: (12)             method = _check_interpolation_as_method(
3394: (4)                 method, interpolation, "percentile")
3395: (4)             a = np.asarray(a)
3396: (8)             if a.dtype.kind == "c":
3397: (4)                 raise TypeError("a must be an array of real numbers")
3398: (4)             q = np.true_divide(q, 100)
3399: (4)             q = asanyarray(q) # undo any decay that the ufunc performed (see gh-
13105)
3400: (8)             if not _quantile_is_valid(q):
3401: (4)                 raise ValueError("Percentiles must be in the range [0, 100]")
3402: (8)             return _quantile_unchecked(
3403: (0)                 a, q, axis, out, overwrite_input, method, keepdims)
3404: (25)         def _quantile_dispatcher(a, q, axis=None, out=None, overwrite_input=None,
3405: (4)                         method=None, keepdims=None, *, interpolation=None):
3406: (0)             return (a, q, out)
3407: (0)         @array_function_dispatch(_quantile_dispatcher)
3408: (13)         def quantile(a,
3409: (13)             q,
3410: (13)             axis=None,
3411: (13)             out=None,
3412: (13)             overwrite_input=False,
3413: (13)             method="linear",
3414: (13)             keepdims=False,
3415: (13)             *,
3416: (4)             interpolation=None):
3417: (4)             """
3418: (4)             Compute the q-th quantile of the data along the specified axis.
3419: (4)             ..
3420: (4)             versionadded:: 1.15.0
3421: (4)             Parameters
3422: (4)             -----
3423: (4)             a : array_like of real numbers
3424: (4)                 Input array or object that can be converted to an array.
3425: (4)             q : array_like of float

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3424: (8)                               Probability or sequence of probabilities for the quantiles to compute.
3425: (8)                               Values must be between 0 and 1 inclusive.
3426: (4)                               axis : {int, tuple of int, None}, optional
3427: (8)                               Axis or axes along which the quantiles are computed. The default is
3428: (8)                               to compute the quantile(s) along a flattened version of the array.
3429: (4)                               out : ndarray, optional
3430: (8)                               Alternative output array in which to place the result. It must have
3431: (8)                               the same shape and buffer length as the expected output, but the
3432: (8)                               type (of the output) will be cast if necessary.
3433: (4)                               overwrite_input : bool, optional
3434: (8)                               If True, then allow the input array `a` to be modified by
3435: (8)                               intermediate calculations, to save memory. In this case, the
3436: (8)                               contents of the input `a` after this function completes is
3437: (8)                               undefined.
3438: (4)                               method : str, optional
3439: (8)                               This parameter specifies the method to use for estimating the
3440: (8)                               quantile. There are many different methods, some unique to NumPy.
3441: (8)                               See the notes for explanation. The options sorted by their R type
3442: (8)                               as summarized in the H&F paper [1]_ are:
3443: (8)                               1. 'inverted_cdf'
3444: (8)                               2. 'averaged_inverted_cdf'
3445: (8)                               3. 'closest_observation'
3446: (8)                               4. 'interpolated_inverted_cdf'
3447: (8)                               5. 'hazen'
3448: (8)                               6. 'weibull'
3449: (8)                               7. 'linear' (default)
3450: (8)                               8. 'median_unbiased'
3451: (8)                               9. 'normal_unbiased'
3452: (8)                               The first three methods are discontinuous. NumPy further defines the
3453: (8)                               following discontinuous variations of the default 'linear' (7.)
option:
3454: (8)                               * 'lower'
3455: (8)                               * 'higher',
3456: (8)                               * 'midpoint'
3457: (8)                               * 'nearest'
3458: (8)                               .. versionchanged:: 1.22.0
3459: (12)                              This argument was previously called "interpolation" and only
3460: (12)                              offered the "linear" default and last four options.
keepdims : bool, optional
3461: (4)                               If this is set to True, the axes which are reduced are left in
3462: (8)                               the result as dimensions with size one. With this option, the
3463: (8)                               result will broadcast correctly against the original array `a`.
3464: (8)                               interpolation : str, optional
3465: (4)                               Deprecated name for the method keyword argument.
3466: (8)                               .. deprecated:: 1.22.0
3467: (8)                               Returns
3468: (4)                               -----
3469: (4)                               quantile : scalar or ndarray
3470: (4)                               If `q` is a single probability and `axis=None`, then the result
3471: (8)                               is a scalar. If multiple probabilities levels are given, first axis of
3472: (8)                               the result corresponds to the quantiles. The other axes are
3473: (8)                               the axes that remain after the reduction of `a`. If the input
3474: (8)                               contains integers or floats smaller than ``float64``, the output
3475: (8)                               data-type is ``float64``. Otherwise, the output data-type is the
3476: (8)                               same as that of the input. If `out` is specified, that array is
3477: (8)                               returned instead.
3478: (8)                               See Also
3479: (4)                               -----
3480: (4)                               mean
3481: (4)                               percentile : equivalent to quantile, but with q in the range [0, 100].
3482: (4)                               median : equivalent to ``quantile(..., 0.5)``
3483: (4)                               nanquantile
3484: (4)                               Notes
3485: (4)                               -----
3486: (4)                               Given a vector ``V`` of length ``n``, the q-th quantile of ``V`` is
3487: (4)                               the value ``q`` of the way from the minimum to the maximum in a
3488: (4)                               sorted copy of ``V``. The values and distances of the two nearest
3489: (4)                               neighbors as well as the `method` parameter will determine the
3490: (4)                               quantile if the normalized ranking does not match the location of
3491: (4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3492: (4)           ``q`` exactly. This function is the same as the median if ``q=0.5``, the
3493: (4)           same as the minimum if ``q=0.0`` and the same as the maximum if
3494: (4)           ``q=1.0``.
3495: (4)           The optional `method` parameter specifies the method to use when the
3496: (4)           desired quantile lies between two indexes ``i`` and ``j = i + 1``.
3497: (4)           In that case, we first determine ``i + g``, a virtual index that lies
3498: (4)           between ``i`` and ``j``, where ``i`` is the floor and ``g`` is the
3499: (4)           fractional part of the index. The final result is, then, an interpolation
3500: (4)           of ``a[i]`` and ``a[j]`` based on ``g``. During the computation of ``g``,
3501: (4)           ``i`` and ``j`` are modified using correction constants ``alpha`` and
3502: (4)           ``beta`` whose choices depend on the ``method`` used. Finally, note that
3503: (4)           since Python uses 0-based indexing, the code subtracts another 1 from the
3504: (4)           index internally.
3505: (4)           The following formula determines the virtual index ``i + g``, the location
3506: (4)           of the quantile in the sorted sample:
3507: (4)           .. math::
3508: (8)              $i + g = q * (n - \alpha - \beta + 1) + \alpha$ 
3509: (4)           The different methods then work as follows
3510: (4)           inverted_cdf:
3511: (8)             method 1 of H&F [1]..
3512: (8)             This method gives discontinuous results:
3513: (8)               * if  $g > 0$ ; then take  $j$ 
3514: (8)               * if  $g = 0$ ; then take  $i$ 
3515: (4)           averaged_inverted_cdf:
3516: (8)             method 2 of H&F [1]..
3517: (8)             This method gives discontinuous results:
3518: (8)               * if  $g > 0$ ; then take  $j$ 
3519: (8)               * if  $g = 0$ ; then average between bounds
3520: (4)           closest_observation:
3521: (8)             method 3 of H&F [1]..
3522: (8)             This method gives discontinuous results:
3523: (8)               * if  $g > 0$ ; then take  $j$ 
3524: (8)               * if  $g = 0$  and index is odd; then take  $j$ 
3525: (8)               * if  $g = 0$  and index is even; then take  $i$ 
3526: (4)           interpolated_inverted_cdf:
3527: (8)             method 4 of H&F [1]..
3528: (8)             This method gives continuous results using:
3529: (8)               *  $\alpha = 0$ 
3530: (8)               *  $\beta = 1$ 
3531: (4)           hazen:
3532: (8)             method 5 of H&F [1]..
3533: (8)             This method gives continuous results using:
3534: (8)               *  $\alpha = 1/2$ 
3535: (8)               *  $\beta = 1/2$ 
3536: (4)           weibull:
3537: (8)             method 6 of H&F [1]..
3538: (8)             This method gives continuous results using:
3539: (8)               *  $\alpha = 0$ 
3540: (8)               *  $\beta = 0$ 
3541: (4)           linear:
3542: (8)             method 7 of H&F [1]..
3543: (8)             This method gives continuous results using:
3544: (8)               *  $\alpha = 1$ 
3545: (8)               *  $\beta = 1$ 
3546: (4)           median_unbiased:
3547: (8)             method 8 of H&F [1]..
3548: (8)             This method is probably the best method if the sample
3549: (8)             distribution function is unknown (see reference).
3550: (8)             This method gives continuous results using:
3551: (8)               *  $\alpha = 1/3$ 
3552: (8)               *  $\beta = 1/3$ 
3553: (4)           normal_unbiased:
3554: (8)             method 9 of H&F [1]..
3555: (8)             This method is probably the best method if the sample
3556: (8)             distribution function is known to be normal.
3557: (8)             This method gives continuous results using:
3558: (8)               *  $\alpha = 3/8$ 
3559: (8)               *  $\beta = 3/8$ 
3560: (4)           lower:

```

```

3561: (8)           NumPy method kept for backwards compatibility.
3562: (8)           Takes ``i`` as the interpolation point.
3563: (4)           higher:
3564: (8)             NumPy method kept for backwards compatibility.
3565: (8)             Takes ``j`` as the interpolation point.
3566: (4)           nearest:
3567: (8)             NumPy method kept for backwards compatibility.
3568: (8)             Takes ``i`` or ``j``, whichever is nearest.
3569: (4)           midpoint:
3570: (8)             NumPy method kept for backwards compatibility.
3571: (8)             Uses `` $(i + j) / 2$ ``.
3572: (4)           Examples
3573: (4)           -----
3574: (4)             >>> a = np.array([[10, 7, 4], [3, 2, 1]])
3575: (4)             >>> a
3576: (4)             array([[10, 7, 4],
3577: (11)               [3, 2, 1]])
3578: (4)             >>> np.quantile(a, 0.5)
3579: (4)             3.5
3580: (4)             >>> np.quantile(a, 0.5, axis=0)
3581: (4)             array([6.5, 4.5, 2.5])
3582: (4)             >>> np.quantile(a, 0.5, axis=1)
3583: (4)             array([7., 2.])
3584: (4)             >>> np.quantile(a, 0.5, axis=1, keepdims=True)
3585: (4)             array([[7.],
3586: (11)               [2.]])
3587: (4)             >>> m = np.quantile(a, 0.5, axis=0)
3588: (4)             >>> out = np.zeros_like(m)
3589: (4)             >>> np.quantile(a, 0.5, axis=0, out=out)
3590: (4)             array([6.5, 4.5, 2.5])
3591: (4)             >>> m
3592: (4)             array([6.5, 4.5, 2.5])
3593: (4)             >>> b = a.copy()
3594: (4)             >>> np.quantile(b, 0.5, axis=1, overwrite_input=True)
3595: (4)             array([7., 2.])
3596: (4)             >>> assert not np.all(a == b)
3597: (4)             See also `numpy.percentile` for a visualization of most methods.
3598: (4)             References
3599: (4)             -----
3600: (4)             .. [1] R. J. Hyndman and Y. Fan,
3601: (7)               "Sample quantiles in statistical packages,"
3602: (7)               The American Statistician, 50(4), pp. 361-365, 1996
3603: (4)             """
3604: (4)             if interpolation is not None:
3605: (8)               method = _check_interpolation_as_method(
3606: (12)                 method, interpolation, "quantile")
3607: (4)             a = np.asarray(a)
3608: (4)             if a.dtype.kind == "c":
3609: (8)               raise TypeError("a must be an array of real numbers")
3610: (4)             q = np.asarray(q)
3611: (4)             if not _quantile_is_valid(q):
3612: (8)               raise ValueError("Quantiles must be in the range [0, 1]")
3613: (4)             return _quantile_unchecked(
3614: (8)               a, q, axis, out, overwrite_input, method, keepdims)
3615: (0)             def _quantile_unchecked(a,
3616: (24)                 q,
3617: (24)                   axis=None,
3618: (24)                   out=None,
3619: (24)                   overwrite_input=False,
3620: (24)                   method="linear",
3621: (24)                   keepdims=False):
3622: (4)               """Assumes that q is in [0, 1], and is an ndarray"""
3623: (4)               return _ureduce(a,
3624: (20)                 func=_quantile_ureduce_func,
3625: (20)                   q=q,
3626: (20)                   keepdims=keepdims,
3627: (20)                   axis=axis,
3628: (20)                   out=out,
3629: (20)                   overwrite_input=overwrite_input,

```

```

3630: (20)
3631: (0)
3632: (4)
3633: (8)
3634: (12)
3635: (16)
3636: (4)
3637: (8)
3638: (12)
3639: (4)
3640: (0)
3641: (4)
3642: (8)
3643: (8)
3644: (8)
3645: (8)
3646: (8)
3647: (8)
3648: (4)
3649: (8)
3650: (12)
3651: (12)
3652: (4)
3653: (0)
3654: (4)
3655: (4)
3656: (4)
3657: (4)
3658: (8)
3659: (4)
3660: (8)
3661: (4)
3662: (8)
3663: (4)
3664: (8)
3665: (4)
3666: (4)
3667: (4)
3668: (4)
3669: (4)
3670: (4)
3671: (4)
3672: (12)
3673: (4)
3674: (0)
3675: (4)
3676: (4)
3677: (4)
3678: (4)
3679: (8)
3680: (8)
3681: (4)
3682: (8)
3683: (4)
3684: (8)
3685: (8)
3686: (4)
modified
3687: (4)
3688: (4)
3689: (4)
3690: (4)
3691: (4)
3692: (0)
3693: (4)
3694: (4)
3695: (4)
3696: (4)
3697: (8)

        method=method)
def _quantile_is_valid(q):
    if q.ndim == 1 and q.size < 10:
        for i in range(q.size):
            if not (0.0 <= q[i] <= 1.0):
                return False
    else:
        if not (np.all(0 <= q) and np.all(q <= 1)):
            return False
    return True
def _check_interpolation_as_method(method, interpolation, fname):
    warnings.warn(
        f"the `interpolation` argument to {fname} was renamed to "
        "`method`, which has additional options.\n"
        "Users of the modes 'nearest', 'lower', 'higher', or "
        "'midpoint' are encouraged to review the method they used. "
        "(Deprecated NumPy 1.22)",
        DeprecationWarning, stacklevel=4)
if method != "linear":
    raise TypeError(
        "You shall not pass both `method` and `interpolation`!\n"
        "(`interpolation` is Deprecated in favor of `method`)")
return interpolation
def _compute_virtual_index(n, quantiles, alpha: float, beta: float):
    """
    Compute the floating point indexes of an array for the linear
    interpolation of quantiles.
    n : array_like
        The sample sizes.
    quantiles : array_like
        The quantiles values.
    alpha : float
        A constant used to correct the index computed.
    beta : float
        A constant used to correct the index computed.
    alpha and beta values depend on the chosen method
    (see quantile documentation)
    Reference:
    Hyndman&Fan paper "Sample Quantiles in Statistical Packages",
    DOI: 10.1080/00031305.1996.10473566
    """
    return n * quantiles + (
        alpha + quantiles * (1 - alpha - beta)
    ) - 1
def _get_gamma(virtual_indexes, previous_indexes, method):
    """
    Compute gamma (a.k.a 'm' or 'weight') for the linear interpolation
    of quantiles.
    virtual_indexes : array_like
        The indexes where the percentile is supposed to be found in the sorted
        sample.
    previous_indexes : array_like
        The floor values of virtual_indexes.
    interpolation : dict
        The interpolation method chosen, which may have a specific rule
        modifying gamma.
    gamma is usually the fractional part of virtual_indexes but can be
    modified
    by the interpolation method.
    """
    gamma = np.asarray(virtual_indexes - previous_indexes)
    gamma = method["fix_gamma"](gamma, virtual_indexes)
    return np.asarray(gamma)
def _lerp(a, b, t, out=None):
    """
    Compute the linear interpolation weighted by gamma on each point of
    two same shape array.
    a : array_like
        Left bound.

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

3698: (4)
3699: (8)
3700: (4)
3701: (8)
3702: (4)
3703: (8)
3704: (4)
3705: (4)
3706: (4)
3707: (4)
3708: (13)
3709: (4)
3710: (8)
3711: (4)
3712: (0)
3713: (4)
3714: (4)
3715: (4)
3716: (0)
3717: (4)
3718: (4)
3719: (4)
3720: (4)
3721: (26)
3722: (26)
3723: (26)
3724: (26)
3725: (4)
3726: (4)
3727: (0)
3728: (4)
3729: (4)
3730: (48)
3731: (0)
3732: (4)
3733: (4)
3734: (48)
3735: (0)
3736: (8)
3737: (8)
3738: (8)
3739: (8)
3740: (8)
3741: (8)
3742: (0)
3743: (4)
3744: (8)
3745: (4)
3746: (8)
3747: (12)
3748: (12)
3749: (8)
3750: (12)
3751: (4)
3752: (8)
3753: (12)
3754: (12)
3755: (8)
3756: (12)
3757: (4)
3758: (23)
3759: (23)
3760: (23)
3761: (23)
3762: (4)
3763: (0)
3764: (4)
3765: (4)
3766: (4)

    b : array_like
        Right bound.
    t : array_like
        The interpolation weight.
    out : array_like
        Output array.
    """
    diff_b_a = subtract(b, a)
    lerp_interpolation = asanyarray(add(a, diff_b_a * t, out=out))
    subtract(b, diff_b_a * (1 - t), out=lerp_interpolation, where=t >= 0.5,
              casting='unsafe', dtype=type(lerp_interpolation.dtype))
    if lerp_interpolation.ndim == 0 and out is None:
        lerp_interpolation = lerp_interpolation[()] # unpack 0d arrays
    return lerp_interpolation

def _get_gamma_mask(shape, default_value, conditioned_value, where):
    out = np.full(shape, default_value)
    np.copyto(out, conditioned_value, where=where, casting="unsafe")
    return out

def _discret_interpolation_to_boundaries(index, gamma_condition_fun):
    previous = np.floor(index)
    next = previous + 1
    gamma = index - previous
    res = _get_gamma_mask(shape=index.shape,
                          default_value=next,
                          conditioned_value=previous,
                          where=gamma_condition_fun(gamma, index)
                          ).astype(np.intp)
    res[res < 0] = 0
    return res

def _closest_observation(n, quantiles):
    gamma_fun = lambda gamma, index: (gamma == 0) & (np.floor(index) % 2 == 0)
    return _discret_interpolation_to_boundaries((n * quantiles) - 1 - 0.5,
                                                gamma_fun)

def _inverted_cdf(n, quantiles):
    gamma_fun = lambda gamma, _: (gamma == 0)
    return _discret_interpolation_to_boundaries((n * quantiles) - 1,
                                                gamma_fun)

def _quantile_ureduce_func(
        a: np.array,
        q: np.array,
        axis: int = None,
        out=None,
        overwrite_input: bool = False,
        method="linear",
    ) -> np.array:
    if q.ndim > 2:
        raise ValueError("q must be a scalar or 1d")
    if overwrite_input:
        if axis is None:
            axis = 0
            arr = a.ravel()
        else:
            arr = a
    else:
        if axis is None:
            axis = 0
            arr = a.flatten()
        else:
            arr = a.copy()
    result = _quantile(arr,
                       quantiles=q,
                       axis=axis,
                       method=method,
                       out=out)
    return result

def _get_indexes(arr, virtual_indexes, valid_values_count):
    """
    Get the valid indexes of arr neighbouring virtual_indexes.
    Note

```

```

3767: (4)             This is a companion function to linear interpolation of
3768: (4)             Quantiles
3769: (4)             Returns
3770: (4)             -----
3771: (4)             (previous_indexes, next_indexes): Tuple
3772: (8)               A Tuple of virtual_indexes neighbouring indexes
3773: (4)             """
3774: (4)             previous_indexes = np.asanyarray(np.floor(virtual_indexes))
3775: (4)             next_indexes = np.asanyarray(previous_indexes + 1)
3776: (4)             indexes_above_bounds = virtual_indexes >= valid_values_count - 1
3777: (4)             if indexes_above_bounds.any():
3778: (8)               previous_indexes[indexes_above_bounds] = -1
3779: (8)               next_indexes[indexes_above_bounds] = -1
3780: (4)             indexes_below_bounds = virtual_indexes < 0
3781: (4)             if indexes_below_bounds.any():
3782: (8)               previous_indexes[indexes_below_bounds] = 0
3783: (8)               next_indexes[indexes_below_bounds] = 0
3784: (4)             if np.issubdtype(arr.dtype, np.inexact):
3785: (8)               virtual_indexes_nans = np.isnan(virtual_indexes)
3786: (8)               if virtual_indexes_nans.any():
3787: (12)                 previous_indexes[virtual_indexes_nans] = -1
3788: (12)                 next_indexes[virtual_indexes_nans] = -1
3789: (4)             previous_indexes = previous_indexes.astype(np.intp)
3790: (4)             next_indexes = next_indexes.astype(np.intp)
3791: (4)             return previous_indexes, next_indexes
3792: (0)             def _quantile(
3793: (8)               arr: np.array,
3794: (8)               quantiles: np.array,
3795: (8)               axis: int = -1,
3796: (8)               method="linear",
3797: (8)               out=None,
3798: (0)             ):
3799: (4)             """
3800: (4)             Private function that doesn't support extended axis or keepdims.
3801: (4)             These methods are extended to this function using _ureduce
3802: (4)             See nanpercentile for parameter usage
3803: (4)             It computes the quantiles of the array for the given axis.
3804: (4)             A linear interpolation is performed based on the `interpolation`.
3805: (4)             By default, the method is "linear" where alpha == beta == 1 which
3806: (4)             performs the 7th method of Hyndman&Fan.
3807: (4)             With "median_unbiased" we get alpha == beta == 1/3
3808: (4)             thus the 8th method of Hyndman&Fan.
3809: (4)             """
3810: (4)             arr = np.asanyarray(arr)
3811: (4)             values_count = arr.shape[axis]
3812: (4)             if axis != 0: # But moveaxis is slow, so only call it if necessary.
3813: (8)               arr = np.moveaxis(arr, axis, destination=0)
3814: (4)             try:
3815: (8)               method = _QuantileMethods[method]
3816: (4)             except KeyError:
3817: (8)               raise ValueError(
3818: (12)                 f"{method!r} is not a valid method. Use one of: "
3819: (12)                 f" {_QuantileMethods.keys()}" from None
3820: (4)             virtual_indexes = method["get_virtual_index"](values_count, quantiles)
3821: (4)             virtual_indexes = np.asanyarray(virtual_indexes)
3822: (4)             supports_nans = (
3823: (12)               np.issubdtype(arr.dtype, np.inexact) or arr.dtype.kind in 'Mm')
3824: (4)             if np.issubdtype(virtual_indexes.dtype, np.integer):
3825: (8)               if supports_nans:
3826: (12)                 arr.partition(concatenate((virtual_indexes.ravel(), [-1])),
3827: (12)                   slices_having_nans = np.isnan(arr[-1, ...]))
3828: (8)                 else:
3829: (12)                   arr.partition(virtual_indexes.ravel(), axis=0)
3830: (12)                   slices_having_nans = np.array(False, dtype=bool)
3831: (8)                   result = take(arr, virtual_indexes, axis=0, out=out)
3832: (4)                 else:
3833: (8)                   previous_indexes, next_indexes = _get_indexes(arr,
3834: (54)                               virtual_indexes,

```

```

3835: (54)
3836: (8)
3837: (12)
3838: (38)
3839: (38)
3840: (38)
3841: (12)
3842: (8)
3843: (12)
3844: (8)
3845: (12)
3846: (8)
3847: (8)
3848: (8)
3849: (8)
3850: (8)
3851: (8)
3852: (23)
3853: (23)
3854: (23)
3855: (4)
3856: (8)
3857: (12)
3858: (8)
3859: (12)
3860: (4)
3861: (0)
3862: (4)
3863: (0)
3864: (0)
3865: (4)
3866: (4)
3867: (4)
3868: (4)
3869: (4)
3870: (4)
3871: (4)
3872: (4)
3873: (4)
3874: (4)
3875: (4)
3876: (4)
3877: (8)
3878: (4)
3879: (8)
3880: (8)
3881: (8)
3882: (4)
3883: (8)
3884: (4)
3885: (8)
3886: (4)
3887: (4)
3888: (4)
3889: (8)
3890: (8)
array,
3891: (8)
3892: (8)
3893: (4)
3894: (4)
3895: (4)
3896: (4)
3897: (4)
3898: (4)
3899: (4)
3900: (4)
3901: (4)
3902: (4)

values_count)
    arr.partition(
        np.unique(np.concatenate(([0, -1],
                                previous_indexes.ravel(),
                                next_indexes.ravel(),
                                ))),
        axis=0)
    if supports_nans:
        slices_having_nans = np.isnan(arr[-1, ...])
    else:
        slices_having_nans = None
    previous = arr[previous_indexes]
    next = arr[next_indexes]
    gamma = _get_gamma(virtual_indexes, previous_indexes, method)
    result_shape = virtual_indexes.shape + (1,) * (arr.ndim - 1)
    gamma = gamma.reshape(result_shape)
    result = _lerp(previous,
                    next,
                    gamma,
                    out=out)
    if np.any(slices_having_nans):
        if result.ndim == 0 and out is None:
            result = arr[-1]
        else:
            np.copyto(result, arr[-1, ...], where=slices_having_nans)
    return result
def _trapz_dispatcher(y, x=None, dx=None, axis=None):
    return (y, x)
@array_function_dispatch(_trapz_dispatcher)
def trapz(y, x=None, dx=1.0, axis=-1):
    """
    Integrate along the given axis using the composite trapezoidal rule.
    If `x` is provided, the integration happens in sequence along its
    elements - they are not sorted.
    Integrate `y` (`x`) along each 1d slice on the given axis, compute
    :math:`\int y(x) dx`.
    When `x` is specified, this integrates along the parametric curve,
    computing :math:`\int_t y(t) dt =
    \int_t y(t) \left.\frac{dx}{dt}\right|_{x=x(t)} dt`.
    Parameters
    -----
    y : array_like
        Input array to integrate.
    x : array_like, optional
        The sample points corresponding to the `y` values. If `x` is None,
        the sample points are assumed to be evenly spaced `dx` apart. The
        default is None.
    dx : scalar, optional
        The spacing between sample points when `x` is None. The default is 1.
    axis : int, optional
        The axis along which to integrate.
    Returns
    -----
    trapz : float or ndarray
        Definite integral of `y` = n-dimensional array as approximated along
        a single axis by the trapezoidal rule. If `y` is a 1-dimensional
        then the result is a float. If `n` is greater than 1, then the result
        is an `n`-1 dimensional array.
    See Also
    -----
    sum, cumsum
    Notes
    -----
    Image [2]_ illustrates trapezoidal rule -- y-axis locations of points
    will be taken from `y` array, by default x-axis distances between
    points will be 1.0, alternatively they can be provided with `x` array
    or with `dx` scalar. Return value will be equal to combined area under
    the red lines.

```

```

3903: (4)             References
3904: (4)
3905: (4)             .. [1] Wikipedia page: https://en.wikipedia.org/wiki/Trapezoidal\_rule
3906: (4)
3907: (11)
https://en.wikipedia.org/wiki/File:Composite\_trapezoidal\_rule\_illustration.png
3908: (4)             Examples
3909: (4)
3910: (4)             -----
3911: (4)             Use the trapezoidal rule on evenly spaced points:
3912: (4)             >>> np.trapz([1, 2, 3])
3913: (4)             4.0
3914: (4)             The spacing between sample points can be selected by either the
3915: (4)             ``x`` or ``dx`` arguments:
3916: (4)             >>> np.trapz([1, 2, 3], x=[4, 6, 8])
3917: (4)             8.0
3918: (4)             >>> np.trapz([1, 2, 3], dx=2)
3919: (4)             8.0
3920: (4)             Using a decreasing ``x`` corresponds to integrating in reverse:
3921: (4)             >>> np.trapz([1, 2, 3], x=[8, 6, 4])
3922: (4)             -8.0
3923: (4)             More generally ``x`` is used to integrate along a parametric curve. We can
3924: (4)             estimate the integral :math:`\int_0^1 x^2 = 1/3` using:
3925: (4)             >>> x = np.linspace(0, 1, num=50)
3926: (4)             >>> y = x**2
3927: (4)             0.33340274885464394
3928: (4)             Or estimate the area of a circle, noting we repeat the sample which closes
3929: (4)             the curve:
3930: (4)             >>> theta = np.linspace(0, 2 * np.pi, num=1000, endpoint=True)
3931: (4)             >>> np.trapz(np.cos(theta), x=np.sin(theta))
3932: (4)             3.141571941375841
3933: (4)             ``np.trapz`` can be applied along a specified axis to do multiple
3934: (4)             computations in one call:
3935: (4)             >>> a = np.arange(6).reshape(2, 3)
3936: (4)
3937: (4)             >>> a
3938: (11)            array([[0, 1, 2],
3939: (4)                         [3, 4, 5]])
3940: (4)             >>> np.trapz(a, axis=0)
3941: (4)             array([1.5, 2.5, 3.5])
3942: (4)             >>> np.trapz(a, axis=1)
3943: (4)             array([2., 8.])
3944: (4)
3945: (4)             y = asanyarray(y)
3946: (8)             if x is None:
3947: (4)                 d = dx
3948: (8)             else:
3949: (8)                 x = asanyarray(x)
3950: (12)            if x.ndim == 1:
3951: (12)                d = diff(x)
3952: (12)                shape = [1]*y.ndim
3953: (12)                shape[axis] = d.shape[0]
3954: (8)                d = d.reshape(shape)
3955: (12)            else:
3956: (4)                d = diff(x, axis=axis)
3957: (4)            nd = y.ndim
3958: (4)            slice1 = [slice(None)]*nd
3959: (4)            slice2 = [slice(None)]*nd
3960: (4)            slice1[axis] = slice(1, None)
3961: (4)            slice2[axis] = slice(None, -1)
3962: (8)            try:
3963: (4)                ret = (d * (y[tuple(slice1)] + y[tuple(slice2)])) / 2.0).sum(axis)
3964: (4)            except ValueError:
3965: (8)                d = np.asarray(d)
3966: (8)                y = np.asarray(y)
3967: (8)                ret = add.reduce(d * (y[tuple(slice1)]+y[tuple(slice2)]) / 2.0, axis)
3968: (0)            return ret
3969: (0)            assert not hasattr(trapz, "__code__")
3970: (4)            def _fake_trapz(y, x=None, dx=1.0, axis=-1):
3971: (4)                return trapz(y, x=x, dx=dx, axis=axis)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3971: (0)
3972: (0)
3973: (0)
3974: (0)
3975: (0)
3976: (0)
3977: (4)
3978: (0)
3979: (0)
3980: (4)
3981: (4)
3982: (4)
3983: (4)
3984: (4)
3985: (4)
3986: (7)
3987: (4)
3988: (4)
3989: (4)
3990: (8)
3991: (4)
3992: (8)
3993: (8)
3994: (8)
3995: (4)
3996: (8)
3997: (8)
3998: (8)
3999: (8)
4000: (8)
4001: (8)
4002: (8)
4003: (8)
4004: (4)
4005: (8)
4006: (8)
4007: (8)
4008: (8)
4009: (8)
4010: (8)
4011: (8)
4012: (4)
4013: (4)
4014: (4)
4015: (8)
4016: (8)
4017: (8)
4018: (8)
4019: (8)
4020: (4)
4021: (4)
4022: (4)
4023: (4)
4024: (4)
4025: (4)
4026: (4)
4027: (4)
4028: (4)
4029: (4)
4030: (8)
4031: (8)
4032: (12)
4033: (8)
4034: (8)
4035: (12)
4036: (4)
4037: (4)
4038: (4)
4039: (4)

trapz.__code__ = _fake_trapz.__code__
trapz.__globals__ = _fake_trapz.__globals__
trapz.__defaults__ = _fake_trapz.__defaults__
trapz.__closure__ = _fake_trapz.__closure__
trapz.__kwdefaults__ = _fake_trapz.__kwdefaults__
def _meshgrid_dispatcher(*xi, copy=None, sparse=None, indexing=None):
    return xi
@array_function_dispatch(_meshgrid_dispatcher)
def meshgrid(*xi, copy=True, sparse=False, indexing='xy'):
    """
        Return a list of coordinate matrices from coordinate vectors.
        Make N-D coordinate arrays for vectorized evaluations of
        N-D scalar/vector fields over N-D grids, given
        one-dimensional coordinate arrays x1, x2,..., xn.
        .. versionchanged:: 1.9
            1-D and 0-D cases are allowed.
    Parameters
    -----
    x1, x2,..., xn : array_like
        1-D arrays representing the coordinates of a grid.
    indexing : {'xy', 'ij'}, optional
        Cartesian ('xy', default) or matrix ('ij') indexing of output.
        See Notes for more details.
        .. versionadded:: 1.7.0
    sparse : bool, optional
        If True the shape of the returned coordinate array for dimension *i*
        is reduced from ``(N1, ..., Ni, ... Nn)`` to
        ``(1, ..., 1, Ni, 1, ..., 1)``. These sparse coordinate grids are
        intended to be used with :ref:`basics.broadcasting`. When all
        coordinates are used in an expression, broadcasting still leads to a
        fully-dimensional result array.
        Default is False.
        .. versionadded:: 1.7.0
    copy : bool, optional
        If False, a view into the original arrays are returned in order to
        conserve memory. Default is True. Please note that
        ``sparse=False, copy=False`` will likely return non-contiguous
        arrays. Furthermore, more than one element of a broadcast array
        may refer to a single memory location. If you need to write to the
        arrays, make copies first.
        .. versionadded:: 1.7.0
    Returns
    -----
    X1, X2,..., XN : list of ndarrays
        For vectors `x1`, `x2`,..., `xn` with lengths ``Ni=len(xi)``,
        returns ``(N1, N2, N3,..., Nn)`` shaped arrays if indexing='ij'
        or ``(N2, N1, N3,..., Nn)`` shaped arrays if indexing='xy'
        with the elements of `xi` repeated to fill the matrix along
        the first dimension for `x1`, the second for `x2` and so on.
    Notes
    -----
    This function supports both indexing conventions through the indexing
    keyword argument. Giving the string 'ij' returns a meshgrid with
    matrix indexing, while 'xy' returns a meshgrid with Cartesian indexing.
    In the 2-D case with inputs of length M and N, the outputs are of shape
    (N, M) for 'xy' indexing and (M, N) for 'ij' indexing. In the 3-D case
    with inputs of length M, N and P, outputs are of shape (N, M, P) for
    'xy' indexing and (M, N, P) for 'ij' indexing. The difference is
    illustrated by the following code snippet::
        xv, yv = np.meshgrid(x, y, indexing='ij')
        for i in range(nx):
            for j in range(ny):
                xv, yv = np.meshgrid(x, y, indexing='xy')
                for i in range(nx):
                    for j in range(ny):
    In the 1-D and 0-D case, the indexing and sparse keywords have no effect.
    See Also
    -----
    mgrid : Construct a multi-dimensional "meshgrid" using indexing notation.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4040: (4) ogrid : Construct an open multi-dimensional "meshgrid" using indexing
4041: (12) notation.
4042: (4)
4043: (4)
4044: (4)
4045: (4)
4046: (4)
4047: (4)
4048: (4)
4049: (4)
4050: (4)
4051: (11)
4052: (4)
4053: (4)
4054: (11)
4055: (4) The result of `meshgrid` is a coordinate grid:
4056: (4)
4057: (4)
4058: (4)
4059: (4) You can create sparse output arrays to save memory and computation time.
4060: (4)
4061: (4)
4062: (4)
4063: (4)
4064: (4)
4065: (11)
4066: (4) `meshgrid` is very useful to evaluate functions on a grid. If the
4067: (4) function depends on all coordinates, both dense and sparse outputs can be
4068: (4) used.
4069: (4)
4070: (4)
4071: (4)
4072: (4)
4073: (4)
4074: (4)
4075: (4)
4076: (4)
4077: (4)
4078: (4)
4079: (4)
4080: (4)
4081: (4)
4082: (4)
4083: (4)
4084: (4)
4085: (4)
4086: (4)
4087: (4)
4088: (4)
4089: (4)
4090: (8) if indexing not in ['xy', 'ij']:
4091: (12)     raise ValueError(
4092: (4)         "Valid values for `indexing` are 'xy' and 'ij'.")
4093: (4)
4094: (14)
4095: (4)
4096: (8)
4097: (8)
4098: (4)
4099: (8)
4100: (4)
4101: (8)
4102: (4)
4103: (0)
4104: (4)
4105: (0)
4106: (0)
4107: (4)
4108: (4)     def _delete_dispatcher(arr, obj, axis=None):
4108: (4)         return (arr, obj)
4108: (4)     @array_function_dispatch(_delete_dispatcher)
4108: (4)     def delete(arr, obj, axis=None):
4108: (4)         """
4108: (4)             Return a new array with sub-arrays along an axis deleted. For a one

```

```

4109: (4) dimensional array, this returns those entries not returned by
4110: (4) `arr[obj]`.
4111: (4) Parameters
4112: (4) -----
4113: (4) arr : array_like
4114: (8) Input array.
4115: (4) obj : slice, int or array of ints
4116: (8) Indicate indices of sub-arrays to remove along the specified axis.
4117: (8) .. versionchanged:: 1.19.0
4118: (12) Boolean indices are now treated as a mask of elements to remove,
4119: (12) rather than being cast to the integers 0 and 1.
4120: (4) axis : int, optional
4121: (8) The axis along which to delete the subarray defined by `obj`.
4122: (8) If `axis` is None, `obj` is applied to the flattened array.
4123: (4) Returns
4124: (4) -----
4125: (4) out : ndarray
4126: (8) A copy of `arr` with the elements specified by `obj` removed. Note
4127: (8) that `delete` does not occur in-place. If `axis` is None, `out` is
4128: (8) a flattened array.
4129: (4) See Also
4130: (4) -----
4131: (4) insert : Insert elements into an array.
4132: (4) append : Append elements at the end of an array.
4133: (4) Notes
4134: (4) -----
4135: (4) Often it is preferable to use a boolean mask. For example:
4136: (4) >>> arr = np.arange(12) + 1
4137: (4) >>> mask = np.ones(len(arr), dtype=bool)
4138: (4) >>> mask[[0,2,4]] = False
4139: (4) >>> result = arr[mask,...]
4140: (4) Is equivalent to ``np.delete(arr, [0,2,4], axis=0)``, but allows further
4141: (4) use of `mask`.
4142: (4) Examples
4143: (4) -----
4144: (4) >>> arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
4145: (4) >>> arr
4146: (4) array([[ 1,  2,  3,  4],
4147: (11)      [ 5,  6,  7,  8],
4148: (11)      [ 9, 10, 11, 12]])
4149: (4) >>> np.delete(arr, 1, 0)
4150: (4) array([[ 1,  2,  3,  4],
4151: (11)      [ 9, 10, 11, 12]])
4152: (4) >>> np.delete(arr, np.s_[:2], 1)
4153: (4) array([[ 2,  4],
4154: (11)      [ 6,  8],
4155: (11)      [10, 12]])
4156: (4) >>> np.delete(arr, [1,3,5], None)
4157: (4) array([ 1,  3,  5,  7,  8,  9, 10, 11, 12])
4158: (4) """
4159: (4) wrap = None
4160: (4) if type(arr) is not ndarray:
4161: (8)     try:
4162: (12)         wrap = arr.__array_wrap__
4163: (8)     except AttributeError:
4164: (12)         pass
4165: (4)     arr = asarray(arr)
4166: (4)     ndim = arr.ndim
4167: (4)     arrorder = 'F' if arr.flags.fnc else 'C'
4168: (4)     if axis is None:
4169: (8)         if ndim != 1:
4170: (12)             arr = arr.ravel()
4171: (8)         ndim = arr.ndim
4172: (8)         axis = ndim - 1
4173: (4)     else:
4174: (8)         axis = normalize_axis_index(axis, ndim)
4175: (4)         slobj = [slice(None)]*ndim
4176: (4)         N = arr.shape[axis]
4177: (4)         newshape = list(arr.shape)

```

```

4178: (4)             if isinstance(obj, slice):
4179: (8)                 start, stop, step = obj.indices(N)
4180: (8)                 xr = range(start, stop, step)
4181: (8)                 numtodel = len(xr)
4182: (8)                 if numtodel <= 0:
4183: (12)                     if wrap:
4184: (16)                         return wrap(arr.copy(order=arrorder))
4185: (12)                     else:
4186: (16)                         return arr.copy(order=arrorder)
4187: (8)                 if step < 0:
4188: (12)                     step = -step
4189: (12)                     start = xr[-1]
4190: (12)                     stop = xr[0] + 1
4191: (8)                 newshape[axis] -= numtodel
4192: (8)                 new = empty(newshape, arr.dtype, arrorder)
4193: (8)                 if start == 0:
4194: (12)                     pass
4195: (8)                 else:
4196: (12)                     slobj[axis] = slice(None, start)
4197: (12)                     new[tuple(slobj)] = arr[tuple(slobj)]
4198: (8)                 if stop == N:
4199: (12)                     pass
4200: (8)                 else:
4201: (12)                     slobj[axis] = slice(stop-numtodel, None)
4202: (12)                     slobj2 = [slice(None)]*ndim
4203: (12)                     slobj2[axis] = slice(stop, None)
4204: (12)                     new[tuple(slobj)] = arr[tuple(slobj2)]
4205: (8)                 if step == 1:
4206: (12)                     pass
4207: (8)                 else: # use array indexing.
4208: (12)                     keep = ones(stop-start, dtype=bool)
4209: (12)                     keep[:stop-start:step] = False
4210: (12)                     slobj[axis] = slice(start, stop-numtodel)
4211: (12)                     slobj2 = [slice(None)]*ndim
4212: (12)                     slobj2[axis] = slice(start, stop)
4213: (12)                     arr = arr[tuple(slobj2)]
4214: (12)                     slobj2[axis] = keep
4215: (12)                     new[tuple(slobj)] = arr[tuple(slobj2)]
4216: (8)                 if wrap:
4217: (12)                     return wrap(new)
4218: (8)                 else:
4219: (12)                     return new
4220: (4)                 if isinstance(obj, (int, integer)) and not isinstance(obj, bool):
4221: (8)                     single_value = True
4222: (4)                 else:
4223: (8)                     single_value = False
4224: (8)                     _obj = obj
4225: (8)                     obj = np.asarray(obj)
4226: (8)                     if obj.size == 0 and not isinstance(_obj, np.ndarray):
4227: (12)                         obj = obj.astype(intp)
4228: (8)                     elif obj.size == 1 and obj.dtype.kind in "ui":
4229: (12)                         obj = obj.item()
4230: (12)                         single_value = True
4231: (4)                 if single_value:
4232: (8)                     if (obj < -N or obj >= N):
4233: (12)                         raise IndexError(
4234: (16)                             "index %i is out of bounds for axis %i with "
4235: (16)                             "size %i" % (obj, axis, N))
4236: (8)                     if (obj < 0):
4237: (12)                         obj += N
4238: (8)                     newshape[axis] -= 1
4239: (8)                     new = empty(newshape, arr.dtype, arrorder)
4240: (8)                     slobj[axis] = slice(None, obj)
4241: (8)                     new[tuple(slobj)] = arr[tuple(slobj)]
4242: (8)                     slobj[axis] = slice(obj, None)
4243: (8)                     slobj2 = [slice(None)]*ndim
4244: (8)                     slobj2[axis] = slice(obj+1, None)
4245: (8)                     new[tuple(slobj)] = arr[tuple(slobj2)]
4246: (4)                 else:

```

```

4247: (8)             if obj.dtype == bool:
4248: (12)            if obj.shape != (N,):
4249: (16)              raise ValueError('boolean array argument obj to delete '
4250: (33)                'must be one dimensional and match the axis '
4251: (33)                  'length of {}'.format(N))
4252: (12)            keep = ~obj
4253: (8)        else:
4254: (12)          keep = ones(N, dtype=bool)
4255: (12)          keep[obj,] = False
4256: (8)          slobj[axis] = keep
4257: (8)          new = arr[tuple(slobj)]
4258: (4)      if wrap:
4259: (8)          return wrap(new)
4260: (4)    else:
4261: (8)        return new
4262: (0) def _insert_dispatcher(arr, obj, values, axis=None):
4263: (4)    return (arr, obj, values)
4264: (0) @array_function_dispatch(_insert_dispatcher)
4265: (0) def insert(arr, obj, values, axis=None):
4266: (4)    """
4267: (4)        Insert values along the given axis before the given indices.
4268: (4)        Parameters
4269: (4)        -----
4270: (4)        arr : array_like
4271: (8)          Input array.
4272: (4)        obj : int, slice or sequence of ints
4273: (8)          Object that defines the index or indices before which `values` is
4274: (8)          inserted.
4275: (8)          .. versionadded:: 1.8.0
4276: (8)          Support for multiple insertions when `obj` is a single scalar or a
4277: (8)          sequence with one element (similar to calling insert multiple
4278: (8)          times).
4279: (4)        values : array_like
4280: (8)          Values to insert into `arr`. If the type of `values` is different
4281: (8)          from that of `arr`, `values` is converted to the type of `arr`.
4282: (8)          `values` should be shaped so that ``arr[...,obj,...] = values```
4283: (8)          is legal.
4284: (4)        axis : int, optional
4285: (8)          Axis along which to insert `values`. If `axis` is None then `arr`  

4286: (8)          is flattened first.
4287: (4)        Returns
4288: (4)        -----
4289: (4)        out : ndarray
4290: (8)          A copy of `arr` with `values` inserted. Note that `insert`  

4291: (8)          does not occur in-place: a new array is returned. If
4292: (8)          `axis` is None, `out` is a flattened array.
4293: (4)        See Also
4294: (4)        -----
4295: (4)        append : Append elements at the end of an array.
4296: (4)        concatenate : Join a sequence of arrays along an existing axis.
4297: (4)        delete : Delete elements from an array.
4298: (4)        Notes
4299: (4)        -----
4300: (4)        Note that for higher dimensional inserts ``obj=0`` behaves very different
4301: (4)        from ``obj=[0]`` just like ``arr[:,0,:] = values`` is different from
4302: (4)        ``arr[:,[0],:] = values``.
4303: (4)        Examples
4304: (4)        -----
4305: (4)        >>> a = np.array([[1, 1], [2, 2], [3, 3]])
4306: (4)        >>> a
4307: (4)        array([[1, 1],
4308: (11)          [2, 2],
4309: (11)          [3, 3]])
4310: (4)        >>> np.insert(a, 1, 5)
4311: (4)        array([1, 5, 1, ..., 2, 3, 3])
4312: (4)        >>> np.insert(a, 1, 5, axis=1)
4313: (4)        array([[1, 5, 1],
4314: (11)          [2, 5, 2],
4315: (11)          [3, 5, 3]])

```

```

4316: (4)           Difference between sequence and scalars:
4317: (4)           >>> np.insert(a, [1], [[1],[2],[3]], axis=1)
4318: (4)           array([[1, 1, 1],
4319: (11)             [2, 2, 2],
4320: (11)             [3, 3, 3]])
4321: (4)           >>> np.array_equal(np.insert(a, 1, [1, 2, 3], axis=1),
4322: (4)                         ...                   np.insert(a, [1], [[1],[2],[3]], axis=1))
4323: (4)           True
4324: (4)           >>> b = a.flatten()
4325: (4)           >>> b
4326: (4)           array([1, 1, 2, 2, 3, 3])
4327: (4)           >>> np.insert(b, [2, 2], [5, 6])
4328: (4)           array([1, 1, 5, ..., 2, 3, 3])
4329: (4)           >>> np.insert(b, slice(2, 4), [5, 6])
4330: (4)           array([1, 1, 5, ..., 2, 3, 3])
4331: (4)           >>> np.insert(b, [2, 2], [7.13, False]) # type casting
4332: (4)           array([1, 1, 7, ..., 2, 3, 3])
4333: (4)           >>> x = np.arange(8).reshape(2, 4)
4334: (4)           >>> idx = (1, 3)
4335: (4)           >>> np.insert(x, idx, 999, axis=1)
4336: (4)           array([[ 0, 999,   1,   2, 999,   3],
4337: (11)             [ 4, 999,   5,   6, 999,   7]])
4338: (4)           """
4339: (4)           wrap = None
4340: (4)           if type(arr) is not ndarray:
4341: (8)               try:
4342: (12)                   wrap = arr.__array_wrap__
4343: (8)               except AttributeError:
4344: (12)                   pass
4345: (4)               arr = asarray(arr)
4346: (4)               ndim = arr.ndim
4347: (4)               arrorder = 'F' if arr.flags.fnc else 'C'
4348: (4)               if axis is None:
4349: (8)                   if ndim != 1:
4350: (12)                       arr = arr.ravel()
4351: (8)                   ndim = arr.ndim
4352: (8)                   axis = ndim - 1
4353: (4)               else:
4354: (8)                   axis = normalize_axis_index(axis, ndim)
4355: (4)               slobj = [slice(None)]*ndim
4356: (4)               N = arr.shape[axis]
4357: (4)               newshape = list(arr.shape)
4358: (4)               if isinstance(obj, slice):
4359: (8)                   indices = arange(*obj.indices(N), dtype=intp)
4360: (4)               else:
4361: (8)                   indices = np.array(obj)
4362: (8)                   if indices.dtype == bool:
4363: (12)                       warnings.warn(
4364: (16)                           "in the future insert will treat boolean arrays and "
4365: (16)                           "array-likes as a boolean index instead of casting it to "
4366: (16)                           "integer", FutureWarning, stacklevel=2)
4367: (12)                       indices = indices.astype(intp)
4368: (8)                   elif indices.ndim > 1:
4369: (12)                       raise ValueError(
4370: (16)                           "index array argument obj to insert must be one dimensional "
4371: (16)                           "or scalar")
4372: (4)               if indices.size == 1:
4373: (8)                   index = indices.item()
4374: (8)                   if index < -N or index > N:
4375: (12)                       raise IndexError(f"index {obj} is out of bounds for axis {axis} "
4376: (29)                           f"with size {N}")
4377: (8)                   if (index < 0):
4378: (12)                       index += N
4379: (8)                   values = array(values, copy=False, ndmin=arr.ndim, dtype=arr.dtype)
4380: (8)                   if indices.ndim == 0:
4381: (12)                       values = np.moveaxis(values, 0, axis)
4382: (8)                   numnew = values.shape[axis]
4383: (8)                   newshape[axis] += numnew
4384: (8)                   new = empty(newshape, arr.dtype, arrorder)

```

```

4385: (8)             slobj[axis] = slice(None, index)
4386: (8)             new[tuple(slobj)] = arr[tuple(slobj)]
4387: (8)             slobj[axis] = slice(index, index+numnew)
4388: (8)             new[tuple(slobj)] = values
4389: (8)             slobj[axis] = slice(index+numnew, None)
4390: (8)             slobj2 = [slice(None)] * ndim
4391: (8)             slobj2[axis] = slice(index, None)
4392: (8)             new[tuple(slobj)] = arr[tuple(slobj2)]
4393: (8)             if wrap:
4394: (12)                 return wrap(new)
4395: (8)             return new
4396: (4)             elif indices.size == 0 and not isinstance(obj, np.ndarray):
4397: (8)                 indices = indices.astype(intp)
4398: (4)                 indices[indices < 0] += N
4399: (4)                 numnew = len(indices)
4400: (4)                 order = indices.argsort(kind='mergesort')    # stable sort
4401: (4)                 indices[order] += np.arange(numnew)
4402: (4)                 newshape[axis] += numnew
4403: (4)                 old_mask = ones(newshape[axis], dtype=bool)
4404: (4)                 old_mask[indices] = False
4405: (4)                 new = empty(newshape, arr.dtype, arr.order)
4406: (4)                 slobj2 = [slice(None)]*ndim
4407: (4)                 slobj[axis] = indices
4408: (4)                 slobj2[axis] = old_mask
4409: (4)                 new[tuple(slobj)] = values
4410: (4)                 new[tuple(slobj2)] = arr
4411: (4)                 if wrap:
4412: (8)                     return wrap(new)
4413: (4)             return new
4414: (0)             def _append_dispatcher(arr, values, axis=None):
4415: (4)                 return (arr, values)
4416: (0)             @array_function_dispatch(_append_dispatcher)
4417: (0)             def append(arr, values, axis=None):
4418: (4)                 """
4419: (4)                 Append values to the end of an array.
4420: (4)                 Parameters
4421: (4)                 -----
4422: (4)                 arr : array_like
4423: (8)                     Values are appended to a copy of this array.
4424: (4)                 values : array_like
4425: (8)                     These values are appended to a copy of `arr`. It must be of the
4426: (8)                     correct shape (the same shape as `arr`, excluding `axis`). If
4427: (8)                     `axis` is not specified, `values` can be any shape and will be
4428: (8)                     flattened before use.
4429: (4)                 axis : int, optional
4430: (8)                     The axis along which `values` are appended. If `axis` is not
4431: (8)                     given, both `arr` and `values` are flattened before use.
4432: (4)             Returns
4433: (4)                 -----
4434: (4)                 append : ndarray
4435: (8)                     A copy of `arr` with `values` appended to `axis`. Note that
4436: (8)                     `append` does not occur in-place: a new array is allocated and
4437: (8)                     filled. If `axis` is None, `out` is a flattened array.
4438: (4)             See Also
4439: (4)                 -----
4440: (4)                 insert : Insert elements into an array.
4441: (4)                 delete : Delete elements from an array.
4442: (4)             Examples
4443: (4)                 -----
4444: (4)                 >>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
4445: (4)                 array([1, 2, 3, ..., 7, 8, 9])
4446: (4)                 When `axis` is specified, `values` must have the correct shape.
4447: (4)                 >>> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
4448: (4)                 array([[1, 2, 3],
4449: (11)                     [4, 5, 6],
4450: (11)                     [7, 8, 9]])
4451: (4)                 >>> np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)
4452: (4)                 Traceback (most recent call last):
4453: (8)                     ...

```

```

4454: (4)          ValueError: all the input arrays must have same number of dimensions, but
4455: (4)          the array at index 0 has 2 dimension(s) and the array at index 1 has 1
4456: (4)          dimension(s)
4457: (4)          """
4458: (4)          arr = asanyarray(arr)
4459: (4)          if axis is None:
4460: (8)              if arr.ndim != 1:
4461: (12)                  arr = arr.ravel()
4462: (8)                  values = ravel(values)
4463: (8)                  axis = arr.ndim-1
4464: (4)          return concatenate((arr, values), axis=axis)
4465: (0)          def _digitize_dispatcher(x, bins, right=None):
4466: (4)              return (x, bins)
4467: (0)          @array_function_dispatch(_digitize_dispatcher)
4468: (0)          def digitize(x, bins, right=False):
4469: (4)              """
4470: (4)              Return the indices of the bins to which each value in input array belongs.
4471: (4)              =====
4472: (4)              `right` order of bins returned index `i` satisfies
4473: (4)              =====
4474: (4)              ``False`` increasing   ``bins[i-1] <= x < bins[i]``
4475: (4)              ``True``  increasing   ``bins[i-1] < x <= bins[i]``
4476: (4)              ``False`` decreasing    ``bins[i-1] > x >= bins[i]``
4477: (4)              ``True``  decreasing    ``bins[i-1] >= x > bins[i]``
4478: (4)              =====
4479: (4)              If values in `x` are beyond the bounds of `bins`, 0 or ``len(bins)`` is
4480: (4)              returned as appropriate.
4481: (4)          Parameters
4482: (4)          -----
4483: (4)          x : array_like
4484: (8)              Input array to be binned. Prior to NumPy 1.10.0, this array had to
4485: (8)              be 1-dimensional, but can now have any shape.
4486: (4)          bins : array_like
4487: (8)              Array of bins. It has to be 1-dimensional and monotonic.
4488: (4)          right : bool, optional
4489: (8)              Indicating whether the intervals include the right or the left bin
4490: (8)              edge. Default behavior is (right==False) indicating that the interval
4491: (8)              does not include the right edge. The left bin end is open in this
4492: (8)              case, i.e., bins[i-1] <= x < bins[i] is the default behavior for
4493: (8)              monotonically increasing bins.
4494: (4)          Returns
4495: (4)          -----
4496: (4)          indices : ndarray of ints
4497: (8)              Output array of indices, of same shape as `x`.
4498: (4)          Raises
4499: (4)          -----
4500: (4)          ValueError
4501: (8)              If `bins` is not monotonic.
4502: (4)          TypeError
4503: (8)              If the type of the input is complex.
4504: (4)          See Also
4505: (4)          -----
4506: (4)          bincount, histogram, unique, searchsorted
4507: (4)          Notes
4508: (4)          -----
4509: (4)          If values in `x` are such that they fall outside the bin range,
4510: (4)          attempting to index `bins` with the indices that `digitize` returns
4511: (4)          will result in an IndexError.
4512: (4)          .. versionadded:: 1.10.0
4513: (4)          `np.digitize` is implemented in terms of `np.searchsorted`. This means
4514: (4)          that a binary search is used to bin the values, which scales much better
4515: (4)          for larger number of bins than the previous linear search. It also removes
4516: (4)          the requirement for the input array to be 1-dimensional.
4517: (4)          For monotonically _increasing_ `bins`, the following are equivalent::
4518: (8)              np.digitize(x, bins, right=True)
4519: (8)              np.searchsorted(bins, x, side='left')
4520: (4)          Note that as the order of the arguments are reversed, the side must be
        too.
4521: (4)          The `searchsorted` call is marginally faster, as it does not do any

```

```

4522: (4)    monotonicity checks. Perhaps more importantly, it supports all dtypes.
4523: (4)    Examples
4524: (4)
4525: (4)    -----
4526: (4)    >>> x = np.array([0.2, 6.4, 3.0, 1.6])
4527: (4)    >>> bins = np.array([0.0, 1.0, 2.5, 4.0, 10.0])
4528: (4)    >>> inds = np.digitize(x, bins)
4529: (4)    >>> inds
4530: (4)    array([1, 4, 3, 2])
4531: (4)    >>> for n in range(x.size):
4532: (4)        ...     print(bins[inds[n]-1], "<=", x[n], "<", bins[inds[n]])
4533: (4)        ...
4534: (4)        0.0 <= 0.2 < 1.0
4535: (4)        4.0 <= 6.4 < 10.0
4536: (4)        2.5 <= 3.0 < 4.0
4537: (4)        1.0 <= 1.6 < 2.5
4538: (4)    >>> x = np.array([1.2, 10.0, 12.4, 15.5, 20.])
4539: (4)    >>> bins = np.array([0, 5, 10, 15, 20])
4540: (4)    >>> np.digitize(x,bins,right=True)
4541: (4)    array([1, 2, 3, 4, 4])
4542: (4)    >>> np.digitize(x,bins,right=False)
4543: (4)    array([1, 3, 3, 4, 5])
4544: (4)    """
4545: (4)    x = _nx.asarray(x)
4546: (4)    bins = _nx.asarray(bins)
4547: (8)    if np.issubdtype(x.dtype, _nx.complexfloating):
4548: (4)        raise TypeError("x may not be complex")
4549: (4)    mono = _monotonicity(bins)
4550: (8)    if mono == 0:
4551: (4)        raise ValueError("bins must be monotonically increasing or
decreasing")
4552: (4)    side = 'left' if right else 'right'
4553: (8)    if mono == -1:
4554: (4)        return len(bins) - _nx.searchsorted(bins[::-1], x, side=side)
4555: (8)    else:
4556: (4)        return _nx.searchsorted(bins, x, side=side)

```

-----

## File 209 - histograms.py:

```

1: (0)    """
2: (0)    Histogram-related functions
3: (0)    """
4: (0)    import contextlib
5: (0)    import functools
6: (0)    import operator
7: (0)    import warnings
8: (0)    import numpy as np
9: (0)    from numpy.core import overrides
10: (0)    __all__ = ['histogram', 'histogramdd', 'histogram_bin_edges']
11: (0)    array_function_dispatch = functools.partial(
12: (4)        overrides.array_function_dispatch, module='numpy')
13: (0)    _range = range
14: (0)    def _ptp(x):
15: (4)        """Peak-to-peak value of x.
16: (4)        This implementation avoids the problem of signed integer arrays having a
17: (4)        peak-to-peak value that cannot be represented with the array's data type.
18: (4)        This function returns an unsigned value for signed integer arrays.
19: (4)        """
20: (4)        return _unsigned_subtract(x.max(), x.min())
21: (0)    def _hist_bin_sqrt(x, range):
22: (4)        """
23: (4)        Square root histogram bin estimator.
24: (4)        Bin width is inversely proportional to the data size. Used by many
25: (4)        programs for its simplicity.
26: (4)        Parameters
27: (4)        -----
28: (4)        x : array_like
29: (8)            Input data that is to be histogrammed, trimmed to range. May not

```

```

30: (8)           be empty.
31: (4)           Returns
32: (4)
33: (4)           -----
34: (4)           h : An estimate of the optimal bin width for the given data.
35: (4)           """
36: (4)           del range # unused
37: (0)           return _ptp(x) / np.sqrt(x.size)
38: (4)
39: (4)           def _hist_bin_sturges(x, range):
40: (4)           """
41: (4)           Sturges histogram bin estimator.
42: (4)           A very simplistic estimator based on the assumption of normality of
43: (4)           the data. This estimator has poor performance for non-normal data,
44: (4)           which becomes especially obvious for large data sets. The estimate
45: (4)           depends only on size of the data.
46: (4)           Parameters
47: (4)           -----
48: (8)           x : array_like
49: (8)           Input data that is to be histogrammed, trimmed to range. May not
50: (4)           be empty.
51: (4)           Returns
52: (4)
53: (4)           -----
54: (4)           h : An estimate of the optimal bin width for the given data.
55: (0)
56: (4)
57: (4)           def _hist_bin_rice(x, range):
58: (4)           """
59: (4)           Rice histogram bin estimator.
60: (4)           Another simple estimator with no normality assumption. It has better
61: (4)           performance for large data than Sturges, but tends to overestimate
62: (4)           the number of bins. The number of bins is proportional to the cube
63: (4)           root of data size (asymptotically optimal). The estimate depends
64: (4)           only on size of the data.
65: (4)           Parameters
66: (4)           -----
67: (8)           x : array_like
68: (8)           Input data that is to be histogrammed, trimmed to range. May not
69: (4)           be empty.
70: (4)           Returns
71: (4)
72: (4)           -----
73: (4)           h : An estimate of the optimal bin width for the given data.
74: (0)
75: (4)
76: (4)           def _hist_bin_scott(x, range):
77: (4)           """
78: (4)           Scott histogram bin estimator.
79: (4)           The binwidth is proportional to the standard deviation of the data
80: (4)           and inversely proportional to the cube root of data size
81: (4)           (asymptotically optimal).
82: (4)           Parameters
83: (4)           -----
84: (8)           x : array_like
85: (8)           Input data that is to be histogrammed, trimmed to range. May not
86: (4)           be empty.
87: (4)           Returns
88: (4)
89: (4)           -----
90: (4)           h : An estimate of the optimal bin width for the given data.
91: (0)
92: (4)
93: (4)           def _hist_bin_stone(x, range):
94: (4)           """
95: (4)           Histogram bin estimator based on minimizing the estimated integrated
squared error (ISE).
96: (4)           The number of bins is chosen by minimizing the estimated ISE against the
unknown true distribution.
97: (4)           The ISE is estimated using cross-validation and can be regarded as a
generalization of Scott's rule.

```

```

96: (4) https://en.wikipedia.org/wiki/Histogram#Scott.27s_normal_reference_rule
97: (4) This paper by Stone appears to be the origination of this rule.
98: (4) http://digitalassets.lib.berkeley.edu/sdtr/ucb/text/34.pdf
99: (4) Parameters
100: (4) -----
101: (4) x : array_like
102: (8) Input data that is to be histogrammed, trimmed to range. May not
103: (8) be empty.
104: (4) range : (float, float)
105: (8) The lower and upper range of the bins.
106: (4) Returns
107: (4) -----
108: (4) h : An estimate of the optimal bin width for the given data.
109: (4) """
110: (4) n = x.size
111: (4) ptp_x = np.ptp(x)
112: (4) if n <= 1 or ptp_x == 0:
113: (8)     return 0
114: (4) def jhat(nbins):
115: (8)     hh = ptp_x / nbins
116: (8)     p_k = np.histogram(x, bins=nbins, range=range)[0] / n
117: (8)     return (2 - (n + 1) * p_k.dot(p_k)) / hh
118: (4) nbins_upper_bound = max(100, int(np.sqrt(n)))
119: (4) nbins = min(_range(1, nbins_upper_bound + 1), key=jhat)
120: (4) if nbins == nbins_upper_bound:
121: (8)     warnings.warn("The number of bins estimated may be suboptimal.",
122: (22)             RuntimeWarning, stacklevel=3)
123: (4)     return ptp_x / nbins
124: (0) def _hist_bin_doane(x, range):
125: (4) """
126: (4) Doane's histogram bin estimator.
127: (4) Improved version of Sturges' formula which works better for
128: (4) non-normal data. See
129: (4) stats.stackexchange.com/questions/55134/doanes-formula-for-histogram-
binning
130: (4) Parameters
131: (4) -----
132: (4) x : array_like
133: (8) Input data that is to be histogrammed, trimmed to range. May not
134: (8) be empty.
135: (4) Returns
136: (4) -----
137: (4) h : An estimate of the optimal bin width for the given data.
138: (4) """
139: (4) del range # unused
140: (4) if x.size > 2:
141: (8)     sg1 = np.sqrt(6.0 * (x.size - 2) / ((x.size + 1.0) * (x.size + 3)))
142: (8)     sigma = np.std(x)
143: (8)     if sigma > 0.0:
144: (12)         temp = x - np.mean(x)
145: (12)         np.true_divide(temp, sigma, temp)
146: (12)         np.power(temp, 3, temp)
147: (12)         g1 = np.mean(temp)
148: (12)         return np.ptp(x) / (1.0 + np.log2(x.size) +
149: (36)                                         np.log2(1.0 + np.absolute(g1) / sg1))
150: (4)     return 0.0
151: (0) def _hist_bin_fd(x, range):
152: (4) """
153: (4) The Freedman-Diaconis histogram bin estimator.
154: (4) The Freedman-Diaconis rule uses interquartile range (IQR) to
155: (4) estimate binwidth. It is considered a variation of the Scott rule
156: (4) with more robustness as the IQR is less affected by outliers than
157: (4) the standard deviation. However, the IQR depends on fewer points
158: (4) than the standard deviation, so it is less accurate, especially for
159: (4) long tailed distributions.
160: (4) If the IQR is 0, this function returns 0 for the bin width.
161: (4) Binwidth is inversely proportional to the cube root of data size
162: (4) (asymptotically optimal).
163: (4) Parameters

```

```

164: (4)      -----
165: (4)      x : array_like
166: (8)      Input data that is to be histogrammed, trimmed to range. May not
167: (8)      be empty.
168: (4)      Returns
169: (4)      -----
170: (4)      h : An estimate of the optimal bin width for the given data.
171: (4)      """
172: (4)      del range # unused
173: (4)      iqr = np.subtract(*np.percentile(x, [75, 25]))
174: (4)      return 2.0 * iqr * x.size ** (-1.0 / 3.0)
175: (0)      def _hist_bin_auto(x, range):
176: (4)      """
177: (4)      Histogram bin estimator that uses the minimum width of the
178: (4)      Freedman-Diaconis and Sturges estimators if the FD bin width is non-zero.
179: (4)      If the bin width from the FD estimator is 0, the Sturges estimator is
used.
180: (4)      The FD estimator is usually the most robust method, but its width
181: (4)      estimate tends to be too large for small `x` and bad for data with limited
182: (4)      variance. The Sturges estimator is quite good for small (<1000) datasets
183: (4)      and is the default in the R language. This method gives good off-the-shelf
184: (4)      behaviour.
185: (4)      .. versionchanged:: 1.15.0
186: (4)      If there is limited variance the IQR can be 0, which results in the
187: (4)      FD bin width being 0 too. This is not a valid bin width, so
188: (4)      ``np.histogram_bin_edges`` chooses 1 bin instead, which may not be
optimal.
189: (4)      If the IQR is 0, it's unlikely any variance-based estimators will be of
190: (4)      use, so we revert to the Sturges estimator, which only uses the size of
the
191: (4)      dataset in its calculation.
192: (4)      Parameters
193: (4)      -----
194: (4)      x : array_like
195: (8)      Input data that is to be histogrammed, trimmed to range. May not
196: (8)      be empty.
197: (4)      Returns
198: (4)      -----
199: (4)      h : An estimate of the optimal bin width for the given data.
200: (4)      See Also
201: (4)      -----
202: (4)      _hist_bin_fd, _hist_bin_sturges
203: (4)      """
204: (4)      fd_bw = _hist_bin_fd(x, range)
205: (4)      sturges_bw = _hist_bin_sturges(x, range)
206: (4)      del range # unused
207: (4)      if fd_bw:
208: (8)          return min(fd_bw, sturges_bw)
209: (4)      else:
210: (8)          return sturges_bw
211: (0)      _hist_bin_selectors = {'stone': _hist_bin_stone,
212: (23)          'auto': _hist_bin_auto,
213: (23)          'doane': _hist_bin_doane,
214: (23)          'fd': _hist_bin_fd,
215: (23)          'rice': _hist_bin_rice,
216: (23)          'scott': _hist_bin_scott,
217: (23)          'sqrt': _hist_bin_sqrt,
218: (23)          'sturges': _hist_bin_sturges}
219: (0)      def _ravel_and_check_weights(a, weights):
220: (4)      """ Check a and weights have matching shapes, and ravel both """
221: (4)      a = np.asarray(a)
222: (4)      if a.dtype == np.bool_:
223: (8)          warnings.warn("Converting input from {} to {} for compatibility."
224: (22)              .format(a.dtype, np.uint8),
225: (22)              RuntimeWarning, stacklevel=3)
226: (8)          a = a.astype(np.uint8)
227: (4)      if weights is not None:
228: (8)          weights = np.asarray(weights)
229: (8)          if weights.shape != a.shape:

```

```

230: (12)                     raise ValueError(
231: (16)                         'weights should have the same shape as a.')
232: (8)                         weights = weights.ravel()
233: (4)                         a = a.ravel()
234: (4)                         return a, weights
235: (0) def _get_outer_edges(a, range):
236: (4)     """
237: (4)         Determine the outer bin edges to use, from either the data or the range
238: (4)         argument
239: (4)     """
240: (4)         if range is not None:
241: (8)             first_edge, last_edge = range
242: (8)             if first_edge > last_edge:
243: (12)                 raise ValueError(
244: (16)                     'max must be larger than min in range parameter.')
245: (8)             if not (np.isfinite(first_edge) and np.isfinite(last_edge)):
246: (12)                 raise ValueError(
247: (16)                     "supplied range of [{}, {}] is not finite".format(first_edge,
last_edge))
248: (4)         elif a.size == 0:
249: (8)             first_edge, last_edge = 0, 1
250: (4)         else:
251: (8)             first_edge, last_edge = a.min(), a.max()
252: (8)             if not (np.isfinite(first_edge) and np.isfinite(last_edge)):
253: (12)                 raise ValueError(
254: (16)                     "autodetected range of [{}, {}] is not
finite".format(first_edge, last_edge))
255: (4)             if first_edge == last_edge:
256: (8)                 first_edge = first_edge - 0.5
257: (8)                 last_edge = last_edge + 0.5
258: (4)             return first_edge, last_edge
259: (0) def _unsigned_subtract(a, b):
260: (4)     """
261: (4)         Subtract two values where a >= b, and produce an unsigned result
262: (4)         This is needed when finding the difference between the upper and lower
263: (4)         bound of an int16 histogram
264: (4)     """
265: (4)         signed_to_unsigned = {
266: (8)             np.byte: np.ubyte,
267: (8)             np.short: np.ushort,
268: (8)             np.intc: np.uintc,
269: (8)             np.int_: np.uint,
270: (8)             np.longlong: np ulonglong
271: (4)         }
272: (4)         dt = np.result_type(a, b)
273: (4)         try:
274: (8)             dt = signed_to_unsigned[dt.type]
275: (4)         except KeyError:
276: (8)             return np.subtract(a, b, dtype=dt)
277: (4)         else:
278: (8)             return np.subtract(a, b, casting='unsafe', dtype=dt)
279: (0) def _get_bin_edges(a, bins, range, weights):
280: (4)     """
281: (4)         Computes the bins used internally by `histogram`.
282: (4)         Parameters
283: (4)         =====
284: (4)         a : ndarray
285: (8)             Ravelled data array
286: (4)         bins, range
287: (8)             Forwarded arguments from `histogram`.
288: (4)         weights : ndarray, optional
289: (8)             Ravelled weights array, or None
290: (4)         Returns
291: (4)         =====
292: (4)         bin_edges : ndarray
293: (8)             Array of bin edges
294: (4)         uniform_bins : (Number, Number, int):
295: (8)             The upper bound, lowerbound, and number of bins, used in the optimized
implementation of `histogram` that works on uniform bins.
296: (8)

```

```

297: (4)
298: (4)
299: (4)
300: (4)
301: (8)
302: (8)
303: (12)
304: (16)
305: (8)
306: (12)
307: (28)
308: (8)
309: (8)
310: (12)
311: (12)
312: (12)
313: (16)
314: (8)
315: (12)
316: (8)
317: (12)
318: (12)
319: (16)
first_edge) / width))
320: (12)
321: (16)
322: (4)
323: (8)
324: (12)
325: (8)
326: (12)
327: (16)
328: (8)
329: (12)
330: (8)
331: (4)
332: (8)
333: (8)
334: (12)
335: (16)
336: (4)
337: (8)
338: (4)
339: (8)
340: (8)
341: (12)
342: (8)
343: (12)
344: (12)
345: (8)
346: (4)
347: (8)
348: (0)
349: (4)
350: (4)
right.
351: (4)
352: (4)
353: (4)
354: (8)
355: (8)
356: (4)
357: (0)
358: (4)
359: (0)
360: (0)
361: (4)
362: (4)
363: (4)

        """
        n_equal_bins = None
        bin_edges = None
        if isinstance(bins, str):
            bin_name = bins
            if bin_name not in _hist_bin_selectors:
                raise ValueError(
                    "{}!r} is not a valid estimator for `bins`.".format(bin_name))
        if weights is not None:
            raise TypeError("Automated estimation of the number of "
                            "bins is not supported for weighted data")
        first_edge, last_edge = _get_outer_edges(a, range)
        if range is not None:
            keep = (a >= first_edge)
            keep &= (a <= last_edge)
            if not np.logical_and.reduce(keep):
                a = a[keep]
        if a.size == 0:
            n_equal_bins = 1
        else:
            width = _hist_bin_selectors[bin_name](a, (first_edge, last_edge))
            if width:
                n_equal_bins = int(np.ceil(_unsigned_subtract(last_edge,
first_edge) / width))
            else:
                n_equal_bins = 1
    elif np.ndim(bins) == 0:
        try:
            n_equal_bins = operator.index(bins)
        except TypeError as e:
            raise TypeError(
                '`bins` must be an integer, a string, or an array') from e
        if n_equal_bins < 1:
            raise ValueError(`bins` must be positive, when an integer')
        first_edge, last_edge = _get_outer_edges(a, range)
    elif np.ndim(bins) == 1:
        bin_edges = np.asarray(bins)
        if np.any(bin_edges[:-1] > bin_edges[1:]):
            raise ValueError(
                '`bins` must increase monotonically, when an array')
        else:
            raise ValueError(`bins` must be 1d, when an array')
    if n_equal_bins is not None:
        bin_type = np.result_type(first_edge, last_edge, a)
        if np.issubdtype(bin_type, np.integer):
            bin_type = np.result_type(bin_type, float)
        bin_edges = np.linspace(
            first_edge, last_edge, n_equal_bins + 1,
            endpoint=True, dtype=bin_type)
        return bin_edges, (first_edge, last_edge, n_equal_bins)
    else:
        return bin_edges, None
def _search_sorted_inclusive(a, v):
    """
    Like `searchsorted`, but where the last item in `v` is placed on the
    right.
    In the context of a histogram, this makes the last bin edge inclusive
    """
    return np.concatenate((
        a.searchsorted(v[:-1], 'left'),
        a.searchsorted(v[-1:], 'right')
    ))
def _histogram_bin_edges_dispatcher(a, bins=None, range=None, weights=None):
    return (a, bins, weights)
@array_function_dispatch(_histogram_bin_edges_dispatcher)
def histogram_bin_edges(a, bins=10, range=None, weights=None):
    """
    Function to calculate only the edges of the bins used by the `histogram`
    function.

```

```

364: (4)                                         Parameters
365: (4)                                         -----
366: (4)                                         a : array_like
367: (8)                                         Input data. The histogram is computed over the flattened array.
368: (4)                                         bins : int or sequence of scalars or str, optional
369: (8)                                         If `bins` is an int, it defines the number of equal-width
370: (8)                                         bins in the given range (10, by default). If `bins` is a
371: (8)                                         sequence, it defines the bin edges, including the rightmost
372: (8)                                         edge, allowing for non-uniform bin widths.
373: (8)                                         If `bins` is a string from the list below, `histogram_bin_edges` will
use
374: (8)                                         the method chosen to calculate the optimal bin width and
375: (8)                                         consequently the number of bins (see `Notes` for more detail on
376: (8)                                         the estimators) from the data that falls within the requested
377: (8)                                         range. While the bin width will be optimal for the actual data
378: (8)                                         in the range, the number of bins will be computed to fill the
379: (8)                                         entire range, including the empty portions. For visualisation,
380: (8)                                         using the 'auto' option is suggested. Weighted data is not
381: (8)                                         supported for automated bin size selection.
382: (8)                                         'auto'
383: (12)                                         Maximum of the 'sturges' and 'fd' estimators. Provides good
384: (12)                                         all around performance.
385: (8)                                         'fd' (Freedman Diaconis Estimator)
386: (12)                                         Robust (resilient to outliers) estimator that takes into
387: (12)                                         account data variability and data size.
388: (8)                                         'doane'
389: (12)                                         An improved version of Sturges' estimator that works better
390: (12)                                         with non-normal datasets.
391: (8)                                         'scott'
392: (12)                                         Less robust estimator that takes into account data variability
393: (12)                                         and data size.
394: (8)                                         'stone'
395: (12)                                         Estimator based on leave-one-out cross-validation estimate of
396: (12)                                         the integrated squared error. Can be regarded as a generalization
397: (12)                                         of Scott's rule.
398: (8)                                         'rice'
399: (12)                                         Estimator does not take variability into account, only data
400: (12)                                         size. Commonly overestimates number of bins required.
401: (8)                                         'sturges'
402: (12)                                         R's default method, only accounts for data size. Only
403: (12)                                         optimal for gaussian data and underestimates number of bins
404: (12)                                         for large non-gaussian datasets.
405: (8)                                         'sqrt'
406: (12)                                         Square root (of data size) estimator, used by Excel and
407: (12)                                         other programs for its speed and simplicity.
408: (4)                                         range : (float, float), optional
409: (8)                                         The lower and upper range of the bins. If not provided, range
410: (8)                                         is simply ``a.min(), a.max()``. Values outside the range are
411: (8)                                         ignored. The first element of the range must be less than or
412: (8)                                         equal to the second. `range` affects the automatic bin
413: (8)                                         computation as well. While bin width is computed to be optimal
414: (8)                                         based on the actual data within `range`, the bin count will fill
415: (8)                                         the entire range including portions containing no data.
416: (4)                                         weights : array_like, optional
417: (8)                                         An array of weights, of the same shape as `a`. Each value in
418: (8)                                         `a` only contributes its associated weight towards the bin count
419: (8)                                         (instead of 1). This is currently not used by any of the bin
estimators,
420: (8)                                         but may be in the future.
421: (4)                                         Returns
422: (4)                                         -----
423: (4)                                         bin_edges : array of dtype float
424: (8)                                         The edges to pass into `histogram`
425: (4)                                         See Also
426: (4)                                         -----
427: (4)                                         histogram
428: (4)                                         Notes
429: (4)                                         -----
430: (4)                                         The methods to estimate the optimal number of bins are well founded

```

```

431: (4) in literature, and are inspired by the choices R provides for
432: (4) histogram visualisation. Note that having the number of bins
433: (4) proportional to :math:`n^{1/3}` is asymptotically optimal, which is
434: (4) why it appears in most estimators. These are simply plug-in methods
435: (4) that give good starting points for number of bins. In the equations
436: (4) below, :math:`h` is the binwidth and :math:`n_h` is the number of
437: (4) bins. All estimators that compute bin counts are recast to bin width
438: (4) using the `ptp` of the data. The final bin count is obtained from
439: (4) ``np.round(np.ceil(range / h))``. The final bin width is often less
440: (4) than what is returned by the estimators below.
441: (4) 'auto' (maximum of the 'sturges' and 'fd' estimators)
442: (8) A compromise to get a good value. For small datasets the Sturges
443: (8) value will usually be chosen, while larger datasets will usually
444: (8) default to FD. Avoids the overly conservative behaviour of FD
445: (8) and Sturges for small and large datasets respectively.
446: (8) Switchover point is usually :math:`a.size \approx 1000` .
447: (4) 'fd' (Freedman Diaconis Estimator)
448: (8) .. math:: h = 2 \frac{IQR}{n^{1/3}}
449: (8) The binwidth is proportional to the interquartile range (IQR)
450: (8) and inversely proportional to cube root of a.size. Can be too
451: (8) conservative for small datasets, but is quite good for large
452: (8) datasets. The IQR is very robust to outliers.
453: (4) 'scott'
454: (8) .. math:: h = \sigma \sqrt[3]{\frac{24}{\pi} n}
455: (8) The binwidth is proportional to the standard deviation of the
456: (8) data and inversely proportional to cube root of ``x.size``. Can
457: (8) be too conservative for small datasets, but is quite good for
458: (8) large datasets. The standard deviation is not very robust to
459: (8) outliers. Values are very similar to the Freedman-Diaconis
460: (8) estimator in the absence of outliers.
461: (4) 'rice'
462: (8) .. math:: n_h = 2n^{1/3}
463: (8) The number of bins is only proportional to cube root of
464: (8) ``a.size``. It tends to overestimate the number of bins and it
465: (8) does not take into account data variability.
466: (4) 'sturges'
467: (8) .. math:: n_h = \log_2(n) + 1
468: (8) The number of bins is the base 2 log of ``a.size``. This
469: (8) estimator assumes normality of data and is too conservative for
470: (8) larger, non-normal datasets. This is the default method in R's
471: (8) ``hist`` method.
472: (4) 'doane'
473: (8) .. math:: n_h = 1 + \log_2(n) +
474: (24) \log_2(1 + \frac{|g_1|}{\sigma_{g_1}})
475: (12) g_1 = mean(x - \mu)^3
476: (12) \sigma_{g_1} = \sqrt{\frac{6(n-2)}{(n+1)(n+3)}}
477: (8) An improved version of Sturges' formula that produces better
478: (8) estimates for non-normal datasets. This estimator attempts to
479: (8) account for the skew of the data.
480: (4) 'sqrt'
481: (8) .. math:: n_h = \sqrt{n}
482: (8) The simplest and fastest estimator. Only takes into account the
483: (8) data size.
484: (4) Examples
485: (4) -----
486: (4) >>> arr = np.array([0, 0, 0, 1, 2, 3, 3, 4, 5])
487: (4) >>> np.histogram_bin_edges(arr, bins='auto', range=(0, 1))
488: (4) array([0. , 0.25, 0.5 , 0.75, 1. ])
489: (4) >>> np.histogram_bin_edges(arr, bins=2)
490: (4) array([0. , 2.5, 5. ])
491: (4) For consistency with histogram, an array of pre-computed bins is
492: (4) passed through unmodified:
493: (4) >>> np.histogram_bin_edges(arr, [1, 2])
494: (4) array([1, 2])
495: (4) This function allows one set of bins to be computed, and reused across
496: (4) multiple histograms:
497: (4) >>> shared_bins = np.histogram_bin_edges(arr, bins='auto')
498: (4) >>> shared_bins
499: (4) array([0., 1., 2., 3., 4., 5.])

```

```

500: (4)
501: (4)
502: (4)
503: (4)
504: (4)
505: (4)
506: (4)
507: (4)
508: (4)
509: (4)
510: (4)
511: (4)
512: (4)
513: (4)
514: (4)
515: (4)
516: (4)
517: (4)
518: (4)
519: (4)
520: (0)
521: (8)
522: (4)
523: (0)
524: (0)
525: (4)
526: (4)
527: (4)
528: (4)
529: (4)
530: (8)
531: (4)
532: (8)
533: (8)
534: (8)
535: (8)
536: (8)
537: (8)
538: (8)
539: (4)
540: (8)
541: (8)
542: (8)
543: (8)
544: (8)
545: (8)
546: (8)
547: (4)
548: (8)
549: (8)
550: (8)
551: (8)
552: (8)
553: (4)
554: (8)
555: (8)
556: (8)
557: (8)
558: (8)
559: (8)
560: (4)
561: (4)
562: (4)
563: (8)
564: (8)
565: (4)
566: (8)
567: (4)
568: (4)

    >>> group_id = np.array([0, 1, 1, 0, 1, 1, 0, 1, 1])
    >>> hist_0, _ = np.histogram(arr[group_id == 0], bins=shared_bins)
    >>> hist_1, _ = np.histogram(arr[group_id == 1], bins=shared_bins)
    >>> hist_0; hist_1
    array([1, 1, 0, 1, 0])
    array([2, 0, 1, 1, 2])
    Which gives more easily comparable results than using separate bins for
    each histogram:
    >>> hist_0, bins_0 = np.histogram(arr[group_id == 0], bins='auto')
    >>> hist_1, bins_1 = np.histogram(arr[group_id == 1], bins='auto')
    >>> hist_0; hist_1
    array([1, 1, 1])
    array([2, 1, 1, 2])
    >>> bins_0; bins_1
    array([0., 1., 2., 3.])
    array([0., 1.25, 2.5, 3.75, 5.])
    """
    a, weights = _ravel_and_check_weights(a, weights)
    bin_edges, _ = _get_bin_edges(a, bins, range, weights)
    return bin_edges

def _histogram_dispatcher(
        a, bins=None, range=None, density=None, weights=None):
    return (a, bins, weights)

@array_function_dispatch(_histogram_dispatcher)
def histogram(a, bins=10, range=None, density=None, weights=None):
    """
    Compute the histogram of a dataset.

    Parameters
    -----
    a : array_like
        Input data. The histogram is computed over the flattened array.
    bins : int or sequence of scalars or str, optional
        If `bins` is an int, it defines the number of equal-width
        bins in the given range (10, by default). If `bins` is a
        sequence, it defines a monotonically increasing array of bin edges,
        including the rightmost edge, allowing for non-uniform bin widths.
        .. versionadded:: 1.11.0
        If `bins` is a string, it defines the method used to calculate the
        optimal bin width, as defined by `histogram_bin_edges`.
    range : (float, float), optional
        The lower and upper range of the bins. If not provided, range
        is simply ``(a.min(), a.max())``. Values outside the range are
        ignored. The first element of the range must be less than or
        equal to the second. `range` affects the automatic bin
        computation as well. While bin width is computed to be optimal
        based on the actual data within `range`, the bin count will fill
        the entire range including portions containing no data.
    weights : array_like, optional
        An array of weights, of the same shape as `a`. Each value in
        `a` only contributes its associated weight towards the bin count
        (instead of 1). If `density` is True, the weights are
        normalized, so that the integral of the density over the range
        remains 1.
    density : bool, optional
        If ``False``, the result will contain the number of samples in
        each bin. If ``True``, the result is the value of the
        probability *density* function at the bin, normalized such that
        the *integral* over the range is 1. Note that the sum of the
        histogram values will not be equal to 1 unless bins of unity
        width are chosen; it is not a probability *mass* function.

    Returns
    -----
    hist : array
        The values of the histogram. See `density` and `weights` for a
        description of the possible semantics.
    bin_edges : array of dtype float
        Return the bin edges ``length(hist)+1``.
    See Also
    -----
    """

```

```

569: (4) histogramdd, bincount, searchsorted, digitize, histogram_bin_edges
570: (4) Notes
571: (4)
572: (4) -----
573: (4) All but the last (righthand-most) bin is half-open. In other words,
574: (4) if `bins` is::
575: (4) [1, 2, 3, 4]
576: (4) then the first bin is ``[1, 2)`` (including 1, but excluding 2) and
577: (4) the second ``[2, 3)``. The last bin, however, is ``[3, 4)``, which
578: (4) *includes* 4.
579: (4) Examples
580: (4) -----
581: (4) >>> np.histogram([1, 2, 1], bins=[0, 1, 2, 3])
582: (4) (array([0, 1]), array([0, 1, 2, 3]))
583: (4) >>> np.histogram(np.arange(4), bins=np.arange(5), density=True)
584: (4) (array([0.25, 0.25, 0.25, 0.25]), array([0, 1, 2, 3, 4]))
585: (4) >>> np.histogram([[1, 2, 1], [1, 0, 1]], bins=[0,1,2,3])
586: (4) (array([1, 4, 1]), array([0, 1, 2, 3]))
587: (4) >>> a = np.arange(5)
588: (4) >>> hist, bin_edges = np.histogram(a, density=True)
589: (4) >>> hist
590: (4) array([0.5, 0., 0.5, 0., 0.5, 0., 0.5, 0., 0.5])
591: (4) >>> hist.sum()
592: (4) 2.4999999999999996
593: (4) >>> np.sum(hist * np.diff(bin_edges))
594: (4) 1.0
595: (4) .. versionadded:: 1.11.0
596: (4) Automated Bin Selection Methods example, using 2 peak random data
597: (4) with 2000 points:
598: (4) >>> import matplotlib.pyplot as plt
599: (4) >>> rng = np.random.RandomState(10) # deterministic random data
600: (4) >>> a = np.hstack((rng.normal(size=1000),
601: (4) ...                 rng.normal(loc=5, scale=2, size=1000)))
602: (4) >>> _ = plt.hist(a, bins='auto') # arguments are passed to np.histogram
603: (4) >>> plt.title("Histogram with 'auto' bins")
604: (4) Text(0.5, 1.0, "Histogram with 'auto' bins")
605: (4) >>> plt.show()
606: (4) """
607: (4) a, weights = _ravel_and_check_weights(a, weights)
608: (4) bin_edges, uniform_bins = _get_bin_edges(a, bins, range, weights)
609: (8) if weights is None:
610: (4)     ntype = np.dtype(np.intp)
611: (8) else:
612: (4)     ntype = weights.dtype
613: (4) BLOCK = 65536
614: (8) simple_weights = (
615: (8)     weights is None or
616: (8)     np.can_cast(weights.dtype, np.double) or
617: (8)     np.can_cast(weights.dtype, complex)
618: (4) )
619: (8) if uniform_bins is not None and simple_weights:
620: (8)     first_edge, last_edge, n_equal_bins = uniform_bins
621: (8)     n = np.zeros(n_equal_bins, ntype)
622: (8)     norm_numerator = n_equal_bins
623: (8)     norm_denom = _unsigned_subtract(last_edge, first_edge)
624: (12)    for i in _range(0, len(a), BLOCK):
625: (12)        tmp_a = a[i:i+BLOCK]
626: (16)        if weights is None:
627: (12)            tmp_w = None
628: (16)        else:
629: (12)            tmp_w = weights[i:i + BLOCK]
630: (12)            keep = (tmp_a >= first_edge)
631: (12)            keep &= (tmp_a <= last_edge)
632: (16)            if not np.logical_and.reduce(keep):
633: (16)                tmp_a = tmp_a[keep]
634: (20)                if tmp_w is not None:
635: (12)                    tmp_w = tmp_w[keep]
636: (12)                    tmp_a = tmp_a.astype(bin_edges.dtype, copy=False)
637: (25)                    f_indices = ((_unsigned_subtract(tmp_a, first_edge) / norm_denom)
638: (25)                                * norm_numerator)
```

```

638: (12)             indices = f_indices.astype(np.intp)
639: (12)             indices[indices == n_equal_bins] -= 1
640: (12)             decrement = tmp_a < bin_edges[indices]
641: (12)             indices[decrement] -= 1
642: (12)             increment = ((tmp_a >= bin_edges[indices + 1])
643: (25)                 & (indices != n_equal_bins - 1))
644: (12)             indices[increment] += 1
645: (12)             if ntype.kind == 'c':
646: (16)                 n.real += np.bincount(indices, weights=tmp_w.real,
647: (38)                               minlength=n_equal_bins)
648: (16)                 n.imag += np.bincount(indices, weights=tmp_w.imag,
649: (38)                               minlength=n_equal_bins)
650: (12)             else:
651: (16)                 n += np.bincount(indices, weights=tmp_w,
652: (33)                               minlength=n_equal_bins).astype(ntype)
653: (4)             else:
654: (8)                 cum_n = np.zeros(bin_edges.shape, ntype)
655: (8)                 if weights is None:
656: (12)                     for i in _range(0, len(a), BLOCK):
657: (16)                         sa = np.sort(a[i:i+BLOCK])
658: (16)                         cum_n += _search_sorted_inclusive(sa, bin_edges)
659: (8)             else:
660: (12)                 zero = np.zeros(1, dtype=ntype)
661: (12)                 for i in _range(0, len(a), BLOCK):
662: (16)                     tmp_a = a[i:i+BLOCK]
663: (16)                     tmp_w = weights[i:i+BLOCK]
664: (16)                     sorting_index = np.argsort(tmp_a)
665: (16)                     sa = tmp_a[sorting_index]
666: (16)                     sw = tmp_w[sorting_index]
667: (16)                     cw = np.concatenate((zero, sw.cumsum()))
668: (16)                     bin_index = _search_sorted_inclusive(sa, bin_edges)
669: (16)                     cum_n += cw[bin_index]
670: (8)                 n = np.diff(cum_n)
671: (4)             if density:
672: (8)                 db = np.array(np.diff(bin_edges), float)
673: (8)                 return n/db/n.sum(), bin_edges
674: (4)             return n, bin_edges
675: (0)         def _histogramdd_dispatcher(sample, bins=None, range=None, density=None,
676: (28)                               weights=None):
677: (4)             if hasattr(sample, 'shape'): # same condition as used in histogramdd
678: (8)                 yield sample
679: (4)             else:
680: (8)                 yield from sample
681: (4)             with contextlib.suppress(TypeError):
682: (8)                 yield from bins
683: (4)             yield weights
684: (0)         @array_function_dispatch(_histogramdd_dispatcher)
685: (0)         def histogramdd(sample, bins=10, range=None, density=None, weights=None):
686: (4)             """
687: (4)             Compute the multidimensional histogram of some data.
688: (4)             Parameters
689: (4)             -----
690: (4)             sample : (N, D) array, or (N, D) array_like
691: (8)                 The data to be histogrammed.
692: (8)                 Note the unusual interpretation of sample when an array_like:
693: (8)
694: (10)                 * When an array, each row is a coordinate in a D-dimensional space -
695: (8)                     such as ``histogramdd(np.array([p1, p2, p3]))``.
696: (10)                 * When an array_like, each element is the list of values for single
697: (8)                     coordinate - such as ``histogramdd((X, Y, Z))``.
698: (4)                 The first form should be preferred.
699: (8)             bins : sequence or int, optional
700: (8)                 The bin specification:
701: (10)                 * A sequence of arrays describing the monotonically increasing bin
702: (8)                     edges along each dimension.
703: (8)                 * The number of bins for each dimension (nx, ny, ... =bins)
704: (8)                 * The number of bins for all dimensions (nx=ny=...=bins).
705: (4)             range : sequence, optional
706: (8)                 A sequence of length D, each an optional (lower, upper) tuple giving

```

the outer bin edges to be used if the edges are not given explicitly

```

in
707: (8)           `bins`.
708: (8)           An entry of None in the sequence results in the minimum and maximum
709: (8)           values being used for the corresponding dimension.
710: (8)           The default, None, is equivalent to passing a tuple of D None values.
711: (4)           density : bool, optional
712: (8)           If False, the default, returns the number of samples in each bin.
713: (8)           If True, returns the probability *density* function at the bin,
714: (8)           ``bin_count / sample_count / bin_volume``.
715: (4)           weights : (N,) array_like, optional
716: (8)           An array of values `w_i` weighing each sample `(x_i, y_i, z_i, ...)`.
717: (8)           Weights are normalized to 1 if density is True. If density is False,
718: (8)           the values of the returned histogram are equal to the sum of the
719: (8)           weights belonging to the samples falling into each bin.
720: (4)           Returns
721: (4)           -----
722: (4)           H : ndarray
723: (8)           The multidimensional histogram of sample x. See density and weights
724: (8)           for the different possible semantics.
725: (4)           edges : list
726: (8)           A list of D arrays describing the bin edges for each dimension.
727: (4)           See Also
728: (4)           -----
729: (4)           histogram: 1-D histogram
730: (4)           histogram2d: 2-D histogram
731: (4)           Examples
732: (4)           -----
733: (4)           >>> r = np.random.randn(100,3)
734: (4)           >>> H, edges = np.histogramdd(r, bins = (5, 8, 4))
735: (4)           >>> H.shape, edges[0].size, edges[1].size, edges[2].size
736: (4)           ((5, 8, 4), 6, 9, 5)
737: (4)           """
738: (4)           try:
739: (8)               N, D = sample.shape
740: (4)           except (AttributeError, ValueError):
741: (8)               sample = np.atleast_2d(sample).T
742: (8)               N, D = sample.shape
743: (4)               nbin = np.empty(D, np.intp)
744: (4)               edges = D*[None]
745: (4)               dedges = D*[None]
746: (4)               if weights is not None:
747: (8)                   weights = np.asarray(weights)
748: (4)               try:
749: (8)                   M = len(bins)
750: (8)                   if M != D:
751: (12)                       raise ValueError(
752: (16)                           'The dimension of bins must be equal to the dimension of the '
753: (16)                           'sample x.')
754: (4)               except TypeError:
755: (8)                   bins = D*[bins]
756: (4)               if range is None:
757: (8)                   range = (None,) * D
758: (4)               elif len(range) != D:
759: (8)                   raise ValueError('range argument must have one entry per dimension')
760: (4)               for i in _range(D):
761: (8)                   if np.ndim(bins[i]) == 0:
762: (12)                       if bins[i] < 1:
763: (16)                           raise ValueError(
764: (20)                               '`bins[{}]' must be positive, when an integer'.format(i))
765: (12)                           smin, smax = _get_outer_edges(sample[:,i], range[i])
766: (12)                           try:
767: (16)                               n = operator.index(bins[i])
768: (12)                           except TypeError as e:
769: (16)                               raise TypeError(
770: (20)                                   "`bins[{}]' must be an integer, when a scalar".format(i)
771: (16)                                   ) from e
772: (12)                           edges[i] = np.linspace(smin, smax, n + 1)
773: (8)               elif np.ndim(bins[i]) == 1:
774: (12)                   edges[i] = np.asarray(bins[i])

```

```

775: (12)             if np.any(edges[i][:-1] > edges[i][1:]):
776: (16)                 raise ValueError(
777: (20)                     '`bins[{}]' must be monotonically increasing, when an
array'
778: (20)                         .format(i))
779: (8)             else:
780: (12)                 raise ValueError(
781: (16)                     '`bins[{}]' must be a scalar or 1d array'.format(i))
782: (8)             nbin[i] = len(edges[i]) + 1 # includes an outlier on each end
783: (8)             dedges[i] = np.diff(edges[i])
784: (4)             Ncount = tuple(
785: (8)                 np.searchsorted(edges[i], sample[:, i], side='right')
786: (8)                     for i in _range(D)
787: (4)             )
788: (4)             for i in _range(D):
789: (8)                 on_edge = (sample[:, i] == edges[i][-1])
790: (8)                 Ncount[i][on_edge] -= 1
791: (4)             xy = np.ravel_multi_index(Ncount, nbin)
792: (4)             hist = np.bincount(xy, weights, minlength=nbin.prod())
793: (4)             hist = hist.reshape(nbin)
794: (4)             hist = hist.astype(float, casting='safe')
795: (4)             core = D*(slice(1, -1),)
796: (4)             hist = hist[core]
797: (4)             if density:
798: (8)                 s = hist.sum()
799: (8)                 for i in _range(D):
800: (12)                     shape = np.ones(D, int)
801: (12)                     shape[i] = nbin[i] - 2
802: (12)                     hist = hist / dedges[i].reshape(shape)
803: (8)                     hist /= s
804: (4)             if (hist.shape != nbin - 2).any():
805: (8)                 raise RuntimeError(
806: (12)                     "Internal Shape Error")
807: (4)             return hist, edges
-----
```

## File 210 - index\_tricks.py:

```

1: (0)         import functools
2: (0)         import sys
3: (0)         import math
4: (0)         import warnings
5: (0)         import numpy as np
6: (0)         from .._utils import set_module
7: (0)         import numpy.core.numeric as _nx
8: (0)         from numpy.core.numeric import ScalarType, array
9: (0)         from numpy.core.numerictypes import issubdtype
10: (0)        import numpy.matrixlib as matrixlib
11: (0)        from .function_base import diff
12: (0)        from numpy.core.multiarray import ravel_multi_index, unravel_index
13: (0)        from numpy.core import overrides, linspace
14: (0)        from numpy.lib.stride_tricks import as_strided
15: (0)        array_function_dispatch = functools.partial(
16: (4)            overrides.array_function_dispatch, module='numpy')
17: (0)        __all__ = [
18: (4)            'ravel_multi_index', 'unravel_index', 'mgrid', 'ogrid', 'r_', 'c_',
19: (4)            's_', 'index_exp', 'ix_', 'ndenumerate', 'ndindex', 'fill_diagonal',
20: (4)            'diag_indices', 'diag_indices_from'
21: (0)        ]
22: (0)        def __ix_dispatcher(*args):
23: (4)            return args
24: (0)        @array_function_dispatch(__ix_dispatcher)
25: (0)        def ix_(*args):
26: (4)            """
27: (4)                Construct an open mesh from multiple sequences.
28: (4)                This function takes N 1-D sequences and returns N outputs with N
dimensions each, such that the shape is 1 in all but one dimension
and the dimension with the non-unit shape value cycles through all
30: (4)            """

```

```

31: (4)           N dimensions.
32: (4)           Using `ix_` one can quickly construct index arrays that will index
33: (4)           the cross product. ``a[np.ix_([1,3],[2,5])]`` returns the array
34: (4)           ``[[a[1,2] a[1,5]], [a[3,2] a[3,5]]]``.
35: (4)           Parameters
36: (4)           -----
37: (4)           args : 1-D sequences
38: (8)             Each sequence should be of integer or boolean type.
39: (8)             Boolean sequences will be interpreted as boolean masks for the
40: (8)             corresponding dimension (equivalent to passing in
41: (8)             ``np.nonzero(boolean_sequence)``).
42: (4)           Returns
43: (4)           -----
44: (4)           out : tuple of ndarrays
45: (8)             N arrays with N dimensions each, with N the number of input
46: (8)             sequences. Together these arrays form an open mesh.
47: (4)           See Also
48: (4)           -----
49: (4)           ogrid, mgrid, meshgrid
50: (4)           Examples
51: (4)           -----
52: (4)           >>> a = np.arange(10).reshape(2, 5)
53: (4)
54: (4)           array([[0, 1, 2, 3, 4],
55: (11)             [5, 6, 7, 8, 9]])
56: (4)           >>> ixgrid = np.ix_([0, 1], [2, 4])
57: (4)
58: (4)           >>> ixgrid
59: (11)             (array([[0],
60: (4)               [1]]), array([[2, 4]]))
61: (4)           >>> ixgrid[0].shape, ixgrid[1].shape
62: (4)             ((2, 1), (1, 2))
63: (4)           >>> a[ixgrid]
64: (11)             array([[2, 4],
65: (4)               [7, 9]])
66: (4)           >>> ixgrid = np.ix_([True, True], [2, 4])
67: (4)           >>> a[ixgrid]
68: (11)             array([[2, 4],
69: (4)               [7, 9]])
70: (4)           >>> ixgrid = np.ix_([True, True], [False, False, True, False, True])
71: (4)           >>> a[ixgrid]
72: (11)             array([[2, 4],
73: (4)               [7, 9]])
74: (4)           """
75: (4)           out = []
76: (4)           nd = len(args)
77: (8)           for k, new in enumerate(args):
78: (12)             if not isinstance(new, _nx.ndarray):
79: (12)               new = np.asarray(new)
80: (16)             if new.size == 0:
81: (8)               new = new.astype(_nx.intp)
82: (12)             if new.ndim != 1:
83: (8)               raise ValueError("Cross index must be 1 dimensional")
84: (12)             if issubdtype(new.dtype, _nx.bool_):
85: (8)               new, = new.nonzero()
86: (8)             new = new.reshape((1,) * k + (new.size,) + (1,) * (nd - k - 1))
87: (4)             out.append(new)
88: (0)           return tuple(out)
89: (4)           class nd_grid:
90: (4)             """
91: (4)             Construct a multi-dimensional "meshgrid".
92: (4)             ``grid = nd_grid()`` creates an instance which will return a mesh-grid
93: (4)             when indexed. The dimension and number of the output arrays are equal
94: (4)             to the number of indexing dimensions. If the step length is not a
95: (4)             complex number, then the stop is not inclusive.
96: (4)             However, if the step length is a **complex number** (e.g. 5j), then the
97: (4)             integer part of its magnitude is interpreted as specifying the
98: (4)             number of points to create between the start and stop values, where
99: (4)             the stop value **is inclusive**.

```

If instantiated with an argument of ``sparse=True``, the mesh-grid is

```

100: (4)          open (or not fleshed out) so that only one-dimension of each returned
101: (4)          argument is greater than 1.
102: (4)          Parameters
103: (4)          -----
104: (4)          sparse : bool, optional
105: (8)          Whether the grid is sparse or not. Default is False.
106: (4)          Notes
107: (4)          -----
108: (4)          Two instances of `nd_grid` are made available in the NumPy namespace,
109: (4)          `mgrid` and `ogrid`, approximately defined as::
110: (8)          mgrid = nd_grid(sparse=False)
111: (8)          ogrid = nd_grid(sparse=True)
112: (4)          Users should use these pre-defined instances instead of using `nd_grid`
113: (4)          directly.
114: (4)          """
115: (4)          def __init__(self, sparse=False):
116: (8)          self.sparse = sparse
117: (4)          def __getitem__(self, key):
118: (8)          try:
119: (12)              size = []
120: (12)              num_list = [0]
121: (12)              for k in range(len(key)):
122: (16)                  step = key[k].step
123: (16)                  start = key[k].start
124: (16)                  stop = key[k].stop
125: (16)                  if start is None:
126: (20)                      start = 0
127: (16)                  if step is None:
128: (20)                      step = 1
129: (16)                  if isinstance(step, (_nx.complexfloating, complex)):
130: (20)                      step = abs(step)
131: (20)                      size.append(int(step))
132: (16)                  else:
133: (20)                      size.append(
134: (24)                          int(math.ceil((stop - start) / (step*1.0))))
135: (16)                      num_list += [start, stop, step]
136: (12)              typ = _nx.result_type(*num_list)
137: (12)              if self.sparse:
138: (16)                  nn = [_nx.arange(_x, dtype=_t)
139: (22)                      for _x, _t in zip(size, (typ,) * len(size)))]
140: (12)              else:
141: (16)                  nn = _nx.indices(size, typ)
142: (12)                  for k, kk in enumerate(key):
143: (16)                      step = kk.step
144: (16)                      start = kk.start
145: (16)                      if start is None:
146: (20)                          start = 0
147: (16)                      if step is None:
148: (20)                          step = 1
149: (16)                      if isinstance(step, (_nx.complexfloating, complex)):
150: (20)                          step = int(abs(step))
151: (20)                          if step != 1:
152: (24)                              step = (kk.stop - start) / float(step - 1)
153: (16)                          nn[k] = (nn[k]*step+start)
154: (12)              if self.sparse:
155: (16)                  slobj = [_nx.newaxis]*len(size)
156: (16)                  for k in range(len(size)):
157: (20)                      slobj[k] = slice(None, None)
158: (20)                      nn[k] = nn[k][tuple(slobj)]
159: (20)                      slobj[k] = _nx.newaxis
160: (12)                  return nn
161: (8)          except (IndexError, TypeError):
162: (12)              step = key.step
163: (12)              stop = key.stop
164: (12)              start = key.start
165: (12)              if start is None:
166: (16)                  start = 0
167: (12)              if isinstance(step, (_nx.complexfloating, complex)):
168: (16)                  step_float = abs(step)

```

```

169: (16)                         step = length = int(step_float)
170: (16)                         if step != 1:
171: (20)                             step = (key.stop-start)/float(step-1)
172: (16)                             typ = _nx.result_type(start, stop, step_float)
173: (16)                             return _nx.arange(0, length, 1, dtype=typ)*step + start
174: (12)                         else:
175: (16)                             return _nx.arange(start, stop, step)
176: (0) class MGridClass(nd_grid):
177: (4) """
178: (4)     An instance which returns a dense multi-dimensional "meshgrid".
179: (4)     An instance which returns a dense (or fleshed out) mesh-grid
180: (4)     when indexed, so that each returned argument has the same shape.
181: (4)     The dimensions and number of the output arrays are equal to the
182: (4)     number of indexing dimensions. If the step length is not a complex
183: (4)     number, then the stop is not inclusive.
184: (4)     However, if the step length is a **complex number** (e.g. 5j), then
185: (4)     the integer part of its magnitude is interpreted as specifying the
186: (4)     number of points to create between the start and stop values, where
187: (4)     the stop value **is inclusive**.
188: (4) Returns
189: (4) -----
190: (4) mesh-grid `ndarrays` all of the same dimensions
191: (4) See Also
192: (4) -----
193: (4) ogrid : like `mgrid` but returns open (not fleshed out) mesh grids
194: (4) meshgrid: return coordinate matrices from coordinate vectors
195: (4) r_ : array concatenator
196: (4) :ref:`how-to-partition`
197: (4) Examples
198: (4) -----
199: (4) >>> np.mgrid[0:5, 0:5]
200: (4) array([[[0, 0, 0, 0, 0],
201: (12)             [1, 1, 1, 1, 1],
202: (12)             [2, 2, 2, 2, 2],
203: (12)             [3, 3, 3, 3, 3],
204: (12)             [4, 4, 4, 4, 4]],
205: (11)             [[0, 1, 2, 3, 4],
206: (12)             [0, 1, 2, 3, 4],
207: (12)             [0, 1, 2, 3, 4],
208: (12)             [0, 1, 2, 3, 4],
209: (12)             [0, 1, 2, 3, 4]])
210: (4) >>> np.mgrid[-1:1:5j]
211: (4) array([-1. , -0.5,  0. ,  0.5,  1. ])
212: (4) """
213: (4) def __init__(self):
214: (8)     super().__init__(sparse=False)
215: (0) mgrid = MGridClass()
216: (0) class OGridClass(nd_grid):
217: (4) """
218: (4)     An instance which returns an open multi-dimensional "meshgrid".
219: (4)     An instance which returns an open (i.e. not fleshed out) mesh-grid
220: (4)     when indexed, so that only one dimension of each returned array is
221: (4)     greater than 1. The dimension and number of the output arrays are
222: (4)     equal to the number of indexing dimensions. If the step length is
223: (4)     not a complex number, then the stop is not inclusive.
224: (4)     However, if the step length is a **complex number** (e.g. 5j), then
225: (4)     the integer part of its magnitude is interpreted as specifying the
226: (4)     number of points to create between the start and stop values, where
227: (4)     the stop value **is inclusive**.
228: (4) Returns
229: (4) -----
230: (4) mesh-grid
231: (8)     `ndarrays` with only one dimension not equal to 1
232: (4) See Also
233: (4) -----
234: (4) mgrid : like `ogrid` but returns dense (or fleshed out) mesh grids
235: (4) meshgrid: return coordinate matrices from coordinate vectors
236: (4) r_ : array concatenator
237: (4) :ref:`how-to-partition`
```

```

238: (4)          Examples
239: (4)          -----
240: (4)          >>> from numpy import ogrid
241: (4)          >>> ogrid[-1:1:5j]
242: (4)          array([-1. , -0.5,  0. ,  0.5,  1. ])
243: (4)          >>> ogrid[0:5,0:5]
244: (4)          [array([[0],
245: (12)           [1],
246: (12)           [2],
247: (12)           [3],
248: (12)           [4]]), array([[0, 1, 2, 3, 4]])]
249: (4)          """
250: (4)          def __init__(self):
251: (8)            super().__init__(sparse=True)
252: (0)          ogrid = OGridClass()
253: (0)          class AxisConcatenator:
254: (4)            """
255: (4)            Translates slice objects to concatenation along an axis.
256: (4)            For detailed documentation on usage, see `r_`.
257: (4)            """
258: (4)            concatenate = staticmethod(_nx.concatenate)
259: (4)            makemat = staticmethod(matrixlib.matrix)
260: (4)            def __init__(self, axis=0, matrix=False, ndmin=1, trans1d=-1):
261: (8)              self.axis = axis
262: (8)              self.matrix = matrix
263: (8)              self.trans1d = trans1d
264: (8)              self.ndmin = ndmin
265: (4)            def __getitem__(self, key):
266: (8)              if isinstance(key, str):
267: (12)                frame = sys._getframe().f_back
268: (12)                mymat = matrixlib.bmat(key, frame.f_globals, frame.f_locals)
269: (12)                return mymat
270: (8)              if not isinstance(key, tuple):
271: (12)                key = (key,)
272: (8)              trans1d = self.trans1d
273: (8)              ndmin = self.ndmin
274: (8)              matrix = self.matrix
275: (8)              axis = self.axis
276: (8)              objs = []
277: (8)              result_type_objs = []
278: (8)              for k, item in enumerate(key):
279: (12)                scalar = False
280: (12)                if isinstance(item, slice):
281: (16)                  step = item.step
282: (16)                  start = item.start
283: (16)                  stop = item.stop
284: (16)                  if start is None:
285: (20)                    start = 0
286: (16)                  if step is None:
287: (20)                    step = 1
288: (16)                  if isinstance(step, (_nx.complexfloating, complex)):
289: (20)                    size = int(abs(step))
290: (20)                    newobj = linspace(start, stop, num=size)
291: (16)                  else:
292: (20)                    newobj = _nx.arange(start, stop, step)
293: (16)                  if ndmin > 1:
294: (20)                    newobj = array(newobj, copy=False, ndmin=ndmin)
295: (20)                    if trans1d != -1:
296: (24)                      newobj = newobj.swapaxes(-1, trans1d)
297: (12)                  elif isinstance(item, str):
298: (16)                    if k != 0:
299: (20)                      raise ValueError("special directives must be the "
300: (37)                          "first entry.")
301: (16)                    if item in ('r', 'c'):
302: (20)                      matrix = True
303: (20)                      col = (item == 'c')
304: (20)                      continue
305: (16)                      if ',' in item:
306: (20)                        vec = item.split(',')

```

```

307: (20)
308: (24)
309: (24)
310: (28)
311: (24)
312: (20)
313: (24)
314: (28)
315: (24)
316: (16)
317: (20)
318: (20)
319: (16)
320: (20)
321: (12)
322: (16)
323: (16)
324: (12)
325: (16)
326: (16)
327: (16)
328: (20)
329: (20)
330: (20)
331: (24)
332: (20)
333: (20)
334: (20)
335: (12)
336: (12)
337: (16)
338: (12)
339: (16)
340: (8)
341: (12)
342: (12)
343: (26)
344: (8)
345: (8)
346: (12)
347: (12)
348: (12)
349: (16)
350: (8)
351: (4)
352: (8)
353: (0)
354: (4)
355: (4)
356: (4)
357: (4)
358: (7)
359: (4)
360: (7)
361: (4)
362: (4)
363: (4)
364: (4)
365: (4)
366: (4)
367: (4)
368: (4)
369: (4)
370: (4)
371: (4)
372: (4)
1
373: (4)
374: (4)

try:
    axis, ndmin = [int(x) for x in vec[:2]]
    if len(vec) == 3:
        trans1d = int(vec[2])
    continue
except Exception as e:
    raise ValueError(
        "unknown special directive {!r}".format(item)
    ) from e
try:
    axis = int(item)
    continue
except (ValueError, TypeError) as e:
    raise ValueError("unknown special directive") from e
elif type(item) in ScalarType:
    scalar = True
    newobj = item
else:
    item_ndim = np.ndim(item)
    newobj = array(item, copy=False, subok=True, ndmin=ndmin)
    if trans1d != -1 and item_ndim < ndmin:
        k2 = ndmin - item_ndim
        k1 = trans1d
        if k1 < 0:
            k1 += k2 + 1
        defaxes = list(range(ndmin))
        axes = defaxes[:k1] + defaxes[k2:] + defaxes[k1:k2]
        newobj = newobj.transpose(axes)
    objs.append(newobj)
    if scalar:
        result_type_objs.append(item)
    else:
        result_type_objs.append(newobj.dtype)
if len(result_type_objs) != 0:
    final_dtype = _nx.result_type(*result_type_objs)
    objs = [array(obj, copy=False, subok=True,
                  ndmin=ndmin, dtype=final_dtype) for obj in objs]
res = self.concatenate(tuple(objs), axis=axis)
if matrix:
    oldndim = res.ndim
    res = self.makemat(res)
    if oldndim == 1 and col:
        res = res.T
    return res
def __len__(self):
    return 0
class RClass(AxisConcatenator):
    """
    Translates slice objects to concatenation along the first axis.
    This is a simple way to build up arrays quickly. There are two use cases.
    1. If the index expression contains comma separated arrays, then stack
       them along their first axis.
    2. If the index expression contains slice notation or scalars then create
       a 1-D array with a range indicated by the slice notation.
    If slice notation is used, the syntax ``start:stop:step`` is equivalent
    to ``np.arange(start, stop, step)`` inside of the brackets. However, if
    ``step`` is an imaginary number (i.e. 100j) then its integer portion is
    interpreted as a number-of-points desired and the start and stop are
    inclusive. In other words ``start:stop:stepj`` is interpreted as
    ``np.linspace(start, stop, step, endpoint=1)`` inside of the brackets.
    After expansion of slice notation, all comma separated sequences are
    concatenated together.
    Optional character strings placed as the first element of the index
    expression can be used to change the output. The strings 'r' or 'c' result
    in matrix output. If the result is 1-D and 'r' is specified a 1 x N (row)
    matrix is produced. If the result is 1-D and 'c' is specified, then a N x
    (column) matrix is produced. If the result is 2-D then both provide the
    same matrix result.
    """

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

375: (4) A string integer specifies which axis to stack multiple comma separated
376: (4) arrays along. A string of two comma-separated integers allows indication
377: (4) of the minimum number of dimensions to force each entry into as the
378: (4) second integer (the axis to concatenate along is still the first integer).
379: (4) A string with three comma-separated integers allows specification of the
380: (4) axis to concatenate along, the minimum number of dimensions to force the
381: (4) entries to, and which axis should contain the start of the arrays which
382: (4) are less than the specified number of dimensions. In other words the third
383: (4) integer allows you to specify where the 1's should be placed in the shape
384: (4) of the arrays that have their shapes upgraded. By default, they are placed
385: (4) in the front of the shape tuple. The third argument allows you to specify
386: (4) where the start of the array should be instead. Thus, a third argument of
387: (4) '0' would place the 1's at the end of the array shape. Negative integers
388: (4) specify where in the new shape tuple the last dimension of upgraded arrays
389: (4) should be placed, so the default is '-1'.
390: (4) Parameters
391: (4) -----
392: (4) Not a function, so takes no parameters
393: (4) Returns
394: (4) -----
395: (4) A concatenated ndarray or matrix.
396: (4) See Also
397: (4) -----
398: (4) concatenate : Join a sequence of arrays along an existing axis.
399: (4) c_ : Translates slice objects to concatenation along the second axis.
400: (4) Examples
401: (4) -----
402: (4) >>> np.r_[np.array([1,2,3]), 0, 0, np.array([4,5,6])]
403: (4) array([1, 2, 3, ..., 4, 5, 6])
404: (4) >>> np.r_[-1:1:6j, [0]*3, 5, 6]
405: (4) array([-1. , -0.6, -0.2,  0.2,  0.6,  1. ,  0. ,  0. ,  5. ,  6. ])
406: (4) String integers specify the axis to concatenate along or the minimum
407: (4) number of dimensions to force entries into.
408: (4) >>> a = np.array([[0, 1, 2], [3, 4, 5]])
409: (4) >>> np.r_['-1', a, a] # concatenate along last axis
410: (4) array([[0, 1, 2, 0, 1, 2],
411: (11)      [3, 4, 5, 3, 4, 5]])
412: (4) >>> np.r_['0,2', [1,2,3], [4,5,6]] # concatenate along first axis, dim>=2
413: (4) array([[1, 2, 3],
414: (11)      [4, 5, 6]])
415: (4) >>> np.r_['0,2,0', [1,2,3], [4,5,6]]
416: (4) array([[1],
417: (11)      [2],
418: (11)      [3],
419: (11)      [4],
420: (11)      [5],
421: (11)      [6]])
422: (4) >>> np.r_['1,2,0', [1,2,3], [4,5,6]]
423: (4) array([[1, 4],
424: (11)      [2, 5],
425: (11)      [3, 6]])
426: (4) Using 'r' or 'c' as a first string argument creates a matrix.
427: (4) >>> np.r_['r',[1,2,3], [4,5,6]]
428: (4) matrix([[1, 2, 3, 4, 5, 6]])
429: (4) """
430: (4)     def __init__(self):
431: (8)         AxisConcatenator.__init__(self, 0)
432: (0)     r_ = RClass()
433: (0)     class CClass(AxisConcatenator):
434: (4)         """
435: (4)             Translates slice objects to concatenation along the second axis.
436: (4)             This is short-hand for ``np.r_['-1,2,0', index expression]``, which is
437: (4)             useful because of its common occurrence. In particular, arrays will be
438: (4)             stacked along their last axis after being upgraded to at least 2-D with
439: (4)             1's post-pended to the shape (column vectors made out of 1-D arrays).
440: (4)             See Also
441: (4)             -----
442: (4)             column_stack : Stack 1-D arrays as columns into a 2-D array.
443: (4)             r_ : For more detailed documentation.

```

```

444: (4) Examples
445: (4) -----
446: (4) >>> np.c_[np.array([1,2,3]), np.array([4,5,6])]
447: (4) array([[1, 4],
448: (4)         [2, 5],
449: (4)         [3, 6]])
450: (4) >>> np.c_[np.array([[1,2,3]]), 0, 0, np.array([[4,5,6]])]
451: (4) array([[1, 2, 3, ..., 4, 5, 6]])
452: (4) """
453: (4)     def __init__(self):
454: (8)         AxisConcatenator.__init__(self, -1, ndmin=2, trans1d=0)
455: (0) c_ = CClass()
456: (0) @set_module('numpy')
457: (0) class ndenumerate:
458: (4)     """
459: (4)     Multidimensional index iterator.
460: (4)     Return an iterator yielding pairs of array coordinates and values.
461: (4)     Parameters
462: (4)     -----
463: (4)     arr : ndarray
464: (6)         Input array.
465: (4)     See Also
466: (4)     -----
467: (4)     ndindex, flatiter
468: (4)     Examples
469: (4)     -----
470: (4) >>> a = np.array([[1, 2], [3, 4]])
471: (4) >>> for index, x in np.ndenumerate(a):
472: (4) ...     print(index, x)
473: (4) (0, 0) 1
474: (4) (0, 1) 2
475: (4) (1, 0) 3
476: (4) (1, 1) 4
477: (4) """
478: (4)     def __init__(self, arr):
479: (8)         self.iter = np.asarray(arr).flat
480: (4)     def __next__(self):
481: (8)     """
482: (8)         Standard iterator method, returns the index tuple and array value.
483: (8)     Returns
484: (8)     -----
485: (8)     coords : tuple of ints
486: (12)         The indices of the current iteration.
487: (8)     val : scalar
488: (12)         The array element of the current iteration.
489: (8)     """
490: (8)         return self.iter.coords, next(self.iter)
491: (4)     def __iter__(self):
492: (8)         return self
493: (0) @set_module('numpy')
494: (0)
495: (4)     """
496: (4)     An N-dimensional iterator object to index arrays.
497: (4)     Given the shape of an array, an `ndindex` instance iterates over
498: (4)     the N-dimensional index of the array. At each iteration a tuple
499: (4)     of indices is returned, the last dimension is iterated over first.
500: (4)     Parameters
501: (4)     -----
502: (4)     shape : ints, or a single tuple of ints
503: (8)         The size of each dimension of the array can be passed as
504: (8)             individual parameters or as the elements of a tuple.
505: (4)     See Also
506: (4)     -----
507: (4)     ndenumerate, flatiter
508: (4)     Examples
509: (4)     -----
510: (4)     Dimensions as individual arguments
511: (4) >>> for index in np.ndindex(3, 2, 1):
512: (4) ...     print(index)

```

```

513: (4)           (0, 0, 0)
514: (4)           (0, 1, 0)
515: (4)           (1, 0, 0)
516: (4)           (1, 1, 0)
517: (4)           (2, 0, 0)
518: (4)           (2, 1, 0)
519: (4)           Same dimensions - but in a tuple ``(3, 2, 1)``
520: (4)           >>> for index in np.ndindex((3, 2, 1)):
521: (4)             ...     print(index)
522: (4)           (0, 0, 0)
523: (4)           (0, 1, 0)
524: (4)           (1, 0, 0)
525: (4)           (1, 1, 0)
526: (4)           (2, 0, 0)
527: (4)           (2, 1, 0)
528: (4)           """
529: (4)           def __init__(self, *shape):
530: (8)             if len(shape) == 1 and isinstance(shape[0], tuple):
531: (12)               shape = shape[0]
532: (8)             x = as_strided(_nx.zeros(1), shape=shape,
533: (23)                           strides=_nx.zeros_like(shape))
534: (8)             self._it = _nx.nditer(x, flags=['multi_index', 'zerosize_ok'],
535: (30)                           order='C')
536: (4)           def __iter__(self):
537: (8)             return self
538: (4)           def ndincr(self):
539: (8)             """
540: (8)             Increment the multi-dimensional index by one.
541: (8)             This method is for backward compatibility only: do not use.
542: (8)             .. deprecated:: 1.20.0
543: (12)               This method has been advised against since numpy 1.8.0, but only
544: (12)               started emitting DeprecationWarning as of this version.
545: (8)             """
546: (8)             warnings.warn(
547: (12)               "`ndindex.ndincr()` is deprecated, use `next(ndindex)` instead",
548: (12)               DeprecationWarning, stacklevel=2)
549: (8)             next(self)
550: (4)           def __next__(self):
551: (8)             """
552: (8)             Standard iterator method, updates the index and returns the index
553: (8)             tuple.
554: (8)             Returns
555: (8)             -----
556: (8)             val : tuple of ints
557: (12)               Returns a tuple containing the indices of the current
558: (12)               iteration.
559: (8)             """
560: (8)             next(self._it)
561: (8)             return self._it.multi_index
562: (0)           class IndexExpression:
563: (4)             """
564: (4)               A nicer way to build up index tuples for arrays.
565: (4)               .. note::
566: (7)                 Use one of the two predefined instances `index_exp` or `s_`
567: (7)                 rather than directly using `IndexExpression`.
568: (4)               For any index combination, including slicing and axis insertion,
569: (4)               ``a[indices]`` is the same as ``a[np.index_exp[indices]]`` for any
570: (4)               array `a`. However, ``np.index_exp[indices]`` can be used anywhere
571: (4)               in Python code and returns a tuple of slice objects that can be
572: (4)               used in the construction of complex index expressions.
573: (4)               Parameters
574: (4)               -----
575: (4)               maketuple : bool
576: (8)                 If True, always returns a tuple.
577: (4)               See Also
578: (4)               -----
579: (4)               index_exp : Predefined instance that always returns a tuple:
580: (7)                 `index_exp = IndexExpression(maketuple=True)` .
581: (4)               s_ : Predefined instance without tuple conversion:

```

```

582: (7)           `s_ = IndexExpression(maketuple=False)` .
583: (4)           Notes
584: (4)
585: (4)           -----
586: (4)           You can do all this with `slice()` plus a few special objects,
587: (4)           but there's a lot to remember and this version is simpler because
588: (4)           it uses the standard array indexing syntax.
589: (4)           Examples
590: (4)           -----
591: (4)           >>> np.s_[2::2]
592: (4)           slice(2, None, 2)
593: (4)           >>> np.index_exp[2::2]
594: (4)           (slice(2, None, 2),)
595: (4)           >>> np.array([0, 1, 2, 3, 4])[np.s_[2::2]]
596: (4)           array([2, 4])
597: (4)           """
598: (8)           def __init__(self, maketuple):
599: (4)               self.maketuple = maketuple
600: (8)           def __getitem__(self, item):
601: (12)             if self.maketuple and not isinstance(item, tuple):
602: (8)                 return (item,)
603: (12)             else:
604: (0)                 return item
605: (0)           index_exp = IndexExpression(maketuple=True)
606: (0)           s_ = IndexExpression(maketuple=False)
607: (4)           def _fill_diagonal_dispatcher(a, val, wrap=None):
608: (4)             return (a,)
609: (0)           @array_function_dispatch(_fill_diagonal_dispatcher)
610: (0)           def fill_diagonal(a, val, wrap=False):
611: (4)             """Fill the main diagonal of the given array of any dimensionality.
612: (4)             For an array `a` with ``a.ndim >= 2``, the diagonal is the list of
613: (4)             locations with indices ``a[i, ..., i]`` all identical. This function
614: (4)             modifies the input array in-place, it does not return a value.
615: (4)             Parameters
616: (4)             -----
617: (6)               a : array, at least 2-D.
618: (4)               Array whose diagonal is to be filled, it gets modified in-place.
619: (6)               val : scalar or array_like
620: (6)               Value(s) to write on the diagonal. If `val` is scalar, the value is
621: (6)               written along the diagonal. If array-like, the flattened `val` is
622: (6)               written along the diagonal, repeating if necessary to fill all
623: (4)               diagonal entries.
624: (6)               wrap : bool
625: (6)               For tall matrices in NumPy version up to 1.6.2, the
626: (6)               diagonal "wrapped" after N columns. You can have this behavior
627: (4)               with this option. This affects only tall matrices.
628: (4)           See also
629: (4)           -----
630: (4)           diag_indices, diag_indices_from
631: (4)           Notes
632: (4)           -----
633: (4)           .. versionadded:: 1.4.0
634: (4)           This functionality can be obtained via `diag_indices`, but internally
635: (4)           this version uses a much faster implementation that never constructs the
636: (4)           indices and uses simple slicing.
637: (4)           Examples
638: (4)           -----
639: (4)           >>> a = np.zeros((3, 3), int)
640: (4)           >>> np.fill_diagonal(a, 5)
641: (4)           >>> a
642: (11)          array([[5, 0, 0],
643: (11)                      [0, 5, 0],
644: (4)                      [0, 0, 5]])
645: (4)           The same function can operate on a 4-D array:
646: (4)           >>> a = np.zeros((3, 3, 3, 3), int)
647: (4)           >>> np.fill_diagonal(a, 4)
648: (4)           We only show a few blocks for clarity:
649: (4)           >>> a[0, 0]
650: (11)          array([[4, 0, 0],
651: (11)                      [0, 0, 0],
652: (11)                      [0, 0, 0]])

```

```

651: (11)          [0, 0, 0]])]
652: (4)          >>> a[1, 1]
653: (4)          array([[0, 0, 0],
654: (11)          [0, 4, 0],
655: (11)          [0, 0, 0]])
656: (4)          >>> a[2, 2]
657: (4)          array([[0, 0, 0],
658: (11)          [0, 0, 0],
659: (11)          [0, 0, 4]])
660: (4)          The wrap option affects only tall matrices:
661: (4)          >>> # tall matrices no wrap
662: (4)          >>> a = np.zeros((5, 3), int)
663: (4)          >>> np.fill_diagonal(a, 4)
664: (4)          >>> a
665: (4)          array([[4, 0, 0],
666: (11)          [0, 4, 0],
667: (11)          [0, 0, 4],
668: (11)          [0, 0, 0],
669: (11)          [0, 0, 0]])
670: (4)          >>> # tall matrices wrap
671: (4)          >>> a = np.zeros((5, 3), int)
672: (4)          >>> np.fill_diagonal(a, 4, wrap=True)
673: (4)          >>> a
674: (4)          array([[4, 0, 0],
675: (11)          [0, 4, 0],
676: (11)          [0, 0, 4],
677: (11)          [0, 0, 0],
678: (11)          [4, 0, 0]])
679: (4)          >>> # wide matrices
680: (4)          >>> a = np.zeros((3, 5), int)
681: (4)          >>> np.fill_diagonal(a, 4, wrap=True)
682: (4)          >>> a
683: (4)          array([[4, 0, 0, 0, 0],
684: (11)          [0, 4, 0, 0, 0],
685: (11)          [0, 0, 4, 0, 0]])
686: (4)          The anti-diagonal can be filled by reversing the order of elements
687: (4)          using either `numpy.flipud` or `numpy.fliplr`.
688: (4)          >>> a = np.zeros((3, 3), int);
689: (4)          >>> np.fill_diagonal(np.fliplr(a), [1,2,3]) # Horizontal flip
690: (4)          >>> a
691: (4)          array([[0, 0, 1],
692: (11)          [0, 2, 0],
693: (11)          [3, 0, 0]])
694: (4)          >>> np.fill_diagonal(np.flipud(a), [1,2,3]) # Vertical flip
695: (4)          >>> a
696: (4)          array([[0, 0, 3],
697: (11)          [0, 2, 0],
698: (11)          [1, 0, 0]])
699: (4)          Note that the order in which the diagonal is filled varies depending
700: (4)          on the flip function.
701: (4)          """
702: (4)          if a.ndim < 2:
703: (8)              raise ValueError("array must be at least 2-d")
704: (4)          end = None
705: (4)          if a.ndim == 2:
706: (8)              step = a.shape[1] + 1
707: (8)              if not wrap:
708: (12)                  end = a.shape[1] * a.shape[1]
709: (4)          else:
710: (8)              if not np.all(diff(a.shape) == 0):
711: (12)                  raise ValueError("All dimensions of input must be of equal
length")
712: (8)              step = 1 + (np.cumprod(a.shape[:-1])).sum()
713: (4)              a.flat[:end:step] = val
714: (0)          @set_module('numpy')
715: (0)          def diag_indices(n, ndim=2):
716: (4)          """
717: (4)          Return the indices to access the main diagonal of an array.
718: (4)          This returns a tuple of indices that can be used to access the main

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

719: (4)           diagonal of an array `a` with ``a.ndim >= 2`` dimensions and shape
720: (4)           (n, n, ..., n). For ``a.ndim = 2`` this is the usual diagonal, for
721: (4)           ``a.ndim > 2`` this is the set of indices to access ``a[i, i, ..., i]``.
722: (4)
723: (4)           Parameters
724: (4)           -----
725: (4)           n : int
726: (6)           The size, along each dimension, of the arrays for which the returned
727: (6)           indices can be used.
728: (4)           ndim : int, optional
729: (6)           The number of dimensions.
730: (4)           See Also
731: (4)           -----
732: (4)           diag_indices_from
733: (4)           Notes
734: (4)           -----
735: (4)           .. versionadded:: 1.4.0
736: (4)           Examples
737: (4)           -----
738: (4)           Create a set of indices to access the diagonal of a (4, 4) array:
739: (4)           >>> di = np.diag_indices(4)
740: (4)           >>> di
741: (4)           (array([0, 1, 2, 3]), array([0, 1, 2, 3]))
742: (4)           >>> a = np.arange(16).reshape(4, 4)
743: (4)           >>> a
744: (4)           array([[ 0,  1,  2,  3],
745: (11)             [ 4,  5,  6,  7],
746: (11)             [ 8,  9, 10, 11],
747: (11)             [12, 13, 14, 15]])
748: (4)           >>> a[di] = 100
749: (4)           >>> a
750: (4)           array([[100,  1,  2,  3],
751: (11)             [ 4, 100,  6,  7],
752: (11)             [ 8,  9, 100, 11],
753: (11)             [12, 13, 14, 100]])
754: (4)           Now, we create indices to manipulate a 3-D array:
755: (4)           >>> d3 = np.diag_indices(2, 3)
756: (4)           >>> d3
757: (4)           (array([0, 1]), array([0, 1]), array([0, 1]))
758: (4)           And use it to set the diagonal of an array of zeros to 1:
759: (4)           >>> a = np.zeros((2, 2, 2), dtype=int)
760: (4)           >>> a[d3] = 1
761: (4)           >>> a
762: (4)           array([[[1, 0],
763: (12)             [0, 0]],
764: (11)             [[0, 0],
765: (12)               [0, 1]]])
766: (4)           """
767: (4)           idx = np.arange(n)
768: (4)           return (idx,) * ndim
769: (0)           def _diag_indices_from(arr):
770: (4)               return (arr,)
771: (0)           @array_function_dispatch(_diag_indices_from)
772: (0)           def diag_indices_from(arr):
773: (4)               """
774: (4)               Return the indices to access the main diagonal of an n-dimensional array.
775: (4)               See `diag_indices` for full details.
776: (4)               Parameters
777: (4)               -----
778: (4)               arr : array, at least 2-D
779: (4)               See Also
780: (4)               -----
781: (4)               diag_indices
782: (4)               Notes
783: (4)               -----
784: (4)               .. versionadded:: 1.4.0
785: (4)               Examples
786: (4)               -----
787: (4)               Create a 4 by 4 array.

```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

788: (4)          >>> a = np.arange(16).reshape(4, 4)
789: (4)          >>> a
790: (4)          array([[ 0,  1,  2,  3],
791: (11)           [ 4,  5,  6,  7],
792: (11)           [ 8,  9, 10, 11],
793: (11)           [12, 13, 14, 15]])
794: (4)          Get the indices of the diagonal elements.
795: (4)          >>> di = np.diag_indices_from(a)
796: (4)          >>> di
797: (4)          (array([0, 1, 2, 3]), array([0, 1, 2, 3]))
798: (4)          >>> a[di]
799: (4)          array([ 0,  5, 10, 15])
800: (4)          This is simply syntactic sugar for diag_indices.
801: (4)          >>> np.diag_indices(a.shape[0])
802: (4)          (array([0, 1, 2, 3]), array([0, 1, 2, 3]))
803: (4)          """
804: (4)          if not arr.ndim >= 2:
805: (8)              raise ValueError("input array must be at least 2-d")
806: (4)          if not np.all(diff(arr.shape) == 0):
807: (8)              raise ValueError("All dimensions of input must be of equal length")
808: (4)          return diag_indices(arr.shape[0], arr.ndim)
```

---

File 211 - mixins.py:

```
1: (0)          """Mixin classes for custom array types that don't inherit from ndarray."""
2: (0)          from numpy.core import umath as um
3: (0)          __all__ = ['NDArrayOperatorsMixin']
4: (0)          def __disables_array_ufunc(obj):
5: (4)              """True when __array_ufunc__ is set to None."""
6: (4)              try:
7: (8)                  return obj.__array_ufunc__ is None
8: (4)              except AttributeError:
9: (8)                  return False
10: (0)             def __binary_method(ufunc, name):
11: (4)                 """Implement a forward binary method with a ufunc, e.g., __add__."""
12: (4)                 def func(self, other):
13: (8)                     if __disables_array_ufunc(other):
14: (12)                         return NotImplemented
15: (8)                     return ufunc(self, other)
16: (4)                     func.__name__ = '__{}__'.format(name)
17: (4)                     return func
18: (0)             def __reflected_binary_method(ufunc, name):
19: (4)                 """Implement a reflected binary method with a ufunc, e.g., __radd__."""
20: (4)                 def func(self, other):
21: (8)                     if __disables_array_ufunc(other):
22: (12)                         return NotImplemented
23: (8)                     return ufunc(other, self)
24: (4)                     func.__name__ = '__r{}__'.format(name)
25: (4)                     return func
26: (0)             def __inplace_binary_method(ufunc, name):
27: (4)                 """Implement an in-place binary method with a ufunc, e.g., __iadd__."""
28: (4)                 def func(self, other):
29: (8)                     return ufunc(self, other, out=(self,))
30: (4)                     func.__name__ = '__i{}__'.format(name)
31: (4)                     return func
32: (0)             def __numeric_methods(ufunc, name):
33: (4)                 """Implement forward, reflected and inplace binary methods with a
34: (4)                     ufunc."""
35: (12)                     return (_binary_method(ufunc, name),
36: (12)                         __reflected_binary_method(ufunc, name),
37: (12)                         __inplace_binary_method(ufunc, name))
37: (0)             def __unary_method(ufunc, name):
38: (4)                 """Implement a unary special method with a ufunc."""
39: (4)                 def func(self):
40: (8)                     return ufunc(self)
41: (4)                     func.__name__ = '__{}__'.format(name)
42: (4)                     return func
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

43: (0)
44: (4)
45: (4)
46: (4)
47: (4)
48: (4)
49: (4)
50: (4)
51: (4)
52: (4)
53: (4)
54: (4)
55: (4)
56: (4)
57: (8)
58: (12)
59: (16)
60: (12)
61: (12)
62: (16)
63: (16)
64: (20)
65: (24)
66: (16)
67: (31)
68: (16)
69: (20)
70: (24)
71: (24)
72: (16)
73: (16)
74: (20)
75: (16)
76: (20)
77: (16)
78: (20)
79: (12)
80: (16)
81: (4)
82: (4)
83: (8)
84: (8)
85: (8)
86: (8)
87: (8)
88: (8)
89: (8)
90: (8)
91: (8)
92: (4)
operations
93: (4)
94: (4)
95: (4)
96: (4)
97: (4)
98: (4)
99: (4)
100: (4)
101: (4)
102: (4)
103: (4)
104: (4)
105: (4)
106: (4)
107: (4)
108: (8)
109: (4)
110: (8)

class NDArryOperatorsMixin:
    """Mixin defining all operator special methods using __array_ufunc__.
    This class implements the special methods for almost all of Python's
    builtin operators defined in the `operator` module, including comparisons
    ('`==``', ``>``, etc.) and arithmetic ('`+``', ``*``', ``-``', etc.), by
    deferring to the ``__array_ufunc__`` method, which subclasses must
    implement.
    It is useful for writing classes that do not inherit from `numpy.ndarray`,
    but that should support arithmetic and numpy universal functions like
    arrays as described in `A Mechanism for Overriding Ufuncs
    <https://numpy.org/neps/nep-0013-ufunc-overrides.html>`.
    As an trivial example, consider this implementation of an ``ArrayLike```
    class that simply wraps a NumPy array and ensures that the result of any
    arithmetic operation is also an ``ArrayLike`` object::
        class ArrayLike(np.lib.mixins.NDArryOperatorsMixin):
            def __init__(self, value):
                self.value = np.asarray(value)
            _HANDLED_TYPES = (np.ndarray, numbers.Number)
            def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
                out = kwargs.get('out', ())
                for x in inputs + out:
                    if not isinstance(x, self._HANDLED_TYPES + (ArrayLike,)):
                        return NotImplemented
                    inputs = tuple(x.value if isinstance(x, ArrayLike) else x
                                  for x in inputs)
                if out:
                    kwargs['out'] = tuple(
                        x.value if isinstance(x, ArrayLike) else x
                        for x in out)
                result = getattr(ufunc, method)(*inputs, **kwargs)
                if type(result) is tuple:
                    return tuple(type(self)(x) for x in result)
                elif method == 'at':
                    return None
                else:
                    return type(self)(result)
            def __repr__(self):
                return '%s(%r)' % (type(self).__name__, self.value)
    In interactions between ``ArrayLike`` objects and numbers or numpy arrays,
    the result is always another ``ArrayLike``:
        >>> x = ArrayLike([1, 2, 3])
        >>> x - 1
        ArrayLike(array([0, 1, 2]))
        >>> 1 - x
        ArrayLike(array([ 0, -1, -2]))
        >>> np.arange(3) - x
        ArrayLike(array([-1, -1, -1]))
        >>> x - np.arange(3)
        ArrayLike(array([1, 1, 1]))
    Note that unlike ``numpy.ndarray``, ``ArrayLike`` does not allow
    with arbitrary, unrecognized types. This ensures that interactions with
    ArrayLike preserve a well-defined casting hierarchy.
    .. versionadded:: 1.13
    """
    __slots__ = ()
    __lt__ = _binary_method(um.less, 'lt')
    __le__ = _binary_method(um.less_equal, 'le')
    __eq__ = _binary_method(um.equal, 'eq')
    __ne__ = _binary_method(um.not_equal, 'ne')
    __gt__ = _binary_method(um.greater, 'gt')
    __ge__ = _binary_method(um.greater_equal, 'ge')
    __add__, __radd__, __iadd__ = _numeric_methods(um.add, 'add')
    __sub__, __rsub__, __isub__ = _numeric_methods(um.subtract, 'sub')
    __mul__, __rmul__, __imul__ = _numeric_methods(um.multiply, 'mul')
    __matmul__, __rmatmul__, __imatmul__ = _numeric_methods(
        um.matmul, 'matmul')
    __truediv__, __rtruediv__, __itruediv__ = _numeric_methods(
        um.true_divide, 'truediv')

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

111: (4)          __floordiv__, __rfloordiv__, __ifloordiv__ = _numeric_methods(
112: (8)            um.floor_divide, 'floordiv')
113: (4)          __mod__, __rmod__, __imod__ = _numeric_methods(um.remainder, 'mod')
114: (4)          __divmod__ = _binary_method(um.divmod, 'divmod')
115: (4)          __rdivmod__ = _reflected_binary_method(um.divmod, 'divmod')
116: (4)          __pow__, __rpow__, __ipow__ = _numeric_methods(um.power, 'pow')
117: (4)          __lshift__, __rlshift__, __ilshift__ = _numeric_methods(
118: (8)            um.left_shift, 'lshift')
119: (4)          __rshift__, __rrshift__, __irshift__ = _numeric_methods(
120: (8)            um.right_shift, 'rshift')
121: (4)          __and__, __rand__, __iand__ = _numeric_methods(um.bitwise_and, 'and')
122: (4)          __xor__, __rxor__, __ixor__ = _numeric_methods(um.bitwise_xor, 'xor')
123: (4)          __or__, __ror__, __ior__ = _numeric_methods(um.bitwise_or, 'or')
124: (4)          __neg__ = _unary_method(um.negative, 'neg')
125: (4)          __pos__ = _unary_method(um.positive, 'pos')
126: (4)          __abs__ = _unary_method(um.absolute, 'abs')
127: (4)          __invert__ = _unary_method(um.invert, 'invert')

```

---

## File 212 - nanfunctions.py:

```

1: (0)          """
2: (0)          Functions that ignore NaN.
3: (0)          Functions
4: (0)          -----
5: (0)          - `nanmin` -- minimum non-NaN value
6: (0)          - `nanmax` -- maximum non-NaN value
7: (0)          - `nanargmin` -- index of minimum non-NaN value
8: (0)          - `nanargmax` -- index of maximum non-NaN value
9: (0)          - `nansum` -- sum of non-NaN values
10: (0)         - `nanprod` -- product of non-NaN values
11: (0)         - `nancumsum` -- cumulative sum of non-NaN values
12: (0)         - `nancumprod` -- cumulative product of non-NaN values
13: (0)         - `nanmean` -- mean of non-NaN values
14: (0)         - `nanvar` -- variance of non-NaN values
15: (0)         - `nanstd` -- standard deviation of non-NaN values
16: (0)         - `nanmedian` -- median of non-NaN values
17: (0)         - `nanquantile` -- qth quantile of non-NaN values
18: (0)         - `nanpercentile` -- qth percentile of non-NaN values
19: (0)          """
20: (0)         import functools
21: (0)         import warnings
22: (0)         import numpy as np
23: (0)         from numpy.lib import function_base
24: (0)         from numpy.core import overrides
25: (0)         array_function_dispatch = functools.partial(
26: (4)           overrides.array_function_dispatch, module='numpy')
27: (0)         __all__ = [
28: (4)           'nansum', 'nanmax', 'nanmin', 'nanargmax', 'nanargmin', 'nanmean',
29: (4)           'nanmedian', 'nanpercentile', 'nanvar', 'nanstd', 'nanprod',
30: (4)           'nancumsum', 'nancumprod', 'nanquantile'
31: (4)         ]
32: (0)         def __nan_mask(a, out=None):
33: (4)             """
34: (4)             Parameters
35: (4)             -----
36: (4)             a : array-like
37: (8)               Input array with at least 1 dimension.
38: (4)             out : ndarray, optional
39: (8)               Alternate output array in which to place the result. The default
40: (8)               is ``None``; if provided, it must have the same shape as the
41: (8)               expected output and will prevent the allocation of a new array.
42: (4)             Returns
43: (4)             -----
44: (4)             y : bool ndarray or True
45: (8)               A bool array where ``np.nan`` positions are marked with ``False``
46: (8)               and other positions are marked with ``True``. If the type of ``a``
47: (8)               is such that it can't possibly contain ``np.nan``, returns ``True``.

```

```

48: (4)
49: (4)
50: (8)
51: (4)
52: (4)
53: (4)
54: (0)
55: (4)
56: (4)
57: (4)
58: (4)
59: (4)
60: (4)
61: (4)
62: (4)
63: (4)
64: (4)
65: (4)
66: (8)
67: (4)
68: (8)
69: (4)
70: (4)
71: (4)
72: (8)
73: (8)
74: (4)
75: (8)
76: (8)
77: (4)
78: (4)
79: (4)
80: (8)
81: (4)
82: (8)
83: (4)
84: (8)
85: (4)
86: (8)
87: (8)
88: (4)
89: (0)
90: (4)
91: (4)
92: (4)
93: (4)
94: (4)
95: (4)
96: (8)
97: (8)
98: (4)
99: (8)
100: (4)
101: (8)
102: (8)
103: (4)
104: (4)
105: (4)
106: (8)
107: (4)
108: (4)
109: (8)
110: (4)
111: (8)
112: (4)
113: (0)
114: (4)
115: (4)
116: (4)

    """
        if a.dtype.kind not in 'fc':
            return True
        y = np.isnan(a, out=out)
        y = np.invert(y, out=y)
        return y
    def _replace_nan(a, val):
        """
        If `a` is of inexact type, make a copy of `a`, replace NaNs with
        the `val` value, and return the copy together with a boolean mask
        marking the locations where NaNs were present. If `a` is not of
        inexact type, do nothing and return `a` together with a mask of None.
        Note that scalars will end up as array scalars, which is important
        for using the result as the value of the out argument in some
        operations.
        Parameters
        -----
        a : array-like
            Input array.
        val : float
            NaN values are set to val before doing the operation.
        Returns
        -----
        y : ndarray
            If `a` is of inexact type, return a copy of `a` with the NaNs
            replaced by the fill value, otherwise return `a`.
        mask: {bool, None}
            If `a` is of inexact type, return a boolean mask marking locations of
            NaNs, otherwise return None.
        """
        a = np.asarray(a)
        if a.dtype == np.object_:
            mask = np.not_equal(a, a, dtype=bool)
        elif issubclass(a.dtype.type, np.inexact):
            mask = np.isnan(a)
        else:
            mask = None
        if mask is not None:
            a = np.array(a, subok=True, copy=True)
            np.copyto(a, val, where=mask)
        return a, mask
    def _copyto(a, val, mask):
        """
        Replace values in `a` with NaN where `mask` is True. This differs from
        copyto in that it will deal with the case where `a` is a numpy scalar.
        Parameters
        -----
        a : ndarray or numpy scalar
            Array or numpy scalar some of whose values are to be replaced
            by val.
        val : numpy scalar
            Value used a replacement.
        mask : ndarray, scalar
            Boolean array. Where True the corresponding element of `a` is
            replaced by `val`. Broadcasts.
        Returns
        -----
        res : ndarray, scalar
            Array with elements replaced or scalar `val`.
        """
        if isinstance(a, np.ndarray):
            np.copyto(a, val, where=mask, casting='unsafe')
        else:
            a = a.dtype.type(val)
        return a
    def _remove_nan_1d(arr1d, overwrite_input=False):
        """
        Equivalent to arr1d[~arr1d.isnan()], but in a different order
        Presumably faster as it incurs fewer copies

```

```

117: (4)                               Parameters
118: (4)-----arr1d : ndarray
119: (4)                               Array to remove nans from
120: (8)-----overwrite_input : bool
121: (4)                               True if `arr1d` can be modified in place
122: (8)-----Returns
123: (4)-----res : ndarray
124: (4)                               Array with nan elements removed
125: (4)-----overwrite_input : bool
126: (8)                               True if `res` can be modified in place, given the constraint on the
127: (4)                               input
128: (8)
129: (8)
130: (4)
131: (4)
132: (8)
133: (4)
134: (8)
135: (4)
136: (4)
137: (8)
138: (22)
139: (8)
140: (4)
141: (8)
142: (4)
143: (8)
144: (12)
145: (8)
146: (8)
147: (8)
148: (0)
149: (4)
150: (4)
151: (4)
152: (4)
153: (4)
154: (4)
155: (4)
156: (4)
157: (8)
158: (4)
159: (8)
160: (4)
161: (8)
162: (8)
163: (8)
164: (4)
165: (4)
166: (4)
167: (8)
168: (8)
169: (4)
170: (4)
171: (8)
172: (12)
173: (16)
174: (12)
175: (16)
176: (8)
177: (12)
178: (16)
179: (20)
180: (16)
181: (20)
182: (12)
183: (16)
184: (0)
185: (23)

    Parameters
    -----
    arr1d : ndarray
        Array to remove nans from
    overwrite_input : bool
        True if `arr1d` can be modified in place
    Returns
    -----
    res : ndarray
        Array with nan elements removed
    overwrite_input : bool
        True if `res` can be modified in place, given the constraint on the
        input
    """
    if arr1d.dtype == object:
        c = np.not_equal(arr1d, arr1d, dtype=bool)
    else:
        c = np.isnan(arr1d)
    s = np.nonzero(c)[0]
    if s.size == arr1d.size:
        warnings.warn("All-NaN slice encountered", RuntimeWarning,
                      stacklevel=6)
        return arr1d[:0], True
    elif s.size == 0:
        return arr1d, overwrite_input
    else:
        if not overwrite_input:
            arr1d = arr1d.copy()
        enonan = arr1d[-s.size:][~c[-s.size:]]
        arr1d[s[:enonan.size]] = enonan
        return arr1d[:-s.size], True

def _divide_by_count(a, b, out=None):
    """
    Compute a/b ignoring invalid results. If `a` is an array the division
    is done in place. If `a` is a scalar, then its type is preserved in the
    output. If out is None, then a is used instead so that the division
    is in place. Note that this is only called with `a` an inexact type.

    Parameters
    -----
    a : {ndarray, numpy scalar}
        Numerator. Expected to be of inexact type but not checked.
    b : {ndarray, numpy scalar}
        Denominator.
    out : ndarray, optional
        Alternate output array in which to place the result. The default
        is ``None``; if provided, it must have the same shape as the
        expected output, but the type will be cast if necessary.
    Returns
    -----
    ret : {ndarray, numpy scalar}
        The return value is a/b. If `a` was an ndarray the division is done
        in place. If `a` is a numpy scalar, the division preserves its type.
    """
    with np.errstate(invalid='ignore', divide='ignore'):
        if isinstance(a, np.ndarray):
            if out is None:
                return np.divide(a, b, out=a, casting='unsafe')
            else:
                return np.divide(a, b, out=out, casting='unsafe')
        else:
            if out is None:
                try:
                    return a.dtype.type(a / b)
                except AttributeError:
                    return a / b
            else:
                return np.divide(a, b, out=out, casting='unsafe')

def _nanmin_dispatcher(a, axis=None, out=None, keepdims=None,
                      initial=None, where=None):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

186: (4)           return (a, out)
187: (0)           @array_function_dispatch(_nanmin_dispatcher)
188: (0)           def nanmin(a, axis=None, out=None, keepdims=np._NoValue,
189: (11)             initial=np._NoValue,
190: (4)               where=np._NoValue):
191: (4)               """
192: (4)               Return minimum of an array or minimum along an axis, ignoring any NaNs.
193: (4)               When all-NaN slices are encountered a ``RuntimeWarning`` is raised and
194: (4)               Nan is returned for that slice.
195: (4)               Parameters
196: (4)               -----
197: (4)               a : array_like
198: (8)                 Array containing numbers whose minimum is desired. If `a` is not an
199: (8)                   array, a conversion is attempted.
200: (4)               axis : {int, tuple of int, None}, optional
201: (8)                 Axis or axes along which the minimum is computed. The default is to
202: (4)                   compute
203: (8)                   the minimum of the flattened array.
204: (4)               out : ndarray, optional
205: (8)                 Alternate output array in which to place the result. The default
206: (8)                   is ``None``; if provided, it must have the same shape as the
207: (8)                   expected output, but the type will be cast if necessary. See
208: (4)                     :ref:`ufuncs-output-type` for more details.
209: (8)                     .. versionadded:: 1.8.0
210: (4)               keepdims : bool, optional
211: (8)                 If this is set to True, the axes which are reduced are left
212: (8)                   in the result as dimensions with size one. With this option,
213: (8)                   the result will broadcast correctly against the original `a`.
214: (8)                   If the value is anything but the default, then
215: (8)                     `keepdims` will be passed through to the `min` method
216: (8)                     of sub-classes of `ndarray`. If the sub-classes methods
217: (4)                     does not implement `keepdims` any exceptions will be raised.
218: (8)                     .. versionadded:: 1.8.0
219: (4)               initial : scalar, optional
220: (8)                 The maximum value of an output element. Must be present to allow
221: (8)                   computation on empty slice. See `~numpy.ufunc.reduce` for details.
222: (4)                     .. versionadded:: 1.22.0
223: (8)               where : array_like of bool, optional
224: (8)                 Elements to compare for the minimum. See `~numpy.ufunc.reduce`
225: (4)                   for details.
226: (4)                     .. versionadded:: 1.22.0
227: (4)               Returns
228: (4)               -----
229: (4)               nanmin : ndarray
230: (8)                 An array with the same shape as `a`, with the specified axis
231: (8)                   removed. If `a` is a 0-d array, or if axis is None, an ndarray
232: (8)                   scalar is returned. The same dtype as `a` is returned.
233: (4)               See Also
234: (4)               -----
235: (4)               nanmax :
236: (8)                 The maximum value of an array along a given axis, ignoring any NaNs.
237: (4)               amin :
238: (8)                 The minimum value of an array along a given axis, propagating any
239: (4)                   NaNs.
240: (8)               fmin :
241: (4)                 Element-wise minimum of two arrays, ignoring any NaNs.
242: (8)               minimum :
243: (4)                 Element-wise minimum of two arrays, propagating any NaNs.
244: (4)               isnan :
245: (8)                 Shows which elements are Not a Number (NaN).
246: (4)               isfinite:
247: (4)                 Shows which elements are neither NaN nor infinity.
248: (4)               amax, fmax, maximum
249: (4)               Notes
250: (4)               -----
251: (4)               NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
252: (4)                   (IEEE 754). This means that Not a Number is not equivalent to infinity.
253: (4)                   Positive infinity is treated as a very large number and negative
254: (4)                   infinity is treated as a very small (i.e. negative) number.
255: (4)                   If the input has a integer type the function is equivalent to np.min.

```

```

253: (4)          Examples
254: (4)          -----
255: (4)          >>> a = np.array([[1, 2], [3, np.nan]])
256: (4)          >>> np.nanmin(a)
257: (4)          1.0
258: (4)          >>> np.nanmin(a, axis=0)
259: (4)          array([1., 2.])
260: (4)          >>> np.nanmin(a, axis=1)
261: (4)          array([1., 3.])
262: (4)          When positive infinity and negative infinity are present:
263: (4)          >>> np.nanmin([1, 2, np.nan, np.inf])
264: (4)          1.0
265: (4)          >>> np.nanmin([1, 2, np.nan, np.NINF])
266: (4)          -inf
267: (4)          """
268: (4)          kwargs = {}
269: (4)          if keepdims is not np._NoValue:
270: (8)              kwargs['keepdims'] = keepdims
271: (4)          if initial is not np._NoValue:
272: (8)              kwargs['initial'] = initial
273: (4)          if where is not np._NoValue:
274: (8)              kwargs['where'] = where
275: (4)          if type(a) is np.ndarray and a.dtype != np.object_:
276: (8)              res = np.fmin.reduce(a, axis=axis, out=out, **kwargs)
277: (8)              if np.isnan(res).any():
278: (12)                  warnings.warn("All-NaN slice encountered", RuntimeWarning,
279: (26)                                  stacklevel=2)
280: (4)          else:
281: (8)              a, mask = _replace_nan(a, +np.inf)
282: (8)              res = np.amin(a, axis=axis, out=out, **kwargs)
283: (8)              if mask is None:
284: (12)                  return res
285: (8)              kwargs.pop("initial", None)
286: (8)              mask = np.all(mask, axis=axis, **kwargs)
287: (8)              if np.any(mask):
288: (12)                  res = _copyto(res, np.nan, mask)
289: (12)                  warnings.warn("All-NaN axis encountered", RuntimeWarning,
290: (26)                                  stacklevel=2)
291: (4)          return res
292: (0)          def _nanmax_dispatcher(a, axis=None, out=None, keepdims=None,
293: (23)                      initial=None, where=None):
294: (4)              return (a, out)
295: (0)          @array_function_dispatch(_nanmax_dispatcher)
296: (0)          def nanmax(a, axis=None, out=None, keepdims=np._NoValue, initial=np._NoValue,
297: (11)              where=np._NoValue):
298: (4)              """
299: (4)              Return the maximum of an array or maximum along an axis, ignoring any
300: (4)              NaNs. When all-NaN slices are encountered a ``RuntimeWarning`` is
301: (4)              raised and NaN is returned for that slice.
302: (4)              Parameters
303: (4)              -----
304: (4)              a : array_like
305: (8)                  Array containing numbers whose maximum is desired. If `a` is not an
306: (8)                  array, a conversion is attempted.
307: (4)              axis : {int, tuple of int, None}, optional
308: (8)                  Axis or axes along which the maximum is computed. The default is to
309: (8)                  compute
310: (8)                  the maximum of the flattened array.
311: (4)              out : ndarray, optional
312: (8)                  Alternate output array in which to place the result. The default
313: (8)                  is ``None``; if provided, it must have the same shape as the
314: (8)                  expected output, but the type will be cast if necessary. See
315: (8)                  :ref:`ufuncs-output-type` for more details.
316: (4)                  .. versionadded:: 1.8.0
317: (4)              keepdims : bool, optional
318: (8)                  If this is set to True, the axes which are reduced are left
319: (8)                  in the result as dimensions with size one. With this option,
320: (8)                  the result will broadcast correctly against the original `a`.

```

If the value is anything but the default, then

```

321: (8) `keepdims` will be passed through to the `max` method
322: (8) of sub-classes of `ndarray`. If the sub-classes methods
323: (8) does not implement `keepdims` any exceptions will be raised.
324: (8) .. versionadded:: 1.8.0
325: (4) initial : scalar, optional
326: (8) The minimum value of an output element. Must be present to allow
327: (8) computation on empty slice. See `~numpy.ufunc.reduce` for details.
328: (8) .. versionadded:: 1.22.0
329: (4) where : array_like of bool, optional
330: (8) Elements to compare for the maximum. See `~numpy.ufunc.reduce`
331: (8) for details.
332: (8) .. versionadded:: 1.22.0
333: (4) Returns
334: (4) -----
335: (4) nanmax : ndarray
336: (8) An array with the same shape as `a`, with the specified axis removed.
337: (8) If `a` is a 0-d array, or if axis is None, an ndarray scalar is
338: (8) returned. The same dtype as `a` is returned.
339: (4) See Also
340: (4) -----
341: (4) nanmin :
342: (8) The minimum value of an array along a given axis, ignoring any NaNs.
343: (4) amax :
344: (8) The maximum value of an array along a given axis, propagating any
NaNs.
345: (4) fmax :
346: (8) Element-wise maximum of two arrays, ignoring any NaNs.
347: (4) maximum :
348: (8) Element-wise maximum of two arrays, propagating any NaNs.
349: (4) isnan :
350: (8) Shows which elements are Not a Number (NaN).
351: (4) isnfinite:
352: (8) Shows which elements are neither NaN nor infinity.
353: (4) amin, fmin, minimum
354: (4) Notes
355: (4) -----
356: (4) NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
357: (4) (IEEE 754). This means that Not a Number is not equivalent to infinity.
358: (4) Positive infinity is treated as a very large number and negative
359: (4) infinity is treated as a very small (i.e. negative) number.
360: (4) If the input has a integer type the function is equivalent to np.max.
361: (4) Examples
362: (4) -----
363: (4) >>> a = np.array([[1, 2], [3, np.nan]])
364: (4) >>> np.nanmax(a)
365: (4) 3.0
366: (4) >>> np.nanmax(a, axis=0)
367: (4) array([3., 2.])
368: (4) >>> np.nanmax(a, axis=1)
369: (4) array([2., 3.])
370: (4) When positive infinity and negative infinity are present:
371: (4) >>> np.nanmax([1, 2, np.nan, np.NINF])
372: (4) 2.0
373: (4) >>> np.nanmax([1, 2, np.nan, np.inf])
374: (4) inf
375: (4) """
376: (4)     kwargs = {}
377: (4)     if keepdims is not np._NoValue:
378: (8)         kwargs['keepdims'] = keepdims
379: (4)     if initial is not np._NoValue:
380: (8)         kwargs['initial'] = initial
381: (4)     if where is not np._NoValue:
382: (8)         kwargs['where'] = where
383: (4)     if type(a) is np.ndarray and a.dtype != np.object_:
384: (8)         res = np.fmax.reduce(a, axis=axis, out=out, **kwargs)
385: (8)         if np.isnan(res).any():
386: (12)             warnings.warn("All-NaN slice encountered", RuntimeWarning,
387: (26)                             stacklevel=2)
388: (4)     else:

```

```

389: (8)             a, mask = _replace_nan(a, -np.inf)
390: (8)             res = npamax(a, axis=axis, out=out, **kwargs)
391: (8)             if mask is None:
392: (12)                 return res
393: (8)             kwargs.pop("initial", None)
394: (8)             mask = np.all(mask, axis=axis, **kwargs)
395: (8)             if np.any(mask):
396: (12)                 res = _copyto(res, np.nan, mask)
397: (12)                 warnings.warn("All-NaN axis encountered", RuntimeWarning,
398: (26)                               stacklevel=2)
399: (4)             return res
400: (0)             def _nanargmin_dispatcher(a, axis=None, out=None, *, keepdims=None):
401: (4)                 return (a,)
402: (0)             @array_function_dispatch(_nanargmin_dispatcher)
403: (0)             def nanargmin(a, axis=None, out=None, *, keepdims=np._NoValue):
404: (4)                 """
405: (4)                 Return the indices of the minimum values in the specified axis ignoring
406: (4)                 NaNs. For all-NaN slices ``ValueError`` is raised. Warning: the results
407: (4)                 cannot be trusted if a slice contains only NaNs and Infs.
408: (4)                 Parameters
409: (4)                 -----
410: (4)                 a : array_like
411: (8)                     Input data.
412: (4)                 axis : int, optional
413: (8)                     Axis along which to operate. By default flattened input is used.
414: (4)                 out : array, optional
415: (8)                     If provided, the result will be inserted into this array. It should
416: (8)                     be of the appropriate shape and dtype.
417: (8)                     .. versionadded:: 1.22.0
418: (4)                 keepdims : bool, optional
419: (8)                     If this is set to True, the axes which are reduced are left
420: (8)                     in the result as dimensions with size one. With this option,
421: (8)                     the result will broadcast correctly against the array.
422: (8)                     .. versionadded:: 1.22.0
423: (4)             Returns
424: (4)             -----
425: (4)             index_array : ndarray
426: (8)                 An array of indices or a single index value.
427: (4)             See Also
428: (4)             -----
429: (4)             argmin, nanargmax
430: (4)             Examples
431: (4)             -----
432: (4)             >>> a = np.array([[np.nan, 4], [2, 3]])
433: (4)             >>> np.argmin(a)
434: (4)             0
435: (4)             >>> np.nanargmin(a)
436: (4)             2
437: (4)             >>> np.nanargmin(a, axis=0)
438: (4)             array([1, 1])
439: (4)             >>> np.nanargmin(a, axis=1)
440: (4)             array([1, 0])
441: (4)             """
442: (4)             a, mask = _replace_nan(a, np.inf)
443: (4)             if mask is not None:
444: (8)                 mask = np.all(mask, axis=axis)
445: (8)                 if np.any(mask):
446: (12)                     raise ValueError("All-NaN slice encountered")
447: (4)                 res = np.argmin(a, axis=axis, out=out, keepdims=keepdims)
448: (4)                 return res
449: (0)             def _nanargmax_dispatcher(a, axis=None, out=None, *, keepdims=None):
450: (4)                 return (a,)
451: (0)             @array_function_dispatch(_nanargmax_dispatcher)
452: (0)             def nanargmax(a, axis=None, out=None, *, keepdims=np._NoValue):
453: (4)                 """
454: (4)                 Return the indices of the maximum values in the specified axis ignoring
455: (4)                 NaNs. For all-NaN slices ``ValueError`` is raised. Warning: the
456: (4)                 results cannot be trusted if a slice contains only NaNs and -Infs.
457: (4)                 Parameters

```

```

458: (4)      -----
459: (4)      a : array_like
460: (8)      Input data.
461: (4)      axis : int, optional
462: (8)      Axis along which to operate. By default flattened input is used.
463: (4)      out : array, optional
464: (8)      If provided, the result will be inserted into this array. It should
465: (8)      be of the appropriate shape and dtype.
466: (8)      .. versionadded:: 1.22.0
467: (4)      keepdims : bool, optional
468: (8)      If this is set to True, the axes which are reduced are left
469: (8)      in the result as dimensions with size one. With this option,
470: (8)      the result will broadcast correctly against the array.
471: (8)      .. versionadded:: 1.22.0
472: (4)      Returns
473: (4)      -----
474: (4)      index_array : ndarray
475: (8)      An array of indices or a single index value.
476: (4)      See Also
477: (4)      -----
478: (4)      argmax, nanargmin
479: (4)      Examples
480: (4)      -----
481: (4)      >>> a = np.array([[np.nan, 4], [2, 3]])
482: (4)      >>> np.argmax(a)
483: (4)      0
484: (4)      >>> np.nanargmax(a)
485: (4)      1
486: (4)      >>> np.nanargmax(a, axis=0)
487: (4)      array([1, 0])
488: (4)      >>> np.nanargmax(a, axis=1)
489: (4)      array([1, 1])
490: (4)      """
491: (4)      a, mask = _replace_nan(a, -np.inf)
492: (4)      if mask is not None:
493: (8)          mask = np.all(mask, axis=axis)
494: (8)          if np.any(mask):
495: (12)              raise ValueError("All-NaN slice encountered")
496: (4)          res = np.argmax(a, axis=axis, out=out, keepdims=keepdims)
497: (4)          return res
498: (0)      def _nansum_dispatcher(a, axis=None, dtype=None, out=None,
499: (23)                  keepdims=None,
500: (4)                      initial=None, where=None):
501: (0)          return (a, out)
502: (0)      @array_function_dispatch(_nansum_dispatcher)
503: (0)      def nansum(a, axis=None, dtype=None, out=None, keepdims=np._NoValue,
504: (11)          initial=np._NoValue, where=np._NoValue):
505: (4)          """
506: (4)          Return the sum of array elements over a given axis treating Not a
507: (4)          Numbers (NaNs) as zero.
508: (4)          In NumPy versions <= 1.9.0 Nan is returned for slices that are all-NaN or
509: (4)          empty. In later versions zero is returned.
510: (4)          Parameters
511: (4)          -----
512: (4)          a : array_like
513: (8)          Array containing numbers whose sum is desired. If `a` is not an
514: (8)          array, a conversion is attempted.
515: (4)          axis : {int, tuple of int, None}, optional
516: (8)          Axis or axes along which the sum is computed. The default is to
517: (8)          sum of the flattened array.
518: (4)          dtype : data-type, optional
519: (8)          The type of the returned array and of the accumulator in which the
520: (8)          elements are summed. By default, the dtype of `a` is used. An
521: (8)          exception is when `a` has an integer type with less precision than
522: (8)          the platform (u)intp. In that case, the default will be either
523: (8)          (u)int32 or (u)int64 depending on whether the platform is 32 or 64
524: (8)          bits. For inexact inputs, dtype must be inexact.
525: (4)          .. versionadded:: 1.8.0
526: (4)          out : ndarray, optional

```

```

526: (8)                                Alternate output array in which to place the result. The default
527: (8)                                is ``None``. If provided, it must have the same shape as the
528: (8)                                expected output, but the type will be cast if necessary. See
529: (8)                                :ref:`ufuncs-output-type` for more details. The casting of NaN to
integer
530: (8)                                can yield unexpected results.
531: (8)
532: (4)                                .. versionadded:: 1.8.0
keepdims : bool, optional
533: (8)                                If this is set to True, the axes which are reduced are left
534: (8)                                in the result as dimensions with size one. With this option,
535: (8)                                the result will broadcast correctly against the original `a`.
536: (8)                                If the value is anything but the default, then
537: (8)                                `keepdims` will be passed through to the `mean` or `sum` methods
538: (8)                                of sub-classes of `ndarray`. If the sub-classes methods
539: (8)                                does not implement `keepdims` any exceptions will be raised.
540: (8)                                .. versionadded:: 1.8.0
541: (4)                                initial : scalar, optional
542: (8)                                Starting value for the sum. See `~numpy.ufunc.reduce` for details.
543: (8)                                .. versionadded:: 1.22.0
544: (4)                                where : array_like of bool, optional
545: (8)                                Elements to include in the sum. See `~numpy.ufunc.reduce` for details.
546: (8)                                .. versionadded:: 1.22.0
547: (4)                                Returns
548: (4)
549: (4)                                -----
nansum : ndarray
550: (8)                                A new array holding the result is returned unless `out` is
551: (8)                                specified, in which it is returned. The result has the same
552: (8)                                size as `a`, and the same shape as `a` if `axis` is not None
553: (8)                                or `a` is a 1-d array.
554: (4)                                See Also
555: (4)
556: (4)                                -----
556: (4)                                numpy.sum : Sum across array propagating NaNs.
557: (4)                                isnan : Show which elements are NaN.
558: (4)                                isfinite : Show which elements are not NaN or +/-inf.
559: (4)                                Notes
560: (4)
561: (4)                                -----
561: (4)                                If both positive and negative infinity are present, the sum will be Not
562: (4)                                A Number (NaN).
563: (4)                                Examples
564: (4)
565: (4)                                -----
565: (4)                                >>> np.nansum(1)
566: (4)                                1
567: (4)                                >>> np.nansum([1])
568: (4)                                1
569: (4)                                >>> np.nansum([1, np.nan])
570: (4)                                1.0
571: (4)                                >>> a = np.array([[1, 1], [1, np.nan]])
572: (4)                                >>> np.nansum(a)
573: (4)                                3.0
574: (4)                                >>> np.nansum(a, axis=0)
575: (4)                                array([2., 1.])
576: (4)                                >>> np.nansum([1, np.nan, np.inf])
577: (4)                                inf
578: (4)                                >>> np.nansum([1, np.nan, np.NINF])
579: (4)                                -inf
580: (4)                                >>> from numpy.testing import suppress_warnings
581: (4)                                >>> with suppress_warnings() as sup:
582: (4)                                    ...     sup.filter(RuntimeWarning)
583: (4)                                    ...     np.nansum([1, np.nan, np.inf, -np.inf]) # both +/- infinity
present
584: (4)                                nan
585: (4)                                """
a, mask = _replace_nan(a, 0)
586: (4)                                return np.sum(a, axis=axis, dtype=dtype, out=out, keepdims=keepdims,
587: (4)                                         initial=initial, where=where)
588: (18)
589: (0)                                def _nanprod_dispatcher(a, axis=None, dtype=None, out=None, keepdims=None,
590: (24)                                         initial=None, where=None):
591: (4)                                    return (a, out)
592: (0)                                @array_function_dispatch(_nanprod_dispatcher)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

593: (0)
594: (12)
595: (4)
596: (4)
597: (4)
598: (4)
599: (4)
600: (4)
601: (4)
602: (4)
603: (8)
604: (8)
605: (4)
606: (8)
compute
607: (8)
608: (4)
609: (8)
610: (8)
611: (8)
612: (8)
613: (8)
614: (8)
615: (4)
616: (8)
617: (8)
618: (8)
619: (8)
integer
620: (8)
621: (4)
622: (8)
623: (8)
624: (8)
625: (4)
626: (8)
627: (8)
628: (8)
629: (4)
630: (8)
631: (8)
632: (8)
633: (4)
634: (4)
635: (4)
636: (8)
637: (8)
638: (4)
639: (4)
640: (4)
641: (4)
642: (4)
643: (4)
644: (4)
645: (4)
646: (4)
647: (4)
648: (4)
649: (4)
650: (4)
651: (4)
652: (4)
653: (4)
654: (4)
655: (4)
656: (4)
657: (4)
658: (19)
659: (0)
def nanprod(a, axis=None, dtype=None, out=None, keepdims=np._NoValue,
           initial=np._NoValue, where=np._NoValue):
    """
    Return the product of array elements over a given axis treating Not a
    Numbers (NaNs) as ones.
    One is returned for slices that are all-NaN or empty.
    .. versionadded:: 1.10.0
    Parameters
    -----
    a : array_like
        Array containing numbers whose product is desired. If `a` is not an
        array, a conversion is attempted.
    axis : {int, tuple of int, None}, optional
        Axis or axes along which the product is computed. The default is to
        the product of the flattened array.
    dtype : data-type, optional
        The type of the returned array and of the accumulator in which the
        elements are summed. By default, the dtype of `a` is used. An
        exception is when `a` has an integer type with less precision than
        the platform (u)intp. In that case, the default will be either
        (u)int32 or (u)int64 depending on whether the platform is 32 or 64
        bits. For inexact inputs, dtype must be inexact.
    out : ndarray, optional
        Alternate output array in which to place the result. The default
        is ``None``. If provided, it must have the same shape as the
        expected output, but the type will be cast if necessary. See
        :ref:`ufuncs-output-type` for more details. The casting of NaN to
        can yield unexpected results.
    keepdims : bool, optional
        If True, the axes which are reduced are left in the result as
        dimensions with size one. With this option, the result will
        broadcast correctly against the original `arr`.
    initial : scalar, optional
        The starting value for this product. See `~numpy.ufunc.reduce`
        for details.
        .. versionadded:: 1.22.0
    where : array_like of bool, optional
        Elements to include in the product. See `~numpy.ufunc.reduce`
        for details.
        .. versionadded:: 1.22.0
    Returns
    -----
    nanprod : ndarray
        A new array holding the result is returned unless `out` is
        specified, in which case it is returned.
    See Also
    -----
    numpy.prod : Product across array propagating NaNs.
    isnan : Show which elements are NaN.
    Examples
    -----
    >>> np.nanprod(1)
    1
    >>> np.nanprod([1])
    1
    >>> np.nanprod([1, np.nan])
    1.0
    >>> a = np.array([[1, 2], [3, np.nan]])
    >>> np.nanprod(a)
    6.0
    >>> np.nanprod(a, axis=0)
    array([3., 2.])
    """
    a, mask = _replace_nan(a, 1)
    return np.prod(a, axis=axis, dtype=dtype, out=out, keepdims=keepdims,
                  initial=initial, where=where)
def _nancumsum_dispatcher(a, axis=None, dtype=None, out=None):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

660: (4)             return (a, out)
661: (0)             @array_function_dispatch(_nancumsum_dispatcher)
662: (0)             def nancumsum(a, axis=None, dtype=None, out=None):
663: (4)                 """
664: (4)                 Return the cumulative sum of array elements over a given axis treating Not
a
665: (4)                 Numbers (NaNs) as zero. The cumulative sum does not change when NaNs are
666: (4)                 encountered and leading NaNs are replaced by zeros.
667: (4)                 Zeros are returned for slices that are all-NaN or empty.
668: (4)                 .. versionadded:: 1.12.0
669: (4)             Parameters
670: (4)                 -----
671: (4)                 a : array_like
672: (8)                   Input array.
673: (4)                 axis : int, optional
674: (8)                   Axis along which the cumulative sum is computed. The default
675: (8)                   (None) is to compute the cumsum over the flattened array.
676: (4)                 dtype : dtype, optional
677: (8)                   Type of the returned array and of the accumulator in which the
678: (8)                   elements are summed. If `dtype` is not specified, it defaults
679: (8)                   to the dtype of `a`, unless `a` has an integer dtype with a
680: (8)                   precision less than that of the default platform integer. In
681: (8)                   that case, the default platform integer is used.
682: (4)                 out : ndarray, optional
683: (8)                   Alternative output array in which to place the result. It must
684: (8)                   have the same shape and buffer length as the expected output
685: (8)                   but the type will be cast if necessary. See :ref:`ufuncs-output-type`
for
686: (8)                   more details.
687: (4)             Returns
688: (4)                 -----
689: (4)                 nancumsum : ndarray.
690: (8)                   A new array holding the result is returned unless `out` is
691: (8)                   specified, in which it is returned. The result has the same
692: (8)                   size as `a`, and the same shape as `a` if `axis` is not None
693: (8)                   or `a` is a 1-d array.
694: (4)             See Also
695: (4)                 -----
696: (4)                 numpy.cumsum : Cumulative sum across array propagating NaNs.
697: (4)                 isnan : Show which elements are NaN.
698: (4)             Examples
699: (4)                 -----
700: (4)                 >>> np.nancumsum(1)
701: (4)                 array([1])
702: (4)                 >>> np.nancumsum([1])
703: (4)                 array([1])
704: (4)                 >>> np.nancumsum([1, np.nan])
705: (4)                 array([1., 1.])
706: (4)                 >>> a = np.array([[1, 2], [3, np.nan]])
707: (4)                 >>> np.nancumsum(a)
708: (4)                 array([1., 3., 6., 6.])
709: (4)                 >>> np.nancumsum(a, axis=0)
710: (4)                 array([[1., 2.],
711: (11)                  [4., 2.]])
712: (4)                 >>> np.nancumsum(a, axis=1)
713: (4)                 array([[1., 3.],
714: (11)                  [3., 3.]])
715: (4)                 """
716: (4)                 a, mask = _replace_nan(a, 0)
717: (4)                 return np.cumsum(a, axis=axis, dtype=dtype, out=out)
718: (0)             def _nancumprod_dispatcher(a, axis=None, dtype=None, out=None):
719: (4)                 return (a, out)
720: (0)             @array_function_dispatch(_nancumprod_dispatcher)
721: (0)             def nancumprod(a, axis=None, dtype=None, out=None):
722: (4)                 """
723: (4)                 Return the cumulative product of array elements over a given axis treating Not
a
724: (4)                 Numbers (NaNs) as one. The cumulative product does not change when NaNs
are

```

```

725: (4) encountered and leading NaNs are replaced by ones.
726: (4) Ones are returned for slices that are all-NaN or empty.
727: (4) .. versionadded:: 1.12.0
728: (4) Parameters
729: (4) -----
730: (4) a : array_like
731: (8) Input array.
732: (4) axis : int, optional
733: (8) Axis along which the cumulative product is computed. By default
734: (8) the input is flattened.
735: (4) dtype : dtype, optional
736: (8) Type of the returned array, as well as of the accumulator in which
737: (8) the elements are multiplied. If *dtype* is not specified, it
738: (8) defaults to the dtype of `a`, unless `a` has an integer dtype with
739: (8) a precision less than that of the default platform integer. In
740: (8) that case, the default platform integer is used instead.
741: (4) out : ndarray, optional
742: (8) Alternative output array in which to place the result. It must
743: (8) have the same shape and buffer length as the expected output
744: (8) but the type of the resulting values will be cast if necessary.
745: (4) Returns
746: (4) -----
747: (4) nancumprod : ndarray
748: (8) A new array holding the result is returned unless `out` is
749: (8) specified, in which case it is returned.
750: (4) See Also
751: (4) -----
752: (4) numpy.cumprod : Cumulative product across array propagating NaNs.
753: (4) isnan : Show which elements are NaN.
754: (4) Examples
755: (4) -----
756: (4) >>> np.nancumprod(1)
757: (4) array([1])
758: (4) >>> np.nancumprod([1])
759: (4) array([1])
760: (4) >>> np.nancumprod([1, np.nan])
761: (4) array([1., 1.])
762: (4) >>> a = np.array([[1, 2], [3, np.nan]])
763: (4) >>> np.nancumprod(a)
764: (4) array([1., 2., 6., 6.])
765: (4) >>> np.nancumprod(a, axis=0)
766: (4) array([[1., 2.],
767: (11) [3., 2.]])
768: (4) >>> np.nancumprod(a, axis=1)
769: (4) array([[1., 2.],
770: (11) [3., 3.]])
771: (4) """
772: (4) a, mask = _replace_nan(a, 1)
773: (4) return np.cumprod(a, axis=axis, dtype=dtype, out=out)
774: (0) def _nanmean_dispatcher(a, axis=None, dtype=None, out=None, keepdims=None,
775: (24) *, where=None):
776: (4)     return (a, out)
777: (0) @array_function_dispatch(_nanmean_dispatcher)
778: (0) def nanmean(a, axis=None, dtype=None, out=None, keepdims=np._NoValue,
779: (12) *, where=np._NoValue):
780: (4) """
781: (4) Compute the arithmetic mean along the specified axis, ignoring NaNs.
782: (4) Returns the average of the array elements. The average is taken over
783: (4) the flattened array by default, otherwise over the specified axis.
784: (4) `float64` intermediate and return values are used for integer inputs.
785: (4) For all-NaN slices, NaN is returned and a `RuntimeWarning` is raised.
786: (4) .. versionadded:: 1.8.0
787: (4) Parameters
788: (4) -----
789: (4) a : array_like
790: (8) Array containing numbers whose mean is desired. If `a` is not an
791: (8) array, a conversion is attempted.
792: (4) axis : {int, tuple of int, None}, optional
793: (8) Axis or axes along which the means are computed. The default is to

```

```

compute
794: (8)           the mean of the flattened array.
795: (4)           dtype : data-type, optional
796: (8)           Type to use in computing the mean. For integer inputs, the default
797: (8)           is `float64`; for inexact inputs, it is the same as the input
798: (8)           dtype.
799: (4)           out : ndarray, optional
800: (8)           Alternate output array in which to place the result. The default
801: (8)           is ``None``; if provided, it must have the same shape as the
802: (8)           expected output, but the type will be cast if necessary. See
803: (8)           :ref:`ufuncs-output-type` for more details.
804: (4)           keepdims : bool, optional
805: (8)           If this is set to True, the axes which are reduced are left
806: (8)           in the result as dimensions with size one. With this option,
807: (8)           the result will broadcast correctly against the original `a`.
808: (8)           If the value is anything but the default, then
809: (8)           `keepdims` will be passed through to the `mean` or `sum` methods
810: (8)           of sub-classes of `ndarray`. If the sub-classes methods
811: (8)           does not implement `keepdims` any exceptions will be raised.
812: (4)           where : array_like of bool, optional
813: (8)           Elements to include in the mean. See `~numpy.ufunc.reduce` for
details.
814: (8)           .. versionadded:: 1.22.0
815: (4)           Returns
816: (4)           -----
817: (4)           m : ndarray, see dtype parameter above
818: (8)           If `out=None`, returns a new array containing the mean values,
819: (8)           otherwise a reference to the output array is returned. Nan is
820: (8)           returned for slices that contain only NaNs.
821: (4)           See Also
822: (4)           -----
823: (4)           average : Weighted average
824: (4)           mean : Arithmetic mean taken while not ignoring NaNs
825: (4)           var, nanvar
826: (4)           Notes
827: (4)           -----
828: (4)           The arithmetic mean is the sum of the non-NaN elements along the axis
829: (4)           divided by the number of non-NaN elements.
830: (4)           Note that for floating-point input, the mean is computed using the same
831: (4)           precision the input has. Depending on the input data, this can cause
832: (4)           the results to be inaccurate, especially for `float32`. Specifying a
833: (4)           higher-precision accumulator using the `dtype` keyword can alleviate
834: (4)           this issue.
835: (4)           Examples
836: (4)           -----
837: (4)           >>> a = np.array([[1, np.nan], [3, 4]])
838: (4)           >>> np.nanmean(a)
839: (4)           2.6666666666666665
840: (4)           >>> np.nanmean(a, axis=0)
841: (4)           array([2., 4.])
842: (4)           >>> np.nanmean(a, axis=1)
843: (4)           array([1., 3.5]) # may vary
844: (4)           """
845: (4)           arr, mask = _replace_nan(a, 0)
846: (4)           if mask is None:
847: (8)               return np.mean(arr, axis=axis, dtype=dtype, out=out,
keepdims=keepdims,
848: (23)                           where=where)
849: (4)           if dtype is not None:
850: (8)               dtype = np.dtype(dtype)
851: (4)           if dtype is not None and not issubclass(dtype.type, np.inexact):
852: (8)               raise TypeError("If a is inexact, then dtype must be inexact")
853: (4)           if out is not None and not issubclass(out.dtype.type, np.inexact):
854: (8)               raise TypeError("If a is inexact, then out must be inexact")
855: (4)           cnt = np.sum(~mask, axis=axis, dtype=np.intp, keepdims=keepdims,
856: (17)                           where=where)
857: (4)           tot = np.sum(arr, axis=axis, dtype=dtype, out=out, keepdims=keepdims,
858: (17)                           where=where)
859: (4)           avg = _divide_by_count(tot, cnt, out=out)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

860: (4)             isbad = (cnt == 0)
861: (4)             if isbad.any():
862: (8)                 warnings.warn("Mean of empty slice", RuntimeWarning, stacklevel=2)
863: (4)             return avg
864: (0)             def _nanmedian1d(arr1d, overwrite_input=False):
865: (4)                 """
866: (4)                 Private function for rank 1 arrays. Compute the median ignoring NaNs.
867: (4)                 See nanmedian for parameter usage
868: (4)
869: (4)                 arr1d_parsed, overwrite_input = _remove_nan_1d(
870: (8)                     arr1d, overwrite_input=overwrite_input,
871: (4)                 )
872: (4)                 if arr1d_parsed.size == 0:
873: (8)                     return arr1d[-1]
874: (4)                 return np.median(arr1d_parsed, overwrite_input=overwrite_input)
875: (0)             def _nanmedian(a, axis=None, out=None, overwrite_input=False):
876: (4)                 """
877: (4)                 Private function that doesn't support extended axis or keepdims.
878: (4)                 These methods are extended to this function using _ureduce
879: (4)                 See nanmedian for parameter usage
880: (4)
881: (4)                 if axis is None or a.ndim == 1:
882: (8)                     part = a.ravel()
883: (8)                     if out is None:
884: (12)                         return _nanmedian1d(part, overwrite_input)
885: (8)                     else:
886: (12)                         out[...] = _nanmedian1d(part, overwrite_input)
887: (12)                         return out
888: (4)                     else:
889: (8)                         if a.shape[axis] < 600:
890: (12)                             return _nanmedian_small(a, axis, out, overwrite_input)
891: (8)                         result = np.apply_along_axis(_nanmedian1d, axis, a, overwrite_input)
892: (8)                         if out is not None:
893: (12)                             out[...] = result
894: (8)                         return result
895: (0)             def _nanmedian_small(a, axis=None, out=None, overwrite_input=False):
896: (4)                 """
897: (4)                 sort + indexing median, faster for small medians along multiple
898: (4)                 dimensions due to the high overhead of apply_along_axis
899: (4)                 see nanmedian for parameter usage
900: (4)
901: (4)                 a = np.ma.masked_array(a, np.isnan(a))
902: (4)                 m = np.ma.median(a, axis=axis, overwrite_input=overwrite_input)
903: (4)                 for i in range(np.count_nonzero(m.mask.ravel())):
904: (8)                     warnings.warn("All-NaN slice encountered", RuntimeWarning,
905: (22)                         stacklevel=5)
906: (4)                 fill_value = np.timedelta64("NaT") if m.dtype.kind == "m" else np.nan
907: (4)                 if out is not None:
908: (8)                     out[...] = m.filled(fill_value)
909: (8)                     return out
910: (4)                 return m.filled(fill_value)
911: (0)             def _nanmedian_dispatcher(
912: (8)                 a, axis=None, out=None, overwrite_input=None, keepdims=None):
913: (4)                 return (a, out)
914: (0)             @array_function_dispatch(_nanmedian_dispatcher)
915: (0)             def nanmedian(a, axis=None, out=None, overwrite_input=False,
keepdims=np._NoValue):
916: (4)                 """
917: (4)                 Compute the median along the specified axis, while ignoring NaNs.
918: (4)                 Returns the median of the array elements.
919: (4)                 .. versionadded:: 1.9.0
920: (4)                 Parameters
921: (4)                 -----
922: (4)                 a : array_like
923: (8)                     Input array or object that can be converted to an array.
924: (4)                     axis : {int, sequence of int, None}, optional
925: (8)                         Axis or axes along which the medians are computed. The default
926: (8)                         is to compute the median along a flattened version of the array.
927: (8)                         A sequence of axes is supported since version 1.9.0.

```

```

928: (4)          out : ndarray, optional
929: (8)            Alternative output array in which to place the result. It must
930: (8)            have the same shape and buffer length as the expected output,
931: (8)            but the type (of the output) will be cast if necessary.
932: (4)          overwrite_input : bool, optional
933: (7)            If True, then allow use of memory of input array `a` for
934: (7)            calculations. The input array will be modified by the call to
935: (7)            `median`. This will save memory when you do not need to preserve
936: (7)            the contents of the input array. Treat the input as undefined,
937: (7)            but it will probably be fully or partially sorted. Default is
938: (7)            False. If `overwrite_input` is ``True`` and `a` is not already an
939: (7)            `ndarray`, an error will be raised.
940: (4)          keepdims : bool, optional
941: (8)            If this is set to True, the axes which are reduced are left
942: (8)            in the result as dimensions with size one. With this option,
943: (8)            the result will broadcast correctly against the original `a`.
944: (8)            If this is anything but the default value it will be passed
945: (8)            through (in the special case of an empty array) to the
946: (8)            `mean` function of the underlying array. If the array is
947: (8)            a sub-class and `mean` does not have the kwarg `keepdims` this
948: (8)            will raise a RuntimeError.
949: (4)          Returns
950: (4)          -----
951: (4)          median : ndarray
952: (8)            A new array holding the result. If the input contains integers
953: (8)            or floats smaller than ``float64``, then the output data-type is
954: (8)            ``np.float64``. Otherwise, the data-type of the output is the
955: (8)            same as that of the input. If `out` is specified, that array is
956: (8)            returned instead.
957: (4)          See Also
958: (4)          -----
959: (4)          mean, median, percentile
960: (4)          Notes
961: (4)          -----
962: (4)            Given a vector ``V`` of length ``N``, the median of ``V`` is the
963: (4)            middle value of a sorted copy of ``V``, ``V_sorted`` - i.e.,
964: (4)            ``V_sorted[(N-1)/2]``, when ``N`` is odd and the average of the two
965: (4)            middle values of ``V_sorted`` when ``N`` is even.
966: (4)          Examples
967: (4)          -----
968: (4)          >>> a = np.array([[10.0, 7, 4], [3, 2, 1]])
969: (4)          >>> a[0, 1] = np.nan
970: (4)          >>> a
971: (4)          array([[10., nan,  4.],
972: (11)             [ 3.,  2.,  1.]])
973: (4)          >>> np.median(a)
974: (4)          nan
975: (4)          >>> np.nanmedian(a)
976: (4)          3.0
977: (4)          >>> np.nanmedian(a, axis=0)
978: (4)          array([6.5, 2. , 2.5])
979: (4)          >>> np.median(a, axis=1)
980: (4)          array([nan, 2.])
981: (4)          >>> b = a.copy()
982: (4)          >>> np.nanmedian(b, axis=1, overwrite_input=True)
983: (4)          array([7., 2.])
984: (4)          >>> assert not np.all(a==b)
985: (4)          >>> b = a.copy()
986: (4)          >>> np.nanmedian(b, axis=None, overwrite_input=True)
987: (4)          3.0
988: (4)          >>> assert not np.all(a==b)
989: (4)          """
990: (4)          a = np.asarray(a)
991: (4)          if a.size == 0:
992: (8)              return np.nanmean(a, axis, out=out, keepdims=keepdims)
993: (4)          return function_base._reduce(a, func=_nanmedian, keepdims=keepdims,
994: (34)                           axis=axis, out=out,
995: (34)                           overwrite_input=overwrite_input)
996: (0)          def _nanpercentile_dispatcher(

```

```

997: (8)             a, q, axis=None, out=None, overwrite_input=None,
998: (8)             method=None, keepdims=None, *, interpolation=None):
999: (4)             return (a, q, out)
1000: (0)            @array_function_dispatch(_nanpercentile_dispatcher)
1001: (0)            def nanpercentile(
1002: (8)                a,
1003: (8)                q,
1004: (8)                axis=None,
1005: (8)                out=None,
1006: (8)                overwrite_input=False,
1007: (8)                method="linear",
1008: (8)                keepdims=np._NoValue,
1009: (8)                *,
1010: (8)                interpolation=None,
1011: (0)            ):
1012: (4)            """
1013: (4)            Compute the qth percentile of the data along the specified axis,
1014: (4)            while ignoring nan values.
1015: (4)            Returns the qth percentile(s) of the array elements.
1016: (4)            .. versionadded:: 1.9.0
1017: (4)            Parameters
1018: (4)            -----
1019: (4)            a : array_like
1020: (8)            Input array or object that can be converted to an array, containing
1021: (8)            nan values to be ignored.
1022: (4)            q : array_like of float
1023: (8)            Percentile or sequence of percentiles to compute, which must be
1024: (8)            between 0 and 100 inclusive.
1025: (4)            axis : {int, tuple of int, None}, optional
1026: (8)            Axis or axes along which the percentiles are computed. The default
1027: (8)            is to compute the percentile(s) along a flattened version of the
1028: (8)            array.
1029: (4)            out : ndarray, optional
1030: (8)            Alternative output array in which to place the result. It must have
1031: (8)            the same shape and buffer length as the expected output, but the
1032: (8)            type (of the output) will be cast if necessary.
1033: (4)            overwrite_input : bool, optional
1034: (8)            If True, then allow the input array `a` to be modified by
1035: (8)            intermediate calculations, to save memory. In this case, the
1036: (8)            contents of the input `a` after this function completes is
1037: (8)            undefined.
1038: (4)            method : str, optional
1039: (8)            This parameter specifies the method to use for estimating the
1040: (8)            percentile. There are many different methods, some unique to NumPy.
1041: (8)            See the notes for explanation. The options sorted by their R type
1042: (8)            as summarized in the H&F paper [1]_ are:
1043: (8)            1. 'inverted_cdf'
1044: (8)            2. 'averaged_inverted_cdf'
1045: (8)            3. 'closest_observation'
1046: (8)            4. 'interpolated_inverted_cdf'
1047: (8)            5. 'hazen'
1048: (8)            6. 'weibull'
1049: (8)            7. 'linear' (default)
1050: (8)            8. 'median_unbiased'
1051: (8)            9. 'normal_unbiased'
1052: (8)            The first three methods are discontinuous. NumPy further defines the
1053: (8)            following discontinuous variations of the default 'linear' (7.)
option:
1054: (8)            * 'lower'
1055: (8)            * 'higher',
1056: (8)            * 'midpoint'
1057: (8)            * 'nearest'
1058: (8)            .. versionchanged:: 1.22.0
1059: (12)            This argument was previously called "interpolation" and only
1060: (12)            offered the "linear" default and last four options.
1061: (4)            keepdims : bool, optional
1062: (8)            If this is set to True, the axes which are reduced are left in
1063: (8)            the result as dimensions with size one. With this option, the
1064: (8)            result will broadcast correctly against the original array `a`.

```

```

1065: (8)             If this is anything but the default value it will be passed
1066: (8)             through (in the special case of an empty array) to the
1067: (8)             `mean` function of the underlying array. If the array is
1068: (8)             a sub-class and `mean` does not have the kwarg `keepdims` this
1069: (8)             will raise a RuntimeError.
1070: (4)             interpolation : str, optional
1071: (8)                 Deprecated name for the method keyword argument.
1072: (8)                 .. deprecated:: 1.22.0
1073: (4)             Returns
1074: (4)
1075: (4)             percentile : scalar or ndarray
1076: (8)                 If `q` is a single percentile and `axis=None`, then the result
1077: (8)                 is a scalar. If multiple percentiles are given, first axis of
1078: (8)                 the result corresponds to the percentiles. The other axes are
1079: (8)                 the axes that remain after the reduction of `a`. If the input
1080: (8)                 contains integers or floats smaller than ``float64``, the output
1081: (8)                 data-type is ``float64``. Otherwise, the output data-type is the
1082: (8)                 same as that of the input. If `out` is specified, that array is
1083: (8)                 returned instead.
1084: (4)             See Also
1085: (4)
1086: (4)             nanmean
1087: (4)             nanmedian : equivalent to ``nanpercentile(..., 50)``
1088: (4)             percentile, median, mean
1089: (4)             nanquantile : equivalent to nanpercentile, except q in range [0, 1].
1090: (4)             Notes
1091: (4)
1092: (4)                 For more information please see `numpy.percentile`
1093: (4)             Examples
1094: (4)
1095: (4)             >>> a = np.array([[10., 7., 4.], [3., 2., 1.]])
1096: (4)             >>> a[0][1] = np.nan
1097: (4)             >>> a
1098: (4)             array([[10.,  nan,  4.],
1099: (10)                [ 3.,   2.,   1.]])
1100: (4)             >>> np.percentile(a, 50)
1101: (4)             nan
1102: (4)             >>> np.nanpercentile(a, 50)
1103: (4)             3.0
1104: (4)             >>> np.nanpercentile(a, 50, axis=0)
1105: (4)             array([6.5, 2. , 2.5])
1106: (4)             >>> np.nanpercentile(a, 50, axis=1, keepdims=True)
1107: (4)             array([[7.],
1108: (11)                [2.]])
1109: (4)             >>> m = np.nanpercentile(a, 50, axis=0)
1110: (4)             >>> out = np.zeros_like(m)
1111: (4)             >>> np.nanpercentile(a, 50, axis=0, out=out)
1112: (4)             array([6.5, 2. , 2.5])
1113: (4)             >>> m
1114: (4)             array([6.5, 2. , 2.5])
1115: (4)             >>> b = a.copy()
1116: (4)             >>> np.nanpercentile(b, 50, axis=1, overwrite_input=True)
1117: (4)             array([7., 2.])
1118: (4)             >>> assert not np.all(a==b)
1119: (4)             References
1120: (4)
1121: (4)                 .. [1] R. J. Hyndman and Y. Fan,
1122: (7)                     "Sample quantiles in statistical packages,"
1123: (7)                     The American Statistician, 50(4), pp. 361-365, 1996
1124: (4)
1125: (4)             if interpolation is not None:
1126: (8)                 method = function_base._check_interpolation_as_method(
1127: (12)                     method, interpolation, "nanpercentile")
1128: (4)             a = np.asanyarray(a)
1129: (4)             if a.dtype.kind == "c":
1130: (8)                 raise TypeError("a must be an array of real numbers")
1131: (4)             q = np.true_divide(q, 100.0)
1132: (4)             q = np.asanyarray(q)
1133: (4)             if not function_base._quantile_is_valid(q):

```

```

1134: (8)             raise ValueError("Percentiles must be in the range [0, 100]")
1135: (4)             return _nanquantile_unchecked(
1136: (8)                 a, q, axis, out, overwrite_input, method, keepdims)
1137: (0)             def _nanquantile_dispatcher(a, q, axis=None, out=None,
1138: (28)                           overwrite_input=None, method=None, keepdims=None, *,
interpolation=None):
1139: (4)                 return (a, q, out)
1140: (0)             @array_function_dispatch(_nanquantile_dispatcher)
1141: (0)             def nanquantile(
1142: (8)                 a,
1143: (8)                 q,
1144: (8)                 axis=None,
1145: (8)                 out=None,
1146: (8)                 overwrite_input=False,
1147: (8)                 method="linear",
1148: (8)                 keepdims=np._NoValue,
1149: (8)                 *,
1150: (8)                 interpolation=None,
1151: (0)             ):
1152: (4)             """
1153: (4)             Compute the qth quantile of the data along the specified axis,
1154: (4)             while ignoring nan values.
1155: (4)             Returns the qth quantile(s) of the array elements.
1156: (4)             .. versionadded:: 1.15.0
1157: (4)             Parameters
1158: (4)             -----
1159: (4)             a : array_like
1160: (8)                 Input array or object that can be converted to an array, containing
1161: (8)                 nan values to be ignored
1162: (4)             q : array_like of float
1163: (8)                 Probability or sequence of probabilities for the quantiles to compute.
1164: (8)                 Values must be between 0 and 1 inclusive.
1165: (4)             axis : {int, tuple of int, None}, optional
1166: (8)                 Axis or axes along which the quantiles are computed. The
1167: (8)                 default is to compute the quantile(s) along a flattened
1168: (8)                 version of the array.
1169: (4)             out : ndarray, optional
1170: (8)                 Alternative output array in which to place the result. It must
1171: (8)                 have the same shape and buffer length as the expected output,
1172: (8)                 but the type (of the output) will be cast if necessary.
1173: (4)             overwrite_input : bool, optional
1174: (8)                 If True, then allow the input array `a` to be modified by intermediate
1175: (8)                 calculations, to save memory. In this case, the contents of the input
1176: (8)                 `a` after this function completes is undefined.
1177: (4)             method : str, optional
1178: (8)                 This parameter specifies the method to use for estimating the
1179: (8)                 quantile. There are many different methods, some unique to NumPy.
1180: (8)                 See the notes for explanation. The options sorted by their R type
1181: (8)                 as summarized in the H&F paper [1]_ are:
1182: (8)                 1. 'inverted_cdf'
1183: (8)                 2. 'averaged_inverted_cdf'
1184: (8)                 3. 'closest_observation'
1185: (8)                 4. 'interpolated_inverted_cdf'
1186: (8)                 5. 'hazen'
1187: (8)                 6. 'weibull'
1188: (8)                 7. 'linear' (default)
1189: (8)                 8. 'median_unbiased'
1190: (8)                 9. 'normal_unbiased'
1191: (8)                 The first three methods are discontinuous. NumPy further defines the
1192: (8)                 following discontinuous variations of the default 'linear' (7.)
option:
1193: (8)                 * 'lower'
1194: (8)                 * 'higher',
1195: (8)                 * 'midpoint'
1196: (8)                 * 'nearest'
1197: (8)                 .. versionchanged:: 1.22.0
1198: (12)                   This argument was previously called "interpolation" and only
1199: (12)                   offered the "linear" default and last four options.
1200: (4)             keepdims : bool, optional

```

```

1201: (8)           If this is set to True, the axes which are reduced are left in
1202: (8)           the result as dimensions with size one. With this option, the
1203: (8)           result will broadcast correctly against the original array `a`.
1204: (8)           If this is anything but the default value it will be passed
1205: (8)           through (in the special case of an empty array) to the
1206: (8)           `mean` function of the underlying array. If the array is
1207: (8)           a sub-class and `mean` does not have the kwarg `keepdims` this
1208: (8)           will raise a RuntimeError.
1209: (4)           interpolation : str, optional
1210: (8)               Deprecated name for the method keyword argument.
1211: (8)               .. deprecated:: 1.22.0
1212: (4)           Returns
1213: (4)           -----
1214: (4)           quantile : scalar or ndarray
1215: (8)               If `q` is a single probability and `axis=None`, then the result
1216: (8)               is a scalar. If multiple probability levels are given, first axis of
1217: (8)               the result corresponds to the quantiles. The other axes are
1218: (8)               the axes that remain after the reduction of `a`. If the input
1219: (8)               contains integers or floats smaller than ``float64``, the output
1220: (8)               data-type is ``float64``. Otherwise, the output data-type is the
1221: (8)               same as that of the input. If `out` is specified, that array is
1222: (8)               returned instead.
1223: (4)           See Also
1224: (4)           -----
1225: (4)           quantile
1226: (4)           nanmean, nanmedian
1227: (4)           nanmedian : equivalent to ``nanquantile(..., 0.5)``
1228: (4)           nanpercentile : same as nanquantile, but with q in the range [0, 100].
1229: (4)           Notes
1230: (4)           -----
1231: (4)           For more information please see `numpy.quantile`'
1232: (4)           Examples
1233: (4)           -----
1234: (4)           >>> a = np.array([[10., 7., 4.], [3., 2., 1.]])
1235: (4)           >>> a[0][1] = np.nan
1236: (4)           >>> a
1237: (4)           array([[10.,  nan,   4.],
1238: (10)             [ 3.,   2.,   1.]])
1239: (4)           >>> np.quantile(a, 0.5)
1240: (4)           nan
1241: (4)           >>> np.nanquantile(a, 0.5)
1242: (4)           3.0
1243: (4)           >>> np.nanquantile(a, 0.5, axis=0)
1244: (4)           array([6.5, 2. , 2.5])
1245: (4)           >>> np.nanquantile(a, 0.5, axis=1, keepdims=True)
1246: (4)           array([[7.],
1247: (11)             [2.]])
1248: (4)           >>> m = np.nanquantile(a, 0.5, axis=0)
1249: (4)           >>> out = np.zeros_like(m)
1250: (4)           >>> np.nanquantile(a, 0.5, axis=0, out=out)
1251: (4)           array([6.5, 2. , 2.5])
1252: (4)           >>> m
1253: (4)           array([6.5, 2. , 2.5])
1254: (4)           >>> b = a.copy()
1255: (4)           >>> np.nanquantile(b, 0.5, axis=1, overwrite_input=True)
1256: (4)           array([7., 2.])
1257: (4)           >>> assert not np.all(a==b)
1258: (4)           References
1259: (4)           -----
1260: (4)           .. [1] R. J. Hyndman and Y. Fan,
1261: (7)             "Sample quantiles in statistical packages,"
1262: (7)             The American Statistician, 50(4), pp. 361-365, 1996
1263: (4)           """
1264: (4)           if interpolation is not None:
1265: (8)               method = function_base._check_interpolation_as_method(
1266: (12)                 method, interpolation, "nanquantile")
1267: (4)               a = np.asarray(a)
1268: (4)               if a.dtype.kind == "c":
1269: (8)                   raise TypeError("a must be an array of real numbers")

```

```

1270: (4)             q = np.asanyarray(q)
1271: (4)             if not function_base._quantile_is_valid(q):
1272: (8)                 raise ValueError("Quantiles must be in the range [0, 1]")
1273: (4)             return _nanquantile_unchecked(
1274: (8)                 a, q, axis, out, overwrite_input, method, keepdims)
1275: (0)             def _nanquantile_unchecked(
1276: (8)                 a,
1277: (8)                 q,
1278: (8)                 axis=None,
1279: (8)                 out=None,
1280: (8)                 overwrite_input=False,
1281: (8)                 method="linear",
1282: (8)                 keepdims=np._NoValue,
1283: (0)             ):
1284: (4)                 """Assumes that q is in [0, 1], and is an ndarray"""
1285: (4)                 if a.size == 0:
1286: (8)                     return np.nanmean(a, axis, out=out, keepdims=keepdims)
1287: (4)                 return function_base._ureduce(a,
1288: (34)                         func=_nanquantile_ureduce_func,
1289: (34)                         q=q,
1290: (34)                         keepdims=keepdims,
1291: (34)                         axis=axis,
1292: (34)                         out=out,
1293: (34)                         overwrite_input=overwrite_input,
1294: (34)                         method=method)
1295: (0)             def _nanquantile_ureduce_func(a, q, axis=None, out=None,
1296: (30)                         overwrite_input=False,
1297: (4)                         method="linear"):
1298: (4)                         """
1299: (4)                         Private function that doesn't support extended axis or keepdims.
1300: (4)                         These methods are extended to this function using _ureduce
1301: (4)                         See nanpercentile for parameter usage
1302: (4)                         """
1303: (8)                         if axis is None or a.ndim == 1:
1304: (8)                             part = a.ravel()
1305: (4)                             result = _nanquantile_1d(part, q, overwrite_input, method)
1306: (8)                         else:
1307: (37)                             result = np.apply_along_axis(_nanquantile_1d, axis, a, q,
1308: (8)                                         overwrite_input, method)
1309: (12)                             if q.ndim != 0:
1310: (4)                                 result = np.moveaxis(result, axis, 0)
1311: (8)                             if out is not None:
1312: (4)                                 out [...] = result
1313: (0)                             return result
1314: (4)             def _nanquantile_1d(arr1d, q, overwrite_input=False, method="linear"):
1315: (4)                         """
1316: (4)                         Private function for rank 1 arrays. Compute quantile ignoring NaNs.
1317: (4)                         See nanpercentile for parameter usage
1318: (4)                         """
1319: (8)                         arr1d, overwrite_input = _remove_nan_1d(arr1d,
1320: (4)                             overwrite_input=overwrite_input)
1321: (8)                         if arr1d.size == 0:
1322: (4)                             return np.full(q.shape, np.nan, dtype=arr1d.dtype)[()]
1323: (8)                         return function_base._quantile_unchecked(
1324: (0)                             arr1d, q, overwrite_input=overwrite_input, method=method)
1325: (23)             def _nanvar_dispatcher(a, axis=None, dtype=None, out=None, ddof=None,
1326: (4)                             keepdims=None, *, where=None):
1327: (0)                 return (a, out)
1328: (0)             @array_function_dispatch(_nanvar_dispatcher)
1329: (11)             def nanvar(a, axis=None, dtype=None, out=None, ddof=0, keepdims=np._NoValue,
1330: (4)                             *, where=np._NoValue):
1331: (4)                             """
1332: (4)                             Compute the variance along the specified axis, while ignoring NaNs.
1333: (4)                             Returns the variance of the array elements, a measure of the spread of
1334: (4)                             a distribution. The variance is computed for the flattened array by
1335: (4)                             default, otherwise over the specified axis.
1336: (4)                             For all-NaN slices or slices with zero degrees of freedom, NaN is
1337: (4)                             returned and a `RuntimeWarning` is raised.
.. versionadded:: 1.8.0

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1338: (4)          Parameters
1339: (4)
1340: (4)
1341: (8)          a : array_like
1342: (8)          Array containing numbers whose variance is desired. If `a` is not an
1343: (4)          array, a conversion is attempted.
1344: (8)          axis : {int, tuple of int, None}, optional
1345: (8)          Axis or axes along which the variance is computed. The default is to
compute
1346: (4)          the variance of the flattened array.
1347: (8)          dtype : data-type, optional
1348: (8)          Type to use in computing the variance. For arrays of integer type
1349: (8)          the default is `float64`; for arrays of float types it is the same as
the array type.
1350: (4)          out : ndarray, optional
1351: (8)          Alternate output array in which to place the result. It must have
1352: (8)          the same shape as the expected output, but the type is cast if
necessary.
1353: (8)
1354: (4)          ddof : int, optional
1355: (8)          "Delta Degrees of Freedom": the divisor used in the calculation is
1356: (8)          ``N - ddof``, where ``N`` represents the number of non-NaN
1357: (8)          elements. By default `ddof` is zero.
1358: (4)          keepdims : bool, optional
1359: (8)          If this is set to True, the axes which are reduced are left
1360: (8)          in the result as dimensions with size one. With this option,
1361: (8)          the result will broadcast correctly against the original `a`.
1362: (4)          where : array_like of bool, optional
1363: (8)          Elements to include in the variance. See `~numpy.ufunc.reduce` for
1364: (8)          details.
1365: (8)          .. versionadded:: 1.22.0
1366: (4)          Returns
1367: (4)
1368: (4)          variance : ndarray, see dtype parameter above
1369: (8)          If `out` is None, return a new array containing the variance,
1370: (8)          otherwise return a reference to the output array. If ddof is >= the
1371: (8)          number of non-NaN elements in a slice or the slice contains only
1372: (8)          NaNs, then the result for that slice is NaN.
1373: (4)          See Also
1374: (4)
1375: (4)          std : Standard deviation
1376: (4)          mean : Average
1377: (4)          var : Variance while not ignoring NaNs
1378: (4)          nanstd, nanmean
1379: (4)          :ref:`ufuncs-output-type`
1380: (4)          Notes
1381: (4)
1382: (4)          The variance is the average of the squared deviations from the mean,
1383: (4)          i.e., ``var = mean(abs(x - x.mean())**2)``.
1384: (4)          The mean is normally calculated as ``x.sum() / N``, where ``N = len(x)``.
1385: (4)          If, however, `ddof` is specified, the divisor ``N - ddof`` is used
1386: (4)          instead. In standard statistical practice, ``ddof=1`` provides an
1387: (4)          unbiased estimator of the variance of a hypothetical infinite
1388: (4)          population. ``ddof=0`` provides a maximum likelihood estimate of the
1389: (4)          variance for normally distributed variables.
1390: (4)          Note that for complex numbers, the absolute value is taken before
1391: (4)          squaring, so that the result is always real and nonnegative.
1392: (4)          For floating-point input, the variance is computed using the same
1393: (4)          precision the input has. Depending on the input data, this can cause
1394: (4)          the results to be inaccurate, especially for `float32` (see example
1395: (4)          below). Specifying a higher-accuracy accumulator using the ``dtype``
1396: (4)          keyword can alleviate this issue.
1397: (4)          For this function to work on sub-classes of ndarray, they must define
1398: (4)          `sum` with the kwarg `keepdims`
1399: (4)          Examples
1400: (4)
1401: (4)          >>> a = np.array([[1, np.nan], [3, 4]])
1402: (4)          >>> np.nanvar(a)
1403: (4)          1.5555555555555554
1404: (4)          >>> np.nanvar(a, axis=0)
1405: (4)          array([1.,  0.])

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1406: (4)
1407: (4)
1408: (4)
1409: (4)
1410: (4)
1411: (8)
1412: (22)
1413: (4)
1414: (8)
1415: (4)
1416: (8)
1417: (4)
1418: (8)
1419: (4)
1420: (8)
1421: (4)
1422: (8)
1423: (4)
1424: (17)
1425: (4)
1426: (4)
1427: (4)
1428: (4)
1429: (4)
1430: (8)
1431: (4)
1432: (8)
1433: (4)
1434: (17)
1435: (4)
1436: (8)
1437: (4)
1438: (8)
1439: (4)
1440: (8)
1441: (4)
1442: (4)
1443: (4)
1444: (4)
1445: (8)
1446: (22)
1447: (8)
1448: (4)
1449: (0)
1450: (23)
1451: (4)
1452: (0)
1453: (0)
1454: (11)
1455: (4)
1456: (4)
1457: (4)
1458: (4)
1459: (4)
1460: (4)
1461: (4)
1462: (4)
1463: (4)
1464: (4)
1465: (4)
1466: (4)
1467: (4)
1468: (8)
1469: (4)
1470: (8)
default is
1471: (8)
1472: (4)
1473: (8)

    >>> np.nanvar(a, axis=1)
    array([0.,  0.25]) # may vary
    """
    arr, mask = _replace_nan(a, 0)
    if mask is None:
        return np.var(arr, axis=axis, dtype=dtype, out=out, ddof=ddof,
                     keepdims=keepdims, where=where)
    if dtype is not None:
        dtype = np.dtype(dtype)
    if dtype is not None and not issubclass(dtype.type, np.inexact):
        raise TypeError("If a is inexact, then dtype must be inexact")
    if out is not None and not issubclass(out.dtype.type, np.inexact):
        raise TypeError("If a is inexact, then out must be inexact")
    if type(arr) is np.matrix:
        _keepdims = np._NoValue
    else:
        _keepdims = True
    cnt = np.sum(~mask, axis=axis, dtype=np.intp, keepdims=_keepdims,
                 where=where)
    avg = np.sum(arr, axis=axis, dtype=dtype, keepdims=_keepdims, where=where)
    avg = _divide_by_count(avg, cnt)
    np.subtract(arr, avg, out=arr, casting='unsafe', where=where)
    arr = _copyto(arr, 0, mask)
    if issubclass(arr.dtype.type, np.complexfloating):
        sqr = np.multiply(arr, arr.conj(), out=arr, where=where).real
    else:
        sqr = np.multiply(arr, arr, out=arr, where=where)
    var = np.sum(sqr, axis=axis, dtype=dtype, out=out, keepdims=keepdims,
                 where=where)
    try:
        var_ndim = var.ndim
    except AttributeError:
        var_ndim = np.ndim(var)
    if var_ndim < cnt.ndim:
        cnt = cnt.squeeze(axis)
    dof = cnt - ddof
    var = _divide_by_count(var, dof)
    isbad = (dof <= 0)
    if np.any(isbad):
        warnings.warn("Degrees of freedom <= 0 for slice.", RuntimeWarning,
                      stacklevel=2)
    var = _copyto(var, np.nan, isbad)
    return var
def _nanstd_dispatcher(a, axis=None, dtype=None, out=None, ddof=None,
                      keepdims=None, *, where=None):
    return (a, out)
@array_function_dispatch(_nanstd_dispatcher)
def nanstd(a, axis=None, dtype=None, out=None, ddof=0, keepdims=np._NoValue,
           *, where=np._NoValue):
    """
    Compute the standard deviation along the specified axis, while
    ignoring NaNs.
    Returns the standard deviation, a measure of the spread of a
    distribution, of the non-NaN array elements. The standard deviation is
    computed for the flattened array by default, otherwise over the
    specified axis.
    For all-NaN slices or slices with zero degrees of freedom, NaN is
    returned and a `RuntimeWarning` is raised.
    .. versionadded:: 1.8.0
    Parameters
    -----
    a : array_like
        Calculate the standard deviation of the non-NaN values.
    axis : {int, tuple of int, None}, optional
        Axis or axes along which the standard deviation is computed. The
        to compute the standard deviation of the flattened array.
    dtype : dtype, optional
        Type to use in computing the standard deviation. For arrays of
    """

```

```

1474: (8)             integer type the default is float64, for arrays of float types it
1475: (8)             is the same as the array type.
1476: (4)             out : ndarray, optional
1477: (8)                 Alternative output array in which to place the result. It must have
1478: (8)                 the same shape as the expected output but the type (of the
1479: (8)                 calculated values) will be cast if necessary.
1480: (4)             ddof : int, optional
1481: (8)                 Means Delta Degrees of Freedom. The divisor used in calculations
1482: (8)                 is ``N - ddof``, where ``N`` represents the number of non-NaN
1483: (8)                 elements. By default `ddof` is zero.
1484: (4)             keepdims : bool, optional
1485: (8)                 If this is set to True, the axes which are reduced are left
1486: (8)                 in the result as dimensions with size one. With this option,
1487: (8)                 the result will broadcast correctly against the original `a`.
1488: (8)                 If this value is anything but the default it is passed through
1489: (8)                 as-is to the relevant functions of the sub-classes. If these
1490: (8)                 functions do not have a `keepdims` kwarg, a RuntimeError will
1491: (8)                 be raised.
1492: (4)             where : array_like of bool, optional
1493: (8)                 Elements to include in the standard deviation.
1494: (8)                 See `~numpy.ufunc.reduce` for details.
1495: (8)             .. versionadded:: 1.22.0
1496: (4)             Returns
1497: (4)             -----
1498: (4)             standard_deviation : ndarray, see dtype parameter above.
1499: (8)                 If `out` is None, return a new array containing the standard
1500: (8)                 deviation, otherwise return a reference to the output array. If
1501: (8)                 ddof is >= the number of non-NaN elements in a slice or the slice
1502: (8)                 contains only NaNs, then the result for that slice is NaN.
1503: (4)             See Also
1504: (4)             -----
1505: (4)             var, mean, std
1506: (4)             nanvar, nanmean
1507: (4)             :ref:`ufuncs-output-type`
1508: (4)             Notes
1509: (4)             -----
1510: (4)             The standard deviation is the square root of the average of the squared
1511: (4)             deviations from the mean: ``std = sqrt(mean(abs(x - x.mean())**2))``.
1512: (4)             The average squared deviation is normally calculated as
1513: (4)             ``x.sum() / N``, where ``N = len(x)``. If, however, `ddof` is
1514: (4)             specified, the divisor ``N - ddof`` is used instead. In standard
1515: (4)             statistical practice, ``ddof=1`` provides an unbiased estimator of the
1516: (4)             variance of the infinite population. ``ddof=0`` provides a maximum
1517: (4)             likelihood estimate of the variance for normally distributed variables.
1518: (4)             The standard deviation computed in this function is the square root of
1519: (4)             the estimated variance, so even with ``ddof=1``, it will not be an
1520: (4)             unbiased estimate of the standard deviation per se.
1521: (4)             Note that, for complex numbers, `std` takes the absolute value before
1522: (4)             squaring, so that the result is always real and nonnegative.
1523: (4)             For floating-point input, the *std* is computed using the same
1524: (4)             precision the input has. Depending on the input data, this can cause
1525: (4)             the results to be inaccurate, especially for float32 (see example
1526: (4)             below). Specifying a higher-accuracy accumulator using the `dtype`
1527: (4)             keyword can alleviate this issue.
1528: (4)             Examples
1529: (4)             -----
1530: (4)             >>> a = np.array([[1, np.nan], [3, 4]])
1531: (4)             >>> np.nanstd(a)
1532: (4)             1.247219128924647
1533: (4)             >>> np.nanstd(a, axis=0)
1534: (4)             array([1., 0.])
1535: (4)             >>> np.nanstd(a, axis=1)
1536: (4)             array([0.,  0.5]) # may vary
1537: (4)             """
1538: (4)             var = nanvar(a, axis=axis, dtype=dtype, out=out, ddof=ddof,
1539: (17)                         keepdims=keepdims, where=where)
1540: (4)             if isinstance(var, np.ndarray):
1541: (8)                 std = np.sqrt(var, out=var)
1542: (4)             elif hasattr(var, 'dtype'):
```

```

1543: (8)         std = var.dtype.type(np.sqrt(var))
1544: (4)     else:
1545: (8)         std = np.sqrt(var)
1546: (4)     return std
-----
```

File 213 - npyio.py:

```

1: (0)         import os
2: (0)         import re
3: (0)         import functools
4: (0)         import itertools
5: (0)         import warnings
6: (0)         import weakref
7: (0)         import contextlib
8: (0)         import operator
9: (0)         from operator import itemgetter, index as opindex, methodcaller
10: (0)        from collections.abc import Mapping
11: (0)        import numpy as np
12: (0)        from . import format
13: (0)        from ._datasource import DataSource
14: (0)        from numpy.core import overrides
15: (0)        from numpy.core.multiarray import packbits, unpackbits
16: (0)        from numpy.core._multiarray_umath import _load_from_filelike
17: (0)        from numpy.core.overrides import set_array_function_like_doc, set_module
18: (0)        from ._iotools import (
19: (4)            LineSplitter, NameValidator, StringConverter, ConverterError,
20: (4)            ConverterLockError, ConversionWarning, _is_string_like,
21: (4)            has_nested_fields, flatten_dtype, easy_dtype, _decode_line
22: (4)        )
23: (0)        from numpy.compat import (
24: (4)            asbytes, asstr, asunicode, os_fspath, os_PathLike,
25: (4)            pickle
26: (4)        )
27: (0)        __all__ = [
28: (4)            'savetxt', 'loadtxt', 'genfromtxt',
29: (4)            'recfromtxt', 'recfromcsv', 'load', 'save', 'savez',
30: (4)            'savez_compressed', 'packbits', 'unpackbits', 'fromregex', 'DataSource'
31: (4)        ]
32: (0)        array_function_dispatch = functools.partial(
33: (4)            overrides.array_function_dispatch, module='numpy')
34: (0)        class BagObj:
35: (4)            """
36: (4)            BagObj(obj)
37: (4)            Convert attribute look-ups to getitems on the object passed in.
38: (4)            Parameters
39: (4)            -----
40: (4)            obj : class instance
41: (8)            Object on which attribute look-up is performed.
42: (4)            Examples
43: (4)            -----
44: (4)            >>> from numpy.lib.npyio import BagObj as BO
45: (4)            >>> class BagDemo:
46: (4)                def __getitem__(self, key): # An instance of BagObj(BagDemo)
47: (4)                    # will call this method when any
48: (4)                    # attribute look-up is required
49: (4)                    result = "Doesn't matter what you want, "
50: (4)                    return result + "you're gonna get this"
51: (4)
52: (4)                >>> demo_obj = BagDemo()
53: (4)                >>> bagobj = BO(demo_obj)
54: (4)                >>> bagobj.hello_there
55: (4)                "Doesn't matter what you want, you're gonna get this"
56: (4)                >>> bagobj.I_can_be_anything
57: (4)                "Doesn't matter what you want, you're gonna get this"
58: (4)
59: (4)                def __init__(self, obj):
60: (8)                    self._obj = weakref.proxy(obj)
```

```

61: (4) def __getattribute__(self, key):
62: (8)     try:
63: (12)         return object.__getattribute__(self, '_obj')[key]
64: (8)     except KeyError:
65: (12)         raise AttributeError(key) from None
66: (4) def __dir__(self):
67: (8)     """
68: (8)         Enables dir(bagobj) to list the files in an NpzFile.
69: (8)         This also enables tab-completion in an interpreter or IPython.
70: (8)     """
71: (8)     return list(object.__getattribute__(self, '_obj').keys())
72: (0) def zipfile_factory(file, *args, **kwargs):
73: (4)     """
74: (4)         Create a ZipFile.
75: (4)         Allows for Zip64, and the `file` argument can accept file, str, or
76: (4)         pathlib.Path objects. `args` and `kwargs` are passed to the
zipfile.ZipFile
77: (4)     constructor.
78: (4)     """
79: (4)     if not hasattr(file, 'read'):
80: (8)         file = os_fspath(file)
81: (4)     import zipfile
82: (4)     kwargs['allowZip64'] = True
83: (4)     return zipfile.ZipFile(file, *args, **kwargs)
84: (0) class NpzFile(Mapping):
85: (4)     """
86: (4)         NpzFile(fid)
87: (4)         A dictionary-like object with lazy-loading of files in the zipped
88: (4)         archive provided on construction.
89: (4)         `NpzFile` is used to load files in the NumPy ``.npz`` data archive
90: (4)         format. It assumes that files in the archive have a ``.npy`` extension,
91: (4)         other files are ignored.
92: (4)         The arrays and file strings are lazily loaded on either
93: (4)         getitem access using ``obj['key']`` or attribute lookup using
94: (4)         ``obj.f.key``. A list of all files (without ``.npy`` extensions) can
95: (4)         be obtained with ``obj.files`` and the ZipFile object itself using
96: (4)         ``obj.zip``.
97: (4)         Attributes
98: (4)         -----
99: (4)         files : list of str
100: (8)             List of all files in the archive with a ``.npy`` extension.
101: (4)         zip : ZipFile instance
102: (8)             The ZipFile object initialized with the zipped archive.
103: (4)         f : BagObj instance
104: (8)             An object on which attribute can be performed as an alternative
105: (8)             to getitem access on the `NpzFile` instance itself.
106: (4)         allow_pickle : bool, optional
107: (8)             Allow loading pickled data. Default: False
108: (8)             .. versionchanged:: 1.16.3
109: (12)                 Made default False in response to CVE-2019-6446.
110: (4)         pickle_kwds : dict, optional
111: (8)             Additional keyword arguments to pass on to pickle.load.
112: (8)             These are only useful when loading object arrays saved on
113: (8)             Python 2 when using Python 3.
114: (4)         max_header_size : int, optional
115: (8)             Maximum allowed size of the header. Large headers may not be safe
116: (8)             to load securely and thus require explicitly passing a larger value.
117: (8)             See :py:func:`ast.literal_eval()` for details.
118: (8)             This option is ignored when `allow_pickle` is passed. In that case
119: (8)             the file is by definition trusted and the limit is unnecessary.
120: (4)         Parameters
121: (4)         -----
122: (4)         fid : file or str
123: (8)             The zipped archive to open. This is either a file-like object
124: (8)             or a string containing the path to the archive.
125: (4)         own_fid : bool, optional
126: (8)             Whether NpzFile should close the file handle.
127: (8)             Requires that `fid` is a file-like object.
128: (4)         Examples

```

```

129: (4)          -----
130: (4)          >>> from tempfile import TemporaryFile
131: (4)          >>> outfile = TemporaryFile()
132: (4)          >>> x = np.arange(10)
133: (4)          >>> y = np.sin(x)
134: (4)          >>> np.savez(outfile, x=x, y=y)
135: (4)          >>> _ = outfile.seek(0)
136: (4)          >>> npz = np.load(outfile)
137: (4)          >>> isinstance(npz, np.lib.npyio.NpzFile)
138: (4)          True
139: (4)          >>> npz
140: (4)          NpzFile 'object' with keys x, y
141: (4)          >>> sorted(npz.files)
142: (4)          ['x', 'y']
143: (4)          >>> npz['x'] # getitem access
144: (4)          array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
145: (4)          >>> npz.f.x # attribute lookup
146: (4)          array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
147: (4)          """
148: (4)          zip = None
149: (4)          fid = None
150: (4)          _MAX_REPR_ARRAY_COUNT = 5
151: (4)          def __init__(self, fid, own_fid=False, allow_pickle=False,
152: (17)                  pickle_kw_args=None, *,
153: (17)                  max_header_size=format._MAX_HEADER_SIZE):
154: (8)          _zip = zipfile_factory(fid)
155: (8)          self._files = _zip.namelist()
156: (8)          self._files = []
157: (8)          self.allow_pickle = allow_pickle
158: (8)          self.max_header_size = max_header_size
159: (8)          self.pickle_kw_args = pickle_kw_args
160: (8)          for x in self._files:
161: (12)              if x.endswith('.npy'):
162: (16)                  self._files.append(x[:-4])
163: (12)              else:
164: (16)                  self._files.append(x)
165: (8)          self.zip = _zip
166: (8)          self.f = BagObj(self)
167: (8)          if own_fid:
168: (12)              self.fid = fid
169: (4)          def __enter__(self):
170: (8)              return self
171: (4)          def __exit__(self, exc_type, exc_value, traceback):
172: (8)              self.close()
173: (4)          def close(self):
174: (8)          """
175: (8)              Close the file.
176: (8)          """
177: (8)          if self.zip is not None:
178: (12)              self.zip.close()
179: (12)              self.zip = None
180: (8)          if self.fid is not None:
181: (12)              self.fid.close()
182: (12)              self.fid = None
183: (8)          self.f = None # break reference cycle
184: (4)          def __del__(self):
185: (8)              self.close()
186: (4)          def __iter__(self):
187: (8)              return iter(self._files)
188: (4)          def __len__(self):
189: (8)              return len(self._files)
190: (4)          def __getitem__(self, key):
191: (8)              member = False
192: (8)              if key in self._files:
193: (12)                  member = True
194: (8)              elif key in self._files:
195: (12)                  member = True
196: (12)                  key += '.npy'
197: (8)              if member:

```

```

198: (12)             bytes = self.zip.open(key)
199: (12)             magic = bytes.read(len(format.MAGIC_PREFIX))
200: (12)             bytes.close()
201: (12)             if magic == format.MAGIC_PREFIX:
202: (16)                 bytes = self.zip.open(key)
203: (16)                 return format.read_array(bytes,
204: (41)                               allow_pickle=self.allow_pickle,
205: (41)                               pickle_kwarg=self.pickle_kwarg,
206: (41)                               max_header_size=self.max_header_size)
207: (12)             else:
208: (16)                 return self.zip.read(key)
209: (8)             else:
210: (12)                 raise KeyError(f"{key} is not a file in the archive")
211: (4)             def __contains__(self, key):
212: (8)                 return (key in self._files or key in self.files)
213: (4)             def __repr__(self):
214: (8)                 if isinstance(self.fid, str):
215: (12)                     filename = self.fid
216: (8)                 else:
217: (12)                     filename = getattr(self.fid, "name", "object")
218: (8)                     array_names = ', '.join(self.files[:self._MAX_REPR_ARRAY_COUNT])
219: (8)                     if len(self.files) > self._MAX_REPR_ARRAY_COUNT:
220: (12)                         array_names += "..."
221: (8)                     return f"NpzFile {filename!r} with keys: {array_names}"
222: (0)             @set_module('numpy')
223: (0)             def load(file, mmap_mode=None, allow_pickle=False, fix_imports=True,
224: (9)                           encoding='ASCII', *, max_header_size=format._MAX_HEADER_SIZE):
225: (4)             """
226: (4)                 Load arrays or pickled objects from ``.npy``, ``.npz`` or pickled files.
227: (4)                 .. warning:: Loading files that contain object arrays uses the ``pickle``
228: (17)                     module, which is not secure against erroneous or maliciously
229: (17)                     constructed data. Consider passing ``allow_pickle=False`` to
230: (17)                     load data that is known not to contain object arrays for the
231: (17)                     safer handling of untrusted sources.
232: (4)             Parameters
233: (4)             -----
234: (4)             file : file-like object, string, or pathlib.Path
235: (8)                 The file to read. File-like objects must support the
236: (8)                 ``seek()`` and ``read()`` methods and must always
237: (8)                 be opened in binary mode. Pickled files require that the
238: (8)                 file-like object support the ``readline()`` method as well.
239: (4)             mmap_mode : {None, 'r+', 'r', 'w+', 'c'}, optional
240: (8)                 If not None, then memory-map the file, using the given mode (see
241: (8)                   `numpy.memmap` for a detailed description of the modes). A
242: (8)                   memory-mapped array is kept on disk. However, it can be accessed
243: (8)                   and sliced like any ndarray. Memory mapping is especially useful
244: (8)                   for accessing small fragments of large files without reading the
245: (8)                   entire file into memory.
246: (4)             allow_pickle : bool, optional
247: (8)                 Allow loading pickled object arrays stored in npy files. Reasons for
248: (8)                 disallowing pickles include security, as loading pickled data can
249: (8)                 execute arbitrary code. If pickles are disallowed, loading object
250: (8)                 arrays will fail. Default: False
251: (8)                 .. versionchanged:: 1.16.3
252: (12)                     Made default False in response to CVE-2019-6446.
253: (4)             fix_imports : bool, optional
254: (8)                 Only useful when loading Python 2 generated pickled files on Python 3,
255: (8)                 which includes npy/npz files containing object arrays. If
`fix_imports`:
256: (8)                 is True, pickle will try to map the old Python 2 names to the new
names
257: (8)                 used in Python 3.
258: (4)             encoding : str, optional
259: (8)                 What encoding to use when reading Python 2 strings. Only useful when
260: (8)                 loading Python 2 generated pickled files in Python 3, which includes
261: (8)                 npy/npz files containing object arrays. Values other than 'latin1',
262: (8)                 'ASCII', and 'bytes' are not allowed, as they can corrupt numerical
263: (8)                 data. Default: 'ASCII'
264: (4)             max_header_size : int, optional

```

```

265: (8)                         Maximum allowed size of the header. Large headers may not be safe
266: (8)                         to load securely and thus require explicitly passing a larger value.
267: (8)                         See :py:func:`ast.literal_eval()` for details.
268: (8)                         This option is ignored when `allow_pickle` is passed. In that case
269: (8)                         the file is by definition trusted and the limit is unnecessary.
270: (4)                         Returns
271: (4)                         -----
272: (4)                         result : array, tuple, dict, etc.
273: (8)                         Data stored in the file. For ``.npz`` files, the returned instance
274: (8)                         of NpzFile class must be closed to avoid leaking file descriptors.
275: (4)                         Raises
276: (4)                         -----
277: (4)                         OSError
278: (8)                         If the input file does not exist or cannot be read.
279: (4)                         UnpicklingError
280: (8)                         If ``allow_pickle=True``, but the file cannot be loaded as a pickle.
281: (4)                         ValueError
282: (8)                         The file contains an object array, but ``allow_pickle=False`` given.
283: (4)                         EOFError
284: (8)                         When calling ``np.load`` multiple times on the same file handle,
285: (8)                         if all data has already been read
286: (4)                         See Also
287: (4)                         -----
288: (4)                         save, savez, savez_compressed, loadtxt
289: (4)                         memmap : Create a memory-map to an array stored in a file on disk.
290: (4)                         lib.format.open_memmap : Create or load a memory-mapped ``.npy`` file.
291: (4)                         Notes
292: (4)                         -----
293: (4)                         - If the file contains pickle data, then whatever object is stored
294: (6)                         in the pickle is returned.
295: (4)                         - If the file is a ``.npy`` file, then a single array is returned.
296: (4)                         - If the file is a ``.npz`` file, then a dictionary-like object is
297: (6)                         returned, containing ``{filename: array}`` key-value pairs, one for
298: (6)                         each file in the archive.
299: (4)                         - If the file is a ``.npz`` file, the returned value supports the
300: (6)                         context manager protocol in a similar fashion to the open function::
301: (8)                         with load('foo.npz') as data:
302: (12)                         a = data['a']
303: (6)                         The underlying file descriptor is closed when exiting the 'with'
304: (6)                         block.
305: (4)                         Examples
306: (4)                         -----
307: (4)                         Store data to disk, and load it again:
308: (4)                         >>> np.save('/tmp/123', np.array([[1, 2, 3], [4, 5, 6]]))
309: (4)                         >>> np.load('/tmp/123.npy')
310: (4)                         array([[1, 2, 3],
311: (11)                           [4, 5, 6]])
312: (4)                         Store compressed data to disk, and load it again:
313: (4)                         >>> a=np.array([[1, 2, 3], [4, 5, 6]])
314: (4)                         >>> b=np.array([1, 2])
315: (4)                         >>> np.savez('/tmp/123.npz', a=a, b=b)
316: (4)                         >>> data = np.load('/tmp/123.npz')
317: (4)                         >>> data['a']
318: (4)                         array([[1, 2, 3],
319: (11)                           [4, 5, 6]])
320: (4)                         >>> data['b']
321: (4)                         array([1, 2])
322: (4)                         >>> data.close()
323: (4)                         Mem-map the stored array, and then access the second row
324: (4)                         directly from disk:
325: (4)                         >>> X = np.load('/tmp/123.npy', mmap_mode='r')
326: (4)                         >>> X[1, :]
327: (4)                         memmap([4, 5, 6])
328: (4)                         """
329: (4)                         if encoding not in ('ASCII', 'latin1', 'bytes'):
330: (8)                             raise ValueError("encoding must be 'ASCII', 'latin1', or 'bytes'")
331: (4)                         pickle_kwds = dict(encoding=encoding, fix_imports=fix_imports)
332: (4)                         with contextlib.ExitStack() as stack:
333: (8)                             if hasattr(file, 'read'): 
```

```

334: (12)             fid = file
335: (12)             own_fid = False
336: (8)
337: (12)             else:
338: (12)                 fid = stack.enter_context(open(os_fspath(file), "rb"))
339: (8)                 own_fid = True
340: (8)                 _ZIP_PREFIX = b'PK\x03\x04'
341: (8)                 _ZIP_SUFFIX = b'PK\x05\x06' # empty zip files start with this
342: (8)                 N = len(format.MAGIC_PREFIX)
343: (8)                 magic = fid.read(N)
344: (12)                 if not magic:
345: (8)                     raise EOFError("No data left in file")
346: (8)                 fid.seek(-min(N, len(magic)), 1) # back-up
347: (12)                 if magic.startswith(_ZIP_PREFIX) or magic.startswith(_ZIP_SUFFIX):
348: (12)                     stack.pop_all()
349: (26)                     ret = NpzFile(fid, own_fid=own_fid, allow_pickle=allow_pickle,
350: (26)                                     pickle_kwds=pickle_kwds,
351: (12)                                     max_header_size=max_header_size)
352: (8)
353: (12)                     return ret
354: (16)                 elif magic == format.MAGIC_PREFIX:
355: (20)                     if mmap_mode:
356: (16)                         if allow_pickle:
357: (42)                             max_header_size = 2**64
358: (12)                         return format.open_memmap(file, mode=mmap_mode,
359: (16)                                         max_header_size=max_header_size)
360: (41)                     else:
361: (41)                         return format.read_array(fid, allow_pickle=allow_pickle,
362: (8)                                         pickle_kwds=pickle_kwds,
363: (12)                                         max_header_size=max_header_size)
364: (16)                 else:
365: (33)                     if not allow_pickle:
366: (12)                         raise ValueError("Cannot load file containing pickled data "
367: (16)                             "when allow_pickle=False")
368: (12)                     try:
369: (16)                         return pickle.load(fid, **pickle_kwds)
370: (20)                     except Exception as e:
371: (0)                         raise pickle.UnpicklingError(
372: (4)                             f"Failed to interpret file {file!r} as a pickle") from e
373: (0)             def _save_dispatcher(file, arr, allow_pickle=None, fix_imports=None):
374: (0)                 return (arr,)
375: (4)             @array_function_dispatch(_save_dispatcher)
376: (4)             def save(file, arr, allow_pickle=True, fix_imports=True):
377: (4)                 """
378: (4)                 Save an array to a binary file in NumPy ``.npy`` format.
379: (4)                 Parameters
380: (4)                 -----
381: (4)                 file : file, str, or pathlib.Path
382: (8)                     File or filename to which the data is saved. If file is a file-
383: (8)                     then the filename is unchanged. If file is a string or Path, a
384: (8)                     extension will be appended to the filename if it does not already
385: (8)                     have one.
386: (4)                 arr : array_like
387: (8)                     Array data to be saved.
388: (8)                 allow_pickle : bool, optional
389: (8)                     Allow saving object arrays using Python pickles. Reasons for
390: (8)                     pickles include security (loading pickled data can execute arbitrary
391: (8)                     code) and portability (pickled objects may not be loadable on
392: (8)                     different
393: (8)                     libraries
394: (4)                     that are not available, and not all pickled data is compatible between
395: (8)                     Python 2 and Python 3).
396: (8)                     Default: True
397: (8)                 fix_imports : bool, optional
398: (8)                     Only useful in forcing objects in object arrays on Python 3 to be
399: (8)                     pickled in a Python 2 compatible way. If `fix_imports` is True, pickle
400: (8)                     will try to map the new Python 3 names to the old module names used in

```

```

398: (8)                                Python 2, so that the pickle data stream is readable with Python 2.
399: (4)                                See Also
400: (4)
401: (4)                                savez : Save several arrays into a ``.npz`` archive
402: (4)                                savetxt, load
403: (4)                                Notes
404: (4)
405: (4)                                For a description of the ``.npy`` format, see :py:mod:`numpy.lib.format`.
406: (4)                                Any data saved to the file is appended to the end of the file.
407: (4)                                Examples
408: (4)
409: (4)                                >>> from tempfile import TemporaryFile
410: (4)                                >>> outfile = TemporaryFile()
411: (4)                                >>> x = np.arange(10)
412: (4)                                >>> np.save(outfile, x)
413: (4)                                >>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening
file
414: (4)                                >>> np.load(outfile)
415: (4)                                array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
416: (4)                                >>> with open('test.npy', 'wb') as f:
417: (4)                                    ...     np.save(f, np.array([1, 2]))
418: (4)                                    ...     np.save(f, np.array([1, 3]))
419: (4)                                >>> with open('test.npy', 'rb') as f:
420: (4)                                    ...     a = np.load(f)
421: (4)                                    ...     b = np.load(f)
422: (4)                                >>> print(a, b)
423: (4)
424: (4)                                """
425: (8)                                if hasattr(file, 'write'):
426: (4)                                    file_ctx = contextlib.nullcontext(file)
427: (8)                                else:
428: (8)                                    file = os_fspath(file)
429: (12)                                    if not file.endswith('.npy'):
430: (8)                                        file = file + '.npy'
431: (4)                                    file_ctx = open(file, "wb")
432: (8)                                with file_ctx as fid:
433: (8)                                    arr = np.asanyarray(arr)
434: (27)                                    format.write_array(fid, arr, allow_pickle=allow_pickle,
435: (0)                                         pickle_kwds=dict(fix_imports=fix_imports))
def _savez_dispatcher(file, *args, **kwds):
436: (4)                                yield from args
437: (4)                                yield from kwds.values()
@array_function_dispatch(_savez_dispatcher)
439: (0)                                def savez(file, *args, **kwds):
440: (4)                                    """Save several arrays into a single file in uncompressed ``.npz`` format.
441: (4)                                    Provide arrays as keyword arguments to store them under the
442: (4)                                    corresponding name in the output file: ``savez(fn, x=x, y=y)``.
443: (4)                                    If arrays are specified as positional arguments, i.e., ``savez(fn,
444: (4)                                    x, y)``, their names will be `arr_0`, `arr_1`, etc.
445: (4)                                    Parameters
446: (4)
447: (4)                                    file : str or file
448: (8)                                    Either the filename (string) or an open file (file-like object)
449: (8)                                    where the data will be saved. If file is a string or a Path, the
450: (8)                                    ``.npz`` extension will be appended to the filename if it is not
451: (8)                                    already there.
452: (4)                                    args : Arguments, optional
453: (8)                                    Arrays to save to the file. Please use keyword arguments (see
454: (8)                                    `kwds` below) to assign names to arrays. Arrays specified as
455: (8)                                    args will be named "arr_0", "arr_1", and so on.
456: (4)                                    kwds : Keyword arguments, optional
457: (8)                                    Arrays to save to the file. Each array will be saved to the
458: (8)                                    output file with its corresponding keyword name.
459: (4)                                    Returns
460: (4)
461: (4)                                    None
462: (4)                                    See Also
463: (4)
464: (4)                                    save : Save a single array to a binary file in NumPy format.
465: (4)                                    savetxt : Save an array to a file as plain text.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

466: (4)          savez_compressed : Save several arrays into a compressed ``.npz`` archive
467: (4)          Notes
468: (4)
469: (4)          -----
470: (4)          The ``.npz`` file format is a zipped archive of files named after the
471: (4)          variables they contain. The archive is not compressed and each file
472: (4)          in the archive contains one variable in ``.npy`` format. For a
473: (4)          description of the ``.npy`` format, see :py:mod:`numpy.lib.format`.
474: (4)          When opening the saved ``.npz`` file with `load` a `NpzFile` object is
475: (4)          returned. This is a dictionary-like object which can be queried for
476: (4)          its list of arrays (with the ``.files`` attribute), and for the arrays
477: (4)          themselves.
478: (4)          Keys passed in `kwds` are used as filenames inside the ZIP archive.
479: (4)          Therefore, keys should be valid filenames; e.g., avoid keys that begin
480: (4)          with
481: (4)          ``/`` or contain ``...``.
482: (4)          When naming variables with keyword arguments, it is not possible to name a
483: (4)          variable ``file``, as this would cause the ``file`` argument to be defined
484: (4)          twice in the call to ``savez``.
485: (4)          Examples
486: (4)          -----
487: (4)          >>> from tempfile import TemporaryFile
488: (4)          >>> outfile = TemporaryFile()
489: (4)          >>> x = np.arange(10)
490: (4)          >>> y = np.sin(x)
491: (4)          Using `savez` with ``*args, the arrays are saved with default names.
492: (4)          >>> np.savez(outfile, x, y)
493: (4)          >>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening
494: (4)          file
495: (4)          >>> npzfile = np.load(outfile)
496: (4)          >>> npzfile.files
497: (4)          ['arr_0', 'arr_1']
498: (4)          >>> npzfile['arr_0']
499: (4)          array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
500: (4)          Using `savez` with ``**kwds, the arrays are saved with the keyword names.
501: (4)          >>> outfile = TemporaryFile()
502: (4)          >>> np.savez(outfile, x=x, y=y)
503: (4)          >>> _ = outfile.seek(0)
504: (4)          >>> npzfile = np.load(outfile)
505: (4)          >>> sorted(npzfile.files)
506: (4)          ['x', 'y']
507: (4)          >>> npzfile['x']
508: (0)          >>> _ = savez(file, args, kwds, False)
509: (4)          def _savez_compressed_dispatcher(file, *args, **kwds):
510: (4)              yield from args
511: (0)              yield from kwds.values()
512: (0)          @array_function_dispatch(_savez_compressed_dispatcher)
513: (4)          def savez_compressed(file, *args, **kwds):
514: (4)              """
515: (4)              Save several arrays into a single file in compressed ``.npz`` format.
516: (4)              Provide arrays as keyword arguments to store them under the
517: (4)              corresponding name in the output file: ``savez(fn, x=x, y=y)``.
518: (4)              If arrays are specified as positional arguments, i.e., ``savez(fn,
519: (4)              x, y)``, their names will be `arr_0`, `arr_1`, etc.
520: (4)              Parameters
521: (4)              -----
522: (8)              file : str or file
523: (8)                  Either the filename (string) or an open file (file-like object)
524: (8)                  where the data will be saved. If file is a string or a Path, the
525: (8)                  ``.npz`` extension will be appended to the filename if it is not
526: (4)                  already there.
527: (8)              args : Arguments, optional
528: (8)                  Arrays to save to the file. Please use keyword arguments (see
529: (8)                  `kwds` below) to assign names to arrays. Arrays specified as
530: (8)                  args will be named "arr_0", "arr_1", and so on.
531: (8)              kwds : Keyword arguments, optional
532: (8)                  Arrays to save to the file. Each array will be saved to the

```

```

533: (4)                                Returns
534: (4)                                -----
535: (4)                                None
536: (4)                                See Also
537: (4)                                -----
538: (4)                                numpy.save : Save a single array to a binary file in NumPy format.
539: (4)                                numpy.savetxt : Save an array to a file as plain text.
540: (4)                                numpy.savez : Save several arrays into an uncompressed ``.npz`` file
format
541: (4)                                numpy.load : Load the files created by savez_compressed.
542: (4)                                Notes
543: (4)                                -----
544: (4)                                The ``.npz`` file format is a zipped archive of files named after the
545: (4)                                variables they contain. The archive is compressed with
546: (4)                                ``zipfile.ZIP_DEFLATED`` and each file in the archive contains one
variable
547: (4)                                in ``.npy`` format. For a description of the ``.npy`` format, see
548: (4)                                :py:mod:`numpy.lib.format`.
549: (4)                                When opening the saved ``.npz`` file with `load` a `NpzFile` object is
550: (4)                                returned. This is a dictionary-like object which can be queried for
551: (4)                                its list of arrays (with the ``.files`` attribute), and for the arrays
552: (4)                                themselves.
553: (4)                                Examples
554: (4)                                -----
555: (4)                                >>> test_array = np.random.rand(3, 2)
556: (4)                                >>> test_vector = np.random.rand(4)
557: (4)                                >>> np.savez_compressed('/tmp/123', a=test_array, b=test_vector)
558: (4)                                >>> loaded = np.load('/tmp/123.npz')
559: (4)                                >>> print(np.array_equal(test_array, loaded['a']))
560: (4)                                True
561: (4)                                >>> print(np.array_equal(test_vector, loaded['b']))
562: (4)                                True
563: (4)                                """
564: (4)                                _savez(file, args, kwds, True)
565: (0)                                def _savez(file, args, kwds, compress, allow_pickle=True, pickle_kwarg=None):
566: (4)                                import zipfile
567: (4)                                if not hasattr(file, 'write'):
568: (8)                                    file = os_fspath(file)
569: (8)                                    if not file.endswith('.npz'):
570: (12)                                        file = file + '.npz'
571: (4)                                namedict = kwds
572: (4)                                for i, val in enumerate(args):
573: (8)                                    key = 'arr_%d' % i
574: (8)                                    if key in namedict.keys():
575: (12)                                        raise ValueError(
576: (16)                                            "Cannot use un-named variables and keyword %s" % key)
577: (8)                                        namedict[key] = val
578: (4)                                if compress:
579: (8)                                    compression = zipfile.ZIP_DEFLATED
580: (4)                                else:
581: (8)                                    compression = zipfile.ZIP_STORED
582: (4)                                zipf = zipfile_factory(file, mode="w", compression=compression)
583: (4)                                for key, val in namedict.items():
584: (8)                                    fname = key + '.npy'
585: (8)                                    val = np.asarray(val)
586: (8)                                    with zipf.open(fname, 'w', force_zip64=True) as fid:
587: (12)                                        format.write_array(fid, val,
588: (31)                                            allow_pickle=allow_pickle,
589: (31)                                            pickle_kwarg=pickle_kwarg)
590: (4)                                zipf.close()
591: (0)                                def _ensure_ndmin_ndarray_check_param(ndmin):
592: (4)                                """Just checks if the param ndmin is supported on
593: (8)                                    _ensure_ndmin_ndarray. It is intended to be used as
594: (8)                                    verification before running anything expensive.
595: (8)                                    e.g. loadtxt, genfromtxt
596: (4)                                """
597: (4)                                if ndmin not in [0, 1, 2]:
598: (8)                                    raise ValueError(f"Illegal value of ndmin keyword: {ndmin}")
599: (0)                                def _ensure_ndmin_ndarray(a, *, ndmin: int):

```

```

600: (4)         """This is a helper function of loadtxt and genfromtxt to ensure
601: (8)           proper minimum dimension as requested
602: (8)           ndim : int. Supported values 1, 2, 3
603: (20)          ^^^ whenever this changes, keep in sync with
604: (23)          _ensure_ndmin_ndarray_check_param
605: (4)
606: (4)          """
607: (8)          if a.ndim > ndmin:
608: (4)              a = np.squeeze(a)
609: (8)          if a.ndim < ndmin:
610: (12)              if ndmin == 1:
611: (8)                  a = np.atleast_1d(a)
612: (12)              elif ndmin == 2:
613: (8)                  a = np.atleast_2d(a).T
614: (4)          return a
615: (0)      _loadtxt_chunksize = 50000
616: (0)      def _check_nonneg_int(value, name="argument"):
617: (4)          try:
618: (8)              operator.index(value)
619: (4)          except TypeError:
620: (8)              raise TypeError(f"{name} must be an integer") from None
621: (4)          if value < 0:
622: (8)              raise ValueError(f"{name} must be nonnegative")
623: (0)      def _preprocess_comments(iterable, comments, encoding):
624: (4)          """
625: (4)              Generator that consumes a line iterated iterable and strips out the
626: (4)              multiple (or multi-character) comments from lines.
627: (4)              This is a pre-processing step to achieve feature parity with loadtxt
628: (4)              (we assume that this feature is a nieche feature).
629: (4)
630: (8)          for line in iterable:
631: (12)              if isinstance(line, bytes):
632: (8)                  line = line.decode(encoding)
633: (12)              for c in comments:
634: (8)                  line = line.split(c, 1)[0]
635: (8)              yield line
636: (0)      _loadtxt_chunksize = 50000
637: (0)      def _read(fname, *, delimiter=',', comment='#', quote='',
638: (10)          imaginary_unit='j', usecols=None, skiplines=0,
639: (10)          max_rows=None, converters=None, ndmin=None, unpack=False,
640: (10)          dtype=np.float64, encoding="bytes"):
641: (4)          """
642: (4)              Read a NumPy array from a text file.
643: (4)              Parameters
644: (4)              -----
645: (8)              fname : str or file object
646: (8)                  The filename or the file to be read.
647: (4)              delimiter : str, optional
648: (8)                  Field delimiter of the fields in line of the file.
649: (8)                  Default is a comma, ','. If None any sequence of whitespace is
650: (8)                  considered a delimiter.
651: (4)              comment : str or sequence of str or None, optional
652: (8)                  Character that begins a comment. All text from the comment
653: (8)                  character to the end of the line is ignored.
654: (8)                  Multiple comments or multiple-character comment strings are supported,
655: (8)                  but may be slower and `quote` must be empty if used.
656: (8)                  Use None to disable all use of comments.
657: (4)              quote : str or None, optional
658: (8)                  Character that is used to quote string fields. Default is ''
659: (8)                  (a double quote). Use None to disable quote support.
660: (4)              imaginary_unit : str, optional
661: (8)                  Character that represent the imaginay unit `sqrt(-1)` .
662: (8)                  Default is 'j'.
663: (4)              usecols : array_like, optional
664: (8)                  A one-dimensional array of integer column numbers. These are the
665: (8)                  columns from the file to be included in the array. If this value
666: (8)                  is not given, all the columns are used.
667: (4)              skiplines : int, optional
668: (8)                  Number of lines to skip before interpreting the data in the file.
669: (4)              max_rows : int, optional

```

```

669: (8)                         Maximum number of rows of data to read. Default is to read the
670: (8)                         entire file.
671: (4)                         converters : dict or callable, optional
672: (8)                         A function to parse all columns strings into the desired value, or
673: (8)                         a dictionary mapping column number to a parser function.
674: (8)                         E.g. if column 0 is a date string: ``converters = {0: datestr2num}``.
675: (8)                         Converters can also be used to provide a default value for missing
676: (8)                         data, e.g. ``converters = lambda s: float(s.strip() or 0)`` will
677: (8)                         convert empty fields to 0.
678: (8)                         Default: None
679: (4)                         ndmin : int, optional
680: (8)                         Minimum dimension of the array returned.
681: (8)                         Allowed values are 0, 1 or 2. Default is 0.
682: (4)                         unpack : bool, optional
683: (8)                         If True, the returned array is transposed, so that arguments may be
684: (8)                         unpacked using ``x, y, z = read(...)``. When used with a structured
685: (8)                         data-type, arrays are returned for each field. Default is False.
686: (4)                         dtype : numpy data type
687: (8)                         A NumPy dtype instance, can be a structured dtype to map to the
688: (8)                         columns of the file.
689: (4)                         encoding : str, optional
690: (8)                         Encoding used to decode the inputfile. The special value 'bytes'
691: (8)                         (the default) enables backwards-compatible behavior for `converters`,
692: (8)                         ensuring that inputs to the converter functions are encoded
693: (8)                         bytes objects. The special value 'bytes' has no additional effect if
694: (8)                         ``converters=None``. If encoding is ``'bytes'`` or ``None``, the
695: (8)                         default system encoding is used.

696: (4)                         Returns
697: (4)                         -----
698: (4)                         ndarray
699: (8)                         NumPy array.

700: (4)                         Examples
701: (4)                         -----
702: (4)                         First we create a file for the example.
703: (4)                         >>> s1 = '1.0,2.0,3.0\n4.0,5.0,6.0\n'
704: (4)                         >>> with open('example1.csv', 'w') as f:
705: (4)                             ...     f.write(s1)
706: (4)                         >>> a1 = read_from_filename('example1.csv')
707: (4)                         >>> a1
708: (4)                         array([[1., 2., 3.],
709: (11)                           [4., 5., 6.]])
710: (4)                         The second example has columns with different data types, so a
711: (4)                         one-dimensional array with a structured data type is returned.
712: (4)                         The tab character is used as the field delimiter.
713: (4)                         >>> s2 = '1.0\t10\talpha\n2.3\t25\tbeta\n4.5\t16\tgamma\n'
714: (4)                         >>> with open('example2.tsv', 'w') as f:
715: (4)                             ...     f.write(s2)
716: (4)                         >>> a2 = read_from_filename('example2.tsv', delimiter='\t')
717: (4)                         >>> a2
718: (4)                         array([(1. , 10, b'alpha'), (2.3, 25, b'beta'), (4.5, 16, b'gamma')]),
719: (10)                           dtype=[('f0', 'f8'), ('f1', 'u1'), ('f2', 'S5')])
720: (4)                         """
721: (4)                         byte_converters = False
722: (4)                         if encoding == 'bytes':
723: (8)                             encoding = None
724: (8)                             byte_converters = True
725: (4)                         if dtype is None:
726: (8)                             raise TypeError("a dtype must be provided.")
727: (4)                         dtype = np.dtype(dtype)
728: (4)                         read_dtype_via_object_chunks = None
729: (4)                         if dtype.kind in 'SUM' and (
730: (12)                             dtype == "S0" or dtype == "U0" or dtype == "M8" or dtype == 'm8'):
731: (8)                             read_dtype_via_object_chunks = dtype
732: (8)                             dtype = np.dtype(object)
733: (4)                         if usecols is not None:
734: (8)                             try:
735: (12)                                 usecols = list(usecols)
736: (8)                             except TypeError:
737: (12)                                 usecols = [usecols]

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

738: (4)             _ensure_ndmin_ndarray_check_param(ndmin)
739: (4)             if comment is None:
740: (8)                 comments = None
741: (4)             else:
742: (8)                 if "" in comment:
743: (12)                     raise ValueError(
744: (16)                         "comments cannot be an empty string. Use comments=None to "
745: (16)                         "disable comments."
746: (12)                     )
747: (8)             comments = tuple(comment)
748: (8)             comment = None
749: (8)             if len(comments) == 0:
750: (12)                 comments = None # No comments at all
751: (8)             elif len(comments) == 1:
752: (12)                 if isinstance(comments[0], str) and len(comments[0]) == 1:
753: (16)                     comment = comments[0]
754: (16)                     comments = None
755: (8)             else:
756: (12)                 if delimiter in comments:
757: (16)                     raise TypeError(
758: (20)                         f"Comment characters '{comments}' cannot include the "
759: (20)                         f"delimiter '{delimiter}'"
760: (16)                     )
761: (4)             if comments is not None:
762: (8)                 if quote is not None:
763: (12)                     raise ValueError(
764: (16)                         "when multiple comments or a multi-character comment is "
765: (16)                         "given, quotes are not supported. In this case quotechar "
766: (16)                         "must be set to None.")
767: (4)             if len(imaginary_unit) != 1:
768: (8)                 raise ValueError('len(imaginary_unit) must be 1.')
769: (4)             _check_nonneg_int(skiplines)
770: (4)             if max_rows is not None:
771: (8)                 _check_nonneg_int(max_rows)
772: (4)             else:
773: (8)                 max_rows = -1
774: (4)             fh_closing_ctx = contextlib.nullcontext()
775: (4)             filelike = False
776: (4)             try:
777: (8)                 if isinstance(fname, os.PathLike):
778: (12)                     fname = os.fspath(fname)
779: (8)                 if isinstance(fname, str):
780: (12)                     fh = np.lib._datasource.open(fname, 'rt', encoding=encoding)
781: (12)                     if encoding is None:
782: (16)                         encoding = getattr(fh, 'encoding', 'latin1')
783: (12)                     fh_closing_ctx = contextlib.closing(fh)
784: (12)                     data = fh
785: (12)                     filelike = True
786: (8)                 else:
787: (12)                     if encoding is None:
788: (16)                         encoding = getattr(fname, 'encoding', 'latin1')
789: (12)                     data = iter(fname)
790: (4)             except TypeError as e:
791: (8)                 raise ValueError(
792: (12)                     f"fname must be a string, filehandle, list of strings,\n"
793: (12)                     f"or generator. Got {type(fname)} instead.") from e
794: (4)             with fh_closing_ctx:
795: (8)                 if comments is not None:
796: (12)                     if filelike:
797: (16)                         data = iter(data)
798: (16)                         filelike = False
799: (12)                         data = _preprocess_comments(data, comments, encoding)
800: (8)             if read_dtype_via_object_chunks is None:
801: (12)                 arr = _load_from_filelike(
802: (16)                     data, delimiter=delimiter, comment=comment, quote=quote,
803: (16)                     imaginary_unit=imaginary_unit,
804: (16)                     usecols=usecols, skiplines=skiplines, max_rows=max_rows,
805: (16)                     converters=converters, dtype=dtype,
806: (16)                     encoding=encoding, filelike=filelike,

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

807: (16)                                byte_converters=byte_converters)
808: (8)
809: (12)
810: (16)                                if filelike:
811: (12)                                  data = iter(data) # cannot chunk when reading from file
812: (12)                                  c_byte_converters = False
813: (16)                                  if read_dtype_via_object_chunks == "S":
814: (12)                                      c_byte_converters = True # Use latin1 rather than ascii
815: (12)                                      chunks = []
816: (16)                                      while max_rows != 0:
817: (20)                                          if max_rows < 0:
818: (16)                                              chunk_size = _loadtxt_chunksize
819: (20)                                          else:
820: (16)                                              chunk_size = min(_loadtxt_chunksize, max_rows)
821: (20)                                          next_arr = _load_from_filelike(
822: (20)                                              data, delimiter=delimiter, comment=comment, quote=quote,
823: (20)                                              imaginary_unit=imaginary_unit,
824: (20)                                              usecols=usecols, skiplines=skiplines, max_rows=max_rows,
825: (20)                                              converters=converters, dtype=dtype,
826: (20)                                              encoding=encoding, filelike=filelike,
827: (20)                                              byte_converters=byte_converters,
828: (16)                                              c_byte_converters=c_byte_converters)
829: (16)                                          chunks.append(next_arr.astype(read_dtype_via_object_chunks))
830: (16)                                          skiprows = 0 # Only have to skip for first chunk
831: (16)                                          if max_rows >= 0:
832: (20)                                              max_rows -= chunk_size
833: (16)                                              if len(next_arr) < chunk_size:
834: (20)                                                  break
835: (12)                                              if len(chunks) > 1 and len(chunks[-1]) == 0:
836: (16)                                                  del chunks[-1]
837: (12)                                              if len(chunks) == 1:
838: (16)                                                  arr = chunks[0]
839: (12)                                              else:
840: (16)                                                  arr = np.concatenate(chunks, axis=0)
841: (4)                                              arr = _ensure_ndmin_ndarray(arr, ndmin=ndmin)
842: (4)                                              if arr.shape:
843: (12)                                                  if arr.shape[0] == 0:
844: (16)                                                      warnings.warn(
845: (16)                                                          f'loadtxt: input contained no data: "{fname}"',
846: (16)                                                          category=UserWarning,
847: (12)                                                          stacklevel=3
848: (4)
849: (8)                                              )
850: (8)
851: (12)                                              if unpack:
852: (8)                                                  dt = arr.dtype
853: (12)                                                  if dt.names is not None:
854: (8)                                                      return [arr[field] for field in dt.names]
855: (4)
856: (0)                                              else:
857: (0)                                                  return arr.T
858: (0)
859: (12)                                              else:
860: (12)                                                  return arr
861: (12)                                              @set_array_function_like_doc
862: (4)                                              @set_module('numpy')
863: (4)                                              def loadtxt(fname, dtype=float, comments='#', delimiter=None,
864: (4)                                              converters=None, skiprows=0, usecols=None, unpack=False,
865: (4)                                              ndmin=0, encoding='bytes', max_rows=None, *, quotechar=None,
866: (4)                                              like=None):
867: (4)                                              """
868: (4)                                              Load data from a text file.
869: (4)                                              Parameters
870: (4)                                              -----
871: (4)                                              fname : file, str, pathlib.Path, list of str, generator
872: (4)                                              File, filename, list, or generator to read. If the filename
873: (4)                                              extension is ``.gz`` or ``.bz2``, the file is first decompressed. Note
874: (4)                                              that generators must return bytes or strings. The strings
875: (4)                                              in a list or produced by a generator are treated as lines.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

876: (8)           the data-type.
877: (4)           comments : str or sequence of str or None, optional
878: (8)           The characters or list of characters used to indicate the start of a
879: (8)           comment. None implies no comments. For backwards compatibility, byte
880: (8)           strings will be decoded as 'latin1'. The default is '#'.
881: (4)           delimiter : str, optional
882: (8)           The character used to separate the values. For backwards
compatibility,
883: (8)           byte strings will be decoded as 'latin1'. The default is whitespace.
884: (8)           .. versionchanged:: 1.23.0
885: (11)          Only single character delimiters are supported. Newline characters
886: (11)          cannot be used as the delimiter.
887: (4)           converters : dict or callable, optional
888: (8)           Converter functions to customize value parsing. If `converters` is
889: (8)           callable, the function is applied to all columns, else it must be a
890: (8)           dict that maps column number to a parser function.
891: (8)           See examples for further details.
892: (8)           Default: None.
893: (8)           .. versionchanged:: 1.23.0
894: (11)          The ability to pass a single callable to be applied to all columns
895: (11)          was added.
896: (4)           skiprows : int, optional
897: (8)           Skip the first `skiprows` lines, including comments; default: 0.
898: (4)           usecols : int or sequence, optional
899: (8)           Which columns to read, with 0 being the first. For example,
900: (8)           ``usecols = (1,4,5)`` will extract the 2nd, 5th and 6th columns.
901: (8)           The default, None, results in all columns being read.
902: (8)           .. versionchanged:: 1.11.0
903: (12)          When a single column has to be read it is possible to use
904: (12)          an integer instead of a tuple. E.g ``usecols = 3`` reads the
905: (12)          fourth column the same way as ``usecols = (3,)`` would.
906: (4)           unpack : bool, optional
907: (8)           If True, the returned array is transposed, so that arguments may be
908: (8)           unpacked using ``x, y, z = loadtxt(...)``. When used with a
909: (8)           structured data-type, arrays are returned for each field.
910: (8)           Default is False.
911: (4)           ndmin : int, optional
912: (8)           The returned array will have at least `ndmin` dimensions.
913: (8)           Otherwise mono-dimensional axes will be squeezed.
914: (8)           Legal values: 0 (default), 1 or 2.
915: (8)           .. versionadded:: 1.6.0
916: (4)           encoding : str, optional
917: (8)           Encoding used to decode the inputfile. Does not apply to input
streams.
918: (8)           The special value 'bytes' enables backward compatibility workarounds
919: (8)           that ensures you receive byte arrays as results if possible and passes
920: (8)           'latin1' encoded strings to converters. Override this value to receive
921: (8)           unicode arrays and pass strings as input to converters. If set to
None
922: (8)           the system default is used. The default value is 'bytes'.
923: (8)           .. versionadded:: 1.14.0
924: (4)           max_rows : int, optional
925: (8)           Read `max_rows` rows of content after `skiprows` lines. The default is
926: (8)           to read all the rows. Note that empty rows containing no data such as
927: (8)           empty lines and comment lines are not counted towards `max_rows`,
928: (8)           while such lines are counted in `skiprows`.
929: (8)           .. versionadded:: 1.16.0
930: (8)           .. versionchanged:: 1.23.0
931: (12)          Lines containing no data, including comment lines (e.g., lines
932: (12)          starting with '#' or as specified via `comments`) are not counted
933: (12)          towards `max_rows`.
934: (4)           quotechar : unicode character or None, optional
935: (8)           The character used to denote the start and end of a quoted item.
936: (8)           Occurrences of the delimiter or comment characters are ignored within
937: (8)           a quoted item. The default value is ``quotechar=None``, which means
938: (8)           quoting support is disabled.
939: (8)           If two consecutive instances of `quotechar` are found within a quoted
940: (8)           field, the first is treated as an escape character. See examples.
941: (8)           .. versionadded:: 1.23.0

```

```

942: (4) ${ARRAY_FUNCTION_LIKE}
943: (8)     .. versionadded:: 1.20.0
944: (4) Returns
945: (4)
946: (4) out : ndarray
947: (8)     Data read from the text file.
948: (4) See Also
949: (4)
950: (4) -----
951: (4) load, fromstring, fromregex
952: (4) genfromtxt : Load data with missing values handled as specified.
953: (4) scipy.io.loadmat : reads MATLAB data files
954: (4) Notes
955: (4) -----
956: (4) This function aims to be a fast reader for simply formatted files. The
957: (4) `genfromtxt` function provides more sophisticated handling of, e.g.,
958: (4) lines with missing values.
959: (4) Each row in the input text file must have the same number of values to be
960: (4) able to read all values. If all rows do not have same number of values, a
961: (4) subset of up to n columns (where n is the least number of values present
962: (4) in all rows) can be read by specifying the columns via `usecols`.
963: (4) .. versionadded:: 1.10.0
964: (4) The strings produced by the Python float.hex method can be used as
965: (4) input for floats.
966: (4) Examples
967: (4) -----
968: (4) >>> from io import StringIO # StringIO behaves like a file object
969: (4) >>> c = StringIO("0 1\n2 3")
970: (4) >>> np.loadtxt(c)
971: (11) array([[0., 1.],
972: (4)           [2., 3.]])
973: (4) >>> d = StringIO("M 21 72\nF 35 58")
974: (4) >>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
975: (4)                           ...                               'formats': ('S1', 'i4', 'f4')})
976: (10) array([(b'M', 21, 72.), (b'F', 35, 58.)],
977: (4)         dtype=[('gender', 'S1'), ('age', '<i4'), ('weight', '<f4')])
978: (4) >>> c = StringIO("1,0,2\n3,0,4")
979: (4) >>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
980: (4) >>> x
981: (4) array([1., 3.])
982: (4) >>> y
983: (4) array([2., 4.])
984: (4) The `converters` argument is used to specify functions to preprocess the
985: (4) text prior to parsing. `converters` can be a dictionary that maps
986: (4) preprocessing functions to each column:
987: (4) >>> s = StringIO("1.618, 2.296\n3.141, 4.669\n")
988: (4) >>> conv = {
989: (4)   ...     0: lambda x: np.floor(float(x)), # conversion fn for column 0
990: (4)   ...     1: lambda x: np.ceil(float(x)), # conversion fn for column 1
991: (4)   ...
992: (4) >>> np.loadtxt(s, delimiter=",", converters=conv)
993: (11) array([[1., 3.],
994: (4)           [3., 5.]])
995: (4) `converters` can be a callable instead of a dictionary, in which case it
996: (4) is applied to all columns:
997: (4) >>> s = StringIO("0xDE 0xAD\n0xC0 0xDE")
998: (4) >>> import functools
999: (4) >>> conv = functools.partial(int, base=16)
1000: (4) >>> np.loadtxt(s, converters=conv)
1001: (11) array([[222., 173.],
1002: (4)           [192., 222.]])
1003: (4) This example shows how `converters` can be used to convert a field
1004: (4) with a trailing minus sign into a negative number.
1005: (4) >>> s = StringIO('10.01 31.25-\n19.22 64.31\n17.57- 63.94')
1006: (4) >>> def conv(fld):
1007: (4)   ...     return -float(fld[:-1]) if fld.endswith(b'-') else float(fld)
1008: (4)   ...
1009: (4) >>> np.loadtxt(s, converters=conv)
1010: (11) array([[ 10.01, -31.25],
1011: (4)           [ 19.22,  64.31],
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1011: (11) [-17.57, 63.94]])
1012: (4) Using a callable as the converter can be particularly useful for handling
1013: (4) values with different formatting, e.g. floats with underscores:
1014: (4) >>> s = StringIO("1 2.7 100_000")
1015: (4) >>> np.loadtxt(s, converters=float)
1016: (4) array([1.e+00, 2.7e+00, 1.e+05])
1017: (4) This idea can be extended to automatically handle values specified in
1018: (4) many different formats:
1019: (4) >>> def conv(val):
1020: (4) ...     try:
1021: (4) ...         return float(val)
1022: (4) ...     except ValueError:
1023: (4) ...         return float.fromhex(val)
1024: (4) >>> s = StringIO("1, 2.5, 3_000, 0b4, 0x1.4000000000000p+2")
1025: (4) >>> np.loadtxt(s, delimiter=",", converters=conv, encoding=None)
1026: (4) array([1.0e+00, 2.5e+00, 3.0e+03, 1.8e+02, 5.0e+00])
1027: (4) Note that with the default ``encoding="bytes```, the inputs to the
1028: (4) converter function are latin-1 encoded byte strings. To deactivate the
1029: (4) implicit encoding prior to conversion, use ``encoding=None``
1030: (4) >>> s = StringIO('10.01 31.25-\n19.22 64.31\n17.57- 63.94')
1031: (4) >>> conv = lambda x: -float(x[:-1]) if x.endswith('-') else float(x)
1032: (4) >>> np.loadtxt(s, converters=conv, encoding=None)
1033: (4) array([[ 10.01, -31.25],
1034: (11) [ 19.22, 64.31],
1035: (11) [-17.57, 63.94]])
1036: (4) Support for quoted fields is enabled with the `quotechar` parameter.
1037: (4) Comment and delimiter characters are ignored when they appear within a
1038: (4) quoted item delineated by `quotechar`:
1039: (4) >>> s = StringIO('"alpha, #42", 10.0\n"beta, #64", 2.0\n')
1040: (4) >>> dtype = np.dtype([('label', 'U12'), ('value', float)])
1041: (4) >>> np.loadtxt(s, dtype=dtype, delimiter=',', quotechar='\'')
1042: (4) array([('alpha, #42', 10.), ('beta, #64', 2.)],
1043: (10) dtype=[('label', '<U12'), ('value', '<f8')])
1044: (4) Quoted fields can be separated by multiple whitespace characters:
1045: (4) >>> s = StringIO('"alpha, #42"      10.0\n"beta, #64" 2.0\n')
1046: (4) >>> dtype = np.dtype([('label', 'U12'), ('value', float)])
1047: (4) >>> np.loadtxt(s, dtype=dtype, delimiter=None, quotechar='\'')
1048: (4) array([('alpha, #42', 10.), ('beta, #64', 2.)],
1049: (10) dtype=[('label', '<U12'), ('value', '<f8')])
1050: (4) Two consecutive quote characters within a quoted field are treated as a
1051: (4) single escaped character:
1052: (4) >>> s = StringIO('Hello, my name is ""Monty""!"')
1053: (4) >>> np.loadtxt(s, dtype="U", delimiter=",", quotechar='\'')
1054: (4) array('Hello, my name is "Monty"!', dtype='<U26')
1055: (4) Read subset of columns when all rows do not contain equal number of
values:
1056: (4) >>> d = StringIO("1 2\n2 4\n3 9 12\n4 16 20")
1057: (4) >>> np.loadtxt(d, usecols=(0, 1))
1058: (4) array([[ 1.,  2.],
1059: (11) [ 2.,  4.],
1060: (11) [ 3.,  9.],
1061: (11) [ 4., 16.]])
1062: (4) """
1063: (4) if like is not None:
1064: (8)     return _loadtxt_with_like(
1065: (12)         like, fname, dtype=dtype, comments=comments, delimiter=delimiter,
1066: (12)         converters=converters, skiprows=skiprows, usecols=usecols,
1067: (12)         unpack=unpack, ndmin=ndmin, encoding=encoding,
1068: (12)         max_rows=max_rows
1069: (8)     )
1070: (4) if isinstance(delimiter, bytes):
1071: (8)     delimiter.decode("latin1")
1072: (4) if dtype is None:
1073: (8)     dtype = np.float64
1074: (4) comment = comments
1075: (4) if comment is not None:
1076: (8)     if isinstance(comment, (str, bytes)):
1077: (12)         comment = [comment]
1078: (8)     comment = [

```

```

1079: (12)
comment]
1080: (4)
1081: (8)
1082: (4)
1083: (16)
1084: (16)
1085: (16)
1086: (4)
1087: (0)
1088: (0)
1089: (24)
1090: (24)
1091: (4)
1092: (0)
1093: (0)
1094: (12)
1095: (4)
1096: (4)
1097: (4)
1098: (4)
1099: (4)
1100: (8)
1101: (8)
1102: (8)
1103: (4)
1104: (8)
1105: (4)
1106: (8)
1107: (8)
1108: (8)
1109: (8)
1110: (8)
1111: (10)
1112: (8)
1113: (10)
1114: (8)
1115: (10)
1116: (10)
1117: (4)
1118: (8)
1119: (4)
1120: (8)
1121: (8)
1122: (4)
1123: (8)
1124: (8)
1125: (4)
1126: (8)
1127: (8)
1128: (4)
1129: (8)
strings,
1130: (8)
1131: (8)
1132: (8)
1133: (4)
encoding : {None, str}, optional
1134: (8)
1135: (8)
1136: (8)
Default
1137: (8)
1138: (8)
1139: (4)
1140: (4)
1141: (4)
1142: (4)
1143: (4)
1144: (4)

    x.decode('latin1') if isinstance(x, bytes) else x for x in
    if isinstance(delimiter, bytes):
        delimiter = delimiter.decode('latin1')
    arr = _read(fname, dtype=dtype, comment=comment, delimiter=delimiter,
                converters=converters, skiprows=skiprows, usecols=usecols,
                unpack=unpack, ndmin=ndmin, encoding=encoding,
                max_rows=max_rows, quote=quotechar)
    return arr
_loadtxt_with_like = array_function_dispatch()(loadtxt)
def _savetxt_dispatcher(fname, X, fmt=None, delimiter=None, newline=None,
                      header=None, footer=None, comments=None,
                      encoding=None):
    return (X,)
@array_function_dispatch(_savetxt_dispatcher)
def savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='',
            footer='', comments='# ', encoding=None):
    """
    Save an array to a text file.

    Parameters
    -----
    fname : filename or file handle
        If the filename ends in ``.gz``, the file is automatically saved in
        compressed gzip format. `loadtxt` understands gzipped files
        transparently.
    X : 1D or 2D array_like
        Data to be saved to a text file.
    fmt : str or sequence of strs, optional
        A single format (%10.5f), a sequence of formats, or a
        multi-format string, e.g. 'Iteration %d -- %10.5f', in which
        case `delimiter` is ignored. For complex `X`, the legal options
        for `fmt` are:
        * a single specifier, `fmt='%.4e'`, resulting in numbers formatted
          like `'( %s+%sj)' % (fmt, fmt)`
        * a full string specifying every real and imaginary part, e.g.
          ``%.4e %+4ej %.4e %+4ej %.4e %+4ej`` for 3 columns
        * a list of specifiers, one per column - in this case, the real
          and imaginary part must have separate specifiers,
          e.g. `'[%.3e + %.3ej', '(%.15e%+.15ej)'` for 2 columns
    delimiter : str, optional
        String or character separating columns.
    newline : str, optional
        String or character separating lines.
        .. versionadded:: 1.5.0
    header : str, optional
        String that will be written at the beginning of the file.
        .. versionadded:: 1.7.0
    footer : str, optional
        String that will be written at the end of the file.
        .. versionadded:: 1.7.0
    comments : str, optional
        String that will be prepended to the ``header`` and ``footer``
        to mark them as comments. Default: '# ', as expected by e.g.
        ``numpy.loadtxt``.
        .. versionadded:: 1.7.0
    encoding : {None, str}, optional
        Encoding used to encode the outputfile. Does not apply to output
        streams. If the encoding is something other than 'bytes' or 'latin1'
        you will not be able to load the file in NumPy versions < 1.14.
        is 'latin1'.
        .. versionadded:: 1.14.0
See Also
-----
save : Save an array to a binary file in NumPy ``.npy`` format
savez : Save several arrays into an uncompressed ``.npz`` archive
savez_compressed : Save several arrays into a compressed ``.npz`` archive
Notes

```

```

1145: (4)      -----
1146: (4)      Further explanation of the `fmt` parameter
1147: (4)      (`%[flag]width[.precision]specifier`):
1148: (4)      flags:
1149: (8)        ``-`` : left justify
1150: (8)        ``+`` : Forces to precede result with + or -.
1151: (8)        ``0`` : Left pad the number with zeros instead of space (see width).
1152: (4)      width:
1153: (8)        Minimum number of characters to be printed. The value is not truncated
1154: (8)        if it has more characters.
1155: (4)      precision:
1156: (8)        - For integer specifiers (eg. ``d,i,o,x``), the minimum number of
1157: (10)       digits.
1158: (8)        - For ``e, E`` and ``f`` specifiers, the number of digits to print
1159: (10)       after the decimal point.
1160: (8)        - For ``g`` and ``G``, the maximum number of significant digits.
1161: (8)        - For ``s``, the maximum number of characters.
1162: (4)      specifiers:
1163: (8)        ``c`` : character
1164: (8)        ``d`` or ``i`` : signed decimal integer
1165: (8)        ``e`` or ``E`` : scientific notation with ``e`` or ``E``.
1166: (8)        ``f`` : decimal floating point
1167: (8)        ``g,G`` : use the shorter of ``e,E`` or ``f``
1168: (8)        ``o`` : signed octal
1169: (8)        ``s`` : string of characters
1170: (8)        ``u`` : unsigned decimal integer
1171: (8)        ``x,X`` : unsigned hexadecimal integer
1172: (4)      This explanation of ``fmt`` is not complete, for an exhaustive
1173: (4)      specification see [1]_.
1174: (4)      References
1175: (4)      -----
1176: (4)      .. [1] `Format Specification Mini-Language
1177: (11)       <https://docs.python.org/library/string.html#format-specification-`_
1178: (11)       Python Documentation.
1179: (4)      Examples
1180: (4)      -----
1181: (4)      >>> x = y = z = np.arange(0.0,5.0,1.0)
1182: (4)      >>> np.savetxt('test.out', x, delimiter=',')    # X is an array
1183: (4)      >>> np.savetxt('test.out', (x,y,z))    # x,y,z equal sized 1D arrays
1184: (4)      >>> np.savetxt('test.out', x, fmt='%1.4e')    # use exponential notation
1185: (4)      """
1186: (4)      if isinstance(fmt, bytes):
1187: (8)          fmt = asstr(fmt)
1188: (4)      delimiter = asstr(delimiter)
1189: (4)      class WriteWrap:
1190: (8)          """Convert to bytes on bytestream inputs.
1191: (8)          """
1192: (8)          def __init__(self, fh, encoding):
1193: (12)              self.fh = fh
1194: (12)              self.encoding = encoding
1195: (12)              self.do_write = self.first_write
1196: (8)          def close(self):
1197: (12)              self.fh.close()
1198: (8)          def write(self, v):
1199: (12)              self.do_write(v)
1200: (8)          def write_bytes(self, v):
1201: (12)              if isinstance(v, bytes):
1202: (16)                  self.fh.write(v)
1203: (12)              else:
1204: (16)                  self.fh.write(v.encode(self.encoding))
1205: (8)          def write_normal(self, v):
1206: (12)              self.fh.write(asunicode(v))
1207: (8)          def first_write(self, v):
1208: (12)              try:
1209: (16)                  self.write_normal(v)
1210: (16)                  self.write = self.write_normal
1211: (12)              except TypeError:
1212: (16)                  self.write_bytes(v)

```

```

1213: (16)                     self.write = self.write_bytes
1214: (4)          own_fh = False
1215: (4)          if isinstance(fname, os_PathLike):
1216: (8)              fname = os_fspath(fname)
1217: (4)          if _is_string_like(fname):
1218: (8)              open(fname, 'wt').close()
1219: (8)              fh = np.lib._datasource.open(fname, 'wt', encoding=encoding)
1220: (8)              own_fh = True
1221: (4)          elif hasattr(fname, 'write'):
1222: (8)              fh = WriteWrap(fname, encoding or 'latin1')
1223: (4)          else:
1224: (8)              raise ValueError('fname must be a string or file handle')
1225: (4)          try:
1226: (8)              X = np.asarray(X)
1227: (8)              if X.ndim == 0 or X.ndim > 2:
1228: (12)                  raise ValueError(
1229: (16)                      "Expected 1D or 2D array, got %dD array instead" % X.ndim)
1230: (8)              elif X.ndim == 1:
1231: (12)                  if X.dtype.names is None:
1232: (16)                      X = np.atleast_2d(X).T
1233: (16)                      ncol = 1
1234: (12)                  else:
1235: (16)                      ncol = len(X.dtype.names)
1236: (8)              else:
1237: (12)                  ncol = X.shape[1]
1238: (8)              iscomplex_X = np.iscomplexobj(X)
1239: (8)              if type(fmt) in (list, tuple):
1240: (12)                  if len(fmt) != ncol:
1241: (16)                      raise AttributeError('fmt has wrong shape. %s' % str(fmt))
1242: (12)                  format = asstr(delimiter).join(map(asstr, fmt))
1243: (8)              elif isinstance(fmt, str):
1244: (12)                  n_fmt_chars = fmt.count('%')
1245: (12)                  error = ValueError('fmt has wrong number of %% formats: %s' %
fmt)
1246: (12)                  if n_fmt_chars == 1:
1247: (16)                      if iscomplex_X:
1248: (20)                          fmt = ['(%s+%sj)' % (fmt, fmt), ] * ncol
1249: (16)                      else:
1250: (20)                          fmt = [fmt, ] * ncol
1251: (16)                      format = delimiter.join(fmt)
1252: (12)                  elif iscomplex_X and n_fmt_chars != (2 * ncol):
1253: (16)                      raise error
1254: (12)                  elif ((not iscomplex_X) and n_fmt_chars != ncol):
1255: (16)                      raise error
1256: (12)                  else:
1257: (16)                      format = fmt
1258: (8)              else:
1259: (12)                  raise ValueError('invalid fmt: %r' % (fmt,))
1260: (8)          if len(header) > 0:
1261: (12)              header = header.replace('\n', '\n' + comments)
1262: (12)              fh.write(comments + header + newline)
1263: (8)          if iscomplex_X:
1264: (12)              for row in X:
1265: (16)                  row2 = []
1266: (16)                  for number in row:
1267: (20)                      row2.append(number.real)
1268: (20)                      row2.append(number.imag)
1269: (16)                      s = format % tuple(row2) + newline
1270: (16)                      fh.write(s.replace('+-', '-'))
1271: (8)          else:
1272: (12)              for row in X:
1273: (16)                  try:
1274: (20)                      v = format % tuple(row) + newline
1275: (16)                  except TypeError as e:
1276: (20)                      raise TypeError("Mismatch between array dtype ('%s') and "
1277: (36)                          "format specifier ('%s')"
1278: (36)                          % (str(X.dtype), format)) from e
1279: (16)                      fh.write(v)
1280: (8)          if len(footer) > 0:

```

```

1281: (12)
1282: (12)
1283: (4)
1284: (8)
1285: (12)
1286: (0)
1287: (0)
1288: (4)
1289: (4)
1290: (4)
1291: (4)
1292: (4)
1293: (4)
1294: (4)
1295: (4)
1296: (8)
1297: (8)
1298: (12)
1299: (4)
1300: (8)
1301: (8)
1302: (4)
1303: (8)
1304: (4)
1305: (8)
streams.
1306: (8)
1307: (4)
1308: (4)
1309: (4)
1310: (8)
1311: (8)
1312: (4)
1313: (4)
1314: (4)
1315: (8)
1316: (4)
1317: (4)
1318: (4)
1319: (4)
1320: (4)
1321: (4)
1322: (4)
1323: (4)
1324: (4)
1325: (4)
1326: (4)
1327: (4)
1328: (4)
1329: (4)
1330: (4)
1331: (4)
1332: (4)
1333: (10)
1334: (4)
1335: (4)
1336: (4)
1337: (4)
1338: (4)
1339: (8)
1340: (8)
1341: (8)
1342: (4)
1343: (8)
1344: (12)
1345: (8)
1346: (12)
1347: (8)
1348: (8)

        footer = footer.replace('\n', '\n' + comments)
        fh.write(comments + footer + newline)
    finally:
        if own_fh:
            fh.close()
@set_module('numpy')
def fromregex(file, regexp, dtype, encoding=None):
    """
    Construct an array from a text file, using regular expression parsing.
    The returned array is always a structured array, and is constructed from
    all matches of the regular expression in the file. Groups in the regular
    expression are converted to fields of the structured array.

    Parameters
    -----
    file : path or file
        Filename or file object to read.
        .. versionchanged:: 1.22.0
            Now accepts `os.PathLike` implementations.
    regexp : str or regexp
        Regular expression used to parse the file.
        Groups in the regular expression correspond to fields in the dtype.
    dtype : dtype or list of dtypes
        Dtype for the structured array; must be a structured datatype.
    encoding : str, optional
        Encoding used to decode the inputfile. Does not apply to input
        .. versionadded:: 1.14.0
    Returns
    -----
    output : ndarray
        The output array, containing the part of the content of `file` that
        was matched by `regexp`. `output` is always a structured array.
    Raises
    -----
    TypeError
        When `dtype` is not a valid dtype for a structured array.
    See Also
    -----
    fromstring, loadtxt
    Notes
    -----
    Dtypes for structured arrays can be specified in several forms, but all
    forms specify at least the data type and field name. For details see
    `basics.rec`.
    Examples
    -----
    >>> from io import StringIO
    >>> text = StringIO("1312 foo\n1534 bar\n444 qux")
    >>> regexp = r"(\d+)\s+(...)" # match [digits, whitespace, anything]
    >>> output = np.fromregex(text, regexp,
    ...                         [('num', np.int64), ('key', 'S3')])
    >>> output
    array([(1312, b'foo'), (1534, b'bar'), (444, b'qux')],  

          dtype=[('num', '<i8'), ('key', 'S3')])
    >>> output['num']
    array([1312, 1534, 444])
    """
    own_fh = False
    if not hasattr(file, "read"):
        file = os.fspath(file)
        file = np.lib._datasource.open(file, 'rt', encoding=encoding)
        own_fh = True
    try:
        if not isinstance(dtype, np.dtype):
            dtype = np.dtype(dtype)
        if dtype.names is None:
            raise TypeError('dtype must be a structured datatype.')
        content = file.read()
        if isinstance(content, bytes) and isinstance(regexp, str):

```

```

1349: (12)             regexp = asbytes(regexp)
1350: (8)              elif isinstance(content, str) and isinstance(regexp, bytes):
1351: (12)                regexp = asstr(regexp)
1352: (8)              if not hasattr(regexp, 'match'):
1353: (12)                regexp = re.compile(regexp)
1354: (8)              seq = regexp.findall(content)
1355: (8)              if seq and not isinstance(seq[0], tuple):
1356: (12)                newdtype = np.dtype(dtype[dtype.names[0]])
1357: (12)                output = np.array(seq, dtype=newdtype)
1358: (12)                output.dtype = dtype
1359: (8)              else:
1360: (12)                output = np.array(seq, dtype=dtype)
1361: (8)              return output
1362: (4)              finally:
1363: (8)                if own_fh:
1364: (12)                  file.close()
@set_array_function_like_doc
@set_module('numpy')
def genfromtxt(fname, dtype=float, comments='#', delimiter=None,
               skip_header=0, skip_footer=0, converters=None,
               missing_values=None, filling_values=None, usecols=None,
               names=None, excludelist=None,
               deletechars=''.join(sorted(NameValidator.defaultdeletechars)),
               replace_space=' ', autostrip=False, case_sensitive=True,
               defaultfmt="f%i", unpack=None, usemask=False, loose=True,
               invalid_raise=True, max_rows=None, encoding='bytes',
               *, ndmin=0, like=None):
    """
    Load data from a text file, with missing values handled as specified.
    Each line past the first `skip_header` lines is split at the `delimiter`
    character, and characters following the `comments` character are
    discarded.

    Parameters
    -----
    fname : file, str, pathlib.Path, list of str, generator
        File, filename, list, or generator to read. If the filename
        extension is ``.gz`` or ``.bz2``, the file is first decompressed. Note
        that generators must return bytes or strings. The strings
        in a list or produced by a generator are treated as lines.
    dtype : dtype, optional
        Data type of the resulting array.
        If None, the dtypes will be determined by the contents of each
        column, individually.
    comments : str, optional
        The character used to indicate the start of a comment.
        All the characters occurring on a line after a comment are discarded.
    delimiter : str, int, or sequence, optional
        The string used to separate values. By default, any consecutive
        whitespaces act as delimiter. An integer or sequence of integers
        can also be provided as width(s) of each field.
    skiprows : int, optional
        `skiprows` was removed in numpy 1.10. Please use `skip_header` instead.
    skip_header : int, optional
        The number of lines to skip at the beginning of the file.
    skip_footer : int, optional
        The number of lines to skip at the end of the file.
    converters : variable, optional
        The set of functions that convert the data of a column to a value.
        The converters can also be used to provide a default value
        for missing data: ``converters = {3: lambda s: float(s or 0)}``.
    missing : variable, optional
        `missing` was removed in numpy 1.10. Please use `missing_values` instead.
    missing_values : variable, optional
        The set of strings corresponding to missing data.
    filling_values : variable, optional
        The set of values to be used as default when the data are missing.
    usecols : sequence, optional

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1416: (8)           Which columns to read, with 0 being the first. For example,
1417: (8)           ``usecols = (1, 4, 5)`` will extract the 2nd, 5th and 6th columns.
1418: (4)           names : {None, True, str, sequence}, optional
1419: (8)           If `names` is True, the field names are read from the first line after
1420: (8)           the first `skip_header` lines. This line can optionally be preceded
1421: (8)           by a comment delimiter. If `names` is a sequence or a single-string of
1422: (8)           comma-separated names, the names will be used to define the field
names
1423: (8)           in a structured dtype. If `names` is None, the names of the dtype
1424: (8)           fields will be used, if any.
1425: (4)           excludelist : sequence, optional
1426: (8)           A list of names to exclude. This list is appended to the default list
1427: (8)           ['return', 'file', 'print']. Excluded names are appended with an
1428: (8)           underscore: for example, `file` would become `file_`.
deletechars : str, optional
1429: (4)           A string combining invalid characters that must be deleted from the
1430: (8)           names.
defaultfmt : str, optional
1431: (8)           A format used to define default field names, such as "f%i" or
"f %02i".
1432: (4)
1433: (8)           A format used to define default field names, such as "f%i" or
"f %02i".
1434: (4)           autostrip : bool, optional
1435: (8)           Whether to automatically strip white spaces from the variables.
1436: (4)           replace_space : char, optional
1437: (8)           Character(s) used in replacement of white spaces in the variable
1438: (8)           names. By default, use a '_'.
case_sensitive : {True, False, 'upper', 'lower'}, optional
1439: (4)           If True, field names are case sensitive.
1440: (8)           If False or 'upper', field names are converted to upper case.
1441: (8)           If 'lower', field names are converted to lower case.
1442: (8)
1443: (4)           unpack : bool, optional
1444: (8)           If True, the returned array is transposed, so that arguments may be
1445: (8)           unpacked using ``x, y, z = genfromtxt(...)``. When used with a
1446: (8)           structured data-type, arrays are returned for each field.
1447: (8)           Default is False.
1448: (4)           usemask : bool, optional
1449: (8)           If True, return a masked array.
1450: (8)           If False, return a regular array.
1451: (4)           loose : bool, optional
1452: (8)           If True, do not raise errors for invalid values.
invalid_raise : bool, optional
1453: (4)           If True, an exception is raised if an inconsistency is detected in the
1454: (8)           number of columns.
1455: (8)           If False, a warning is emitted and the offending lines are skipped.
1456: (8)
1457: (4)           max_rows : int, optional
1458: (8)           The maximum number of rows to read. Must not be used with skip_footer
1459: (8)           at the same time. If given, the value must be at least 1. Default is
1460: (8)           to read the entire file.
1461: (8)           .. versionadded:: 1.10.0
encoding : str, optional
1462: (4)           Encoding used to decode the inputfile. Does not apply when `fname` is
1463: (8)           a file object. The special value 'bytes' enables backward
compatibility
1464: (8)
1465: (8)           workarounds that ensure that you receive byte arrays when possible
1466: (8)           and passes latin1 encoded strings to converters. Override this value
to
1467: (8)           receive unicode arrays and pass strings as input to converters. If
set
1468: (8)           to None the system default is used. The default value is 'bytes'.
1469: (8)           .. versionadded:: 1.14.0
1470: (4)           ndmin : int, optional
1471: (8)           Same parameter as `loadtxt`
1472: (8)           .. versionadded:: 1.23.0
${ARRAY_FUNCTION_LIKE}
1473: (4)
1474: (8)           .. versionadded:: 1.20.0
1475: (4)           Returns
1476: (4)           -----
1477: (4)           out : ndarray
1478: (8)           Data read from the text file. If `usemask` is True, this is a
1479: (8)           masked array.

```

```

1480: (4)             See Also
1481: (4)             -----
1482: (4)             numpy.loadtxt : equivalent function when no data is missing.
1483: (4)             Notes
1484: (4)             -----
1485: (4)             * When spaces are used as delimiters, or when no delimiter has been given
1486: (6)             as input, there should not be any missing data between two fields.
1487: (4)             * When the variables are named (either by a flexible dtype or with
`names`),
1488: (6)             there must not be any header in the file (else a ValueError
1489: (6)             exception is raised).
1490: (4)             * Individual values are not stripped of spaces by default.
1491: (6)             When using a custom converter, make sure the function does remove
spaces.
1492: (4)             References
1493: (4)             -----
1494: (4)             .. [1] NumPy User Guide, section `I/O with NumPy
1495: (11)
<https://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html>`_.
1496: (4)             Examples
1497: (4)             -----
1498: (4)             >>> from io import StringIO
1499: (4)             >>> import numpy as np
1500: (4)             Comma delimited file with mixed dtype
1501: (4)             >>> s = StringIO(u"1,1.3,abcde")
1502: (4)             >>> data = np.genfromtxt(s, dtype=[('myint','i8'),('myfloat','f8'),
1503: (4)             ... ('mystring','S5')], delimiter=",")
1504: (4)             >>> data
1505: (4)             array((1, 1.3, b'abcde'),
1506: (10)             ...           dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])
1507: (4)             Using dtype = None
1508: (4)             >>> _ = s.seek(0) # needed for StringIO example only
1509: (4)             >>> data = np.genfromtxt(s, dtype=None,
1510: (4)             ... names = ['myint','myfloat','mystring'], delimiter=",")
1511: (4)             >>> data
1512: (4)             array((1, 1.3, b'abcde'),
1513: (10)             ...           dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])
1514: (4)             Specifying dtype and names
1515: (4)             >>> _ = s.seek(0)
1516: (4)             >>> data = np.genfromtxt(s, dtype="i8,f8,S5",
1517: (4)             ... names=['myint','myfloat','mystring'], delimiter=",")
1518: (4)             >>> data
1519: (4)             array((1, 1.3, b'abcde'),
1520: (10)             ...           dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])
1521: (4)             An example with fixed-width columns
1522: (4)             >>> s = StringIO(u"11.3abcde")
1523: (4)             >>> data = np.genfromtxt(s, dtype=None, names=
['intvar','fltvar','strvar'],
1524: (4)             ...           delimiter=[1,3,5])
1525: (4)             >>> data
1526: (4)             array((1, 1.3, b'abcde'),
1527: (10)             ...           dtype=[('intvar', '<i8'), ('fltvar', '<f8'), ('strvar', 'S5')])
1528: (4)             An example to show comments
1529: (4)             >>> f = StringIO('''
1530: (4)             ... text,# of chars
1531: (4)             ... hello world,11
1532: (4)             ... numpy,5''')
1533: (4)             >>> np.genfromtxt(f, dtype='S12,S12', delimiter=',')
1534: (4)             array([(b'text', b''), (b'hello world', b'11'), (b'numpy', b'5')]),
1535: (6)             ...           dtype=[('f0', 'S12'), ('f1', 'S12')])
1536: (4)             """
1537: (4)             if like is not None:
1538: (8)                 return _genfromtxt_with_like(
1539: (12)                     like, fname, dtype=dtype, comments=comments, delimiter=delimiter,
1540: (12)                     skip_header=skip_header, skip_footer=skip_footer,
1541: (12)                     converters=converters, missing_values=missing_values,
1542: (12)                     filling_values=filling_values, usecols=usecols, names=names,
1543: (12)                     excludelist=excludelist, deletechars=deletechars,
1544: (12)                     replace_space=replace_space, autostrip=autostrip,

```

```

1545: (12)             case_sensitive=case_sensitive, defaultfmt=defaultfmt,
1546: (12)             unpack=unpack, usemask=usemask, loose=loose,
1547: (12)             invalid_raise=invalid_raise, max_rows=max_rows, encoding=encoding,
1548: (12)             ndmin=ndmin,
1549: (8)
1550: (4)             )
1551: (4)             _ensure_ndmin_ndarray_check_param(ndmin)
1552: (8)             if max_rows is not None:
1553: (12)                 if skip_footer:
1554: (20)                     raise ValueError(
1555: (20)                         "The keywords 'skip_footer' and 'max_rows' can not be "
1556: (8)                         "specified at the same time.")
1557: (12)             if max_rows < 1:
1558: (4)                 raise ValueError("'max_rows' must be at least 1.")
1559: (8)             if usemask:
1560: (4)                 from numpy.ma import MaskedArray, make_mask_descr
1561: (4)                 user_converters = converters or {}
1562: (8)                 if not isinstance(user_converters, dict):
1563: (12)                     raise TypeError(
1564: (12)                         "The input argument 'converter' should be a valid dictionary "
1565: (4)                         "(got '%s' instead)" % type(user_converters))
1566: (8)             if encoding == 'bytes':
1567: (8)                 encoding = None
1568: (4)                 byte_converters = True
1569: (8)             else:
1570: (4)                 byte_converters = False
1571: (8)             if isinstance(fname, os_PathLike):
1572: (4)                 fname = os_fspath(fname)
1573: (8)             if isinstance(fname, str):
1574: (8)                 fid = np.lib._datasource.open(fname, 'rt', encoding=encoding)
1575: (4)                 fid_ctx = contextlib.closing(fid)
1576: (8)             else:
1577: (8)                 fid = fname
1578: (4)                 fid_ctx = contextlib.nullcontext(fid)
1579: (8)             try:
1580: (4)                 fhd = iter(fid)
1581: (8)             except TypeError as e:
1582: (12)                 raise TypeError(
1583: (12)                     "fname must be a string, a filehandle, a sequence of strings,\n"
1584: (8)                     "or an iterator of strings. Got {type(fname)} instead."
1585: (4)                     ) from e
1586: (8)             with fid_ctx:
1587: (34)                 split_line = LineSplitter(delimiter=delimiter, comments=comments,
1588: (8)                               autostrip=autostrip, encoding=encoding)
1589: (39)                 validate_names = NameValidator(excludelist=excludelist,
1590: (39)                               deletechars=deletechars,
1591: (39)                               case_sensitive=case_sensitive,
1592: (8)                               replace_space=replace_space)
1593: (12)             try:
1594: (16)                 for i in range(skip_header):
1595: (12)                     next(fhd)
1596: (12)                     first_values = None
1597: (16)                     while not first_values:
1598: (16)                         first_line = _decode_line(next(fhd), encoding)
1599: (20)                         if (names is True) and (comments is not None):
1600: (24)                             if comments in first_line:
1601: (28)                                 first_line = (
1602: (16)                                     ''.join(first_line.split(comments)[1:]))
1603: (8)                                 first_values = split_line(first_line)
1604: (12)             except StopIteration:
1605: (12)                 first_line = ''
1606: (12)                 first_values = []
1607: (8)                 warnings.warn('genfromtxt: Empty input file: "%s"' % fname,
1608: (12) stacklevel=2)
1609: (12)             if names is True:
1610: (16)                 fval = first_values[0].strip()
1611: (20)                 if comments is not None:
1612: (8)                     if fval in comments:

```

```

1613: (12)
1614: (16)
1615: (12)
1616: (16)
1617: (20)
1618: (16)
1619: (20)
1620: (8)
1621: (8)
1622: (12)
1623: (12)
1624: (8)
1625: (12)
1626: (8)
1627: (12)
1628: (8)
1629: (12)
1630: (31)
1631: (31)
1632: (31)
1633: (31)
1634: (8)
1635: (12)
1636: (8)
1637: (12)
1638: (16)
1639: (20)
1640: (16)
1641: (20)
1642: (12)
1643: (16)
1644: (16)
1645: (16)
1646: (12)
1647: (16)
1648: (8)
1649: (12)
1650: (8)
1651: (8)
1652: (12)
1653: (8)
1654: (8)
1655: (12)
1656: (16)
1657: (20)
1658: (24)
1659: (20)
1660: (24)
1661: (16)
1662: (20)
1663: (24)
1664: (20)
1665: (24)
1666: (16)
1667: (20)
1668: (16)
1669: (20)
1670: (16)
1671: (20)
1672: (24)
1673: (16)
1674: (20)
1675: (8)
1676: (12)
1677: (16)
1678: (16)
1679: (20)
1680: (8)
1681: (12)

        try:
            usecols = [_.strip() for _ in usecols.split(",")]
        except AttributeError:
            try:
                usecols = list(usecols)
            except TypeError:
                usecols = [usecols, ]
        nbcols = len(usecols or first_values)
        if names is True:
            names = validate_names([str(_.strip()) for _ in first_values])
            first_line = ''
        elif _is_string_like(names):
            names = validate_names([_.strip() for _ in names.split(',')])
        elif names:
            names = validate_names(names)
        if dtype is not None:
            dtype = easy_dtype(dtype, defaultfmt=defaultfmt, names=names,
                               excludelist=excludelist,
                               deletechars=deletechars,
                               case_sensitive=case_sensitive,
                               replace_space=replace_space)
        if names is not None:
            names = list(names)
        if usecols:
            for (i, current) in enumerate(usecols):
                if _is_string_like(current):
                    usecols[i] = names.index(current)
                elif current < 0:
                    usecols[i] = current + len(first_values)
        if (dtype is not None) and (len(dtype) > nbcols):
            descr = dtype.descr
            dtype = np.dtype([descr[_] for _ in usecols])
            names = list(dtype.names)
        elif (names is not None) and (len(names) > nbcols):
            names = [names[_] for _ in usecols]
        elif (names is not None) and (dtype is not None):
            names = list(dtype.names)
        user_missing_values = missing_values or {}
        if isinstance(user_missing_values, bytes):
            user_missing_values = user_missing_values.decode('latin1')
        missing_values = [list(['']) for _ in range(nbcols)]
        if isinstance(user_missing_values, dict):
            for (key, val) in user_missing_values.items():
                if _is_string_like(key):
                    try:
                        key = names.index(key)
                    except ValueError:
                        continue
                if usecols:
                    try:
                        key = usecols.index(key)
                    except ValueError:
                        pass
                    if isinstance(val, (list, tuple)):
                        val = [str(_) for _ in val]
                    else:
                        val = [str(val), ]
                    if key is None:
                        for miss in missing_values:
                            miss.extend(val)
                    else:
                        missing_values[key].extend(val)
            elif isinstance(user_missing_values, (list, tuple)):
                for (value, entry) in zip(user_missing_values, missing_values):
                    value = str(value)
                    if value not in entry:
                        entry.append(value)
            elif isinstance(user_missing_values, str):
                user_value = user_missing_values.split(",")

```

```

1682: (12)
1683: (16)
1684: (8)
1685: (12)
1686: (16)
1687: (8)
1688: (8)
1689: (12)
1690: (8)
1691: (8)
1692: (12)
1693: (16)
1694: (20)
1695: (24)
1696: (20)
1697: (24)
1698: (16)
1699: (20)
1700: (24)
1701: (20)
1702: (24)
1703: (16)
1704: (8)
1705: (12)
1706: (12)
1707: (16)
1708: (12)
1709: (16)
1710: (8)
1711: (12)
1712: (8)
1713: (12)
default=fill)
1714: (26)
filling_values)]
1715: (8)
1716: (12)
1717: (12)
1718: (16)
1719: (16)
1720: (46)
default=fill)
1721: (30)
1722: (12)
1723: (16)
1724: (16)
1725: (46)
default=fill)
1726: (30)
1727: (8)
1728: (8)
1729: (12)
1730: (16)
1731: (20)
1732: (20)
1733: (16)
1734: (20)
1735: (12)
1736: (16)
1737: (20)
1738: (16)
1739: (20)
1740: (12)
1741: (16)
1742: (12)
1743: (16)
1744: (12)
1745: (16)
1746: (12)

        for entry in missing_values:
            entry.extend(user_value)
        else:
            for entry in missing_values:
                entry.extend([str(user_missing_values)])
    user_filling_values = filling_values
    if user_filling_values is None:
        user_filling_values = []
    filling_values = [None] * nbcols
    if isinstance(user_filling_values, dict):
        for (key, val) in user_filling_values.items():
            if _is_string_like(key):
                try:
                    key = names.index(key)
                except ValueError:
                    continue
            if usecols:
                try:
                    key = usecols.index(key)
                except ValueError:
                    pass
                filling_values[key] = val
            elif isinstance(user_filling_values, (list, tuple)):
                n = len(user_filling_values)
                if (n <= nbcols):
                    filling_values[:n] = user_filling_values
                else:
                    filling_values = user_filling_values[:nbcols]
            else:
                filling_values = [user_filling_values] * nbcols
    if dtype is None:
        converters = [StringConverter(None, missing_values=miss,
                                      for (miss, fill) in zip(missing_values,
                                                               filling_values))]
    else:
        dtype_flat = flatten_dtype(dtype, flatten_base=True)
        if len(dtype_flat) > 1:
            zipit = zip(dtype_flat, missing_values, filling_values)
            converters = [StringConverter(dt, locked=True,
                                          missing_values=miss,
                                          for (dt, miss, fill) in zipit)]
        else:
            zipit = zip(missing_values, filling_values)
            converters = [StringConverter(dtype, locked=True,
                                          missing_values=miss,
                                          for (miss, fill) in zipit)]
    uc_update = []
    for (j, conv) in user_converters.items():
        if _is_string_like(j):
            try:
                j = names.index(j)
                i = j
            except ValueError:
                continue
        elif usecols:
            try:
                i = usecols.index(j)
            except ValueError:
                continue
        else:
            i = j
        if len(first_line):
            testing_value = first_values[j]
        else:
            testing_value = None
        if conv is bytes:

```

```

1747: (16)             user_conv = asbytes
1748: (12)             elif byte_converters:
1749: (16)                 def tobytes_first(x, conv):
1750: (20)                     if type(x) is bytes:
1751: (24)                         return conv(x)
1752: (20)                     return conv(x.encode("latin1"))
1753: (16)                 user_conv = functools.partial(tobytes_first, conv=conv)
1754: (12)             else:
1755: (16)                 user_conv = conv
1756: (12)             converters[i].update(user_conv, locked=True,
1757: (33)                             testing_value=testing_value,
1758: (33)                             default=filling_values[i],
1759: (33)                             missing_values=missing_values[i],)
1760: (12)             uc_update.append((i, user_conv))
1761: (8)             user_converters.update(uc_update)
1762: (8)             rows = []
1763: (8)             append_to_rows = rows.append
1764: (8)             if usemask:
1765: (12)                 masks = []
1766: (12)                 append_to_masks = masks.append
1767: (8)             invalid = []
1768: (8)             append_to_invalid = invalid.append
1769: (8)             for (i, line) in enumerate(itertools.chain([first_line, ], fhd)):
1770: (12)                 values = split_line(line)
1771: (12)                 nbvalues = len(values)
1772: (12)                 if nbvalues == 0:
1773: (16)                     continue
1774: (12)                 if usecols:
1775: (16)                     try:
1776: (20)                         values = [values[_] for _ in usecols]
1777: (16)                     except IndexError:
1778: (20)                         append_to_invalid((i + skip_header + 1, nbvalues))
1779: (20)                         continue
1780: (12)                     elif nbvalues != nbcols:
1781: (16)                         append_to_invalid((i + skip_header + 1, nbvalues))
1782: (16)                         continue
1783: (12)                     append_to_rows(tuple(values))
1784: (12)                     if usemask:
1785: (16)                         append_to_masks(tuple([v.strip() in m
1786: (39)                             for (v, m) in zip(values,
1787: (57)                               missing_values)]))
1788: (12)                     if len(rows) == max_rows:
1789: (16)                         break
1790: (4)             if dtype is None:
1791: (8)                 for (i, converter) in enumerate(converters):
1792: (12)                     current_column = [itemgetter(i)(_m) for _m in rows]
1793: (12)                     try:
1794: (16)                         converter.iterupgrade(current_column)
1795: (12)                     except ConverterLockError:
1796: (16)                         errmsg = "Converter #{} is locked and cannot be upgraded: {}"
1797: (16)                         current_column = map(itemgetter(i), rows)
1798: (16)                         for (j, value) in enumerate(current_column):
1799: (20)                             try:
1800: (24)                                 converter.upgrade(value)
1801: (20)                             except (ConverterError, ValueError):
1802: (24)                                 errmsg += "(occurred line #{} for value '{}')"
1803: (24)                                 errmsg %= (j + 1 + skip_header, value)
1804: (24)                                 raise ConverterError(errmsg)
1805: (4)             nbinvalid = len(invalid)
1806: (4)             if nbinvalid > 0:
1807: (8)                 nbrows = len(rows) + nbinvalid - skip_footer
1808: (8)                 template = "Line #{} (got {} columns instead of {})" % nbcols
1809: (8)                 if skip_footer > 0:
1810: (12)                     nbinvalid_skipped = len([_ for _ in invalid
1811: (37)                         if [_[0]] > nbrows + skip_header])
1812: (12)                     invalid = invalid[:nbinvalid - nbinvalid_skipped]
1813: (12)                     skip_footer -= nbinvalid_skipped
1814: (8)                     errmsg = [template % (i, nb)

```

```

1815: (18)
1816: (8)
1817: (12)
1818: (12)
1819: (12)
1820: (16)
1821: (12)
1822: (16)
1823: (4)
1824: (8)
1825: (8)
1826: (12)
1827: (4)
1828: (8)
1829: (12)
1830: (18)
1831: (4)
1832: (8)
1833: (12)
1834: (18)
1835: (4)
1836: (4)
1837: (8)
1838: (8)
1839: (21)
1840: (8)
1841: (12)
1842: (16)
1843: (16)
1844: (16)
1845: (16)
1846: (12)
1847: (16)
1848: (16)
1849: (20)
1850: (16)
1851: (12)
1852: (16)
1853: (12)
1854: (16)
1855: (12)
1856: (16)
1857: (20)
1858: (8)
1859: (8)
1860: (12)
1861: (16)
1862: (16)
1863: (8)
1864: (12)
1865: (16)
1866: (16)
1867: (16)
1868: (12)
1869: (16)
1870: (16)
1871: (12)
1872: (16)
1873: (26)
1874: (16)
1875: (20)
1876: (30)
1877: (8)
1878: (12)
1879: (12)
1880: (8)
1881: (8)
1882: (12)
1883: (4)

        for (i, nb) in invalid]
    if len(errmsg):
        errmsg.insert(0, "Some errors were detected !")
        errmsg = "\n".join(errmsg)
        if invalid_raise:
            raise ValueError(errmsg)
        else:
            warnings.warn(errmsg, ConversionWarning, stacklevel=2)
    if skip_footer > 0:
        rows = rows[:-skip_footer]
        if usemask:
            masks = masks[:-skip_footer]
    if loose:
        rows = list(
            zip(*[[conv._loose_call(_r) for _r in map(itemgetter(i), rows)]
                  for (i, conv) in enumerate(converters)]))
    else:
        rows = list(
            zip(*[[conv._strict_call(_r) for _r in map(itemgetter(i), rows)]
                  for (i, conv) in enumerate(converters)]))
    data = rows
    if dtype is None:
        column_types = [conv.type for conv in converters]
        strcolidx = [i for (i, v) in enumerate(column_types)
                     if v == np.str_]
        if byte_converters and strcolidx:
            warnings.warn(
                "Reading unicode strings without specifying the encoding "
                "argument is deprecated. Set the encoding, use None for the "
                "system default.",
                np.VisibleDeprecationWarning, stacklevel=2)
    def encode_unicode_cols(row_tup):
        row = list(row_tup)
        for i in strcolidx:
            row[i] = row[i].encode('latin1')
        return tuple(row)
    try:
        data = [encode_unicode_cols(r) for r in data]
    except UnicodeEncodeError:
        pass
    else:
        for i in strcolidx:
            column_types[i] = np.bytes_
    sized_column_types = column_types[:]
    for i, col_type in enumerate(column_types):
        if np.issubdtype(col_type, np.character):
            n_chars = max(len(row[i]) for row in data)
            sized_column_types[i] = (col_type, n_chars)
    if names is None:
        base = {
            c_type
            for c, c_type in zip(converters, column_types)
            if c._checked}
        if len(base) == 1:
            uniform_type, = base
            (ddtype, mdtype) = (uniform_type, bool)
        else:
            ddtype = [(defaultfmt % i, dt)
                      for (i, dt) in enumerate(sized_column_types)]
            if usemask:
                mdtype = [(defaultfmt % i, bool)
                           for (i, dt) in enumerate(sized_column_types)]
            else:
                ddtype = list(zip(names, sized_column_types))
                mdtype = list(zip(names, [bool] * len(sized_column_types)))
        output = np.array(data, dtype=ddtype)
        if usemask:
            outputmask = np.array(masks, dtype=mdtype)
    else:
        ddtype = list(zip(names, sized_column_types))
        mdtype = list(zip(names, [bool] * len(sized_column_types)))
        output = np.array(data, dtype=ddtype)
        if usemask:
            outputmask = np.array(masks, dtype=mdtype)
else:

```

```

1884: (8)             if names and dtype.names is not None:
1885: (12)             dtype.names = names
1886: (8)             if len(dtype_flat) > 1:
1887: (12)                 if '0' in (_.char for _ in dtype_flat):
1888: (16)                     if has_nested_fields(dtype):
1889: (20)                         raise NotImplementedError(
1890: (24)                             "Nested fields involving objects are not
supported...")
1891: (16)             else:
1892: (20)                 output = np.array(data, dtype=dtype)
1893: (12)             else:
1894: (16)                 rows = np.array(data, dtype=[(' ', _) for _ in dtype_flat])
1895: (16)                 output = rows.view(dtype)
1896: (12)             if usemask:
1897: (16)                 rowmasks = np.array(
1898: (20)                     masks, dtype=np.dtype([(' ', bool) for t in dtype_flat]))
1899: (16)                     mdtype = make_mask_descr(dtype)
1900: (16)                     outputmask = rowmasks.view(mdtype)
1901: (8)             else:
1902: (12)                 if user_converters:
1903: (16)                     ishomogeneous = True
1904: (16)                     descr = []
1905: (16)                     for i, ttype in enumerate([conv.type for conv in converters]):
1906: (20)                         if i in user_converters:
1907: (24)                             ishomogeneous &= (ttype == dtype.type)
1908: (24)                             if np.issubdtype(ttype, np.character):
1909: (28)                                 ttype = (ttype, max(len(row[i]) for row in data))
1910: (24)                                 descr.append(' ', ttype)
1911: (20)                             else:
1912: (24)                                 descr.append(' ', dtype))
1913: (16)                     if not ishomogeneous:
1914: (20)                         if len(descr) > 1:
1915: (24)                             dtype = np.dtype(descr)
1916: (20)                         else:
1917: (24)                             dtype = np.dtype(ttype)
1918: (12)                     output = np.array(data, dtype)
1919: (12)                     if usemask:
1920: (16)                         if dtype.names is not None:
1921: (20)                             mdtype = [(_, bool) for _ in dtype.names]
1922: (16)                         else:
1923: (20)                             mdtype = bool
1924: (16)                             outputmask = np.array(masks, dtype=mdtype)
1925: (4)                     names = output.dtype.names
1926: (4)                     if usemask and names:
1927: (8)                         for (name, conv) in zip(names, converters):
1928: (12)                             missing_values = [conv(_) for _ in conv.missing_values
1929: (30)                                 if _ != '']
1930: (12)                             for mval in missing_values:
1931: (16)                                 outputmask[name] |= (output[name] == mval)
1932: (4)                     if usemask:
1933: (8)                         output = output.view(MaskedArray)
1934: (8)                         output._mask = outputmask
1935: (4)                     output = _ensure_ndmin_ndarray(output, ndmin=ndmin)
1936: (4)                     if unpack:
1937: (8)                         if names is None:
1938: (12)                             return output.T
1939: (8)                         elif len(names) == 1:
1940: (12)                             return output[names[0]]
1941: (8)                         else:
1942: (12)                             return [output[field] for field in names]
1943: (4)                     return output
1944: (0)             _genfromtxt_with_like = array_function_dispatch()(genfromtxt)
1945: (0)             def recfromtxt(fname, **kwargs):
1946: (4)                 """
1947: (4)                 Load ASCII data from a file and return it in a record array.
1948: (4)                 If ``usemask=False`` a standard `recarray` is returned,
1949: (4)                 if ``usemask=True`` a MaskedRecords array is returned.
1950: (4)                 Parameters
1951: (4)                     -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1952: (4)         fname, kwargs : For a description of input parameters, see `genfromtxt`.
1953: (4)         See Also
1954: (4)
1955: (4)         -----
1956: (4)         numpy.genfromtxt : generic function
1957: (4)         Notes
1958: (4)         -----
1959: (4)         By default, `dtype` is None, which means that the data-type of the output
1960: (4)         array will be determined from the data.
1961: (4)         """
1962: (4)         kwargs.setdefault("dtype", None)
1963: (4)         usemask = kwargs.get('usemask', False)
1964: (4)         output = genfromtxt(fname, **kwargs)
1965: (8)         if usemask:
1966: (8)             from numpy.ma.mrecords import MaskedRecords
1967: (8)             output = output.view(MaskedRecords)
1968: (8)         else:
1969: (8)             output = output.view(np.recarray)
1970: (0)         return output
1971: (4)     def recfromcsv(fname, **kwargs):
1972: (4)         """
1973: (4)         Load ASCII data stored in a comma-separated file.
1974: (4)         The returned array is a record array (if ``usemask=False``, see
1975: (4)         ``recarray``) or a masked record array (if ``usemask=True``, see
1976: (4)         ``ma.mrecords.MaskedRecords``).
1977: (4)         Parameters
1978: (4)         -----
1979: (4)         fname, kwargs : For a description of input parameters, see `genfromtxt`.
1980: (4)         See Also
1981: (4)         -----
1982: (4)         numpy.genfromtxt : generic function to load ASCII data.
1983: (4)         Notes
1984: (4)         -----
1985: (4)         By default, `dtype` is None, which means that the data-type of the output
1986: (4)         array will be determined from the data.
1987: (4)         """
1988: (4)         kwargs.setdefault("case_sensitive", "lower")
1989: (4)         kwargs.setdefault("names", True)
1990: (4)         kwargs.setdefault("delimiter", ",")
1991: (4)         kwargs.setdefault("dtype", None)
1992: (4)         output = genfromtxt(fname, **kwargs)
1993: (4)         usemask = kwargs.get("usemask", False)
1994: (4)         if usemask:
1995: (8)             from numpy.ma.mrecords import MaskedRecords
1996: (8)             output = output.view(MaskedRecords)
1997: (8)         else:
1998: (8)             output = output.view(np.recarray)
1999: (4)         return output

```

---

#### File 214 - polynomial.py:

```

1: (0)         """
2: (0)         Functions to operate on polynomials.
3: (0)         """
4: (0)         __all__ = ['poly', 'roots', 'polyint', 'polyder', 'polyadd',
5: (11)           'polysub', 'polymul', 'polydiv', 'polyval', 'poly1d',
6: (11)           'polyfit', 'RankWarning']
7: (0)         import functools
8: (0)         import re
9: (0)         import warnings
10: (0)        from .._utils import set_module
11: (0)        import numpy.core.numeric as NX
12: (0)        from numpy.core import (isscalar, abs, finfo, atleast_1d, hstack, dot, array,
13: (24)          ones)
14: (0)        from numpy.core import overrides
15: (0)        from numpy.lib.twodim_base import diag, vander
16: (0)        from numpy.lib.function_base import trim_zeros
17: (0)        from numpy.lib.type_check import iscomplex, real, imag, mintypecode

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

18: (0)
19: (0)
20: (4)     from numpy.linalg import eigvals, lstsq, inv
21: (0)     array_function_dispatch = functools.partial(
22: (0)         overrides.array_function_dispatch, module='numpy')
23: (4)     @set_module('numpy')
24: (4)     class RankWarning(UserWarning):
25: (4)         """
26: (4)             Issued by `polyfit` when the Vandermonde matrix is rank deficient.
27: (4)             For more information, a way to suppress the warning, and an example of
28: (4)             `RankWarning` being issued, see `polyfit`.
29: (4)         """
30: (0)         pass
31: (0)     def _poly_dispatcher(seq_of_zeros):
32: (4)         return seq_of_zeros
33: (0)     @array_function_dispatch(_poly_dispatcher)
34: (0)     def poly(seq_of_zeros):
35: (4)         """
36: (4)             Find the coefficients of a polynomial with the given sequence of roots.
37: (4)             .. note::
38: (7)                 This forms part of the old polynomial API. Since version 1.4, the
39: (7)                 new polynomial API defined in `numpy.polynomial` is preferred.
40: (7)                 A summary of the differences can be found in the
41: (7)                     :doc:`transition guide 

```

```

87: (4) Given a sequence of a polynomial's zeros:
88: (4) >>> np.poly((0, 0, 0)) # Multiple root example
89: (4) array([1., 0., 0., 0.])
90: (4) The line above represents  $z^{**3} + 0*z^{**2} + 0*z + 0$ .
91: (4) >>> np.poly((-1./2, 0, 1./2))
92: (4) array([ 1. , 0. , -0.25, 0. ])
93: (4) The line above represents  $z^{**3} - z/4$ 
94: (4) >>> np.poly((np.random.random(1)[0], 0, np.random.random(1)[0]))
95: (4) array([ 1. , -0.77086955, 0.08618131, 0. ]) # random
96: (4) Given a square array object:
97: (4) >>> P = np.array([[0, 1./3], [-1./2, 0]])
98: (4) >>> np.poly(P)
99: (4) array([1. , 0. , 0.16666667])
100: (4) Note how in all cases the leading coefficient is always 1.
101: (4) """
102: (4) seq_of_zeros = atleast_1d(seq_of_zeros)
103: (4) sh = seq_of_zeros.shape
104: (4) if len(sh) == 2 and sh[0] == sh[1] and sh[0] != 0:
105: (8)     seq_of_zeros = eigvals(seq_of_zeros)
106: (4) elif len(sh) == 1:
107: (8)     dt = seq_of_zeros.dtype
108: (8)     if dt != object:
109: (12)         seq_of_zeros = seq_of_zeros.astype(mintypecode(dt.char))
110: (4) else:
111: (8)     raise ValueError("input must be 1d or non-empty square 2d array.")
112: (4) if len(seq_of_zeros) == 0:
113: (8)     return 1.0
114: (4) dt = seq_of_zeros.dtype
115: (4) a = ones((1,), dtype=dt)
116: (4) for zero in seq_of_zeros:
117: (8)     a = NX.convolve(a, array([1, -zero], dtype=dt), mode='full')
118: (4) if issubclass(a.dtype.type, NX.complexfloating):
119: (8)     roots = NX.asarray(seq_of_zeros, complex)
120: (8)     if NX.all(NX.sort(roots) == NX.sort(roots.conjugate())):
121: (12)         a = a.real.copy()
122: (4)     return a
123: (0) def _roots_dispatcher(p):
124: (4)     return p
125: (0) @array_function_dispatch(_roots_dispatcher)
126: (0) def roots(p):
127: (4) """
128: (4)     Return the roots of a polynomial with coefficients given in p.
129: (4) .. note::
130: (7)     This forms part of the old polynomial API. Since version 1.4, the
131: (7)     new polynomial API defined in `numpy.polynomial` is preferred.
132: (7)     A summary of the differences can be found in the
133: (7)     :doc:`transition guide 

```

```

156: (4)          Notes
157: (4)          -----
158: (4)          The algorithm relies on computing the eigenvalues of the
159: (4)          companion matrix [1]_.
160: (4)          References
161: (4)          -----
162: (4)          .. [1] R. A. Horn & C. R. Johnson, *Matrix Analysis*. Cambridge, UK:
163: (8)          Cambridge University Press, 1999, pp. 146-7.
164: (4)          Examples
165: (4)          -----
166: (4)          >>> coeff = [3.2, 2, 1]
167: (4)          >>> np.roots(coeff)
168: (4)          array([-0.3125+0.46351241j, -0.3125-0.46351241j])
169: (4)          """
170: (4)          p = atleast_1d(p)
171: (4)          if p.ndim != 1:
172: (8)              raise ValueError("Input must be a rank-1 array.")
173: (4)          non_zero = NX.nonzero(NX.ravel(p))[0]
174: (4)          if len(non_zero) == 0:
175: (8)              return NX.array([])
176: (4)          trailing_zeros = len(p) - non_zero[-1] - 1
177: (4)          p = p[int(non_zero[0]):int(non_zero[-1])+1]
178: (4)          if not issubclass(p.dtype.type, (NX.floating, NX.complexfloating)):
179: (8)              p = p.astype(float)
180: (4)          N = len(p)
181: (4)          if N > 1:
182: (8)              A = diag(NX.ones((N-2,), p.dtype), -1)
183: (8)              A[0,:] = -p[1:] / p[0]
184: (8)              roots = eigvals(A)
185: (4)          else:
186: (8)              roots = NX.array([])
187: (4)          roots = hstack((roots, NX.zeros(trailing_zeros, roots.dtype)))
188: (4)          return roots
189: (0)          def _polyint_dispatcher(p, m=None, k=None):
190: (4)              return (p,)
191: (0)          @array_function_dispatch(_polyint_dispatcher)
192: (0)          def polyint(p, m=1, k=None):
193: (4)              """
194: (4)              Return an antiderivative (indefinite integral) of a polynomial.
195: (4)              ..
196: (7)                  note:: This forms part of the old polynomial API. Since version 1.4, the
197: (7)                  new polynomial API defined in `numpy.polynomial` is preferred.
198: (7)                  A summary of the differences can be found in the
199: (7)                  :doc:`transition guide 

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

225: (4)          >>> p = np.poly1d([1,1,1])
226: (4)          >>> P = np.polyint(p)
227: (4)          >>> P
228: (5)          poly1d([ 0.33333333,  0.5           ,  1.           ,  0.           ]) # may vary
229: (4)          >>> np.polyder(P) == p
230: (4)          True
231: (4)          The integration constants default to zero, but can be specified:
232: (4)          >>> P = np.polyint(p, 3)
233: (4)          >>> P(0)
234: (4)          0.0
235: (4)          >>> np.polyder(P)(0)
236: (4)          0.0
237: (4)          >>> np.polyder(P, 2)(0)
238: (4)          0.0
239: (4)          >>> P = np.polyint(p, 3, k=[6,5,3])
240: (4)          >>> P
241: (4)          poly1d([ 0.01666667,  0.04166667,  0.16666667,  3. ,  5. ,  3. ]) # may
vary
242: (4)          Note that 3 = 6 / 2!, and that the constants are given in the order of
243: (4)          integrations. Constant of the highest-order polynomial term comes first:
244: (4)          >>> np.polyder(P, 2)(0)
245: (4)          6.0
246: (4)          >>> np.polyder(P, 1)(0)
247: (4)          5.0
248: (4)          >>> P(0)
249: (4)          3.0
250: (4)          """
251: (4)          m = int(m)
252: (4)          if m < 0:
253: (8)          raise ValueError("Order of integral must be positive (see polyder())")
254: (4)          if k is None:
255: (8)          k = NX.zeros(m, float)
256: (4)          k = atleast_1d(k)
257: (4)          if len(k) == 1 and m > 1:
258: (8)          k = k[0]*NX.ones(m, float)
259: (4)          if len(k) < m:
260: (8)          raise ValueError(
261: (14)          "k must be a scalar or a rank-1 array of length 1 or >m.")
262: (4)          truepoly = isinstance(p, poly1d)
263: (4)          p = NX.asarray(p)
264: (4)          if m == 0:
265: (8)          if truepoly:
266: (12)          return poly1d(p)
267: (8)          return p
268: (4)          else:
269: (8)          y = NX.concatenate((p.__truediv__(NX.arange(len(p), 0, -1)), [k[0]]))
270: (8)          val = polyint(y, m - 1, k=k[1:])
271: (8)          if truepoly:
272: (12)          return poly1d(val)
273: (8)          return val
274: (0)          def _polyder_dispatcher(p, m=None):
275: (4)          return (p,)
276: (0)          @array_function_dispatch(_polyder_dispatcher)
277: (0)          def polyder(p, m=1):
278: (4)          """
279: (4)          Return the derivative of the specified order of a polynomial.
280: (4)          .. note::
281: (7)          This forms part of the old polynomial API. Since version 1.4, the
282: (7)          new polynomial API defined in `numpy.polynomial` is preferred.
283: (7)          A summary of the differences can be found in the
284: (7)          :doc:`transition guide ` .
285: (4)          Parameters
286: (4)          -----
287: (4)          p : poly1d or sequence
288: (8)          Polynomial to differentiate.
289: (8)          A sequence is interpreted as polynomial coefficients, see `poly1d` .
290: (4)          m : int, optional
291: (8)          Order of differentiation (default: 1)
292: (4)          Returns

```

```

293: (4)
294: (4)
295: (8)     der : poly1d
296: (4)         A new polynomial representing the derivative.
297: (4)
298: (4)     See Also
299: (4)         -----
300: (4)         polyint : Anti-derivative of a polynomial.
301: (4)         poly1d : Class for one-dimensional polynomials.
302: (4)         Examples
303: (4)         -----
304: (4)         The derivative of the polynomial :math:`x^3 + x^2 + x^1 + 1` is:
305: (4)         >>> p = np.poly1d([1,1,1,1])
306: (4)         >>> p2 = np.polyder(p)
307: (4)         >>> p2
308: (4)         poly1d([3, 2, 1])
309: (4)         which evaluates to:
310: (4)         >>> p2(2.)
311: (4)         17.0
312: (4)         We can verify this, approximating the derivative with
313: (4)         `` $(f(x + h) - f(x))/h$ ``:
314: (4)         >>> (p(2. + 0.001) - p(2.)) / 0.001
315: (4)         17.007000999997857
316: (4)         The fourth-order derivative of a 3rd-order polynomial is zero:
317: (4)         >>> np.polyder(p, 2)
318: (4)         poly1d([6, 2])
319: (4)         >>> np.polyder(p, 3)
320: (4)         poly1d([6])
321: (4)         >>> np.polyder(p, 4)
322: (4)         poly1d([0])
323: (4)         """
324: (4)         m = int(m)
325: (4)         if m < 0:
326: (4)             raise ValueError("Order of derivative must be positive (see polyint)")
327: (4)         truepoly = isinstance(p, poly1d)
328: (4)         p = NX.asarray(p)
329: (4)         n = len(p) - 1
330: (4)         y = p[:-1] * NX.arange(n, 0, -1)
331: (4)         if m == 0:
332: (4)             val = p
333: (4)         else:
334: (4)             val = polyder(y, m - 1)
335: (4)         if truepoly:
336: (4)             val = poly1d(val)
337: (4)         return val
338: (0)     def _polyfit_dispatcher(x, y, deg, rcond=None, full=None, w=None, cov=None):
339: (4)         return (x, y, w)
340: (0)     @array_function_dispatch(_polyfit_dispatcher)
341: (4)     def polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False):
342: (4)         """
343: (4)             Least squares polynomial fit.
344: (4)             .. note::
345: (4)                 This forms part of the old polynomial API. Since version 1.4, the
346: (4)                 new polynomial API defined in `numpy.polynomial` is preferred.
347: (4)                 A summary of the differences can be found in the
348: (4)                 :doc:`transition guide p(x) = p[0] * x^{*deg} + \dots + p[deg]`` of degree `deg`
350: (4)                 to points `(x, y)`. Returns a vector of coefficients `p` that minimises
351: (4)                 the squared error in the order `deg`, `deg-1`, ... `0`.
352: (4)                 The `Polynomial.fit <numpy.polynomial.polynomial.Polynomial.fit>` class
353: (4)                 method is recommended for new code as it is more stable numerically. See
354: (4)                 the documentation of the method for more information.
355: (4)             Parameters
356: (4)             -----
357: (4)             x : array_like, shape (M,)
358: (4)                 x-coordinates of the M sample points `` $(x[i], y[i])$ ``.
359: (4)             y : array_like, shape (M,) or (M, K)
360: (4)                 y-coordinates of the sample points. Several data sets of sample
361: (4)                 points sharing the same x-coordinates can be fitted at once by
362: (4)                 passing in a 2D-array that contains one dataset per column.
363: (4)             deg : int

```

```

362: (8) Degree of the fitting polynomial
363: (4) rcond : float, optional
364: (8) Relative condition number of the fit. Singular values smaller than
365: (8) this relative to the largest singular value will be ignored. The
366: (8) default value is len(x)*eps, where eps is the relative precision of
367: (8) the float type, about 2e-16 in most cases.
368: (4) full : bool, optional
369: (8) Switch determining nature of return value. When it is False (the
370: (8) default) just the coefficients are returned, when True diagnostic
371: (8) information from the singular value decomposition is also returned.
372: (4) w : array_like, shape (M,), optional
373: (8) Weights. If not None, the weight ``w[i]`` applies to the unsquared
374: (8) residual ``y[i] - y_hat[i]`` at ``x[i]``. Ideally the weights are
375: (8) chosen so that the errors of the products ``w[i]*y[i]`` all have the
376: (8) same variance. When using inverse-variance weighting, use
377: (8) ``w[i] = 1/sigma(y[i])``. The default value is None.
378: (4) cov : bool or str, optional
379: (8) If given and not `False`, return not just the estimate but also its
380: (8) covariance matrix. By default, the covariance are scaled by
381: (8) chi2/dof, where dof = M - (deg + 1), i.e., the weights are presumed
382: (8) to be unreliable except in a relative sense and everything is scaled
383: (8) such that the reduced chi2 is unity. This scaling is omitted if
384: (8) ``cov='unscaled'``, as is relevant for the case that the weights are
385: (8) w = 1/sigma, with sigma known to be a reliable estimate of the
386: (8) uncertainty.
387: (4) Returns
388: (4) -----
389: (4) p : ndarray, shape (deg + 1,) or (deg + 1, K)
390: (8) Polynomial coefficients, highest power first. If `y` was 2-D, the
391: (8) coefficients for `k`-th data set are in ``p[:,k]``.
392: (4) residuals, rank, singular_values, rcond
393: (8) These values are only returned if ``full == True``
394: (8) - residuals -- sum of squared residuals of the least squares fit
395: (8) - rank -- the effective rank of the scaled Vandermonde
396: (11) coefficient matrix
397: (8) - singular_values -- singular values of the scaled Vandermonde
398: (11) coefficient matrix
399: (8) - rcond -- value of `rcond`.
400: (8) For more details, see `numpy.linalg.lstsq`.
401: (4) V : ndarray, shape (M,M) or (M,M,K)
402: (8) Present only if ``full == False`` and ``cov == True``. The covariance
403: (8) matrix of the polynomial coefficient estimates. The diagonal of
404: (8) this matrix are the variance estimates for each coefficient. If y
405: (8) is a 2-D array, then the covariance matrix for the `k`-th data set
406: (8) are in ``V[:, :, k]``
407: (4) Warns
408: (4) -----
409: (4) RankWarning
410: (8) The rank of the coefficient matrix in the least-squares fit is
411: (8) deficient. The warning is only raised if ``full == False``.
412: (8) The warnings can be turned off by
413: (8) >>> import warnings
414: (8) >>> warnings.simplefilter('ignore', np.RankWarning)
415: (4) See Also
416: (4) -----
417: (4) polyval : Compute polynomial values.
418: (4) linalg.lstsq : Computes a least-squares fit.
419: (4) scipy.interpolate.UnivariateSpline : Computes spline fits.
420: (4) Notes
421: (4) -----
422: (4) The solution minimizes the squared error
423: (4) .. math::
424: (8) E = \sum_{j=0}^k |p(x_j) - y_j|^2
425: (4) in the equations::
426: (8) x[0]**n * p[0] + ... + x[0] * p[n-1] + p[n] = y[0]
427: (8) x[1]**n * p[0] + ... + x[1] * p[n-1] + p[n] = y[1]
428: (8) ...
429: (8) x[k]**n * p[0] + ... + x[k] * p[n-1] + p[n] = y[k]
430: (4) The coefficient matrix of the coefficients `p` is a Vandermonde matrix.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

431: (4) `polyfit` issues a `RankWarning` when the least-squares fit is badly
432: (4) conditioned. This implies that the best fit is not well-defined due
433: (4) to numerical error. The results may be improved by lowering the polynomial
434: (4) degree or by replacing `x` by `x` - `x`.mean(). The `rcond` parameter
435: (4) can also be set to a value smaller than its default, but the resulting
436: (4) fit may be spurious: including contributions from the small singular
437: (4) values can add numerical noise to the result.
438: (4) Note that fitting polynomial coefficients is inherently badly conditioned
439: (4) when the degree of the polynomial is large or the interval of sample
points
440: (4) is badly centered. The quality of the fit should always be checked in
these
441: (4) cases. When polynomial fits are not satisfactory, splines may be a good
442: (4) alternative.
443: (4) References
444: (4) -----
445: (4) .. [1] Wikipedia, "Curve fitting",
446: (11) https://en.wikipedia.org/wiki/Curve_fitting
447: (4) .. [2] Wikipedia, "Polynomial interpolation",
448: (11) https://en.wikipedia.org/wiki/Polynomial_interpolation
449: (4) Examples
450: (4) -----
451: (4) >>> import warnings
452: (4) >>> x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
453: (4) >>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
454: (4) >>> z = np.polyfit(x, y, 3)
455: (4) >>> z
456: (4) array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254]) # may vary
457: (4) It is convenient to use `poly1d` objects for dealing with polynomials:
458: (4) >>> p = np.poly1d(z)
459: (4) >>> p(0.5)
460: (4) 0.6143849206349179 # may vary
461: (4) >>> p(3.5)
462: (4) -0.34732142857143039 # may vary
463: (4) >>> p(10)
464: (4) 22.579365079365115 # may vary
465: (4) High-order polynomials may oscillate wildly:
466: (4) >>> with warnings.catch_warnings():
467: (4) ...     warnings.simplefilter('ignore', np.RankWarning)
468: (4) ...     p30 = np.poly1d(np.polyfit(x, y, 30))
469: (4) ...
470: (4) >>> p30(4)
471: (4) -0.80000000000000204 # may vary
472: (4) >>> p30(5)
473: (4) -0.9999999999999445 # may vary
474: (4) >>> p30(4.5)
475: (4) -0.10547061179440398 # may vary
476: (4) Illustration:
477: (4) >>> import matplotlib.pyplot as plt
478: (4) >>> xp = np.linspace(-2, 6, 100)
479: (4) >>> _ = plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
480: (4) >>> plt.ylim(-2,2)
481: (4) (-2, 2)
482: (4) >>> plt.show()
483: (4) """
484: (4)     order = int(deg) + 1
485: (4)     x = NX.asarray(x) + 0.0
486: (4)     y = NX.asarray(y) + 0.0
487: (4)     if deg < 0:
488: (8)         raise ValueError("expected deg >= 0")
489: (4)     if x.ndim != 1:
490: (8)         raise TypeError("expected 1D vector for x")
491: (4)     if x.size == 0:
492: (8)         raise TypeError("expected non-empty vector for x")
493: (4)     if y.ndim < 1 or y.ndim > 2:
494: (8)         raise TypeError("expected 1D or 2D array for y")
495: (4)     if x.shape[0] != y.shape[0]:
496: (8)         raise TypeError("expected x and y to have same length")
497: (4)     if rcond is None:

```

```

498: (8)          rcond = len(x)*finfo(x.dtype).eps
499: (4)          lhs = vander(x, order)
500: (4)          rhs = y
501: (4)          if w is not None:
502: (8)            w = NX.asarray(w) + 0.0
503: (8)          if w.ndim != 1:
504: (12)            raise TypeError("expected a 1-d array for weights")
505: (8)          if w.shape[0] != y.shape[0]:
506: (12)            raise TypeError("expected w and y to have the same length")
507: (8)          lhs *= w[:, NX.newaxis]
508: (8)          if rhs.ndim == 2:
509: (12)            rhs *= w[:, NX.newaxis]
510: (8)          else:
511: (12)            rhs *= w
512: (4)          scale = NX.sqrt((lhs*lhs).sum(axis=0))
513: (4)          lhs /= scale
514: (4)          c, resids, rank, s = lstsq(lhs, rhs, rcond)
515: (4)          c = (c.T/scale).T # broadcast scale coefficients
516: (4)          if rank != order and not full:
517: (8)            msg = "Polyfit may be poorly conditioned"
518: (8)            warnings.warn(msg, RankWarning, stacklevel=2)
519: (4)          if full:
520: (8)            return c, resids, rank, s, rcond
521: (4)          elif cov:
522: (8)            Vbase = inv(dot(lhs.T, lhs))
523: (8)            Vbase /= NX.outer(scale, scale)
524: (8)            if cov == "unscaled":
525: (12)              fac = 1
526: (8)            else:
527: (12)              if len(x) <= order:
528: (16)                raise ValueError("the number of data points must exceed order
"
529: (33)                  "to scale the covariance matrix")
530: (12)                fac = resids / (len(x) - order)
531: (8)                if y.ndim == 1:
532: (12)                  return c, Vbase * fac
533: (8)                else:
534: (12)                  return c, Vbase[:, :, NX.newaxis] * fac
535: (4)                else:
536: (8)                  return c
537: (0)          def _polyval_dispatcher(p, x):
538: (4)            return (p, x)
539: (0)          @array_function_dispatch(_polyval_dispatcher)
540: (0)          def polyval(p, x):
541: (4)            """
542: (4)              Evaluate a polynomial at specific values.
543: (4)              .. note::
544: (7)                This forms part of the old polynomial API. Since version 1.4, the
545: (7)                new polynomial API defined in `numpy.polynomial` is preferred.
546: (7)                A summary of the differences can be found in the
547: (7)                  :doc:`transition guide 

```

```

566: (7) polynomials, i.e., `x` is "substituted" in `p` and the simplified
567: (7) result is returned. In addition, the type of `x` - array_like or
568: (7) poly1d - governs the type of the output: `x` array_like => `values`
569: (7) array_like, `x` a poly1d object => `values` is also.
570: (4) See Also
571: (4) -----
572: (4) poly1d: A polynomial class.
573: (4) Notes
574: (4) -----
575: (4) Horner's scheme [1]_ is used to evaluate the polynomial. Even so,
576: (4) for polynomials of high degree the values may be inaccurate due to
577: (4) rounding errors. Use carefully.
578: (4) If `x` is a subtype of `ndarray` the return value will be of the same
type.
579: (4) References
580: (4) -----
581: (4) .. [1] I. N. Bronshtein, K. A. Semendyayev, and K. A. Hirsch (Eng.
582: (7) trans. Ed.), *Handbook of Mathematics*, New York, Van Nostrand
583: (7) Reinhold Co., 1985, pg. 720.
584: (4) Examples
585: (4) -----
586: (4) >>> np.polyval([3,0,1], 5) # 3 * 5**2 + 0 * 5**1 + 1
587: (4) 76
588: (4) >>> np.polyval([3,0,1], np.poly1d(5))
589: (4) poly1d([76])
590: (4) >>> np.polyval(np.poly1d([3,0,1]), 5)
591: (4) 76
592: (4) >>> np.polyval(np.poly1d([3,0,1]), np.poly1d(5))
593: (4) poly1d([76])
594: (4) """
595: (4) p = NX.asarray(p)
596: (4) if isinstance(x, poly1d):
597: (8)     y = 0
598: (4) else:
599: (8)     x = NX.asarray(x)
600: (8)     y = NX.zeros_like(x)
601: (4)     for pv in p:
602: (8)         y = y * x + pv
603: (4)     return y
604: (0) def _binary_op_dispatcher(a1, a2):
605: (4)     return (a1, a2)
606: (0) @array_function_dispatch(_binary_op_dispatcher)
607: (0) def polyadd(a1, a2):
608: (4) """
609: (4) Find the sum of two polynomials.
610: (4) .. note::
611: (7) This forms part of the old polynomial API. Since version 1.4, the
612: (7) new polynomial API defined in `numpy.polynomial` is preferred.
613: (7) A summary of the differences can be found in the
614: (7) :doc:`transition guide ` .
615: (4) Returns the polynomial resulting from the sum of two input polynomials.
616: (4) Each input must be either a poly1d object or a 1D sequence of polynomial
617: (4) coefficients, from highest to lowest degree.
618: (4) Parameters
619: (4) -----
620: (4) a1, a2 : array_like or poly1d object
621: (8)     Input polynomials.
622: (4) Returns
623: (4) -----
624: (4) out : ndarray or poly1d object
625: (8)     The sum of the inputs. If either input is a poly1d object, then the
626: (8)     output is also a poly1d object. Otherwise, it is a 1D array of
627: (8)     polynomial coefficients from highest to lowest degree.
628: (4) See Also
629: (4) -----
630: (4) poly1d : A one-dimensional polynomial class.
631: (4) poly, polyadd, polyder, polydiv, polyfit, polyint, polysub, polyval
632: (4) Examples
633: (4) -----

```

```

634: (4)          >>> np.polyadd([1, 2], [9, 5, 4])
635: (4)          array([9, 6, 6])
636: (4)          Using poly1d objects:
637: (4)          >>> p1 = np.poly1d([1, 2])
638: (4)          >>> p2 = np.poly1d([9, 5, 4])
639: (4)          >>> print(p1)
640: (4)          1 x + 2
641: (4)          >>> print(p2)
642: (7)          2
643: (4)          9 x + 5 x + 4
644: (4)          >>> print(np.polyadd(p1, p2))
645: (7)          2
646: (4)          9 x + 6 x + 6
647: (4)          """
648: (4)          truepoly = (isinstance(a1, poly1d) or isinstance(a2, poly1d))
649: (4)          a1 = atleast_1d(a1)
650: (4)          a2 = atleast_1d(a2)
651: (4)          diff = len(a2) - len(a1)
652: (4)          if diff == 0:
653: (8)              val = a1 + a2
654: (4)          elif diff > 0:
655: (8)              zr = NX.zeros(diff, a1.dtype)
656: (8)              val = NX.concatenate((zr, a1)) + a2
657: (4)          else:
658: (8)              zr = NX.zeros(abs(diff), a2.dtype)
659: (8)              val = a1 + NX.concatenate((zr, a2))
660: (4)          if truepoly:
661: (8)              val = poly1d(val)
662: (4)          return val
663: (0)          @array_function_dispatch(_binary_op_dispatcher)
664: (0)          def polysub(a1, a2):
665: (4)          """
666: (4)          Difference (subtraction) of two polynomials.
667: (4)          .. note::
668: (7)              This forms part of the old polynomial API. Since version 1.4, the
669: (7)              new polynomial API defined in `numpy.polynomial` is preferred.
670: (7)              A summary of the differences can be found in the
671: (7)              :doc:`transition guide 

```

```

703: (8)             val = a1 - NX.concatenate((zr, a2))
704: (4)             if truepoly:
705: (8)                 val = poly1d(val)
706: (4)             return val
707: (0) @array_function_dispatch(_binary_op_dispatcher)
708: (0) def polymul(a1, a2):
709: (4)     """
710: (4)         Find the product of two polynomials.
711: (4)         .. note::
712: (7)             This forms part of the old polynomial API. Since version 1.4, the
713: (7)             new polynomial API defined in `numpy.polynomial` is preferred.
714: (7)             A summary of the differences can be found in the
715: (7)                 :doc:`transition guide </reference/routines.polynomials>`.
716: (4)             Finds the polynomial resulting from the multiplication of the two input
717: (4)             polynomials. Each input must be either a poly1d object or a 1D sequence
718: (4)             of polynomial coefficients, from highest to lowest degree.
719: (4)             Parameters
720: (4)             -----
721: (4)             a1, a2 : array_like or poly1d object
722: (8)                 Input polynomials.
723: (4)             Returns
724: (4)             -----
725: (4)             out : ndarray or poly1d object
726: (8)                 The polynomial resulting from the multiplication of the inputs. If
727: (8)                 either inputs is a poly1d object, then the output is also a poly1d
728: (8)                 object. Otherwise, it is a 1D array of polynomial coefficients from
729: (8)                 highest to lowest degree.
730: (4)             See Also
731: (4)             -----
732: (4)             poly1d : A one-dimensional polynomial class.
733: (4)             poly, polyadd, polyder, polydiv, polyfit, polyint, polysub, polyval
734: (4)             convolve : Array convolution. Same output as polymul, but has parameter
735: (15)                for overlap mode.
736: (4)             Examples
737: (4)             -----
738: (4)             >>> np.polymul([1, 2, 3], [9, 5, 1])
739: (4)             array([ 9, 23, 38, 17,  3])
740: (4)             Using poly1d objects:
741: (4)             >>> p1 = np.poly1d([1, 2, 3])
742: (4)             >>> p2 = np.poly1d([9, 5, 1])
743: (4)             >>> print(p1)
744: (7)                 2
745: (4)                 1 x + 2 x + 3
746: (4)             >>> print(p2)
747: (7)                 2
748: (4)                 9 x + 5 x + 1
749: (4)             >>> print(np.polymul(p1, p2))
750: (7)                 4      3      2
751: (4)                 9 x + 23 x + 38 x + 17 x + 3
752: (4)                 """
753: (4)             truepoly = (isinstance(a1, poly1d) or isinstance(a2, poly1d))
754: (4)             a1, a2 = poly1d(a1), poly1d(a2)
755: (4)             val = NX.convolve(a1, a2)
756: (4)             if truepoly:
757: (8)                 val = poly1d(val)
758: (4)             return val
759: (0) def _polydiv_dispatcher(u, v):
760: (4)             return (u, v)
761: (0) @array_function_dispatch(_polydiv_dispatcher)
762: (0) def polydiv(u, v):
763: (4)             """
764: (4)                 Returns the quotient and remainder of polynomial division.
765: (4)                 .. note::
766: (7)                     This forms part of the old polynomial API. Since version 1.4, the
767: (7)                     new polynomial API defined in `numpy.polynomial` is preferred.
768: (7)                     A summary of the differences can be found in the
769: (7)                         :doc:`transition guide </reference/routines.polynomials>`.
770: (4)                     The input arrays are the coefficients (including any coefficients
771: (4)                     equal to zero) of the "numerator" (dividend) and "denominator"

```

```

772: (4)                               (divisor) polynomials, respectively.
773: (4)                               Parameters
774: (4)                               -----
775: (4)                                 u : array_like or poly1d
776: (8)                                 Dividend polynomial's coefficients.
777: (4)                                 v : array_like or poly1d
778: (8)                                 Divisor polynomial's coefficients.
779: (4)                               Returns
780: (4)                               -----
781: (4)                                 q : ndarray
782: (8)                                 Coefficients, including those equal to zero, of the quotient.
783: (4)                                 r : ndarray
784: (8)                                 Coefficients, including those equal to zero, of the remainder.
785: (4)                               See Also
786: (4)                               -----
787: (4)                                 poly, polyadd, polyder, polydiv, polyfit, polyint, polymul, polysub
788: (4)                                 polyval
789: (4)                               Notes
790: (4)                               -----
791: (4)                                 Both `u` and `v` must be 0-d or 1-d (ndim = 0 or 1), but `u.ndim` need
792: (4)                                 not equal `v.ndim`. In other words, all four possible combinations -
793: (4)                                 ``u.ndim = v.ndim = 0``, ``u.ndim = v.ndim = 1``,
794: (4)                                 ``u.ndim = 1, v.ndim = 0``, and ``u.ndim = 0, v.ndim = 1`` - work.
795: (4)                               Examples
796: (4)                               -----
797: (4)                                 .. math:: \frac{3x^2 + 5x + 2}{2x + 1} = 1.5x + 1.75, \text{ remainder } 0.25
798: (4)                                 >>> x = np.array([3.0, 5.0, 2.0])
799: (4)                                 >>> y = np.array([2.0, 1.0])
800: (4)                                 >>> np.polydiv(x, y)
801: (4)                                 (array([1.5, 1.75]), array([0.25]))
802: (4)                                 """
803: (4)                                 truepoly = (isinstance(u, poly1d) or isinstance(v, poly1d))
804: (4)                                 u = atleast_1d(u) + 0.0
805: (4)                                 v = atleast_1d(v) + 0.0
806: (4)                                 w = u[0] + v[0]
807: (4)                                 m = len(u) - 1
808: (4)                                 n = len(v) - 1
809: (4)                                 scale = 1. / v[0]
810: (4)                                 q = NX.zeros((max(m - n + 1, 1),), w.dtype)
811: (4)                                 r = u.astype(w.dtype)
812: (4)                                 for k in range(0, m-n+1):
813: (8)                                   d = scale * r[k]
814: (8)                                   q[k] = d
815: (8)                                   r[k:k+n+1] -= d*v
816: (4)                                 while NX.allclose(r[0], 0, rtol=1e-14) and (r.shape[-1] > 1):
817: (8)                                   r = r[1:]
818: (4)                                 if truepoly:
819: (8)                                   return poly1d(q), poly1d(r)
820: (4)                                 return q, r
821: (0)                               _poly_mat = re.compile(r"\*\*([0-9]*")
822: (0)                               def _raise_power(astr, wrap=70):
823: (4)                                 n = 0
824: (4)                                 line1 = ''
825: (4)                                 line2 = ''
826: (4)                                 output = ' '
827: (4)                                 while True:
828: (8)                                   mat = _poly_mat.search(astr, n)
829: (8)                                   if mat is None:
830: (12)                                     break
831: (8)                                   span = mat.span()
832: (8)                                   power = mat.groups()[0]
833: (8)                                   partstr = astr[n:span[0]]
834: (8)                                   n = span[1]
835: (8)                                   toadd2 = partstr + ' '*(len(power)-1)
836: (8)                                   toadd1 = ' '*(len(partstr)-1) + power
837: (8)                                   if ((len(line2) + len(toadd2) > wrap) or
838: (16)                                     (len(line1) + len(toadd1) > wrap)):
839: (12)                                     output += line1 + "\n" + line2 + "\n "
840: (12)                                     line1 = toadd1

```

```

841: (12)           line2 = toadd2
842: (8)            else:
843: (12)              line2 += partstr + ' *(len(power)-1)'
844: (12)              line1 += ' *(len(partstr)-1) + power'
845: (4)            output += line1 + "\n" + line2
846: (4)            return output + astr[n:]
847: (0)          @set_module('numpy')
848: (0)          class poly1d:
849: (4)              """
850: (4)                  A one-dimensional polynomial class.
851: (4)                  .. note::
852: (7)                      This forms part of the old polynomial API. Since version 1.4, the
853: (7)                      new polynomial API defined in `numpy.polynomial` is preferred.
854: (7)                      A summary of the differences can be found in the
855: (7)                          :doc:`transition guide 

```

```

910: (4)
911: (4)
912: (4)
913: (4)
914: (4)
915: (4)
916: (7)
917: (4)
918: (4)
919: (4)
920: (4)
921: (4)
922: (4)
923: (4)
924: (4)
925: (4)
926: (4)
927: (4)
928: (8)
929: (8)
930: (4)
931: (4)
932: (8)
933: (12)
934: (4)
935: (4)
936: (8)
937: (8)
938: (4)
939: (4)
940: (8)
941: (8)
942: (4)
943: (4)
944: (8)
945: (8)
946: (4)
947: (4)
948: (8)
949: (4)
950: (4)
951: (8)
952: (4)
953: (4)
954: (4)
955: (4)
956: (8)
957: (12)
958: (12)
959: (12)
960: (16)
961: (23)
962: (16)
963: (16)
964: (12)
965: (16)
966: (12)
967: (8)
968: (12)
969: (8)
970: (8)
971: (12)
972: (8)
973: (8)
974: (12)
975: (8)
976: (8)
977: (12)
978: (8)

        >>> np.square(p) # square of individual coefficients
        array([1, 4, 9])
        The variable used in the string representation of `p` can be modified,
        using the `variable` parameter:
        >>> p = np.poly1d([1,2,3], variable='z')
        >>> print(p)
            2
            1 z + 2 z + 3
        Construct a polynomial from its roots:
        >>> np.poly1d([1, 2], True)
        poly1d([ 1., -3.,  2.])
        This is the same polynomial as obtained by:
        >>> np.poly1d([1, -1]) * np.poly1d([1, -2])
        poly1d([ 1, -3,  2])
        """
        __hash__ = None
    @property
    def coeffs(self):
        """ The polynomial coefficients """
        return self._coeffs
    @coeffs.setter
    def coeffs(self, value):
        if value is not self._coeffs:
            raise AttributeError("Cannot set attribute")
    @property
    def variable(self):
        """ The name of the polynomial variable """
        return self._variable
    @property
    def order(self):
        """ The order or degree of the polynomial """
        return len(self._coeffs) - 1
    @property
    def roots(self):
        """ The roots of the polynomial, where self(x) == 0 """
        return roots(self._coeffs)
    @property
    def _coeffs(self):
        return self.__dict__['coeffs']
    @_coeffs.setter
    def _coeffs(self, coeffs):
        self.__dict__['coeffs'] = coeffs
    r = roots
    c = coef = coefficients = coeffs
    o = order
    def __init__(self, c_or_r, r=False, variable=None):
        if isinstance(c_or_r, poly1d):
            self._variable = c_or_r._variable
            self._coeffs = c_or_r._coeffs
            if set(c_or_r.__dict__) - set(self.__dict__):
                msg = ("In the future extra properties will not be copied "
                       "across when constructing one poly1d from another")
                warnings.warn(msg, FutureWarning, stacklevel=2)
            self.__dict__.update(c_or_r.__dict__)
        if variable is not None:
            self._variable = variable
        return
    if r:
        c_or_r = poly(c_or_r)
        c_or_r = atleast_1d(c_or_r)
        if c_or_r.ndim > 1:
            raise ValueError("Polynomial must be 1d only.")
        c_or_r = trim_zeros(c_or_r, trim='f')
        if len(c_or_r) == 0:
            c_or_r = NX.array([0], dtype=c_or_r.dtype)
        self._coeffs = c_or_r
        if variable is None:
            variable = 'x'
        self._variable = variable

```

```

979: (4)
980: (8)
981: (12)
982: (8)
983: (12)
984: (4)
985: (8)
986: (8)
987: (8)
988: (4)
989: (8)
990: (4)
991: (8)
992: (8)
993: (8)
994: (8)
995: (8)
996: (12)
997: (12)
998: (16)
999: (12)
1000: (8)
1001: (12)
1002: (16)
1003: (12)
1004: (16)
1005: (12)
1006: (16)
1007: (42)
1008: (12)
1009: (12)
1010: (16)
1011: (20)
1012: (16)
1013: (20)
1014: (24)
1015: (20)
1016: (24)
1017: (12)
1018: (16)
1019: (20)
1020: (16)
1021: (20)
1022: (16)
1023: (20)
1024: (12)
1025: (16)
1026: (20)
1027: (16)
1028: (20)
1029: (16)
1030: (20)
1031: (12)
1032: (16)
1033: (20)
1034: (24)
1035: (20)
1036: (24)
1037: (12)
1038: (16)
1039: (8)
1040: (4)
1041: (8)
1042: (4)
1043: (8)
1044: (4)
1045: (8)
1046: (4)
1047: (8)

    def __array__(self, t=None):
        if t:
            return NX.asarray(self.coeffs, t)
        else:
            return NX.asarray(self.coeffs)
    def __repr__(self):
        vals = repr(self.coeffs)
        vals = vals[6:-1]
        return "poly1d(%s)" % vals
    def __len__(self):
        return self.order
    def __str__(self):
        thestr = "0"
        var = self.variable
        coeffs = self.coeffs[NX.logical_or.accumulate(self.coeffs != 0)]
        N = len(coeffs)-1
        def fmt_float(q):
            s = '%.4g' % q
            if s.endswith('.0000'):
                s = s[:-5]
            return s
        for k, coeff in enumerate(coeffs):
            if not iscomplex(coeff):
                coefstr = fmt_float(real(coeff))
            elif real(coeff) == 0:
                coefstr = '%sj' % fmt_float(imag(coeff))
            else:
                coefstr = '(%s + %sj)' % (fmt_float(real(coeff)),
                                            fmt_float(imag(coeff)))
            power = (N-k)
            if power == 0:
                if coefstr != '0':
                    newstr = '%s' % (coefstr,)
                else:
                    if k == 0:
                        newstr = '0'
                    else:
                        newstr = ''
            elif power == 1:
                if coefstr == '0':
                    newstr = ''
                elif coefstr == 'b':
                    newstr = var
                else:
                    newstr = '%s %s' % (coefstr, var)
            else:
                if coefstr == '0':
                    newstr = ''
                elif coefstr == 'b':
                    newstr = '%s**%d' % (var, power,)
                else:
                    newstr = '%s %s**%d' % (coefstr, var, power)
            if k > 0:
                if newstr != '':
                    if newstr.startswith('-'):
                        thestr = "%s - %s" % (thestr, newstr[1:])
                    else:
                        thestr = "%s + %s" % (thestr, newstr)
                else:
                    thestr = newstr
        return _raise_power(thestr)
    def __call__(self, val):
        return polyval(self.coeffs, val)
    def __neg__(self):
        return poly1d(-self.coeffs)
    def __pos__(self):
        return self
    def __mul__(self, other):
        if isscalar(other):

```

```

1048: (12)             return poly1d(self.coeffs * other)
1049: (8)              else:
1050: (12)                  other = poly1d(other)
1051: (12)                  return poly1d(polymul(self.coeffs, other.coeffs))
1052: (4)  def __rmul__(self, other):
1053: (8)      if isscalar(other):
1054: (12)          return poly1d(other * self.coeffs)
1055: (8)      else:
1056: (12)          other = poly1d(other)
1057: (12)          return poly1d(polymul(self.coeffs, other.coeffs))
1058: (4)  def __add__(self, other):
1059: (8)      other = poly1d(other)
1060: (8)      return poly1d(polyadd(self.coeffs, other.coeffs))
1061: (4)  def __radd__(self, other):
1062: (8)      other = poly1d(other)
1063: (8)      return poly1d(polyadd(self.coeffs, other.coeffs))
1064: (4)  def __pow__(self, val):
1065: (8)      if not isscalar(val) or int(val) != val or val < 0:
1066: (12)          raise ValueError("Power to non-negative integers only.")
1067: (8)      res = [1]
1068: (8)      for _ in range(val):
1069: (12)          res = polymul(self.coeffs, res)
1070: (8)      return poly1d(res)
1071: (4)  def __sub__(self, other):
1072: (8)      other = poly1d(other)
1073: (8)      return poly1d(polysub(self.coeffs, other.coeffs))
1074: (4)  def __rsub__(self, other):
1075: (8)      other = poly1d(other)
1076: (8)      return poly1d(polysub(other.coeffs, self.coeffs))
1077: (4)  def __div__(self, other):
1078: (8)      if isscalar(other):
1079: (12)          return poly1d(self.coeffs/other)
1080: (8)      else:
1081: (12)          other = poly1d(other)
1082: (12)          return polydiv(self, other)
1083: (4)  __truediv__ = __div__
1084: (4)  def __rdiv__(self, other):
1085: (8)      if isscalar(other):
1086: (12)          return poly1d(other/self.coeffs)
1087: (8)      else:
1088: (12)          other = poly1d(other)
1089: (12)          return polydiv(other, self)
1090: (4)  __rtruediv__ = __rdiv__
1091: (4)  def __eq__(self, other):
1092: (8)      if not isinstance(other, poly1d):
1093: (12)          return NotImplemented
1094: (8)      if self.coeffs.shape != other.coeffs.shape:
1095: (12)          return False
1096: (8)      return (self.coeffs == other.coeffs).all()
1097: (4)  def __ne__(self, other):
1098: (8)      if not isinstance(other, poly1d):
1099: (12)          return NotImplemented
1100: (8)          return not self.__eq__(other)
1101: (4)  def __getitem__(self, val):
1102: (8)      ind = self.order - val
1103: (8)      if val > self.order:
1104: (12)          return self.coeffs.dtype.type(0)
1105: (8)      if val < 0:
1106: (12)          return self.coeffs.dtype.type(0)
1107: (8)      return self.coeffs[ind]
1108: (4)  def __setitem__(self, key, val):
1109: (8)      ind = self.order - key
1110: (8)      if key < 0:
1111: (12)          raise ValueError("Does not support negative powers.")
1112: (8)      if key > self.order:
1113: (12)          zr = NX.zeros(key-self.order, self.coeffs.dtype)
1114: (12)          self._coeffs = NX.concatenate((zr, self.coeffs))
1115: (12)          ind = 0
1116: (8)          self._coeffs[ind] = val

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1117: (8)         return
1118: (4)         def __iter__(self):
1119: (8)             return iter(self.coeffs)
1120: (4)         def integ(self, m=1, k=0):
1121: (8)             """
1122: (8)                 Return an antiderivative (indefinite integral) of this polynomial.
1123: (8)                 Refer to `polyint` for full documentation.
1124: (8)                 See Also
1125: (8)                     -----
1126: (8)                         polyint : equivalent function
1127: (8)                     """
1128: (8)             return poly1d(polyint(self.coeffs, m=m, k=k))
1129: (4)         def deriv(self, m=1):
1130: (8)             """
1131: (8)                 Return a derivative of this polynomial.
1132: (8)                 Refer to `polyder` for full documentation.
1133: (8)                 See Also
1134: (8)                     -----
1135: (8)                         polyder : equivalent function
1136: (8)                     """
1137: (8)             return poly1d(polyder(self.coeffs, m=m))
1138: (0)         warnings.simplefilter('always', RankWarning)

```

---

## File 215 - recfunctions.py:

```

1: (0)         """
2: (0)             Collection of utilities to manipulate structured arrays.
3: (0)             Most of these functions were initially implemented by John Hunter for
4: (0)                 matplotlib. They have been rewritten and extended for convenience.
5: (0)         """
6: (0)         import itertools
7: (0)         import numpy as np
8: (0)         import numpy.ma as ma
9: (0)         from numpy import ndarray, recarray
10: (0)        from numpy.ma import MaskedArray
11: (0)        from numpy.ma.mrecords import MaskedRecords
12: (0)        from numpy.core.overrides import array_function_dispatch
13: (0)        from numpy.lib._iotools import _is_string_like
14: (0)        _check_fill_value = np.ma.core._check_fill_value
15: (0)        __all__ = [
16: (4)            'append_fields', 'apply_along_fields', 'assign_fields_by_name',
17: (4)            'drop_fields', 'find_duplicates', 'flatten_descr',
18: (4)            'get_fieldstructure', 'get_names', 'get_names_flat',
19: (4)            'join_by', 'merge_arrays', 'rec_append_fields',
20: (4)            'rec_drop_fields', 'rec_join', 'recursive_fill_fields',
21: (4)            'rename_fields', 'repack_fields', 'require_fields',
22: (4)            'stack_arrays', 'structured_to_unstructured',
'unstructured_to_structured',
23: (4)        ]
24: (0)        def _recursive_fill_fields_dispatcher(input, output):
25: (4)            return (input, output)
26: (0)        @array_function_dispatch(_recursive_fill_fields_dispatcher)
27: (0)        def recursive_fill_fields(input, output):
28: (4)            """
29: (4)                Fills fields from output with fields from input,
30: (4)                with support for nested structures.
31: (4)                Parameters
32: (4)                    -----
33: (4)                        input : ndarray
34: (8)                            Input array.
35: (4)                        output : ndarray
36: (8)                            Output array.
37: (4)                Notes
38: (4)                    -----
39: (4)                    * `output` should be at least the same size as `input`
40: (4)                Examples
41: (4)                    -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

42: (4)
43: (4)
np.float64)])
44: (4)
45: (4)
46: (4)
47: (4)
48: (4)
49: (4)
50: (8)
51: (12)
52: (8)
53: (12)
54: (8)
55: (12)
56: (8)
57: (12)
58: (4)
59: (0)
60: (4)
61: (4)
62: (4)
a
63: (4)
64: (4)
that
65: (4)
66: (4)
Examples
-----
67: (4)
68: (4)
>>> dt = np.dtype([(('a', 'A'), np.int64), ('b', np.double, 3)])
69: (4)
70: (4)
71: (4)
72: (4)
73: (4)
74: (4)
75: (8)
76: (4)
77: (8)
78: (8)
79: (12)
80: (12)
81: (8)
82: (0)
83: (4)
84: (4)
85: (4)
86: (4)
87: (4)
88: (4)
89: (8)
90: (4)
91: (4)
92: (4)
93: (4)
94: (4)
95: (4)
96: (4)
97: (4)
98: (4)
99: (4)
100: (4)
101: (4)
102: (4)
103: (4)
104: (8)
105: (8)
106: (12)
107: (8)

    >>> from numpy.lib import recfunctions as rfn
    >>> a = np.array([(1, 10.), (2, 20.)], dtype=[('A', np.int64), ('B',
        >>> b = np.zeros((3,), dtype=a.dtype)
        >>> rfn.recursive_fill_fields(a, b)
        array([(1, 10.), (2, 20.), (0, 0.)], dtype=[('A', '<i8'), ('B', '<f8')])
    """
    newdtype = output.dtype
    for field in newdtype.names:
        try:
            current = input[field]
        except ValueError:
            continue
        if current.dtype.names is not None:
            recursive_fill_fields(current, output[field])
        else:
            output[field][:len(current)] = current
    return output
def _get_fieldspec(dtype):
    """
    Produce a list of name/dtype pairs corresponding to the dtype fields
    Similar to dtype.descr, but the second item of each tuple is a dtype, not
    a
    string. As a result, this handles subarray dtypes
    Can be passed to the dtype constructor to reconstruct the dtype, noting
    that
    this (deliberately) discards field offsets.
    Examples
    -----
    >>> dt = np.dtype([(('a', 'A'), np.int64), ('b', np.double, 3)])
    >>> dt.descr
    [(('a', 'A'), '<i8'), ('b', '<f8', (3,))]
    >>> _get_fieldspec(dt)
    [(('a', 'A'), dtype('int64')), ('b', dtype('<f8', (3,)))]
    """
    if dtype.names is None:
        return [('', dtype)]
    else:
        fields = ((name, dtype.fields[name]) for name in dtype.names)
        return [
            (name if len(f) == 2 else (f[2], name), f[0])
            for name, f in fields
        ]
def get_names(adtype):
    """
    Returns the field names of the input datatype as a tuple. Input datatype
    must have fields otherwise error is raised.
    Parameters
    -----
    adtype : dtype
        Input datatype
    Examples
    -----
    >>> from numpy.lib import recfunctions as rfn
    >>> rfn.get_names(np.empty((1,), dtype=[('A', int)]).dtype)
    ('A',)
    >>> rfn.get_names(np.empty((1,), dtype=[('A', int), ('B', float)]).dtype)
    ('A', 'B')
    >>> adtype = np.dtype([('a', int), ('b', [('ba', int), ('bb', int)])])
    >>> rfn.get_names(adtype)
    ('a', ('b', ('ba', 'bb')))

    listnames = []
    names = adtype.names
    for name in names:
        current = adtype[name]
        if current.names is not None:
            listnames.append((name, tuple(get_names(current))))
        else:

```

```

108: (12)           listnames.append(name)
109: (4)            return tuple(listnames)
110: (0)             def get_names_flat(adtype):
111: (4)             """
112: (4)               Returns the field names of the input datatype as a tuple. Input datatype
113: (4)               must have fields otherwise error is raised.
114: (4)               Nested structure are flattened beforehand.
115: (4)               Parameters
116: (4)               -----
117: (4)               adtype : dtype
118: (8)                 Input datatype
119: (4)               Examples
119: (4)               -----
121: (4)               >>> from numpy.lib import recfunctions as rfn
122: (4)               >>> rfn.get_names_flat(np.empty((1,), dtype=[('A', int)]).dtype) is None
123: (4)               False
124: (4)               >>> rfn.get_names_flat(np.empty((1,), dtype=[('A',int), ('B',
125: (4)               str)]).dtype)
126: (4)               ('A', 'B')
127: (4)               >>> adtype = np.dtype([('a', int), ('b', [('ba', int), ('bb', int)])])
128: (4)               >>> rfn.get_names_flat(adtype)
129: (4)               ('a', 'b', 'ba', 'bb')
129: (4)               """
130: (4)               listnames = []
131: (4)               names = adtype.names
132: (4)               for name in names:
133: (8)                 listnames.append(name)
134: (8)                 current = adtype[name]
135: (8)                 if current.names is not None:
136: (12)                   listnames.extend(get_names_flat(current))
137: (4)               return tuple(listnames)
138: (0)             def flatten_descr(ndtype):
139: (4)             """
140: (4)               Flatten a structured data-type description.
141: (4)               Examples
142: (4)               -----
143: (4)               >>> from numpy.lib import recfunctions as rfn
144: (4)               >>> ndtype = np.dtype([('a', '<i4'), ('b', [('ba', '<f8'), ('bb',
145: (4)               '<i4')])])
145: (4)               >>> rfn.flatten_descr(ndtype)
146: (4)               ('a', dtype('int32')), ('ba', dtype('float64')), ('bb', dtype('int32'))
147: (4)               """
148: (4)               names = ndtype.names
149: (4)               if names is None:
150: (8)                 return ('', ndtype,)
151: (4)               else:
152: (8)                 descr = []
153: (8)                 for field in names:
154: (12)                   (typ, _) = ndtype.fields[field]
155: (12)                   if typ.names is not None:
156: (16)                     descr.extend(flatten_descr(typ))
157: (12)                   else:
158: (16)                     descr.append((field, typ))
159: (8)                 return tuple(descr)
160: (0)             def _zip_dtype(seqarrays, flatten=False):
161: (4)               newdtype = []
162: (4)               if flatten:
163: (8)                 for a in seqarrays:
164: (12)                   newdtype.extend(flatten_descr(a.dtype))
165: (4)               else:
166: (8)                 for a in seqarrays:
167: (12)                   current = a.dtype
168: (12)                   if current.names is not None and len(current.names) == 1:
169: (16)                     newdtype.extend(_get_fieldspec(current))
170: (12)                   else:
171: (16)                     newdtype.append('', current))
172: (4)               return np.dtype(newdtype)
173: (0)             def _zip_descr(seqarrays, flatten=False):
174: (4)             """

```

```

175: (4)           Combine the dtype description of a series of arrays.
176: (4)           Parameters
177: (4)
178: (4)           seqarrays : sequence of arrays
179: (8)             Sequence of arrays
180: (4)           flatten : {boolean}, optional
181: (8)             Whether to collapse nested descriptions.
182: (4)
183: (4)           """
184: (0)           return _zip_dtype(seqarrays, flatten=flatten).descr
185: (4)           def get_fieldstructure(adtype, lastname=None, parents=None,):
186: (4)             """
187: (4)               Returns a dictionary with fields indexing lists of their parent fields.
188: (4)               This function is used to simplify access to fields nested in other fields.
189: (4)               Parameters
190: (4)
191: (8)                 adtype : np.dtype
192: (4)                   Input datatype
193: (4)                 lastname : optional
194: (8)                   Last processed field name (used internally during recursion).
195: (4)                 parents : dictionary
196: (8)                   Dictionary of parent fields (used interbally during recursion).
197: (4)               Examples
198: (4)
199: (4)               >>> from numpy.lib import recfunctions as rfn
200: (4)               >>> ndtype = np.dtype([('A', int),
201: (4)                             ('B', [('BA', int),
202: (4)                               ('BB', [('BBA', int), ('BBB', int)]))]))
203: (4)               >>> rfn.get_fieldstructure(ndtype)
204: (4)               ... # XXX: possible regression, order of BBA and BBB is swapped
205: (4)               {'A': [], 'B': [], 'BA': ['B'], 'BB': ['B'], 'BBA': ['B', 'BB'], 'BBB':
206: (4)                 ['B', 'BB']}}
207: (4)
208: (8)           if parents is None:
209: (4)             parents = {}
210: (4)             names = adtype.names
211: (8)             for name in names:
212: (12)               current = adtype[name]
213: (16)               if current.names is not None:
214: (12)                 if lastname:
215: (16)                   parents[name] = [lastname, ]
216: (12)                 else:
217: (16)                   parents[name] = []
218: (12)                   parents.update(get_fieldstructure(current, name, parents))
219: (12)                 else:
220: (16)                   lastparent = [_ for _ in (parents.get(lastname, []) or [])]
221: (12)                   if lastparent:
222: (16)                     lastparent.append(lastname)
223: (12)                   elif lastname:
224: (16)                     lastparent = [lastname, ]
225: (12)                     parents[name] = lastparent or []
226: (4)             return parents
227: (4)           def _izip_fields_flat(iterable):
228: (4)             """
229: (4)               Returns an iterator of concatenated fields from a sequence of arrays,
230: (4)               collapsing any nested structure.
231: (4)
232: (8)               for element in iterable:
233: (12)                 if isinstance(element, np.void):
234: (8)                   yield from _izip_fields_flat(tuple(element))
235: (4)                 else:
236: (8)                   yield element
237: (4)           def _izip_fields(iterable):
238: (4)             """
239: (4)               Returns an iterator of concatenated fields from a sequence of arrays.
240: (4)
241: (8)               for element in iterable:
242: (16)                 if (hasattr(element, '__iter__') and
243: (12)                   not isinstance(element, str)):
244: (8)                     yield from _izip_fields(element)

```

```

243: (8)           elif isinstance(element, np.void) and len(tuple(element)) == 1:
244: (12)             yield from _izip_fields(element)
245: (8)
246: (12)             yield element
247: (0) def _izip_records(seqarrays, fill_value=None, flatten=True):
248: (4)     """
249: (4)         Returns an iterator of concatenated items from a sequence of arrays.
250: (4)         Parameters
251: (4)         -----
252: (4)         seqarrays : sequence of arrays
253: (8)             Sequence of arrays.
254: (4)         fill_value : {None, integer}
255: (8)             Value used to pad shorter iterables.
256: (4)         flatten : {True, False},
257: (8)             Whether to
258: (4)             """
259: (4)         if flatten:
260: (8)             zipfunc = _izip_fields_flat
261: (4)         else:
262: (8)             zipfunc = _izip_fields
263: (4)         for tup in itertools.zip_longest(*seqarrays, fillvalue=fill_value):
264: (8)             yield tuple(zipfunc(tup))
265: (0) def _fix_output(output, usemask=True, asrecarray=False):
266: (4)     """
267: (4)         Private function: return a recarray, a ndarray, a MaskedArray
268: (4)         or a MaskedRecords depending on the input parameters
269: (4)         """
270: (4)         if not isinstance(output, MaskedArray):
271: (8)             usemask = False
272: (4)         if usemask:
273: (8)             if asrecarray:
274: (12)                 output = output.view(MaskedRecords)
275: (4)             else:
276: (8)                 output = ma.filled(output)
277: (8)                 if asrecarray:
278: (12)                     output = output.view(recarray)
279: (4)             return output
280: (0) def _fix_defaults(output, defaults=None):
281: (4)     """
282: (4)         Update the fill_value and masked data of `output`
283: (4)         from the default given in a dictionary defaults.
284: (4)         """
285: (4)         names = output.dtype.names
286: (4)         (data, mask, fill_value) = (output.data, output.mask, output.fill_value)
287: (4)         for (k, v) in (defaults or {}).items():
288: (8)             if k in names:
289: (12)                 fill_value[k] = v
290: (12)                 data[k][mask[k]] = v
291: (4)         return output
292: (0) def _merge_arrays_dispatcher(seqarrays, fill_value=None, flatten=None,
293: (29)                               usemask=None, asrecarray=None):
294: (4)     return seqarrays
295: (0) @array_function_dispatch(_merge_arrays_dispatcher)
296: (0) def merge_arrays(seqarrays, fill_value=-1, flatten=False,
297: (17)                   usemask=False, asrecarray=False):
298: (4)     """
299: (4)         Merge arrays field by field.
300: (4)         Parameters
301: (4)         -----
302: (4)         seqarrays : sequence of ndarrays
303: (8)             Sequence of arrays
304: (4)         fill_value : {float}, optional
305: (8)             Filling value used to pad missing data on the shorter arrays.
306: (4)         flatten : {False, True}, optional
307: (8)             Whether to collapse nested fields.
308: (4)         usemask : {False, True}, optional
309: (8)             Whether to return a masked array or not.
310: (4)         asrecarray : {False, True}, optional
311: (8)             Whether to return a recarray (MaskedRecords) or not.

```

```

312: (4)          Examples
313: (4)
314: (4)
315: (4)
316: (4)
317: (10)
318: (4)
319: (4)
320: (5)
321: (13)
322: (4)
323: (4)
324: (4)
325: (4)
326: (14)
327: (4)
328: (4)
329: (4)          -----
330: (6)          * Without a mask, the missing value will be filled with something,
331: (6)          depending on what its corresponding type:
332: (6)          * ``-1``   for integers
333: (6)          * ``-1.0`` for floating point numbers
334: (6)          * ``'-'``  for characters
335: (6)          * ``'-1``  for strings
336: (4)          * ``True`` for boolean values
337: (4)
338: (4)          * XXX: I just obtained these values empirically
339: (8)
340: (4)
341: (8)
342: (8)
343: (12)
344: (8)
345: (12)
346: (12)
347: (16)
348: (20)
349: (16)
350: (20)
351: (12)
352: (16)
353: (12)
354: (16)
355: (12)
356: (8)
357: (12)
358: (4)
359: (8)
360: (4)
361: (4)
362: (4)
363: (4)
364: (4)
365: (4)
366: (8)
367: (12)
368: (12)
369: (12)
370: (12)
371: (16)
372: (16)
373: (20)
374: (24)
375: (24)
376: (20)
377: (24)
378: (24)
379: (12)
380: (16)

-----
```

**Examples**

```

>>> from numpy.lib import recfunctions as rfn
>>> rfn.merge_arrays((np.array([1, 2]), np.array([10., 20., 30.])))
array([( 1, 10.), ( 2, 20.), (-1, 30.)],
      dtype=[('f0', '<i8'), ('f1', '<f8')])
>>> rfn.merge_arrays((np.array([1, 2], dtype=np.int64),
...                   np.array([10., 20., 30.])), usemask=False)
array([(1, 10.0), (2, 20.0), (-1, 30.0)],
      dtype=[('f0', '<i8'), ('f1', '<f8')])
>>> rfn.merge_arrays((np.array([1, 2]).view([('a', np.int64)]),
...                   np.array([10., 20., 30.])),
...                   usemask=False, asrecarray=True)
rec.array([( 1, 10.), ( 2, 20.), (-1, 30.)],
      dtype=[('a', '<i8'), ('f1', '<f8')])
```

**Notes**

```

-----
```

- \* Without a mask, the missing value will be filled with something, depending on what its corresponding type:
  - \* ``-1`` for integers
  - \* ``-1.0`` for floating point numbers
  - \* ``'-'`` for characters
  - \* ``'-1`` for strings
  - \* ``True`` for boolean values
- \* XXX: I just obtained these values empirically

```

"""
if (len(seqarrays) == 1):
    seqarrays = np.asarray(seqarrays[0])
if isinstance(seqarrays, (ndarray, np.void)):
    seqdtype = seqarrays.dtype
    if seqdtype.names is None:
        seqdtype = np.dtype([(' ', seqdtype)])
    if not flatten or _zip_dtype((seqarrays,), flatten=True) == seqdtype:
        seqarrays = seqarrays.ravel()
        if usemask:
            if asrecarray:
                seqtype = MaskedRecords
            else:
                seqtype = MaskedArray
        elif asrecarray:
            seqtype = recarray
        else:
            seqtype = ndarray
        return seqarrays.view(dtype=seqdtype, type=seqtype)
    else:
        seqarrays = (seqarrays,)
else:
    seqarrays = [np.asarray(_m) for _m in seqarrays]
sizes = tuple(a.size for a in seqarrays)
maxlength = max(sizes)
newdtype = _zip_dtype(seqarrays, flatten=flatten)
seqdata = []
seqmask = []
if usemask:
    for (a, n) in zip(seqarrays, sizes):
        nbmissing = (maxlength - n)
        data = a.ravel().__array__()
        mask = ma.getmaskarray(a).ravel()
        if nbmissing:
            fval = _check_fill_value(fill_value, a.dtype)
            if isinstance(fval, (ndarray, np.void)):
                if len(fval.dtype) == 1:
                    fval = fval.item()[0]
                    fmsk = True
                else:
                    fval = np.array(fval, dtype=a.dtype, ndmin=1)
                    fmsk = np.ones((1,), dtype=mask.dtype)
            else:
                fval = None
```

```

381: (16)                      fmsk = True
382: (12)                      seqdata.append(itertools.chain(data, [fval] * nbmissing))
383: (12)                      seqmask.append(itertools.chain(mask, [fmsk] * nbmissing))
384: (8)                       data = tuple(_izip_records(seqdata, flatten=flatten))
385: (8)                       output = ma.array(np.fromiter(data, dtype=newdtype, count=maxlength),
386:                                         mask=list(_izip_records(seqmask, flatten=flatten)))
387: (8)                       if asrecarray:
388: (12)                         output = output.view(MaskedRecords)
389: (4) else:
390: (8)   for (a, n) in zip(seqarrays, sizes):
391: (12)     nbmissing = (maxlength - n)
392: (12)     data = a.ravel().__array__()
393: (12)     if nbmissing:
394: (16)       fval = _check_fill_value(fill_value, a.dtype)
395: (16)       if isinstance(fval, (ndarray, np.void)):
396: (20)         if len(fval.dtype) == 1:
397: (24)           fval = fval.item()[0]
398: (20)         else:
399: (24)           fval = np.array(fval, dtype=a.dtype, ndmin=1)
400: (12)       else:
401: (16)         fval = None
402: (12)       seqdata.append(itertools.chain(data, [fval] * nbmissing))
403: (8)       output = np.fromiter(tuple(_izip_records(seqdata, flatten=flatten)),
404:                                     dtype=newdtype, count=maxlength)
405: (8)       if asrecarray:
406: (12)         output = output.view(recarray)
407: (4) return output
408: (0) def _drop_fields_dispatcher(base, drop_names, usemask=None, asrecarray=None):
409: (4)   return (base,)
410: (0) @array_function_dispatch(_drop_fields_dispatcher)
411: (0) def drop_fields(base, drop_names, usemask=True, asrecarray=False):
412: (4)   """
413: (4)   Return a new array with fields in `drop_names` dropped.
414: (4)   Nested fields are supported.
415: (4)   .. versionchanged:: 1.18.0
416: (8)     `drop_fields` returns an array with 0 fields if all fields are
dropped,
417: (8)   rather than returning ``None`` as it did previously.
418: (4) Parameters
419: (4) -----
420: (4)   base : array
421: (8)     Input array
422: (4)   drop_names : string or sequence
423: (8)     String or sequence of strings corresponding to the names of the
424: (8)     fields to drop.
425: (4)   usemask : {False, True}, optional
426: (8)     Whether to return a masked array or not.
427: (4)   asrecarray : string or sequence, optional
428: (8)     Whether to return a recarray or a mrecarray (`asrecarray=True`) or
429: (8)     a plain ndarray or masked array with flexible dtype. The default
430: (8)     is False.
431: (4) Examples
432: (4) -----
433: (4) >>> from numpy.lib import recfunctions as rfn
434: (4) >>> a = np.array([(1, (2, 3.0)), (4, (5, 6.0))],
435: (4) ...   dtype=[('a', np.int64), ('b', [('ba', np.double), ('bb',
np.int64)])])
436: (4) >>> rfn.drop_fields(a, 'a')
437: (4) array([(2., 3.), (5., 6.)],
438: (10)   dtype=[('b', [('ba', '<f8'), ('bb', '<i8')])])
439: (4) >>> rfn.drop_fields(a, 'ba')
440: (4) array([(1, (3.,)), (4, (6.,))], dtype=[('a', '<i8'), ('b', [('bb',
'<i8')])])
441: (4) >>> rfn.drop_fields(a, ['ba', 'bb'])
442: (4) array([(1,), (4,)], dtype=[('a', '<i8')])
443: (4) """
444: (4) if _is_string_like(drop_names):
445: (8)   drop_names = [drop_names]
446: (4)

```

```

447: (8)             drop_names = set(drop_names)
448: (4)             def _drop_descr(ndtype, drop_names):
449: (8)                 names = ndtype.names
450: (8)                 newdtype = []
451: (8)                 for name in names:
452: (12)                     current = ndtype[name]
453: (12)                     if name in drop_names:
454: (16)                         continue
455: (12)                     if current.names is not None:
456: (16)                         descr = _drop_descr(current, drop_names)
457: (16)                         if descr:
458: (20)                             newdtype.append((name, descr))
459: (12)                         else:
460: (16)                             newdtype.append((name, current))
461: (8)             return newdtype
462: (4)             newdtype = _drop_descr(base.dtype, drop_names)
463: (4)             output = np.empty(base.shape, dtype=newdtype)
464: (4)             output = recursive_fill_fields(base, output)
465: (4)             return _fix_output(output, usemask=usemask, asrecarray=asrecarray)
466: (0)             def _keep_fields(base, keep_names, usemask=True, asrecarray=False):
467: (4)                 """
468: (4)                 Return a new array keeping only the fields in `keep_names` ,
469: (4)                 and preserving the order of those fields.
470: (4)                 Parameters
471: (4)                 -----
472: (4)                 base : array
473: (8)                     Input array
474: (4)                 keep_names : string or sequence
475: (8)                     String or sequence of strings corresponding to the names of the
476: (8)                     fields to keep. Order of the names will be preserved.
477: (4)                 usemask : {False, True}, optional
478: (8)                     Whether to return a masked array or not.
479: (4)                 asrecarray : string or sequence, optional
480: (8)                     Whether to return a recarray or a mrecarray (`asrecarray=True`) or
481: (8)                     a plain ndarray or masked array with flexible dtype. The default
482: (8)                     is False.
483: (4)                 """
484: (4)                 newdtype = [(n, base.dtype[n]) for n in keep_names]
485: (4)                 output = np.empty(base.shape, dtype=newdtype)
486: (4)                 output = recursive_fill_fields(base, output)
487: (4)                 return _fix_output(output, usemask=usemask, asrecarray=asrecarray)
488: (0)             def _rec_drop_fields_dispatcher(base, drop_names):
489: (4)                 return (base,)
490: (0)             @array_function_dispatch(_rec_drop_fields_dispatcher)
491: (0)             def rec_drop_fields(base, drop_names):
492: (4)                 """
493: (4)                 Returns a new numpy.recarray with fields in `drop_names` dropped.
494: (4)                 """
495: (4)                 return drop_fields(base, drop_names, usemask=False, asrecarray=True)
496: (0)             def _rename_fields_dispatcher(base, namemapper):
497: (4)                 return (base,)
498: (0)             @array_function_dispatch(_rename_fields_dispatcher)
499: (0)             def rename_fields(base, namemapper):
500: (4)                 """
501: (4)                 Rename the fields from a flexible-datatype ndarray or recarray.
502: (4)                 Nested fields are supported.
503: (4)                 Parameters
504: (4)                 -----
505: (4)                 base : ndarray
506: (8)                     Input array whose fields must be modified.
507: (4)                 namemapper : dictionary
508: (8)                     Dictionary mapping old field names to their new version.
509: (4)                 Examples
509: (4)                 -----
511: (4)                 >>> from numpy.lib import recfunctions as rfn
512: (4)                 >>> a = np.array([(1, (2, [3.0, 30.])), (4, (5, [6.0, 60.]))]),
513: (4)                     ...      dtype=[('a', int), ('b', [('ba', float), ('bb', (float, 2))])])
514: (4)                 >>> rfn.rename_fields(a, {'a':'A', 'bb':'BB'})
515: (4)                 array([(1, (2., [ 3., 30.])), (4, (5., [ 6., 60.]))]),
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

516: (10)           dtype=[('A', '<i8'), ('b', [('ba', '<f8'), ('BB', '<f8', (2,))])])
517: (4)
518: (4)           """
519: (8)           def _recursive_rename_fields(ndtype, namemapper):
520: (8)               newdtype = []
521: (12)             for name in ndtype.names:
522: (12)                 newname = namemapper.get(name, name)
523: (12)                 current = ndtype[name]
524: (16)                 if current.names is not None:
525: (20)                     newdtype.append(
526: (20)                         (newname, _recursive_rename_fields(current, namemapper)))
527: (12)                     )
528: (16)                 else:
529: (8)                     newdtype.append((newname, current))
530: (4)             return newdtype
531: (4)             newdtype = _recursive_rename_fields(base.dtype, namemapper)
532: (0)             return base.view(newdtype)
533: (30)           def _append_fields_dispatcher(base, names, data, dtypes=None,
534: (4)                           fill_value=None, usemask=None, asrecarray=None):
535: (4)               yield base
536: (0)               yield from data
537: (0)               @array_function_dispatch(_append_fields_dispatcher)
538: (18)             def append_fields(base, names, data, dtypes=None,
539: (4)                           fill_value=-1, usemask=True, asrecarray=False):
540: (4)               """
541: (4)               Add new fields to an existing array.
542: (4)               The names of the fields are given with the `names` arguments,
543: (4)               the corresponding values with the `data` arguments.
544: (4)               If a single field is appended, `names`, `data` and `dtypes` do not have
545: (4)               to be lists but just values.
546: (4)               Parameters
547: (4)               -----
548: (8)               base : array
549: (4)                   Input array to extend.
550: (8)               names : string, sequence
551: (8)                   String or sequence of strings corresponding to the names
552: (4)                   of the new fields.
553: (8)               data : array or sequence of arrays
554: (8)                   Array or sequence of arrays storing the fields to add to the base.
555: (8)               dtypes : sequence of datatypes, optional
556: (8)                   Datatype or sequence of datatypes.
557: (4)                   If None, the datatypes are estimated from the `data`.
558: (8)               fill_value : {float}, optional
559: (4)                   Filling value used to pad missing data on the shorter arrays.
560: (8)               usemask : {False, True}, optional
561: (4)                   Whether to return a masked array or not.
562: (8)               asrecarray : {False, True}, optional
563: (4)                   Whether to return a recarray (MaskedRecords) or not.
564: (4)
565: (8)           if isinstance(names, (tuple, list)):
566: (12)             if len(names) != len(data):
567: (12)                 msg = "The number of arrays does not match the number of names"
568: (12)                 raise ValueError(msg)
569: (8)             elif isinstance(names, str):
570: (8)                 names = [names, ]
571: (8)                 data = [data, ]
572: (8)             if dtypes is None:
573: (8)                 data = [np.array(a, copy=False, subok=True) for a in data]
574: (8)                 data = [a.view([(name, a.dtype)]) for (name, a) in zip(names, data)]
575: (8)             else:
576: (12)               if not isinstance(dtypes, (tuple, list)):
577: (12)                 dtypes = [dtypes, ]
578: (12)               if len(data) != len(dtypes):
579: (16)                 if len(dtypes) == 1:
580: (16)                     dtypes = dtypes * len(data)
581: (16)                 else:
582: (16)                     msg = "The dtypes argument must be None, a dtype, or a list."
583: (8)                     raise ValueError(msg)
584: (16)             data = [np.array(a, copy=False, subok=True, dtype=d).view([(n, d)])
584: (16)                           for (a, n, d) in zip(data, names, dtypes)]
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

585: (4)           base = merge_arrays(base, usemask=usemask, fill_value=fill_value)
586: (4)           if len(data) > 1:
587: (8)             data = merge_arrays(data, flatten=True, usemask=usemask,
588:                                         fill_value=fill_value)
589: (4)           else:
590: (8)             data = data.pop()
591: (4)           output = ma.masked_all(
592: (8)             max(len(base), len(data)),
593: (8)             dtype=_get_fieldspec(base.dtype) + _get_fieldspec(data.dtype))
594: (4)           output = recursive_fill_fields(base, output)
595: (4)           output = recursive_fill_fields(data, output)
596: (4)           return _fix_output(output, usemask=usemask, asrecarray=asrecarray)
597: (0)           def _rec_append_fields_dispatcher(base, names, data, dtypes=None):
598: (4)             yield base
599: (4)             yield from data
600: (0)           @array_function_dispatch(_rec_append_fields_dispatcher)
601: (0)           def rec_append_fields(base, names, data, dtypes=None):
602: (4)             """
603: (4)             Add new fields to an existing array.
604: (4)             The names of the fields are given with the `names` arguments,
605: (4)             the corresponding values with the `data` arguments.
606: (4)             If a single field is appended, `names`, `data` and `dtypes` do not have
607: (4)             to be lists but just values.
608: (4)             Parameters
609: (4)             -----
610: (4)             base : array
611: (8)               Input array to extend.
612: (4)             names : string, sequence
613: (8)               String or sequence of strings corresponding to the names
614: (8)               of the new fields.
615: (4)             data : array or sequence of arrays
616: (8)               Array or sequence of arrays storing the fields to add to the base.
617: (4)             dtypes : sequence of datatypes, optional
618: (8)               Datatype or sequence of datatypes.
619: (8)               If None, the datatypes are estimated from the `data`.
620: (4)             See Also
621: (4)             -----
622: (4)             append_fields
623: (4)             Returns
624: (4)             -----
625: (4)             appended_array : np.recarray
626: (4)             """
627: (4)             return append_fields(base, names, data=data, dtypes=dtypes,
628: (25)                           asrecarray=True, usemask=False)
629: (0)           def _repack_fields_dispatcher(a, align=None, recurse=None):
630: (4)             return (a,)
631: (0)           @array_function_dispatch(_repack_fields_dispatcher)
632: (0)           def repack_fields(a, align=False, recurse=False):
633: (4)             """
634: (4)             Re-pack the fields of a structured array or dtype in memory.
635: (4)             The memory layout of structured datatypes allows fields at arbitrary
636: (4)             byte offsets. This means the fields can be separated by padding bytes,
637: (4)             their offsets can be non-monotonically increasing, and they can overlap.
638: (4)             This method removes any overlaps and reorders the fields in memory so they
639: (4)             have increasing byte offsets, and adds or removes padding bytes depending
640: (4)             on the `align` option, which behaves like the `align` option to
641: (4)             `numpy.dtype`.
642: (4)             If `align=False`, this method produces a "packed" memory layout in which
643: (4)             each field starts at the byte the previous field ended, and any padding
644: (4)             bytes are removed.
645: (4)             If `align=True`, this method produces an "aligned" memory layout in which
646: (4)             each field's offset is a multiple of its alignment, and the total itemsize
647: (4)             is a multiple of the largest alignment, by adding padding bytes as needed.
648: (4)             Parameters
649: (4)             -----
650: (4)             a : ndarray or dtype
651: (7)               array or dtype for which to repack the fields.
652: (4)             align : boolean
653: (7)               If true, use an "aligned" memory layout, otherwise use a "packed"

```

```

layout.
654: (4)             recurse : boolean
655: (7)             If True, also repack nested structures.
656: (4)
657: (4)
658: (4)             repacked : ndarray or dtype
659: (7)             Copy of `a` with fields repacked, or `a` itself if no repacking was
660: (7)             needed.
661: (4)
662: (4)
663: (4)             Examples
664: (4)
665: (4)             >>> from numpy.lib import recfunctions as rfn
666: (4)             >>> def print_offsets(d):
667: (4)                 ...     print("offsets:", [d.fields[name][1] for name in d.names])
668: (4)                 ...     print("itemsize:", d.itemsize)
669: (4)
670: (4)                 ...
671: (0)                 >>> dt = np.dtype('u1, <i8, <f8', align=True)
672: (4)                 >>> dt
673: (4)                 dtype({'names': ['f0', 'f1', 'f2'], 'formats': ['u1', '<i8', '<f8'], \
674: (4) 'offsets': [0, 8, 16], 'itemsize': 24}, align=True)
675: (4)                 >>> print_offsets(dt)
676: (4)                 offsets: [0, 8, 16]
677: (4)                 itemsize: 24
678: (4)                 >>> packed_dt = rfn.repack_fields(dt)
679: (4)                 >>> packed_dt
680: (4)                 dtype([('f0', 'u1'), ('f1', '<i8'), ('f2', '<f8')])
681: (4)                 >>> print_offsets(packed_dt)
682: (4)                 offsets: [0, 1, 9]
683: (8)                 itemsize: 17
684: (8)
685: (4)             """
686: (8)             if not isinstance(a, np.dtype):
687: (8)                 dt = repack_fields(a.dtype, align=align, recurse=recurse)
688: (8)                 return a.astype(dt, copy=False)
689: (8)
690: (8)
691: (12)             if a.names is None:
692: (8)
693: (12)                 return a
694: (8)
695: (12)             fieldinfo = []
696: (8)
697: (4)             for name in a.names:
698: (4)                 tup = a.fields[name]
699: (4)                 if recurse:
700: (4)                     fmt = repack_fields(tup[0], align=align, recurse=True)
701: (4)                 else:
702: (4)                     fmt = tup[0]
703: (4)                     if len(tup) == 3:
704: (4)                         name = (tup[2], name)
705: (4)                         fieldinfo.append((name, fmt))
706: (8)
707: (8)
708: (12)
709: (16)
710: (12)
711: (8)
712: (4)
713: (4)
714: (8)
715: (8)
716: (8)
717: (8)
718: (12)
719: (8)
720: (12)
721: (12)
    def _get_fields_and_offsets(dt, offset=0):
        """
        Returns a flat list of (dtype, count, offset) tuples of all the
        scalar fields in the dtype "dt", including nested fields, in left
        to right order.
        """
        def count_elem(dt):
            count = 1
            while dt.shape != ():
                for size in dt.shape:
                    count *= size
                dt = dt.base
            return dt, count
        fields = []
        for name in dt.names:
            field = dt.fields[name]
            f_dt, f_offset = field[0], field[1]
            f_dt, n = count_elem(f_dt)
            if f_dt.names is None:
                fields.append((np.dtype((f_dt, (n,))), n, f_offset + offset))
            else:
                subfields = _get_fields_and_offsets(f_dt, f_offset + offset)
                size = f_dt.itemsize

```

```

722: (12)
723: (16)
724: (20)
725: (16)
726: (20)
subfields])
727: (4)
728: (0)
729: (4)
730: (4)
731: (4)
732: (4)
733: (4)
734: (4)
735: (4)
736: (8)
737: (4)
738: (4)
739: (8)
740: (4)
741: (8)
742: (4)
743: (4)
744: (4)
745: (8)
746: (12)
747: (16)
748: (12)
749: (16)
750: (12)
751: (16)
752: (12)
753: (8)
754: (12)
755: (8)
756: (12)
757: (12)
758: (16)
759: (12)
760: (16)
761: (8)
762: (4)
763: (8)
764: (4)
765: (0)
766: (43)
767: (4)
768: (0)
769: (0)
770: (4)
771: (4)
772: (4)
773: (4)
774: (4)
775: (4)
776: (4)
777: (4)
778: (4)
779: (4)
780: (4)
781: (7)
782: (4)
783: (7)
784: (4)
785: (8)
786: (8)
787: (8)
788: (8)
789: (8)

                    for i in range(n):
                        if i == 0:
                            fields.extend(subfields)
                        else:
                            fields.extend([(d, c, o + i*size) for d, c, o in
subfields])
727: (4)
728: (0)
729: (4)
730: (4)
731: (4)
732: (4)
733: (4)
734: (4)
735: (4)
736: (8)
737: (4)
738: (4)
739: (8)
740: (4)
741: (8)
742: (4)
743: (4)
744: (4)
745: (8)
746: (12)
747: (16)
748: (12)
749: (16)
750: (12)
751: (16)
752: (12)
753: (8)
754: (12)
755: (8)
756: (12)
757: (12)
758: (16)
759: (12)
760: (16)
761: (8)
762: (4)
763: (8)
764: (4)
765: (0)
766: (43)
767: (4)
768: (0)
769: (0)
770: (4)
771: (4)
772: (4)
773: (4)
774: (4)
775: (4)
776: (4)
777: (4)
778: (4)
779: (4)
780: (4)
781: (7)
782: (4)
783: (7)
784: (4)
785: (8)
786: (8)
787: (8)
788: (8)
789: (8)

                    return fields
def _common_stride(offsets, counts, itemsize):
    """
    Returns the stride between the fields, or None if the stride is not
    constant. The values in "counts" designate the lengths of
    subarrays. Subarrays are treated as many contiguous fields, with
    always positive stride.
    """
    if len(offsets) <= 1:
        return itemsize
    negative = offsets[1] < offsets[0] # negative stride
    if negative:
        it = zip(reversed(offsets), reversed(counts))
    else:
        it = zip(offsets, counts)
    prev_offset = None
    stride = None
    for offset, count in it:
        if count != 1: # subarray: always c-contiguous
            if negative:
                return None # subarrays can never have a negative stride
            if stride is None:
                stride = itemsize
            if stride != itemsize:
                return None
            end_offset = offset + (count - 1) * itemsize
        else:
            end_offset = offset
        if prev_offset is not None:
            new_stride = offset - prev_offset
            if stride is None:
                stride = new_stride
            if stride != new_stride:
                return None
            prev_offset = end_offset
    if negative:
        return -stride
    return stride
def _structured_to_unstructured_dispatcher(arr, dtype=None, copy=None,
                                         casting=None):
    return (arr,)
@array_function_dispatch(_structured_to_unstructured_dispatcher)
def structured_to_unstructured(arr, dtype=None, copy=False, casting='unsafe'):
    """
    Converts an n-D structured array into an (n+1)-D unstructured array.
    The new array will have a new last dimension equal in size to the
    number of field-elements of the input array. If not supplied, the output
    datatype is determined from the numpy type promotion rules applied to all
    the field datatypes.
    Nested fields, as well as each element of any subarray fields, all count
    as a single field-elements.
    Parameters
    -----
    arr : ndarray
        Structured array or dtype to convert. Cannot contain object datatype.
    dtype : dtype, optional
        The dtype of the output unstructured array.
    copy : bool, optional
        If true, always return a copy. If false, a view is returned if
        possible, such as when the `dtype` and strides of the fields are
        suitable and the array subtype is one of `np.ndarray`, `np.recarray`
        or `np.memmap`.
    .. versionchanged:: 1.25.0
    """
    pass

```

```

790: (12)                                A view can now be returned if the fields are separated by a
791: (12)                                uniform stride.
792: (4)      casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
793: (8)          See casting argument of `numpy.ndarray.astype` . Controls what kind of
794: (8)          data casting may occur.
795: (4)      Returns
796: (4)      -----
797: (4)      unstructured : ndarray
798: (7)          Unstructured array with one more dimension.
799: (4)      Examples
800: (4)      -----
801: (4)      >>> from numpy.lib import recfunctions as rfn
802: (4)      >>> a = np.zeros(4, dtype=[('a', 'i4'), ('b', 'f4,u2'), ('c', 'f4', 2)])
803: (4)      >>> a
804: (4)      array([(0, (0., 0), [0., 0.]), (0, (0., 0), [0., 0.]),
805: (11)          (0, (0., 0), [0., 0.]), (0, (0., 0), [0., 0.])],
806: (10)          dtype=[('a', '<i4'), ('b', [('f0', '<f4'), ('f1', '<u2')]), ('c',
807: ('<f4', (2,)))])
808: (4)      >>> rfn.structured_to_unstructured(a)
809: (11)      array([[0., 0., 0., 0., 0.],
810: (11)          [0., 0., 0., 0., 0.],
811: (11)          [0., 0., 0., 0., 0.],
812: (4)          [0., 0., 0., 0., 0.]])
813: (4)      >>> b = np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
814: (4)          ...           dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
815: (4)      >>> np.mean(rfn.structured_to_unstructured(b[['x', 'z']]), axis=-1)
816: (4)      array([ 3. ,  5.5,  9. , 11. ])
817: (4)      """
818: (8)      if arr.dtype.names is None:
819: (4)          raise ValueError('arr must be a structured array')
820: (4)      fields = _get_fields_and_offsets(arr.dtype)
821: (4)      n_fields = len(fields)
822: (8)      if n_fields == 0 and dtype is None:
823: (4)          raise ValueError("arr has no fields. Unable to guess dtype")
824: (8)      elif n_fields == 0:
825: (4)          raise NotImplementedError("arr with no fields is not supported")
826: (4)      dts, counts, offsets = zip(*fields)
827: (4)      names = ['f{}'.format(n) for n in range(n_fields)]
828: (8)      if dtype is None:
829: (4)          out_dtype = np.result_type(*[dt.base for dt in dts])
830: (8)      else:
831: (4)          out_dtype = np.dtype(dtype)
832: (33)      flattened_fields = np.dtype({'names': names,
833: (33)          'formats': dts,
834: (33)          'offsets': offsets,
835: (4)          'itemsize': arr.dtype.itemsize})
836: (4)      arr = arr.view(flattened_fields)
837: (4)      can_view = type(arr) in (np.ndarray, np.recarray, np.memmap)
838: (8)      if (not copy) and can_view and all(dt.base == out_dtype for dt in dts):
839: (8)          common_stride = _common_stride(offsets, counts, out_dtype.itemsize)
840: (12)          if common_stride is not None:
841: (12)              wrap = arr.__array_wrap__
842: (12)              new_shape = arr.shape + (sum(counts), out_dtype.itemsize)
843: (12)              new_strides = arr.strides + (abs(common_stride), 1)
844: (12)              arr = arr[..., np.newaxis].view(np.uint8) # view as bytes
845: (12)              arr = arr[..., min(offsets):] # remove the leading unused data
846: (50)              arr = np.lib.stride_tricks.as_strided(arr,
847: (50)                  new_shape,
848: (50)                  new_strides,
849: (12)                  subok=True)
850: (12)          arr = arr.view(out_dtype)[..., 0]
851: (16)          if common_stride < 0:
852: (12)              arr = arr[..., ::-1] # reverse, if the stride was negative
853: (16)          if type(arr) is not type(wrap.__self__):
854: (12)              arr = wrap(arr)
855: (4)          return arr
856: (30)      packed_fields = np.dtype({'names': names,
857: (30)          'formats': [(out_dtype, dt.shape) for dt in
dts]}))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

857: (4)
858: (4)
859: (0)
860: (43)
casting=None):
861: (4)
862: (0)
863: (0)
864: (31)
865: (4)
866: (4)
867: (4)
868: (4)
869: (4)
but
870: (4)
871: (4)
872: (4)
873: (4)
874: (4)
875: (4)
876: (4)
877: (7)
878: (4)
879: (7)
880: (4)
881: (7)
882: (7)
883: (4)
884: (7)
885: (4)
886: (8)
887: (8)
888: (8)
889: (4)
890: (8)
891: (8)
892: (4)
893: (4)
894: (4)
895: (7)
896: (4)
897: (4)
898: (4)
899: (4)
900: (4)
901: (4)
902: (4)
903: (11)
904: (11)
905: (11)
906: (4)
907: (4)
908: (11)
909: (10)
'<f4', (2,))])
910: (4)
911: (4)
912: (8)
913: (4)
914: (4)
915: (8)
916: (4)
917: (8)
918: (12)
919: (8)
920: (8)
921: (8)
922: (4)

        arr = arr.astype(packed_fields, copy=copy, casting=casting)
        return arr.view(out_dtype, (sum(counts),)))
def _unstructured_to_structured_dispatcher(arr, dtype=None, names=None,
                                         align=None, copy=None,
                                         return (arr,))
@array_function_dispatch(_unstructured_to_structured_dispatcher)
def unstructured_to_structured(arr, dtype=None, names=None, align=False,
                               copy=False, casting='unsafe'):
    """
    Converts an n-D unstructured array into an (n-1)-D structured array.
    The last dimension of the input array is converted into a structure, with
    number of field-elements equal to the size of the last dimension of the
    input array. By default all output fields have the input array's dtype,
    an output structured dtype with an equal number of fields-elements can be
    supplied instead.
    Nested fields, as well as each element of any subarray fields, all count
    towards the number of field-elements.
    Parameters
    -----
    arr : ndarray
        Unstructured array or dtype to convert.
    dtype : dtype, optional
        The structured dtype of the output array
    names : list of strings, optional
        If dtype is not supplied, this specifies the field names for the output
        dtype, in order. The field dtypes will be the same as the input array.
    align : boolean, optional
        Whether to create an aligned memory layout.
    copy : bool, optional
        See copy argument to `numpy.ndarray.astype`. If true, always return a
        copy. If false, and `dtype` requirements are satisfied, a view is
        returned.
    casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
        See casting argument of `numpy.ndarray.astype`. Controls what kind of
        data casting may occur.
    Returns
    -----
    structured : ndarray
        Structured array with fewer dimensions.
    Examples
    -----
    >>> from numpy.lib import recfunctions as rfn
    >>> dt = np.dtype([('a', 'i4'), ('b', 'f4,u2'), ('c', 'f4', 2)])
    >>> a = np.arange(20).reshape((4,5))
    >>> a
    array([[ 0,  1,  2,  3,  4],
           [ 5,  6,  7,  8,  9],
           [10, 11, 12, 13, 14],
           [15, 16, 17, 18, 19]])
    >>> rfn.unstructured_to_structured(a, dt)
    array([( 0, ( 1.,  2), [ 3.,  4.]), ( 5, ( 6.,  7), [ 8.,  9.]),
           (10, (11., 12), [13., 14.]), (15, (16., 17), [18., 19.])],
          dtype=[('a', '<i4'), ('b', [('f0', '<f4'), ('f1', '<u2')]), ('c',
'<f4', (2,))])
    """
    if arr.shape == ():
        raise ValueError('arr must have at least one dimension')
    n_elem = arr.shape[-1]
    if n_elem == 0:
        raise NotImplementedError("last axis with size 0 is not supported")
    if dtype is None:
        if names is None:
            names = ['f{}'.format(n) for n in range(n_elem)]
        out_dtype = np.dtype([(n, arr.dtype) for n in names], align=align)
        fields = _get_fields_and_offsets(out_dtype)
        dts, counts, offsets = zip(*fields)
    else:
        if names is None:
            names = ['f{}'.format(n) for n in range(n_elem)]
        out_dtype = np.dtype([(n, arr.dtype) for n in names], align=align)
        fields = _get_fields_and_offsets(out_dtype)
        dts, counts, offsets = zip(*fields)

```

```

923: (8)             if names is not None:
924: (12)            raise ValueError("don't supply both dtype and names")
925: (8)            dtype = np.dtype(dtype)
926: (8)            fields = _get_fields_and_offsets(dtype)
927: (8)            if len(fields) == 0:
928: (12)              dts, counts, offsets = [], [], []
929: (8)            else:
930: (12)              dts, counts, offsets = zip(*fields)
931: (8)            if n_elem != sum(counts):
932: (12)              raise ValueError('The length of the last dimension of arr must '
933: (29)                            'be equal to the number of fields in dtype')
934: (8)            out_dtype = dtype
935: (8)            if align and not out_dtype.isalignedstruct:
936: (12)              raise ValueError("align was True but dtype is not aligned")
937: (4)            names = ['f{}'.format(n) for n in range(len(fields))]
938: (4)            packed_fields = np.dtype({'names': names,
939: (30)                          'formats': [(arr.dtype, dt.shape) for dt in
dts]})

940: (4)            arr = np.ascontiguousarray(arr).view(packed_fields)
941: (4)            flattened_fields = np.dtype({'names': names,
942: (33)                          'formats': dts,
943: (33)                          'offsets': offsets,
944: (33)                          'itemsize': out_dtype.itemsize})
945: (4)            arr = arr.astype(flattened_fields, copy=copy, casting=casting)
946: (4)            return arr.view(out_dtype)[..., 0]
def _apply_along_fields_dispatcher(func, arr):
947: (0)            return (arr,)
948: (4)        @array_function_dispatch(_apply_along_fields_dispatcher)
949: (0)        def apply_along_fields(func, arr):
950: (0)            """
951: (4)            Apply function 'func' as a reduction across fields of a structured array.
952: (4)            This is similar to `apply_along_axis`, but treats the fields of a
953: (4)            structured array as an extra axis. The fields are all first cast to a
954: (4)            common type following the type-promotion rules from `numpy.result_type`
955: (4)            applied to the field's dtypes.
956: (4)            Parameters
957: (4)            -----
958: (4)            func : function
959: (4)            Function to apply on the "field" dimension. This function must
960: (7)            support an `axis` argument, like np.mean, np.sum, etc.
961: (7)        arr : ndarray
962: (4)            Structured array for which to apply func.
963: (7)        Returns
964: (4)        -----
965: (4)        out : ndarray
966: (4)            Result of the reduction operation
967: (7)        Examples
968: (4)        -----
969: (4)        >>> from numpy.lib import recfunctions as rfn
970: (4)        >>> b = np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
971: (4)                      dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
972: (4)        >>> rfn.apply_along_fields(np.mean, b)
973: (4)        array([ 2.66666667,  5.33333333,  8.66666667, 11.        ])
974: (4)        >>> rfn.apply_along_fields(np.mean, b[['x', 'z']])
975: (4)        array([ 3. ,  5.5,  9. , 11. ])
976: (4)        """
977: (4)
978: (4)        if arr.dtype.names is None:
979: (8)            raise ValueError('arr must be a structured array')
980: (4)        uarr = structured_to_unstructured(arr)
981: (4)        return func(uarr, axis=-1)
def _assign_fields_by_name_dispatcher(dst, src, zero_unassigned=None):
982: (0)            return dst, src
983: (4)        @array_function_dispatch(_assign_fields_by_name_dispatcher)
984: (0)        def assign_fields_by_name(dst, src, zero_unassigned=True):
985: (0)            """
986: (4)            Assigns values from one structured array to another by field name.
987: (4)            Normally in numpy >= 1.14, assignment of one structured array to another
988: (4)            copies fields "by position", meaning that the first field from the src is
989: (4)            copied to the first field of the dst, and so on, regardless of field name.
990: (4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

991: (4) This function instead copies "by field name", such that fields in the dst
992: (4) are assigned from the identically named field in the src. This applies
993: (4) recursively for nested structures. This is how structure assignment worked
994: (4) in numpy >= 1.6 to <= 1.13.
995: (4) Parameters
996: (4) -----
997: (4) dst : ndarray
998: (4) src : ndarray
999: (8) The source and destination arrays during assignment.
1000: (4) zero_unassigned : bool, optional
1001: (8) If True, fields in the dst for which there was no matching
1002: (8) field in the src are filled with the value 0 (zero). This
1003: (8) was the behavior of numpy <= 1.13. If False, those fields
1004: (8) are not modified.
1005: (4) """
1006: (4) if dst.dtype.names is None:
1007: (8)     dst[...] = src
1008: (8)     return
1009: (4) for name in dst.dtype.names:
1010: (8)     if name not in src.dtype.names:
1011: (12)         if zero_unassigned:
1012: (16)             dst[name] = 0
1013: (8)         else:
1014: (12)             assign_fields_by_name(dst[name], src[name],
1015: (34)                         zero_unassigned)
1016: (0) def _require_fields_dispatcher(array, required_dtype):
1017: (4)     return (array,)
1018: (0) @array_function_dispatch(_require_fields_dispatcher)
1019: (0) def require_fields(array, required_dtype):
1020: (4) """
1021: (4) Casts a structured array to a new dtype using assignment by field-name.
1022: (4) This function assigns from the old to the new array by name, so the
1023: (4) value of a field in the output array is the value of the field with the
1024: (4) same name in the source array. This has the effect of creating a new
1025: (4) ndarray containing only the fields "required" by the required_dtype.
1026: (4) If a field name in the required_dtype does not exist in the
1027: (4) input array, that field is created and set to 0 in the output array.
1028: (4) Parameters
1029: (4) -----
1030: (4) a : ndarray
1031: (7)     array to cast
1032: (4) required_dtype : dtype
1033: (7)     datatype for output array
1034: (4) Returns
1035: (4) -----
1036: (4) out : ndarray
1037: (8)     array with the new dtype, with field values copied from the fields in
1038: (8)     the input array with the same name
1039: (4) Examples
1040: (4) -----
1041: (4) >>> from numpy.lib import recfunctions as rfn
1042: (4) >>> a = np.ones(4, dtype=[('a', 'i4'), ('b', 'f8'), ('c', 'u1')])
1043: (4) >>> rfn.require_fields(a, [('b', 'f4'), ('c', 'u1')])
1044: (4) array([(1., 1), (1., 1), (1., 1), (1., 1)],
1045: (6)     dtype=[('b', '<f4'), ('c', 'u1')])
1046: (4) >>> rfn.require_fields(a, [('b', 'f4'), ('newf', 'u1')])
1047: (4) array([(1., 0), (1., 0), (1., 0), (1., 0)],
1048: (6)     dtype=[('b', '<f4'), ('newf', 'u1')])
1049: (4) """
1050: (4) out = np.empty(array.shape, dtype=required_dtype)
1051: (4) assign_fields_by_name(out, array)
1052: (4) return out
1053: (0) def _stack_arrays_dispatcher(arrays, defaults=None, usemask=None,
1054: (29)                     asrecarray=None, autoconvert=None):
1055: (4)     return arrays
1056: (0) @array_function_dispatch(_stack_arrays_dispatcher)
1057: (0) def stack_arrays(arrays, defaults=None, usemask=True, asrecarray=False,
1058: (17)                 autoconvert=False):
1059: (4) """

```

```

1060: (4)           Superposes arrays fields by fields
1061: (4)           Parameters
1062: (4)
1063: (4)           arrays : array or sequence
1064: (8)             Sequence of input arrays.
1065: (4)           defaults : dictionary, optional
1066: (8)             Dictionary mapping field names to the corresponding default values.
1067: (4)           usemask : {True, False}, optional
1068: (8)             Whether to return a MaskedArray (or MaskedRecords if `usemask==True`)
1069: (8)               `asrecarray==True` ) or a ndarray.
1070: (4)           asrecarray : {False, True}, optional
1071: (8)             Whether to return a recarray (or MaskedRecords if `usemask==True`)
1072: (8)               or just a flexible-type ndarray.
1073: (4)           autoconvert : {False, True}, optional
1074: (8)             Whether automatically cast the type of the field to the maximum.
1075: (4)           Examples
1076: (4)
1077: (4)             >>> from numpy.lib import recfunctions as rfn
1078: (4)             >>> x = np.array([1, 2,])
1079: (4)             >>> rfn.stack_arrays(x) is x
1080: (4)             True
1081: (4)             >>> z = np.array([('A', 1), ('B', 2)], dtype=[('A', '|S3'), ('B', float)])
1082: (4)             >>> zz = np.array([(a, 10., 100.), (b, 20., 200.), (c, 30., 300.)],
1083: (4)               ...   dtype=[('A', '|S3'), ('B', np.double), ('C', np.double)])
1084: (4)             >>> test = rfn.stack_arrays((z,zz))
1085: (4)             >>> test
1086: (4)             masked_array(data=[(b'A', 1.0, --), (b'B', 2.0, --), (b'a', 10.0, 100.0),
1087: (23)                   (b'b', 20.0, 200.0), (b'c', 30.0, 300.0)],
1088: (17)                 mask=[(False, False, True), (False, False, True),
1089: (23)                   (False, False, False), (False, False, False),
1090: (23)                   (False, False, False)],
1091: (11)                 fill_value=(b'N/A', 1.e+20, 1.e+20),
1092: (16)                   dtype=[('A', 'S3'), ('B', '<f8'), ('C', '<f8')])
1093: (4)
1094: (4)             """
1095: (8)             if isinstance(arrays, ndarray):
1096: (4)                 return arrays
1097: (8)             elif len(arrays) == 1:
1098: (4)                 return arrays[0]
1099: (4)             seqarrays = [np.asanyarray(a).ravel() for a in arrays]
1100: (4)             nrecords = [len(a) for a in seqarrays]
1101: (4)             ndtype = [a.dtype for a in seqarrays]
1102: (4)             fldnames = [d.names for d in ndtype]
1103: (4)             dtype_1 = ndtype[0]
1104: (4)             newdescr = _get_fieldspec(dtype_1)
1105: (4)             names = [n for n, d in newdescr]
1106: (4)             for dtype_n in ndtype[1:]:
1107: (8)                 for fname, fdtype in _get_fieldspec(dtype_n):
1108: (12)                     if fname not in names:
1109: (16)                         newdescr.append((fname, fdtype))
1110: (16)                         names.append(fname)
1111: (12)                     else:
1112: (16)                         nameidx = names.index(fname)
1113: (16)                         _, cdtype = newdescr[nameidx]
1114: (16)                         if autoconvert:
1115: (20)                             newdescr[nameidx] = (fname, max(fdtype, cdtype))
1116: (16)                         elif fdtype != cdtype:
1117: (20)                             raise TypeError("Incompatible type '%s' <> '%s'" %
1118: (36)                               (cdtype, fdtype))
1119: (4)             if len(newdescr) == 1:
1120: (8)                 output = ma.concatenate(seqarrays)
1121: (4)             else:
1122: (8)                 output = ma.masked_all((np.sum(nrecords),), newdescr)
1123: (8)                 offset = np.cumsum(np.r_[0, nrecords])
1124: (8)                 seen = []
1125: (8)                 for (a, n, i, j) in zip(seqarrays, fldnames, offset[:-1], offset[1:]):
1126: (12)                     names = a.dtype.names
1127: (12)                     if names is None:
1128: (16)                         output['f%i' % len(seen)][i:j] = a

```

```

1129: (16)
1130: (20)
1131: (20)
1132: (24)
1133: (4)
1134: (23)
1135: (0)
1136: (8)
1137: (4)
1138: (0)
1139: (0)
1140: (4)
1141: (4)
1142: (4)
1143: (4)
1144: (4)
1145: (8)
1146: (4)
1147: (8)
1148: (8)
1149: (4)
1150: (8)
1151: (4)
1152: (8)
1153: (4)
1154: (4)
1155: (4)
1156: (4)
1157: (4)
1158: (4)
1159: (4)
1160: (4)
1161: (17)
1162: (11)
1163: (16)
1164: (4)
1165: (4)
1166: (4)
1167: (4)
1168: (4)
1169: (8)
1170: (12)
1171: (8)
1172: (4)
1173: (4)
1174: (4)
1175: (4)
1176: (4)
1177: (8)
1178: (8)
1179: (4)
1180: (4)
1181: (4)
1182: (4)
1183: (8)
1184: (4)
1185: (8)
1186: (0)
1187: (8)
1188: (8)
1189: (4)
1190: (0)
1191: (0)
1192: (12)
1193: (4)
1194: (4)
1195: (4)
1196: (4)
1197: (4)

        for name in n:
            output[name][i:j] = a[name]
            if name not in seen:
                seen.append(name)
        return _fix_output(_fix_defaults(output, defaults),
                           usemask=usemask, asrecarray=asrecarray)
def _find_duplicates_dispatcher(
    a, key=None, ignoremask=None, return_index=None):
    return (a,)
@array_function_dispatch(_find_duplicates_dispatcher)
def find_duplicates(a, key=None, ignoremask=True, return_index=False):
    """
        Find the duplicates in a structured array along a given key
    Parameters
    -----
    a : array-like
        Input array
    key : {string, None}, optional
        Name of the fields along which to check the duplicates.
        If None, the search is performed by records
    ignoremask : {True, False}, optional
        Whether masked data should be discarded or considered as duplicates.
    return_index : {False, True}, optional
        Whether to return the indices of the duplicated values.
    Examples
    -----
    >>> from numpy.lib import recfunctions as rfn
    >>> ndtype = [('a', int)]
    >>> a = np.ma.array([1, 1, 1, 2, 2, 3, 3],
    ...                  mask=[0, 0, 1, 0, 0, 0, 1]).view(ndtype)
    >>> rfn.find_duplicates(a, ignoremask=True, return_index=True)
    (masked_array(data=[(1,), (1,), (2,), (2,)],
                  mask=[(False,), (False,), (False,), (False,)],
                  fill_value=(999999,)),
     dtype=[('a', '<i8')]), array([0, 1, 3, 4]))
    """
    a = np.asarray(a).ravel()
    fields = get_fieldstructure(a.dtype)
    base = a
    if key:
        for f in fields[key]:
            base = base[f]
        base = base[key]
    sortidx = base.argsort()
    sortedbase = base[sortidx]
    sorteddata = sortedbase.filled()
    flag = (sorteddata[:-1] == sorteddata[1:])
    if ignoremask:
        sortedmask = sortedbase.recordmask
        flag[sortedmask[1:]] = False
    flag = np.concatenate(([False], flag))
    flag[:-1] = flag[:-1] + flag[1:]
    duplicates = a[sortidx][flag]
    if return_index:
        return (duplicates, sortidx[flag])
    else:
        return duplicates
def _join_by_dispatcher(
    key, r1, r2, jointype=None, r1postfix=None, r2postfix=None,
    defaults=None, usemask=None, asrecarray=None):
    return (r1, r2)
@array_function_dispatch(_join_by_dispatcher)
def join_by(key, r1, r2, jointype='inner', r1postfix='1', r2postfix='2',
            defaults=None, usemask=True, asrecarray=False):
    """
        Join arrays `r1` and `r2` on key `key`.
        The key should be either a string or a sequence of string corresponding
        to the fields used to join the array. An exception is raised if the
        `key` field cannot be found in the two input arrays. Neither `r1` nor

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1198: (4)           `r2` should have any duplicates along `key`: the presence of duplicates
1199: (4)           will make the output quite unreliable. Note that duplicates are not
1200: (4)           looked for by the algorithm.
1201: (4)           Parameters
1202: (4)           -----
1203: (4)           key : {string, sequence}
1204: (8)             A string or a sequence of strings corresponding to the fields used
1205: (8)             for comparison.
1206: (4)           r1, r2 : arrays
1207: (8)             Structured arrays.
1208: (4)           jointype : {'inner', 'outer', 'leftouter'}, optional
1209: (8)             If 'inner', returns the elements common to both r1 and r2.
1210: (8)             If 'outer', returns the common elements as well as the elements of
1211: (8)               r1 not in r2 and the elements of not in r2.
1212: (8)             If 'leftouter', returns the common elements and the elements of r1
1213: (8)             not in r2.
1214: (4)           r1postfix : string, optional
1215: (8)             String appended to the names of the fields of r1 that are present
1216: (8)             in r2 but absent of the key.
1217: (4)           r2postfix : string, optional
1218: (8)             String appended to the names of the fields of r2 that are present
1219: (8)             in r1 but absent of the key.
1220: (4)           defaults : {dictionary}, optional
1221: (8)             Dictionary mapping field names to the corresponding default values.
1222: (4)           usemask : {True, False}, optional
1223: (8)             Whether to return a MaskedArray (or MaskedRecords is
1224: (8)               `asrecarray==True`) or a ndarray.
1225: (4)           asrecarray : {False, True}, optional
1226: (8)             Whether to return a recarray (or MaskedRecords if `usemask==True`)
1227: (8)             or just a flexible-type ndarray.
1228: (4)           Notes
1229: (4)           -----
1230: (4)             * The output is sorted along the key.
1231: (4)             * A temporary array is formed by dropping the fields not in the key for
1232: (6)               the two arrays and concatenating the result. This array is then
1233: (6)               sorted, and the common entries selected. The output is constructed by
1234: (6)               filling the fields with the selected entries. Matching is not
1235: (6)               preserved if there are some duplicates...
1236: (4)
1237: (4)           """
1238: (8)           if jointype not in ('inner', 'outer', 'leftouter'):
1239: (16)             raise ValueError(
1240: (16)               "The 'jointype' argument should be in 'inner', "
1241: (16)               "'outer' or 'leftouter' (got '%s' instead)" % jointype
1242: (4)
1243: (8)           if isinstance(key, str):
1244: (4)             key = (key,)
1245: (8)           if len(set(key)) != len(key):
1246: (8)             dup = next(x for n,x in enumerate(key) if x in key[n+1:])
1247: (8)             raise ValueError("duplicate join key %r" % dup)
1248: (4)           for name in key:
1249: (8)             if name not in r1.dtype.names:
1250: (8)               raise ValueError('r1 does not have key field %r' % name)
1251: (8)             if name not in r2.dtype.names:
1252: (8)               raise ValueError('r2 does not have key field %r' % name)
1253: (4)           r1 = r1.ravel()
1254: (4)           r2 = r2.ravel()
1255: (4)           nb1 = len(r1)
1256: (4)           (r1names, r2names) = (r1.dtype.names, r2.dtype.names)
1257: (4)           collisions = (set(r1names) & set(r2names)) - set(key)
1258: (8)           if collisions and not (r1postfix or r2postfix):
1259: (8)             msg = "r1 and r2 contain common names, r1postfix and r2postfix "
1260: (8)             msg += "can't both be empty"
1261: (8)             raise ValueError(msg)
1262: (4)           key1 = [ n for n in r1names if n in key ]
1263: (4)           r1k = _keep_fields(r1, key1)
1264: (4)           r2k = _keep_fields(r2, key1)
1265: (4)           aux = ma.concatenate((r1k, r2k))
1266: (4)           idx_sort = aux.argsort(order=key)

```

```

1267: (4)         flag_in = ma.concatenate(([False], aux[1:] == aux[:-1]))
1268: (4)         flag_in[:-1] = flag_in[1:] + flag_in[:-1]
1269: (4)         idx_in = idx_sort[flag_in]
1270: (4)         idx_1 = idx_in[(idx_in < nb1)]
1271: (4)         idx_2 = idx_in[(idx_in >= nb1)] - nb1
1272: (4)         (r1cmn, r2cmn) = (len(idx_1), len(idx_2))
1273: (4)         if jointype == 'inner':
1274: (8)             (r1spc, r2spc) = (0, 0)
1275: (4)         elif jointype == 'outer':
1276: (8)             idx_out = idx_sort[~flag_in]
1277: (8)             idx_1 = np.concatenate((idx_1, idx_out[(idx_out < nb1)]))
1278: (8)             idx_2 = np.concatenate((idx_2, idx_out[(idx_out >= nb1)] - nb1))
1279: (8)             (r1spc, r2spc) = (len(idx_1) - r1cmn, len(idx_2) - r2cmn)
1280: (4)         elif jointype == 'leftouter':
1281: (8)             idx_out = idx_sort[~flag_in]
1282: (8)             idx_1 = np.concatenate((idx_1, idx_out[(idx_out < nb1)]))
1283: (8)             (r1spc, r2spc) = (len(idx_1) - r1cmn, 0)
1284: (4)             (s1, s2) = (r1[idx_1], r2[idx_2])
1285: (4)             ndtype = _get_fieldspec(r1k.dtype)
1286: (4)             for fname, fdtype in _get_fieldspec(r1.dtype):
1287: (8)                 if fname not in key:
1288: (12)                     ndtype.append((fname, fdtype))
1289: (4)             for fname, fdtype in _get_fieldspec(r2.dtype):
1290: (8)                 names = list(name for name, dtype in ndtype)
1291: (8)                 try:
1292: (12)                     nameidx = names.index(fname)
1293: (8)                 except ValueError:
1294: (12)                     ndtype.append((fname, fdtype))
1295: (8)                 else:
1296: (12)                     _, cdtype = ndtype[nameidx]
1297: (12)                     if fname in key:
1298: (16)                         ndtype[nameidx] = (fname, max(fdtype, cdtype))
1299: (12)
1300: (16)
1301: (20)
1302: (20)
1303: (16)
1304: (4)
1305: (4)
1306: (4)
1307: (4)
1308: (4)
1309: (8)
1310: (8)
key):
1311: (12)
1312: (8)
1313: (8)
1314: (8)
1315: (12)
1316: (4)
1317: (8)
1318: (8)
key):
1319: (12)
1320: (8)
1321: (8)
1322: (8)
1323: (12)
1324: (4)
1325: (4)
1326: (4)
1327: (0)
1328: (8)
1329: (8)
1330: (4)
1331: (0)
1332: (0)
1333: (13)

        if jointype in ('outer', 'leftouter'):
            current[cmn:cmn + r1spc] = selected[r1cmn:]
        for f in r2names:
            selected = s2[f]
            if f not in names or (f in r1names and not r1postfix and f not in
key):
                f += r2postfix
            current = output[f]
            current[:r1cmn] = selected[:r1cmn]
            if jointype in ('outer', 'leftouter'):
                current[cmn:cmn + r1spc] = selected[r1cmn:]
        for f in r2names:
            selected = s2[f]
            if f not in names or (f in r1names and not r1postfix and f not in
key):
                f += r2postfix
            current = output[f]
            current[:r2cmn] = selected[:r2cmn]
            if (jointype == 'outer') and r2spc:
                current[-r2spc:] = selected[r2cmn:]
        output.sort(order=key)
        kwargs = dict(usemask=usemask, asrecarray=asrecarray)
        return _fix_output(_fix_defaults(output, defaults), **kwargs)
def _rec_join_dispatcher(
    key, r1, r2, jointype=None, r1postfix=None, r2postfix=None,
    defaults=None):
    return (r1, r2)
@array_function_dispatch(_rec_join_dispatcher)
def rec_join(key, r1, r2, jointype='inner', r1postfix='1', r2postfix='2',
    defaults=None):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1334: (4)      """
1335: (4)      Join arrays `r1` and `r2` on keys.
1336: (4)      Alternative to join_by, that always returns a np.recarray.
1337: (4)      See Also
1338: (4)      -----
1339: (4)      join_by : equivalent function
1340: (4)      """
1341: (4)      kwargs = dict(jointype=jointype, r1postfix=r1postfix, r2postfix=r2postfix,
1342: (18)          defaults=defaults, usemask=False, asrecarray=True)
1343: (4)      return join_by(key, r1, r2, **kwargs)
-----
```

## File 216 - scimath.py:

```

1: (0)      """
2: (0)      Wrapper functions to more user-friendly calling of certain math functions
3: (0)      whose output data-type is different than the input data-type in certain
4: (0)      domains of the input.
5: (0)      For example, for functions like `log` with branch cuts, the versions in this
6: (0)      module provide the mathematically valid answers in the complex plane::
7: (2)      >>> import math
8: (2)      >>> np.emath.log(-math.exp(1)) == (1+1j*math.pi)
9: (2)      True
10: (0)      Similarly, `sqrt`, other base logarithms, `power` and trig functions are
11: (0)      correctly handled. See their respective docstrings for specific examples.
12: (0)      Functions
13: (0)      -----
14: (0)      .. autosummary::
15: (3)          :toctree: generated/
16: (3)          sqrt
17: (3)          log
18: (3)          log2
19: (3)          logn
20: (3)          log10
21: (3)          power
22: (3)          arccos
23: (3)          arcsin
24: (3)          arctanh
25: (0)      """
26: (0)      import numpy.core.numeric as nx
27: (0)      import numpy.core.numerictypes as nt
28: (0)      from numpy.core.numeric import asarray, any
29: (0)      from numpy.core.overrides import array_function_dispatch
30: (0)      from numpy.lib.type_check import isreal
31: (0)      __all__ = [
32: (4)          'sqrt', 'log', 'log2', 'logn', 'log10', 'power', 'arccos', 'arcsin',
33: (4)          'arctanh'
34: (4)      ]
35: (0)      _ln2 = nx.log(2.0)
36: (0)      def _tocomplex(arr):
37: (4)          """Convert its input `arr` to a complex array.
38: (4)          The input is returned as a complex array of the smallest type that will
39: (4)          fit
40: (4)          the original data: types like single, byte, short, etc. become csingle,
41: (4)          while others become cdouble.
42: (4)          A copy of the input is always made.
43: (4)          Parameters
44: (4)          -----
45: (4)          arr : array
46: (4)          Returns
47: (4)          -----
48: (8)          array
49: (4)          An array with the same input data as the input but in complex form.
50: (4)          Examples
51: (4)          -----
52: (4)          First, consider an input of type short:
53: (4)          >>> a = np.array([1,2,3],np.short)
53: (4)          >>> ac = np.lib.scimath._tocomplex(a); ac
```

```

54: (4) array([1.+0.j, 2.+0.j, 3.+0.j], dtype=complex64)
55: (4) >>> ac.dtype
56: (4) dtype('complex64')
57: (4) If the input is of type double, the output is correspondingly of the
58: (4) complex double type as well:
59: (4) >>> b = np.array([1,2,3],np.double)
60: (4) >>> bc = np.lib.scimath._tocomplex(b); bc
61: (4) array([1.+0.j, 2.+0.j, 3.+0.j])
62: (4) >>> bc.dtype
63: (4) dtype('complex128')
64: (4) Note that even if the input was complex to begin with, a copy is still
65: (4) made, since the astype() method always copies:
66: (4) >>> c = np.array([1,2,3],np.csingle)
67: (4) >>> cc = np.lib.scimath._tocomplex(c); cc
68: (4) array([1.+0.j, 2.+0.j, 3.+0.j], dtype=complex64)
69: (4) >>> c *= 2; c
70: (4) array([2.+0.j, 4.+0.j, 6.+0.j], dtype=complex64)
71: (4) >>> cc
72: (4) array([1.+0.j, 2.+0.j, 3.+0.j], dtype=complex64)
73: (4) """
74: (4) if issubclass(arr.dtype.type, (nt.single, nt.byte, nt.short, nt.ubyte,
75: (35) nt ushort, nt.csingle)):
76: (8)     return arr.astype(nt.csingle)
77: (4) else:
78: (8)     return arr.astype(nt.cdouble)
79: (0) def _fix_real_lt_zero(x):
80: (4) """Convert `x` to complex if it has real, negative components.
81: (4) Otherwise, output is just the array version of the input (via asarray).
82: (4) Parameters
83: (4) -----
84: (4) x : array_like
85: (4) Returns
86: (4) -----
87: (4) array
88: (4) Examples
89: (4) -----
90: (4) >>> np.lib.scimath._fix_real_lt_zero([1,2])
91: (4) array([1, 2])
92: (4) >>> np.lib.scimath._fix_real_lt_zero([-1,2])
93: (4) array([-1.+0.j, 2.+0.j])
94: (4) """
95: (4) x = asarray(x)
96: (4) if any(isreal(x) & (x < 0)):
97: (8)     x = _tocomplex(x)
98: (4) return x
99: (0) def _fix_int_lt_zero(x):
100: (4) """Convert `x` to double if it has real, negative components.
101: (4) Otherwise, output is just the array version of the input (via asarray).
102: (4) Parameters
103: (4) -----
104: (4) x : array_like
105: (4) Returns
106: (4) -----
107: (4) array
108: (4) Examples
109: (4) -----
110: (4) >>> np.lib.scimath._fix_int_lt_zero([1,2])
111: (4) array([1, 2])
112: (4) >>> np.lib.scimath._fix_int_lt_zero([-1,2])
113: (4) array([-1., 2.])
114: (4) """
115: (4) x = asarray(x)
116: (4) if any(isreal(x) & (x < 0)):
117: (8)     x = x * 1.0
118: (4) return x
119: (0) def _fix_real_abs_gt_1(x):
120: (4) """Convert `x` to complex if it has real components x_i with abs(x_i)>1.
121: (4) Otherwise, output is just the array version of the input (via asarray).
122: (4) Parameters

```

```

123: (4)      -----
124: (4)      x : array_like
125: (4)      Returns
126: (4)      -----
127: (4)      array
128: (4)      Examples
129: (4)      -----
130: (4)      >>> np.lib.scimath._fix_real_abs_gt_1([0,1])
131: (4)      array([0, 1])
132: (4)      >>> np.lib.scimath._fix_real_abs_gt_1([0,2])
133: (4)      array([0.+0.j, 2.+0.j])
134: (4)      """
135: (4)      x = asarray(x)
136: (4)      if any(isreal(x) & (abs(x) > 1)):
137: (8)          x = _tocomplex(x)
138: (4)      return x
139: (0)      def _unary_dispatcher(x):
140: (4)          return (x,)
141: (0)      @array_function_dispatch(_unary_dispatcher)
142: (0)      def sqrt(x):
143: (4)          """
144: (4)          Compute the square root of x.
145: (4)          For negative input elements, a complex value is returned
146: (4)          (unlike `numpy.sqrt` which returns NaN).
147: (4)          Parameters
148: (4)          -----
149: (4)          x : array_like
150: (7)              The input value(s).
151: (4)          Returns
152: (4)          -----
153: (4)          out : ndarray or scalar
154: (7)              The square root of `x`. If `x` was a scalar, so is `out`,
155: (7)              otherwise an array is returned.
156: (4)          See Also
157: (4)          -----
158: (4)          numpy.sqrt
159: (4)          Examples
160: (4)          -----
161: (4)          For real, non-negative inputs this works just like `numpy.sqrt`:
162: (4)          >>> np.emath.sqrt(1)
163: (4)          1.0
164: (4)          >>> np.emath.sqrt([1, 4])
165: (4)          array([1., 2.])
166: (4)          But it automatically handles negative inputs:
167: (4)          >>> np.emath.sqrt(-1)
168: (4)          1j
169: (4)          >>> np.emath.sqrt([-1,4])
170: (4)          array([0.+1.j, 2.+0.j])
171: (4)          Different results are expected because:
172: (4)          floating point 0.0 and -0.0 are distinct.
173: (4)          For more control, explicitly use complex() as follows:
174: (4)          >>> np.emath.sqrt(complex(-4.0, 0.0))
175: (4)          2j
176: (4)          >>> np.emath.sqrt(complex(-4.0, -0.0))
177: (4)          -2j
178: (4)          """
179: (4)          x = _fix_real_lt_zero(x)
180: (4)          return nx.sqrt(x)
181: (0)          @array_function_dispatch(_unary_dispatcher)
182: (0)          def log(x):
183: (4)              """
184: (4)              Compute the natural logarithm of `x`.
185: (4)              Return the "principal value" (for a description of this, see `numpy.log`)
186: (4)              of :math:`\log_e(x)`. For real `x > 0`, this is a real number (`\log(0)``
187: (4)              returns ``-inf`` and ``\log(np.inf)`` returns ``inf``). Otherwise, the
188: (4)              complex principle value is returned.
189: (4)              Parameters
190: (4)              -----
191: (4)              x : array_like

```

```

192: (7)           The value(s) whose log is (are) required.
193: (4)           Returns
194: (4)
195: (4)           out : ndarray or scalar
196: (7)           The log of the `x` value(s). If `x` was a scalar, so is `out`,
197: (7)           otherwise an array is returned.
198: (4)           See Also
199: (4)
200: (4)           numpy.log
201: (4)           Notes
202: (4)
203: (4)           -----
204: (4)           For a log() that returns ``NAN`` when real `x < 0`, use `numpy.log`
205: (4)           (note, however, that otherwise `numpy.log` and this `log` are identical,
206: (4)           i.e., both return ``-inf`` for `x = 0`, ``inf`` for `x = inf`, and,
207: (4)           notably, the complex principle value if ``x.imag != 0``).
208: (4)           Examples
209: (4)
210: (4)           >>> np.emath.log(np.exp(1))
211: (4)           1.0
212: (4)           Negative arguments are handled "correctly" (recall that
213: (4)           ``exp(log(x)) == x`` does *not* hold for real ``x < 0``):
214: (4)           >>> np.emath.log(-np.exp(1)) == (1 + np.pi * 1j)
215: (4)           True
216: (4)           """
217: (4)           x = _fix_real_lt_zero(x)
218: (0)           return nx.log(x)
219: (0)           @array_function_dispatch(_unary_dispatcher)
220: (4)           def log10(x):
221: (4)               """
222: (4)               Compute the logarithm base 10 of `x`.
223: (4)               Return the "principal value" (for a description of this, see
224: (4)               `numpy.log10`) of :math:`\log_{10}(x)`. For real `x > 0`, this
225: (4)               is a real number (``log10(0)`` returns ``-inf`` and ``log10(np.inf)``
226: (4)               returns ``inf``). Otherwise, the complex principle value is returned.
227: (4)               Parameters
228: (4)
229: (7)               x : array_like or scalar
230: (4)               The value(s) whose log base 10 is (are) required.
231: (4)           Returns
232: (4)
233: (7)           out : ndarray or scalar
234: (7)           The log base 10 of the `x` value(s). If `x` was a scalar, so is `out`,
235: (4)           otherwise an array object is returned.
236: (4)           See Also
237: (4)
238: (4)           numpy.log10
239: (4)           Notes
240: (4)
241: (4)           -----
242: (4)           For a log10() that returns ``NAN`` when real `x < 0`, use `numpy.log10`
243: (4)           (note, however, that otherwise `numpy.log10` and this `log10` are
244: (4)           identical, i.e., both return ``-inf`` for `x = 0`, ``inf`` for `x = inf`,
245: (4)           and, notably, the complex principle value if ``x.imag != 0``).
246: (4)           Examples
247: (4)
248: (4)           (We set the printing precision so the example can be auto-tested)
249: (4)           >>> np.set_printoptions(precision=4)
250: (4)           >>> np.emath.log10(10**1)
251: (4)           1.0
252: (4)           >>> np.emath.log10([-10**1, -10**2, 10**2])
253: (4)           array([1.+1.3644j, 2.+1.3644j, 2.+0.j      ])
254: (4)           """
255: (0)           x = _fix_real_lt_zero(x)
256: (4)           return nx.log10(x)
257: (0)           @array_function_dispatch(_logn_dispatcher)
258: (0)           def logn(n, x):
259: (4)               """
260: (4)               Take log base n of x.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

261: (4)             If `x` contains negative inputs, the answer is computed and returned in
the
262: (4)             complex domain.
263: (4)             Parameters
264: (4)             -----
265: (4)             n : array_like
266: (7)             The integer base(s) in which the log is taken.
267: (4)             x : array_like
268: (7)             The value(s) whose log base `n` is (are) required.
269: (4)             Returns
270: (4)             -----
271: (4)             out : ndarray or scalar
272: (7)             The log base `n` of the `x` value(s). If `x` was a scalar, so is
`out`, otherwise an array is returned.
273: (4)             Examples
274: (4)             -----
275: (4)             >>> np.set_printoptions(precision=4)
276: (4)             >>> np.emath.logn(2, [4, 8])
277: (4)             array([2., 3.])
278: (4)             >>> np.emath.logn(2, [-4, -8, 8])
279: (4)             array([2.+4.5324j, 3.+4.5324j, 3.+0.j      ])
280: (4)             """
281: (4)
282: (4)             x = _fix_real_lt_zero(x)
283: (4)             n = _fix_real_lt_zero(n)
284: (4)             return nx.log(x)/nx.log(n)
285: (0)             @array_function_dispatch(_unary_dispatcher)
286: (0)             def log2(x):
287: (4)             """
288: (4)             Compute the logarithm base 2 of `x`.
289: (4)             Return the "principal value" (for a description of this, see
`numpy.log2`) of :math:`\log_2(x)`. For real `x > 0`, this is
290: (4)             a real number (`log2(0)` returns ``-inf`` and `log2(np.inf)` returns
`inf`). Otherwise, the complex principle value is returned.
291: (4)             Parameters
292: (4)             -----
293: (4)             x : array_like
294: (4)             The value(s) whose log base 2 is (are) required.
295: (4)             Returns
296: (4)             -----
297: (4)             out : ndarray or scalar
298: (4)             The log base 2 of the `x` value(s). If `x` was a scalar, so is `out`,
299: (4)             otherwise an array is returned.
300: (7)             See Also
301: (7)             -----
302: (4)             numpy.log2
303: (4)             Notes
304: (4)             -----
305: (4)             For a log2() that returns ``NAN`` when real `x < 0`, use `numpy.log2`
306: (4)             (note, however, that otherwise `numpy.log2` and this `log2` are
307: (4)             identical, i.e., both return ``-inf`` for `x = 0`, ``inf`` for `x = inf`,
308: (4)             and, notably, the complex principle value if ``x.imag != 0``).
309: (4)             Examples
310: (4)             -----
311: (4)             We set the printing precision so the example can be auto-tested:
312: (4)             >>> np.set_printoptions(precision=4)
313: (4)             >>> np.emath.log2(8)
314: (4)             3.0
315: (4)             >>> np.emath.log2([-4, -8, 8])
316: (4)             array([2.+4.5324j, 3.+4.5324j, 3.+0.j      ])
317: (4)             """
318: (4)
319: (4)
320: (4)             x = _fix_real_lt_zero(x)
321: (4)             return nx.log2(x)
322: (0)             def _power_dispatcher(x, p):
323: (4)             return (x, p)
324: (0)             @array_function_dispatch(_power_dispatcher)
325: (0)             def power(x, p):
326: (4)             """
327: (4)             Return x to the power p, ( $x^{**p}$ ).
328: (4)             If `x` contains negative values, the output is converted to the

```

```

329: (4)           complex domain.
330: (4)           Parameters
331: (4)
332: (4)           -----
333: (8)           x : array_like
334: (4)           The input value(s).
335: (8)           p : array_like of ints
336: (8)           The power(s) to which `x` is raised. If `x` contains multiple values,
337: (8)           `p` has to either be a scalar, or contain the same number of values
338: (8)           as `x`. In the latter case, the result is
339: (4)           ``x[0]**p[0], x[1]**p[1], ...``.
340: (4)           Returns
341: (4)
342: (8)           out : ndarray or scalar
343: (8)           The result of ``x**p``. If `x` and `p` are scalars, so is `out`,
344: (4)           otherwise an array is returned.
345: (4)           See Also
346: (4)
347: (4)           numpy.power
348: (4)
349: (4)           >>> np.set_printoptions(precision=4)
350: (4)           >>> np.emath.power([2, 4], 2)
351: (4)           array([ 4, 16])
352: (4)           >>> np.emath.power([2, 4], -2)
353: (4)           array([0.25, 0.0625])
354: (4)           >>> np.emath.power([-2, 4], 2)
355: (4)           array([ 4.-0.j, 16.+0.j])
356: (4)
357: (4)           """
358: (4)           x = _fix_real_lt_zero(x)
359: (4)           p = _fix_int_lt_zero(p)
360: (0)           return nx.power(x, p)
361: (0)           @array_function_dispatch(_unary_dispatcher)
362: (4)           def arccos(x):
363: (4)               """
364: (4)               Compute the inverse cosine of x.
365: (4)               Return the "principal value" (for a description of this, see
366: (4)               `numpy.arccos`) of the inverse cosine of `x`. For real `x` such that
367: (4)               `abs(x) <= 1`, this is a real number in the closed interval
368: (4)               :math:`[0, \pi]`. Otherwise, the complex principle value is returned.
369: (4)               Parameters
370: (4)
371: (7)               x : array_like or scalar
372: (4)               The value(s) whose arccos is (are) required.
373: (4)               Returns
374: (4)
375: (7)               out : ndarray or scalar
376: (7)               The inverse cosine(s) of the `x` value(s). If `x` was a scalar, so
377: (4)               is `out`, otherwise an array object is returned.
378: (4)               See Also
379: (4)
380: (4)               numpy.arccos
381: (4)
382: (4)               Notes
383: (4)
384: (4)               -----
385: (4)               For an arccos() that returns ``NAN`` when real `x` is not in the
386: (4)               interval ``[-1,1]``, use `numpy.arccos`.
387: (4)               Examples
388: (4)
389: (4)               >>> np.set_printoptions(precision=4)
390: (4)               >>> np.emath.arccos(1) # a scalar is returned
391: (4)               0.0
392: (4)               >>> np.emath.arccos([1,2])
393: (4)               array([0.-0.j, 0.-1.317j])
394: (4)
395: (4)               """
396: (4)               x = _fix_real_abs_gt_1(x)
397: (4)               return nx.arccos(x)
398: (0)               @array_function_dispatch(_unary_dispatcher)
399: (0)               def arcsin(x):
400: (4)
401: (4)               """
402: (4)               Compute the inverse sine of x.

```

```

398: (4)             Return the "principal value" (for a description of this, see
399: (4)             `numpy.arcsin` ) of the inverse sine of `x` . For real `x` such that
400: (4)             `abs(x) <= 1` , this is a real number in the closed interval
401: (4)             :math: `[-\pi/2, \pi/2]` . Otherwise, the complex principle value is
402: (4)             returned.
403: (4)             Parameters
404: (4)             -----
405: (4)             x : array_like or scalar
406: (7)                 The value(s) whose arcsin is (are) required.
407: (4)             Returns
408: (4)             -----
409: (4)             out : ndarray or scalar
410: (7)                 The inverse sine(s) of the `x` value(s). If `x` was a scalar, so
411: (7)                 is `out` , otherwise an array object is returned.
412: (4)             See Also
413: (4)             -----
414: (4)             numpy.arcsin
415: (4)             Notes
416: (4)             -----
417: (4)             For an arcsin() that returns ``NAN`` when real `x` is not in the
418: (4)             interval ``[-1,1]`` , use `numpy.arcsin` .
419: (4)             Examples
420: (4)             -----
421: (4)             >>> np.set_printoptions(precision=4)
422: (4)             >>> np.emath.arcsin(0)
423: (4)             0.0
424: (4)             >>> np.emath.arcsin([0,1])
425: (4)             array([0.        , 1.5708        ])
426: (4)             """
427: (4)             x = _fix_real_abs_gt_1(x)
428: (4)             return nx.arcsin(x)
429: (0)             @array_function_dispatch(_unary_dispatcher)
430: (0)             def arctanh(x):
431: (4)                 """
432: (4)                 Compute the inverse hyperbolic tangent of `x` .
433: (4)                 Return the "principal value" (for a description of this, see
434: (4)                 `numpy.arctanh` ) of ``arctanh(x)`` . For real `x` such that
435: (4)                 ``abs(x) < 1`` , this is a real number. If `abs(x) > 1` , or if `x` is
436: (4)                 complex, the result is complex. Finally, `x = 1` returns ``inf`` and
437: (4)                 ``x=-1`` returns ``-inf`` .
438: (4)                 Parameters
439: (4)                 -----
440: (4)                 x : array_like
441: (7)                     The value(s) whose arctanh is (are) required.
442: (4)                 Returns
443: (4)                 -----
444: (4)                 out : ndarray or scalar
445: (7)                     The inverse hyperbolic tangent(s) of the `x` value(s). If `x` was
446: (7)                     a scalar so is `out` , otherwise an array is returned.
447: (4)                 See Also
448: (4)                 -----
449: (4)                 numpy.arctanh
450: (4)                 Notes
451: (4)                 -----
452: (4)                 For an arctanh() that returns ``NAN`` when real `x` is not in the
453: (4)                 interval ``(-1,1)`` , use `numpy.arctanh` (this latter, however, does
454: (4)                 return +/-inf for ``x = +/-1`` ).
455: (4)                 Examples
456: (4)                 -----
457: (4)                 >>> np.set_printoptions(precision=4)
458: (4)                 >>> from numpy.testing import suppress_warnings
459: (4)                 >>> with suppress_warnings() as sup:
460: (4)                     ...     sup.filter(RuntimeWarning)
461: (4)                     ...     np.emath.arctanh(np.eye(2))
462: (4)                     array([[inf,  0.],
463: (11)                         [ 0., inf]])
464: (4)                     >>> np.emath.arctanh([1j])
465: (4)                     array([0.+0.7854j])
466: (4)                     """

```

```
467: (4)         x = _fix_real_abs_gt_1(x)
468: (4)         return nx.arctanh(x)
```

-----  
File 217 - setup.py:

```
1: (0)             def configuration(parent_package='', top_path=None):
2: (4)                 from numpy.distutils.misc_util import Configuration
3: (4)                 config = Configuration('lib', parent_package, top_path)
4: (4)                 config.add_subpackage('tests')
5: (4)                 config.add_data_dir('tests/data')
6: (4)                 config.add_data_files('*/*.pyi')
7: (4)                 return config
8: (0)             if __name__ == '__main__':
9: (4)                 from numpy.distutils.core import setup
10: (4)                setup(configuration=configuration)
```

-----  
File 218 - shape\_base.py:

```
1: (0)             import functools
2: (0)             import numpy.core.numeric as _nx
3: (0)             from numpy.core.numeric import asarray, zeros, array, asanyarray
4: (0)             from numpy.core.fromnumeric import reshape, transpose
5: (0)             from numpy.core.multiarray import normalize_axis_index
6: (0)             from numpy.core import overrides
7: (0)             from numpy.core import vstack, atleast_3d
8: (0)             from numpy.core.numeric import normalize_axis_tuple
9: (0)             from numpy.core.shape_base import _arrays_for_stack_dispatcher
10: (0)            from numpy.lib.index_tricks import ndindex
11: (0)            from numpy.matrixlib.defmatrix import matrix # this raises all the right
alarm bells
12: (0)            __all__ = [
13: (4)                'column_stack', 'row_stack', 'dstack', 'array_split', 'split',
14: (4)                'hsplit', 'vsplit', 'dsplit', 'apply_over_axes', 'expand_dims',
15: (4)                'apply_along_axis', 'kron', 'tile', 'get_array_wrap', 'take_along_axis',
16: (4)                'put_along_axis'
17: (4)            ]
18: (0)            array_function_dispatch = functools.partial(
19: (4)                overrides.array_function_dispatch, module='numpy')
20: (0)            def _make_along_axis_idx(arr_shape, indices, axis):
21: (4)                if not _nx.issubdtype(indices.dtype, _nx.integer):
22: (8)                    raise IndexError(`indices` must be an integer array)
23: (4)                if len(arr_shape) != indices.ndim:
24: (8)                    raise ValueError(
25: (12)                        ``indices` and `arr` must have the same number of dimensions")
26: (4)                shape_ones = (1,) * indices.ndim
27: (4)                dest_dims = list(range(axis)) + [None] + list(range(axis+1, indices.ndim))
28: (4)                fancy_index = []
29: (4)                for dim, n in zip(dest_dims, arr_shape):
30: (8)                    if dim is None:
31: (12)                        fancy_index.append(indices)
32: (8)                    else:
33: (12)                        ind_shape = shape_ones[:dim] + (-1,) + shape_ones[dim+1:]
34: (12)                        fancy_index.append(_nx.arange(n).reshape(ind_shape))
35: (4)                return tuple(fancy_index)
36: (0)            def _take_along_axis_dispatcher(arr, indices, axis):
37: (4)                return (arr, indices)
38: (0)                @array_function_dispatch(_take_along_axis_dispatcher)
39: (0)            def take_along_axis(arr, indices, axis):
40: (4)                """
41: (4)                    Take values from the input array by matching 1d index and data slices.
42: (4)                    This iterates over matching 1d slices oriented along the specified axis in
43: (4)                    the index and data arrays, and uses the former to look up values in the
44: (4)                    latter. These slices can be different lengths.
45: (4)                    Functions returning an index along an axis, like `argsort` and
46: (4)                    `argpartition`, produce suitable indices for this function.
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

47: (4) .. versionadded:: 1.15.0
48: (4) Parameters
49: (4) -----
50: (4) arr : ndarray (Ni..., M, Nk...)
51: (8)     Source array
52: (4) indices : ndarray (Ni..., J, Nk...)
53: (8)     Indices to take along each 1d slice of `arr`. This must match the
54: (8)     dimension of arr, but dimensions Ni and Nj only need to broadcast
55: (8)     against `arr`.
56: (4) axis : int
57: (8)     The axis to take 1d slices along. If axis is None, the input array is
58: (8)     treated as if it had first been flattened to 1d, for consistency with
59: (8)     `sort` and `argsort`.
60: (4) Returns
61: (4) -----
62: (4) out: ndarray (Ni..., J, Nk...)
63: (8)     The indexed result.
64: (4) Notes
65: (4) -----
66: (4) This is equivalent to (but faster than) the following use of `ndindex` and
67: (4) `s_`, which sets each of ``ii`` and ``kk`` to a tuple of indices::
68: (8) Ni, M, Nk = a.shape[:axis], a.shape[axis], a.shape[axis+1:]
69: (8) J = indices.shape[axis] # Need not equal M
70: (8) out = np.empty(Ni + (J,) + Nk)
71: (8) for ii in ndindex(Ni):
72: (12)     for kk in ndindex(Nk):
73: (16)         a_1d      = a      [ii + s_[:,] + kk]
74: (16)         indices_1d = indices[ii + s_[:,] + kk]
75: (16)         out_1d    = out    [ii + s_[:,] + kk]
76: (16)         for j in range(J):
77: (20)             out_1d[j] = a_1d[indices_1d[j]]
78: (4) Equivalently, eliminating the inner loop, the last two lines would be::
79: (16)         out_1d[:] = a_1d[indices_1d]
80: (4) See Also
81: (4) -----
82: (4) take : Take along an axis, using the same indices for every 1d slice
83: (4) put_along_axis :
84: (8)     Put values into the destination array by matching 1d index and data
slices
85: (4) Examples
86: (4) -----
87: (4) For this sample array
88: (4) >>> a = np.array([[10, 30, 20], [60, 40, 50]])
89: (4) We can sort either by using sort directly, or argsort and this function
90: (4) >>> np.sort(a, axis=1)
91: (4) array([[10, 20, 30],
92: (11)      [40, 50, 60]])
93: (4) >>> ai = np.argsort(a, axis=1)
94: (4) >>> ai
95: (4) array([[0, 2, 1],
96: (11)      [1, 2, 0]])
97: (4) >>> np.take_along_axis(a, ai, axis=1)
98: (4) array([[10, 20, 30],
99: (11)      [40, 50, 60]])
100: (4) The same works for max and min, if you maintain the trivial dimension
101: (4) with ``keepdims``:
102: (4) >>> np.max(a, axis=1, keepdims=True)
103: (4) array([[30],
104: (11)      [60]])
105: (4) >>> ai = np.argmax(a, axis=1, keepdims=True)
106: (4) >>> ai
107: (4) array([[1],
108: (11)      [0]])
109: (4) >>> np.take_along_axis(a, ai, axis=1)
110: (4) array([[30],
111: (11)      [60]])
112: (4) If we want to get the max and min at the same time, we can stack the
113: (4) indices first
114: (4) >>> ai_min = np.argmin(a, axis=1, keepdims=True)

```

```

115: (4)                                     >>> ai_max = np.argmax(a, axis=1, keepdims=True)
116: (4)                                     >>> ai = np.concatenate([ai_min, ai_max], axis=1)
117: (4)                                     >>> ai
118: (4)                                     array([[0, 1],
119: (11)                                         [1, 0]])
120: (4)                                     >>> np.take_along_axis(a, ai, axis=1)
121: (4)                                     array([[10, 30],
122: (11)                                         [40, 60]])
123: (4)                                     """
124: (4)                                     if axis is None:
125: (8)                                         arr = arr.flat
126: (8)                                         arr_shape = (len(arr),) # flatiter has no .shape
127: (8)                                         axis = 0
128: (4)                                     else:
129: (8)                                         axis = normalize_axis_index(axis, arr.ndim)
130: (8)                                         arr_shape = arr.shape
131: (4)                                     return arr[_make_along_axis_idx(arr_shape, indices, axis)]
132: (0)                                     def _put_along_axis_dispatcher(arr, indices, values, axis):
133: (4)                                         return (arr, indices, values)
134: (0)                                         @array_function_dispatch(_put_along_axis_dispatcher)
135: (0)                                         def put_along_axis(arr, indices, values, axis):
136: (4)                                         """
137: (4)                                         Put values into the destination array by matching 1d index and data
slices.
138: (4)                                         This iterates over matching 1d slices oriented along the specified axis in
139: (4)                                         the index and data arrays, and uses the former to place values into the
140: (4)                                         latter. These slices can be different lengths.
141: (4)                                         Functions returning an index along an axis, like `argsort` and
142: (4)                                         `argpartition`, produce suitable indices for this function.
143: (4)                                         .. versionadded:: 1.15.0
144: (4)                                         Parameters
145: (4)                                         -----
146: (4)                                         arr : ndarray (Ni..., M, Nk...)
147: (8)                                         Destination array.
148: (4)                                         indices : ndarray (Ni..., J, Nk...)
149: (8)                                         Indices to change along each 1d slice of `arr`. This must match the
150: (8)                                         dimension of arr, but dimensions in Ni and Nj may be 1 to broadcast
151: (8)                                         against `arr`.
152: (4)                                         values : array_like (Ni..., J, Nk...)
153: (8)                                         values to insert at those indices. Its shape and dimension are
154: (8)                                         broadcast to match that of `indices`.
155: (4)                                         axis : int
156: (8)                                         The axis to take 1d slices along. If axis is None, the destination
157: (8)                                         array is treated as if a flattened 1d view had been created of it.
158: (4)                                         Notes
159: (4)                                         -----
160: (4)                                         This is equivalent to (but faster than) the following use of `ndindex` and
161: (4)                                         `s_`, which sets each of ``ii`` and ``kk`` to a tuple of indices::
162: (8)                                         Ni, M, Nk = a.shape[:axis], a.shape[axis], a.shape[axis+1:]
163: (8)                                         J = indices.shape[axis] # Need not equal M
164: (8)                                         for ii in ndindex(Ni):
165: (12)                                         for kk in ndindex(Nk):
166: (16)                                         a_1d      = a      [ii + s_[:,] + kk]
167: (16)                                         indices_1d = indices[ii + s_[:,] + kk]
168: (16)                                         values_1d = values [ii + s_[:,] + kk]
169: (16)                                         for j in range(J):
170: (20)                                         a_1d[indices_1d[j]] = values_1d[j]
171: (4)                                         Equivalently, eliminating the inner loop, the last two lines would be::
172: (16)                                         a_1d[indices_1d] = values_1d
173: (4)                                         See Also
174: (4)                                         -----
175: (4)                                         take_along_axis :
176: (8)                                         Take values from the input array by matching 1d index and data slices
177: (4)                                         Examples
178: (4)                                         -----
179: (4)                                         For this sample array
180: (4)                                         >>> a = np.array([[10, 30, 20], [60, 40, 50]])
181: (4)                                         We can replace the maximum values with:
182: (4)                                         >>> ai = np.argmax(a, axis=1, keepdims=True)

```

```

183: (4)             >>> ai
184: (4)             array([[1],
185: (11)            [0]])
186: (4)             >>> np.put_along_axis(a, ai, 99, axis=1)
187: (4)             >>> a
188: (4)             array([[10, 99, 20],
189: (11)            [99, 40, 50]])
190: (4)             """
191: (4)             if axis is None:
192: (8)                 arr = arr.flat
193: (8)                 axis = 0
194: (8)                 arr_shape = (len(arr),) # flatiter has no .shape
195: (4)             else:
196: (8)                 axis = normalize_axis_index(axis, arr.ndim)
197: (8)                 arr_shape = arr.shape
198: (4)             arr[_make_along_axis_idx(arr_shape, indices, axis)] = values
199: (0)             def _apply_along_axis_dispatcher(func1d, axis, arr, *args, **kwargs):
200: (4)                 return (arr,)
201: (0)             @array_function_dispatch(_apply_along_axis_dispatcher)
202: (0)             def apply_along_axis(func1d, axis, arr, *args, **kwargs):
203: (4)                 """
204: (4)                 Apply a function to 1-D slices along the given axis.
205: (4)                 Execute `func1d(a, *args, **kwargs)` where `func1d` operates on 1-D arrays
206: (4)                 and `a` is a 1-D slice of `arr` along `axis`.
207: (4)                 This is equivalent to (but faster than) the following use of `ndindex` and
208: (4)                 `s_`, which sets each of ``ii``, ``jj``, and ``kk`` to a tuple of
indices:::
209: (8)                 Ni, Nk = a.shape[:axis], a.shape[axis+1:]
210: (8)                 for ii in ndindex(Ni):
211: (12)                   for kk in ndindex(Nk):
212: (16)                     f = func1d(arr[ii + s_[:,] + kk])
213: (16)                     Nj = f.shape
214: (16)                     for jj in ndindex(Nj):
215: (20)                       out[ii + jj + kk] = f[jj]
216: (4)             Equivalently, eliminating the inner loop, this can be expressed as::
217: (8)                 Ni, Nk = a.shape[:axis], a.shape[axis+1:]
218: (8)                 for ii in ndindex(Ni):
219: (12)                   for kk in ndindex(Nk):
220: (16)                     out[ii + s_[...,] + kk] = func1d(arr[ii + s_[:,] + kk])
221: (4)             Parameters
222: (4)             -----
223: (4)             func1d : function (M,) -> (Nj...)
224: (8)                 This function should accept 1-D arrays. It is applied to 1-D
225: (8)                 slices of `arr` along the specified axis.
226: (4)             axis : integer
227: (8)                 Axis along which `arr` is sliced.
228: (4)             arr : ndarray (Ni..., M, Nk...)
229: (8)                 Input array.
230: (4)             args : any
231: (8)                 Additional arguments to `func1d`.
232: (4)             kwargs : any
233: (8)                 Additional named arguments to `func1d`.
234: (8)                 .. versionadded:: 1.9.0
235: (4)             Returns
236: (4)             -----
237: (4)             out : ndarray (Ni..., Nj..., Nk...)
238: (8)                 The output array. The shape of `out` is identical to the shape of
239: (8)                 `arr`, except along the `axis` dimension. This axis is removed, and
240: (8)                 replaced with new dimensions equal to the shape of the return value
241: (8)                 of `func1d`. So if `func1d` returns a scalar `out` will have one
242: (8)                 fewer dimensions than `arr`.
243: (4)             See Also
244: (4)             -----
245: (4)             apply_over_axes : Apply a function repeatedly over multiple axes.
246: (4)             Examples
247: (4)             -----
248: (4)             >>> def my_func(a):
249: (4)                 ...     """Average first and last element of a 1-D array"""
250: (4)                 ...     return (a[0] + a[-1]) * 0.5

```

```

251: (4)
252: (4)
253: (4)
254: (4)
255: (4)
256: (4)
257: (4)
258: (4)
259: (4)
260: (4)
261: (11)
262: (11)
263: (4)
264: (4)
265: (4)
266: (4)
267: (4)
268: (12)
269: (12)
270: (11)
271: (12)
272: (12)
273: (11)
274: (12)
275: (12)
276: (4)
277: (4)
278: (4)
279: (4)
280: (4)
281: (4)
282: (4)
283: (4)
284: (4)
285: (8)
286: (4)
287: (8)
288: (12)
289: (8)
290: (4)
291: (4)
292: (4)
293: (4)
294: (8)
295: (8)
296: (8)
297: (4)
298: (4)
299: (8)
300: (4)
301: (4)
302: (8)
303: (4)
304: (8)
305: (8)
306: (4)
307: (8)
308: (8)
309: (0)
310: (4)
311: (0)
312: (0)
313: (4)
314: (4)
315: (4)
316: (4)
317: (4)
318: (4)
319: (4)

        >>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
        >>> np.apply_along_axis(my_func, 0, b)
        array([4., 5., 6.])
        >>> np.apply_along_axis(my_func, 1, b)
        array([2., 5., 8.])
        For a function that returns a 1D array, the number of dimensions in
`outarr` is the same as `arr`.
        >>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
        >>> np.apply_along_axis(sorted, 1, b)
        array([[1, 7, 8],
               [3, 4, 9],
               [2, 5, 6]])
        For a function that returns a higher dimensional array, those dimensions
are inserted in place of the `axis` dimension.
        >>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
        >>> np.apply_along_axis(np.diag, -1, b)
        array([[[1, 0, 0],
               [0, 2, 0],
               [0, 0, 3]],
               [[4, 0, 0],
               [0, 5, 0],
               [0, 0, 6]],
               [[7, 0, 0],
               [0, 8, 0],
               [0, 0, 9]]])
        """
        arr = asanyarray(arr)
        nd = arr.ndim
        axis = normalize_axis_index(axis, nd)
        in_dims = list(range(nd))
        inarr_view = transpose(arr, in_dims[:axis] + in_dims[axis+1:] + [axis])
        inds = ndindex(inarr_view.shape[:-1])
        inds = (ind + (Ellipsis,) for ind in inds)
        try:
            ind0 = next(inds)
        except StopIteration:
            raise ValueError(
                'Cannot apply_along_axis when any iteration dimensions are 0'
            ) from None
        res = asanyarray(func1d(inarr_view[ind0], *args, **kwargs))
        buff = zeros(inarr_view.shape[:-1] + res.shape, res.dtype)
        buff_dims = list(range(buff.ndim))
        buff_permute =
            buff_dims[0 : axis] +
            buff_dims[buff.ndim-res.ndim : buff.ndim] +
            buff_dims[axis : buff.ndim-res.ndim]
        )
        if not isinstance(res, matrix):
            buff = res.__array_prepare__(buff)
        buff[ind0] = res
        for ind in inds:
            buff[ind] = asanyarray(func1d(inarr_view[ind], *args, **kwargs))
        if not isinstance(res, matrix):
            buff = res.__array_wrap__(buff)
            return transpose(buff, buff_permute)
        else:
            out_arr = transpose(buff, buff_permute)
            return res.__array_wrap__(out_arr)
    def _apply_over_axes_dispatcher(func, a, axes):
        return (a,)
    @array_function_dispatch(_apply_over_axes_dispatcher)
    def apply_over_axes(func, a, axes):
        """
        Apply a function repeatedly over multiple axes.
        `func` is called as `res = func(a, axis)`, where `axis` is the first
        element of `axes`. The result `res` of the function call must have
        either the same dimensions as `a` or one less dimension. If `res`
        has one less dimension than `a`, a dimension is inserted before
        `axis`. The call to `func` is then repeated for each axis in `axes`,

```

```

320: (4)           with `res` as the first argument.
321: (4)           Parameters
322: (4)           -----
323: (4)           func : function
324: (8)             This function must take two arguments, `func(a, axis)` .
325: (4)           a : array_like
326: (8)             Input array.
327: (4)           axes : array_like
328: (8)             Axes over which `func` is applied; the elements must be integers.
329: (4)           Returns
330: (4)           -----
331: (4)           apply_over_axis : ndarray
332: (8)             The output array. The number of dimensions is the same as `a` ,
333: (8)             but the shape can be different. This depends on whether `func` 
334: (8)             changes the shape of its output with respect to its input.
335: (4)           See Also
336: (4)           -----
337: (4)           apply_along_axis :
338: (8)             Apply a function to 1-D slices of an array along the given axis.
339: (4)           Notes
340: (4)           -----
341: (4)           This function is equivalent to tuple axis arguments to reorderable ufuncs
342: (4)           with keepdims=True. Tuple axis arguments to ufuncs have been available
since
343: (4)           version 1.7.0.
344: (4)           Examples
345: (4)           -----
346: (4)           >>> a = np.arange(24).reshape(2,3,4)
347: (4)           >>> a
348: (4)           array([[[ 0,  1,  2,  3],
349: (12)             [ 4,  5,  6,  7],
350: (12)             [ 8,  9, 10, 11]],
351: (11)             [[12, 13, 14, 15],
352: (12)             [16, 17, 18, 19],
353: (12)             [20, 21, 22, 23]]])
354: (4)           Sum over axes 0 and 2. The result has same number of dimensions
355: (4)           as the original array:
356: (4)           >>> np.apply_over_axes(np.sum, a, [0,2])
357: (4)           array([[ 60,
358: (12)             [ 92],
359: (12)             [124]]])
360: (4)           Tuple axis arguments to ufuncs are equivalent:
361: (4)           >>> np.sum(a, axis=(0,2), keepdims=True)
362: (4)           array([[ 60],
363: (12)             [ 92],
364: (12)             [124]]])
365: (4)           """
366: (4)           val = asarray(a)
367: (4)           N = a.ndim
368: (4)           if array(axes).ndim == 0:
369: (8)             axes = (axes,)
370: (4)           for axis in axes:
371: (8)             if axis < 0:
372: (12)               axis = N + axis
373: (8)             args = (val, axis)
374: (8)             res = func(*args)
375: (8)             if res.ndim == val.ndim:
376: (12)               val = res
377: (8)             else:
378: (12)               res = expand_dims(res, axis)
379: (12)               if res.ndim == val.ndim:
380: (16)                 val = res
381: (12)               else:
382: (16)                 raise ValueError("function is not returning "
383: (33)                               "an array of the correct shape")
384: (4)               return val
385: (0)           def _expand_dims_dispatcher(a, axis):
386: (4)             return (a,)
387: (0)           @array_function_dispatch(_expand_dims_dispatcher)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

388: (0)
389: (4)
390: (4)
391: (4)
392: (4)
393: (4)
394: (4)
395: (4)
396: (8)
397: (4)
398: (8)
399: (8)
400: (12)
401: (12)
402: (12)
403: (8)
404: (12)
405: (12)
406: (4)
407: (4)
408: (4)
409: (8)
410: (4)
411: (4)
412: (4)
413: (4)
414: (4)
415: (4)
416: (4)
417: (4)
418: (4)
419: (4)
420: (4)
421: (4)
422: (4)
423: (4)
424: (4)
425: (4)
426: (4)
427: (4)
428: (4)
429: (4)
430: (11)
431: (4)
432: (4)
433: (4)
434: (4)
435: (4)
436: (4)
437: (4)
438: (4)
439: (4)
440: (12)
441: (4)
442: (4)
443: (4)
444: (4)
445: (4)
446: (4)
447: (8)
448: (4)
449: (8)
450: (4)
451: (8)
452: (4)
453: (4)
454: (4)
455: (4)
456: (4)

def expand_dims(a, axis):
    """
        Expand the shape of an array.
        Insert a new axis that will appear at the `axis` position in the expanded
        array shape.
        Parameters
        -----
        a : array_like
            Input array.
        axis : int or tuple of ints
            Position in the expanded axes where the new axis (or axes) is placed.
            .. deprecated:: 1.13.0
                Passing an axis where ``axis > a.ndim`` will be treated as
                ``axis == a.ndim``, and passing ``axis < -a.ndim - 1`` will
                be treated as ``axis == 0``. This behavior is deprecated.
            .. versionchanged:: 1.18.0
                A tuple of axes is now supported. Out of range axes as
                described above are now forbidden and raise an `AxisError`.
        Returns
        -----
        result : ndarray
            View of `a` with the number of dimensions increased.
        See Also
        -----
        squeeze : The inverse operation, removing singleton dimensions
        reshape : Insert, remove, and combine dimensions, and resize existing ones
        doc.indexing, atleast_1d, atleast_2d, atleast_3d
        Examples
        -----
        >>> x = np.array([1, 2])
        >>> x.shape
        (2,)
        The following is equivalent to ``x[np.newaxis, :]`` or ``x[np.newaxis]``:
        >>> y = np.expand_dims(x, axis=0)
        >>> y
        array([[1, 2]])
        >>> y.shape
        (1, 2)
        The following is equivalent to ``x[:, np.newaxis]``:
        >>> y = np.expand_dims(x, axis=1)
        >>> y
        array([[1],
               [2]])
        >>> y.shape
        (2, 1)
        ``axis`` may also be a tuple:
        >>> y = np.expand_dims(x, axis=(0, 1))
        >>> y
        array([[[1, 2]]])
        >>> y = np.expand_dims(x, axis=(2, 0))
        >>> y
        array([[[[1,
                  [2]]]]]
        Note that some examples may use ``None`` instead of ``np.newaxis``. These
        are the same objects:
        >>> np.newaxis is None
        True
        """
        if isinstance(a, matrix):
            a = asarray(a)
        else:
            a = asanyarray(a)
        if type(axis) not in (tuple, list):
            axis = (axis,)
        out_ndim = len(axis) + a.ndim
        axis = normalize_axis_tuple(axis, out_ndim)
        shape_it = iter(a.shape)
        shape = [1 if ax in axis else next(shape_it) for ax in range(out_ndim)]
        return a.reshape(shape)

```

```

457: (0)           row_stack = vstack
458: (0)           def _column_stack_dispatcher(tup):
459: (4)             return _arrays_for_stack_dispatcher(tup)
460: (0)           @array_function_dispatch(_column_stack_dispatcher)
461: (0)           def column_stack(tup):
462: (4)             """
463: (4)               Stack 1-D arrays as columns into a 2-D array.
464: (4)               Take a sequence of 1-D arrays and stack them as columns
465: (4)               to make a single 2-D array. 2-D arrays are stacked as-is,
466: (4)               just like with `hstack`. 1-D arrays are turned into 2-D columns
467: (4)               first.
468: (4)               Parameters
469: (4)                 -----
470: (4)                 tup : sequence of 1-D or 2-D arrays.
471: (8)                   Arrays to stack. All of them must have the same first dimension.
472: (4)               Returns
473: (4)                 -----
474: (4)                 stacked : 2-D array
475: (8)                   The array formed by stacking the given arrays.
476: (4)               See Also
477: (4)                 -----
478: (4)                 stack, hstack, vstack, concatenate
479: (4)               Examples
480: (4)                 -----
481: (4)                 >>> a = np.array((1,2,3))
482: (4)                 >>> b = np.array((2,3,4))
483: (4)                 >>> np.column_stack((a,b))
484: (4)                 array([[1, 2],
485: (11)                     [2, 3],
486: (11)                     [3, 4]])
487: (4)                 """
488: (4)                 arrays = []
489: (4)                 for v in tup:
490: (8)                   arr = asanyarray(v)
491: (8)                   if arr.ndim < 2:
492: (12)                     arr = array(arr, copy=False, subok=True, ndmin=2).T
493: (8)                     arrays.append(arr)
494: (4)                 return _nx.concatenate(arrays, 1)
495: (0)           def _dstack_dispatcher(tup):
496: (4)             return _arrays_for_stack_dispatcher(tup)
497: (0)           @array_function_dispatch(_dstack_dispatcher)
498: (0)           def dstack(tup):
499: (4)             """
500: (4)               Stack arrays in sequence depth wise (along third axis).
501: (4)               This is equivalent to concatenation along the third axis after 2-D arrays
502: (4)               of shape `(M,N)` have been reshaped to `(M,N,1)` and 1-D arrays of shape
503: (4)               `(N,)` have been reshaped to `(1,N,1)`. Rebuilds arrays divided by
504: (4)               `dsplit`.
505: (4)               This function makes most sense for arrays with up to 3 dimensions. For
506: (4)               instance, for pixel-data with a height (first axis), width (second axis),
507: (4)               and r/g/b channels (third axis). The functions `concatenate`, `stack` and
508: (4)               `block` provide more general stacking and concatenation operations.
509: (4)               Parameters
510: (4)                 -----
511: (4)                 tup : sequence of arrays
512: (8)                   The arrays must have the same shape along all but the third axis.
513: (8)                   1-D or 2-D arrays must have the same shape.
514: (4)               Returns
515: (4)                 -----
516: (4)                 stacked : ndarray
517: (8)                   The array formed by stacking the given arrays, will be at least 3-D.
518: (4)               See Also
519: (4)                 -----
520: (4)                 concatenate : Join a sequence of arrays along an existing axis.
521: (4)                 stack : Join a sequence of arrays along a new axis.
522: (4)                 block : Assemble an nd-array from nested lists of blocks.
523: (4)                 vstack : Stack arrays in sequence vertically (row wise).
524: (4)                 hstack : Stack arrays in sequence horizontally (column wise).
525: (4)                 column_stack : Stack 1-D arrays as columns into a 2-D array.

```

```

526: (4)                         dsplit : Split array along third axis.
527: (4)                         Examples
528: (4)                         -----
529: (4)                         >>> a = np.array((1,2,3))
530: (4)                         >>> b = np.array((2,3,4))
531: (4)                         >>> np.dstack((a,b))
532: (4)                         array([[[1, 2],
533: (12)                           [2, 3],
534: (12)                           [3, 4]]])
535: (4)                         >>> a = np.array([[1],[2],[3]])
536: (4)                         >>> b = np.array([[2],[3],[4]])
537: (4)                         >>> np.dstack((a,b))
538: (4)                         array([[[1, 2],
539: (11)                           [[2, 3]],
540: (11)                           [[3, 4]]])
541: (4)                         """
542: (4)                         arrs = atleast_3d(*tup)
543: (4)                         if not isinstance(arrs, list):
544: (8)                           arrs = [arrs]
545: (4)                         return _nx.concatenate(arrs, 2)
546: (0)                         def _replace_zero_by_x_arrays(sub_arys):
547: (4)                           for i in range(len(sub_arys)):
548: (8)                             if _nx.ndim(sub_arys[i]) == 0:
549: (12)                               sub_arys[i] = _nx.empty(0, dtype=sub_arys[i].dtype)
550: (8)                             elif _nx.sometrue(_nx.equal(_nx.shape(sub_arys[i]), 0)):
551: (12)                               sub_arys[i] = _nx.empty(0, dtype=sub_arys[i].dtype)
552: (4)                           return sub_arys
553: (0)                         def _array_split_dispatcher(ary, indices_or_sections, axis=None):
554: (4)                           return (ary, indices_or_sections)
555: (0)                         @array_function_dispatch(_array_split_dispatcher)
556: (0)                         def array_split(ary, indices_or_sections, axis=0):
557: (4)                         """
558: (4)                         Split an array into multiple sub-arrays.
559: (4)                         Please refer to the ``split`` documentation. The only difference
560: (4)                         between these functions is that ``array_split`` allows
561: (4)                         `indices_or_sections` to be an integer that does *not* equally
562: (4)                         divide the axis. For an array of length l that should be split
563: (4)                         into n sections, it returns l % n sub-arrays of size l//n + 1
564: (4)                         and the rest of size l//n.
565: (4)                         See Also
566: (4)                         -----
567: (4)                         split : Split array into multiple sub-arrays of equal size.
568: (4)                         Examples
569: (4)                         -----
570: (4)                         >>> x = np.arange(8.0)
571: (4)                         >>> np.array_split(x, 3)
572: (4)                         [array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7.])]
573: (4)                         >>> x = np.arange(9)
574: (4)                         >>> np.array_split(x, 4)
575: (4)                         [array([0, 1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]
576: (4)                         """
577: (4)                         try:
578: (8)                           Ntotal = ary.shape[axis]
579: (4)                         except AttributeError:
580: (8)                           Ntotal = len(ary)
581: (4)                         try:
582: (8)                           Nsections = len(indices_or_sections) + 1
583: (8)                           div_points = [0] + list(indices_or_sections) + [Ntotal]
584: (4)                         except TypeError:
585: (8)                           Nsections = int(indices_or_sections)
586: (8)                           if Nsections <= 0:
587: (12)                             raise ValueError('number sections must be larger than 0.') from
None
588: (8)                           Neach_section, extras = divmod(Ntotal, Nsections)
589: (8)                           section_sizes = ([0] +
590: (25)                             extras * [Neach_section+1] +
591: (25)                             (Nsections-extras) * [Neach_section])
592: (8)                           div_points = _nx.array(section_sizes, dtype=_nx.intp).cumsum()
593: (4)                           sub_arys = []

```

```

594: (4)             sary = _nx.swapaxes(ary, axis, 0)
595: (4)             for i in range(Nsections):
596: (8)                 st = div_points[i]
597: (8)                 end = div_points[i + 1]
598: (8)                 sub_arys.append(_nx.swapaxes(sary[st:end], axis, 0))
599: (4)             return sub_arys
600: (0)         def _split_dispatcher(ary, indices_or_sections, axis=None):
601: (4)             return (ary, indices_or_sections)
602: (0)         @array_function_dispatch(_split_dispatcher)
603: (0)         def split(ary, indices_or_sections, axis=0):
604: (4)             """
605: (4)             Split an array into multiple sub-arrays as views into `ary`.
606: (4)             Parameters
607: (4)             -----
608: (4)             ary : ndarray
609: (8)                 Array to be divided into sub-arrays.
610: (4)             indices_or_sections : int or 1-D array
611: (8)                 If `indices_or_sections` is an integer, N, the array will be divided
612: (8)                 into N equal arrays along `axis`. If such a split is not possible,
613: (8)                 an error is raised.
614: (8)                 If `indices_or_sections` is a 1-D array of sorted integers, the
entries
615: (8)                 indicate where along `axis` the array is split. For example,
616: (8)                 ``[2, 3]`` would, for ``axis=0``, result in
617: (10)                   - ary[:2]
618: (10)                   - ary[2:3]
619: (10)                   - ary[3:]
620: (8)                 If an index exceeds the dimension of the array along `axis`,
621: (8)                 an empty sub-array is returned correspondingly.
axis : int, optional
622: (4)             The axis along which to split, default is 0.
Returns
623: (4)             -----
624: (4)             sub-arrays : list of ndarrays
625: (4)                 A list of sub-arrays as views into `ary`.
Raises
626: (4)             -----
627: (4)             ValueError
628: (4)                 If `indices_or_sections` is given as an integer, but
629: (4)                 a split does not result in equal division.
See Also
630: (4)             -----
631: (4)             array_split : Split an array into multiple sub-arrays of equal or
632: (18)                 near-equal size. Does not raise an exception if
633: (18)                 an equal division cannot be made.
634: (4)             hsplit : Split array into multiple sub-arrays horizontally (column-wise).
635: (4)             vsplit : Split array into multiple sub-arrays vertically (row wise).
636: (4)             dsplit : Split array into multiple sub-arrays along the 3rd axis (depth).
637: (4)             concatenate : Join a sequence of arrays along an existing axis.
638: (4)             stack : Join a sequence of arrays along a new axis.
639: (4)             hstack : Stack arrays in sequence horizontally (column wise).
640: (4)             vstack : Stack arrays in sequence vertically (row wise).
641: (4)             dstack : Stack arrays in sequence depth wise (along third dimension).
Examples
642: (4)             -----
643: (4)             >>> x = np.arange(9.0)
644: (4)             >>> np.split(x, 3)
645: (4)             [array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7., 8.])]
646: (4)             -----
647: (4)             try:
648: (4)                 len(indices_or_sections)
649: (4)             except TypeError:
650: (4)                 """
651: (4)             """
652: (4)             """
653: (4)             """
654: (5)             """
655: (5)             """
656: (5)             """
657: (5)             """
658: (4)             """
659: (4)             """
660: (8)             """
661: (4)             """

```

```

662: (8)             sections = indices_or_sections
663: (8)             N = ary.shape[axis]
664: (8)             if N % sections:
665: (12)                raise ValueError(
666: (16)                    "'array split does not result in an equal division'") from None
667: (4)            return array_split(ary, indices_or_sections, axis)
668: (0)        def _hvdsplit_dispatcher(ary, indices_or_sections):
669: (4)            return (ary, indices_or_sections)
670: (0)        @array_function_dispatch(_hvdsplit_dispatcher)
671: (0)        def hsplit(ary, indices_or_sections):
672: (4)            """
673: (4)                Split an array into multiple sub-arrays horizontally (column-wise).
674: (4)                Please refer to the `split` documentation. `hsplit` is equivalent
675: (4)                to `split` with ``axis=1``, the array is always split along the second
676: (4)                axis except for 1-D arrays, where it is split at ``axis=0``.
677: (4)                See Also
678: (4)                -----
679: (4)                split : Split an array into multiple sub-arrays of equal size.
680: (4)            Examples
681: (4)            -----
682: (4)            >>> x = np.arange(16.0).reshape(4, 4)
683: (4)            >>> x
684: (4)            array([[ 0.,   1.,   2.,   3.],
685: (11)              [ 4.,   5.,   6.,   7.],
686: (11)              [ 8.,   9.,  10.,  11.],
687: (11)              [12.,  13.,  14.,  15.]])
688: (4)            >>> np.hsplit(x, 2)
689: (4)            [array([[ 0.,   1.],
690: (11)              [ 4.,   5.],
691: (11)              [ 8.,   9.],
692: (11)              [12.,  13.]]),
693: (5)            array([[ 2.,   3.],
694: (11)              [ 6.,   7.],
695: (11)              [10.,  11.],
696: (11)              [14.,  15.]])]
697: (4)            >>> np.hsplit(x, np.array([3, 6]))
698: (4)            [array([[ 0.,   1.,   2.],
699: (11)              [ 4.,   5.,   6.],
700: (11)              [ 8.,   9.,  10.],
701: (11)              [12.,  13.,  14.]]),
702: (5)            array([[ 3.],
703: (11)              [ 7.],
704: (11)              [11.],
705: (11)              [15.]]),
706: (5)            array([], shape=(4, 0), dtype=float64)]
707: (4)            With a higher dimensional array the split is still along the second axis.
708: (4)            >>> x = np.arange(8.0).reshape(2, 2, 2)
709: (4)            >>> x
710: (4)            array([[[[0.,  1.],
711: (12)              [ 2.,  3.]],
712: (11)              [[4.,  5.],
713: (12)              [ 6.,  7.]]]])
714: (4)            >>> np.hsplit(x, 2)
715: (4)            [array([[[0.,  1.]],
716: (11)              [[4.,  5.]]]),
717: (5)            array([[[2.,  3.]],
718: (11)              [[6.,  7.]]])]
719: (4)            With a 1-D array, the split is along axis 0.
720: (4)            >>> x = np.array([0, 1, 2, 3, 4, 5])
721: (4)            >>> np.hsplit(x, 2)
722: (4)            [array([0, 1, 2]), array([3, 4, 5])]
723: (4)            """
724: (4)            if _nx.ndim(ary) == 0:
725: (8)                raise ValueError('hsplit only works on arrays of 1 or more
dimensions')
726: (4)            if ary.ndim > 1:
727: (8)                return split(ary, indices_or_sections, 1)
728: (4)            else:
729: (8)                return split(ary, indices_or_sections, 0)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

730: (0)
731: (0)
732: (4)
733: (4)      Split an array into multiple sub-arrays vertically (row-wise).
734: (4)      Please refer to the ``split`` documentation. ``vsplit`` is equivalent
735: (4)      to ``split`` with `axis=0` (default), the array is always split along the
736: (4)      first axis regardless of the array dimension.
737: (4)      See Also
738: (4)
739: (4)      -----
740: (4)      split : Split an array into multiple sub-arrays of equal size.
741: (4)      Examples
742: (4)
743: (4)
744: (4)
745: (11)
746: (11)
747: (11)
748: (4)
749: (4)
750: (11)
751: (11)
752: (4)
753: (4)
754: (11)
755: (11)
shape=(0, 4), dtype=float64]
756: (4)      With a higher dimensional array the split is still along the first axis.
757: (4)
758: (4)
759: (4)
760: (12)
761: (11)
762: (12)
763: (4)
764: (4)
765: (12)
766: (12)
767: (4)
768: (4)
769: (8)
dimensions')
770: (4)
771: (0)
772: (0)
773: (4)
774: (4)      Split array into multiple sub-arrays along the 3rd axis (depth).
775: (4)      Please refer to the `split` documentation. `dsplit` is equivalent
776: (4)      to `split` with ``axis=2``, the array is always split along the third
777: (4)      axis provided the array dimension is greater than or equal to 3.
778: (4)      See Also
779: (4)
780: (4)
781: (4)
782: (4)
783: (4)
784: (4)
785: (4)
786: (12)
787: (11)
788: (12)
789: (4)
790: (4)
791: (12)
792: (11)
793: (12)
794: (12)
795: (11)
796: (12)

```

```

797: (4)
798: (4)
799: (12)
800: (11)
801: (12)
802: (5)
803: (12)
804: (11)
805: (12)
806: (4)
807: (4)
808: (4)
809: (8)
dimensions')
810: (4)
811: (0)
812: (4)
813: (4)
814: (4)
815: (4)
816: (17)
817: (35)
818: (4)
819: (8)
820: (4)
821: (0)
822: (4)
823: (4)
824: (4)
825: (4)
826: (17)
827: (35)
828: (4)
829: (8)
830: (4)
831: (0)
832: (4)
833: (0)
834: (0)
835: (4)
836: (4)
837: (4)
838: (4)
839: (4)
840: (4)
841: (4)
842: (4)
843: (4)
844: (4)
845: (4)
846: (4)
847: (4)
848: (4)
849: (4)
850: (4)
851: (4)
852: (4)
853: (4)
854: (4)
855: (4)
856: (8)
857: (4)
858: (8)
859: (4)
860: (8)
861: (9)
862: (9)
863: (4)
864: (4)

    >>> np.dsplit(x, np.array([3, 6]))
    [array([[ 0.,   1.,   2.],
           [ 4.,   5.,   6.],
           [ 8.,   9.,  10.],
           [12.,  13.,  14.]]),
     array([[ 3.],
           [ 7.],
           [11.],
           [15.]]),
     array([], shape=(2, 2, 0), dtype=float64)]
    """
    if _nx.ndim(ary) < 3:
        raise ValueError('dsplit only works on arrays of 3 or more
dimensions')

    return split(ary, indices_or_sections, 2)
def get_array_prepare(*args):
    """Find the wrapper for the array with the highest priority.
    In case of ties, leftmost wins. If no wrapper is found, return None
    """
    wrappers = sorted((getattr(x, '__array_priority__', 0), -i,
                       x.__array_prepare__) for i, x in enumerate(args)
                      if hasattr(x, '__array_prepare__'))

    if wrappers:
        return wrappers[-1][-1]
    return None
def get_array_wrap(*args):
    """Find the wrapper for the array with the highest priority.
    In case of ties, leftmost wins. If no wrapper is found, return None
    """
    wrappers = sorted((getattr(x, '__array_priority__', 0), -i,
                       x.__array_wrap__) for i, x in enumerate(args)
                      if hasattr(x, '__array_wrap__'))

    if wrappers:
        return wrappers[-1][-1]
    return None
def _kron_dispatcher(a, b):
    return (a, b)
@array_function_dispatch(_kron_dispatcher)
def kron(a, b):
    """
    Kronecker product of two arrays.
    Computes the Kronecker product, a composite array made of blocks of the
    second array scaled by the first.
    Parameters
    -----
    a, b : array_like
    Returns
    -----
    out : ndarray
    See Also
    -----
    outer : The outer product
    Notes
    -----
    The function assumes that the number of dimensions of `a` and `b`
    are the same, if necessary prepending the smallest with ones.
    If ``a.shape = (r0,r1,...,rN)`` and ``b.shape = (s0,s1,...,sN)``,
    the Kronecker product has shape ``(r0*s0, r1*s1, ..., rN*sN)``.
    The elements are products of elements from `a` and `b`, organized
    explicitly by::
        kron(a,b)[k0,k1,...,kN] = a[i0,i1,...,iN] * b[j0,j1,...,jN]
    where::
        kt = it * st + jt, t = 0,...,N
    In the common 2-D case (N=1), the block structure can be visualized::
        [[ a[0,0]*b, a[0,1]*b, ... , a[0,-1]*b ],
         [ ...           ...       ],
         [ a[-1,0]*b, a[-1,1]*b, ... , a[-1,-1]*b ]]
    Examples
    -----
    
```

```

865: (4)          >>> np.kron([1,10,100], [5,6,7])
866: (4)          array([ 5, 6, 7, ..., 500, 600, 700])
867: (4)          >>> np.kron([5,6,7], [1,10,100])
868: (4)          array([ 5, 50, 500, ..., 7, 70, 700])
869: (4)          >>> np.kron(np.eye(2), np.ones((2,2)))
870: (4)          array([[1., 1., 0., 0.],
871: (11)           [1., 1., 0., 0.],
872: (11)           [0., 0., 1., 1.],
873: (11)           [0., 0., 1., 1.]])
874: (4)          >>> a = np.arange(100).reshape((2,5,2,5))
875: (4)          >>> b = np.arange(24).reshape((2,3,4))
876: (4)          >>> c = np.kron(a,b)
877: (4)          >>> c.shape
878: (4)          (2, 10, 6, 20)
879: (4)          >>> I = (1,3,0,2)
880: (4)          >>> J = (0,2,1)
881: (4)          >>> J1 = (0,) + J          # extend to ndim=4
882: (4)          >>> S1 = (1,) + b.shape
883: (4)          >>> K = tuple(np.array(I) * np.array(S1) + np.array(J1))
884: (4)          >>> c[K] == a[I]*b[J]
885: (4)          True
886: (4)          """
887: (4)          b = asanyarray(b)
888: (4)          a = array(a, copy=False, subok=True, ndmin=b.ndim)
889: (4)          is_any_mat = isinstance(a, matrix) or isinstance(b, matrix)
890: (4)          ndb, nda = b.ndim, a.ndim
891: (4)          nd = max(ndb, nda)
892: (4)          if (nda == 0 or ndb == 0):
893: (8)              return _nx.multiply(a, b)
894: (4)          as_ = a.shape
895: (4)          bs = b.shape
896: (4)          if not a.flags.contiguous:
897: (8)              a = reshape(a, as_)
898: (4)          if not b.flags.contiguous:
899: (8)              b = reshape(b, bs)
900: (4)          as_ = (1,)*max(0, ndb-nda) + as_
901: (4)          bs = (1,)*max(0, nda-ndb) + bs
902: (4)          a_arr = expand_dims(a, axis=tuple(range(ndb-nda)))
903: (4)          b_arr = expand_dims(b, axis=tuple(range(ndb-nda)))
904: (4)          a_arr = expand_dims(a_arr, axis=tuple(range(1, nd*2, 2)))
905: (4)          b_arr = expand_dims(b_arr, axis=tuple(range(0, nd*2, 2)))
906: (4)          result = _nx.multiply(a_arr, b_arr, subok=(not is_any_mat))
907: (4)          result = result.reshape(_nx.multiply(as_, bs))
908: (4)          return result if not is_any_mat else matrix(result, copy=False)
909: (0)          def _tile_dispatcher(A, reps):
910: (4)              return (A, reps)
911: (0)          @array_function_dispatch(_tile_dispatcher)
912: (0)          def tile(A, reps):
913: (4)              """
914: (4)              Construct an array by repeating A the number of times given by reps.
915: (4)              If `reps` has length ``d``, the result will have dimension of
916: (4)              ``max(d, A.ndim)``.
917: (4)              If ``A.ndim < d``, `A` is promoted to be d-dimensional by prepending new
918: (4)              axes. So a shape (3,) array is promoted to (1, 3) for 2-D replication,
919: (4)              or shape (1, 1, 3) for 3-D replication. If this is not the desired
920: (4)              behavior, promote `A` to d-dimensions manually before calling this
921: (4)              function.
922: (4)              If ``A.ndim > d``, `reps` is promoted to `A`.ndim by pre-pending 1's to
923: (4)              it.
924: (4)              Thus for an `A` of shape (2, 3, 4, 5), a `reps` of (2, 2) is treated as
925: (4)              (1, 1, 2, 2).
926: (4)              Note : Although tile may be used for broadcasting, it is strongly
927: (4)              recommended to use numpy's broadcasting operations and functions.
928: (4)              Parameters
929: (4)              -----
930: (8)              A : array_like
931: (4)                  The input array.
932: (8)              reps : array_like
933: (4)                  The number of repetitions of `A` along each axis.

```

```

933: (4)          Returns
934: (4)          -----
935: (4)          c : ndarray
936: (8)          The tiled output array.
937: (4)          See Also
938: (4)          -----
939: (4)          repeat : Repeat elements of an array.
940: (4)          broadcast_to : Broadcast an array to a new shape
941: (4)          Examples
942: (4)          -----
943: (4)          >>> a = np.array([0, 1, 2])
944: (4)          >>> np.tile(a, 2)
945: (4)          array([0, 1, 2, 0, 1, 2])
946: (4)          >>> np.tile(a, (2, 2))
947: (4)          array([[0, 1, 2, 0, 1, 2],
948: (11)           [0, 1, 2, 0, 1, 2]])
949: (4)          >>> np.tile(a, (2, 1, 2))
950: (4)          array([[[0, 1, 2, 0, 1, 2]],
951: (11)           [[0, 1, 2, 0, 1, 2]]])
952: (4)          >>> b = np.array([[1, 2], [3, 4]])
953: (4)          >>> np.tile(b, 2)
954: (4)          array([[1, 2, 1, 2],
955: (11)           [3, 4, 3, 4]])
956: (4)          >>> np.tile(b, (2, 1))
957: (4)          array([[1, 2],
958: (11)           [3, 4],
959: (11)           [1, 2],
960: (11)           [3, 4]])
961: (4)          >>> c = np.array([1, 2, 3, 4])
962: (4)          >>> np.tile(c, (4, 1))
963: (4)          array([[1, 2, 3, 4],
964: (11)           [1, 2, 3, 4],
965: (11)           [1, 2, 3, 4],
966: (11)           [1, 2, 3, 4]])
967: (4)          """
968: (4)          try:
969: (8)              tup = tuple(rep)
970: (4)          except TypeError:
971: (8)              tup = (rep,)
972: (4)          d = len(tup)
973: (4)          if all(x == 1 for x in tup) and isinstance(A, _nx.ndarray):
974: (8)              return _nx.array(A, copy=True, subok=True, ndmin=d)
975: (4)          else:
976: (8)              c = _nx.array(A, copy=False, subok=True, ndmin=d)
977: (4)          if (d < c.ndim):
978: (8)              tup = (1,) * (c.ndim - d) + tup
979: (4)          shape_out = tuple(s * t for s, t in zip(c.shape, tup))
980: (4)          n = c.size
981: (4)          if n > 0:
982: (8)              for dim_in, nrep in zip(c.shape, tup):
983: (12)                  if nrep != 1:
984: (16)                      c = c.reshape(-1, n).repeat(nrep, 0)
985: (12)                  n //= dim_in
986: (4)          return c.reshape(shape_out)

-----

```

## File 219 - stride\_tricks.py:

```

1: (0)          """
2: (0)          Utilities that manipulate strides to achieve desirable effects.
3: (0)          An explanation of strides can be found in the "ndarray.rst" file in the
4: (0)          NumPy reference guide.
5: (0)          """
6: (0)          import numpy as np
7: (0)          from numpy.core.numeric import normalize_axis_tuple
8: (0)          from numpy.core.overrides import array_function_dispatch, set_module
9: (0)          __all__ = ['broadcast_to', 'broadcast_arrays', 'broadcast_shapes']
10: (0)         class DummyArray:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

11: (4)         """Dummy object that just exists to hang __array_interface__ dictionaries
12: (4)         and possibly keep alive a reference to a base array.
13: (4)
14: (4)         def __init__(self, interface, base=None):
15: (8)             self.__array_interface__ = interface
16: (8)             self.base = base
17: (0)         def _maybe_view_as_subclass(original_array, new_array):
18: (4)             if type(original_array) is not type(new_array):
19: (8)                 new_array = new_array.view(type=type(original_array))
20: (8)                 if new_array.__array_finalize__:
21: (12)                     new_array.__array_finalize__(original_array)
22: (4)             return new_array
23: (0)         def as_strided(x, shape=None, strides=None, subok=False, writeable=True):
24: (4)             """
25: (4)             Create a view into the array with the given shape and strides.
26: (4)             .. warning:: This function has to be used with extreme care, see notes.
27: (4)             Parameters
28: (4)             -----
29: (4)             x : ndarray
30: (8)                 Array to create a new.
31: (4)             shape : sequence of int, optional
32: (8)                 The shape of the new array. Defaults to ``x.shape``.
33: (4)             strides : sequence of int, optional
34: (8)                 The strides of the new array. Defaults to ``x.strides``.
35: (4)             subok : bool, optional
36: (8)                 .. versionadded:: 1.10
37: (8)                 If True, subclasses are preserved.
38: (4)             writeable : bool, optional
39: (8)                 .. versionadded:: 1.12
40: (8)                 If set to False, the returned array will always be readonly.
41: (8)                 Otherwise it will be writable if the original array was. It
42: (8)                 is advisable to set this to False if possible (see Notes).
43: (4)             Returns
44: (4)             -----
45: (4)             view : ndarray
46: (4)             See also
47: (4)             -----
48: (4)             broadcast_to : broadcast an array to a given shape.
49: (4)             reshape : reshape an array.
50: (4)             lib.stride_tricks.sliding_window_view :
51: (8)                 userfriendly and safe function for the creation of sliding window
views.
52: (4)             Notes
53: (4)             -----
54: (4)             ``as_strided`` creates a view into the array given the exact strides
55: (4)             and shape. This means it manipulates the internal data structure of
56: (4)             ndarray and, if done incorrectly, the array elements can point to
57: (4)             invalid memory and can corrupt results or crash your program.
58: (4)             It is advisable to always use the original ``x.strides`` when
59: (4)             calculating new strides to avoid reliance on a contiguous memory
60: (4)             layout.
61: (4)             Furthermore, arrays created with this function often contain self
62: (4)             overlapping memory, so that two elements are identical.
63: (4)             Vectorized write operations on such arrays will typically be
64: (4)             unpredictable. They may even give different results for small, large,
65: (4)             or transposed arrays.
66: (4)             Since writing to these arrays has to be tested and done with great
67: (4)             care, you may want to use ``writeable=False`` to avoid accidental write
68: (4)             operations.
69: (4)             For these reasons it is advisable to avoid ``as_strided`` when
70: (4)             possible.
71: (4)             """
72: (4)             x = np.array(x, copy=False, subok=subok)
73: (4)             interface = dict(x.__array_interface__)
74: (4)             if shape is not None:
75: (8)                 interface['shape'] = tuple(shape)
76: (4)             if strides is not None:
77: (8)                 interface['strides'] = tuple(strides)
78: (4)             array = np.asarray(DummyArray(interface, base=x))

```

```

79: (4)             array.dtype = x.dtype
80: (4)             view = _maybe_view_as_subclass(x, array)
81: (4)             if view.flags.writeable and not writeable:
82: (8)                 view.flags.writeable = False
83: (4)             return view
84: (0)         def _sliding_window_view_dispatcher(x, window_shape, axis=None, *,
85: (36)                           subok=None, writeable=None):
86: (4)             return (x,)
87: (0)         @array_function_dispatch(_sliding_window_view_dispatcher)
88: (0)         def sliding_window_view(x, window_shape, axis=None, *,
89: (24)                           subok=False, writeable=False):
90: (4)             """
91: (4)             Create a sliding window view into the array with the given window shape.
92: (4)             Also known as rolling or moving window, the window slides across all
93: (4)             dimensions of the array and extracts subsets of the array at all window
94: (4)             positions.
95: (4)             .. versionadded:: 1.20.0
96: (4)         Parameters
97: (4)             -----
98: (4)             x : array_like
99: (8)             Array to create the sliding window view from.
100: (4)             window_shape : int or tuple of int
101: (8)             Size of window over each axis that takes part in the sliding window.
102: (8)             If `axis` is not present, must have same length as the number of input
103: (8)             array dimensions. Single integers `i` are treated as if they were the
104: (8)             tuple `(i,)`.
105: (4)             axis : int or tuple of int, optional
106: (8)             Axis or axes along which the sliding window is applied.
107: (8)             By default, the sliding window is applied to all axes and
108: (8)             `window_shape[i]` will refer to axis `i` of `x`.
109: (8)             If `axis` is given as a `tuple of int`, `window_shape[i]` will refer
to
110: (8)             the axis `axis[i]` of `x`.
111: (8)             Single integers `i` are treated as if they were the tuple `(i,)`.
112: (4)             subok : bool, optional
113: (8)             If True, sub-classes will be passed-through, otherwise the returned
114: (8)             array will be forced to be a base-class array (default).
115: (4)             writeable : bool, optional
116: (8)             When true, allow writing to the returned view. The default is false,
117: (8)             as this should be used with caution: the returned view contains the
118: (8)             same memory location multiple times, so writing to one location will
119: (8)             cause others to change.
120: (4)         Returns
121: (4)             -----
122: (4)             view : ndarray
123: (8)             Sliding window view of the array. The sliding window dimensions are
124: (8)             inserted at the end, and the original dimensions are trimmed as
125: (8)             required by the size of the sliding window.
126: (8)             That is, ``view.shape = x_shape_trimmed + window_shape``, where
127: (8)             ``x_shape_trimmed`` is ``x.shape`` with every entry reduced by one
less
128: (8)             than the corresponding window size.
129: (4)         See Also
130: (4)             -----
131: (4)             lib.stride_tricks.as_strided: A lower-level and less safe routine for
132: (8)                 creating arbitrary views from custom shape and strides.
133: (4)             broadcast_to: broadcast an array to a given shape.
134: (4)         Notes
135: (4)             -----
136: (4)             For many applications using a sliding window view can be convenient, but
137: (4)             potentially very slow. Often specialized solutions exist, for example:
138: (4)             - `scipy.signal.fftconvolve`
139: (4)             - filtering functions in `scipy.ndimage`
140: (4)             - moving window functions provided by
141: (6)                 `bottleneck <https://github.com/pydata/bottleneck>`_.
142: (4)             As a rough estimate, a sliding window approach with an input size of `N`
143: (4)             and a window size of `W` will scale as `O(N*W)` where frequently a special
144: (4)             algorithm can achieve `O(N)`. That means that the sliding window variant
for a window size of 100 can be a 100 times slower than a more specialized
145: (4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

146: (4)           version.
147: (4)           Nevertheless, for small window sizes, when no custom algorithm exists, or
148: (4)           as a prototyping and developing tool, this function can be a good
solution.
149: (4)           Examples
150: (4)           -----
151: (4)           >>> x = np.arange(6)
152: (4)           >>> x.shape
153: (4)           (6,)
154: (4)           >>> v = sliding_window_view(x, 3)
155: (4)           >>> v.shape
156: (4)           (4, 3)
157: (4)           >>> v
158: (4)           array([[0, 1, 2],
159: (11)             [1, 2, 3],
160: (11)             [2, 3, 4],
161: (11)             [3, 4, 5]])
162: (4)           This also works in more dimensions, e.g.
163: (4)           >>> i, j = np.ogrid[:3, :4]
164: (4)           >>> x = 10*i + j
165: (4)           >>> x.shape
166: (4)           (3, 4)
167: (4)           >>> x
168: (4)           array([[ 0,  1,  2,  3],
169: (11)             [10, 11, 12, 13],
170: (11)             [20, 21, 22, 23]])
171: (4)           >>> shape = (2,2)
172: (4)           >>> v = sliding_window_view(x, shape)
173: (4)           >>> v.shape
174: (4)           (2, 3, 2, 2)
175: (4)           >>> v
176: (4)           array([[[[ 0,  1,
177: (13)             [10, 11],
178: (12)               [[ 1,  2],
179: (13)                 [11, 12]],
180: (12)                 [[ 2,  3],
181: (13)                   [12, 13]]],
182: (11)                   [[[10, 11],
183: (13)                     [20, 21]],
184: (12)                     [[11, 12],
185: (13)                       [21, 22]],
186: (12)                       [[12, 13],
187: (13)                         [22, 23]]]])
188: (4)           The axis can be specified explicitly:
189: (4)           >>> v = sliding_window_view(x, 3, 0)
190: (4)           >>> v.shape
191: (4)           (1, 4, 3)
192: (4)           >>> v
193: (4)           array([[[ 0, 10, 20],
194: (12)             [ 1, 11, 21],
195: (12)             [ 2, 12, 22],
196: (12)             [ 3, 13, 23]]])
197: (4)           The same axis can be used several times. In that case, every use reduces
198: (4)           the corresponding original dimension:
199: (4)           >>> v = sliding_window_view(x, (2, 3), (1, 1))
200: (4)           >>> v.shape
201: (4)           (3, 1, 2, 3)
202: (4)           >>> v
203: (4)           array([[[[ 0,  1,  2],
204: (13)             [ 1,  2,  3]]],
205: (11)               [[[10, 11, 12],
206: (13)                 [11, 12, 13]]],
207: (11)                 [[[20, 21, 22],
208: (13)                   [21, 22, 23]]]])
209: (4)           Combining with stepped slicing (`::step`), this can be used to take
sliding
210: (4)           views which skip elements:
211: (4)           >>> x = np.arange(7)
212: (4)           >>> sliding_window_view(x, 5)[:, ::2]
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

213: (4)
214: (11)
215: (11)
216: (4)
217: (4)
218: (4)
219: (4)
220: (11)
221: (11)
222: (4)
running
223: (4)
224: (4)
225: (4)
226: (4)
227: (4)
228: (4)
229: (4)
230: (4)
231: (4)
232: (4)
233: (11)
234: (11)
235: (11)
236: (4)
237: (4)
238: (4)
239: (4)
240: (4)
Note that a sliding window approach is often **not** optimal (see Notes).
"""
241: (4)
242: (20)
243: (20)
244: (4)
245: (4)
246: (4)
247: (8)
248: (4)
249: (8)
250: (8)
251: (12)
252: (29)
253: (29)
254: (29)
255: (4)
256: (8)
257: (8)
258: (12)
259: (29)
260: (29)
261: (4)
262: (4)
263: (4)
264: (8)
265: (12)
266: (16)
267: (8)
268: (4)
269: (4)
270: (22)
def _broadcast_to(array, shape, subok, readonly):
    shape = tuple(shape) if np.iterable(shape) else (shape,)
    array = np.array(array, copy=False, subok=subok)
    if not shape and array.shape:
        raise ValueError('cannot broadcast a non-scalar to a scalar array')
    if any(size < 0 for size in shape):
        raise ValueError('all elements of broadcast shape must be non-'
                         'negative')
    extras = []
    it = np.nditer(
        array([[[0, 2, 4],
               [1, 3, 5],
               [2, 4, 6]]]
            or views which move by multiple elements
        >>> x = np.arange(7)
        >>> sliding_window_view(x, 3)[::2, :]
        array([[0, 1, 2],
               [2, 3, 4],
               [4, 5, 6]])
A common application of `sliding_window_view` is the calculation of
statistics. The simplest example is the
`moving average <https://en.wikipedia.org/wiki/Moving\_average>`_:
>>> x = np.arange(6)
>>> x.shape
(6,)
>>> v = sliding_window_view(x, 3)
>>> v.shape
(4, 3)
>>> v
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
>>> moving_average = v.mean(axis=-1)
>>> moving_average
array([1., 2., 3., 4.])
window_shape = (tuple(window_shape)
                if np.iterable(window_shape)
                else (window_shape,))
x = np.array(x, copy=False, subok=subok)
window_shape_array = np.array(window_shape)
if np.any(window_shape_array < 0):
    raise ValueError(`window_shape` cannot contain negative values')
if axis is None:
    axis = tuple(range(x.ndim))
    if len(window_shape) != len(axis):
        raise ValueError(f'Since axis is `None`, must provide '
                        f'window_shape for all dimensions of `x`; '
                        f'got {len(window_shape)} window_shape elements '
                        f'and `x.ndim` is {x.ndim}.')
else:
    axis = normalize_axis_tuple(axis, x.ndim, allow_duplicate=True)
    if len(window_shape) != len(axis):
        raise ValueError(f'Must provide matching length window_shape and '
                        f'axis; got {len(window_shape)} window_shape '
                        f'elements and {len(axis)} axes elements.')
out_strides = x.strides + tuple(x.strides[ax] for ax in axis)
x_shape_trimmed = list(x.shape)
for ax, dim in zip(axis, window_shape):
    if x_shape_trimmed[ax] < dim:
        raise ValueError(
            'window shape cannot be larger than input array shape')
    x_shape_trimmed[ax] -= dim - 1
out_shape = tuple(x_shape_trimmed) + window_shape
return as_strided(x, strides=out_strides, shape=out_shape,
                  subok=subok, writeable=writeable)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

281: (8)             (array,), flags=['multi_index', 'refs_ok', 'zerosize_ok'] + extras,
282: (8)             op_flags=['readonly'], itershape=shape, order='C')
283: (4)             with it:
284: (8)                 broadcast = it.itviews[0]
285: (4)                 result = _maybe_view_as_subclass(array, broadcast)
286: (4)                 if not readonly and array.flags._writeable_no_warn:
287: (8)                     result.flags.writeable = True
288: (8)                     result.flags._warn_on_write = True
289: (4)             return result
290: (0)         def _broadcast_to_dispatcher(array, shape, subok=None):
291: (4)             return (array,)
292: (0)     @array_function_dispatch(_broadcast_to_dispatcher, module='numpy')
293: (0)     def broadcast_to(array, shape, subok=False):
294: (4)         """Broadcast an array to a new shape.
295: (4)         Parameters
296: (4)         -----
297: (4)         array : array_like
298: (8)             The array to broadcast.
299: (4)         shape : tuple or int
300: (8)             The shape of the desired array. A single integer ``i`` is interpreted
301: (8)             as ``(i,)``.
302: (4)         subok : bool, optional
303: (8)             If True, then sub-classes will be passed-through, otherwise
304: (8)             the returned array will be forced to be a base-class array (default).
305: (4)     Returns
306: (4)         -----
307: (4)         broadcast : array
308: (8)             A readonly view on the original array with the given shape. It is
309: (8)             typically not contiguous. Furthermore, more than one element of a
310: (8)             broadcasted array may refer to a single memory location.
311: (4)     Raises
312: (4)         -----
313: (4)         ValueError
314: (8)             If the array is not compatible with the new shape according to NumPy's
315: (8)             broadcasting rules.
316: (4)     See Also
317: (4)         -----
318: (4)         broadcast
319: (4)         broadcast_arrays
320: (4)         broadcast_shapes
321: (4)     Notes
322: (4)         -----
323: (4)         .. versionadded:: 1.10.0
324: (4)     Examples
325: (4)         -----
326: (4)         >>> x = np.array([1, 2, 3])
327: (4)         >>> np.broadcast_to(x, (3, 3))
328: (4)         array([[1, 2, 3],
329: (11)             [1, 2, 3],
330: (11)             [1, 2, 3]])
331: (4)         """
332: (4)         return _broadcast_to(array, shape, subok=subok, readonly=True)
333: (0)     def _broadcast_shape(*args):
334: (4)         """Returns the shape of the arrays that would result from broadcasting the
335: (4)         supplied arrays against each other.
336: (4)         """
337: (4)         b = np.broadcast(*args[:32])
338: (4)         for pos in range(32, len(args), 31):
339: (8)             b = broadcast_to(b, b.shape)
340: (8)             b = np.broadcast(b, *args[pos:(pos + 31)])
341: (4)         return b.shape
342: (0)     @set_module('numpy')
343: (0)     def broadcast_shapes(*args):
344: (4)         """
345: (4)             Broadcast the input shapes into a single shape.
346: (4)             :ref:`Learn more about broadcasting here <basics.broadcasting>`.
347: (4)             .. versionadded:: 1.20.0
348: (4)             Parameters
349: (4)             -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

350: (4)           `*args` : tuples of ints, or ints
351: (8)           The shapes to be broadcast against each other.
352: (4)           Returns
353: (4)
354: (4)
355: (8)           tuple
356: (4)           Broadcasted shape.
357: (4)
358: (4)           Raises
359: (8)           -----
360: (8)           ValueError
361: (4)           If the shapes are not compatible and cannot be broadcast according
362: (4)           to NumPy's broadcasting rules.
363: (4)           See Also
364: (4)           -----
365: (4)           broadcast
366: (4)           broadcast_arrays
367: (4)           broadcast_to
368: (4)           Examples
369: (4)
370: (4)           >>> np.broadcast_shapes((1, 2), (3, 1), (3, 2))
371: (4)           (3, 2)
372: (4)           >>> np.broadcast_shapes((6, 7), (5, 6, 1), (7,), (5, 1, 7))
373: (4)           (5, 6, 7)
374: (4)           """
375: (0)           arrays = [np.empty(x, dtype=[]) for x in args]
376: (4)           return _broadcast_shape(*arrays)
377: (0)           def _broadcast_arrays_dispatcher(*args, subok=None):
378: (0)               return args
379: (4)           @array_function_dispatch(_broadcast_arrays_dispatcher, module='numpy')
380: (4)           def broadcast_arrays(*args, subok=False):
381: (4)               """
382: (4)               Broadcast any number of arrays against each other.
383: (4)               Parameters
384: (8)               `*args` : array_likes
385: (4)               The arrays to broadcast.
386: (8)               subok : bool, optional
387: (8)               If True, then sub-classes will be passed-through, otherwise
388: (8)               the returned arrays will be forced to be a base-class array (default).
389: (4)               Returns
390: (4)               broadcasted : list of arrays
391: (8)               These arrays are views on the original arrays. They are typically
392: (8)               not contiguous. Furthermore, more than one element of a
393: (8)               broadcasted array may refer to a single memory location. If you need
394: (8)               to write to the arrays, make copies first. While you can set the
395: (8)               ``writable`` flag True, writing to a single output value may end up
396: (8)               changing more than one location in the output array.
397: (8)               .. deprecated:: 1.17
398: (12)              The output is currently marked so that if written to, a
deprecation
399: (12)              warning will be emitted. A future version will set the
400: (12)              ``writable`` flag False so writing to it will raise an error.
401: (4)           See Also
402: (4)
403: (4)           broadcast
404: (4)           broadcast_to
405: (4)           broadcast_shapes
406: (4)           Examples
407: (4)
408: (4)           >>> x = np.array([[1,2,3]])
409: (4)           >>> y = np.array([[4],[5]])
410: (4)           >>> np.broadcast_arrays(x, y)
411: (4)           [array([[1, 2, 3],
412: (11)             [1, 2, 3]]), array([[4, 4, 4],
413: (11)               [5, 5, 5]])]
414: (4)           Here is a useful idiom for getting contiguous copies instead of
415: (4)           non-contiguous views.
416: (4)           >>> [np.array(a) for a in np.broadcast_arrays(x, y)]
417: (4)           [array([[1, 2, 3],

```

```

418: (11)             [1, 2, 3]]], array([[4, 4, 4],
419: (11)             [5, 5, 5]]])
420: (4)
421: (4)             """
422: (4)             args = [np.array(_m, copy=False, subok=subok) for _m in args]
423: (4)             shape = _broadcast_shape(*args)
424: (8)             if all(array.shape == shape for array in args):
425: (4)                 return args
426: (12)             return [_broadcast_to(array, shape, subok=subok, readonly=False)
427: (4)                         for array in args]

```

---

## File 220 - twodim\_base.py:

```

1: (0)             """ Basic functions for manipulating 2d arrays
2: (0)             """
3: (0)             import functools
4: (0)             import operator
5: (0)             from numpy.core.numeric import (
6: (4)                 asanyarray, arange, zeros, greater_equal, multiply, ones,
7: (4)                 asarray, where, int8, int16, int32, int64, intp, empty, promote_types,
8: (4)                 diagonal, nonzero, indices
9: (4)             )
10: (0)            from numpy.core.overrides import set_array_function_like_doc, set_module
11: (0)            from numpy.core import overrides
12: (0)            from numpy.core import iinfo
13: (0)            from numpy.lib.stride_tricks import broadcast_to
14: (0)            __all__ = [
15: (4)                'diag', 'diagflat', 'eye', 'fliplr', 'flipud', 'tri', 'triu',
16: (4)                'tril', 'vander', 'histogram2d', 'mask_indices', 'tril_indices',
17: (4)                'tril_indices_from', 'triu_indices', 'triu_indices_from', ]
18: (0)            array_function_dispatch = functools.partial(
19: (4)                overrides.array_function_dispatch, module='numpy')
20: (0)            i1 = iinfo(int8)
21: (0)            i2 = iinfo(int16)
22: (0)            i4 = iinfo(int32)
23: (0)            def _min_int(low, high):
24: (4)                """ get small int that fits the range """
25: (4)                if high <= i1.max and low >= i1.min:
26: (8)                    return int8
27: (4)                if high <= i2.max and low >= i2.min:
28: (8)                    return int16
29: (4)                if high <= i4.max and low >= i4.min:
30: (8)                    return int32
31: (4)                return int64
32: (0)            def _flip_dispatcher(m):
33: (4)                return (m,)
34: (0)            @array_function_dispatch(_flip_dispatcher)
35: (0)            def fliplr(m):
36: (4)                """
37: (4)                    Reverse the order of elements along axis 1 (left/right).
38: (4)                    For a 2-D array, this flips the entries in each row in the left/right
39: (4)                    direction. Columns are preserved, but appear in a different order than
40: (4)                    before.
41: (4)                    Parameters
42: (4)                    -----
43: (4)                    m : array_like
44: (8)                        Input array, must be at least 2-D.
45: (4)                    Returns
46: (4)                    -----
47: (4)                    f : ndarray
48: (8)                        A view of `m` with the columns reversed. Since a view
49: (8)                        is returned, this operation is :math:`\mathcal{O}(1)`.
50: (4)                    See Also
51: (4)                    -----
52: (4)                    flipud : Flip array in the up/down direction.
53: (4)                    flip : Flip array in one or more dimensions.
54: (4)                    rot90 : Rotate array counterclockwise.
55: (4)                    Notes

```

```

56: (4)      -----
57: (4)      Equivalent to ``m[:,::-1]`` or ``np.flip(m, axis=1)``.
58: (4)      Requires the array to be at least 2-D.
59: (4)      Examples
60: (4)      -----
61: (4)      >>> A = np.diag([1.,2.,3.])
62: (4)      >>> A
63: (4)      array([[1.,  0.,  0.],
64: (11)         [0.,  2.,  0.],
65: (11)         [0.,  0.,  3.]])
66: (4)      >>> np.fliplr(A)
67: (4)      array([[0.,  0.,  1.],
68: (11)         [0.,  2.,  0.],
69: (11)         [3.,  0.,  0.]])
70: (4)      >>> A = np.random.randn(2,3,5)
71: (4)      >>> np.all(np.fliplr(A) == A[:,::-1,...])
72: (4)      True
73: (4)      """
74: (4)      m = asanyarray(m)
75: (4)      if m.ndim < 2:
76: (8)          raise ValueError("Input must be >= 2-d.")
77: (4)      return m[:, ::-1]
78: (0)      @array_function_dispatch(_flip_dispatcher)
79: (0)      def flipud(m):
80: (4)          """
81: (4)          Reverse the order of elements along axis 0 (up/down).
82: (4)          For a 2-D array, this flips the entries in each column in the up/down
83: (4)          direction. Rows are preserved, but appear in a different order than
before.
84: (4)          Parameters
85: (4)          -----
86: (4)          m : array_like
87: (8)              Input array.
88: (4)          Returns
89: (4)          -----
90: (4)          out : array_like
91: (8)              A view of `m` with the rows reversed. Since a view is
92: (8)              returned, this operation is :math:`\mathcal{O}(1)` .
93: (4)          See Also
94: (4)          -----
95: (4)          fliplr : Flip array in the left/right direction.
96: (4)          flip : Flip array in one or more dimensions.
97: (4)          rot90 : Rotate array counterclockwise.
98: (4)          Notes
99: (4)          -----
100: (4)          Equivalent to ``m[::-1, ...]`` or ``np.flip(m, axis=0)``.
101: (4)          Requires the array to be at least 1-D.
102: (4)          Examples
103: (4)          -----
104: (4)          >>> A = np.diag([1.0, 2, 3])
105: (4)          >>> A
106: (4)          array([[1.,  0.,  0.],
107: (11)            [0.,  2.,  0.],
108: (11)            [0.,  0.,  3.]])
109: (4)          >>> np.flipud(A)
110: (4)          array([[0.,  0.,  3.],
111: (11)            [0.,  2.,  0.],
112: (11)            [1.,  0.,  0.]])
113: (4)          >>> A = np.random.randn(2,3,5)
114: (4)          >>> np.all(np.flipud(A) == A[::-1,...])
115: (4)          True
116: (4)          >>> np.flipud([1,2])
117: (4)          array([2, 1])
118: (4)          """
119: (4)          m = asanyarray(m)
120: (4)          if m.ndim < 1:
121: (8)              raise ValueError("Input must be >= 1-d.")
122: (4)              return m[::-1, ...]
123: (0)      @set_array_function_like_doc

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

124: (0) @set_module('numpy')
125: (0) def eye(N, M=None, k=0, dtype=float, order='C', *, like=None):
126: (4)     """
127: (4)         Return a 2-D array with ones on the diagonal and zeros elsewhere.
128: (4)         Parameters
129: (4)         -----
130: (4)         N : int
131: (6)             Number of rows in the output.
132: (4)         M : int, optional
133: (6)             Number of columns in the output. If None, defaults to `N`.
134: (4)         k : int, optional
135: (6)             Index of the diagonal: 0 (the default) refers to the main diagonal,
136: (6)             a positive value refers to an upper diagonal, and a negative value
137: (6)             to a lower diagonal.
138: (4)         dtype : data-type, optional
139: (6)             Data-type of the returned array.
140: (4)         order : {'C', 'F'}, optional
141: (8)             Whether the output should be stored in row-major (C-style) or
142: (8)             column-major (Fortran-style) order in memory.
143: (8)             .. versionadded:: 1.14.0
144: (4) ${ARRAY_FUNCTION_LIKE}
145: (8)             .. versionadded:: 1.20.0
146: (4) Returns
147: (4) -----
148: (4) I : ndarray of shape (N,M)
149: (6)     An array where all elements are equal to zero, except for the `k`-th
150: (6)     diagonal, whose values are equal to one.
151: (4) See Also
152: (4) -----
153: (4) identity : (almost) equivalent function
154: (4) diag : diagonal 2-D array from a 1-D array specified by the user.
155: (4) Examples
156: (4) -----
157: (4) >>> np.eye(2, dtype=int)
158: (4) array([[1, 0],
159: (11)           [0, 1]])
160: (4) >>> np.eye(3, k=1)
161: (4) array([[0., 1., 0.],
162: (11)           [0., 0., 1.],
163: (11)           [0., 0., 0.]])
164: (4) """
165: (4) if like is not None:
166: (8)     return _eye_with_like(like, N, M=M, k=k, dtype=dtype, order=order)
167: (4) if M is None:
168: (8)     M = N
169: (4) m = zeros((N, M), dtype=dtype, order=order)
170: (4) if k >= M:
171: (8)     return m
172: (4) M = operator.index(M)
173: (4) k = operator.index(k)
174: (4) if k >= 0:
175: (8)     i = k
176: (4) else:
177: (8)     i = (-k) * M
178: (4) m[:M-k].flat[i::M+1] = 1
179: (4) return m
180: (0) _eye_with_like = array_function_dispatch()(eye)
181: (0) def _diag_dispatcher(v, k=None):
182: (4)     return (v,)
183: (0) @array_function_dispatch(_diag_dispatcher)
184: (0) def diag(v, k=0):
185: (4) """
186: (4)     Extract a diagonal or construct a diagonal array.
187: (4)     See the more detailed documentation for ``numpy.diagonal`` if you use this
188: (4)     function to extract a diagonal and wish to write to the resulting array;
189: (4)     whether it returns a copy or a view depends on what version of numpy you
190: (4)     are using.
191: (4)     Parameters
192: (4)     -----

```

```

193: (4)           v : array_like
194: (8)             If `v` is a 2-D array, return a copy of its `k`-th diagonal.
195: (8)             If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th
196: (8)             diagonal.
197: (4)           k : int, optional
198: (8)             Diagonal in question. The default is 0. Use `k>0` for diagonals
199: (8)             above the main diagonal, and `k<0` for diagonals below the main
200: (8)             diagonal.
201: (4)           Returns
202: (4)           -----
203: (4)           out : ndarray
204: (8)             The extracted diagonal or constructed diagonal array.
205: (4)           See Also
206: (4)           -----
207: (4)             diagonal : Return specified diagonals.
208: (4)             diagflat : Create a 2-D array with the flattened input as a diagonal.
209: (4)             trace : Sum along diagonals.
210: (4)             triu : Upper triangle of an array.
211: (4)             tril : Lower triangle of an array.
212: (4)           Examples
213: (4)           -----
214: (4)             >>> x = np.arange(9).reshape((3,3))
215: (4)
216: (4)             array([[0, 1, 2],
217: (11)                 [3, 4, 5],
218: (11)                 [6, 7, 8]])
219: (4)             >>> np.diag(x)
220: (4)             array([0, 4, 8])
221: (4)             >>> np.diag(x, k=1)
222: (4)             array([1, 5])
223: (4)             >>> np.diag(x, k=-1)
224: (4)             array([3, 7])
225: (4)             >>> np.diag(np.diag(x))
226: (4)
227: (11)             array([[0, 0, 0],
228: (11)                 [0, 4, 0],
229: (4)                   [0, 0, 8]])
229: (4)             """
230: (4)             v = asanyarray(v)
231: (4)             s = v.shape
232: (4)             if len(s) == 1:
233: (8)                 n = s[0]+abs(k)
234: (8)                 res = zeros((n, n), v.dtype)
235: (8)                 if k >= 0:
236: (12)                     i = k
237: (8)                 else:
238: (12)                     i = (-k) * n
239: (8)                 res[:n-k].flat[i::n+1] = v
240: (8)                 return res
241: (4)             elif len(s) == 2:
242: (8)                 return diagonal(v, k)
243: (4)             else:
244: (8)                 raise ValueError("Input must be 1- or 2-d.")
245: (0)             @array_function_dispatch(_diag_dispatcher)
246: (0)             def diagflat(v, k=0):
247: (4)                 """
248: (4)                   Create a two-dimensional array with the flattened input as a diagonal.
249: (4)                   Parameters
250: (4)                   -----
251: (4)                   v : array_like
252: (8)                     Input data, which is flattened and set as the `k`-th
253: (8)                     diagonal of the output.
254: (4)                   k : int, optional
255: (8)                     Diagonal to set; 0, the default, corresponds to the "main" diagonal,
256: (8)                     a positive (negative) `k` giving the number of the diagonal above
257: (8)                     (below) the main.
258: (4)                   Returns
259: (4)                   -----
260: (4)                   out : ndarray
261: (8)                     The 2-D output array.

```

```

262: (4)          See Also
263: (4)          -----
264: (4)          diag : MATLAB work-alike for 1-D and 2-D arrays.
265: (4)          diagonal : Return specified diagonals.
266: (4)          trace : Sum along diagonals.
267: (4)          Examples
268: (4)          -----
269: (4)          >>> np.diagflat([[1,2], [3,4]])
270: (4)          array([[1, 0, 0, 0],
271: (11)             [0, 2, 0, 0],
272: (11)             [0, 0, 3, 0],
273: (11)             [0, 0, 0, 4]])
274: (4)          >>> np.diagflat([1,2], 1)
275: (4)          array([[0, 1, 0],
276: (11)             [0, 0, 2],
277: (11)             [0, 0, 0]])
278: (4)          """
279: (4)          try:
280: (8)              wrap = v.__array_wrap__
281: (4)          except AttributeError:
282: (8)              wrap = None
283: (4)          v = asarray(v).ravel()
284: (4)          s = len(v)
285: (4)          n = s + abs(k)
286: (4)          res = zeros((n, n), v.dtype)
287: (4)          if (k >= 0):
288: (8)              i = arange(0, n-k, dtype=intp)
289: (8)              fi = i+k+i*n
290: (4)          else:
291: (8)              i = arange(0, n+k, dtype=intp)
292: (8)              fi = i+(i-k)*n
293: (4)          res.flat[fi] = v
294: (4)          if not wrap:
295: (8)              return res
296: (4)          return wrap(res)
297: (0)          @set_array_function_like_doc
298: (0)          @set_module('numpy')
299: (0)          def tri(N, M=None, k=0, dtype=float, *, like=None):
300: (4)          """
301: (4)          An array with ones at and below the given diagonal and zeros elsewhere.
302: (4)          Parameters
303: (4)          -----
304: (4)          N : int
305: (8)              Number of rows in the array.
306: (4)          M : int, optional
307: (8)              Number of columns in the array.
308: (8)              By default, `M` is taken equal to `N`.
309: (4)          k : int, optional
310: (8)              The sub-diagonal at and below which the array is filled.
311: (8)              `k` = 0 is the main diagonal, while `k` < 0 is below it,
312: (8)              and `k` > 0 is above. The default is 0.
313: (4)          dtype : dtype, optional
314: (8)              Data type of the returned array. The default is float.
315: (4)          ${ARRAY_FUNCTION_LIKE}
316: (8)              .. versionadded:: 1.20.0
317: (4)          Returns
318: (4)          -----
319: (4)          tri : ndarray of shape (N, M)
320: (8)              Array with its lower triangle filled with ones and zero elsewhere;
321: (8)              in other words ``T[i,j] == 1`` for ``j <= i + k``, 0 otherwise.
322: (4)          Examples
323: (4)          -----
324: (4)          >>> np.tri(3, 5, 2, dtype=int)
325: (4)          array([[1, 1, 1, 0, 0],
326: (11)             [1, 1, 1, 1, 0],
327: (11)             [1, 1, 1, 1, 1]])
328: (4)          >>> np.tri(3, 5, -1)
329: (4)          array([[0., 0., 0., 0., 0.],
330: (11)             [1., 0., 0., 0., 0.]])

```

```

331: (11)           [1.,  1.,  0.,  0.,  0.]])
332: (4)           """
333: (4)           if like is not None:
334: (8)           return _tri_with_like(like, N, M=M, k=k, dtype=dtype)
335: (4)           if M is None:
336: (8)               M = N
337: (4)           m = greater_equal.outer(arange(N, dtype=_min_int(0, N)),
338: (28)                           arange(-k, M-k, dtype=_min_int(-k, M - k)))
339: (4)           m = m.astype(dtype, copy=False)
340: (4)           return m
341: (0)           _tri_with_like = array_function_dispatch()(tri)
342: (0)           def _trilu_dispatcher(m, k=None):
343: (4)               return (m,)
344: (0)           @array_function_dispatch(_trilu_dispatcher)
345: (0)           def tril(m, k=0):
346: (4)               """
347: (4)               Lower triangle of an array.
348: (4)               Return a copy of an array with elements above the `k`-th diagonal zeroed.
349: (4)               For arrays with ``ndim`` exceeding 2, `tril` will apply to the final two
350: (4)               axes.
351: (4)               Parameters
352: (4)               -----
353: (4)               m : array_like, shape (... , M, N)
354: (8)                   Input array.
355: (4)               k : int, optional
356: (8)                   Diagonal above which to zero elements. `k = 0` (the default) is the
357: (8)                   main diagonal, `k < 0` is below it and `k > 0` is above.
358: (4)               Returns
359: (4)               -----
360: (4)               tril : ndarray, shape (... , M, N)
361: (8)                   Lower triangle of `m` , of same shape and data-type as `m` .
362: (4)               See Also
363: (4)               -----
364: (4)               triu : same thing, only for the upper triangle
365: (4)               Examples
366: (4)               -----
367: (4)               >>> np.tril([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
368: (4)               array([[ 0,  0,  0],
369: (11)                 [ 4,  0,  0],
370: (11)                 [ 7,  8,  0],
371: (11)                 [10, 11, 12]])
372: (4)               >>> np.tril(np.arange(3*4*5).reshape(3, 4, 5))
373: (4)               array([[[ 0,  0,  0,  0,  0],
374: (12)                 [ 5,  6,  0,  0,  0],
375: (12)                 [10, 11, 12,  0,  0],
376: (12)                 [15, 16, 17, 18,  0]],
377: (11)                 [[20,  0,  0,  0,  0],
378: (12)                 [25, 26,  0,  0,  0],
379: (12)                 [30, 31, 32,  0,  0],
380: (12)                 [35, 36, 37, 38,  0]],
381: (11)                 [[40,  0,  0,  0,  0],
382: (12)                 [45, 46,  0,  0,  0],
383: (12)                 [50, 51, 52,  0,  0],
384: (12)                 [55, 56, 57, 58,  0]]])
385: (4)               """
386: (4)               m = asanyarray(m)
387: (4)               mask = tri(*m.shape[-2:], k=k, dtype=bool)
388: (4)               return where(mask, m, zeros(1, m.dtype))
389: (0)               @array_function_dispatch(_trilu_dispatcher)
390: (0)               def triu(m, k=0):
391: (4)                   """
392: (4)                   Upper triangle of an array.
393: (4)                   Return a copy of an array with the elements below the `k`-th diagonal
394: (4)                   zeroed. For arrays with ``ndim`` exceeding 2, `triu` will apply to the
395: (4)                   final two axes.
396: (4)                   Please refer to the documentation for `tril` for further details.
397: (4)                   See Also
398: (4)                   -----
399: (4)                   tril : lower triangle of an array

```

```

400: (4) Examples
401: (4)
402: (4) -----
403: (4) >>> np.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
404: (4) array([[ 1,  2,  3],
405: (4)           [ 4,  5,  6],
406: (4)           [ 0,  8,  9],
407: (4)           [ 0,  0, 12]])
408: (4) >>> np.triu(np.arange(3*4*5).reshape(3, 4, 5))
409: (4) array([[[ 0,  1,  2,  3,  4],
410: (4)           [ 0,  6,  7,  8,  9],
411: (4)           [ 0,  0, 12, 13, 14],
412: (4)           [ 0,  0,  0, 18, 19]],
413: (4)           [[20, 21, 22, 23, 24],
414: (4)             [ 0, 26, 27, 28, 29],
415: (4)             [ 0,  0, 32, 33, 34],
416: (4)             [ 0,  0,  0, 38, 39]],
417: (4)             [[40, 41, 42, 43, 44],
418: (4)               [ 0, 46, 47, 48, 49],
419: (4)               [ 0,  0, 52, 53, 54],
420: (4)               [ 0,  0,  0, 58, 59]]])
421: (4) """
422: (4) m = asanyarray(m)
423: (4) mask = tri(*m.shape[-2:], k=k-1, dtype=bool)
424: (0) return where(mask, zeros(1, m.dtype), m)
425: (4) def _vander_dispatcher(x, N=None, increasing=None):
426: (0)     return (x,)
427: (0) @array_function_dispatch(_vander_dispatcher)
428: (4) def vander(x, N=None, increasing=False):
429: (4) """
430: (4) Generate a Vandermonde matrix.
431: (4) The columns of the output matrix are powers of the input vector. The
432: (4) order of the powers is determined by the `increasing` boolean argument.
433: (4) Specifically, when `increasing` is False, the `i`-th output column is
434: (4) the input vector raised element-wise to the power of ``N - i - 1``. Such
435: (4) a matrix with a geometric progression in each row is named for Alexandre-
436: (4) Theophile Vandermonde.
437: (4) Parameters
438: (4) -----
439: (8) x : array_like
440: (4)     1-D input array.
441: (8) N : int, optional
442: (8)     Number of columns in the output. If `N` is not specified, a square
443: (4)     array is returned (``N = len(x)``).
444: (8) increasing : bool, optional
445: (8)     Order of the powers of the columns. If True, the powers increase
446: (8)     from left to right, if False (the default) they are reversed.
447: (4) .. versionadded:: 1.9.0
448: (4) Returns
449: (4) -----
450: (8) out : ndarray
451: (8)     Vandermonde matrix. If `increasing` is False, the first column is
452: (8)     ``x^(N-1)``, the second ``x^(N-2)`` and so forth. If `increasing` is
453: (4)     True, the columns are ``x^0, x^1, ..., x^(N-1)``.
454: (4) See Also
455: (4) -----
456: (4) polynomial.polynomial.polyvander
457: (4) Examples
458: (4) -----
459: (4) >>> x = np.array([1, 2, 3, 5])
460: (4) >>> N = 3
461: (4) >>> np.vander(x, N)
462: (4) array([[ 1,  1,  1],
463: (11)          [ 4,  2,  1],
464: (11)          [ 9,  3,  1],
465: (11)          [25,  5,  1]])
466: (4) >>> np.column_stack([x**((N-1)-i) for i in range(N)])
467: (4) array([[ 1,  1,  1],
468: (11)          [ 4,  2,  1],
468: (11)          [ 9,  3,  1],
468: (11)          [25,  5,  1]])

```

```

469: (11)          [25, 5, 1]])
470: (4)          >>> x = np.array([1, 2, 3, 5])
471: (4)          >>> np.vander(x)
472: (4)          array([[ 1,  1,  1,  1],
473: (11)             [ 8,  4,  2,  1],
474: (11)             [ 27,  9,  3,  1],
475: (11)             [125, 25,  5,  1]])
476: (4)          >>> np.vander(x, increasing=True)
477: (4)          array([[ 1,  1,  1,  1],
478: (11)             [ 1,  2,  4,  8],
479: (11)             [ 1,  3,  9, 27],
480: (11)             [ 1,  5, 25, 125]])
481: (4)          The determinant of a square Vandermonde matrix is the product
482: (4)          of the differences between the values of the input vector:
483: (4)          >>> np.linalg.det(np.vander(x))
484: (4)          48.00000000000043 # may vary
485: (4)          >>> (5-3)*(5-2)*(5-1)*(3-2)*(3-1)*(2-1)
486: (4)          48
487: (4)          """
488: (4)          x = asarray(x)
489: (4)          if x.ndim != 1:
490: (8)              raise ValueError("x must be a one-dimensional array or sequence.")
491: (4)          if N is None:
492: (8)              N = len(x)
493: (4)          v = empty((len(x), N), dtype=promote_types(x.dtype, int))
494: (4)          tmp = v[:, ::-1] if not increasing else v
495: (4)          if N > 0:
496: (8)              tmp[:, 0] = 1
497: (4)          if N > 1:
498: (8)              tmp[:, 1:] = x[:, None]
499: (8)              multiply.accumulate(tmp[:, 1:], out=tmp[:, 1:], axis=1)
500: (4)          return v
501: (0)          def _histogram2d_dispatcher(x, y, bins=None, range=None, density=None,
502: (28)                                weights=None):
503: (4)          yield x
504: (4)          yield y
505: (4)          try:
506: (8)              N = len(bins)
507: (4)          except TypeError:
508: (8)              N = 1
509: (4)          if N == 2:
510: (8)              yield from bins # bins=[x, y]
511: (4)          else:
512: (8)              yield bins
513: (4)          yield weights
514: (0)          @array_function_dispatch(_histogram2d_dispatcher)
515: (0)          def histogram2d(x, y, bins=10, range=None, density=None, weights=None):
516: (4)          """
517: (4)          Compute the bi-dimensional histogram of two data samples.
518: (4)          Parameters
519: (4)          -----
520: (4)          x : array_like, shape (N,)
521: (8)              An array containing the x coordinates of the points to be
522: (8)              histogrammed.
523: (4)          y : array_like, shape (N,)
524: (8)              An array containing the y coordinates of the points to be
525: (8)              histogrammed.
526: (4)          bins : int or array_like or [int, int] or [array, array], optional
527: (8)              The bin specification:
528: (10)                 * If int, the number of bins for the two dimensions (nx=ny=bins).
529: (10)                 * If array_like, the bin edges for the two dimensions
530: (12)                     (x_edges=y_edges=bins).
531: (10)                 * If [int, int], the number of bins in each dimension
532: (12)                     (nx, ny = bins).
533: (10)                 * If [array, array], the bin edges in each dimension
534: (12)                     (x_edges, y_edges = bins).
535: (10)                 * A combination [int, array] or [array, int], where int
536: (12)                     is the number of bins and array is the bin edges.
537: (4)          range : array_like, shape(2,2), optional

```

```

538: (8)           The leftmost and rightmost edges of the bins along each dimension
539: (8)           (if not specified explicitly in the `bins` parameters):
540: (8)           ``[[xmin, xmax], [ymin, ymax]]``. All values outside of this range
541: (8)           will be considered outliers and not tallied in the histogram.
542: (4)           density : bool, optional
543: (8)           If False, the default, returns the number of samples in each bin.
544: (8)           If True, returns the probability *density* function at the bin,
545: (8)           ``bin_count / sample_count / bin_area``.
546: (4)           weights : array_like, shape(N,), optional
547: (8)           An array of values ``w_i`` weighing each sample ``(x_i, y_i)``.
548: (8)           Weights are normalized to 1 if `density` is True. If `density` is
549: (8)           False, the values of the returned histogram are equal to the sum of
550: (8)           the weights belonging to the samples falling into each bin.
551: (4)           Returns
552: (4)           -----
553: (4)           H : ndarray, shape(nx, ny)
554: (8)           The bi-dimensional histogram of samples `x` and `y`. Values in `x`
555: (8)           are histogrammed along the first dimension and values in `y` are
556: (8)           histogrammed along the second dimension.
557: (4)           xedges : ndarray, shape(nx+1,)
558: (8)           The bin edges along the first dimension.
559: (4)           yedges : ndarray, shape(ny+1,)
560: (8)           The bin edges along the second dimension.
561: (4)           See Also
562: (4)           -----
563: (4)           histogram : 1D histogram
564: (4)           histogramdd : Multidimensional histogram
565: (4)           Notes
566: (4)           -----
567: (4)           When `density` is True, then the returned histogram is the sample
568: (4)           density, defined such that the sum over bins of the product
569: (4)           ``bin_value * bin_area`` is 1.
570: (4)           Please note that the histogram does not follow the Cartesian convention
571: (4)           where `x` values are on the abscissa and `y` values on the ordinate
572: (4)           axis. Rather, `x` is histogrammed along the first dimension of the
573: (4)           array (vertical), and `y` along the second dimension of the array
574: (4)           (horizontal). This ensures compatibility with `histogramdd`.
575: (4)           Examples
576: (4)           -----
577: (4)           >>> from matplotlib.image import NonUniformImage
578: (4)           >>> import matplotlib.pyplot as plt
579: (4)           Construct a 2-D histogram with variable bin width. First define the bin
580: (4)           edges:
581: (4)           >>> xedges = [0, 1, 3, 5]
582: (4)           >>> yedges = [0, 2, 3, 4, 6]
583: (4)           Next we create a histogram H with random bin content:
584: (4)           >>> x = np.random.normal(2, 1, 100)
585: (4)           >>> y = np.random.normal(1, 1, 100)
586: (4)           >>> H, xedges, yedges = np.histogram2d(x, y, bins=(xedges, yedges))
587: (4)           >>> # Histogram does not follow Cartesian convention (see Notes),
588: (4)           >>> # therefore transpose H for visualization purposes.
589: (4)           >>> H = H.T
590: (4)           :func:`imshow <matplotlib.pyplot.imshow>` can only display square bins:
591: (4)           >>> fig = plt.figure(figsize=(7, 3))
592: (4)           >>> ax = fig.add_subplot(131, title='imshow: square bins')
593: (4)           >>> plt.imshow(H, interpolation='nearest', origin='lower',
594: (4)               ...          extent=[xedges[0], xedges[-1], yedges[0], yedges[-1]])
595: (4)           <matplotlib.image.AxesImage object at 0x...>
596: (4)           :func:`pcolormesh <matplotlib.pyplot.pcormesh>` can display actual
edges:
597: (4)           >>> ax = fig.add_subplot(132, title='pcolormesh: actual edges',
598: (4)               ...          aspect='equal')
599: (4)           >>> X, Y = np.meshgrid(xedges, yedges)
600: (4)           >>> ax.pcormesh(X, Y, H)
601: (4)           <matplotlib.collections.QuadMesh object at 0x...>
602: (4)           :class:`NonUniformImage <matplotlib.image.NonUniformImage>` can be used to
603: (4)           display actual bin edges with interpolation:
604: (4)           >>> ax = fig.add_subplot(133, title='NonUniformImage: interpolated',
605: (4)               ...          aspect='equal', xlim=xedges[[0, -1]], ylim=yedges[[0, -1]])

```

```

606: (4)             >>> im = NonUniformImage(ax, interpolation='bilinear')
607: (4)             >>> xcenters = (xedges[:-1] + xedges[1:]) / 2
608: (4)             >>> ycenters = (yedges[:-1] + yedges[1:]) / 2
609: (4)             >>> im.set_data(xcenters, ycenters, H)
610: (4)             >>> ax.add_image(im)
611: (4)             >>> plt.show()
612: (4)             It is also possible to construct a 2-D histogram without specifying bin
613: (4)             edges:
614: (4)             >>> # Generate non-symmetric test data
615: (4)             >>> n = 10000
616: (4)             >>> x = np.linspace(1, 100, n)
617: (4)             >>> y = 2*np.log(x) + np.random.rand(n) - 0.5
618: (4)             >>> # Compute 2d histogram. Note the order of x/y and xedges/yedges
619: (4)             >>> H, yedges, xedges = np.histogram2d(y, x, bins=20)
620: (4)             Now we can plot the histogram using
621: (4)             :func:`pcolormesh <matplotlib.pyplot.pcolormesh>`, and a
622: (4)             :func:`hexbin <matplotlib.pyplot.hexbin>` for comparison.
623: (4)             >>> # Plot histogram using pcolormesh
624: (4)             >>> fig, (ax1, ax2) = plt.subplots(ncols=2, sharey=True)
625: (4)             >>> ax1.pcolormesh(xedges, yedges, H, cmap='rainbow')
626: (4)             >>> ax1.plot(x, 2*np.log(x), 'k-')
627: (4)             >>> ax1.set_xlim(x.min(), x.max())
628: (4)             >>> ax1.set_ylim(y.min(), y.max())
629: (4)             >>> ax1.set_xlabel('x')
630: (4)             >>> ax1.set_ylabel('y')
631: (4)             >>> ax1.set_title('histogram2d')
632: (4)             >>> ax1.grid()
633: (4)             >>> # Create hexbin plot for comparison
634: (4)             >>> ax2.hexbin(x, y, gridsize=20, cmap='rainbow')
635: (4)             >>> ax2.plot(x, 2*np.log(x), 'k-')
636: (4)             >>> ax2.set_title('hexbin')
637: (4)             >>> ax2.set_xlim(x.min(), x.max())
638: (4)             >>> ax2.set_xlabel('x')
639: (4)             >>> ax2.grid()
640: (4)             >>> plt.show()
641: (4)             """
642: (4)             from numpy import histogramdd
643: (4)             if len(x) != len(y):
644: (8)                 raise ValueError('x and y must have the same length.')
645: (4)             try:
646: (8)                 N = len(bins)
647: (4)             except TypeError:
648: (8)                 N = 1
649: (4)             if N != 1 and N != 2:
650: (8)                 xedges = yedges = asarray(bins)
651: (8)                 bins = [xedges, yedges]
652: (4)             hist, edges = histogramdd([x, y], bins, range, density, weights)
653: (4)             return hist, edges[0], edges[1]
654: (0)             @set_module('numpy')
655: (0)             def mask_indices(n, mask_func, k=0):
656: (4)             """
657: (4)             Return the indices to access (n, n) arrays, given a masking function.
658: (4)             Assume `mask_func` is a function that, for a square array a of size
659: (4)             ````(n, n)```` with a possible offset argument `k`, when called as
660: (4)             ````mask_func(a, k)```` returns a new array with zeros in certain locations
661: (4)             (functions like `triu` or `tril` do precisely this). Then this function
662: (4)             returns the indices where the non-zero values would be located.
663: (4)             Parameters
664: (4)             -----
665: (4)             n : int
666: (8)                 The returned indices will be valid to access arrays of shape (n, n).
667: (4)             mask_func : callable
668: (8)                 A function whose call signature is similar to that of `triu`, `tril`.
669: (8)                 That is, ````mask_func(x, k)```` returns a boolean array, shaped like `x`.
670: (8)                 `k` is an optional argument to the function.
671: (4)             k : scalar
672: (8)                 An optional argument which is passed through to `mask_func`. Functions
673: (8)                 like `triu`, `tril` take a second argument that is interpreted as an
674: (8)                 offset.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

675: (4)           Returns
676: (4)           -----
677: (4)           indices : tuple of arrays.
678: (8)             The `n` arrays of indices corresponding to the locations where
679: (8)             ``mask_func(np.ones((n, n)), k)`` is True.
680: (4)           See Also
681: (4)           -----
682: (4)           triu, tril, triu_indices, tril_indices
683: (4)           Notes
684: (4)           -----
685: (4)             .. versionadded:: 1.4.0
686: (4)           Examples
687: (4)           -----
688: (4)           These are the indices that would allow you to access the upper triangular
689: (4)           part of any 3x3 array:
690: (4)             >>> iu = np.mask_indices(3, np.triu)
691: (4)             For example, if `a` is a 3x3 array:
692: (4)             >>> a = np.arange(9).reshape(3, 3)
693: (4)             >>> a
694: (4)             array([[0, 1, 2],
695: (11)               [3, 4, 5],
696: (11)               [6, 7, 8]])
697: (4)             >>> a[iu]
698: (4)             array([0, 1, 2, 4, 5, 8])
699: (4)           An offset can be passed also to the masking function. This gets us the
700: (4)           indices starting on the first diagonal right of the main one:
701: (4)             >>> iu1 = np.mask_indices(3, np.triu, 1)
702: (4)             with which we now extract only three elements:
703: (4)             >>> a[iu1]
704: (4)             array([1, 2, 5])
705: (4)             """
706: (4)             m = ones((n, n), int)
707: (4)             a = mask_func(m, k)
708: (4)             return nonzero(a != 0)
709: (0)           @set_module('numpy')
710: (0)           def tril_indices(n, k=0, m=None):
711: (4)             """
712: (4)             Return the indices for the lower-triangle of an (n, m) array.
713: (4)             Parameters
714: (4)             -----
715: (4)             n : int
716: (8)               The row dimension of the arrays for which the returned
717: (8)               indices will be valid.
718: (4)             k : int, optional
719: (8)               Diagonal offset (see `tril` for details).
720: (4)             m : int, optional
721: (8)               .. versionadded:: 1.9.0
722: (8)               The column dimension of the arrays for which the returned
723: (8)               arrays will be valid.
724: (8)               By default `m` is taken equal to `n`.
725: (4)             Returns
726: (4)             -----
727: (4)             inds : tuple of arrays
728: (8)               The indices for the triangle. The returned tuple contains two arrays,
729: (8)               each with the indices along one dimension of the array.
730: (4)             See also
731: (4)             -----
732: (4)             triu_indices : similar function, for upper-triangular.
733: (4)             mask_indices : generic function accepting an arbitrary mask function.
734: (4)             tril, triu
735: (4)             Notes
736: (4)             -----
737: (4)             .. versionadded:: 1.4.0
738: (4)             Examples
739: (4)             -----
740: (4)             Compute two different sets of indices to access 4x4 arrays, one for the
741: (4)             lower triangular part starting at the main diagonal, and one starting two
742: (4)             diagonals further right:
743: (4)             >>> ill1 = np.tril_indices(4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

744: (4)          >>> il2 = np.tril_indices(4, 2)
745: (4)          Here is how they can be used with a sample array:
746: (4)          >>> a = np.arange(16).reshape(4, 4)
747: (4)          >>> a
748: (4)          array([[ 0,  1,  2,  3],
749: (11)             [ 4,  5,  6,  7],
750: (11)             [ 8,  9, 10, 11],
751: (11)             [12, 13, 14, 15]])
752: (4)          Both for indexing:
753: (4)          >>> a[il1]
754: (4)          array([ 0,  4,  5, ..., 13, 14, 15])
755: (4)          And for assigning values:
756: (4)          >>> a[il1] = -1
757: (4)          >>> a
758: (4)          array([[-1,  1,  2,  3],
759: (11)             [-1, -1,  6,  7],
760: (11)             [-1, -1, -1, 11],
761: (11)             [-1, -1, -1, -1]])
762: (4)          These cover almost the whole array (two diagonals right of the main one):
763: (4)          >>> a[il2] = -10
764: (4)          >>> a
765: (4)          array([[-10, -10, -10,   3],
766: (11)             [-10, -10, -10, -10],
767: (11)             [-10, -10, -10, -10],
768: (11)             [-10, -10, -10, -10]])
769: (4)          """
770: (4)          tri_ = tri(n, m, k=k, dtype=bool)
771: (4)          return tuple(broadcast_to(inds, tri_.shape)[tri_]
772: (17)                         for inds in indices(tri_.shape, sparse=True))
773: (0)          def _trilu_indices_form_dispatcher(arr, k=None):
774: (4)          return (arr,)
775: (0)          @array_function_dispatch(_trilu_indices_form_dispatcher)
776: (0)          def tril_indices_from(arr, k=0):
777: (4)          """
778: (4)          Return the indices for the lower-triangle of arr.
779: (4)          See `tril_indices` for full details.
780: (4)          Parameters
781: (4)          -----
782: (4)          arr : array_like
783: (8)          The indices will be valid for square arrays whose dimensions are
784: (8)          the same as arr.
785: (4)          k : int, optional
786: (8)          Diagonal offset (see `tril` for details).
787: (4)          Examples
788: (4)          -----
789: (4)          Create a 4 by 4 array.
790: (4)          >>> a = np.arange(16).reshape(4, 4)
791: (4)          >>> a
792: (4)          array([[ 0,  1,  2,  3],
793: (11)             [ 4,  5,  6,  7],
794: (11)             [ 8,  9, 10, 11],
795: (11)             [12, 13, 14, 15]])
796: (4)          Pass the array to get the indices of the lower triangular elements.
797: (4)          >>> trili = np.tril_indices_from(a)
798: (4)          >>> trili
799: (4)          (array([0, 1, 1, 2, 2, 3, 3, 3]), array([0, 0, 1, 0, 1, 2, 0, 1, 2,
3]))
800: (4)          >>> a[trili]
801: (4)          array([ 0,  4,  5,  8,  9, 10, 12, 13, 14, 15])
802: (4)          This is syntactic sugar for tril_indices().
803: (4)          >>> np.tril_indices(a.shape[0])
804: (4)          (array([0, 1, 1, 2, 2, 3, 3, 3]), array([0, 0, 1, 0, 1, 2, 0, 1, 2,
3]))
805: (4)          Use the `k` parameter to return the indices for the lower triangular array
806: (4)          up to the k-th diagonal.
807: (4)          >>> trili1 = np.tril_indices_from(a, k=1)
808: (4)          >>> a[trili1]
809: (4)          array([ 0,  1,  4,  5,  6,  8,  9, 10, 11, 12, 13, 14, 15])
810: (4)          See Also

```

```

811: (4)      -----
812: (4)      tril_indices, tril, triu_indices_from
813: (4)      Notes
814: (4)      -----
815: (4)      .. versionadded:: 1.4.0
816: (4)      """
817: (4)      if arr.ndim != 2:
818: (8)          raise ValueError("input array must be 2-d")
819: (4)      return tril_indices(arr.shape[-2], k=k, m=arr.shape[-1])
820: (0)      @set_module('numpy')
821: (0)      def triu_indices(n, k=0, m=None):
822: (4)          """
823: (4)          Return the indices for the upper-triangle of an (n, m) array.
824: (4)          Parameters
825: (4)          -----
826: (4)          n : int
827: (8)              The size of the arrays for which the returned indices will
828: (8)              be valid.
829: (4)          k : int, optional
830: (8)              Diagonal offset (see `triu` for details).
831: (4)          m : int, optional
832: (8)              .. versionadded:: 1.9.0
833: (8)              The column dimension of the arrays for which the returned
834: (8)              arrays will be valid.
835: (8)              By default `m` is taken equal to `n`.
836: (4)          Returns
837: (4)          -----
838: (4)          inds : tuple, shape(2) of ndarrays, shape(`n`)
839: (8)              The indices for the triangle. The returned tuple contains two arrays,
840: (8)              each with the indices along one dimension of the array. Can be used
841: (8)              to slice a ndarray of shape(`n`, `n`).
842: (4)          See also
843: (4)          -----
844: (4)          tril_indices : similar function, for lower-triangular.
845: (4)          mask_indices : generic function accepting an arbitrary mask function.
846: (4)          triu, tril
847: (4)          Notes
848: (4)          -----
849: (4)          .. versionadded:: 1.4.0
850: (4)          Examples
851: (4)          -----
852: (4)          Compute two different sets of indices to access 4x4 arrays, one for the
853: (4)          upper triangular part starting at the main diagonal, and one starting two
854: (4)          diagonals further right:
855: (4)          >>> iu1 = np.triu_indices(4)
856: (4)          >>> iu2 = np.triu_indices(4, 2)
857: (4)          Here is how they can be used with a sample array:
858: (4)          >>> a = np.arange(16).reshape(4, 4)
859: (4)          >>> a
860: (4)          array([[ 0,  1,  2,  3],
861: (11)             [ 4,  5,  6,  7],
862: (11)             [ 8,  9, 10, 11],
863: (11)             [12, 13, 14, 15]])
864: (4)          Both for indexing:
865: (4)          >>> a[iu1]
866: (4)          array([ 0,  1,  2, ..., 10, 11, 15])
867: (4)          And for assigning values:
868: (4)          >>> a[iu1] = -1
869: (4)          >>> a
870: (4)          array([[ -1, -1, -1, -1],
871: (11)             [ 4, -1, -1, -1],
872: (11)             [ 8,  9, -1, -1],
873: (11)             [12, 13, 14, -1]])
874: (4)          These cover only a small part of the whole array (two diagonals right
875: (4)          of the main one):
876: (4)          >>> a[iu2] = -10
877: (4)          >>> a
878: (4)          array([[ -1, -1, -10, -10],
879: (11)             [ 4, -1, -1, -10]],

```

```

880: (11)          [ 8,   9,  -1,  -1],
881: (11)          [ 12,  13,  14,  -1]])
882: (4)
883: (4)          """
884: (4)          tri_ = ~tri(n, m, k=k - 1, dtype=bool)
885: (17)          return tuple(broadcast_to(indxs, tri_.shape)[tri_]
886: (0)                  for indxs in indices(tri_.shape, sparse=True))
887: (0) @array_function_dispatch(_trilu_indices_form_dispatcher)
888: (4) def triu_indices_from(arr, k=0):
889: (4) """
890: (4)     Return the indices for the upper-triangle of arr.
891: (4)     See `triu_indices` for full details.
892: (4)     Parameters
893: (4)     -----
894: (8)     arr : ndarray, shape(N, N)
895: (4)         The indices will be valid for square arrays.
896: (8)     k : int, optional
897: (4)         Diagonal offset (see `triu` for details).
898: (4)     Returns
899: (4)     -----
900: (8)     triu_indices_from : tuple, shape(2) of ndarray, shape(N)
901: (4)         Indices for the upper-triangle of `arr`.
902: (4)     Examples
903: (4)     -----
904: (4)     Create a 4 by 4 array.
905: (4)     >>> a = np.arange(16).reshape(4, 4)
906: (4)     >>> a
907: (11)    array([[ 0,  1,  2,  3],
908: (11)              [ 4,  5,  6,  7],
909: (11)              [ 8,  9, 10, 11],
910: (11)              [12, 13, 14, 15]])
911: (4)     Pass the array to get the indices of the upper triangular elements.
912: (4)     >>> triui = np.triu_indices_from(a)
913: (4)     >>> triui
914: (4)     (array([0, 0, 0, 0, 1, 1, 1, 2, 2, 3]), array([0, 1, 2, 3, 1, 2, 3, 2, 3,
915: (4)     3]))
916: (4)     >>> a[triui]
917: (4)     array([ 0,  1,  2,  3,  5,  6,  7, 10, 11, 15])
918: (4)     This is syntactic sugar for triu_indices().
919: (4)     >>> np.triu_indices(a.shape[0])
920: (4)     (array([0, 0, 0, 0, 1, 1, 1, 2, 2, 3]), array([0, 1, 2, 3, 1, 2, 3, 2, 3,
921: (4)     3]))
922: (4)     Use the `k` parameter to return the indices for the upper triangular array
923: (4)     from the k-th diagonal.
924: (4)     >>> triuim1 = np.triu_indices_from(a, k=1)
925: (4)     >>> a[triuim1]
926: (4)     array([ 1,  2,  3,  6,  7, 11])
927: (4)     See Also
928: (4)     -----
929: (4)     triu_indices, triu, tril_indices_from
930: (4)     Notes
931: (4)     -----
932: (4)     .. versionadded:: 1.4.0
933: (4)     """
934: (4)     if arr.ndim != 2:
935: (8)         raise ValueError("input array must be 2-d")
936: (4)     return triu_indices(arr.shape[-2], k=k, m=arr.shape[-1])

```

-----  
File 221 - type\_check.py:

```

1: (0)     """Automatically adapted for numpy Sep 19, 2005 by convertcode.py
2: (0)
3: (0)     import functools
4: (0)     __all__ = ['iscomplexobj', 'isrealobj', 'imag', 'iscomplex',
5: (11)         'isreal', 'nan_to_num', 'real', 'real_if_close',
6: (11)         'typename', 'asfarray', 'mintypecode',
7: (11)         'common_type']
8: (0)     from .._utils import set_module

```

```

9: (0) import numpy.core.numeric as _nx
10: (0) from numpy.core.numeric import asarray, asanyarray, isnan, zeros
11: (0) from numpy.core import overrides, getlimits
12: (0) from .ufunclike import isneginf, isposinf
13: (0) array_function_dispatch = functools.partial(
14:     (4)     overrides.array_function_dispatch, module='numpy')
15: (0) _typecodes_by_elsize = 'GDFgdfQqLlIiHhBb?'
16: (0) @set_module('numpy')
17: (0) def mintypecode(typechars, typeset='GDFgdf', default='d'):
18:     (4)
19:     (4)         """
20:     (4)             Return the character for the minimum-size type to which given types can
21:             (4)             be safely cast.
22:             (4)             The returned type character must represent the smallest size dtype such
23:             (4)             that an array of the returned type can handle the data from an array of
24:             (4)             all types in `typechars` (or if `typechars` is an array, then its
25:             (4)             dtype.char).
26:             (4)             Parameters
27:             (4)             -----
28:             (4)             typechars : list of str or array_like
29:             (8)                 If a list of strings, each string should represent a dtype.
30:             (8)                 If array_like, the character representation of the array dtype is
31:             used.
32:             (4)             typeset : str or list of str, optional
33:             (8)                 The set of characters that the returned character is chosen from.
34:             (8)                 The default set is 'GDFgdf'.
35:             (4)             default : str, optional
36:             (8)                 The default character, this is returned if none of the characters in
37:             (8)                 `typechars` matches a character in `typeset`.
38:             (4)             Returns
39:             (4)             -----
40:             (4)             typechar : str
41:             (8)                 The character representing the minimum-size type that was found.
42:             See Also
43:             (4)             -----
44:             (4)             dtype, sctype2char, maximum_sctype
45:             Examples
46:             (4)             -----
47:             (4)             >>> np.mintypecode(['d', 'f', 'S'])
48:             (4)             'd'
49:             (4)             >>> x = np.array([1.1, 2-3.j])
50:             (4)             >>> np.mintypecode(x)
51:             (4)             'D'
52:             (4)             >>> np.mintypecode('abceh', default='G')
53:             (4)             'G'
54:             (4)             """
55:             (4)             typecodes = ((isinstance(t, str) and t) or asarray(t).dtype.char
56:                         (17)                         for t in typechars)
57:             intersection = set(t for t in typecodes if t in typeset)
58:             if not intersection:
59:                 return default
60:             if 'F' in intersection and 'd' in intersection:
61:                 return 'D'
62:             return min(intersection, key=_typecodes_by_elsize.index)
63:             def _asfarray_dispatcher(a, dtype=None):
64:                 (0)                     return (a,)
65:             @array_function_dispatch(_asfarray_dispatcher)
66:             def asfarray(a, dtype=_nx.float_):
67:                 (4)
68:                     """
69:                     (4)             Return an array converted to a float type.
70:                     (4)             Parameters
71:                     (4)             -----
72:                     (4)             a : array_like
73:                     (8)                 The input array.
74:                     (4)             dtype : str or dtype object, optional
75:                     (8)                 Float type code to coerce input array `a`. If `dtype` is one of the
76:                     (8)                 'int' dtypes, it is replaced with float64.
77:                     Returns
78:                     -----
79:                     out : ndarray

```

```

77: (8)          The input `a` as a float ndarray.
78: (4)          Examples
79: (4)          -----
80: (4)          >>> np.asarray([2, 3])
81: (4)          array([2., 3.])
82: (4)          >>> np.asarray([2, 3], dtype='float')
83: (4)          array([2., 3.])
84: (4)          >>> np.asarray([2, 3], dtype='int8')
85: (4)          array([2., 3.])
86: (4)          """
87: (4)          if not _nx.issubdtype(dtype, _nx.inexact):
88: (8)              dtype = _nx.float_
89: (4)          return asarray(a, dtype=dtype)
90: (0)          def _real_dispatcher(val):
91: (4)              return (val,)
92: (0)          @array_function_dispatch(_real_dispatcher)
93: (0)
94: (4)          def real(val):
95: (4)              """
96: (4)              Return the real part of the complex argument.
97: (4)              Parameters
98: (4)              -----
99: (8)              val : array_like
100: (4)                  Input array.
101: (4)              Returns
102: (4)              -----
103: (4)              out : ndarray or scalar
104: (8)                  The real component of the complex argument. If `val` is real, the type
105: (8)                  of `val` is used for the output. If `val` has complex elements, the
106: (8)                  returned type is float.
107: (4)              See Also
108: (4)              -----
109: (4)              real_if_close, imag, angle
110: (4)              Examples
111: (4)              -----
112: (4)              >>> a = np.array([1+2j, 3+4j, 5+6j])
113: (4)              >>> a.real
114: (4)              array([1., 3., 5.])
115: (4)              >>> a.real = 9
116: (4)              >>> a
117: (4)              array([9.+2.j, 9.+4.j, 9.+6.j])
118: (4)              >>> a.real = np.array([9, 8, 7])
119: (4)              >>> a
120: (4)              array([9.+2.j, 8.+4.j, 7.+6.j])
121: (4)              >>> np.real(1 + 1j)
122: (4)              1.0
123: (4)              """
124: (4)              try:
125: (8)                  return val.real
126: (4)              except AttributeError:
127: (8)                  return asanyarray(val).real
128: (0)          def _imag_dispatcher(val):
129: (4)              return (val,)
130: (0)          @array_function_dispatch(_imag_dispatcher)
131: (0)
132: (4)          def imag(val):
133: (4)              """
134: (4)              Return the imaginary part of the complex argument.
135: (4)              Parameters
136: (4)              -----
137: (4)              val : array_like
138: (4)                  Input array.
139: (4)              Returns
140: (4)              -----
141: (4)              out : ndarray or scalar
142: (8)                  The imaginary component of the complex argument. If `val` is real,
143: (8)                  the type of `val` is used for the output. If `val` has complex
144: (8)                  elements, the returned type is float.
145: (4)              See Also
146: (4)              -----
147: (4)              real, angle, real_if_close

```

```

146: (4)          Examples
147: (4)
148: (4)
149: (4)
150: (4)
151: (4)
152: (4)
153: (4)
154: (4)
155: (4)
156: (4)
157: (4)
158: (8)
159: (4)
160: (8)
161: (0)          -----
162: (4)          >>> a = np.array([1+2j, 3+4j, 5+6j])
163: (0)          >>> a.imag
164: (0)          array([2., 4., 6.])
165: (4)          >>> a.imag = np.array([8, 10, 12])
166: (4)          >>> a
167: (4)          array([1. +8.j, 3.+10.j, 5.+12.j])
168: (4)          >>> np.imag(1 + 1j)
169: (4)          1.0
170: (4)          """
171: (4)          try:
172: (8)              return val.imag
173: (4)          except AttributeError:
174: (8)              return asanyarray(val).imag
175: (0)          def _is_type_dispatcher(x):
176: (4)              return (x,)
177: (0)          @array_function_dispatch(_is_type_dispatcher)
178: (0)          def iscomplex(x):
179: (4)              """
180: (4)              Returns a bool array, where True if input element is complex.
181: (4)              What is tested is whether the input has a non-zero imaginary part, not if
182: (4)              the input type is complex.
183: (4)          Parameters
184: (4)          -----
185: (4)          x : array_like
186: (8)              Input array.
187: (4)          Returns
188: (4)          -----
189: (4)          out : ndarray of bools
190: (8)              Output array.
191: (4)          See Also
192: (4)          -----
193: (4)          isreal
194: (4)          iscomplexobj : Return True if x is a complex type or an array of complex
195: (19)             numbers.
196: (4)          Examples
197: (4)
198: (4)
199: (4)
200: (4)
201: (8)
202: (4)
203: (4)
204: (4)
205: (8)
206: (4)
207: (4)
208: (4)
209: (4)
210: (4)
211: (4)
212: (4)
213: (4)

-----
```

>>> np.iscomplex([1+1j, 1+0j, 4.5, 3, 2, 2j])  
array([ True, False, False, False, False, True])

"""

ax = asanyarray(x)  
if issubclass(ax.dtype.type, \_nx.complexfloating):  
 return ax.imag != 0  
res = zeros(ax.shape, bool)  
return res[()] # convert to scalar if needed

@array\_function\_dispatch(\_is\_type\_dispatcher)

def isreal(x):

"""
-----

Returns a bool array, where True if input element is real.  
If element has complex type with zero complex part, the return value  
for that element is True.

Parameters

-----

x : array\_like  
Input array.

Returns

-----

out : ndarray, bool  
Boolean array of same shape as `x`.

Notes

-----

`isreal` may behave unexpectedly for string or object arrays (see  
examples)

See Also

-----

iscomplex

isrealobj : Return True if x is not a complex type.

Examples

```

214: (4)      -----
215: (4)      >>> a = np.array([1+1j, 1+0j, 4.5, 3, 2, 2j], dtype=complex)
216: (4)      >>> np.isreal(a)
217: (4)      array([False, True, True, True, False])
218: (4)      The function does not work on string arrays.
219: (4)      >>> a = np.array([2j, "a"], dtype="U")
220: (4)      >>> np.isreal(a) # Warns about non-elementwise comparison
221: (4)      False
222: (4)      Returns True for all elements in input array of ``dtype=object`` even if
223: (4)      any of the elements is complex.
224: (4)      >>> a = np.array([1, "2", 3+4j], dtype=object)
225: (4)      >>> np.isreal(a)
226: (4)      array([ True, True, True])
227: (4)      isreal should not be used with object arrays
228: (4)      >>> a = np.array([1+2j, 2+1j], dtype=object)
229: (4)      >>> np.isreal(a)
230: (4)      array([ True, True])
231: (4)      """
232: (4)      return imag(x) == 0
233: (0)      @array_function_dispatch(_is_type_dispatcher)
234: (0)      def iscomplexobj(x):
235: (4)      """
236: (4)      Check for a complex type or an array of complex numbers.
237: (4)      The type of the input is checked, not the value. Even if the input
238: (4)      has an imaginary part equal to zero, `iscomplexobj` evaluates to True.
239: (4)      Parameters
240: (4)      -----
241: (4)      x : any
242: (8)      The input can be of any type and shape.
243: (4)      Returns
244: (4)      -----
245: (4)      iscomplexobj : bool
246: (8)      The return value, True if `x` is of a complex type or has at least
247: (8)      one complex element.
248: (4)      See Also
249: (4)      -----
250: (4)      isrealobj, iscomplex
251: (4)      Examples
252: (4)      -----
253: (4)      >>> np.iscomplexobj(1)
254: (4)      False
255: (4)      >>> np.iscomplexobj(1+0j)
256: (4)      True
257: (4)      >>> np.iscomplexobj([3, 1+0j, True])
258: (4)      True
259: (4)      """
260: (4)      try:
261: (8)          dtype = x.dtype
262: (8)          type_ = dtype.type
263: (4)      except AttributeError:
264: (8)          type_ = asarray(x).dtype.type
265: (4)      return issubclass(type_, _nx.complexfloating)
266: (0)      @array_function_dispatch(_is_type_dispatcher)
267: (0)      def isrealobj(x):
268: (4)      """
269: (4)      Return True if x is a not complex type or an array of complex numbers.
270: (4)      The type of the input is checked, not the value. So even if the input
271: (4)      has an imaginary part equal to zero, `isrealobj` evaluates to False
272: (4)      if the data type is complex.
273: (4)      Parameters
274: (4)      -----
275: (4)      x : any
276: (8)      The input can be of any type and shape.
277: (4)      Returns
278: (4)      -----
279: (4)      y : bool
280: (8)      The return value, False if `x` is of a complex type.
281: (4)      See Also
282: (4)      -----

```

```

283: (4)           iscomplexobj, isreal
284: (4)           Notes
285: (4)           -----
286: (4)           The function is only meant for arrays with numerical values but it
287: (4)           accepts all other objects. Since it assumes array input, the return
288: (4)           value of other objects may be True.
289: (4)           >>> np.isrealobj('A string')
290: (4)           True
291: (4)           >>> np.isrealobj(False)
292: (4)           True
293: (4)           >>> np.isrealobj(None)
294: (4)           True
295: (4)           Examples
296: (4)           -----
297: (4)           >>> np.isrealobj(1)
298: (4)           True
299: (4)           >>> np.isrealobj(1+0j)
300: (4)           False
301: (4)           >>> np.isrealobj([3, 1+0j, True])
302: (4)           False
303: (4)           """
304: (4)           return not iscomplexobj(x)
305: (0)           def _getmaxmin(t):
306: (4)               from numpy.core import getlimits
307: (4)               f = getlimits.finfo(t)
308: (4)               return f.max, f.min
309: (0)           def _nan_to_num_dispatcher(x, copy=None, nan=None, posinf=None, neginf=None):
310: (4)               return (x,)
311: (0)           @array_function_dispatch(_nan_to_num_dispatcher)
312: (0)           def nan_to_num(x, copy=True, nan=0.0, posinf=None, neginf=None):
313: (4)               """
314: (4)               Replace NaN with zero and infinity with large finite numbers (default
315: (4)               behaviour) or with the numbers defined by the user using the `nan`,
316: (4)               `posinf` and/or `neginf` keywords.
317: (4)               If `x` is inexact, NaN is replaced by zero or by the user defined value in
318: (4)               `nan` keyword, infinity is replaced by the largest finite floating point
319: (4)               values representable by ``x.dtype`` or by the user defined value in
320: (4)               `posinf` keyword and -infinity is replaced by the most negative finite
321: (4)               floating point values representable by ``x.dtype`` or by the user defined
322: (4)               value in `neginf` keyword.
323: (4)               For complex dtypes, the above is applied to each of the real and
324: (4)               imaginary components of `x` separately.
325: (4)               If `x` is not inexact, then no replacements are made.
326: (4)               Parameters
327: (4)               -----
328: (4)               x : scalar or array_like
329: (8)                   Input data.
330: (4)               copy : bool, optional
331: (8)                   Whether to create a copy of `x` (True) or to replace values
332: (8)                   in-place (False). The in-place operation only occurs if
333: (8)                   casting to an array does not require a copy.
334: (8)                   Default is True.
335: (8)                   .. versionadded:: 1.13
336: (4)               nan : int, float, optional
337: (8)                   Value to be used to fill NaN values. If no value is passed
338: (8)                   then NaN values will be replaced with 0.0.
339: (8)                   .. versionadded:: 1.17
340: (4)               posinf : int, float, optional
341: (8)                   Value to be used to fill positive infinity values. If no value is
342: (8)                   passed then positive infinity values will be replaced with a very
343: (8)                   large number.
344: (8)                   .. versionadded:: 1.17
345: (4)               neginf : int, float, optional
346: (8)                   Value to be used to fill negative infinity values. If no value is
347: (8)                   passed then negative infinity values will be replaced with a very
348: (8)                   small (or negative) number.
349: (8)                   .. versionadded:: 1.17
350: (4)               Returns
351: (4)               -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

352: (4)          out : ndarray
353: (8)            `x`, with the non-finite values replaced. If `copy` is False, this may
354: (8)            be `x` itself.
355: (4)          See Also
356: (4)            -----
357: (4)            isinf : Shows which elements are positive or negative infinity.
358: (4)            isneginf : Shows which elements are negative infinity.
359: (4)            isposinf : Shows which elements are positive infinity.
360: (4)            isnan : Shows which elements are Not a Number (NaN).
361: (4)            isfinite : Shows which elements are finite (not NaN, not infinity)
362: (4)          Notes
363: (4)            -----
364: (4)            NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
365: (4)            (IEEE 754). This means that Not a Number is not equivalent to infinity.
366: (4)          Examples
367: (4)            -----
368: (4)            >>> np.nan_to_num(np.inf)
369: (4)            1.7976931348623157e+308
370: (4)            >>> np.nan_to_num(-np.inf)
371: (4)            -1.7976931348623157e+308
372: (4)            >>> np.nan_to_num(np.nan)
373: (4)            0.0
374: (4)            >>> x = np.array([np.inf, -np.inf, np.nan, -128, 128])
375: (4)            >>> np.nan_to_num(x)
376: (4)            array([ 1.79769313e+308, -1.79769313e+308,  0.0000000e+000, # may vary
377: (11)              -1.2800000e+002,  1.2800000e+002])
378: (4)            >>> np.nan_to_num(x, nan=-9999, posinf=33333333, neginf=33333333)
379: (4)            array([ 3.3333333e+07,  3.3333333e+07, -9.9990000e+03,
380: (11)              -1.2800000e+02,  1.2800000e+02])
381: (4)            >>> y = np.array([complex(np.inf, np.nan), np.nan, complex(np.nan,
382: (4)              np.inf)])
383: (9)            array([ 1.79769313e+308, -1.79769313e+308,  0.0000000e+000, # may vary
384: (4)              -1.2800000e+002,  1.2800000e+002])
385: (4)            >>> np.nan_to_num(y)
386: (13)            array([ 1.79769313e+308 +0.0000000e+000j, # may vary
387: (13)              0.0000000e+000 +0.0000000e+000j,
388: (4)              0.0000000e+000 +1.79769313e+308j])
389: (4)            >>> np.nan_to_num(y, nan=111111, posinf=222222)
390: (4)            array([222222.+111111.j, 111111.      +0.j, 111111.+222222.j])
391: (4)            """
392: (4)            x = _nx.array(x, subok=True, copy=copy)
393: (4)            xtype = x.dtype.type
394: (4)            isscalar = (x.ndim == 0)
395: (8)            if not issubclass(xtype, _nx.inexact):
396: (4)                return x[()] if isscalar else x
397: (4)            iscomplex = issubclass(xtype, _nx.complexfloating)
398: (4)            dest = (x.real, x.imag) if iscomplex else (x,)
399: (4)            maxf, minf = _getmaxmin(x.real.dtype)
400: (8)            if posinf is not None:
401: (4)                maxf = posinf
402: (8)            if neginf is not None:
403: (4)                minf = neginf
404: (8)            for d in dest:
405: (8)                idx_nan = isnan(d)
406: (8)                idx_posinf = isposinf(d)
407: (8)                idx_neginf = isneginf(d)
408: (8)                _nx.copyto(d, nan, where=idx_nan)
409: (8)                _nx.copyto(d, maxf, where=idx_posinf)
410: (8)                _nx.copyto(d, minf, where=idx_neginf)
411: (0)            return x[()] if isscalar else x
412: (4)          def _real_if_close_dispatcher(a, tol=None):
413: (0)            return (a,)
414: (0)          @array_function_dispatch(_real_if_close_dispatcher)
415: (4)          def real_if_close(a, tol=100):
416: (4)            """
417: (4)            If input is complex with all imaginary parts close to zero, return
418: (4)            real parts.
419: (4)            "Close to zero" is defined as `tol` * (machine epsilon of the type for
419: (4)            `a`).

```

```

420: (4)          Parameters
421: (4)          -----
422: (4)          a : array_like
423: (8)          Input array.
424: (4)          tol : float
425: (8)          Tolerance in machine epsilons for the complex part of the elements
426: (8)          in the array. If the tolerance is <=1, then the absolute tolerance
427: (8)          is used.
428: (4)          Returns
429: (4)          -----
430: (4)          out : ndarray
431: (8)          If `a` is real, the type of `a` is used for the output. If `a`
432: (8)          has complex elements, the returned type is float.
433: (4)          See Also
434: (4)          -----
435: (4)          real, imag, angle
436: (4)          Notes
437: (4)          -----
438: (4)          Machine epsilon varies from machine to machine and between data types
439: (4)          but Python floats on most platforms have a machine epsilon equal to
440: (4)          2.2204460492503131e-16. You can use 'np.finfo(float).eps' to print
441: (4)          out the machine epsilon for floats.
442: (4)          Examples
443: (4)          -----
444: (4)          >>> np.finfo(float).eps
445: (4)          2.2204460492503131e-16 # may vary
446: (4)          >>> np.real_if_close([2.1 + 4e-14j, 5.2 + 3e-15j], tol=1000)
447: (4)          array([2.1, 5.2])
448: (4)          >>> np.real_if_close([2.1 + 4e-13j, 5.2 + 3e-15j], tol=1000)
449: (4)          array([2.1+4.e-13j, 5.2 + 3e-15j])
450: (4)          """
451: (4)          a = asanyarray(a)
452: (4)          type_ = a.dtype.type
453: (4)          if not issubclass(type_, _nx.complexfloating):
454: (8)              return a
455: (4)          if tol > 1:
456: (8)              f = getlimits.finfo(type_)
457: (8)              tol = f.eps * tol
458: (4)          if _nx.all(_nx.absolute(a.imag) < tol):
459: (8)              a = a.real
460: (4)          return a
461: (0)          _namefromtype = {'S1': 'character',
462: (17)                  '?': 'bool',
463: (17)                  'b': 'signed char',
464: (17)                  'B': 'unsigned char',
465: (17)                  'h': 'short',
466: (17)                  'H': 'unsigned short',
467: (17)                  'i': 'integer',
468: (17)                  'I': 'unsigned integer',
469: (17)                  'l': 'long integer',
470: (17)                  'L': 'unsigned long integer',
471: (17)                  'q': 'long long integer',
472: (17)                  'Q': 'unsigned long long integer',
473: (17)                  'f': 'single precision',
474: (17)                  'd': 'double precision',
475: (17)                  'g': 'long precision',
476: (17)                  'F': 'complex single precision',
477: (17)                  'D': 'complex double precision',
478: (17)                  'G': 'complex long double precision',
479: (17)                  'S': 'string',
480: (17)                  'U': 'unicode',
481: (17)                  'V': 'void',
482: (17)                  'O': 'object'
483: (17)                  }
484: (0)          @set_module('numpy')
485: (0)          def typename(char):
486: (4)          """
487: (4)          Return a description for the given data type code.
488: (4)          Parameters

```

```

489: (4)      -----
490: (4)      char : str
491: (8)      Data type code.
492: (4)      Returns
493: (4)      -----
494: (4)      out : str
495: (8)      Description of the input data type code.
496: (4)      See Also
497: (4)      -----
498: (4)      dtype, typecodes
499: (4)      Examples
500: (4)      -----
501: (4)      >>> typechars = ['S1', '?', 'B', 'D', 'G', 'F', 'I', 'H', 'L', 'O', 'Q',
502: (4)      ...           'S', 'U', 'V', 'b', 'd', 'g', 'f', 'i', 'h', 'l', 'q']
503: (4)      >>> for typechar in typechars:
504: (4)      ...     print(typechar, ' : ', np.typename(typechar))
505: (4)      ...
506: (4)      S1   : character
507: (4)      ?    : bool
508: (4)      B    : unsigned char
509: (4)      D    : complex double precision
510: (4)      G    : complex long double precision
511: (4)      F    : complex single precision
512: (4)      I    : unsigned integer
513: (4)      H    : unsigned short
514: (4)      L    : unsigned long integer
515: (4)      O    : object
516: (4)      Q    : unsigned long long integer
517: (4)      S    : string
518: (4)      U    : unicode
519: (4)      V    : void
520: (4)      b    : signed char
521: (4)      d    : double precision
522: (4)      g    : long precision
523: (4)      f    : single precision
524: (4)      i    : integer
525: (4)      h    : short
526: (4)      l    : long integer
527: (4)      q    : long long integer
528: (4)      """
529: (4)      return _namefromtype[char]
530: (0)      array_type = [[_nx.half, _nx.single, _nx.double, _nx.longdouble],
531: (14)          [None, _nx.csingle, _nx.cdouble, _nx.clongdouble]]
532: (0)      array_precision = {_nx.half: 0,
533: (19)          _nx.single: 1,
534: (19)          _nx.double: 2,
535: (19)          _nx.longdouble: 3,
536: (19)          _nx.csingle: 1,
537: (19)          _nx.cdouble: 2,
538: (19)          _nx.clongdouble: 3}
539: (0)      def _common_type_dispatcher(*arrays):
540: (4)          return arrays
541: (0)      @array_function_dispatch(_common_type_dispatcher)
542: (0)      def common_type(*arrays):
543: (4)          """
544: (4)          Return a scalar type which is common to the input arrays.
545: (4)          The return type will always be an inexact (i.e. floating point) scalar
546: (4)          type, even if all the arrays are integer arrays. If one of the inputs is
547: (4)          an integer array, the minimum precision type that is returned is a
548: (4)          64-bit floating point dtype.
549: (4)          All input arrays except int64 and uint64 can be safely cast to the
550: (4)          returned dtype without loss of information.
551: (4)          Parameters
552: (4)          -----
553: (4)          array1, array2, ... : ndarrays
554: (8)              Input arrays.
555: (4)          Returns
556: (4)          -----
557: (4)          out : data type code

```

```

558: (8)             Data type code.
559: (4)             See Also
560: (4)
561: (4)             -----
562: (4)             dtype, mintypecode
563: (4)
564: (4)             Examples
565: (4)
566: (4)             -----
567: (4)             >>> np.common_type(np.arange(2, dtype=np.float32))
568: (4)             <class 'numpy.float32'>
569: (4)             >>> np.common_type(np.arange(2, dtype=np.float32), np.arange(2))
570: (4)             <class 'numpy.float64'>
571: (4)             >>> np.common_type(np.arange(4), np.array([45, 6.j]), np.array([45.0]))
572: (4)             <class 'numpy.complex128'>
573: (4)
574: (8)             """
575: (8)             is_complex = False
576: (12)            precision = 0
577: (8)             for a in arrays:
578: (12)                t = a.dtype.type
579: (8)                if iscomplexobj(a):
580: (12)                    is_complex = True
581: (12)                if issubclass(t, _nx.integer):
582: (16)                    p = 2 # array_precision[_nx.double]
583: (8)                else:
584: (12)                    p = array_precision.get(t, None)
585: (8)                    if p is None:
586: (12)                        raise TypeError("can't get common type for non-numeric array")
587: (8)                    precision = max(precision, p)
588: (4)             if is_complex:
589: (8)                 return array_type[1][precision]
590: (4)             else:
591: (8)                 return array_type[0][precision]

```

-----

## File 222 - ufunclike.py:

```

1: (0)             """
2: (0)             Module of functions that are like ufuncs in acting on arrays and optionally
3: (0)             storing results in an output array.
4: (0)
5: (0)             __all__ = ['fix', 'isneginf', 'isposinf']
6: (0)             import numpy.core.numeric as nx
7: (0)             from numpy.core.overrides import array_function_dispatch
8: (0)             import warnings
9: (0)             import functools
10: (0)            def _dispatcher(x, out=None):
11: (4)                return (x, out)
12: (0)            @array_function_dispatch(_dispatcher, verify=False, module='numpy')
13: (0)            def fix(x, out=None):
14: (4)
15: (4)                Round to nearest integer towards zero.
16: (4)                Round an array of floats element-wise to nearest integer towards zero.
17: (4)                The rounded values are returned as floats.
18: (4)                Parameters
19: (4)
20: (4)                x : array_like
21: (8)                  An array of floats to be rounded
22: (4)                out : ndarray, optional
23: (8)                  A location into which the result is stored. If provided, it must have
24: (8)                  a shape that the input broadcasts to. If not provided or None, a
25: (8)                  freshly-allocated array is returned.
26: (4)                Returns
27: (4)
28: (4)                out : ndarray of floats
29: (8)                  A float array with the same dimensions as the input.
30: (8)                  If second argument is not supplied then a float array is returned
31: (8)                  with the rounded values.
32: (8)                  If a second argument is supplied the result is stored there.
33: (8)                  The return value `out` is then a reference to that array.
34: (4)                See Also

```

```

35: (4)      -----
36: (4)      rint, trunc, floor, ceil
37: (4)      around : Round to given number of decimals
38: (4)      Examples
39: (4)      -----
40: (4)      >>> np.fix(3.14)
41: (4)      3.0
42: (4)      >>> np.fix(3)
43: (4)      3.0
44: (4)      >>> np.fix([2.1, 2.9, -2.1, -2.9])
45: (4)      array([ 2.,  2., -2., -2.])
46: (4)      """
47: (4)      res = nx.asanyarray(nx.ceil(x, out=out))
48: (4)      res = nx.floor(x, out=res, where=nx.greater_equal(x, 0))
49: (4)      if out is None and type(res) is nx.ndarray:
50: (8)          res = res[()]
51: (4)      return res
52: (0)      @array_function_dispatch(_dispatcher, verify=False, module='numpy')
53: (0)      def isposinf(x, out=None):
54: (4)          """
55: (4)          Test element-wise for positive infinity, return result as bool array.
56: (4)          Parameters
57: (4)          -----
58: (4)          x : array_like
59: (8)              The input array.
60: (4)          out : array_like, optional
61: (8)              A location into which the result is stored. If provided, it must have
a
62: (8)              shape that the input broadcasts to. If not provided or None, a
63: (8)              freshly-allocated boolean array is returned.
64: (4)          Returns
65: (4)          -----
66: (4)          out : ndarray
67: (8)              A boolean array with the same dimensions as the input.
68: (8)              If second argument is not supplied then a boolean array is returned
69: (8)              with values True where the corresponding element of the input is
70: (8)              positive infinity and values False where the element of the input is
71: (8)              not positive infinity.
72: (8)              If a second argument is supplied the result is stored there. If the
73: (8)              type of that array is a numeric type the result is represented as
zeros
74: (8)              and ones, if the type is boolean then as False and True.
75: (8)              The return value `out` is then a reference to that array.
76: (4)          See Also
77: (4)          -----
78: (4)          isnan, isneginf, isfinite, isnan
79: (4)          Notes
80: (4)          -----
81: (4)          NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
82: (4)          (IEEE 754).
83: (4)          Errors result if the second argument is also supplied when x is a scalar
84: (4)          input, if first and second arguments have different shapes, or if the
85: (4)          first argument has complex values
86: (4)          Examples
87: (4)          -----
88: (4)          >>> np.isposinf(np.PINF)
89: (4)          True
90: (4)          >>> np.isposinf(np.inf)
91: (4)          True
92: (4)          >>> np.isposinf(np.NINF)
93: (4)          False
94: (4)          >>> np.isposinf([-np.inf, 0., np.inf])
95: (4)          array([False, False, True])
96: (4)          >>> x = np.array([-np.inf, 0., np.inf])
97: (4)          >>> y = np.array([2, 2, 2])
98: (4)          >>> np.isposinf(x, y)
99: (4)          array([0, 0, 1])
100: (4)         >>> y
101: (4)         array([0, 0, 1])

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

102: (4)
103: (4)
104: (4)
105: (8)
106: (4)
107: (8)
108: (8)
109: (24)
110: (4)
111: (8)
112: (0)
113: (0)
114: (4)
115: (4)
116: (4)
117: (4)
118: (4)
119: (8)
120: (4)
121: (8)
a
122: (8)
123: (8)
124: (4)
125: (4)
126: (4)
127: (8)
128: (8)
129: (8)
130: (8)
131: (8)
132: (8)
133: (8)
134: (8)
135: (8)
136: (4)
137: (4)
138: (4)
139: (4)
140: (4)
141: (4)
142: (4)
143: (4)
144: (4)
145: (4)
146: (4)
147: (4)
148: (4)
149: (4)
150: (4)
151: (4)
152: (4)
153: (4)
154: (4)
155: (4)
156: (4)
157: (4)
158: (4)
159: (4)
160: (4)
161: (4)
162: (4)
163: (4)
164: (4)
165: (8)
166: (4)
167: (8)
168: (8)
169: (24)

        """
        is_inf = nx.isinf(x)
        try:
            signbit = ~nx.signbit(x)
        except TypeError as e:
            dtype = nx.asanyarray(x).dtype
            raise TypeError(f'This operation is not supported for {dtype} values '
                           'because it would be ambiguous.') from e
        else:
            return nx.logical_and(is_inf, signbit, out)
    @array_function_dispatch(_dispatcher, verify=False, module='numpy')
    def isneginf(x, out=None):
        """
        Test element-wise for negative infinity, return result as bool array.

        Parameters
        -----
        x : array_like
            The input array.
        out : array_like, optional
            A location into which the result is stored. If provided, it must have
            shape that the input broadcasts to. If not provided or None, a
            freshly-allocated boolean array is returned.

        Returns
        -----
        out : ndarray
            A boolean array with the same dimensions as the input.
            If second argument is not supplied then a numpy boolean array is
            returned with values True where the corresponding element of the
            input is negative infinity and values False where the element of
            the input is not negative infinity.
            If a second argument is supplied the result is stored there. If the
            type of that array is a numeric type the result is represented as
            zeros and ones, if the type is boolean then as False and True. The
            return value `out` is then a reference to that array.

        See Also
        -----
        isinf, isposinf, isnan, isfinite
        Notes
        -----
        NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic
        (IEEE 754).
        Errors result if the second argument is also supplied when x is a scalar
        input, if first and second arguments have different shapes, or if the
        first argument has complex values.

        Examples
        -----
        >>> np.isneginf(np.NINF)
        True
        >>> np.isneginf(np.inf)
        False
        >>> np.isneginf(np.PINF)
        False
        >>> np.isneginf([-np.inf, 0., np.inf])
        array([ True, False, False])
        >>> x = np.array([-np.inf, 0., np.inf])
        >>> y = np.array([2, 2, 2])
        >>> np.isneginf(x, y)
        array([1, 0, 0])
        >>> y
        array([1, 0, 0])
        """
        is_inf = nx.isinf(x)
        try:
            signbit = nx.signbit(x)
        except TypeError as e:
            dtype = nx.asanyarray(x).dtype
            raise TypeError(f'This operation is not supported for {dtype} values '
                           'because it would be ambiguous.') from e

```

```
170: (4)         else:
171: (8)             return nx.logical_and(is_inf, signbit, out)
```

-----  
File 223 - user\_array.py:

```
1: (0)         """
2: (0)             Standard container-class for easy multiple-inheritance.
3: (0)             Try to inherit from the ndarray instead of using this class as this is not
4: (0)             complete.
5: (0)         """
6: (0)             from numpy.core import (
7: (4)                 array, asarray, absolute, add, subtract, multiply, divide,
8: (4)                 remainder, power, left_shift, right_shift, bitwise_and, bitwise_or,
9: (4)                 bitwise_xor, invert, less, less_equal, not_equal, equal, greater,
10: (4)                greater_equal, shape, reshape, arange, sin, sqrt, transpose
11: (0)
12: (0)         class container:
13: (4)             """
14: (4)                 container(data, dtype=None, copy=True)
15: (4)             Standard container-class for easy multiple-inheritance.
16: (4)             Methods
17: (4)             -----
18: (4)                 copy
19: (4)                 tostring
20: (4)                 byteswap
21: (4)                 astype
22: (4)             """
23: (4)             def __init__(self, data, dtype=None, copy=True):
24: (8)                 self.array = array(data, dtype, copy=copy)
25: (4)             def __repr__(self):
26: (8)                 if self.ndim > 0:
27: (12)                     return self.__class__.__name__ + repr(self.array)[len("array"):]
28: (8)                 else:
29: (12)                     return self.__class__.__name__ + "(" + repr(self.array) + ")"
30: (4)             def __array__(self, t=None):
31: (8)                 if t:
32: (12)                     return self.array.astype(t)
33: (8)                 return self.array
34: (4)             def __len__(self):
35: (8)                 return len(self.array)
36: (4)             def __getitem__(self, index):
37: (8)                 return self._rc(self.array[index])
38: (4)             def __setitem__(self, index, value):
39: (8)                 self.array[index] = asarray(value, self.dtype)
40: (4)             def __abs__(self):
41: (8)                 return self._rc(absolute(self.array))
42: (4)             def __neg__(self):
43: (8)                 return self._rc(-self.array)
44: (4)             def __add__(self, other):
45: (8)                 return self._rc(self.array + asarray(other))
46: (4)             __radd__ = __add__
47: (4)             def __iadd__(self, other):
48: (8)                 add(self.array, other, self.array)
49: (8)                 return self
50: (4)             def __sub__(self, other):
51: (8)                 return self._rc(self.array - asarray(other))
52: (4)             def __rsub__(self, other):
53: (8)                 return self._rc(asarray(other) - self.array)
54: (4)             def __isub__(self, other):
55: (8)                 subtract(self.array, other, self.array)
56: (8)                 return self
57: (4)             def __mul__(self, other):
58: (8)                 return self._rc(multiply(self.array, asarray(other)))
59: (4)             __rmul__ = __mul__
60: (4)             def __imul__(self, other):
61: (8)                 multiply(self.array, other, self.array)
62: (8)                 return self
```

```

63: (4)           def __div__(self, other):
64: (8)             return self._rc(divide(self.array, asarray(other)))
65: (4)           def __rdiv__(self, other):
66: (8)             return self._rc(divide(asarray(other), self.array))
67: (4)           def __idiv__(self, other):
68: (8)             divide(self.array, other, self.array)
69: (8)             return self
70: (4)           def __mod__(self, other):
71: (8)             return self._rc(remainder(self.array, other))
72: (4)           def __rmod__(self, other):
73: (8)             return self._rc(remainder(other, self.array))
74: (4)           def __imod__(self, other):
75: (8)             remainder(self.array, other, self.array)
76: (8)             return self
77: (4)           def __divmod__(self, other):
78: (8)             return (self._rc(divide(self.array, other)),
79: (16)                   self._rc(remainder(self.array, other)))
80: (4)           def __rdivmod__(self, other):
81: (8)             return (self._rc(divide(other, self.array)),
82: (16)                   self._rc(remainder(other, self.array)))
83: (4)           def __pow__(self, other):
84: (8)             return self._rc(power(self.array, asarray(other)))
85: (4)           def __rpow__(self, other):
86: (8)             return self._rc(power(asarray(other), self.array))
87: (4)           def __ipow__(self, other):
88: (8)             power(self.array, other, self.array)
89: (8)             return self
90: (4)           def __lshift__(self, other):
91: (8)             return self._rc(left_shift(self.array, other))
92: (4)           def __rshift__(self, other):
93: (8)             return self._rc(right_shift(self.array, other))
94: (4)           def __rlshift__(self, other):
95: (8)             return self._rc(left_shift(other, self.array))
96: (4)           def __rrshift__(self, other):
97: (8)             return self._rc(right_shift(other, self.array))
98: (4)           def __ilshift__(self, other):
99: (8)             left_shift(self.array, other, self.array)
100: (8)            return self
101: (4)           def __irshift__(self, other):
102: (8)             right_shift(self.array, other, self.array)
103: (8)             return self
104: (4)           def __and__(self, other):
105: (8)             return self._rc(bitwise_and(self.array, other))
106: (4)           def __rand__(self, other):
107: (8)             return self._rc(bitwise_and(other, self.array))
108: (4)           def __iand__(self, other):
109: (8)             bitwise_and(self.array, other, self.array)
110: (8)             return self
111: (4)           def __xor__(self, other):
112: (8)             return self._rc(bitwise_xor(self.array, other))
113: (4)           def __rxor__(self, other):
114: (8)             return self._rc(bitwise_xor(other, self.array))
115: (4)           def __ixor__(self, other):
116: (8)             bitwise_xor(self.array, other, self.array)
117: (8)             return self
118: (4)           def __or__(self, other):
119: (8)             return self._rc(bitwise_or(self.array, other))
120: (4)           def __ror__(self, other):
121: (8)             return self._rc(bitwise_or(other, self.array))
122: (4)           def __ior__(self, other):
123: (8)             bitwise_or(self.array, other, self.array)
124: (8)             return self
125: (4)           def __pos__(self):
126: (8)             return self._rc(self.array)
127: (4)           def __invert__(self):
128: (8)             return self._rc(invert(self.array))
129: (4)           def __scalarfunc(self, func):
130: (8)             if self.ndim == 0:
131: (12)               return func(self[0])

```

```

132: (8)
133: (12)
134: (16)
135: (4)
136: (8)
137: (4)
138: (8)
139: (4)
140: (8)
141: (4)
142: (8)
143: (4)
144: (8)
145: (4)
146: (8)
147: (4)
148: (8)
149: (4)
150: (8)
151: (4)
152: (8)
153: (4)
154: (8)
155: (4)
156: (8)
157: (4)
158: (8)
159: (8)
160: (4)
161: (8)
162: (8)
163: (4)
164: (8)
165: (8)
166: (4)
167: (8)
168: (8)
169: (4)
170: (8)
171: (8)
172: (4)
173: (8)
174: (12)
175: (8)
176: (12)
177: (4)
178: (8)
179: (4)
180: (8)
181: (12)
182: (12)
183: (8)
184: (12)
185: (8)
186: (12)
187: (4)
188: (8)
189: (12)
190: (8)
191: (0)
192: (4)
193: (4)
194: (4)
195: (4)
196: (4)
197: (4)
198: (4)
199: (4)
200: (4)

        else:
            raise TypeError(
                "only rank-0 arrays can be converted to Python scalars.")
    def __complex__(self):
        return self._scalarfunc(complex)
    def __float__(self):
        return self._scalarfunc(float)
    def __int__(self):
        return self._scalarfunc(int)
    def __hex__(self):
        return self._scalarfunc(hex)
    def __oct__(self):
        return self._scalarfunc(oct)
    def __lt__(self, other):
        return self._rc(less(self.array, other))
    def __le__(self, other):
        return self._rc(less_equal(self.array, other))
    def __eq__(self, other):
        return self._rc(equal(self.array, other))
    def __ne__(self, other):
        return self._rc(not_equal(self.array, other))
    def __gt__(self, other):
        return self._rc(greater(self.array, other))
    def __ge__(self, other):
        return self._rc(greater_equal(self.array, other))
    def copy(self):
        """
        return self._rc(self.array.copy())
    def tostring(self):
        """
        return self.array.tostring()
    def tobytes(self):
        """
        return self.array.tobytes()
    def byteswap(self):
        """
        return self._rc(self.array.byteswap())
    def astype(self, typecode):
        """
        return self._rc(self.array.astype(typecode))
    def __rc__(self, a):
        if len(shape(a)) == 0:
            return a
        else:
            return self.__class__(a)
    def __array_wrap__(self, *args):
        return self.__class__(args[0])
    def __setattr__(self, attr, value):
        if attr == 'array':
            object.__setattr__(self, attr, value)
            return
        try:
            self.array.__setattr__(attr, value)
        except AttributeError:
            object.__setattr__(self, attr, value)
    def __getattr__(self, attr):
        if (attr == 'array'):
            return object.__getattribute__(self, attr)
        return self.array.__getattribute__(attr)
if __name__ == '__main__':
    temp = reshape(arange(10000), (100, 100))
    ua = container(temp)
    print(dir(ua))
    print(shape(ua), ua.shape) # I have changed Numeric.py
    ua_small = ua[:, :5]
    print(ua_small)
    ua_small[0, 0] = 10
    print(ua_small[0, 0], ua[0, 0])
    print(sin(ua_small) / 3. * 6. + sqrt(ua_small ** 2))

```

```

201: (4)         print(less(ua_small, 103), type(less(ua_small, 103)))
202: (4)         print(type(ua_small * reshape(arange(15), shape(ua_small))))
203: (4)         print(reshape(ua_small, (5, 3)))
204: (4)         print(transpose(ua_small))
-----
```

File 224 - utils.py:

```

1: (0)             import os
2: (0)             import sys
3: (0)             import textwrap
4: (0)             import types
5: (0)             import re
6: (0)             import warnings
7: (0)             import functools
8: (0)             import platform
9: (0)             from ..utils import set_module
10: (0)            from numpy.core.numeric import issubclass_, issubsctype, issubdtype
11: (0)            from numpy.core import ndarray, ufunc, asarray
12: (0)            import numpy as np
13: (0)            __all__ = [
14: (4)                'issubclass_', 'issubsctype', 'issubdtype', 'deprecate',
15: (4)                'deprecate_with_doc', 'get_include', 'info', 'source', 'who',
16: (4)                'lookfor', 'byte_bounds', 'safe_eval', 'show_runtime'
17: (4)            ]
18: (0)            def show_runtime():
19: (4)                """
20: (4)                    Print information about various resources in the system
21: (4)                    including available intrinsic support and BLAS/LAPACK library
22: (4)                    in use
23: (4)                    .. versionadded:: 1.24.0
24: (4)                    See Also
25: (4)                    -----
26: (4)                    show_config : Show libraries in the system on which NumPy was built.
27: (4)                    Notes
28: (4)                    -----
29: (4)                    1. Information is derived with the help of `threadpoolctl
<https://pypi.org/project/threadpoolctl/>`_
30: (7)                        library if available.
31: (4)                    2. SIMD related information is derived from ``__cpu_features__``,
32: (7)                        ``__cpu_baseline__`` and ``__cpu_dispatch__``
33: (4)                    """
34: (4)                    from numpy.core._multiarray_umath import (
35: (8)                        __cpu_features__, __cpu_baseline__, __cpu_dispatch__
36: (4)                    )
37: (4)                    from pprint import pprint
38: (4)                    config_found = [
39: (8)                        "numpy_version": np.__version__,
40: (8)                        "python": sys.version,
41: (8)                        "uname": platform.uname(),
42: (8)                    ]
43: (4)                    features_found, features_not_found = [], []
44: (4)                    for feature in __cpu_dispatch__:
45: (8)                        if __cpu_features__[feature]:
46: (12)                            features_found.append(feature)
47: (8)                        else:
48: (12)                            features_not_found.append(feature)
49: (4)                    config_found.append({
50: (8)                        "simd_extensions": {
51: (12)                            "baseline": __cpu_baseline__,
52: (12)                            "found": features_found,
53: (12)                            "not_found": features_not_found
54: (8)                        }
55: (4)                    })
56: (4)                    try:
57: (8)                        from threadpoolctl import threadpool_info
58: (8)                        config_found.extend(threadpool_info())
59: (4)                    except ImportError:
```

```

60: (8)          print("WARNING: `threadpoolctl` not found in system!")
61: (14)          " Install it by `pip install threadpoolctl`."
62: (14)          " Once installed, try `np.show_runtime` again"
63: (14)          " for more detailed build information")
64: (4)          pprint(config_found)
65: (0)      def get_include():
66: (4)          """
67: (4)              Return the directory that contains the NumPy \*.h header files.
68: (4)              Extension modules that need to compile against NumPy should use this
69: (4)              function to locate the appropriate include directory.
70: (4)          Notes
71: (4)          -----
72: (4)              When using ``distutils``, for example in ``setup.py``::
73: (8)                  import numpy as np
74: (8)                  ...
75: (8)                  Extension('extension_name', ...
76: (16)                      include_dirs=[np.get_include()])
77: (8)                  ...
78: (4)          """
79: (4)          import numpy
80: (4)          if numpy.show_config is None:
81: (8)              d = os.path.join(os.path.dirname(numpy.__file__), 'core', 'include')
82: (4)          else:
83: (8)              import numpy.core as core
84: (8)              d = os.path.join(os.path.dirname(core.__file__), 'include')
85: (4)          return d
86: (0)      class _Deprecate:
87: (4)          """
88: (4)              Decorator class to deprecate old functions.
89: (4)              Refer to `deprecate` for details.
90: (4)              See Also
91: (4)              -----
92: (4)              deprecate
93: (4)          """
94: (4)          def __init__(self, old_name=None, new_name=None, message=None):
95: (8)              self.old_name = old_name
96: (8)              self.new_name = new_name
97: (8)              self.message = message
98: (4)          def __call__(self, func, *args, **kwargs):
99: (8)              """
100: (8)                  Decorator call. Refer to ``decorate``.
101: (8)              """
102: (8)              old_name = self.old_name
103: (8)              new_name = self.new_name
104: (8)              message = self.message
105: (8)              if old_name is None:
106: (12)                  old_name = func.__name__
107: (8)              if new_name is None:
108: (12)                  depdoc = "`%s` is deprecated!" % old_name
109: (8)              else:
110: (12)                  depdoc = "`%s` is deprecated, use `%s` instead!" % \
111: (21)                      (old_name, new_name)
112: (8)              if message is not None:
113: (12)                  depdoc += "\n" + message
114: (8)              @functools.wraps(func)
115: (8)              def newfunc(*args, **kwds):
116: (12)                  warnings.warn(depdoc, DeprecationWarning, stacklevel=2)
117: (12)                  return func(*args, **kwds)
118: (8)              newfunc.__name__ = old_name
119: (8)              doc = func.__doc__
120: (8)              if doc is None:
121: (12)                  doc = depdoc
122: (8)              else:
123: (12)                  lines = doc.expandtabs().split('\n')
124: (12)                  indent = _get_indent(lines[1:])
125: (12)                  if lines[0].lstrip():
126: (16)                      doc = indent * ' ' + doc
127: (12)                  else:
128: (16)                      skip = len(lines[0]) + 1

```

```

129: (16)                         for line in lines[1:]:
130: (20)                         if len(line) > indent:
131: (24)                             break
132: (20)                         skip += len(line) + 1
133: (16)                         doc = doc[skip:]
134: (12)                         depdoc = textwrap.indent(depdoc, ' ' * indent)
135: (12)                         doc = '\n\n'.join([depdoc, doc])
136: (8)                         newfunc.__doc__ = doc
137: (8)                         return newfunc
138: (0) def __get_indent(lines):
139: (4)     """
140: (4)         Determines the leading whitespace that could be removed from all the
141: (4)         lines.
142: (4)     """
143: (4)     indent = sys.maxsize
144: (8)     for line in lines:
145: (8)         content = len(line.lstrip())
146: (12)         if content:
147: (4)             indent = min(indent, len(line) - content)
148: (8)     if indent == sys.maxsize:
149: (4)         indent = 0
150: (0)     return indent
151: (4) def deprecate(*args, **kwargs):
152: (4)     """
153: (4)         Issues a DeprecationWarning, adds warning to `old_name`'s
154: (4)         docstring, rebinds ``old_name.__name__`` and returns the new
155: (4)         function object.
156: (4)         This function may also be used as a decorator.
157: (4)         Parameters
158: (4)         -----
159: (8)         func : function
160: (4)             The function to be deprecated.
161: (8)         old_name : str, optional
162: (4)             The name of the function to be deprecated. Default is None, in
163: (4)             which case the name of `func` is used.
164: (8)         new_name : str, optional
165: (4)             The new name for the function. Default is None, in which case the
166: (4)             deprecation message is that `old_name` is deprecated. If given, the
167: (4)             deprecation message is that `old_name` is deprecated and `new_name`
168: (4)             should be used instead.
169: (8)         message : str, optional
170: (4)             Additional explanation of the deprecation. Displayed in the
171: (4)             docstring after the warning.
172: (4)         Returns
173: (4)         -----
174: (8)         old_func : function
175: (4)             The deprecated function.
176: (4)         Examples
177: (4)         -----
178: (4)         Note that ``olduint`` returns a value after printing Deprecation
179: (4)         Warning:
180: (4)         >>> olduint = np.deprecate(np.uint)
181: (4)         DeprecationWarning: `uint64` is deprecated! # may vary
182: (4)         >>> olduint(6)
183: (4)         6
184: (4)         """
185: (8)         if args:
186: (8)             fn = args[0]
187: (8)             args = args[1:]
188: (4)             return _Deprecate(*args, **kwargs)(fn)
189: (8)         else:
190: (4)             return _Deprecate(*args, **kwargs)
191: (4) def deprecate_with_doc(msg):
192: (4)     """
193: (4)         Deprecates a function and includes the deprecation in its docstring.
194: (4)         This function is used as a decorator. It returns an object that can be
195: (4)         used to issue a DeprecationWarning, by passing the to-be decorated
196: (4)         function as argument, this adds warning to the to-be decorated function's

```

```

197: (4)           See Also
198: (4)           -----
199: (4)           deprecate : Decorate a function such that it issues a `DeprecationWarning`
200: (4)           Parameters
201: (4)           -----
202: (4)           msg : str
203: (8)             Additional explanation of the deprecation. Displayed in the
204: (8)             docstring after the warning.
205: (4)           Returns
206: (4)           -----
207: (4)           obj : object
208: (4)           """
209: (4)           return _Deprecate(message=msg)
210: (0) def byte_bounds(a):
211: (4)           """
212: (4)             Returns pointers to the end-points of an array.
213: (4)             Parameters
214: (4)             -----
215: (4)             a : ndarray
216: (8)               Input array. It must conform to the Python-side of the array
217: (8)               interface.
218: (4)             Returns
219: (4)             -----
220: (4)             (low, high) : tuple of 2 integers
221: (8)               The first integer is the first byte of the array, the second
222: (8)               integer is just past the last byte of the array. If `a` is not
223: (8)               contiguous it will not use every byte between the (`low`, `high`)
224: (8)               values.
225: (4)             Examples
226: (4)             -----
227: (4)             >>> I = np.eye(2, dtype='f'); I.dtype
228: (4)               dtype('float32')
229: (4)             >>> low, high = np.byte_bounds(I)
230: (4)             >>> high - low == I.size*I.itemsize
231: (4)               True
232: (4)             >>> I = np.eye(2); I.dtype
233: (4)               dtype('float64')
234: (4)             >>> low, high = np.byte_bounds(I)
235: (4)             >>> high - low == I.size*I.itemsize
236: (4)               True
237: (4)               """
238: (4)               ai = a.__array_interface__
239: (4)               a_data = ai['data'][0]
240: (4)               astrides = ai['strides']
241: (4)               ashape = ai['shape']
242: (4)               bytes_a = asarray(a).dtype.itemsize
243: (4)               a_low = a_high = a_data
244: (4)               if astrides is None:
245: (8)                 a_high += a.size * bytes_a
246: (4)               else:
247: (8)                 for shape, stride in zip(ashape, astrides):
248: (12)                   if stride < 0:
249: (16)                     a_low += (shape-1)*stride
250: (12)                   else:
251: (16)                     a_high += (shape-1)*stride
252: (8)               a_high += bytes_a
253: (4)               return a_low, a_high
254: (0) def who(vardict=None):
255: (4)               """
256: (4)                 Print the NumPy arrays in the given dictionary.
257: (4)                 If there is no dictionary passed in or `vardict` is None then returns
258: (4)                 NumPy arrays in the globals() dictionary (all NumPy arrays in the
259: (4)                 namespace).
260: (4)                 Parameters
261: (4)                 -----
262: (4)                 vardict : dict, optional
263: (8)                   A dictionary possibly containing ndarrays. Default is globals().
264: (4)                 Returns
265: (4)                 -----

```

```

266: (4)          out : None
267: (8)          Returns 'None'.
268: (4)          Notes
269: (4)
270: (4)          -----
271: (4)          Prints out the name, shape, bytes and type of all of the ndarrays
272: (4)          present in `vardict`.
273: (4)          Examples
274: (4)          -----
275: (4)          >>> a = np.arange(10)
276: (4)          >>> b = np.ones(20)
277: (4)          >>> np.who()
278: (4)          Name           Shape           Bytes           Type
279: (4)          =====
280: (4)          a            10             80            int64
281: (4)          b            20            160           float64
282: (4)          Upper bound on total bytes =      240
283: (4)          >>> d = {'x': np.arange(2.0), 'y': np.arange(3.0), 'txt': 'Some str',
284: (4)          ... 'idx':5}
285: (4)          >>> np.who(d)
286: (4)          Name           Shape           Bytes           Type
287: (4)          =====
288: (4)          x            2              16            float64
289: (4)          y            3              24            float64
290: (4)          Upper bound on total bytes =      40
291: (4)          """
292: (4)          if vardict is None:
293: (8)          frame = sys._getframe().f_back
294: (8)          vardict = frame.f_globals
295: (4)          sta = []
296: (4)          cache = {}
297: (4)          for name in vardict.keys():
298: (8)          if isinstance(vardict[name], ndarray):
299: (12)          var = vardict[name]
300: (12)          idv = id(var)
301: (16)          if idv in cache.keys():
302: (16)          namestr = name + " (%s)" % cache[idv]
303: (12)          original = 0
304: (16)          else:
305: (16)          cache[idv] = name
306: (16)          namestr = name
307: (16)          original = 1
308: (12)          shapestr = " x ".join(map(str, var.shape))
309: (12)          bytestr = str(var.nbytes)
310: (24)          sta.append([namestr, shapestr, bytestr, var.dtype.name,
311: (4)          original])
312: (4)          maxname = 0
313: (4)          maxshape = 0
314: (4)          maxbyte = 0
315: (4)          totalbytes = 0
316: (4)          for val in sta:
317: (8)          if maxname < len(val[0]):
318: (12)          maxname = len(val[0])
319: (8)          if maxshape < len(val[1]):
320: (12)          maxshape = len(val[1])
321: (8)          if maxbyte < len(val[2]):
322: (12)          maxbyte = len(val[2])
323: (8)          if val[4]:
324: (12)          totalbytes += int(val[2])
325: (4)          if len(sta) > 0:
326: (8)          sp1 = max(10, maxname)
327: (8)          sp2 = max(10, maxshape)
328: (8)          sp3 = max(10, maxbyte)
329: (8)          prval = "Name %s Shape %s Bytes %s Type" % (sp1*' ', sp2*' ', sp3*' ')
330: (4)          print(prval + "\n" + "="*(len(prval)+5) + "\n")
331: (8)          for val in sta:
332: (40)          print("%s %s %s %s %s %s" % (val[0], ' '* (sp1-len(val[0])+4),
333: (40)                                val[1], ' '* (sp2-len(val[1])+5),
334: (40)                                val[2], ' '* (sp3-len(val[2])+5),
334: (40)                                val[3]))
```

```

335: (4)           print("\nUpper bound on total bytes = %d" % totalbytes)
336: (4)           return
337: (0)           def _split_line(name, arguments, width):
338: (4)             firstwidth = len(name)
339: (4)             k = firstwidth
340: (4)             newstr = name
341: (4)             sepstr = ", "
342: (4)             arglist = arguments.split(sepstr)
343: (4)             for argument in arglist:
344: (8)               if k == firstwidth:
345: (12)                 addstr = ""
346: (8)               else:
347: (12)                 addstr = sepstr
348: (8)               k = k + len(argument) + len(addstr)
349: (8)               if k > width:
350: (12)                 k = firstwidth + 1 + len(argument)
351: (12)                 newstr = newstr + ",\n" + "*"*(firstwidth+2) + argument
352: (8)               else:
353: (12)                 newstr = newstr + addstr + argument
354: (4)             return newstr
355: (0)           _namedict = None
356: (0)           _dictlist = None
357: (0)           def _makenamedict(module='numpy'):
358: (4)             module = __import__(module, globals(), locals(), [])
359: (4)             thedict = {module.__name__:module.__dict__}
360: (4)             dictlist = [module.__name__]
361: (4)             totraverse = [module.__dict__]
362: (4)             while True:
363: (8)               if len(totraverse) == 0:
364: (12)                 break
365: (8)               thisdict = totraverse.pop(0)
366: (8)               for x in thisdict.keys():
367: (12)                 if isinstance(thisdict[x], types.ModuleType):
368: (16)                   modname = thisdict[x].__name__
369: (16)                   if modname not in dictlist:
370: (20)                     moddict = thisdict[x].__dict__
371: (20)                     dictlist.append(modname)
372: (20)                     totraverse.append(moddict)
373: (20)                     thedict[modname] = moddict
374: (4)             return thedict, dictlist
375: (0)           def _info(obj, output=None):
376: (4)             """Provide information about ndarray obj.
377: (4)             Parameters
378: (4)             -----
379: (4)             obj : ndarray
380: (8)               Must be ndarray, not checked.
381: (4)             output
382: (8)               Where printed output goes.
383: (4)             Notes
384: (4)             -----
385: (4)             Copied over from the numarray module prior to its removal.
386: (4)             Adapted somewhat as only numpy is an option now.
387: (4)             Called by info.
388: (4)             """
389: (4)             extra = ""
390: (4)             tic = ""
391: (4)             bp = lambda x: x
392: (4)             cls = getattr(obj, '__class__', type(obj))
393: (4)             nm = getattr(cls, '__name__', cls)
394: (4)             strides = obj.strides
395: (4)             endian = obj.dtype.byteorder
396: (4)             if output is None:
397: (8)               output = sys.stdout
398: (4)             print("class: ", nm, file=output)
399: (4)             print("shape: ", obj.shape, file=output)
400: (4)             print("strides: ", strides, file=output)
401: (4)             print("itemsize: ", obj.itemsize, file=output)
402: (4)             print("aligned: ", bp(obj.flags.aligned), file=output)
403: (4)             print("contiguous: ", bp(obj.flags.contiguous), file=output)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

404: (4)          print("fortran: ", obj.flags.fortran, file=output)
405: (4)          print(
406: (8)            "data pointer: %s%s" % (hex(obj.ctypes._as_parameter_.value), extra),
407: (8)            file=output
408: (8)            )
409: (4)          print("byteorder: ", end=' ', file=output)
410: (4)          if endian in ['|', '=']:
411: (8)            print("%s%s%s" % (tic, sys.byteorder, tic), file=output)
412: (8)            byteswap = False
413: (4)          elif endian == '>':
414: (8)            print("%sbigr%b" % (tic, tic), file=output)
415: (8)            byteswap = sys.byteorder != "big"
416: (4)          else:
417: (8)            print("%slittle%b" % (tic, tic), file=output)
418: (8)            byteswap = sys.byteorder != "little"
419: (4)          print("byteswap: ", bp(byteswap), file=output)
420: (4)          print("type: %s" % obj.dtype, file=output)
421: (0)          @set_module('numpy')
422: (0)          def info(object=None, maxwidth=76, output=None, toplevel='numpy'):
423: (4)            """
424: (4)              Get help information for an array, function, class, or module.
425: (4)              Parameters
426: (4)              -----
427: (4)              object : object or str, optional
428: (8)                Input object or name to get information about. If `object` is
429: (8)                  an `ndarray` instance, information about the array is printed.
430: (8)                  If `object` is a numpy object, its docstring is given. If it is
431: (8)                  a string, available modules are searched for matching objects.
432: (8)                  If None, information about `info` itself is returned.
433: (4)              maxwidth : int, optional
434: (8)                Printing width.
435: (4)              output : file like object, optional
436: (8)                File like object that the output is written to, default is
437: (8)                  ``None``, in which case ``sys.stdout`` will be used.
438: (8)                  The object has to be opened in 'w' or 'a' mode.
439: (4)              toplevel : str, optional
440: (8)                Start search at this level.
441: (4)              See Also
442: (4)              -----
443: (4)              source, lookfor
444: (4)              Notes
445: (4)              -----
446: (4)              When used interactively with an object, ``np.info(obj)`` is equivalent
447: (4)              to ``help(obj)`` on the Python prompt or ``obj?`` on the IPython
448: (4)              prompt.
449: (4)              Examples
450: (4)              -----
451: (4)              >>> np.info(np.polyval) # doctest: +SKIP
452: (7)                polyval(p, x)
453: (9)                  Evaluate the polynomial p at x.
454: (9)
455: (4)                  ...
456: (4)                  When using a string for `object` it is possible to get multiple results.
457: (4)                  >>> np.info('fft') # doctest: +SKIP
458: (9)                    *** Found in numpy ***
459: (4)                    Core FFT routines
460: (4)                    ...
461: (9)                      *** Found in numpy.fft ***
462: (4)                      fft(a, n=None, axis=-1)
463: (4)                      ...
464: (9)                        *** Repeat reference found in numpy.fft.fftpack ***
465: (9)                        *** Total of 3 references found. ***
466: (4)                        When the argument is an array, information about the array is printed.
467: (4)                        >>> a = np.array([[1 + 2j, 3, -4], [-5j, 6, 0]], dtype=np.complex64)
468: (4)                        >>> np.info(a)
469: (4)                          class: ndarray
470: (4)                          shape: (2, 3)
471: (4)                          strides: (24, 8)
472: (4)                          itemsize: 8
473: (4)                          aligned: True

```

```

473: (4)             contiguous:  True
474: (4)             fortran:   False
475: (4)             data pointer: 0x562b6e0d2860 # may vary
476: (4)             byteorder: little
477: (4)             byteswap:  False
478: (4)             type:     complex64
479: (4)             """
480: (4)             global _namedict, _dictlist
481: (4)             import pydoc
482: (4)             import inspect
483: (4)             if (hasattr(object, '_ppimport_importer') or
484: (11)                 hasattr(object, '_ppimport_module')):
485: (8)                 object = object._ppimport_module
486: (4)             elif hasattr(object, '_ppimport_attr'):
487: (8)                 object = object._ppimport_attr
488: (4)             if output is None:
489: (8)                 output = sys.stdout
490: (4)             if object is None:
491: (8)                 info(info)
492: (4)             elif isinstance(object, ndarray):
493: (8)                 _info(object, output=output)
494: (4)             elif isinstance(object, str):
495: (8)                 if _namedict is None:
496: (12)                     _namedict, _dictlist = _makenamedict(toplevel)
497: (8)                 numfound = 0
498: (8)                 objlist = []
499: (8)                 for namestr in _dictlist:
500: (12)                     try:
501: (16)                         obj = _namedict[namestr][object]
502: (16)                         if id(obj) in objlist:
503: (20)                             print("\n      "
504: (26)                                 "*** Repeat reference found in %s *** " % namestr,
505: (26)                                 file=output
506: (26)                                 )
507: (16)                         else:
508: (20)                             objlist.append(id(obj))
509: (20)                             print("      *** Found in %s ***" % namestr, file=output)
510: (20)                             info(obj)
511: (20)                             print("-"*maxwidth, file=output)
512: (16)                             numfound += 1
513: (12)             except KeyError:
514: (16)                 pass
515: (8)             if numfound == 0:
516: (12)                 print("Help for %s not found." % object, file=output)
517: (8)             else:
518: (12)                 print("\n      "
519: (18)                     "*** Total of %d references found. ***" % numfound,
520: (18)                     file=output
521: (18)                     )
522: (4)             elif inspect.isfunction(object) or inspect.ismethod(object):
523: (8)                 name = object.__name__
524: (8)                 try:
525: (12)                     arguments = str(inspect.signature(object))
526: (8)                 except Exception:
527: (12)                     arguments = "()"
528: (8)                 if len(name+arguments) > maxwidth:
529: (12)                     argstr = _split_line(name, arguments, maxwidth)
530: (8)                 else:
531: (12)                     argstr = name + arguments
532: (8)                 print(" " + argstr + "\n", file=output)
533: (8)                 print(inspect.getdoc(object), file=output)
534: (4)             elif inspect.isclass(object):
535: (8)                 name = object.__name__
536: (8)                 try:
537: (12)                     arguments = str(inspect.signature(object))
538: (8)                 except Exception:
539: (12)                     arguments = "()"
540: (8)                 if len(name+arguments) > maxwidth:
541: (12)                     argstr = _split_line(name, arguments, maxwidth)

```

```

542: (8)
543: (12)
544: (8)
545: (8)
546: (8)
547: (12)
548: (16)
549: (8)
550: (12)
551: (8)
552: (8)
553: (8)
554: (12)
555: (12)
556: (16)
557: (16)
558: (20)
559: (28)
560: (28)
561: (16)
562: (4)
563: (8)
564: (0)
565: (0)
566: (4)
567: (4)
568: (4)
569: (4)
570: (4)
571: (4)
572: (4)
573: (4)
574: (8)
575: (8)
576: (4)
577: (8)
578: (8)
579: (8)
580: (4)
581: (4)
582: (4)
583: (4)
584: (4)
585: (4)
586: (4)
587: (4)
588: (8)
589: (8)
590: (12)
591: (8)
592: (12)
593: (4)
594: (4)
595: (4)
596: (4)
597: (4)
598: (4)
599: (8)
600: (8)
601: (4)
602: (8)
603: (0)
604: (0)
605: (0)
606: (0)
607: (12)
608: (4)
609: (4)
610: (4)

        else:
            argstr = name + arguments
            print(" " + argstr + "\n", file=output)
            doc1 = inspect.getdoc(object)
            if doc1 is None:
                if hasattr(object, '__init__'):
                    print(inspect.getdoc(object.__init__), file=output)
            else:
                print(inspect.getdoc(object), file=output)
            methods = pydoc.allmethods(object)
            public_methods = [meth for meth in methods if meth[0] != '_']
            if public_methods:
                print("\n\nMethods:\n", file=output)
                for meth in public_methods:
                    thisobj = getattr(object, meth, None)
                    if thisobj is not None:
                        methstr, other = pydoc.splitdoc(
                            inspect.getdoc(thisobj) or "None"
                        )
                        print(" %s -- %s" % (meth, methstr), file=output)
            elif hasattr(object, '__doc__'):
                print(inspect.getdoc(object), file=output)
@set_module('numpy')
def source(object, output=sys.stdout):
    """
    Print or write to a file the source code for a NumPy object.
    The source code is only returned for objects written in Python. Many
    functions and classes are defined in C and will therefore not return
    useful information.
    Parameters
    -----
    object : numpy object
        Input object. This can be any object (function, class, module,
        ...).
    output : file object, optional
        If `output` not supplied then source code is printed to screen
        (sys.stdout). File object must be created with either write 'w' or
        append 'a' modes.
    See Also
    -----
    lookfor, info
    Examples
    -----
    >>> np.source(np.interp)                                #doctest: +SKIP
    In file: /usr/lib/python2.6/dist-packages/numpy/lib/function_base.py
    def interp(x, xp, fp, left=None, right=None):
        """.... (full docstring printed)"""
        if isinstance(x, (float, int, number)):
            return compiled_interp([x], xp, fp, left, right).item()
        else:
            return compiled_interp(x, xp, fp, left, right)
    The source code is only returned for objects written in Python.
    >>> np.source(np.array)                                #doctest: +SKIP
    Not available for this object.
    """
    import inspect
    try:
        print("In file: %s\n" % inspect.getsourcefile(object), file=output)
        print(inspect.getsource(object), file=output)
    except Exception:
        print("Not available for this object.", file=output)
    _lookfor_caches = {}
    _function_signature_re = re.compile(r"[a-zA-Z0-9_]+\(.*[,=].*\)", re.I)
@set_module('numpy')
def lookfor(what, module=None, import_modules=True, regenerate=False,
           output=None):
    """
    Do a keyword search on docstrings.
    A list of objects that matched the search is displayed,

```

```

611: (4)                                sorted by relevance. All given keywords need to be found in the
612: (4)                                docstring for it to be returned as a result, but the order does
613: (4)                                not matter.
614: (4)                                Parameters
615: (4)                                -----
616: (4)                                what : str
617: (8)                                String containing words to look for.
618: (4)                                module : str or list, optional
619: (8)                                Name of module(s) whose docstrings to go through.
620: (4)                                import_modules : bool, optional
621: (8)                                Whether to import sub-modules in packages. Default is True.
622: (4)                                regenerate : bool, optional
623: (8)                                Whether to re-generate the docstring cache. Default is False.
624: (4)                                output : file-like, optional
625: (8)                                File-like object to write the output to. If omitted, use a pager.
626: (4)                                See Also
627: (4)                                -----
628: (4)                                source, info
629: (4)                                Notes
630: (4)                                -----
631: (4)                                Relevance is determined only roughly, by checking if the keywords occur
632: (4)                                in the function name, at the start of a docstring, etc.
633: (4)                                Examples
634: (4)                                -----
635: (4)                                >>> np.lookfor('binary representation') # doctest: +SKIP
636: (4)                                Search results for 'binary representation'
637: (4)                                -----
638: (4)                                numpy.binary_repr
639: (8)                                Return the binary representation of the input number as a string.
640: (4)                                numpy.core.setup_common.long_double_representation
641: (8)                                Given a binary dump as given by GNU od -b, look for long double
642: (4)                                numpy.base_repr
643: (8)                                Return a string representation of a number in the given base system.
644: (4)
645: (4)
646: (4)                                import pydoc
647: (4)                                cache = _lookfor_generate_cache(module, import_modules, regenerate)
648: (4)                                found = []
649: (4)                                whats = str(what).lower().split()
650: (4)                                if not whats:
651: (8)                                    return
652: (4)                                for name, (docstring, kind, index) in cache.items():
653: (8)                                    if kind in ('module', 'object'):
654: (12)                                        continue
655: (8)                                    doc = docstring.lower()
656: (8)                                    if all(w in doc for w in whats):
657: (12)                                        found.append(name)
658: (4)                                kind_relevance = {'func': 1000, 'class': 1000,
659: (22)                                         'module': -1000, 'object': -1000}
660: (4)                                def relevance(name, docstr, kind, index):
661: (8)                                    r = 0
662: (8)                                    first_doc = "\n".join(docstr.lower().strip().split("\n")[:3])
663: (8)                                    r += sum([200 for w in whats if w in first_doc])
664: (8)                                    r += sum([30 for w in whats if w in name])
665: (8)                                    r += -len(name) * 5
666: (8)                                    r += kind_relevance.get(kind, -1000)
667: (8)                                    r += -name.count('.') * 10
668: (8)                                    r += max(-index / 100, -100)
669: (8)                                    return r
670: (4)                                def relevance_value(a):
671: (8)                                    return relevance(a, *cache[a])
672: (4)                                found.sort(key=relevance_value)
673: (4)                                s = "Search results for '%s'" % (' '.join(whats))
674: (4)                                help_text = [s, "-"*len(s)]
675: (4)                                for name in found[::-1]:
676: (8)                                    doc, kind, ix = cache[name]
677: (8)                                    doclines = [line.strip() for line in doc.strip().split("\n")
678: (20)                                         if line.strip()]
679: (8)                                try:

```

```

680: (12)                     first_doc = doclines[0].strip()
681: (12)                     if _function_signature_re.search(first_doc):
682: (16)                         first_doc = doclines[1].strip()
683: (8)                     except IndexError:
684: (12)                         first_doc = ""
685: (8)                     help_text.append("%s\n    %s" % (name, first_doc))
686: (4)                 if not found:
687: (8)                     help_text.append("Nothing found.")
688: (4)                 if output is not None:
689: (8)                     output.write("\n".join(help_text))
690: (4)                 elif len(help_text) > 10:
691: (8)                     pager = pydoc.getpager()
692: (8)                     pager("\n".join(help_text))
693: (4)                 else:
694: (8)                     print("\n".join(help_text))
695: (0)             def _lookfor_generate_cache(module, import_modules, regenerate):
696: (4)                 """
697: (4)                     Generate docstring cache for given module.
698: (4)                     Parameters
699: (4)                     -----
700: (4)                     module : str, None, module
701: (8)                         Module for which to generate docstring cache
702: (4)                     import_modules : bool
703: (8)                         Whether to import sub-modules in packages.
704: (4)                     regenerate : bool
705: (8)                         Re-generate the docstring cache
706: (4)                     Returns
707: (4)                     -----
708: (4)                     cache : dict {obj_full_name: (docstring, kind, index), ...}
709: (8)                         Docstring cache for the module, either cached one (regenerate=False)
710: (8)                         or newly generated.
711: (4)                     """
712: (4)                     import inspect
713: (4)                     from io import StringIO
714: (4)                     if module is None:
715: (8)                         module = "numpy"
716: (4)                     if isinstance(module, str):
717: (8)                         try:
718: (12)                             __import__(module)
719: (8)                         except ImportError:
720: (12)                             return {}
721: (8)                         module = sys.modules[module]
722: (4)                     elif isinstance(module, list) or isinstance(module, tuple):
723: (8)                         cache = {}
724: (8)                         for mod in module:
725: (12)                             cache.update(_lookfor_generate_cache(mod, import_modules,
726: (49)                                         regenerate))
727: (8)                         return cache
728: (4)                     if id(module) in _lookfor_caches and not regenerate:
729: (8)                         return _lookfor_caches[id(module)]
730: (4)                     cache = {}
731: (4)                     _lookfor_caches[id(module)] = cache
732: (4)                     seen = {}
733: (4)                     index = 0
734: (4)                     stack = [(module.__name__, module)]
735: (4)                     while stack:
736: (8)                         name, item = stack.pop(0)
737: (8)                         if id(item) in seen:
738: (12)                             continue
739: (8)                         seen[id(item)] = True
740: (8)                         index += 1
741: (8)                         kind = "object"
742: (8)                         if inspect.ismodule(item):
743: (12)                             kind = "module"
744: (12)                             try:
745: (16)                                 __all__ = item.__all__
746: (12)                             except AttributeError:
747: (16)                                 __all__ = None
748: (12)                             if import_modules and hasattr(item, '__path__'):

```

```

749: (16)             for pth in item.__path__:
750: (20)                 for mod_path in os.listdir(pth):
751: (24)                     this_py = os.path.join(pth, mod_path)
752: (24)                     init_py = os.path.join(pth, mod_path, '__init__.py')
753: (24)                     if (os.path.isfile(this_py) and
754: (32)                         mod_path.endswith('.py')):
755: (28)                         to_import = mod_path[:-3]
756: (24)                     elif os.path.isfile(init_py):
757: (28)                         to_import = mod_path
758: (24)                     else:
759: (28)                         continue
760: (24)                     if to_import == '__init__':
761: (28)                         continue
762: (24)                 try:
763: (28)                     old_stdout = sys.stdout
764: (28)                     old_stderr = sys.stderr
765: (28)                     try:
766: (32)                         sys.stdout = StringIO()
767: (32)                         sys.stderr = StringIO()
768: (32)                         __import__( "%s.%s" % (name, to_import))
769: (28)                         finally:
770: (32)                             sys.stdout = old_stdout
771: (32)                             sys.stderr = old_stderr
772: (24)                     except KeyboardInterrupt:
773: (28)                         raise
774: (24)                     except BaseException:
775: (28)                         continue
776: (12)                 for n, v in _getmembers(item):
777: (16)                     try:
778: (20)                         item_name = getattr(v, '__name__', "%s.%s" % (name, n))
779: (20)                         mod_name = getattr(v, '__module__', None)
780: (16)                     except NameError:
781: (20)                         item_name = "%s.%s" % (name, n)
782: (20)                         mod_name = None
783: (16)                     if '.' not in item_name and mod_name:
784: (20)                         item_name = "%s.%s" % (mod_name, item_name)
785: (16)                     if not item_name.startswith(name + '.'):
786: (20)                         if isinstance(v, ufunc):
787: (24)                             pass
788: (20)                         else:
789: (24)                             continue
790: (16)                     elif not (inspect.ismodule(v) or _all is None or n in _all):
791: (20)                         continue
792: (16)                     stack.append(( "%s.%s" % (name, n), v))
793: (8)                     elif inspect.isclass(item):
794: (12)                         kind = "class"
795: (12)                         for n, v in _getmembers(item):
796: (16)                             stack.append(( "%s.%s" % (name, n), v))
797: (8)                     elif hasattr(item, '__call__'):
798: (12)                         kind = "func"
799: (8)                     try:
800: (12)                         doc = inspect.getdoc(item)
801: (8)                     except NameError:
802: (12)                         doc = None
803: (8)                         if doc is not None:
804: (12)                             cache[name] = (doc, kind, index)
805: (4)                     return cache
806: (0)                 def _getmembers(item):
807: (4)                     import inspect
808: (4)                     try:
809: (8)                         members = inspect.getmembers(item)
810: (4)                     except Exception:
811: (8)                         members = [(x, getattr(item, x)) for x in dir(item)
812: (19)                             if hasattr(item, x)]
813: (4)                     return members
814: (0)                 def safe_eval(source):
815: (4)                     """
816: (4)                         Protected string evaluation.
817: (4)                         Evaluate a string containing a Python literal expression without

```

```

818: (4)           allowing the execution of arbitrary non-literal code.
819: (4)
820: (8)
821: (8)
822: (8)
823: (4)
824: (4)
825: (4)
826: (8)
827: (4)
828: (4)
829: (4)
830: (7)
831: (4)
832: (4)
833: (4)
834: (8)
835: (8)
836: (4)
837: (4)
838: (4)
839: (4)
840: (4)
841: (4)
842: (4)
843: (4)
844: (4)
845: (4)
846: (6)
847: (4)
848: (4)
849: (4)
850: (6)
851: (4)
852: (4)
853: (4)
854: (4)
855: (0)
856: (4)
857: (4)
end
858: (4)
859: (4)
860: (4)
861: (4)
862: (8)
863: (4)
864: (8)
865: (4)
866: (8)
867: (4)
868: (4)
869: (4)
870: (8)
871: (8)
872: (8)
873: (4)
874: (4)
875: (8)
876: (4)
877: (4)
878: (4)
879: (8)
880: (4)
881: (8)
882: (4)
883: (8)
884: (4)
885: (4)

          .. warning::
              This function is identical to :py:meth:`ast.literal_eval` and
              has the same security implications. It may not always be safe
              to evaluate large input strings.

          Parameters
          -----
          source : str
              The string to evaluate.
          Returns
          -----
          obj : object
              The result of evaluating `source`.
          Raises
          -----
          SyntaxError
              If the code has invalid Python syntax, or if it contains
              non-literal code.
          Examples
          -----
          >>> np.safe_eval('1')
          1
          >>> np.safe_eval('[1, 2, 3]')
          [1, 2, 3]
          >>> np.safe_eval('{"foo": ("bar", 10.0)}')
          {'foo': ('bar', 10.0)}
          >>> np.safe_eval('import os')
          Traceback (most recent call last):
          ...
          SyntaxError: invalid syntax
          >>> np.safe_eval('open("/home/user/.ssh/id_dsa").read()')
          Traceback (most recent call last):
          ...
          ValueError: malformed node or string: <_ast.Call object at 0x...>
          """
          import ast
          return ast.literal_eval(source)
      def _median_nancheck(data, result, axis):
          """
          Utility function to check median result from data for NaN values at the
          and return NaN in that case. Input result can also be a MaskedArray.

          Parameters
          -----
          data : array
              Sorted input data to median function
          result : Array or MaskedArray
              Result of median function.
          axis : int
              Axis along which the median was computed.
          Returns
          -----
          result : scalar or ndarray
              Median or NaN in axes which contained NaN in the input. If the input
              was an array, NaN will be inserted in-place. If a scalar, either the
              input itself or a scalar NaN.
          """
          if data.size == 0:
              return result
          potential_nans = data.take(-1, axis=axis)
          n = np.isnan(potential_nans)
          if np.ma.isMaskedArray(n):
              n = n.filled(False)
          if not n.any():
              return result
          if isinstance(result, np.generic):
              return potential_nans
          np.copyto(result, potential_nans, where=n)
          return result

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

886: (0)
887: (4)
888: (4)
889: (4)
890: (8)
891: (10)
892: (8)
893: (10)
894: (8)
895: (4)
896: (4)
897: (8)
898: (4)
899: (4)
900: (8)
901: (4)
902: (4)
903: (8)
904: (12)
905: (8)
906: (12)
907: (4)
908: (0)
909: (4)
910: (4)
911: (4)
912: (4)
913: (4)
914: (4)
915: (8)
916: (8)
917: (4)
918: (8)
919: (8)
920: (8)
921: (8)
922: (4)
923: (4)
924: (8)
925: (8)
926: (8)
927: (8)
928: (8)
929: (8)
930: (12)
931: (12)
932: (16)
933: (12)
934: (12)
935: (12)
936: (12)
937: (8)
938: (12)
939: (8)
940: (12)
941: (12)
942: (8)
943: (4)
944: (8)
945: (8)
946: (8)
947: (12)
948: (8)
949: (4)
950: (8)
951: (12)
952: (8)

def _opt_info():
    """
        Returns a string contains the supported CPU features by the current build.
        The string format can be explained as follows:
            - dispatched features that are supported by the running machine
            end with `*`.
            - dispatched features that are "not" supported by the running machine
            end with `?`.
            - remained features are representing the baseline.
    """
    from numpy.core._multiarray_umath import (
        __cpu_features__, __cpu_baseline__, __cpu_dispatch__
    )
    if len(__cpu_baseline__) == 0 and len(__cpu_dispatch__) == 0:
        return ''
    enabled_features = ' '.join(__cpu_baseline__)
    for feature in __cpu_dispatch__:
        if __cpu_features__[feature]:
            enabled_features += f" {feature}*"
        else:
            enabled_features += f" {feature}?"
    return enabled_features

def drop_metadata(dtype, /):
    """
        Returns the dtype unchanged if it contained no metadata or a copy of the
        dtype if it (or any of its structure dtypes) contained metadata.
        This utility is used by `np.save` and `np.savez` to drop metadata before
        saving.
        .. note::
            Due to its limitation this function may move to a more appropriate
            home or change in the future and is considered semi-public API only.
        .. warning::
            This function does not preserve more strange things like record dtypes
            and user dtypes may simply return the wrong thing. If you need to be
            sure about the latter, check the result with:
            ``np.can_cast(new_dtype, dtype, casting="no")``.
    """
    if dtype.fields is not None:
        found_metadata = dtype.metadata is not None
        names = []
        formats = []
        offsets = []
        titles = []
        for name, field in dtype.fields.items():
            field_dt = drop_metadata(field[0])
            if field_dt is not field[0]:
                found_metadata = True
            names.append(name)
            formats.append(field_dt)
            offsets.append(field[1])
            titles.append(None if len(field) < 3 else field[2])
        if not found_metadata:
            return dtype
        structure = dict(
            names=names, formats=formats, offsets=offsets, titles=titles,
            itemsize=dtype.itemsize)
        return np.dtype(structure, align=dtype.isalignedstruct)
    elif dtype.subdtype is not None:
        subdtype, shape = dtype.subdtype
        new_subdtype = drop_metadata(subdtype)
        if dtype.metadata is None and new_subdtype is subdtype:
            return dtype
        return np.dtype((new_subdtype, shape))
    else:
        if dtype.metadata is None:
            return dtype
        return np.dtype(dtype.str)

```

## File 225 - \_datasource.py:

```

1: (0)         """A file interface for handling local and remote data files.
2: (0)         The goal of datasource is to abstract some of the file system operations
3: (0)         when dealing with data files so the researcher doesn't have to know all the
4: (0)         low-level details. Through datasource, a researcher can obtain and use a
5: (0)         file with one function call, regardless of location of the file.
6: (0)         DataSource is meant to augment standard python libraries, not replace them.
7: (0)         It should work seamlessly with standard file IO operations and the os
8: (0)         module.
9: (0)         DataSource files can originate locally or remotely:
10: (0)          - local files : '/home/guido/src/local/data.txt'
11: (0)          - URLs (http, ftp, ...) : 'http://www.scipy.org/not/real/data.txt'
12: (0)         DataSource files can also be compressed or uncompressed. Currently only
13: (0)         gzip, bz2 and xz are supported.
14: (0)         Example::
15: (4)             >>> # Create a DataSource, use os.curdir (default) for local storage.
16: (4)             >>> from numpy import DataSource
17: (4)             >>> ds = DataSource()
18: (4)
19: (4)             >>> # Open a remote file.
20: (4)             >>> # DataSource downloads the file, stores it locally in:
21: (4)                 './www.google.com/index.html'
22: (4)             >>> # opens the file and returns a file object.
23: (4)             >>> fp = ds.open('http://www.google.com/') # doctest: +SKIP
24: (4)
25: (4)             >>> # Use the file as you normally would
26: (4)             >>> fp.read() # doctest: +SKIP
27: (4)             >>> fp.close() # doctest: +SKIP
28: (0)
29: (0)         """
30: (0)         import os
31: (0)         import io
32: (0)         from ..utils import set_module
33: (0)         _open = open
34: (0)         def _check_mode(mode, encoding, newline):
35: (4)             """Check mode and that encoding and newline are compatible.
36: (4)             Parameters
37: (4)             -----
38: (4)             mode : str
39: (8)                 File open mode.
40: (4)             encoding : str
41: (8)                 File encoding.
42: (4)             newline : str
43: (8)                 Newline for text files.
44: (0)
45: (4)             if "t" in mode:
46: (8)                 if "b" in mode:
47: (12)                     raise ValueError("Invalid mode: %r" % (mode,))
48: (4)             else:
49: (8)                 if encoding is not None:
50: (12)                     raise ValueError("Argument 'encoding' not supported in binary
51: (12) mode")
52: (0)         class _FileOpeners:
53: (4)
54: (4)             """
55: (4)                 Container for different methods to open (un-)compressed files.
56: (4)                 `_FileOpeners` contains a dictionary that holds one method for each
57: (4)                 supported file format. Attribute lookup is implemented in such a way
58: (4)                 that an instance of `_FileOpeners` itself can be indexed with the keys
59: (4)                 of that dictionary. Currently uncompressed files as well as files
60: (4)                 compressed with ``gzip``, ``bz2`` or ``xz`` compression are supported.
61: (4)                 Notes
62: (4)                 -----
63: (4)                 `_file_openers`, an instance of `_FileOpeners`, is made available for
64: (4)                 use in the `datasource` module.

```

```

65: (4)      -----
66: (4)      >>> import gzip
67: (4)      >>> np.lib._datasource._file_openers.keys()
68: (4)      [None, '.bz2', '.gz', '.xz', '.lzma']
69: (4)      >>> np.lib._datasource._file_openers['.gz'] is gzip.open
70: (4)      True
71: (4)      """
72: (4)      def __init__(self):
73: (8)          self._loaded = False
74: (8)          self._file_openers = {None: io.open}
75: (4)      def _load(self):
76: (8)          if self._loaded:
77: (12)              return
78: (8)          try:
79: (12)              import bz2
80: (12)              self._file_openers[".bz2"] = bz2.open
81: (8)          except ImportError:
82: (12)              pass
83: (8)          try:
84: (12)              import gzip
85: (12)              self._file_openers[".gz"] = gzip.open
86: (8)          except ImportError:
87: (12)              pass
88: (8)          try:
89: (12)              import lzma
90: (12)              self._file_openers[".xz"] = lzma.open
91: (12)              self._file_openers[".lzma"] = lzma.open
92: (8)          except (ImportError, AttributeError):
93: (12)              pass
94: (8)          self._loaded = True
95: (4)      def keys(self):
96: (8)      """
97: (8)          Return the keys of currently supported file openers.
98: (8)          Parameters
99: (8)          -----
100: (8)          None
101: (8)          Returns
102: (8)          -----
103: (8)          keys : list
104: (12)              The keys are None for uncompressed files and the file extension
105: (12)              strings (i.e. ``'.gz'``, ``'.xz'``) for supported compression
106: (12)              methods.
107: (8)          """
108: (8)          self._load()
109: (8)          return list(self._file_openers.keys())
110: (4)      def __getitem__(self, key):
111: (8)          self._load()
112: (8)          return self._file_openers[key]
113: (0)      _file_openers = _FileOpeners()
114: (0)      def open(path, mode='r', destpath=os.curdir, encoding=None, newline=None):
115: (4)          """
116: (4)              Open `path` with `mode` and return the file object.
117: (4)              If `path` is an URL, it will be downloaded, stored in the
118: (4)              `DataSource` `destpath` directory and opened from there.
119: (4)              Parameters
120: (4)              -----
121: (4)              path : str
122: (8)                  Local file path or URL to open.
123: (4)              mode : str, optional
124: (8)                  Mode to open `path`. Mode 'r' for reading, 'w' for writing, 'a' to
125: (8)                  append. Available modes depend on the type of object specified by
126: (8)                  path. Default is 'r'.
127: (4)              destpath : str, optional
128: (8)                  Path to the directory where the source file gets downloaded to for
129: (8)                  use. If `destpath` is None, a temporary directory will be created.
130: (8)                  The default path is the current directory.
131: (4)              encoding : {None, str}, optional
132: (8)                  Open text file with given encoding. The default encoding will be
133: (8)                  what `io.open` uses.

```

```

134: (4)             newline : {None, str}, optional
135: (8)             Newline to use when reading text file.
136: (4)             Returns
137: (4)             -----
138: (4)             out : file object
139: (8)             The opened file.
140: (4)             Notes
141: (4)             -----
142: (4)             This is a convenience function that instantiates a `DataSource` and
143: (4)             returns the file object from ``DataSource.open(path)``.
144: (4)             """
145: (4)             ds = DataSource(destpath)
146: (4)             return ds.open(path, mode, encoding=encoding, newline=newline)
147: (0)             @set_module('numpy')
148: (0)             class DataSource:
149: (4)                 """
150: (4)                 DataSource(destpath='.')
151: (4)                 A generic data source file (file, http, ftp, ...).
152: (4)                 DataSources can be local files or remote files/URLs. The files may
153: (4)                 also be compressed or uncompressed. DataSource hides some of the
154: (4)                 low-level details of downloading the file, allowing you to simply pass
155: (4)                 in a valid file path (or URL) and obtain a file object.
156: (4)                 Parameters
157: (4)                 -----
158: (4)                 destpath : str or None, optional
159: (8)                 Path to the directory where the source file gets downloaded to for
160: (8)                 use. If `destpath` is None, a temporary directory will be created.
161: (8)                 The default path is the current directory.
162: (4)                 Notes
163: (4)                 -----
164: (4)                 URLs require a scheme string (``http://``) to be used, without it they
165: (4)                 will fail::
166: (8)                 >>> repos = np.DataSource()
167: (8)                 >>> repos.exists('www.google.com/index.html')
168: (8)                 False
169: (8)                 >>> repos.exists('http://www.google.com/index.html')
170: (8)                 True
171: (4)                 Temporary directories are deleted when the DataSource is deleted.
172: (4)                 Examples
173: (4)                 -----
174: (4)                 :::
175: (8)                 >>> ds = np.DataSource('/home/guido')
176: (8)                 >>> urlname = 'http://www.google.com/'
177: (8)                 >>> gfile = ds.open('http://www.google.com/')
178: (8)                 >>> ds.abspath(urlname)
179: (8)                 '/home/guido/www.google.com/index.html'
180: (8)                 >>> ds = np.DataSource(None) # use with temporary file
181: (8)                 >>> ds.open('/home/guido/foobar.txt')
182: (8)                 <open file '/home/guido.foobar.txt', mode 'r' at 0x91d4430>
183: (8)                 >>> ds.abspath('/home/guido/foobar.txt')
184: (8)                 '/tmp/.../home/guido/foobar.txt'
185: (4)                 """
186: (4)             def __init__(self, destpath=os.curdir):
187: (8)                 """Create a DataSource with a local path at destpath."""
188: (8)             if destpath:
189: (12)                 self._destpath = os.path.abspath(destpath)
190: (12)                 self._istmpdest = False
191: (8)             else:
192: (12)                 import tempfile # deferring import to improve startup time
193: (12)                 self._destpath = tempfile.mkdtemp()
194: (12)                 self._istmpdest = True
195: (4)             def __del__(self):
196: (8)                 if hasattr(self, '_istmpdest') and self._istmpdest:
197: (12)                     import shutil
198: (12)                     shutil.rmtree(self._destpath)
199: (4)             def _iszip(self, filename):
200: (8)                 """Test if the filename is a zip file by looking at the file
extension.
201: (8)                 """

```

```

202: (8)                     fname, ext = os.path.splitext(filename)
203: (8)                     return ext in _file_openers.keys()
204: (4) def __iswritemode(self, mode):
205: (8)     """Test if the given mode will open a file for writing."""
206: (8)     _writemodes = ("w", "+")
207: (8)     for c in mode:
208: (12)         if c in _writemodes:
209: (16)             return True
210: (8)     return False
211: (4) def __splitzipext(self, filename):
212: (8)     """Split zip extension from filename and return filename.
213: (8)     Returns
214: (8)     -----
215: (8)     base, zip_ext : {tuple}
216: (8)     """
217: (8)     if self.__iszip(filename):
218: (12)         return os.path.splitext(filename)
219: (8)     else:
220: (12)         return filename, None
221: (4) def __possible_names(self, filename):
222: (8)     """Return a tuple containing compressed filename variations."""
223: (8)     names = [filename]
224: (8)     if not self.__iszip(filename):
225: (12)         for zipext in _file_openers.keys():
226: (16)             if zipext:
227: (20)                 names.append(filename+zipext)
228: (8)     return names
229: (4) def __isurl(self, path):
230: (8)     """Test if path is a net location. Tests the scheme and netloc."""
231: (8)     from urllib.parse import urlparse
232: (8)     scheme, netloc, upath, uparams, uquery, ufrag = urlparse(path)
233: (8)     return bool(scheme and netloc)
234: (4) def __cache(self, path):
235: (8)     """Cache the file specified by path.
236: (8)     Creates a copy of the file in the datasource cache.
237: (8)     """
238: (8)     import shutil
239: (8)     from urllib.request import urlopen
240: (8)     upath = self.abspath(path)
241: (8)     if not os.path.exists(os.path.dirname(upath)):
242: (12)         os.makedirs(os.path.dirname(upath))
243: (8)     if self.__isurl(path):
244: (12)         with urlopen(path) as openedurl:
245: (16)             with __open(upath, 'wb') as f:
246: (20)                 shutil.copyfileobj(openedurl, f)
247: (8)     else:
248: (12)         shutil.copyfile(path, upath)
249: (8)     return upath
250: (4) def __findfile(self, path):
251: (8)     """Searches for ``path`` and returns full path if found.
252: (8)     If path is an URL, __findfile will cache a local copy and return the
253: (8)     path to the cached file. If path is a local file, __findfile will
254: (8)     return a path to that local file.
255: (8)     The search will include possible compressed versions of the file
256: (8)     and return the first occurrence found.
257: (8)     """
258: (8)     if not self.__isurl(path):
259: (12)         filelist = self.__possible_names(path)
260: (12)         filelist += self.__possible_names(self.abspath(path))
261: (8)     else:
262: (12)         filelist = self.__possible_names(self.abspath(path))
263: (12)         filelist = filelist + self.__possible_names(path)
264: (8)     for name in filelist:
265: (12)         if self.exists(name):
266: (16)             if self.__isurl(name):
267: (20)                 name = self.__cache(name)
268: (16)                 return name
269: (8)             return None
270: (4) def abspath(self, path):

```

```
271: (8)                """
272: (8)        Return absolute path of file in the DataSource directory.
273: (8)        If `path` is an URL, then `abspath` will return either the location
274: (8)        the file exists locally or the location it would exist when opened
275: (8)        using the `open` method.
276: (8)    Parameters
277: (8)    -----
278: (8)    path : str
279: (12)        Can be a local file or a remote URL.
280: (8)    Returns
281: (8)    -----
282: (8)    out : str
283: (12)        Complete path, including the `DataSource` destination directory.
284: (8)    Notes
285: (8)    -----
286: (8)        The functionality is based on `os.path.abspath`.
287: (8)    """
288: (8)    from urllib.parse import urlparse
289: (8)    splitpath = path.split(self._destpath, 2)
290: (8)    if len(splitpath) > 1:
291: (12)        path = splitpath[1]
292: (8)    scheme, netloc, upath, uparams, uquery, ufrag = urlparse(path)
293: (8)    netloc = self._sanitize_relative_path(netloc)
294: (8)    upath = self._sanitize_relative_path(upath)
295: (8)    return os.path.join(self._destpath, netloc, upath)
296: (4)    def _sanitize_relative_path(self, path):
297: (8)        """Return a sanitised relative path for which
298: (8)        os.path.abspath(os.path.join(base, path)).startswith(base)
299: (8)        """
300: (8)        last = None
301: (8)        path = os.path.normpath(path)
302: (8)        while path != last:
303: (12)            last = path
304: (12)            path = path.lstrip(os.sep).lstrip('/')
305: (12)            path = path.lstrip(os.pardir).lstrip('..')
306: (12)            drive, path = os.path.splitdrive(path) # for Windows
307: (8)        return path
308: (4)    def exists(self, path):
309: (8)        """
310: (8)        Test if path exists.
311: (8)        Test if `path` exists as (and in this order):
312: (8)        - a local file.
313: (8)        - a remote URL that has been downloaded and stored locally in the
314: (10)        `DataSource` directory.
315: (8)        - a remote URL that has not been downloaded, but is valid and
316: (10)        accessible.
317: (8)    Parameters
318: (8)    -----
319: (8)    path : str
320: (12)        Can be a local file or a remote URL.
321: (8)    Returns
322: (8)    -----
323: (8)    out : bool
324: (12)        True if `path` exists.
325: (8)    Notes
326: (8)    -----
327: (8)        When `path` is an URL, `exists` will return True if it's either
328: (8)        stored locally in the `DataSource` directory, or is a valid remote
329: (8)        URL. `DataSource` does not discriminate between the two, the file
330: (8)        is accessible if it exists in either location.
331: (8)    """
332: (8)    if os.path.exists(path):
333: (12)        return True
334: (8)    from urllib.request import urlopen
335: (8)    from urllib.error import URLError
336: (8)    upath = self.abspath(path)
337: (8)    if os.path.exists(upath):
338: (12)        return True
339: (8)    if self._isurl(path):
```

```

340: (12)
341: (16)
342: (16)
343: (16)
344: (16)
345: (12)
346: (16)
347: (8)
348: (4)
349: (8)
350: (8)
351: (8)
352: (8)
353: (8)
354: (8)
355: (8)
356: (12)
357: (8)
358: (12)
359: (12)
360: (12)
361: (8)
362: (12)
363: (12)
364: (8)
365: (12)
366: (8)
367: (8)
368: (8)
369: (12)
370: (8)
371: (8)
372: (12)
373: (8)
374: (8)
375: (12)
376: (12)
377: (16)
378: (12)
379: (38)
380: (8)
381: (12)
382: (0)
383: (4)
384: (4)
385: (4)
386: (4)
387: (4)
388: (4)
389: (4)
390: (4)
391: (4)
392: (4)
393: (4)
394: (4)
395: (8)
396: (8)
397: (4)
398: (8)
399: (8)
400: (8)
401: (4)
402: (4)
403: (4)
404: (4)
405: (8)
406: (8)
407: (8)
408: (8)

        try:
            netfile = urlopen(path)
            netfile.close()
            del(netfile)
            return True
        except URLError:
            return False
    return False
def open(self, path, mode='r', encoding=None, newline=None):
    """
        Open and return file-like object.
        If `path` is an URL, it will be downloaded, stored in the
        `DataSource` directory and opened from there.
    Parameters
    -----
    path : str
        Local file path or URL to open.
    mode : {'r', 'w', 'a'}, optional
        Mode to open `path`. Mode 'r' for reading, 'w' for writing,
        'a' to append. Available modes depend on the type of object
        specified by `path`. Default is 'r'.
    encoding : {None, str}, optional
        Open text file with given encoding. The default encoding will be
        what `io.open` uses.
    newline : {None, str}, optional
        Newline to use when reading text file.
    Returns
    -----
    out : file object
        File object.
    """
    if self._isurl(path) and self._iswritemode(mode):
        raise ValueError("URLs are not writeable")
    found = self._findfile(path)
    if found:
        fname, ext = self._splitzipext(found)
        if ext == 'bz2':
            mode.replace("+", "")
        return _file_openers[ext](found, mode=mode,
                                 encoding=encoding, newline=newline)
    else:
        raise FileNotFoundError(f"{path} not found.")
class Repository (DataSource):
    """
        Repository(baseurl, destpath='.')
        A data repository where multiple DataSource's share a base
        URL/directory.
        `Repository` extends `DataSource` by prepending a base URL (or
        directory) to all the files it handles. Use `Repository` when you will
        be working with multiple files from one base URL. Initialize
        `Repository` with the base URL, then refer to each file by its filename
        only.
    Parameters
    -----
    baseurl : str
        Path to the local directory or remote location that contains the
        data files.
    destpath : str or None, optional
        Path to the directory where the source file gets downloaded to for
        use. If `destpath` is None, a temporary directory will be created.
        The default path is the current directory.
    Examples
    -----
    To analyze all files in the repository, do something like this
    (note: this is not self-contained code):::
        >>> repos = np.lib._datasource.Repository('/home/user/data/dir/')
        >>> for filename in filelist:
            ...     fp = repos.open(filename)
            ...     fp.analyze()

```

```

409: (8)           ...     fp.close()
410: (4)           Similarly you could use a URL for a repository:
411: (8)           >>> repos = np.lib._datasource.Repository('http://www.xyz.edu/data')
412: (4)
413: (4)           def __init__(self, baseurl, destpath=os.curdir):
414: (8)               """Create a Repository with a shared url or directory of baseurl."""
415: (8)               DataSource.__init__(self, destpath=destpath)
416: (8)               self._baseurl = baseurl
417: (4)           def __del__(self):
418: (8)               DataSource.__del__(self)
419: (4)           def _fullpath(self, path):
420: (8)               """Return complete path for path. Prepends baseurl if necessary."""
421: (8)               splitpath = path.split(self._baseurl, 2)
422: (8)               if len(splitpath) == 1:
423: (12)                   result = os.path.join(self._baseurl, path)
424: (8)               else:
425: (12)                   result = path      # path contains baseurl already
426: (8)               return result
427: (4)           def _findfile(self, path):
428: (8)               """Extend DataSource method to prepend baseurl to ``path``."""
429: (8)               return DataSource._findfile(self, self._fullpath(path))
430: (4)           def abspath(self, path):
431: (8)
432: (8)               Return absolute path of file in the Repository directory.
433: (8)               If `path` is an URL, then `abspath` will return either the location
434: (8)               the file exists locally or the location it would exist when opened
435: (8)               using the `open` method.
436: (8)               Parameters
437: (8)               -----
438: (8)               path : str
439: (12)                   Can be a local file or a remote URL. This may, but does not
440: (12)                   have to, include the `baseurl` with which the `Repository` was
441: (12)                   initialized.
442: (8)               Returns
443: (8)               -----
444: (8)               out : str
445: (12)                   Complete path, including the `DataSource` destination directory.
446: (8)
447: (8)               return DataSource.abspath(self, self._fullpath(path))
448: (4)           def exists(self, path):
449: (8)
450: (8)               Test if path exists prepending Repository base URL to path.
451: (8)               Test if `path` exists as (and in this order):
452: (8)                   - a local file.
453: (8)                   - a remote URL that has been downloaded and stored locally in the
454: (10)                       `DataSource` directory.
455: (8)                   - a remote URL that has not been downloaded, but is valid and
456: (10)                       accessible.
457: (8)               Parameters
458: (8)               -----
459: (8)               path : str
460: (12)                   Can be a local file or a remote URL. This may, but does not
461: (12)                   have to, include the `baseurl` with which the `Repository` was
462: (12)                   initialized.
463: (8)               Returns
464: (8)               -----
465: (8)               out : bool
466: (12)                   True if `path` exists.
467: (8)               Notes
468: (8)               -----
469: (8)               When `path` is an URL, `exists` will return True if it's either
470: (8)               stored locally in the `DataSource` directory, or is a valid remote
471: (8)               URL. `DataSource` does not discriminate between the two, the file
472: (8)               is accessible if it exists in either location.
473: (8)
474: (8)               return DataSource.exists(self, self._fullpath(path))
475: (4)           def open(self, path, mode='r', encoding=None, newline=None):
476: (8)
477: (8)               Open and return file-like object prepending Repository base URL.

```

```

478: (8)             If `path` is an URL, it will be downloaded, stored in the
479: (8)             DataSource directory and opened from there.
480: (8)             Parameters
481: (8)             -----
482: (8)             path : str
483: (12)            Local file path or URL to open. This may, but does not have to,
484: (12)            include the `baseurl` with which the `Repository` was
485: (12)            initialized.
486: (8)             mode : {'r', 'w', 'a'}, optional
487: (12)            Mode to open `path`. Mode 'r' for reading, 'w' for writing,
488: (12)            'a' to append. Available modes depend on the type of object
489: (12)            specified by `path`. Default is 'r'.
490: (8)             encoding : {None, str}, optional
491: (12)            Open text file with given encoding. The default encoding will be
492: (12)            what `io.open` uses.
493: (8)             newline : {None, str}, optional
494: (12)            Newline to use when reading text file.
495: (8)             Returns
496: (8)             -----
497: (8)             out : file object
498: (12)            File object.
499: (8)
500: (8)
501: (31)           return DataSource.open(self, self._fullpath(path), mode,
502: (4)                         encoding=encoding, newline=newline)
503: (8)
504: (8)             def listdir(self):
505: (8)               """
506: (8)               List files in the source Repository.
507: (8)             Returns
508: (8)             -----
509: (8)             files : list of str
510: (8)               List of file names (not containing a directory part).
511: (8)             Notes
512: (8)             -----
513: (8)             Does not currently work for remote repositories.
514: (12)           """
515: (18)           if self._isurl(self._baseurl):
516: (8)             raise NotImplementedError(
517: (12)               "Directory listing of URLs, not supported yet.")
518: (8)
519: (8)             return os.listdir(self._baseurl)

-----

```

## File 226 - \_iotoools.py:

```

1: (0)             """A collection of functions designed to help I/O with ascii files.
2: (0)             """
3: (0)             __docformat__ = "restructuredtext en"
4: (0)             import numpy as np
5: (0)             import numpy.core.numeric as nx
6: (0)             from numpy.compat import asbytes, asunicode
7: (0)             def _decode_line(line, encoding=None):
8: (4)               """Decode bytes from binary input streams.
9: (4)               Defaults to decoding from 'latin1'. That differs from the behavior of
10: (4)                 np.compat.asunicode that decodes from 'ascii'.
11: (4)               Parameters
12: (4)               -----
13: (4)               line : str or bytes
14: (9)                 Line to be decoded.
15: (4)               encoding : str
16: (9)                 Encoding used to decode `line`.
17: (4)               Returns
18: (4)               -----
19: (4)               decoded_line : str
20: (4)               """
21: (4)               if type(line) is bytes:
22: (8)                 if encoding is None:
23: (12)                   encoding = "latin1"
24: (8)                   line = line.decode(encoding)

```

```

25: (4)             return line
26: (0)         def _is_string_like(obj):
27: (4)             """
28: (4)                 Check whether obj behaves like a string.
29: (4)             """
30: (4)             try:
31: (8)                 obj + ''
32: (4)             except (TypeError, ValueError):
33: (8)                 return False
34: (4)             return True
35: (0)         def _is_bytes_like(obj):
36: (4)             """
37: (4)                 Check whether obj behaves like a bytes object.
38: (4)             """
39: (4)             try:
40: (8)                 obj + b''
41: (4)             except (TypeError, ValueError):
42: (8)                 return False
43: (4)             return True
44: (0)         def has_nested_fields(ndtype):
45: (4)             """
46: (4)                 Returns whether one or several fields of a dtype are nested.
47: (4)             Parameters
48: (4)             -----
49: (4)             ndtype : dtype
50: (8)                 Data-type of a structured array.
51: (4)             Raises
52: (4)             -----
53: (4)             AttributeError
54: (8)                 If `ndtype` does not have a `names` attribute.
55: (4)             Examples
56: (4)             -----
57: (4)             >>> dt = np.dtype([('name', 'S4'), ('x', float), ('y', float)])
58: (4)             >>> np.lib._iotools.has_nested_fields(dt)
59: (4)             False
60: (4)             """
61: (4)             for name in ndtype.names or []:
62: (8)                 if ndtype[name].names is not None:
63: (12)                     return True
64: (4)             return False
65: (0)         def flatten_dtype(ndtype, flatten_base=False):
66: (4)             """
67: (4)                 Unpack a structured data-type by collapsing nested fields and/or fields
68: (4)                 with a shape.
69: (4)                 Note that the field names are lost.
70: (4)             Parameters
71: (4)             -----
72: (4)             ndtype : dtype
73: (8)                 The datatype to collapse
74: (4)             flatten_base : bool, optional
75: (7)                 If True, transform a field with a shape into several fields. Default is
76: (7)                 False.
77: (4)             Examples
78: (4)             -----
79: (4)             >>> dt = np.dtype([('name', 'S4'), ('x', float), ('y', float),
80: (4)                         ...,
81: (4)                         ('block', int, (2, 3))])
82: (4)             >>> np.lib._iotoools.flatten_dtype(dt)
83: (4)             [dtype('S4'), dtype('float64'), dtype('float64'), dtype('int64')]
84: (4)             >>> np.lib._iotoools.flatten_dtype(dt, flatten_base=True)
85: (5)             [dtype('S4'),
86: (5)                 dtype('float64'),
87: (5)                 dtype('float64'),
88: (5)                 dtype('int64'),
89: (5)                 dtype('int64'),
90: (5)                 dtype('int64'),
91: (5)                 dtype('int64'),
92: (5)                 dtype('int64')]
93: (4)             """

```

```

94: (4)             names = ndtype.names
95: (4)             if names is None:
96: (8)                 if flatten_base:
97: (12)                     return [ndtype.base] * int(np.prod(ndtype.shape))
98: (8)                     return [ndtype.base]
99: (4)             else:
100: (8)                 types = []
101: (8)                 for field in names:
102: (12)                     info = ndtype.fields[field]
103: (12)                     flat_dt = flatten_dtype(info[0], flatten_base)
104: (12)                     types.extend(flat_dt)
105: (8)                 return types
106: (0)             class LineSplitter:
107: (4)                 """
108: (4)                     Object to split a string at a given delimiter or at given places.
109: (4)                     Parameters
110: (4)                     -----
111: (4)                     delimiter : str, int, or sequence of ints, optional
112: (8)                         If a string, character used to delimit consecutive fields.
113: (8)                         If an integer or a sequence of integers, width(s) of each field.
114: (4)                     comments : str, optional
115: (8)                         Character used to mark the beginning of a comment. Default is '#'.
116: (4)                     autostrip : bool, optional
117: (8)                         Whether to strip each individual field. Default is True.
118: (4)                     """
119: (4)             def autostrip(self, method):
120: (8)                 """
121: (8)                     Wrapper to strip each member of the output of `method` .
122: (8)                     Parameters
123: (8)                     -----
124: (8)                     method : function
125: (12)                         Function that takes a single argument and returns a sequence of
126: (12)                         strings.
127: (8)                     Returns
128: (8)                     -----
129: (8)                     wrapped : function
130: (12)                         The result of wrapping `method` . `wrapped` takes a single input
131: (12)                         argument and returns a list of strings that are stripped of
132: (12)                         white-space.
133: (8)                     """
134: (8)             return lambda input: [_.strip() for _ in method(input)]
135: (4)             def __init__(self, delimiter=None, comments='#', autostrip=True,
136: (17)                         encoding=None):
137: (8)                 delimiter = _decode_line(delimiter)
138: (8)                 comments = _decode_line(comments)
139: (8)                 self.comments = comments
140: (8)                 if (delimiter is None) or isinstance(delimiter, str):
141: (12)                     delimiter = delimiter or None
142: (12)                     _handyman = self._delimited_splitter
143: (8)                 elif hasattr(delimiter, '__iter__'):
144: (12)                     _handyman = self._variablewidth_splitter
145: (12)                     idx = np.cumsum([0] + list(delimiter))
146: (12)                     delimiter = [slice(i, j) for (i, j) in zip(idx[:-1], idx[1:])]

147: (8)                 elif int(delimiter):
148: (12)                     (_handyman, delimiter) = (
149: (20)                         self._fixedwidth_splitter, int(delimiter))
150: (8)                 else:
151: (12)                     (_handyman, delimiter) = (self._delimited_splitter, None)
152: (8)                 self.delimiter = delimiter
153: (8)                 if autostrip:
154: (12)                     self._handyman = self.autostrip(_handyman)
155: (8)                 else:
156: (12)                     self._handyman = _handyman
157: (8)                 self.encoding = encoding
158: (4)             def _delimited_splitter(self, line):
159: (8)                 """Chop off comments, strip, and split at delimiter. """
160: (8)                 if self.comments is not None:
161: (12)                     line = line.split(self.comments)[0]
162: (8)                     line = line.strip("\r\n")

```

```

163: (8)
164: (12)
165: (8)
166: (4)
167: (8)
168: (12)
169: (8)
170: (8)
171: (12)
172: (8)
173: (8)
174: (8)
175: (4)
176: (8)
177: (12)
178: (8)
179: (12)
180: (8)
181: (8)
182: (4)
183: (8)
184: (0)
185: (4)
186: (4)
187: (4)
188: (4)
189: (4)
190: (4)
191: (4)
192: (4)
193: (4)
194: (4)
195: (4)
196: (4)
197: (4)
198: (4)
199: (8)
200: (8)
201: (8)
202: (4)
203: (8)
204: (8)
205: (4)
206: (8)
207: (8)
208: (8)
209: (8)
210: (4)
211: (8)
212: (4)
213: (4)
214: (4)
215: (4)
216: (4)
217: (4)
218: (4)
219: (4)
220: (4)
221: (4)
222: (4)
223: (4)
224: (4)
225: (4)
226: (4)
227: (4)
228: (4)
229: (4)
230: (17)
231: (8)

        if not line:
            return []
        return line.split(self.delimiter)
    def _fixedwidth_splitter(self, line):
        if self.comments is not None:
            line = line.split(self.comments)[0]
        line = line.strip("\r\n")
        if not line:
            return []
        fixed = self.delimiter
        slices = [slice(i, i + fixed) for i in range(0, len(line), fixed)]
        return [line[s] for s in slices]
    def _variablewidth_splitter(self, line):
        if self.comments is not None:
            line = line.split(self.comments)[0]
        if not line:
            return []
        slices = self.delimiter
        return [line[s] for s in slices]
    def __call__(self, line):
        return self._handyman(_decode_line(line, self.encoding))

class NameValidator:
    """
        Object to validate a list of strings to use as field names.
        The strings are stripped of any non alphanumeric character, and spaces
        are replaced by '_'. During instantiation, the user can define a list
        of names to exclude, as well as a list of invalid characters. Names in
        the exclusion list are appended a '_' character.
        Once an instance has been created, it can be called with a list of
        names, and a list of valid names will be created. The `__call__`
        method accepts an optional keyword "default" that sets the default name
        in case of ambiguity. By default this is 'f', so that names will
        default to `f0`, `f1`, etc.
        Parameters
        -----
        excludelist : sequence, optional
            A list of names to exclude. This list is appended to the default
            list ['return', 'file', 'print']. Excluded names are appended an
            underscore: for example, `file` becomes `file_` if supplied.
        deletechars : str, optional
            A string combining invalid characters that must be deleted from the
            names.
        case_sensitive : {True, False, 'upper', 'lower'}, optional
            * If True, field names are case-sensitive.
            * If False or 'upper', field names are converted to upper case.
            * If 'lower', field names are converted to lower case.
            The default value is True.
        replace_space : '_', optional
            Character(s) used in replacement of white spaces.
        Notes
        -----
        Calling an instance of `NameValidator` is the same as calling its
        method `validate`.
        Examples
        -----
        >>> validator = np.lib._iotoools.NameValidator()
        >>> validator(['file', 'field2', 'with space', 'CaSe'])
        ('file_', 'field2', 'with_space', 'CaSe')
        >>> validator = np.lib._iotoools.NameValidator(excludelist=['excl'],
        ...                                              deletechars='q',
        ...                                              case_sensitive=False)
        >>> validator(['excl', 'field2', 'no_q', 'with space', 'CaSe'])
        ('EXCL', 'FIELD2', 'NO_Q', 'WITH_SPACE', 'CASE')
        """
        defaultexcludelist = ['return', 'file', 'print']
        defaultdeletechars = set(r"""\~!@#$%^&*()=-+~\|]{\';: /?.>,<""")
        def __init__(self, excludelist=None, deletechars=None,
                     case_sensitive=None, replace_space='_'):
            if excludelist is None:

```

```

232: (12)           excludelist = []
233: (8)            excludelist.extend(self.defaultexcludelist)
234: (8)            self.excludelist = excludelist
235: (8)            if deletechars is None:
236: (12)              delete = self.defaultdeletechars
237: (8)            else:
238: (12)              delete = set(deletechars)
239: (8)            delete.add('')
240: (8)            self.deletechars = delete
241: (8)            if (case_sensitive is None) or (case_sensitive is True):
242: (12)              self.case_converter = lambda x: x
243: (8)            elif (case_sensitive is False) or case_sensitive.startswith('u'):
244: (12)              self.case_converter = lambda x: x.upper()
245: (8)            elif case_sensitive.startswith('l'):
246: (12)              self.case_converter = lambda x: x.lower()
247: (8)            else:
248: (12)              msg = 'unrecognized case_sensitive value %s.' % case_sensitive
249: (12)              raise ValueError(msg)
250: (8)            self.replace_space = replace_space
251: (4)            def validate(self, names, defaultfmt="f%i", nbfields=None):
252: (8)                """
253: (8)                    Validate a list of strings as field names for a structured array.
254: (8)                    Parameters
255: (8)                    -----
256: (8)                    names : sequence of str
257: (12)                      Strings to be validated.
258: (8)                    defaultfmt : str, optional
259: (12)                      Default format string, used if validating a given string
260: (12)                      reduces its length to zero.
261: (8)                    nbfields : integer, optional
262: (12)                      Final number of validated names, used to expand or shrink the
263: (12)                      initial list of names.
264: (8)                    Returns
265: (8)                    -----
266: (8)                    validatednames : list of str
267: (12)                      The list of validated field names.
268: (8)                    Notes
269: (8)                    -----
270: (8)                    A `NameValidator` instance can be called directly, which is the
271: (8)                    same as calling `validate`. For examples, see `NameValidator`.
272: (8)                    """
273: (8)                    if (names is None):
274: (12)                      if (nbfields is None):
275: (16)                        return None
276: (12)                      names = []
277: (8)                    if isinstance(names, str):
278: (12)                      names = [names, ]
279: (8)                    if nbfields is not None:
280: (12)                      nbnames = len(names)
281: (12)                      if (nbnames < nbfields):
282: (16)                        names = list(names) + [''] * (nbfields - nbnames)
283: (12)                      elif (nbnames > nbfields):
284: (16)                        names = names[:nbfields]
285: (8)                    deletechars = self.deletechars
286: (8)                    excludelist = self.excludelist
287: (8)                    case_converter = self.case_converter
288: (8)                    replace_space = self.replace_space
289: (8)                    validatednames = []
290: (8)                    seen = dict()
291: (8)                    nbempty = 0
292: (8)                    for item in names:
293: (12)                      item = case_converter(item).strip()
294: (12)                      if replace_space:
295: (16)                        item = item.replace(' ', replace_space)
296: (12)                      item = ''.join([c for c in item if c not in deletechars])
297: (12)                      if item == '':
298: (16)                        item = defaultfmt % nbempty
299: (16)                        while item in names:
300: (20)                            nbempty += 1

```

```

301: (20)                     item = defaultfmt % nbempty
302: (16)                     nbempty += 1
303: (12)                     elif item in excludelist:
304: (16)                         item += '_'
305: (12)                         cnt = seen.get(item, 0)
306: (12)                         if cnt > 0:
307: (16)                             validatednames.append(item + '_%d' % cnt)
308: (12)                         else:
309: (16)                             validatednames.append(item)
310: (12)                         seen[item] = cnt + 1
311: (8)                     return tuple(validatednames)
312: (4)                     def __call__(self, names, defaultfmt="f%i", nbfields=None):
313: (8)                         return self.validate(names, defaultfmt=defaultfmt, nbfields=nbfields)
314: (0)                     def str2bool(value):
315: (4)                         """
316: (4)             Tries to transform a string supposed to represent a boolean to a boolean.
317: (4)             Parameters
318: (4)                 -----
319: (4)                 value : str
320: (8)                     The string that is transformed to a boolean.
321: (4)             Returns
322: (4)                 -----
323: (4)                 boolval : bool
324: (8)                     The boolean representation of `value`.
325: (4)             Raises
326: (4)                 -----
327: (4)             ValueError
328: (8)                 If the string is not 'True' or 'False' (case independent)
329: (4)             Examples
330: (4)                 -----
331: (4)                 >>> np.lib._iotools.str2bool('TRUE')
332: (4)                 True
333: (4)                 >>> np.lib._iotoools.str2bool('false')
334: (4)                 False
335: (4)                 """
336: (4)                 value = value.upper()
337: (4)                 if value == 'TRUE':
338: (8)                     return True
339: (4)                 elif value == 'FALSE':
340: (8)                     return False
341: (4)                 else:
342: (8)                     raise ValueError("Invalid boolean")
343: (0)                     class ConverterError(Exception):
344: (4)                         """
345: (4)                         Exception raised when an error occurs in a converter for string values.
346: (4)                         """
347: (4)                         pass
348: (0)                     class ConverterLockError(ConverterError):
349: (4)                         """
350: (4)                         Exception raised when an attempt is made to upgrade a locked converter.
351: (4)                         """
352: (4)                         pass
353: (0)                     class ConversionWarning(UserWarning):
354: (4)                         """
355: (4)                         Warning issued when a string converter has a problem.
356: (4)                         Notes
357: (4)                         -----
358: (4)                         In `genfromtxt` a `ConversionWarning` is issued if raising exceptions
359: (4)                         is explicitly suppressed with the "invalid_raise" keyword.
360: (4)                         """
361: (4)                         pass
362: (0)                     class StringConverter:
363: (4)                         """
364: (4)                         Factory class for function transforming a string into another object
365: (4)                         (int, float).
366: (4)                         After initialization, an instance can be called to transform a string
367: (4)                         into another object. If the string is recognized as representing a
368: (4)                         missing value, a default value is returned.
369: (4)                         Attributes

```

```

370: (4)
371: (4)
372: (8)
373: (4)
374: (8)
375: (8)
376: (4)
377: (8)
378: (4)
379: (8)
380: (4)
381: (8)
382: (8)
383: (4)
384: (8)
385: (4)
386: (4)
387: (4)
388: (8)
389: (8)
390: (8)
391: (8)
392: (8)
393: (8)
394: (4)
395: (8)
396: (8)
397: (8)
398: (4)
399: (8)
``None``
400: (8)
401: (8)
402: (4)
403: (8)
404: (8)
405: (4)
406: (4)
407: (15)
408: (4)
409: (8)
410: (4)
411: (20)
412: (20)
413: (20)
414: (20)
415: (20)
416: (20)
417: (20)
418: (20)
419: (4)
420: (4)
421: (8)
422: (8)
423: (4)
424: (4)
425: (8)
426: (8)
427: (4)
428: (4)
429: (8)
430: (8)
431: (12)
432: (8)
433: (4)
434: (4)
435: (8)
436: (8)
437: (8)

      -----
      func : function
          Function used for the conversion.
      default : any
          Default value to return when the input corresponds to a missing
          value.
      type : type
          Type of the output.
      _status : int
          Integer representing the order of the conversion.
      _mapper : sequence of tuples
          Sequence of tuples (dtype, function, default value) to evaluate in
          order.
      _locked : bool
          Holds `locked` parameter.

      Parameters
      -----
      dtype_or_func : {None, dtype, function}, optional
          If a `dtype`, specifies the input data type, used to define a basic
          function and a default value for missing data. For example, when
          `dtype` is float, the `func` attribute is set to `float` and the
          default value to `np.nan`. If a function, this function is used to
          convert a string to another object. In this case, it is recommended
          to give an associated default value as input.
      default : any, optional
          Value to return by default, that is, when the string to be
          converted is flagged as missing. If not given, `StringConverter`
          tries to supply a reasonable default value.
      missing_values : {None, sequence of str}, optional
          ``None`` or sequence of strings indicating a missing value. If
          ``None``
          then missing values are indicated by empty entries. The default is
          ``None``.
      locked : bool, optional
          Whether the StringConverter should be locked to prevent automatic
          upgrade or not. Default is False.

      """
      _mapper = [(nx.bool_, str2bool, False),
                 (nx.int_, int, -1),]
      if nx.dtype(nx.int_).itemsize < nx.dtype(nx.int64).itemsize:
          _mapper.append((nx.int64, int, -1))
      _mapper.extend([(nx.float64, float, nx.nan),
                     (nx.complex128, complex, nx.nan + 0j),
                     (nx.longdouble, nx.longdouble, nx.nan),
                     (nx.integer, int, -1),
                     (nx.floating, float, nx.nan),
                     (nx.complexfloating, complex, nx.nan + 0j),
                     (nx.str_, asunicode, '???'),
                     (nx.bytes_, asbytes, '???'),
                     ])
      @classmethod
      def _getdtype(cls, val):
          """Returns the dtype of the input variable."""
          return np.array(val).dtype
      @classmethod
      def _getsubtype(cls, val):
          """Returns the type of the dtype of the input variable."""
          return np.array(val).dtype.type
      @classmethod
      def _dtypeortype(cls, dtype):
          """Returns dtype for datetime64 and type of dtype otherwise."""
          if dtype.type == np.datetime64:
              return dtype
          return dtype.type
      @classmethod
      def upgrade_mapper(cls, func, default=None):
          """
          Upgrade the mapper of a StringConverter by adding a new function and
          its corresponding default.

```

```

438: (8)             The input function (or sequence of functions) and its associated
439: (8)             default value (if any) is inserted in penultimate position of the
440: (8)             mapper. The corresponding type is estimated from the dtype of the
441: (8)             default value.
442: (8)             Parameters
443: (8)
444: (8)             -----
445: (12)            func : var
446: (8)                 Function, or sequence of functions
447: (8)             Examples
448: (8)
449: (8)
450: (8)
451: (8)
452: (8)
453: (8)
454: (8)
455: (12)
456: (12)
457: (8)
458: (12)
459: (16)
460: (20)
461: (16)
462: (12)
463: (16)
464: (12)
465: (16)
466: (16)
467: (12)
468: (16)
469: (4)
470: (4)
471: (8)
472: (12)
473: (16)
474: (8)
475: (12)
476: (16)
477: (8)
478: (4)
479: (17)
480: (8)
481: (8)
482: (12)
483: (12)
484: (12)
485: (12)
486: (8)
487: (12)
488: (16)
489: (16)
490: (12)
491: (16)
492: (20)
493: (30)
494: (20)
495: (16)
496: (16)
497: (20)
498: (24)
499: (20)
500: (24)
501: (16)
502: (12)
503: (16)
self._find_map_entry(dtype)
504: (12)
505: (16)

The input function (or sequence of functions) and its associated
default value (if any) is inserted in penultimate position of the
mapper. The corresponding type is estimated from the dtype of the
default value.

Parameters
-----
func : var
Function, or sequence of functions

Examples
-----
>>> import dateutil.parser
>>> import datetime
>>> dateparser = dateutil.parser.parse
>>> defaultdate = datetime.date(2000, 1, 1)
>>> StringConverter.upgrade_mapper(dateparser, default=defaultdate)
"""

if hasattr(func, '__call__'):
    cls._mapper.insert(-1, (cls._getsubtype(default), func, default))
    return
elif hasattr(func, '__iter__'):
    if isinstance(func[0], (tuple, list)):
        for _ in func:
            cls._mapper.insert(-1, _)
        return
    if default is None:
        default = [None] * len(func)
    else:
        default = list(default)
        default.append([None] * (len(func) - len(default)))
    for fct, dft in zip(func, default):
        cls._mapper.insert(-1, (cls._getsubtype(dft), fct, dft))

@classmethod
def _find_map_entry(cls, dtype):
    for i, (deftype, func, default_def) in enumerate(cls._mapper):
        if dtype.type == deftype:
            return i, (deftype, func, default_def)
    for i, (deftype, func, default_def) in enumerate(cls._mapper):
        if np.issubdtype(dtype.type, deftype):
            return i, (deftype, func, default_def)
    raise LookupError
def __init__(self, dtype_or_func=None, default=None, missing_values=None,
            locked=False):
    self._locked = bool(locked)
    if dtype_or_func is None:
        self.func = str2bool
        self._status = 0
        self.default = default or False
        dtype = np.dtype('bool')
    else:
        try:
            self.func = None
            dtype = np.dtype(dtype_or_func)
        except TypeError:
            if not hasattr(dtype_or_func, '__call__'):
                errmsg = ("The input argument `dtype` is neither a"
                          " function nor a dtype (got '%s' instead)")
                raise TypeError(errmsg % type(dtype_or_func))
            self.func = dtype_or_func
            if default is None:
                try:
                    default = self.func('0')
                except ValueError:
                    default = None
            dtype = self._getdtype(default)
        try:
            self._status, (_, func, default_def) =
        except LookupError:
            self.default = default

```

```

506: (16)             _, func, _ = self._mapper[-1]
507: (16)             self._status = 0
508: (12)
509: (16)             else:
510: (20)                 if default is None:
511: (16)                     self.default = default_def
512: (20)                 else:
513: (12)                     self.default = default
514: (16)
515: (12)             if self.func is None:
516: (16)                 self.func = func
517: (20)             if self.func == self._mapper[1][1]:
518: (16)                 if issubclass(dtype.type, np.uint64):
519: (20)                     self.func = np.uint64
520: (16)                 elif issubclass(dtype.type, np.int64):
521: (20)                     self.func = np.int64
522: (8)                 else:
523: (12)                     self.func = lambda x: int(float(x))
524: (8)
525: (12)             if missing_values is None:
526: (16)                 self.missing_values = {''}
527: (12)
528: (8)
529: (8)
530: (8)
531: (8)
532: (4)             self._initial_default = default
533: (8)
534: (12)         def _loose_call(self, value):
535: (8)
536: (12)             try:
537: (4)                 return self.func(value)
538: (8)
539: (12)
540: (12)
541: (16)
542: (20)
543: (16)
544: (20)
545: (12)
546: (8)
547: (12)
548: (16)
549: (20)
550: (16)
551: (12)
552: (4)             def __call__(self, value):
553: (8)                 return self._callingfunction(value)
554: (4)
555: (8)
556: (12)
557: (12)
558: (8)
559: (8)
560: (8)
561: (12)
562: (12)
563: (8)
564: (12)
565: (8)
566: (8)
567: (8)
568: (12)
569: (8)
570: (12)
571: (4)
572: (8)
573: (8)
574: (8)
575: (12)
576: (16)
577: (20)
578: (16)
579: (20)
580: (16)
581: (12)
582: (16)
583: (20)
584: (16)
585: (20)
586: (16)
587: (20)
588: (16)
589: (20)
590: (16)
591: (20)
592: (16)
593: (20)
594: (16)
595: (20)
596: (16)
597: (20)
598: (16)
599: (20)
600: (16)
601: (20)
602: (16)
603: (20)
604: (16)
605: (20)
606: (16)
607: (20)
608: (16)
609: (20)
610: (16)
611: (20)
612: (16)
613: (20)
614: (16)
615: (20)
616: (16)
617: (20)
618: (16)
619: (20)
620: (16)
621: (20)
622: (16)
623: (20)
624: (16)
625: (20)
626: (16)
627: (20)
628: (16)
629: (20)
630: (16)
631: (20)
632: (16)
633: (20)
634: (16)
635: (20)
636: (16)
637: (20)
638: (16)
639: (20)
640: (16)
641: (20)
642: (16)
643: (20)
644: (16)
645: (20)
646: (16)
647: (20)
648: (16)
649: (20)
650: (16)
651: (20)
652: (16)
653: (20)
654: (16)
655: (20)
656: (16)
657: (20)
658: (16)
659: (20)
660: (16)
661: (20)
662: (16)
663: (20)
664: (16)
665: (20)
666: (16)
667: (20)
668: (16)
669: (20)
670: (16)
671: (20)
672: (16)
673: (20)
674: (16)
675: (20)
676: (16)
677: (20)
678: (16)
679: (20)
680: (16)
681: (20)
682: (16)
683: (20)
684: (16)
685: (20)
686: (16)
687: (20)
688: (16)
689: (20)
690: (16)
691: (20)
692: (16)
693: (20)
694: (16)
695: (20)
696: (16)
697: (20)
698: (16)
699: (20)
700: (16)
701: (20)
702: (16)
703: (20)
704: (16)
705: (20)
706: (16)
707: (20)
708: (16)
709: (20)
710: (16)
711: (20)
712: (16)
713: (20)
714: (16)
715: (20)
716: (16)
717: (20)
718: (16)
719: (20)
720: (16)
721: (20)
722: (16)
723: (20)
724: (16)
725: (20)
726: (16)
727: (20)
728: (16)
729: (20)
730: (16)
731: (20)
732: (16)
733: (20)
734: (16)
735: (20)
736: (16)
737: (20)
738: (16)
739: (20)
740: (16)
741: (20)
742: (16)
743: (20)
744: (16)
745: (20)
746: (16)
747: (20)
748: (16)
749: (20)
750: (16)
751: (20)
752: (16)
753: (20)
754: (16)
755: (20)
756: (16)
757: (20)
758: (16)
759: (20)
760: (16)
761: (20)
762: (16)
763: (20)
764: (16)
765: (20)
766: (16)
767: (20)
768: (16)
769: (20)
770: (16)
771: (20)
772: (16)
773: (20)
774: (16)
775: (20)
776: (16)
777: (20)
778: (16)
779: (20)
780: (16)
781: (20)
782: (16)
783: (20)
784: (16)
785: (20)
786: (16)
787: (20)
788: (16)
789: (20)
790: (16)
791: (20)
792: (16)
793: (20)
794: (16)
795: (20)
796: (16)
797: (20)
798: (16)
799: (20)
800: (16)
801: (20)
802: (16)
803: (20)
804: (16)
805: (20)
806: (16)
807: (20)
808: (16)
809: (20)
810: (16)
811: (20)
812: (16)
813: (20)
814: (16)
815: (20)
816: (16)
817: (20)
818: (16)
819: (20)
820: (16)
821: (20)
822: (16)
823: (20)
824: (16)
825: (20)
826: (16)
827: (20)
828: (16)
829: (20)
830: (16)
831: (20)
832: (16)
833: (20)
834: (16)
835: (20)
836: (16)
837: (20)
838: (16)
839: (20)
840: (16)
841: (20)
842: (16)
843: (20)
844: (16)
845: (20)
846: (16)
847: (20)
848: (16)
849: (20)
850: (16)
851: (20)
852: (16)
853: (20)
854: (16)
855: (20)
856: (16)
857: (20)
858: (16)
859: (20)
860: (16)
861: (20)
862: (16)
863: (20)
864: (16)
865: (20)
866: (16)
867: (20)
868: (16)
869: (20)
870: (16)
871: (20)
872: (16)
873: (20)
874: (16)
875: (20)
876: (16)
877: (20)
878: (16)
879: (20)
880: (16)
881: (20)
882: (16)
883: (20)
884: (16)
885: (20)
886: (16)
887: (20)
888: (16)
889: (20)
890: (16)
891: (20)
892: (16)
893: (20)
894: (16)
895: (20)
896: (16)
897: (20)
898: (16)
899: (20)
900: (16)
901: (20)
902: (16)
903: (20)
904: (16)
905: (20)
906: (16)
907: (20)
908: (16)
909: (20)
910: (16)
911: (20)
912: (16)
913: (20)
914: (16)
915: (20)
916: (16)
917: (20)
918: (16)
919: (20)
920: (16)
921: (20)
922: (16)
923: (20)
924: (16)
925: (20)
926: (16)
927: (20)
928: (16)
929: (20)
930: (16)
931: (20)
932: (16)
933: (20)
934: (16)
935: (20)
936: (16)
937: (20)
938: (16)
939: (20)
940: (16)
941: (20)
942: (16)
943: (20)
944: (16)
945: (20)
946: (16)
947: (20)
948: (16)
949: (20)
950: (16)
951: (20)
952: (16)
953: (20)
954: (16)
955: (20)
956: (16)
957: (20)
958: (16)
959: (20)
960: (16)
961: (20)
962: (16)
963: (20)
964: (16)
965: (20)
966: (16)
967: (20)
968: (16)
969: (20)
970: (16)
971: (20)
972: (16)
973: (20)
974: (16)
975: (20)
976: (16)
977: (20)
978: (16)
979: (20)
980: (16)
981: (20)
982: (16)
983: (20)
984: (16)
985: (20)
986: (16)
987: (20)
988: (16)
989: (20)
990: (16)
991: (20)
992: (16)
993: (20)
994: (16)
995: (20)
996: (16)
997: (20)
998: (16)
999: (20)
1000: (16)
1001: (20)
1002: (16)
1003: (20)
1004: (16)
1005: (20)
1006: (16)
1007: (20)
1008: (16)
1009: (20)
1010: (16)
1011: (20)
1012: (16)
1013: (20)
1014: (16)
1015: (20)
1016: (16)
1017: (20)
1018: (16)
1019: (20)
1020: (16)
1021: (20)
1022: (16)
1023: (20)
1024: (16)
1025: (20)
1026: (16)
1027: (20)
1028: (16)
1029: (20)
1030: (16)
1031: (20)
1032: (16)
1033: (20)
1034: (16)
1035: (20)
1036: (16)
1037: (20)
1038: (16)
1039: (20)
1040: (16)
1041: (20)
1042: (16)
1043: (20)
1044: (16)
1045: (20)
1046: (16)
1047: (20)
1048: (16)
1049: (20)
1050: (16)
1051: (20)
1052: (16)
1053: (20)
1054: (16)
1055: (20)
1056: (16)
1057: (20)
1058: (16)
1059: (20)
1060: (16)
1061: (20)
1062: (16)
1063: (20)
1064: (16)
1065: (20)
1066: (16)
1067: (20)
1068: (16)
1069: (20)
1070: (16)
1071: (20)
1072: (16)
1073: (20)
1074: (16)
1075: (20)
1076: (16)
1077: (20)
1078: (16)
1079: (20)
1080: (16)
1081: (20)
1082: (16)
1083: (20)
1084: (16)
1085: (20)
1086: (16)
1087: (20)
1088: (16)
1089: (20)
1090: (16)
1091: (20)
1092: (16)
1093: (20)
1094: (16)
1095: (20)
1096: (16)
1097: (20)
1098: (16)
1099: (20)
1100: (16)
1101: (20)
1102: (16)
1103: (20)
1104: (16)
1105: (20)
1106: (16)
1107: (20)
1108: (16)
1109: (20)
1110: (16)
1111: (20)
1112: (16)
1113: (20)
1114: (16)
1115: (20)
1116: (16)
1117: (20)
1118: (16)
1119: (20)
1120: (16)
1121: (20)
1122: (16)
1123: (20)
1124: (16)
1125: (20)
1126: (16)
1127: (20)
1128: (16)
1129: (20)
1130: (16)
1131: (20)
1132: (16)
1133: (20)
1134: (16)
1135: (20)
1136: (16)
1137: (20)
1138: (16)
1139: (20)
1140: (16)
1141: (20)
1142: (16)
1143: (20)
1144: (16)
1145: (20)
1146: (16)
1147: (20)
1148: (16)
1149: (20)
1150: (16)
1151: (20)
1152: (16)
1153: (20)
1154: (16)
1155: (20)
1156: (16)
1157: (20)
1158: (16)
1159: (20)
1160: (16)
1161: (20)
1162: (16)
1163: (20)
1164: (16)
1165: (20)
1166: (16)
1167: (20)
1168: (16)
1169: (20)
1170: (16)
1171: (20)
1172: (16)
1173: (20)
1174: (16)
1175: (20)
1176: (16)
1177: (20)
1178: (16)
1179: (20)
1180: (16)
1181: (20)
1182: (16)
1183: (20)
1184: (16)
1185: (20)
1186: (16)
1187: (20)
1188: (16)
1189: (20)
1190: (16)
1191: (20)
1192: (16)
1193: (20)
1194: (16)
1195: (20)
1196: (16)
1197: (20)
1198: (16)
1199: (20)
1200: (16)
1201: (20)
1202: (16)
1203: (20)
1204: (16)
1205: (20)
1206: (16)
1207: (20)
1208: (16)
1209: (20)
1210: (16)
1211: (20)
1212: (16)
1213: (20)
1214: (16)
1215: (20)
1216: (16)
1217: (20)
1218: (16)
1219: (20)
1220: (16)
1221: (20)
1222: (16)
1223: (20)
1224: (16)
1225: (20)
1226: (16)
1227: (20)
1228: (16)
1229: (20)
1230: (16)
1231: (20)
1232: (16)
1233: (20)
1234: (16)
1235: (20)
1236: (16)
1237: (20)
1238: (16)
1239: (20)
1240: (16)
1241: (20)
1242: (16)
1243: (20)
1244: (16)
1245: (20)
1246: (16)
1247: (20)
1248: (16)
1249: (20)
1250: (16)
1251: (20)
1252: (16)
1253: (20)
1254: (16)
1255: (20)
1256: (16)
1257: (20)
1258: (16)
1259: (20)
1260: (16)
1261: (20)
1262: (16)
1263: (20)
1264: (16)
1265: (20)
1266: (16)
1267: (20)
1268: (16)
1269: (20)
1270: (16)
1271: (20)
1272: (16)
1273: (20)
1274: (16)
1275: (20)
1276: (16)
1277: (20)
1278: (16)
1279: (20)
1280: (16)
1281: (20)
1282: (16)
1283: (20)
1284: (16)
1285: (20)
1286: (16)
1287: (20)
1288: (16)
1289: (20)
1290: (16)
1291: (20)
1292: (16)
1293: (20)
1294: (16)
1295: (20)
1296: (16)
1297: (20)
1298: (16)
1299: (20)
1300: (16)
1301: (20)
1302: (16)
1303: (20)
1304: (16)
1305: (20)
1306: (16)
1307: (20)
1308: (16)
1309: (20)
1310: (16)
1311: (20)
1312: (16)
1313: (20)
1314: (16)
1315: (20)
1316: (16)
1317: (20)
1318: (16)
1319: (20)
1320: (16)
1321: (20)
1322: (16)
1323: (20)
1324: (16)
1325: (20)
1326: (16)
1327: (20)
1328: (16)
1329: (20)
1330: (16)
1331: (20)
1332: (16)
1333: (20)
1334: (16)
1335: (20)
1336: (16)
1337: (20)
1338: (16)
1339: (20)
1340: (16)
1341: (20)
1342: (16)
1343: (20)
1344: (16)
1345: (20)
1346: (16)
1347: (20)
1348: (16)
1349: (20)
1350: (16)
1351: (20)
1352: (16)
1353: (20)
1354: (16)
1355: (20)
1356: (16)
1357: (20)
1358: (16)
1359: (20)
1360: (16)
1361: (20)
1362: (16)
1363: (20)
1364: (16)
1365: (20)
1366: (16)
1367: (20)
1368: (16)
1369: (20)
1370: (16)
1371: (20)
1372: (16)
1373: (20)
1374: (16)
1375: (20)
1376: (16)
1377: (20)
1378: (16)
1379: (20)
1380: (16)
1381: (20)
1382: (16)
1383: (20)
1384: (16)
1385: (20)
1386: (16)
1387: (20)
1388: (16)
1389: (20)
1390: (16)
1391: (20)
1392: (16)
1393: (20)
1394: (16)
1395: (20)
1396: (16)
1397: (20)
1398: (16)
1399: (20)
1400: (16)
1401: (20)
1402: (16)
1403: (20)
1404: (16)
1405: (20)
1406: (16)
1407: (20)
1408: (16)
1409: (20)
1410: (16)
1411: (20)
1412: (16)
1413: (20)
1414: (16)
1415: (20)
1416: (16)
1417: (20)
1418: (16)
1419: (20)
1420: (16)
1421: (20)
1422: (16)
1423: (20)
1424: (16)
1425: (20)
1426: (16)
1427: (20)
1428: (16)
1429: (20)
1430: (16)
1431: (20)
1432: (16)
1433: (20)
1434: (16)
1435: (20)
1436: (16)
1437: (20)
1438: (16)
1439: (20)
1440: (16)
1441: (20)
1442: (16)
1443: (20)
1444: (16)
1445: (20)
1446: (16)
1447: (20)
1448: (16)
1449: (20)
1450: (16)
1451: (20)
1452: (16)
1453: (20)
1454: (16)
1455: (20)
1456: (16)
1457: (20)
1458: (16)
1459: (20)
1460: (16)
1461: (20)
1462: (16)
1463: (20)
1464: (16)
1465: (20)
1466: (16)
1467: (20)
1468: (16)
1469: (20)
1470: (16)
1471: (20)
1472: (16)
1473: (20)
1474: (16)
1475: (20)
1476: (16)
1477: (20)
1478: (16)
1479: (20)
1480: (16)
1481: (20)
1482: (16)
1483: (20)
1484: (16)
1485: (20)
1486: (16)
1487: (20)
1488: (16)
1489: (20)
1490: (16)
1491: (20)
1492: (16)
1493: (20)
1494: (16)
1495: (20)
1496: (16)
1497: (20)
1498: (16)
1499: (20)
1500: (16)
1501: (20)
1502: (16)
1503: (20)
1504: (16)
1505: (20)
1506: (16)
1507: (20)
1508: (16)
1509: (20)
1510: (16)
1511: (20)
1512: (16)
1513: (20)
1514: (16)
1515: (20)
1516: (16)
1517: (20)
1518: (16)
1519: (20)
1520: (16)
1521: (20)
1522: (16)
1523: (20)
1524: (16)
1525: (20)
1526: (16)
1527: (20)
1528: (16)
1529: (20)
1530: (16)
1531: (20)
1532: (16)
1533: (20)
1534: (16)
1535: (20)
1536: (16)
1537: (20)
1538: (16)
1539: (20)
1540: (16)
1541: (20)
1542: (16)
1543: (20)
1544: (16)
1545: (20)
1546: (16)
1547: (20)
1548: (16)
1549: (20)
1550: (16)
1551: (20)
1552: (16)
1553: (20)
1554: (16)
1555: (20)
1556: (16)
1557: (20)
1558: (16)
1559: (20)
1560: (16)
1561: (20)
1562: (16)
1563: (20)
1564: (16)
1565: (20)
1566: (16)
1567: (20)
1568: (16)
1569: (20)
1570: (16)
1571: (20)
1572: (16)
1573: (20)
1574: (16)
1575: (20)
1576: (16)
1577: (20)
1578: (16)
1579: (20)
1580: (16)
1581: (20)
1582: (16)
1583: (20)
1584: (16)
1585: (20)
1586: (16)
1587: (20)
1588: (16)
1589: (20)
1590: (16)
1591: (20)
1592: (16)
1593: (20)
1594: (16)
1595: (20)
1596: (16)
1597: (20)
1598: (16)
1599: (20)
1600: (16)
1601: (20)
1602: (16)
1603: (20)
1604: (16)
1605: (20)
1606: (16)
1607: (20)
1608: (16)
1609: (20)
1610: (16)
1611: (20)
1612: (16)
1613: (20)
1614: (16)
1615: (20)
1616: (16)
1617: (20)
1618: (16)
1619: (20)
1620: (16)
1621: (20)
1622: (16)
1623: (20)
1624: (16)
1625: (20)
1626: (16)
1627: (20)
1628: (16)
1629: (20)
1630: (16)
1631: (20)
1632: (16)
1633: (20)
1634: (16)
1635: (20)
1636: (16)
1637: (20)
1638: (16)
1639: (20)
1640: (16)
1641: (20)
1642: (16)
1643: (20)
1644: (16)
1645: (20)
1646: (16)
1647: (20)
1648: (16)
1649: (20)
1650: (16)
1651: (20)
1652: (16)
1653: (20)
1654: (16)
1655: (20)
1656: (16)
1657: (20)
1658: (16)
1659: (20)
1660: (16)
1661: (20)
1662: (16)
1663: (20)
1664: (16)
1665: (20)
1666: (16)
1667: (20)
1668: (16)
1669: (20)
1670: (16)
1671: (20)
1672: (16)
1673: (20)
1674: (16)
1675: (20)
1676: (16)
1677: (20)
1678: (16)
1679: (20)
1680: (16)
1681: (20)
1682: (16)
1683: (20)
1684: (16)
1685: (20)
1686: (16)
1687: (20)
1688: (16)
1689: (20)
1690: (16)
1691: (20)
1692: (16)
1693: (20)
1694: (16)
1695: (20)
1696: (16)
1697: (20)
1698: (16)
1699: (20)
1700: (16)
1701: (20)
1702: (16)
1703: (20)
1704: (16)
1705: (20)
1706: (16)
1707: (20)
1708: (16)
1709: (20)
1710: (16)
1711: (20)
1712: (16)
1713: (20)
1714: (16)
1715: (20)
1716: (16)
1717: (20)
1718: (16)
1719: (20)
1720: (16)
1721: (20)
1722: (16)
1723: (20)
1724: (16)
1725: (20)
1726: (16)
1727: (20)
1728: (16)
1729: (20)
1730: (16)
1731: (20)
1732: (16)
1733: (20)
1734: (16)
1735: (20)
1736: (16)
1737: (20)
1738: (16)
1739: (20)
1740: (16)
1741: (20)
1742: (16)
1743: (20)
1744: (16)
1745: (20)
1746: (16)
1747: (20)
1748: (16)
1749: (20)
1750: (16)
1751: (20)
1752: (16)
1753: (20)
1754: (16)
1755: (20)
1756: (16)
1757: (20)
1758: (16)
1759: (20)
1760: (16)
1761: (20)
1762: (16)
1763: (20)
1764: (16)
1765: (20)
1766: (16)
1767: (20)
1768: (16)
1769: (20)
1770: (16)
1771: (20)

```

```

575: (8) converters in order. First the `func` method of the
576: (8) `StringConverter` instance is tried, if this fails other available
577: (8) converters are tried. The order in which these other converters
578: (8) are tried is determined by the `_status` attribute of the instance.
579: (8) Parameters
580: (8) -----
581: (8) value : str
582: (12)     The string to convert.
583: (8) Returns
584: (8) -----
585: (8) out : any
586: (12)     The result of converting `value` with the appropriate converter.
587: (8)
588: (8)     self._checked = True
589: (8) try:
590: (12)     return self._strict_call(value)
591: (8) except ValueError:
592: (12)     self._do_upgrade()
593: (12)     return self.upgrade(value)
594: (4) def iterupgrade(self, value):
595: (8)     self._checked = True
596: (8)     if not hasattr(value, '__iter__'):
597: (12)         value = (value,)
598: (8)     _strict_call = self._strict_call
599: (8) try:
600: (12)     for _m in value:
601: (16)         _strict_call(_m)
602: (8)     except ValueError:
603: (12)         self._do_upgrade()
604: (12)         self.iterupgrade(value)
605: (4) def update(self, func, default=None, testing_value=None,
606: (15)     missing_values='', locked=False):
607: (8)
608: (8)     Set StringConverter attributes directly.
609: (8) Parameters
610: (8) -----
611: (8) func : function
612: (12)     Conversion function.
613: (8) default : any, optional
614: (12)     Value to return by default, that is, when the string to be
615: (12)     converted is flagged as missing. If not given,
616: (12)     `StringConverter` tries to supply a reasonable default value.
617: (8) testing_value : str, optional
618: (12)     A string representing a standard input value of the converter.
619: (12)     This string is used to help defining a reasonable default
620: (12)     value.
621: (8) missing_values : {sequence of str, None}, optional
622: (12)     Sequence of strings indicating a missing value. If ``None``, then
623: (12)     the existing `missing_values` are cleared. The default is ````.
624: (8) locked : bool, optional
625: (12)     Whether the StringConverter should be locked to prevent
626: (12)     automatic upgrade or not. Default is False.
627: (8) Notes
628: (8) -----
629: (8)     `update` takes the same parameters as the constructor of
630: (8)     `StringConverter`, except that `func` does not accept a `dtype`-
631: (8)     whereas `dtype_or_func` in the constructor does.
632: (8)
633: (8)     self.func = func
634: (8)     self._locked = locked
635: (8)     if default is not None:
636: (12)         self.default = default
637: (12)         self.type = self._dtypeortype(self._getdtype(default))
638: (8)     else:
639: (12)         try:
640: (16)             tester = func(testing_value or '1')
641: (12)         except (TypeError, ValueError):
642: (16)             tester = None
643: (12)         self.type = self._dtypeortype(self._getdtype(tester))

```

```

644: (8)             if missing_values is None:
645: (12)            self.missing_values = set()
646: (8)
647: (12)            else:
648: (16)              if not np.iterable(missing_values):
649: (12)                missing_values = [missing_values]
650: (16)                if not all(isinstance(v, str) for v in missing_values):
651: (12)                  raise TypeError("missing_values must be strings or unicode")
652: (0)                  self.missing_values.update(missing_values)
653: (4)    def easy_dtype(ndtype, names=None, defaultfmt="f%i", **validationargs):
654: (4)        """
655: (4)            Convenience function to create a `np.dtype` object.
656: (4)            The function processes the input `dtype` and matches it with the given
657: (4)            names.
658: (4)            Parameters
659: (4)            -----
660: (4)            ndtype : var
661: (8)              Definition of the dtype. Can be any string or dictionary recognized
662: (8)              by the `np.dtype` function, or a sequence of types.
663: (4)            names : str or sequence, optional
664: (8)              Sequence of strings to use as field names for a structured dtype.
665: (8)              For convenience, `names` can be a string of a comma-separated list
666: (8)              of names.
667: (4)            defaultfmt : str, optional
668: (8)              Format string used to define missing names, such as ``"f%i"```
669: (8)              (default) or ``"fields_%02i"``.
670: (4)            validationargs : optional
671: (8)              A series of optional arguments used to initialize a
672: (8)              `NameValidator`.
673: (4)            Examples
674: (4)            -----
675: (4)            >>> np.lib._iotools.easy_dtype(float)
676: (4)            dtype('float64')
677: (4)            >>> np.lib._iotools.easy_dtype("i4, f8")
678: (4)            dtype([('f0', '<i4'), ('f1', '<f8')])
679: (4)            >>> np.lib._iotoools.easy_dtype("i4, f8", defaultfmt="field_%03i")
680: (4)            dtype([('field_000', '<i4'), ('field_001', '<f8')])
681: (4)            >>> np.lib._iotoools.easy_dtype((int, float, float), names="a,b,c")
682: (4)            dtype([('a', '<i8'), ('b', '<f8'), ('c', '<f8')])
683: (4)            >>> np.lib._iotoools.easy_dtype(float, names="a,b,c")
684: (4)            dtype([('a', '<f8'), ('b', '<f8'), ('c', '<f8')])
685: (4)            """
686: (4)            try:
687: (8)                ndtype = np.dtype(ndtype)
688: (4)            except TypeError:
689: (8)                validate = NameValidator(**validationargs)
690: (8)                nbfields = len(ndtype)
691: (8)                if names is None:
692: (12)                  names = [''] * len(ndtype)
693: (8)                elif isinstance(names, str):
694: (12)                  names = names.split(",")
695: (8)                names = validate(names, nbfields=nbfields, defaultfmt=defaultfmt)
696: (4)                ndtype = np.dtype(dict(formats=ndtype, names=names))
697: (8)            else:
698: (12)              if names is not None:
699: (12)                validate = NameValidator(**validationargs)
700: (16)                if isinstance(names, str):
701: (12)                  names = names.split ","
702: (16)                  if ndtype.names is None:
703: (16)                    formats = tuple([ndtype.type] * len(names))
704: (16)                    names = validate(names, defaultfmt=defaultfmt)
705: (12)                    ndtype = np.dtype(list(zip(names, formats)))
706: (16)                else:
707: (40)                  ndtype.names = validate(names, nbfields=len(ndtype.names),
708: (8)                                  defaultfmt=defaultfmt)
709: (12)            elif ndtype.names is not None:
710: (12)              validate = NameValidator(**validationargs)
711: (12)              numbered_names = tuple("f%i" % i for i in
range(len(ndtype.names)))
711: (12)              if ((ndtype.names == numbered_names) and (defaultfmt != "f%i")):
```

```

712: (16)             ndtype.names = validate([''] * len(ndtype.names),
713: (40)                         defaultfmt=defaultfmt)
714: (12)             else:
715: (16)                 ndtype.names = validate(ndtype.names, defaultfmt=defaultfmt)
716: (4)             return ndtype
-----
```

## File 227 - \_version.py:

```

1: (0)             """Utility to compare (NumPy) version strings.
2: (0)             The NumpyVersion class allows properly comparing numpy version strings.
3: (0)             The LooseVersion and StrictVersion classes that distutils provides don't
4: (0)             work; they don't recognize anything like alpha/beta/rc/dev versions.
5: (0)             """
6: (0)             import re
7: (0)             __all__ = ['NumpyVersion']
8: (0)             class NumpyVersion():
9: (4)                 """Parse and compare numpy version strings.
10: (4)                 NumPy has the following versioning scheme (numbers given are examples;
they
11: (4)                     can be > 9 in principle):
12: (4)                     - Released version: '1.8.0', '1.8.1', etc.
13: (4)                     - Alpha: '1.8.0a1', '1.8.0a2', etc.
14: (4)                     - Beta: '1.8.0b1', '1.8.0b2', etc.
15: (4)                     - Release candidates: '1.8.0rc1', '1.8.0rc2', etc.
16: (4)                     - Development versions: '1.8.0.dev-f1234afa' (git commit hash appended)
17: (4)                     - Development versions after a1: '1.8.0a1.dev-f1234afa',
18: (37)                             '1.8.0b2.dev-f1234afa',
19: (37)                             '1.8.1rc1.dev-f1234afa', etc.
20: (4)                     - Development versions (no git hash available): '1.8.0.dev-Unknown'
21: (4)                     Comparing needs to be done against a valid version string or other
22: (4)                     `NumpyVersion` instance. Note that all development versions of the same
23: (4)                     (pre-)release compare equal.
24: (4)                     .. versionadded:: 1.9.0
25: (4)                     Parameters
26: (4)                     -----
27: (4)                     vstring : str
28: (8)                         NumPy version string (``np.__version__``).
29: (4)                     Examples
30: (4)                     -----
31: (4)                     >>> from numpy.lib import NumpyVersion
32: (4)                     >>> if NumpyVersion(np.__version__) < '1.7.0':
33: (4)                         ...     print('skip')
34: (4)                     >>> # skip
35: (4)                     >>> NumpyVersion('1.7') # raises ValueError, add ".0"
36: (4)                     Traceback (most recent call last):
37: (8)                         ...
38: (4)                         ValueError: Not a valid numpy version string
39: (4)                         """
40: (4)                     def __init__(self, vstring):
41: (8)                         self.vstring = vstring
42: (8)                         ver_main = re.match(r'\d+\.\d+\.\d+', vstring)
43: (8)                         if not ver_main:
44: (12)                             raise ValueError("Not a valid numpy version string")
45: (8)                         self.version = ver_main.group()
46: (8)                         self.major, self.minor, self.bugfix = [int(x) for x in
47: (12)                             self.version.split('.')]
48: (8)                         if len(vstring) == ver_main.end():
49: (12)                             self.pre_release = 'final'
50: (8)                         else:
51: (12)                             alpha = re.match(r'a\d', vstring[ver_main.end():])
52: (12)                             beta = re.match(r'b\d', vstring[ver_main.end():])
53: (12)                             rc = re.match(r'rc\d', vstring[ver_main.end():])
54: (12)                             pre_rel = [m for m in [alpha, beta, rc] if m is not None]
55: (12)                             if pre_rel:
56: (16)                                 self.pre_release = pre_rel[0].group()
57: (12)                             else:
58: (16)                                 self.pre_release = ''
```

```

59: (8)             self.is_devversion = bool(re.search(r'.dev', vstring))
60: (4)         def _compare_version(self, other):
61: (8)             """Compare major.minor.bugfix"""
62: (8)             if self.major == other.major:
63: (12)                 if self.minor == other.minor:
64: (16)                     if self.bugfix == other.bugfix:
65: (20)                         vercmp = 0
66: (16)                     elif self.bugfix > other.bugfix:
67: (20)                         vercmp = 1
68: (16)                     else:
69: (20)                         vercmp = -1
70: (12)                 elif self.minor > other.minor:
71: (16)                     vercmp = 1
72: (12)                 else:
73: (16)                     vercmp = -1
74: (8)             elif self.major > other.major:
75: (12)                 vercmp = 1
76: (8)             else:
77: (12)                 vercmp = -1
78: (8)             return vercmp
79: (4)         def _compare_pre_release(self, other):
80: (8)             """Compare alpha/beta/rc/final."""
81: (8)             if self.pre_release == other.pre_release:
82: (12)                 vercmp = 0
83: (8)             elif self.pre_release == 'final':
84: (12)                 vercmp = 1
85: (8)             elif other.pre_release == 'final':
86: (12)                 vercmp = -1
87: (8)             elif self.pre_release > other.pre_release:
88: (12)                 vercmp = 1
89: (8)             else:
90: (12)                 vercmp = -1
91: (8)             return vercmp
92: (4)         def _compare(self, other):
93: (8)             if not isinstance(other, (str, NumpyVersion)):
94: (12)                 raise ValueError("Invalid object to compare with NumpyVersion.")
95: (8)             if isinstance(other, str):
96: (12)                 other = NumpyVersion(other)
97: (8)             vercmp = self._compare_version(other)
98: (8)             if vercmp == 0:
99: (12)                 vercmp = self._compare_pre_release(other)
100: (12)                 if vercmp == 0:
101: (16)                     if self.is_devversion is other.is_devversion:
102: (20)                         vercmp = 0
103: (16)                     elif self.is_devversion:
104: (20)                         vercmp = -1
105: (16)                     else:
106: (20)                         vercmp = 1
107: (8)             return vercmp
108: (4)         def __lt__(self, other):
109: (8)             return self._compare(other) < 0
110: (4)         def __le__(self, other):
111: (8)             return self._compare(other) <= 0
112: (4)         def __eq__(self, other):
113: (8)             return self._compare(other) == 0
114: (4)         def __ne__(self, other):
115: (8)             return self._compare(other) != 0
116: (4)         def __gt__(self, other):
117: (8)             return self._compare(other) > 0
118: (4)         def __ge__(self, other):
119: (8)             return self._compare(other) >= 0
120: (4)         def __repr__(self):
121: (8)             return "NumpyVersion(%s)" % self.vstring

```

-----  
File 228 - \_\_init\_\_.py:

```
1: (0) """

```

```

2: (0) **Note:** almost all functions in the ``numpy.lib`` namespace
3: (0) are also present in the main ``numpy`` namespace. Please use the
4: (0) functions as ``np.<funcname>`` where possible.
5: (0) ``numpy.lib`` is mostly a space for implementing functions that don't
6: (0) belong in core or in another NumPy submodule with a clear purpose
7: (0) (e.g. ``random``, ``fft``, ``linalg``, ``ma``).
8: (0) Most contains basic functions that are used by several submodules and are
9: (0) useful to have in the main name-space.
10: (0)
11: (0) """
12: (0)     from . import mixins
13: (0)     from . import scimath as emath
14: (0)     from . import type_check
15: (0)     from . import index_tricks
16: (0)     from . import function_base
17: (0)     from . import nanfunctions
18: (0)     from . import shape_base
19: (0)     from . import stride_tricks
20: (0)     from . import twodim_base
21: (0)     from . import ufunclike
22: (0)     from . import histograms
23: (0)     from . import polynomial
24: (0)     from . import utils
25: (0)     from . import arraysetops
26: (0)     from . import npyio
27: (0)     from . import arrayterator
28: (0)     from . import _version
29: (0)     from .type_check import *
30: (0)     from .index_tricks import *
31: (0)     from .function_base import *
32: (0)     from .nanfunctions import *
33: (0)     from .shape_base import *
34: (0)     from .stride_tricks import *
35: (0)     from .twodim_base import *
36: (0)     from .ufunclike import *
37: (0)     from .histograms import *
38: (0)     from .polynomial import *
39: (0)     from .utils import *
40: (0)     from .arraysetops import *
41: (0)     from .npyio import *
42: (0)     from .arrayterator import Arrayterator
43: (0)     from .arraypad import *
44: (0)     from ._version import *
45: (0)     from numpy.core._multiarray_umath import tracemalloc_domain
46: (0)     __all__ = ['emath', 'tracemalloc_domain', 'Arrayterator']
47: (0)     __all__ += type_check.__all__
48: (0)     __all__ += index_tricks.__all__
49: (0)     __all__ += function_base.__all__
50: (0)     __all__ += shape_base.__all__
51: (0)     __all__ += stride_tricks.__all__
52: (0)     __all__ += twodim_base.__all__
53: (0)     __all__ += ufunclike.__all__
54: (0)     __all__ += arraypad.__all__
55: (0)     __all__ += polynomial.__all__
56: (0)     __all__ += utils.__all__
57: (0)     __all__ += arraysetops.__all__
58: (0)     __all__ += npyio.__all__
59: (0)     __all__ += nanfunctions.__all__
60: (0)     __all__ += histograms.__all__
61: (0)     from numpy._pytesttester import PytestTester
62: (0)     test = PytestTester(__name__)
63: (0)     del PytestTester
64: (0)     def __getattr__(attr):
65: (4)         import math
66: (4)         import warnings
67: (4)         if attr == 'math':
68: (8)             warnings.warn(
69: (12)                 "``np.lib.math`` is a deprecated alias for the standard library `"
70: (12)                 "`math` module (Deprecated Numpy 1.25). Replace usages of "

```

```

71: (12)                     ``numpy.lib.math`` with `math`, DeprecationWarning, stacklevel=2)
72: (8)                      return math
73: (4)                      else:
74: (8)                          raise AttributeError("module {!r} has no attribute "
75: (29)                                "{}".format(__name__, attr))
-----
```

File 229 - test\_arraypad.py:

```

1: (0)                      """Tests for the array padding functions.
2: (0)                      """
3: (0)                      import pytest
4: (0)                      import numpy as np
5: (0)                      from numpy.testing import assert_array_equal, assert_allclose, assert_equal
6: (0)                      from numpy.lib.arraypad import _as_pairs
7: (0)                      _numeric_dtypes = (
8: (4)                          np.sctypes["uint"]
9: (4)                          + np.sctypes["int"]
10: (4)                         + np.sctypes["float"]
11: (4)                         + np.sctypes["complex"]
12: (0) )
13: (0)                      _all_modes = {
14: (4)                          'constant': {'constant_values': 0},
15: (4)                          'edge': {},
16: (4)                          'linear_ramp': {'end_values': 0},
17: (4)                          'maximum': {'stat_length': None},
18: (4)                          'mean': {'stat_length': None},
19: (4)                          'median': {'stat_length': None},
20: (4)                          'minimum': {'stat_length': None},
21: (4)                          'reflect': {'reflect_type': 'even'},
22: (4)                          'symmetric': {'reflect_type': 'even'},
23: (4)                          'wrap': {},
24: (4)                          'empty': {}
25: (0) }
26: (0)                      class TestAsPairs:
27: (4)                          def test_single_value(self):
28: (8)                              """Test casting for a single value."""
29: (8)                              expected = np.array([[3, 3]] * 10)
30: (8)                              for x in (3, [3], [[3]]):
31: (12)                                 result = _as_pairs(x, 10)
32: (12)                                 assert_equal(result, expected)
33: (8)                                 obj = object()
34: (8)                                 assert_equal(
35: (12)                                     _as_pairs(obj, 10),
36: (12)                                     np.array([[obj, obj]] * 10)
37: (8)                               )
38: (4)                          def test_two_values(self):
39: (8)                              """Test proper casting for two different values."""
40: (8)                              expected = np.array([[3, 4]] * 10)
41: (8)                              for x in ([3, 4], [[3, 4]]):
42: (12)                                 result = _as_pairs(x, 10)
43: (12)                                 assert_equal(result, expected)
44: (8)                                 obj = object()
45: (8)                                 assert_equal(
46: (12)                                     _as_pairs(["a", obj], 10),
47: (12)                                     np.array([["a", obj]] * 10)
48: (8)                               )
49: (8)                                 assert_equal(
50: (12)                                     _as_pairs([[3], [4]], 2),
51: (12)                                     np.array([[3, 3], [4, 4]])
52: (8)                               )
53: (8)                                 assert_equal(
54: (12)                                     _as_pairs([["a"], [obj]], 2),
55: (12)                                     np.array([["a", "a"], [obj, obj]])
56: (8)                               )
57: (4)                          def test_with_none(self):
58: (8)                              expected = ((None, None), (None, None), (None, None))
59: (8)                              assert_equal(
```

```

60: (12)           _as_pairs(None, 3, as_index=False),
61: (12)           expected
62: (8)
63: (8)
64: (12)           assert_equal(
65: (12)             _as_pairs(None, 3, as_index=True),
66: (8)             expected
67: (4)           )
68: (8)           def test_pass_through(self):
69: (8)             """Test if `x` already matching desired output are passed through."""
70: (8)             expected = np.arange(12).reshape((6, 2))
71: (12)             assert_equal(
72: (12)               _as_pairs(expected, 6),
73: (8)               expected
74: (4)           )
75: (8)           def test_as_index(self):
76: (8)             """Test results if `as_index=True`."""
77: (12)             assert_equal(
78: (12)               _as_pairs([2.6, 3.3], 10, as_index=True),
79: (8)                 np.array([[3, 3]] * 10, dtype=np.intp)
80: (8)           )
81: (8)           assert_equal(
82: (12)             _as_pairs([2.6, 4.49], 10, as_index=True),
83: (12)               np.array([[3, 4]] * 10, dtype=np.intp)
84: (8)           )
85: (18)           for x in (-3, [-3], [[-3]], [-3, 4], [3, -4], [[-3, 4]], [[4, -3]],
86: (12)             [[1, 2]] * 9 + [[1, -2]]):
87: (16)             with pytest.raises(ValueError, match="negative values"):
88: (4)               _as_pairs(x, 10, as_index=True)
89: (8)
90: (8)
91: (12)           def test_exceptions(self):
92: (8)             """Ensure faulty usage is discovered."""
93: (12)             with pytest.raises(ValueError, match="more dimensions than allowed"):
94: (8)               _as_pairs([[3]], 10)
95: (12)             with pytest.raises(ValueError, match="could not be broadcast"):
96: (0)               _as_pairs([1, 2], [3, 4], 3)
97: (4)             with pytest.raises(ValueError, match="could not be broadcast"):
98: (4)               _as_pairs(np.ones((2, 3)), 3)
99: (8)
100: (8)
101: (8)
102: (4)
103: (4)
104: (8)
105: (8)
106: (8)
107: (27)
108: (4)
109: (4)
110: (8)
111: (8)
112: (8)
113: (27)
114: (0)
115: (4)
116: (8)
117: (8)
118: (8)
119: (12)
120: (13)
121: (13)
122: (13)
123: (13)
124: (13)
125: (13)
126: (13)

class TestConditionalShortcuts:
    @pytest.mark.parametrize("mode", _all_modes.keys())
    def test_zero_padding_shortcuts(self, mode):
        test = np.arange(120).reshape(4, 5, 6)
        pad_amt = [(0, 0) for _ in test.shape]
        assert_array_equal(test, np.pad(test, pad_amt, mode=mode))
    @pytest.mark.parametrize("mode", ['maximum', 'mean', 'median',
                                   'minimum',])
    def test_shallow_statistic_range(self, mode):
        test = np.arange(120).reshape(4, 5, 6)
        pad_amt = [(1, 1) for _ in test.shape]
        assert_array_equal(np.pad(test, pad_amt, mode='edge'),
                           np.pad(test, pad_amt, mode=mode, stat_length=1))
    @pytest.mark.parametrize("mode", ['maximum', 'mean', 'median',
                                   'minimum',])
    def test_clip_statistic_range(self, mode):
        test = np.arange(30).reshape(5, 6)
        pad_amt = [(3, 3) for _ in test.shape]
        assert_array_equal(np.pad(test, pad_amt, mode=mode),
                           np.pad(test, pad_amt, mode=mode, stat_length=30))

class TestStatistic:
    def test_check_mean_stat_length(self):
        a = np.arange(100).astype('f')
        a = np.pad(a, ((25, 20), ), 'mean', stat_length=((2, 3), ))
        b = np.array([
            0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
            0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,
            0.5, 0.5, 0.5, 0.5, 0.5,
            0., 1., 2., 3., 4., 5., 6., 7., 8., 9.,
            10., 11., 12., 13., 14., 15., 16., 17., 18., 19.,
            20., 21., 22., 23., 24., 25., 26., 27., 28., 29.,
            30., 31., 32., 33., 34., 35., 36., 37., 38., 39.,
            40., 41., 42., 43., 44., 45., 46., 47., 48., 49.,
        ])

```

```

127: (13)           50., 51., 52., 53., 54., 55., 56., 57., 58., 59.,
128: (13)           60., 61., 62., 63., 64., 65., 66., 67., 68., 69.,
129: (13)           70., 71., 72., 73., 74., 75., 76., 77., 78., 79.,
130: (13)           80., 81., 82., 83., 84., 85., 86., 87., 88., 89.,
131: (13)           90., 91., 92., 93., 94., 95., 96., 97., 98., 99.,
132: (13)           98., 98., 98., 98., 98., 98., 98., 98., 98.,
133: (13)           98., 98., 98., 98., 98., 98., 98., 98., 98.,
134: (13)           98.
135: (8)            ]
136: (4)             assert_array_equal(a, b)
def test_check_maximum_1(self):
137: (8)             a = np.arange(100)
138: (8)             a = np.pad(a, (25, 20), 'maximum')
139: (8)             b = np.array(
140: (12)               [99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
141: (13)                 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
142: (13)                 99, 99, 99, 99, 99,
143: (13)                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
144: (13)                 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
145: (13)                 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
146: (13)                 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
147: (13)                 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
148: (13)                 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
149: (13)                 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
150: (13)                 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
151: (13)                 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
152: (13)                 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
153: (13)                 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
154: (13)                 99, 99, 99, 99, 99, 99, 99, 99, 99, 99]
155: (12)               )
156: (8)             assert_array_equal(a, b)
def test_check_maximum_2(self):
157: (4)             a = np.arange(100) + 1
158: (8)             a = np.pad(a, (25, 20), 'maximum')
159: (8)             b = np.array(
160: (8)               [
161: (12)                 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
162: (13)                 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
163: (13)                 100, 100, 100, 100,
164: (13)                 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
165: (13)                 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
166: (13)                 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
167: (13)                 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
168: (13)                 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
169: (13)                 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
170: (13)                 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
171: (13)                 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
172: (13)                 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
173: (13)                 91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
174: (13)                 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
175: (13)                 100, 100, 100, 100, 100, 100, 100, 100, 100, 100]
176: (12)               )
177: (8)             assert_array_equal(a, b)
def test_check_maximum_stat_length(self):
178: (4)             a = np.arange(100) + 1
179: (8)             a = np.pad(a, (25, 20), 'maximum', stat_length=10)
180: (8)             b = np.array(
181: (8)               [
182: (12)                 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
183: (13)                 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
184: (13)                 10, 10, 10, 10,
185: (14)                 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
186: (13)                 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
187: (13)                 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
188: (13)                 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
189: (13)                 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
190: (13)                 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
191: (13)                 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
192: (13)                 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
193: (13)                 81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
194: (13)                 91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
195: (13)                 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,

```

```

196: (13)           100, 100, 100, 100, 100, 100, 100, 100, 100, 100]
197: (12)         )
198: (8)          assert_array_equal(a, b)
199: (4)          def test_check_minimum_1(self):
200: (8)            a = np.arange(100)
201: (8)            a = np.pad(a, (25, 20), 'minimum')
202: (8)            b = np.array(
203: (12)              [0, 0, 0, 0, 0, 0, 0, 0, 0,
204: (13)                0, 0, 0, 0, 0, 0, 0, 0, 0,
205: (13)                0, 0, 0, 0, 0,
206: (13)                0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
207: (13)                10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
208: (13)                20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
209: (13)                30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
210: (13)                40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
211: (13)                50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
212: (13)                60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
213: (13)                70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
214: (13)                80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
215: (13)                90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
216: (13)                0, 0, 0, 0, 0, 0, 0, 0, 0,
217: (13)                0, 0, 0, 0, 0, 0, 0, 0, 0]
218: (12)          )
219: (8)          assert_array_equal(a, b)
220: (4)          def test_check_minimum_2(self):
221: (8)            a = np.arange(100) + 2
222: (8)            a = np.pad(a, (25, 20), 'minimum')
223: (8)            b = np.array(
224: (12)              [2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
225: (13)                2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
226: (13)                2, 2, 2, 2, 2,
227: (13)                2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
228: (13)                12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
229: (13)                22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
230: (13)                32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
231: (13)                42, 43, 44, 45, 46, 47, 48, 49, 50, 51,
232: (13)                52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
233: (13)                62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
234: (13)                72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
235: (13)                82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
236: (13)                92, 93, 94, 95, 96, 97, 98, 99, 100, 101,
237: (13)                2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
238: (13)                2, 2, 2, 2, 2, 2, 2, 2, 2]
239: (12)          )
240: (8)          assert_array_equal(a, b)
241: (4)          def test_check_minimum_stat_length(self):
242: (8)            a = np.arange(100) + 1
243: (8)            a = np.pad(a, (25, 20), 'minimum', stat_length=10)
244: (8)            b = np.array(
245: (12)              [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
246: (14)                1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
247: (14)                1, 1, 1, 1, 1,
248: (14)                1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
249: (13)                11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
250: (13)                21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
251: (13)                31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
252: (13)                41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
253: (13)                51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
254: (13)                61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
255: (13)                71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
256: (13)                81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
257: (13)                91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
258: (13)                91, 91, 91, 91, 91, 91, 91, 91, 91, 91,
259: (13)                91, 91, 91, 91, 91, 91, 91, 91, 91, 91]
260: (12)          )
261: (8)          assert_array_equal(a, b)
262: (4)          def test_check_median(self):
263: (8)            a = np.arange(100).astype('f')
264: (8)            a = np.pad(a, (25, 20), 'median')

```

```

265: (8)             b = np.array(
266: (12)            [49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5,
267: (13)            49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5,
268: (13)            49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5,
269: (13)            0., 1., 2., 3., 4., 5., 6., 7., 8., 9.,
270: (13)            10., 11., 12., 13., 14., 15., 16., 17., 18., 19.,
271: (13)            20., 21., 22., 23., 24., 25., 26., 27., 28., 29.,
272: (13)            30., 31., 32., 33., 34., 35., 36., 37., 38., 39.,
273: (13)            40., 41., 42., 43., 44., 45., 46., 47., 48., 49.,
274: (13)            50., 51., 52., 53., 54., 55., 56., 57., 58., 59.,
275: (13)            60., 61., 62., 63., 64., 65., 66., 67., 68., 69.,
276: (13)            70., 71., 72., 73., 74., 75., 76., 77., 78., 79.,
277: (13)            80., 81., 82., 83., 84., 85., 86., 87., 88., 89.,
278: (13)            90., 91., 92., 93., 94., 95., 96., 97., 98., 99.,
279: (13)            49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5,
280: (13)            49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5
281: (12)           )
282: (8)             assert_array_equal(a, b)
283: (4)             def test_check_median_01(self):
284: (8)               a = np.array([[3, 1, 4], [4, 5, 9], [9, 8, 2]])
285: (8)               a = np.pad(a, 1, 'median')
286: (8)               b = np.array(
287: (12)                 [[4, 4, 5, 4, 4],
288: (13)                   [3, 3, 1, 4, 3],
289: (13)                   [5, 4, 5, 9, 5],
290: (13)                   [8, 9, 8, 2, 8],
291: (13)                   [4, 4, 5, 4, 4]]
292: (12)                 )
293: (8)               assert_array_equal(a, b)
294: (4)               def test_check_median_02(self):
295: (8)                 a = np.array([[3, 1, 4], [4, 5, 9], [9, 8, 2]])
296: (8)                 a = np.pad(a.T, 1, 'median').T
297: (8)                 b = np.array(
298: (12)                   [[5, 4, 5, 4, 5],
299: (13)                     [3, 3, 1, 4, 3],
300: (13)                     [5, 4, 5, 9, 5],
301: (13)                     [8, 9, 8, 2, 8],
302: (13)                     [5, 4, 5, 4, 5]]
303: (12)                   )
304: (8)               assert_array_equal(a, b)
305: (4)               def test_check_median_stat_length(self):
306: (8)                 a = np.arange(100).astype('f')
307: (8)                 a[1] = 2.
308: (8)                 a[97] = 96.
309: (8)                 a = np.pad(a, (25, 20), 'median', stat_length=(3, 5))
310: (8)                 b = np.array(
311: (12)                   [ 2., 2., 2., 2., 2., 2., 2., 2., 2.,
312: (14)                     2., 2., 2., 2., 2., 2., 2., 2., 2.,
313: (14)                     2., 2., 2., 2., 2.,
314: (14)                     0., 2., 2., 3., 4., 5., 6., 7., 8., 9.,
315: (13)                     10., 11., 12., 13., 14., 15., 16., 17., 18., 19.,
316: (13)                     20., 21., 22., 23., 24., 25., 26., 27., 28., 29.,
317: (13)                     30., 31., 32., 33., 34., 35., 36., 37., 38., 39.,
318: (13)                     40., 41., 42., 43., 44., 45., 46., 47., 48., 49.,
319: (13)                     50., 51., 52., 53., 54., 55., 56., 57., 58., 59.,
320: (13)                     60., 61., 62., 63., 64., 65., 66., 67., 68., 69.,
321: (13)                     70., 71., 72., 73., 74., 75., 76., 77., 78., 79.,
322: (13)                     80., 81., 82., 83., 84., 85., 86., 87., 88., 89.,
323: (13)                     90., 91., 92., 93., 94., 95., 96., 96., 98., 99.,
324: (13)                     96., 96., 96., 96., 96., 96., 96., 96., 96., 96.,
325: (13)                     96., 96., 96., 96., 96., 96., 96., 96., 96., 96.]
326: (12)                   )
327: (8)               assert_array_equal(a, b)
328: (4)               def test_check_mean_shape_one(self):
329: (8)                 a = [[4, 5, 6]]
330: (8)                 a = np.pad(a, (5, 7), 'mean', stat_length=2)
331: (8)                 b = np.array(
332: (12)                   [[4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6],
333: (13)                     [4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6],

```

```

334: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
335: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
336: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
337: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
338: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
339: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
340: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
341: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
342: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
343: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6],
344: (13) [4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6]
345: (12) )
346: (8) assert_array_equal(a, b)
347: (4) def test_check_mean_2(self):
348: (8)     a = np.arange(100).astype('f')
349: (8)     a = np.pad(a, (25, 20), 'mean')
350: (8)     b = np.array(
351: (12)         [49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5,
352: (13)         49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5,
353: (13)         49.5, 49.5, 49.5, 49.5, 49.5,
354: (13)         0., 1., 2., 3., 4., 5., 6., 7., 8., 9.,
355: (13)         10., 11., 12., 13., 14., 15., 16., 17., 18., 19.,
356: (13)         20., 21., 22., 23., 24., 25., 26., 27., 28., 29.,
357: (13)         30., 31., 32., 33., 34., 35., 36., 37., 38., 39.,
358: (13)         40., 41., 42., 43., 44., 45., 46., 47., 48., 49.,
359: (13)         50., 51., 52., 53., 54., 55., 56., 57., 58., 59.,
360: (13)         60., 61., 62., 63., 64., 65., 66., 67., 68., 69.,
361: (13)         70., 71., 72., 73., 74., 75., 76., 77., 78., 79.,
362: (13)         80., 81., 82., 83., 84., 85., 86., 87., 88., 89.,
363: (13)         90., 91., 92., 93., 94., 95., 96., 97., 98., 99.,
364: (13)         49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5,
365: (13)         49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5, 49.5]
366: (12) )
367: (8) assert_array_equal(a, b)
368: (4) @pytest.mark.parametrize("mode", [
369: (8)     "mean",
370: (8)     "median",
371: (8)     "minimum",
372: (8)     "maximum"
373: (4) ])
374: (4) def test_same_prepend_append(self, mode):
375: (8)     """ Test that appended and prepended values are equal """
376: (8)     a = np.array([-1, 2, -1]) + np.array([0, 1e-12, 0], dtype=np.float64)
377: (8)     a = np.pad(a, (1, 1), mode)
378: (8)     assert_equal(a[0], a[-1])
379: (4) @pytest.mark.parametrize("mode", ["mean", "median", "minimum", "maximum"])
380: (4) @pytest.mark.parametrize(
381: (8)     "stat_length", [-2, (-2,), (3, -1), ((5, 2), (-2, 3)), ((-4,), (2,))]
382: (4) )
383: (4) def test_check_negative_stat_length(self, mode, stat_length):
384: (8)     arr = np.arange(30).reshape((6, 5))
385: (8)     match = "index can't contain negative values"
386: (8)     with pytest.raises(ValueError, match=match):
387: (12)         np.pad(arr, 2, mode, stat_length=stat_length)
388: (4) def test_simple_stat_length(self):
389: (8)     a = np.arange(30)
390: (8)     a = np.reshape(a, (6, 5))
391: (8)     a = np.pad(a, ((2, 3), (3, 2)), mode='mean', stat_length=(3,))
392: (8)     b = np.array(
393: (12)         [[6, 6, 6, 5, 6, 7, 8, 9, 8, 8],
394: (13)         [6, 6, 6, 5, 6, 7, 8, 9, 8, 8],
395: (13)         [1, 1, 1, 0, 1, 2, 3, 4, 3, 3],
396: (13)         [6, 6, 6, 5, 6, 7, 8, 9, 8, 8],
397: (13)         [11, 11, 11, 10, 11, 12, 13, 14, 13, 13],
398: (13)         [16, 16, 16, 15, 16, 17, 18, 19, 18, 18],
399: (13)         [21, 21, 21, 20, 21, 22, 23, 24, 23, 23],
400: (13)         [26, 26, 26, 25, 26, 27, 28, 29, 28, 28],
401: (13)         [21, 21, 21, 20, 21, 22, 23, 24, 23, 23],
402: (13)         [21, 21, 21, 20, 21, 22, 23, 24, 23, 23],

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

403: (13) [21, 21, 21, 20, 21, 22, 23, 24, 23, 23]]
404: (12)
405: (8) )
406: (4) assert_array_equal(a, b)
407: (4) @pytest.mark.filterwarnings("ignore:Mean of empty slice:RuntimeWarning")
408: (8) @pytest.mark.filterwarnings(
409: (4) "ignore:invalid value encountered in( scalar)? divide:RuntimeWarning"
410: (4) )
411: (4) @pytest.mark.parametrize("mode", ["mean", "median"])
412: (8) def test_zero_stat_length_valid(self, mode):
413: (8) arr = np.pad([1., 2.], (1, 2), mode, stat_length=0)
414: (8) expected = np.array([np.nan, 1., 2., np.nan, np.nan])
415: (8) assert_equal(arr, expected)
416: (4) @pytest.mark.parametrize("mode", ["minimum", "maximum"])
417: (8) def test_zero_stat_length_invalid(self, mode):
418: (8) match = "stat_length of 0 yields no value for padding"
419: (12) with pytest.raises(ValueError, match=match):
420: (8) np.pad([1., 2.], 0, mode, stat_length=0)
421: (12) with pytest.raises(ValueError, match=match):
422: (8) np.pad([1., 2.], 0, mode, stat_length=(1, 0))
423: (12) with pytest.raises(ValueError, match=match):
424: (8) np.pad([1., 2.], 1, mode, stat_length=0)
425: (12) with pytest.raises(ValueError, match=match):
426: (0) np.pad([1., 2.], 1, mode, stat_length=(1, 0))

class TestConstant:
427: (4)     def test_check_constant(self):
428: (8)         a = np.arange(100)
429: (8)         a = np.pad(a, (25, 20), 'constant', constant_values=(10, 20))
430: (8)         b = np.array(
431: (12)             [10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
432: (13)                 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
433: (13)                 10, 10, 10, 10, 10,
434: (13)                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
435: (13)                 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
436: (13)                 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
437: (13)                 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
438: (13)                 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
439: (13)                 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
440: (13)                 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
441: (13)                 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
442: (13)                 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
443: (13)                 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
444: (13)                 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
445: (13)                 20, 20, 20, 20, 20, 20, 20, 20, 20, 20]
446: (12)
447: (8)         )
448: (4)         assert_array_equal(a, b)
449: (8)     def test_check_constant_zeros(self):
450: (8)         a = np.arange(100)
451: (8)         a = np.pad(a, (25, 20), 'constant')
452: (8)         b = np.array(
453: (12)             [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
454: (14)                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
455: (14)                 0, 0, 0, 0, 0,
456: (13)                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
457: (13)                 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
458: (13)                 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
459: (13)                 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
460: (13)                 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
461: (13)                 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
462: (13)                 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
463: (13)                 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
464: (13)                 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
465: (13)                 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
466: (14)                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
467: (12)                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
468: (8)
469: (4)         )
470: (8)         assert_array_equal(a, b)
471: (8)     def test_check_constant_float(self):
472: (8)         arr = np.arange(30).reshape(5, 6)
473: (8)         test = np.pad(arr, (1, 2), mode='constant',

```

```

472: (19)
473: (8)         constant_values=1.1)
474: (12)        expected = np.array(
475: (13)          [[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
476: (13)          [ 1,  0,  1,  2,  3,  4,  5,  1,  1],
477: (13)          [ 1,  6,  7,  8,  9, 10, 11,  1,  1],
478: (13)          [ 1, 12, 13, 14, 15, 16, 17,  1,  1],
479: (13)          [ 1, 18, 19, 20, 21, 22, 23,  1,  1],
480: (13)          [ 1, 24, 25, 26, 27, 28, 29,  1,  1],
481: (13)          [ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1]
482: (12)        )
483: (8)        assert_allclose(test, expected)
484: (4)        def test_check_constant_float2(self):
485: (8)          arr = np.arange(30).reshape(5, 6)
486: (8)          arr_float = arr.astype(np.float64)
487: (8)          test = np.pad(arr_float, ((1, 2), (1, 2)), mode='constant',
488: (19)            constant_values=1.1)
489: (8)        expected = np.array(
490: (12)          [[ 1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1],
491: (13)          [ 1.1,  0. ,  1. ,  2. ,  3. ,  4. ,  5. ,  1.1,  1.1],
492: (13)          [ 1.1,  6. ,  7. ,  8. ,  9. , 10. , 11. ,  1.1,  1.1],
493: (13)          [ 1.1, 12. , 13. , 14. , 15. , 16. , 17. ,  1.1,  1.1],
494: (13)          [ 1.1, 18. , 19. , 20. , 21. , 22. , 23. ,  1.1,  1.1],
495: (13)          [ 1.1, 24. , 25. , 26. , 27. , 28. , 29. ,  1.1,  1.1],
496: (13)          [ 1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1],
497: (13)          [ 1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1,  1.1]
498: (12)        )
499: (8)        assert_allclose(test, expected)
500: (4)        def test_check_constant_float3(self):
501: (8)          a = np.arange(100, dtype=float)
502: (8)          a = np.pad(a, (25, 20), 'constant', constant_values=(-1.1, -1.2))
503: (8)        b = np.array(
504: (12)          [-1.1, -1.1, -1.1, -1.1, -1.1, -1.1, -1.1, -1.1, -1.1,
505: (13)          -1.1, -1.1, -1.1, -1.1, -1.1, -1.1, -1.1, -1.1, -1.1,
506: (13)          -1.1, -1.1, -1.1, -1.1, -1.1,
507: (13)          0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
508: (13)          10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
509: (13)          20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
510: (13)          30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
511: (13)          40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
512: (13)          50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
513: (13)          60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
514: (13)          70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
515: (13)          80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
516: (13)          90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
517: (13)          -1.2, -1.2, -1.2, -1.2, -1.2, -1.2, -1.2, -1.2,
518: (13)          -1.2, -1.2, -1.2, -1.2, -1.2, -1.2, -1.2, -1.2]
519: (12)        )
520: (8)        assert_allclose(a, b)
521: (4)        def test_check_constant_odd_pad_amount(self):
522: (8)          arr = np.arange(30).reshape(5, 6)
523: (8)          test = np.pad(arr, ((1,), (2,)), mode='constant',
524: (19)            constant_values=3)
525: (8)        expected = np.array(
526: (12)          [[ 3,  3,  3,  3,  3,  3,  3,  3,  3,  3],
527: (13)          [ 3,  3,  0,  1,  2,  3,  4,  5,  3,  3],
528: (13)          [ 3,  3,  6,  7,  8,  9, 10, 11,  3,  3],
529: (13)          [ 3,  3, 12, 13, 14, 15, 16, 17,  3,  3],
530: (13)          [ 3,  3, 18, 19, 20, 21, 22, 23,  3,  3],
531: (13)          [ 3,  3, 24, 25, 26, 27, 28, 29,  3,  3],
532: (13)          [ 3,  3,  3,  3,  3,  3,  3,  3,  3,  3]
533: (12)        )
534: (8)        assert_allclose(test, expected)
535: (4)        def test_check_constant_pad_2d(self):
536: (8)          arr = np.arange(4).reshape(2, 2)
537: (8)          test = np.lib.pad(arr, ((1, 2), (1, 3)), mode='constant',
538: (26)            constant_values=((1, 2), (3, 4)))
539: (8)        expected = np.array(
540: (12)          [[3, 1, 1, 4, 4, 4],

```

```

541: (13) [3, 0, 1, 4, 4, 4],
542: (13) [3, 2, 3, 4, 4, 4],
543: (13) [3, 2, 2, 4, 4, 4],
544: (13) [3, 2, 2, 4, 4, 4]
545: (8) )
546: (8) assert_allclose(test, expected)
547: (4) def test_check_large_integers(self):
548: (8)     uint64_max = 2 ** 64 - 1
549: (8)     arr = np.full(5, uint64_max, dtype=np.uint64)
550: (8)     test = np.pad(arr, 1, mode="constant", constant_values=arr.min())
551: (8)     expected = np.full(7, uint64_max, dtype=np.uint64)
552: (8)     assert_array_equal(test, expected)
553: (8)     int64_max = 2 ** 63 - 1
554: (8)     arr = np.full(5, int64_max, dtype=np.int64)
555: (8)     test = np.pad(arr, 1, mode="constant", constant_values=arr.min())
556: (8)     expected = np.full(7, int64_max, dtype=np.int64)
557: (8)     assert_array_equal(test, expected)
558: (4) def test_check_object_array(self):
559: (8)     arr = np.empty(1, dtype=object)
560: (8)     obj_a = object()
561: (8)     arr[0] = obj_a
562: (8)     obj_b = object()
563: (8)     obj_c = object()
564: (8)     arr = np.pad(arr, pad_width=1, mode='constant',
565: (21)         constant_values=(obj_b, obj_c))
566: (8)     expected = np.empty((3,), dtype=object)
567: (8)     expected[0] = obj_b
568: (8)     expected[1] = obj_a
569: (8)     expected[2] = obj_c
570: (8)     assert_array_equal(arr, expected)
571: (4) def test_pad_empty_dimension(self):
572: (8)     arr = np.zeros((3, 0, 2))
573: (8)     result = np.pad(arr, [(0,), (2,), (1,)], mode="constant")
574: (8)     assert result.shape == (3, 4, 4)
575: (0) class TestLinearRamp:
576: (4)     def test_check_simple(self):
577: (8)         a = np.arange(100).astype('f')
578: (8)         a = np.pad(a, (25, 20), 'linear_ramp', end_values=(4, 5))
579: (8)         b = np.array(
580: (12)             [4.00, 3.84, 3.68, 3.52, 3.36, 3.20, 3.04, 2.88, 2.72, 2.56,
581: (13)                 2.40, 2.24, 2.08, 1.92, 1.76, 1.60, 1.44, 1.28, 1.12, 0.96,
582: (13)                 0.80, 0.64, 0.48, 0.32, 0.16,
583: (13)                 0.00, 1.00, 2.00, 3.00, 4.00, 5.00, 6.00, 7.00, 8.00, 9.00,
584: (13)                 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0,
585: (13)                 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0,
586: (13)                 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0,
587: (13)                 40.0, 41.0, 42.0, 43.0, 44.0, 45.0, 46.0, 47.0, 48.0, 49.0,
588: (13)                 50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0, 57.0, 58.0, 59.0,
589: (13)                 60.0, 61.0, 62.0, 63.0, 64.0, 65.0, 66.0, 67.0, 68.0, 69.0,
590: (13)                 70.0, 71.0, 72.0, 73.0, 74.0, 75.0, 76.0, 77.0, 78.0, 79.0,
591: (13)                 80.0, 81.0, 82.0, 83.0, 84.0, 85.0, 86.0, 87.0, 88.0, 89.0,
592: (13)                 90.0, 91.0, 92.0, 93.0, 94.0, 95.0, 96.0, 97.0, 98.0, 99.0,
593: (13)                 94.3, 89.6, 84.9, 80.2, 75.5, 70.8, 66.1, 61.4, 56.7, 52.0,
594: (13)                 47.3, 42.6, 37.9, 33.2, 28.5, 23.8, 19.1, 14.4, 9.7, 5.]
595: (12)             )
596: (8)         assert_allclose(a, b, rtol=1e-5, atol=1e-5)
597: (4)     def test_check_2d(self):
598: (8)         arr = np.arange(20).reshape(4, 5).astype(np.float64)
599: (8)         test = np.pad(arr, (2, 2), mode='linear_ramp', end_values=(0, 0))
600: (8)         expected = np.array(
601: (12)             [[0., 0., 0., 0., 0., 0., 0., 0., 0.],
602: (13)                 [0., 0., 0., 0.5, 1., 1.5, 2., 1., 0.],
603: (13)                 [0., 0., 0., 1., 2., 3., 4., 2., 0.],
604: (13)                 [0., 2.5, 5., 6., 7., 8., 9., 4.5, 0.],
605: (13)                 [0., 5., 10., 11., 12., 13., 14., 7., 0.],
606: (13)                 [0., 7.5, 15., 16., 17., 18., 19., 9.5, 0.],
607: (13)                 [0., 3.75, 7.5, 8., 8.5, 9., 9.5, 4.75, 0.],
608: (13)                 [0., 0., 0., 0., 0., 0., 0., 0., 0.]])
609: (8)         assert_allclose(test, expected)

```

```

610: (4) @pytest.mark.xfail(exceptions=(AssertionError,))
611: (4) def test_object_array(self):
612: (8)     from fractions import Fraction
613: (8)     arr = np.array([Fraction(1, 2), Fraction(-1, 2)])
614: (8)     actual = np.pad(arr, (2, 3), mode='linear_ramp', end_values=0)
615: (8)     expected = np.array([
616: (12)         Fraction( 0, 12),
617: (12)         Fraction( 3, 12),
618: (12)         Fraction( 6, 12),
619: (12)         Fraction(-6, 12),
620: (12)         Fraction(-4, 12),
621: (12)         Fraction(-2, 12),
622: (12)         Fraction(-0, 12),
623: (8)     ])
624: (8)     assert_equal(actual, expected)
625: (4) def test_end_values(self):
626: (8)     """Ensure that end values are exact."""
627: (8)     a = np.pad(np.ones(10).reshape(2, 5), (223, 123), mode="linear_ramp")
628: (8)     assert_equal(a[:, 0], 0.)
629: (8)     assert_equal(a[:, -1], 0.)
630: (8)     assert_equal(a[0, :], 0.)
631: (8)     assert_equal(a[-1, :], 0.)
632: (4) @pytest.mark.parametrize("dtype", _numeric_dtypes)
633: (4) def test_negative_difference(self, dtype):
634: (8)     """
635: (8)     Check correct behavior of unsigned dtypes if there is a negative
636: (8)     difference between the edge to pad and `end_values`. Check both cases
637: (8)     to be independent of implementation. Test behavior for all other
638: (8)     dtypes
639: (8)     in case dtype casting interferes with complex dtypes. See gh-14191.
640: (8)     """
641: (8)     x = np.array([3], dtype=dtype)
642: (8)     result = np.pad(x, 3, mode="linear_ramp", end_values=0)
643: (8)     expected = np.array([0, 1, 2, 3, 2, 1, 0], dtype=dtype)
644: (8)     assert_equal(result, expected)
645: (8)     x = np.array([0], dtype=dtype)
646: (8)     result = np.pad(x, 3, mode="linear_ramp", end_values=3)
647: (8)     expected = np.array([3, 2, 1, 0, 1, 2, 3], dtype=dtype)
648: (8)     assert_equal(result, expected)
649: (0) class TestReflect:
650: (4)     def test_check_simple(self):
651: (8)         a = np.arange(100)
652: (8)         a = np.pad(a, (25, 20), 'reflect')
653: (12)         b = np.array(
654: (13)             [25, 24, 23, 22, 21, 20, 19, 18, 17, 16,
655: (13)                 15, 14, 13, 12, 11, 10, 9, 8, 7, 6,
656: (13)                 5, 4, 3, 2, 1,
657: (13)                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
658: (13)                 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
659: (13)                 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
660: (13)                 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
661: (13)                 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
662: (13)                 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
663: (13)                 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
664: (13)                 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
665: (13)                 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
666: (13)                 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
667: (13)                 98, 97, 96, 95, 94, 93, 92, 91, 90, 89,
668: (12)                 88, 87, 86, 85, 84, 83, 82, 81, 80, 79]
669: (8)         )
670: (4)         assert_array_equal(a, b)
671: (8)     def test_check_odd_method(self):
672: (8)         a = np.arange(100)
673: (8)         a = np.pad(a, (25, 20), 'reflect', reflect_type='odd')
674: (12)         b = np.array(
675: (13)             [-25, -24, -23, -22, -21, -20, -19, -18, -17, -16,
676: (13)                 -15, -14, -13, -12, -11, -10, -9, -8, -7, -6,
677: (13)                 -5, -4, -3, -2, -1,
678: (13)                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

```

```

678: (13)          10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
679: (13)          20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
680: (13)          30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
681: (13)          40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
682: (13)          50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
683: (13)          60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
684: (13)          70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
685: (13)          80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
686: (13)          90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
687: (13)          100, 101, 102, 103, 104, 105, 106, 107, 108, 109,
688: (13)          110, 111, 112, 113, 114, 115, 116, 117, 118, 119]
689: (12)
690: (8)           )
691: (4)           assert_array_equal(a, b)
def test_check_large_pad(self):
692: (8)           a = [[4, 5, 6], [6, 7, 8]]
693: (8)           a = np.pad(a, (5, 7), 'reflect')
694: (8)           b = np.array(
695: (12)             [[7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7],
696: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
697: (13)               [7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7],
698: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
699: (13)               [7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7],
700: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
701: (13)               [7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7],
702: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
703: (13)               [7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7],
704: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
705: (13)               [7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7],
706: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
707: (13)               [7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7, 8, 7, 6, 7],
708: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5]]
709: (12)
710: (8)           )
711: (4)           assert_array_equal(a, b)
def test_check_shape(self):
712: (8)           a = [[4, 5, 6]]
713: (8)           a = np.pad(a, (5, 7), 'reflect')
714: (8)           b = np.array(
715: (12)             [[5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
716: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
717: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
718: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
719: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
720: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
721: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
722: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
723: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
724: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
725: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
726: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5],
727: (13)               [5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5, 6, 5, 4, 5]]
728: (12)
729: (8)           )
730: (4)           assert_array_equal(a, b)
def test_check_01(self):
731: (8)           a = np.pad([1, 2, 3], 2, 'reflect')
732: (8)           b = np.array([3, 2, 1, 2, 3, 2, 1])
733: (8)           assert_array_equal(a, b)
def test_check_02(self):
735: (8)           a = np.pad([1, 2, 3], 3, 'reflect')
736: (8)           b = np.array([2, 3, 2, 1, 2, 3, 2, 1, 2])
737: (8)           assert_array_equal(a, b)
def test_check_03(self):
739: (8)           a = np.pad([1, 2, 3], 4, 'reflect')
740: (8)           b = np.array([1, 2, 3, 2, 1, 2, 3, 2, 1, 2, 3])
741: (8)           assert_array_equal(a, b)
742: (0)           class TestEmptyArray:
743: (4)             """Check how padding behaves on arrays with an empty dimension."""
744: (4)             @pytest.mark.parametrize(
745: (8)               "mode", sorted(_all_modes.keys() - {"constant", "empty"})
746: (4)             )

```

```

747: (4)     def test_pad_empty_dimension(self, mode):
748: (8)         match = ("can't extend empty axis 0 using modes other than 'constant'
749: (17)                 "or 'empty'")
750: (8)         with pytest.raises(ValueError, match=match):
751: (12)             np.pad([], 4, mode=mode)
752: (8)         with pytest.raises(ValueError, match=match):
753: (12)             np.pad(np.ndarray(0), 4, mode=mode)
754: (8)         with pytest.raises(ValueError, match=match):
755: (12)             np.pad(np.zeros((0, 3)), ((1,), (0,)), mode=mode)
756: (4)     @pytest.mark.parametrize("mode", _all_modes.keys())
757: (4)     def test_pad_non_empty_dimension(self, mode):
758: (8)         result = np.pad(np.ones((2, 0, 2)), ((3,), (0,), (1,)), mode=mode)
759: (8)         assert result.shape == (8, 0, 4)
760: (0) class TestSymmetric:
761: (4)     def test_check_simple(self):
762: (8)         a = np.arange(100)
763: (8)         a = np.pad(a, (25, 20), 'symmetric')
764: (8)         b = np.array(
765: (12)             [24, 23, 22, 21, 20, 19, 18, 17, 16, 15,
766: (13)                 14, 13, 12, 11, 10, 9, 8, 7, 6, 5,
767: (13)                 4, 3, 2, 1, 0,
768: (13)                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
769: (13)                 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
770: (13)                 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
771: (13)                 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
772: (13)                 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
773: (13)                 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
774: (13)                 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
775: (13)                 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
776: (13)                 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
777: (13)                 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
778: (13)                 99, 98, 97, 96, 95, 94, 93, 92, 91, 90,
779: (13)                 89, 88, 87, 86, 85, 84, 83, 82, 81, 80]
780: (12)             )
781: (8)         assert_array_equal(a, b)
782: (4)     def test_check_odd_method(self):
783: (8)         a = np.arange(100)
784: (8)         a = np.pad(a, (25, 20), 'symmetric', reflect_type='odd')
785: (8)         b = np.array(
786: (12)             [-24, -23, -22, -21, -20, -19, -18, -17, -16, -15,
787: (13)                 -14, -13, -12, -11, -10, -9, -8, -7, -6, -5,
788: (13)                 -4, -3, -2, -1, 0,
789: (13)                 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
790: (13)                 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
791: (13)                 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
792: (13)                 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
793: (13)                 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
794: (13)                 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
795: (13)                 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
796: (13)                 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
797: (13)                 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
798: (13)                 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
799: (13)                 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
800: (13)                 109, 110, 111, 112, 113, 114, 115, 116, 117, 118]
801: (12)             )
802: (8)         assert_array_equal(a, b)
803: (4)     def test_check_large_pad(self):
804: (8)         a = [[4, 5, 6], [6, 7, 8]]
805: (8)         a = np.pad(a, (5, 7), 'symmetric')
806: (8)         b = np.array(
807: (12)             [[5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
808: (13)                 [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
809: (13)                 [7, 8, 8, 7, 6, 6, 7, 8, 8, 7, 6, 6, 7, 8, 8],
810: (13)                 [7, 8, 8, 7, 6, 6, 7, 8, 8, 7, 6, 6, 7, 8, 8],
811: (13)                 [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
812: (13)                 [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
813: (13)                 [7, 8, 8, 7, 6, 6, 7, 8, 8, 7, 6, 6, 7, 8, 8],
814: (13)                 [7, 8, 8, 7, 6, 6, 7, 8, 8, 7, 6, 6, 7, 8, 8],

```

```

815: (13) [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
816: (13) [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
817: (13) [7, 8, 8, 7, 6, 6, 7, 8, 8, 7, 6, 6, 7, 8, 8],
818: (13) [7, 8, 8, 7, 6, 6, 7, 8, 8, 7, 6, 6, 7, 8, 8],
819: (13) [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
820: (13) [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6]]
821: (12) )
822: (8) assert_array_equal(a, b)
823: (4) def test_check_large_pad_odd(self):
824: (8)     a = [[4, 5, 6], [6, 7, 8]]
825: (8)     a = np.pad(a, (5, 7), 'symmetric', reflect_type='odd')
826: (8)     b = np.array(
827: (12)         [[-3, -2, -2, -1, 0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6],
828: (13)         [-3, -2, -2, -1, 0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6],
829: (13)         [-1, 0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8],
830: (13)         [-1, 0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8],
831: (13)         [1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8, 9, 10, 10],
832: (13)         [1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8, 9, 10, 10],
833: (13)         [3, 4, 4, 5, 6, 6, 7, 8, 8, 9, 10, 10, 11, 12, 12],
834: (13)         [3, 4, 4, 5, 6, 6, 7, 8, 8, 9, 10, 10, 11, 12, 12],
835: (13)         [5, 6, 6, 7, 8, 8, 9, 10, 10, 11, 12, 12, 13, 14, 14],
836: (13)         [5, 6, 6, 7, 8, 8, 9, 10, 10, 11, 12, 12, 13, 14, 14],
837: (13)         [7, 8, 8, 9, 10, 10, 11, 12, 12, 13, 14, 14, 15, 16, 16],
838: (13)         [7, 8, 8, 9, 10, 10, 11, 12, 12, 13, 14, 14, 15, 16, 16],
839: (13)         [9, 10, 10, 11, 12, 12, 13, 14, 14, 15, 16, 16, 17, 18, 18],
840: (13)         [9, 10, 10, 11, 12, 12, 13, 14, 14, 15, 16, 16, 17, 18, 18]])
841: (12) )
842: (8) assert_array_equal(a, b)
843: (4) def test_check_shape(self):
844: (8)     a = [[4, 5, 6]]
845: (8)     a = np.pad(a, (5, 7), 'symmetric')
846: (8)     b = np.array(
847: (12)         [[5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
848: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
849: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
850: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
851: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
852: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
853: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
854: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
855: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
856: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
857: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
858: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6],
859: (13)         [5, 6, 6, 5, 4, 4, 5, 6, 6, 5, 4, 4, 5, 6, 6]])
860: (12) )
861: (8) assert_array_equal(a, b)
862: (4) def test_check_01(self):
863: (8)     a = np.pad([1, 2, 3], 2, 'symmetric')
864: (8)     b = np.array([2, 1, 1, 2, 3, 3, 2])
865: (8)     assert_array_equal(a, b)
866: (4) def test_check_02(self):
867: (8)     a = np.pad([1, 2, 3], 3, 'symmetric')
868: (8)     b = np.array([3, 2, 1, 1, 2, 3, 3, 2, 1])
869: (8)     assert_array_equal(a, b)
870: (4) def test_check_03(self):
871: (8)     a = np.pad([1, 2, 3], 6, 'symmetric')
872: (8)     b = np.array([1, 2, 3, 3, 2, 1, 1, 2, 3, 3, 2, 1, 1, 2, 3])
873: (8)     assert_array_equal(a, b)
874: (0) class TestWrap:
875: (4)     def test_check_simple(self):
876: (8)         a = np.arange(100)
877: (8)         a = np.pad(a, (25, 20), 'wrap')
878: (8)         b = np.array(
879: (12)             [75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
880: (13)             85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
881: (13)             95, 96, 97, 98, 99,
882: (13)             0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
883: (13)             10, 11, 12, 13, 14, 15, 16, 17, 18, 19,

```

```

884: (13)           20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
885: (13)           30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
886: (13)           40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
887: (13)           50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
888: (13)           60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
889: (13)           70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
890: (13)           80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
891: (13)           90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
892: (13)           0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
893: (13)           10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
894: (12)          )
895: (8)           assert_array_equal(a, b)
def test_check_large_pad(self):
897: (8)           a = np.arange(12)
898: (8)           a = np.reshape(a, (3, 4))
899: (8)           a = np.pad(a, (10, 12), 'wrap')
900: (8)           b = np.array(
901: (12)           [[10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10,
902: (14)             11, 8, 9, 10, 11, 8, 9, 10, 11],
903: (13)             [2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2,
904: (14)               3, 0, 1, 2, 3, 0, 1, 2, 3],
905: (13)             [6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6,
906: (14)               7, 4, 5, 6, 7, 4, 5, 6, 7],
907: (13)             [10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11,
908: (14)               8, 9, 10, 11, 8, 9, 10, 11],
909: (13)             [2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2,
910: (14)               3, 0, 1, 2, 3, 0, 1, 2, 3],
911: (13)             [6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6,
912: (14)               7, 4, 5, 6, 7, 4, 5, 6, 7],
913: (13)             [10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11,
914: (14)               11, 8, 9, 10, 11, 8, 9, 10, 11],
915: (13)             [2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2,
916: (14)               3, 0, 1, 2, 3, 0, 1, 2, 3],
917: (13)             [6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6,
918: (14)               7, 4, 5, 6, 7, 4, 5, 6, 7],
919: (13)             [10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11,
920: (14)               11, 8, 9, 10, 11, 8, 9, 10, 11],
921: (13)             [2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2,
922: (14)               3, 0, 1, 2, 3, 0, 1, 2, 3],
923: (13)             [6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6,
924: (14)               7, 4, 5, 6, 7, 4, 5, 6, 7],
925: (13)             [10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11,
926: (14)               11, 8, 9, 10, 11, 8, 9, 10, 11],
927: (13)             [2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2,
928: (14)               3, 0, 1, 2, 3, 0, 1, 2, 3],
929: (13)             [6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6,
930: (14)               7, 4, 5, 6, 7, 4, 5, 6, 7],
931: (13)             [10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11,
932: (14)               11, 8, 9, 10, 11, 8, 9, 10, 11],
933: (13)             [2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2,
934: (14)               3, 0, 1, 2, 3, 0, 1, 2, 3],
935: (13)             [6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6,
936: (14)               7, 4, 5, 6, 7, 4, 5, 6, 7],
937: (13)             [10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11,
938: (14)               11, 8, 9, 10, 11, 8, 9, 10, 11],
939: (13)             [2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2,
940: (14)               3, 0, 1, 2, 3, 0, 1, 2, 3],
941: (13)             [6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6,
942: (14)               7, 4, 5, 6, 7, 4, 5, 6, 7],
943: (13)             [10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11,
944: (14)               11, 8, 9, 10, 11, 8, 9, 10, 11],
945: (13)             [2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2,
946: (14)               3, 0, 1, 2, 3, 0, 1, 2, 3],
947: (13)             [6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6,
948: (14)               7, 4, 5, 6, 7, 4, 5, 6, 7],
949: (13)             [10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11,
950: (14)               11, 8, 9, 10, 11, 8, 9, 10, 11],
951: (12)          )
952: (8)           assert_array_equal(a, b)

```

```

953: (4) def test_check_01(self):
954: (8)     a = np.pad([1, 2, 3], 3, 'wrap')
955: (8)     b = np.array([1, 2, 3, 1, 2, 3, 1, 2, 3])
956: (8)     assert_array_equal(a, b)
957: (4) def test_check_02(self):
958: (8)     a = np.pad([1, 2, 3], 4, 'wrap')
959: (8)     b = np.array([3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1])
960: (8)     assert_array_equal(a, b)
961: (4) def test_pad_with_zero(self):
962: (8)     a = np.ones((3, 5))
963: (8)     b = np.pad(a, (0, 5), mode="wrap")
964: (8)     assert_array_equal(a, b[:-5, :-5])
965: (4) def test_repeated_wrapping(self):
966: (8)     """
967: (8)         Check wrapping on each side individually if the wrapped area is longer
968: (8)         than the original array.
969: (8)     """
970: (8)     a = np.arange(5)
971: (8)     b = np.pad(a, (12, 0), mode="wrap")
972: (8)     assert_array_equal(np.r_[a, a, a, a][3:], b)
973: (8)     a = np.arange(5)
974: (8)     b = np.pad(a, (0, 12), mode="wrap")
975: (8)     assert_array_equal(np.r_[a, a, a, a][-3:], b)
976: (4) def test_repeated_wrapping_multiple_origin(self):
977: (8)     """
978: (8)         Assert that 'wrap' pads only with multiples of the original area if
979: (8)         the pad width is larger than the original array.
980: (8)     """
981: (8)     a = np.arange(4).reshape(2, 2)
982: (8)     a = np.pad(a, [(1, 3), (3, 1)], mode='wrap')
983: (8)     b = np.array(
984: (12)         [[3, 2, 3, 2, 3, 2],
985: (13)             [1, 0, 1, 0, 1, 0],
986: (13)             [3, 2, 3, 2, 3, 2],
987: (13)             [1, 0, 1, 0, 1, 0],
988: (13)             [3, 2, 3, 2, 3, 2],
989: (13)             [1, 0, 1, 0, 1, 0]]
990: (8)     )
991: (8)     assert_array_equal(a, b)
992: (0) class TestEdge:
993: (4)     def test_check_simple(self):
994: (8)         a = np.arange(12)
995: (8)         a = np.reshape(a, (4, 3))
996: (8)         a = np.pad(a, ((2, 3), (3, 2)), 'edge')
997: (8)         b = np.array(
998: (12)             [[0, 0, 0, 0, 1, 2, 2, 2],
999: (13)                 [0, 0, 0, 0, 1, 2, 2, 2],
1000: (13)                 [0, 0, 0, 0, 1, 2, 2, 2],
1001: (13)                 [3, 3, 3, 3, 4, 5, 5, 5],
1002: (13)                 [6, 6, 6, 6, 7, 8, 8, 8],
1003: (13)                 [9, 9, 9, 9, 10, 11, 11, 11],
1004: (13)                 [9, 9, 9, 9, 10, 11, 11, 11],
1005: (13)                 [9, 9, 9, 9, 10, 11, 11, 11],
1006: (13)                 [9, 9, 9, 9, 10, 11, 11, 11]]
1007: (12)             )
1008: (8)         assert_array_equal(a, b)
1009: (4)     def test_check_width_shape_1_2(self):
1010: (8)         a = np.array([1, 2, 3])
1011: (8)         padded = np.pad(a, ((1, 2),), 'edge')
1012: (8)         expected = np.array([1, 1, 2, 3, 3, 3])
1013: (8)         assert_array_equal(padded, expected)
1014: (8)         a = np.array([[1, 2, 3], [4, 5, 6]])
1015: (8)         padded = np.pad(a, ((1, 2),), 'edge')
1016: (8)         expected = np.pad(a, ((1, 2), (1, 2)), 'edge')
1017: (8)         assert_array_equal(padded, expected)
1018: (8)         a = np.arange(24).reshape(2, 3, 4)
1019: (8)         padded = np.pad(a, ((1, 2),), 'edge')
1020: (8)         expected = np.pad(a, ((1, 2), (1, 2), (1, 2)), 'edge')
1021: (8)         assert_array_equal(padded, expected)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1022: (0)
1023: (4)
1024: (8)
1025: (8)
1026: (8)
1027: (8)
1028: (4)
1029: (8)
1030: (8)
1031: (8)
1032: (0)
1033: (4)
1034: (8)
1035: (8)
1036: (4)
1037: (4)
1038: (4)
1039: (8)
1040: (9)
1041: (9)
1042: (9)
1043: (9)
1044: (9)
1045: (8)
1046: (4)
1047: (0)
1048: (4)
1049: (4)
1050: (4)
1051: (0)
1052: (0)
1053: (4)
1054: (4)
1055: (4)
1056: (4)
1057: (0)
1058: (4)
1059: (8)
1060: (8)
1061: (8)
1062: (8)
1063: (4)
1064: (4)
1065: (4)
1066: (8)
1067: (8)
1068: (8)
1069: (12)
1070: (4)
1071: (4)
1072: (8)
1073: (8)
1074: (17)
1075: (8)
1076: (12)
1077: (4)
1078: (8)
1079: (4)
1080: (4)
1081: (8)
1082: (8)
1083: (8)
1084: (12)
1085: (4)
1086: (8)
1087: (8)
1088: (8)
1089: (8)
1090: (8)

class TestEmpty:
    def test_simple(self):
        arr = np.arange(24).reshape(4, 6)
        result = np.pad(arr, [(2, 3), (3, 1)], mode="empty")
        assert result.shape == (9, 10)
        assert_equal(arr, result[2:-3, 3:-1])
    def test_pad_empty_dimension(self):
        arr = np.zeros((3, 0, 2))
        result = np.pad(arr, [(0,), (2,), (1,)], mode="empty")
        assert result.shape == (3, 4, 4)
    def test_legacy_vector_functionality():
        def _padwithtens(vector, pad_width, iaxis, kwargs):
            vector[:pad_width[0]] = 10
            vector[-pad_width[1]:] = 10
        a = np.arange(6).reshape(2, 3)
        a = np.pad(a, 2, _padwithtens)
        b = np.array(
            [[10, 10, 10, 10, 10, 10, 10],
             [10, 10, 10, 10, 10, 10, 10],
             [10, 10, 0, 1, 2, 10, 10],
             [10, 10, 3, 4, 5, 10, 10],
             [10, 10, 10, 10, 10, 10, 10],
             [10, 10, 10, 10, 10, 10, 10]])
        assert_array_equal(a, b)
    def test_unicode_mode():
        a = np.pad([1], 2, mode='constant')
        b = np.array([0, 0, 1, 0, 0])
        assert_array_equal(a, b)
@pytest.mark.parametrize("mode", ["edge", "symmetric", "reflect", "wrap"])
def test_object_input(mode):
    a = np.full((4, 3), fill_value=None)
    pad_amt = ((2, 3), (3, 2))
    b = np.full((9, 8), fill_value=None)
    assert_array_equal(np.pad(a, pad_amt, mode=mode), b)
class TestPadWidth:
    @pytest.mark.parametrize("pad_width", [
        (4, 5, 6, 7),
        ((1,), (2,), (3,)),
        ((1, 2), (3, 4), (5, 6)),
        ((3, 4, 5), (0, 1, 2)),
    ])
    @pytest.mark.parametrize("mode", _all_modes.keys())
    def test_misshaped_pad_width(self, pad_width, mode):
        arr = np.arange(30).reshape((6, 5))
        match = "operands could not be broadcast together"
        with pytest.raises(ValueError, match=match):
            np.pad(arr, pad_width, mode)
    @pytest.mark.parametrize("mode", _all_modes.keys())
    def test_misshaped_pad_width_2(self, mode):
        arr = np.arange(30).reshape((6, 5))
        match = ("input operand has more dimensions than allowed by the axis "
                 "remapping")
        with pytest.raises(ValueError, match=match):
            np.pad(arr, (((3,), (4,), (5,)), ((0,), (1,), (2,))), mode)
    @pytest.mark.parametrize(
        "pad_width", [-2, (-2,), (3, -1), ((5, 2), (-2, 3)), ((-4,), (2,))])
    @pytest.mark.parametrize("mode", _all_modes.keys())
    def test_negative_pad_width(self, pad_width, mode):
        arr = np.arange(30).reshape((6, 5))
        match = "index can't contain negative values"
        with pytest.raises(ValueError, match=match):
            np.pad(arr, pad_width, mode)
    @pytest.mark.parametrize("pad_width, dtype", [
        ("3", None),
        ("word", None),
        (None, None),
        (object(), None),
        (3.4, None),
    ])

```

```

1091: (8)             (((2, 3, 4), (3, 2)), object),
1092: (8)             (complex(1, -1), None),
1093: (8)             (((-2.1, 3), (3, 2)), None),
1094: (4)         ])
1095: (4)     @pytest.mark.parametrize("mode", _all_modes.keys())
1096: (4)     def test_bad_type(self, pad_width, dtype, mode):
1097: (8)         arr = np.arange(30).reshape((6, 5))
1098: (8)         match = "`pad_width` must be of integral type."
1099: (8)         if dtype is not None:
1100: (12)             with pytest.raises(TypeError, match=match):
1101: (16)                 np.pad(arr, np.array(pad_width, dtype=dtype), mode)
1102: (8)         else:
1103: (12)             with pytest.raises(TypeError, match=match):
1104: (16)                 np.pad(arr, pad_width, mode)
1105: (12)             with pytest.raises(TypeError, match=match):
1106: (16)                 np.pad(arr, np.array(pad_width), mode)
1107: (4)     def test_pad_width_as_ndarray(self):
1108: (8)         a = np.arange(12)
1109: (8)         a = np.reshape(a, (4, 3))
1110: (8)         a = np.pad(a, np.array(((2, 3), (3, 2))), 'edge')
1111: (8)         b = np.array(
1112: (12)             [[0, 0, 0, 0, 1, 2, 2, 2],
1113: (13)             [0, 0, 0, 0, 1, 2, 2, 2],
1114: (13)             [0, 0, 0, 0, 1, 2, 2, 2],
1115: (13)             [3, 3, 3, 3, 4, 5, 5, 5],
1116: (13)             [6, 6, 6, 6, 7, 8, 8, 8],
1117: (13)             [9, 9, 9, 9, 10, 11, 11, 11],
1118: (13)             [9, 9, 9, 9, 10, 11, 11, 11],
1119: (13)             [9, 9, 9, 9, 10, 11, 11, 11],
1120: (13)             [9, 9, 9, 9, 10, 11, 11, 11]])
1121: (12)         )
1122: (8)         assert_array_equal(a, b)
1123: (4)     @pytest.mark.parametrize("pad_width", [0, (0, 0), ((0, 0), (0, 0))])
1124: (4)     @pytest.mark.parametrize("mode", _all_modes.keys())
1125: (4)     def test_zero_pad_width(self, pad_width, mode):
1126: (8)         arr = np.arange(30).reshape(6, 5)
1127: (8)         assert_array_equal(arr, np.pad(arr, pad_width, mode=mode))
1128: (0)     @pytest.mark.parametrize("mode", _all_modes.keys())
1129: (0)     def test_kwargs(mode):
1130: (4)         """Test behavior of pad's kwargs for the given mode."""
1131: (4)         allowed = _all_modes[mode]
1132: (4)         not_allowed = {}
1133: (4)         for kwargs in _all_modes.values():
1134: (8)             if kwargs != allowed:
1135: (12)                 not_allowed.update(kwargs)
1136: (4)         np.pad([1, 2, 3], 1, mode, **allowed)
1137: (4)         for key, value in not_allowed.items():
1138: (8)             match = "unsupported keyword arguments for mode '{}'.format(mode)"
1139: (8)             with pytest.raises(ValueError, match=match):
1140: (12)                 np.pad([1, 2, 3], 1, mode, **{key: value})
1141: (0)     def test_constant_zero_default():
1142: (4)         arr = np.array([1, 1])
1143: (4)         assert_array_equal(np.pad(arr, 2), [0, 0, 1, 1, 0, 0])
1144: (0)     @pytest.mark.parametrize("mode", [1, "const", object(), None, True, False])
1145: (0)     def test_unsupported_mode(mode):
1146: (4)         match = "mode '{}' is not supported".format(mode)
1147: (4)         with pytest.raises(ValueError, match=match):
1148: (8)             np.pad([1, 2, 3], 4, mode=mode)
1149: (0)     @pytest.mark.parametrize("mode", _all_modes.keys())
1150: (0)     def test_non_contiguous_array(mode):
1151: (4)         arr = np.arange(24).reshape(4, 6)[::2, ::2]
1152: (4)         result = np.pad(arr, (2, 3), mode)
1153: (4)         assert result.shape == (7, 8)
1154: (4)         assert_equal(result[2:-3, 2:-3], arr)
1155: (0)     @pytest.mark.parametrize("mode", _all_modes.keys())
1156: (0)     def test_memory_layout_persistence(mode):
1157: (4)         """Test if C and F order is preserved for all pad modes."""
1158: (4)         x = np.ones((5, 10), order='C')
1159: (4)         assert np.pad(x, 5, mode).flags["C_CONTIGUOUS"]

```

```

1160: (4)         x = np.ones((5, 10), order='F')
1161: (4)         assert np.pad(x, 5, mode).flags["F_CONTIGUOUS"]
1162: (0)         @pytest.mark.parametrize("dtype", _numeric_dtypes)
1163: (0)         @pytest.mark.parametrize("mode", _all_modes.keys())
1164: (0)         def test_dtype_persistence(dtype, mode):
1165: (4)             arr = np.zeros((3, 2, 1), dtype=dtype)
1166: (4)             result = np.pad(arr, 1, mode=mode)
1167: (4)             assert result.dtype == dtype

```

---

## File 230 - test\_arraysetops.py:

```

1: (0)         """Test functions for 1D array set operations.
2: (0)         """
3: (0)         import numpy as np
4: (0)         from numpy.testing import (assert_array_equal, assert_equal,
5: (27)                 assert_raises, assert_raises_regex)
6: (0)         from numpy.lib.arraysetops import (
7: (4)             ediff1d, intersect1d, setxor1d, union1d, setdiff1d, unique, in1d, isin
8: (4)                 )
9: (0)         import pytest
10: (0)         class TestSetOps:
11: (4)             def test_intersect1d(self):
12: (8)                 a = np.array([5, 7, 1, 2])
13: (8)                 b = np.array([2, 4, 3, 1, 5])
14: (8)                 ec = np.array([1, 2, 5])
15: (8)                 c = intersect1d(a, b, assume_unique=True)
16: (8)                 assert_array_equal(c, ec)
17: (8)                 a = np.array([5, 5, 7, 1, 2])
18: (8)                 b = np.array([2, 1, 4, 3, 3, 1, 5])
19: (8)                 ed = np.array([1, 2, 5])
20: (8)                 c = intersect1d(a, b)
21: (8)                 assert_array_equal(c, ed)
22: (8)                 assert_array_equal([], intersect1d([], []))
23: (4)             def test_intersect1d_array_like(self):
24: (8)                 class Test:
25: (12)                     def __array__(self):
26: (16)                         return np.arange(3)
27: (8)                     a = Test()
28: (8)                     res = intersect1d(a, a)
29: (8)                     assert_array_equal(res, a)
30: (8)                     res = intersect1d([1, 2, 3], [1, 2, 3])
31: (8)                     assert_array_equal(res, [1, 2, 3])
32: (4)             def test_intersect1d_indices(self):
33: (8)                 a = np.array([1, 2, 3, 4])
34: (8)                 b = np.array([2, 1, 4, 6])
35: (8)                 c, i1, i2 = intersect1d(a, b, assume_unique=True, return_indices=True)
36: (8)                 ee = np.array([1, 2, 4])
37: (8)                 assert_array_equal(c, ee)
38: (8)                 assert_array_equal(a[i1], ee)
39: (8)                 assert_array_equal(b[i2], ee)
40: (8)                 a = np.array([1, 2, 2, 3, 4, 3, 2])
41: (8)                 b = np.array([1, 8, 4, 2, 2, 3, 2, 3])
42: (8)                 c, i1, i2 = intersect1d(a, b, return_indices=True)
43: (8)                 ef = np.array([1, 2, 3, 4])
44: (8)                 assert_array_equal(c, ef)
45: (8)                 assert_array_equal(a[i1], ef)
46: (8)                 assert_array_equal(b[i2], ef)
47: (8)                 a = np.array([[2, 4, 5, 6], [7, 8, 1, 15]])
48: (8)                 b = np.array([[3, 2, 7, 6], [10, 12, 8, 9]])
49: (8)                 c, i1, i2 = intersect1d(a, b, assume_unique=True, return_indices=True)
50: (8)                 ui1 = np.unravel_index(i1, a.shape)
51: (8)                 ui2 = np.unravel_index(i2, b.shape)
52: (8)                 ea = np.array([2, 6, 7, 8])
53: (8)                 assert_array_equal(ea, a[ui1])
54: (8)                 assert_array_equal(ea, b[ui2])
55: (8)                 a = np.array([[2, 4, 5, 6, 6], [4, 7, 8, 7, 2]])
56: (8)                 b = np.array([[3, 2, 7, 7], [10, 12, 8, 7]])

```

```

57: (8)             c, i1, i2 = intersect1d(a, b, return_indices=True)
58: (8)             ui1 = np.unravel_index(i1, a.shape)
59: (8)             ui2 = np.unravel_index(i2, b.shape)
60: (8)             ea = np.array([2, 7, 8])
61: (8)             assert_array_equal(ea, a[ui1])
62: (8)             assert_array_equal(ea, b[ui2])
63: (4)             def test_setxor1d(self):
64: (8)                 a = np.array([5, 7, 1, 2])
65: (8)                 b = np.array([2, 4, 3, 1, 5])
66: (8)                 ec = np.array([3, 4, 7])
67: (8)                 c = setxor1d(a, b)
68: (8)                 assert_array_equal(c, ec)
69: (8)                 a = np.array([1, 2, 3])
70: (8)                 b = np.array([6, 5, 4])
71: (8)                 ec = np.array([1, 2, 3, 4, 5, 6])
72: (8)                 c = setxor1d(a, b)
73: (8)                 assert_array_equal(c, ec)
74: (8)                 a = np.array([1, 8, 2, 3])
75: (8)                 b = np.array([6, 5, 4, 8])
76: (8)                 ec = np.array([1, 2, 3, 4, 5, 6])
77: (8)                 c = setxor1d(a, b)
78: (8)                 assert_array_equal(c, ec)
79: (8)                 assert_array_equal([], setxor1d([], []))
80: (4)             def test_ediff1d(self):
81: (8)                 zero_elem = np.array([])
82: (8)                 one_elem = np.array([1])
83: (8)                 two_elem = np.array([1, 2])
84: (8)                 assert_array_equal([], ediff1d(zero_elem))
85: (8)                 assert_array_equal([0], ediff1d(zero_elem, to_begin=0))
86: (8)                 assert_array_equal([0], ediff1d(zero_elem, to_end=0))
87: (8)                 assert_array_equal([-1, 0], ediff1d(zero_elem, to_begin=-1, to_end=0))
88: (8)                 assert_array_equal([], ediff1d(one_elem))
89: (8)                 assert_array_equal([1], ediff1d(two_elem))
90: (8)                 assert_array_equal([7, 1, 9], ediff1d(two_elem, to_begin=7, to_end=9))
91: (8)                 assert_array_equal([5, 6, 1, 7, 8],
92: (27)                         ediff1d(two_elem, to_begin=[5, 6], to_end=[7, 8]))
93: (8)                 assert_array_equal([1, 9], ediff1d(two_elem, to_end=9))
94: (8)                 assert_array_equal([1, 7, 8], ediff1d(two_elem, to_end=[7, 8]))
95: (8)                 assert_array_equal([7, 1], ediff1d(two_elem, to_begin=7))
96: (8)                 assert_array_equal([5, 6, 1], ediff1d(two_elem, to_begin=[5, 6]))
97: (4)             @pytest.mark.parametrize("ary, prepend, append, expected", [
98: (8)                 (np.array([1, 2, 3], dtype=np.int64),
99: (9)                     None,
100: (9)                     np.nan,
101: (9)                     'to_end'),
102: (8)                 (np.array([1, 2, 3], dtype=np.int64),
103: (9)                     np.array([5, 7, 2], dtype=np.float32),
104: (9)                     None,
105: (9)                     'to_begin'),
106: (8)                 (np.array([1., 3., 9.], dtype=np.int8),
107: (9)                     np.nan,
108: (9)                     np.nan,
109: (9)                     'to_begin'),
110: (9)                     ])
111: (4)             def test_ediff1d_forbidden_type_casts(self, ary, prepend, append,
112: (8)             expected):
113: (8)                 msg = 'dtype of `{}` must be compatible'.format(expected)
114: (12)                 with assert_raises_regex(TypeError, msg):
115: (20)                     ediff1d(ary=ary,
116: (20)                         to_end=append,
117: (4)                         to_begin=prepend)
118: (8)                     @pytest.mark.parametrize(
119: (8)                         "ary,prepend,append,expected",
120: (9)                         [
121: (10)                             (np.array([1, 2, 3], dtype=np.int16),
122: (10)                                 2**16, # will be cast to int16 under same kind rule.
123: (10)                                 2**16 + 4,
124: (9)                                 np.array([0, 1, 1, 4], dtype=np.int16)),
124: (9)                             (np.array([1, 2, 3], dtype=np.float32),

```

```

125: (10)                      np.array([5], dtype=np.float64),
126: (10)                      None,
127: (10)                      np.array([5, 1, 1], dtype=np.float32)),
128: (9)                       (np.array([1, 2, 3], dtype=np.int32),
129: (10)                         0,
130: (10)                         0,
131: (10)                         np.array([0, 1, 1, 0], dtype=np.int32)),
132: (9)                          (np.array([1, 2, 3], dtype=np.int64),
133: (10)                            3,
134: (10)                            -9,
135: (10)                            np.array([3, 1, 1, -9], dtype=np.int64)),
136: (8)                           ]
137: (4)
138: (4)      def test_ediff1d_scalar_handling(self,
139: (37)                      ary,
140: (37)                      prepend,
141: (37)                      append,
142: (37)                      expected):
143: (8)                        actual = np.ediff1d(ary=ary,
144: (28)                          to_end=append,
145: (28)                          to_begin=prepend)
146: (8)                        assert_equal(actual, expected)
147: (8)                        assert actual.dtype == expected.dtype
148: (4) @pytest.mark.parametrize("kind", [None, "sort", "table"])
149: (4)      def test_isin(self, kind):
150: (8)        def _isin_slow(a, b):
151: (12)          b = np.asarray(b).flatten().tolist()
152: (12)          return a in b
153: (8)        isin_slow = np.vectorize(_isin_slow, otypes=[bool], excluded={1})
154: (8)      def assert_isin_equal(a, b):
155: (12)        x = isin(a, b, kind=kind)
156: (12)        y = isin_slow(a, b)
157: (12)        assert_array_equal(x, y)
158: (8)      a = np.arange(24).reshape([2, 3, 4])
159: (8)      b = np.array([[10, 20, 30], [0, 1, 3], [11, 22, 33]])
160: (8)      assert_isin_equal(a, b)
161: (8)      c = [(9, 8), (7, 6)]
162: (8)      d = (9, 7)
163: (8)      assert_isin_equal(c, d)
164: (8)      f = np.array(3)
165: (8)      assert_isin_equal(f, b)
166: (8)      assert_isin_equal(a, f)
167: (8)      assert_isin_equal(f, f)
168: (8)      assert_isin_equal(5, b)
169: (8)      assert_isin_equal(a, 6)
170: (8)      assert_isin_equal(5, 6)
171: (8)      if kind != "table":
172: (12)        x = []
173: (12)        assert_isin_equal(x, b)
174: (12)        assert_isin_equal(a, x)
175: (12)        assert_isin_equal(x, x)
176: (8)      for dtype in [bool, np.int64, np.float64]:
177: (12)        if kind == "table" and dtype == np.float64:
178: (16)          continue
179: (12)        if dtype in {np.int64, np.float64}:
180: (16)          ar = np.array([10, 20, 30], dtype=dtype)
181: (12)        elif dtype in {bool}:
182: (16)          ar = np.array([True, False, False])
183: (12)          empty_array = np.array([], dtype=dtype)
184: (12)          assert_isin_equal(empty_array, ar)
185: (12)          assert_isin_equal(ar, empty_array)
186: (12)          assert_isin_equal(empty_array, empty_array)
187: (4) @pytest.mark.parametrize("kind", [None, "sort", "table"])
188: (4)      def test_in1d(self, kind):
189: (8)        for mult in (1, 10):
190: (12)          a = [5, 7, 1, 2]
191: (12)          b = [2, 4, 3, 1, 5] * mult
192: (12)          ec = np.array([True, False, True, True])
193: (12)          c = in1d(a, b, assume_unique=True, kind=kind)

```

```

194: (12) assert_array_equal(c, ec)
195: (12) a[0] = 8
196: (12) ec = np.array([False, False, True, True])
197: (12) c = in1d(a, b, assume_unique=True, kind=kind)
198: (12) assert_array_equal(c, ec)
199: (12) a[0], a[3] = 4, 8
200: (12) ec = np.array([True, False, True, False])
201: (12) c = in1d(a, b, assume_unique=True, kind=kind)
202: (12) assert_array_equal(c, ec)
203: (12) a = np.array([5, 4, 5, 3, 4, 4, 3, 4, 3, 5, 2, 1, 5, 5])
204: (12) b = [2, 3, 4] * mult
205: (12) ec = [False, True, False, True, True, True, True, True,
206: (18) False, True, False, False, False]
207: (12) c = in1d(a, b, kind=kind)
208: (12) assert_array_equal(c, ec)
209: (12) b = b + [5, 5, 4] * mult
210: (12) ec = [True, True, True, True, True, True, True, True, True,
211: (18) True, False, True, True]
212: (12) c = in1d(a, b, kind=kind)
213: (12) assert_array_equal(c, ec)
214: (12) a = np.array([5, 7, 1, 2])
215: (12) b = np.array([2, 4, 3, 1, 5] * mult)
216: (12) ec = np.array([True, False, True, True])
217: (12) c = in1d(a, b, kind=kind)
218: (12) assert_array_equal(c, ec)
219: (12) a = np.array([5, 7, 1, 1, 2])
220: (12) b = np.array([2, 4, 3, 3, 1, 5] * mult)
221: (12) ec = np.array([True, False, True, True, True])
222: (12) c = in1d(a, b, kind=kind)
223: (12) assert_array_equal(c, ec)
224: (12) a = np.array([5, 5])
225: (12) b = np.array([2, 2] * mult)
226: (12) ec = np.array([False, False])
227: (12) c = in1d(a, b, kind=kind)
228: (12) assert_array_equal(c, ec)
229: (8) a = np.array([5])
230: (8) b = np.array([2])
231: (8) ec = np.array([False])
232: (8) c = in1d(a, b, kind=kind)
233: (8) assert_array_equal(c, ec)
234: (8) if kind in {None, "sort"}:
235: (12)     assert_array_equal(in1d([], [], kind=kind), [])
236: (4) def test_in1d_char_array(self):
237: (8)     a = np.array(['a', 'b', 'c', 'd', 'e', 'c', 'e', 'b'])
238: (8)     b = np.array(['a', 'c'])
239: (8)     ec = np.array([True, False, True, False, False, True, False, False])
240: (8)     c = in1d(a, b)
241: (8)     assert_array_equal(c, ec)
242: (4) @pytest.mark.parametrize("kind", [None, "sort", "table"])
243: (4) def test_in1d_invert(self, kind):
244: (8)     "Test in1d's invert parameter"
245: (8)     for mult in (1, 10):
246: (12)         a = np.array([5, 4, 5, 3, 4, 4, 3, 4, 3, 5, 2, 1, 5, 5])
247: (12)         b = [2, 3, 4] * mult
248: (12)         assert_array_equal(np.invert(in1d(a, b, kind=kind)),
249: (31)                         in1d(a, b, invert=True, kind=kind))
250: (8)     if kind in {None, "sort"}:
251: (12)         for mult in (1, 10):
252: (16)             a = np.array([5, 4, 5, 3, 4, 4, 3, 4, 3, 5, 2, 1, 5, 5],
253: (28)                 dtype=np.float32)
254: (16)             b = [2, 3, 4] * mult
255: (16)             b = np.array(b, dtype=np.float32)
256: (16)             assert_array_equal(np.invert(in1d(a, b, kind=kind)),
257: (35)                 in1d(a, b, invert=True, kind=kind))
258: (4) @pytest.mark.parametrize("kind", [None, "sort", "table"])
259: (4) def test_in1d_ravel(self, kind):
260: (8)     a = np.arange(6).reshape(2, 3)
261: (8)     b = np.arange(3, 9).reshape(3, 2)
262: (8)     long_b = np.arange(3, 63).reshape(30, 2)

```

```

263: (8)           ec = np.array([False, False, False, True, True, True])
264: (8)           assert_array_equal(in1d(a, b, assume_unique=True, kind=kind),
265: (27)                     ec)
266: (8)           assert_array_equal(in1d(a, b, assume_unique=False,
267: (32)                     kind=kind),
268: (27)                     ec)
269: (8)           assert_array_equal(in1d(a, long_b, assume_unique=True,
270: (32)                     kind=kind),
271: (27)                     ec)
272: (8)           assert_array_equal(in1d(a, long_b, assume_unique=False,
273: (32)                     kind=kind),
274: (27)                     ec)
275: (4)           def test_in1d_hit_alternate_algorithm(self):
276: (8)             """Hit the standard isin code with integers"""
277: (8)             a = np.array([5, 4, 5, 3, 4, 4, 1e9], dtype=np.int64)
278: (8)             b = np.array([2, 3, 4, 1e9], dtype=np.int64)
279: (8)             expected = np.array([0, 1, 0, 1, 1, 1, 1], dtype=bool)
280: (8)             assert_array_equal(expected, in1d(a, b))
281: (8)             assert_array_equal(np.invert(expected), in1d(a, b, invert=True))
282: (8)             a = np.array([5, 7, 1, 2], dtype=np.int64)
283: (8)             b = np.array([2, 4, 3, 1, 5, 1e9], dtype=np.int64)
284: (8)             ec = np.array([True, False, True, True])
285: (8)             c = in1d(a, b, assume_unique=True)
286: (8)             assert_array_equal(c, ec)
287: (4)           @pytest.mark.parametrize("kind", [None, "sort", "table"])
288: (4)           def test_in1d_boolean(self, kind):
289: (8)             """Test that in1d works for boolean input"""
290: (8)             a = np.array([True, False])
291: (8)             b = np.array([False, False, False])
292: (8)             expected = np.array([False, True])
293: (8)             assert_array_equal(expected,
294: (27)                         in1d(a, b, kind=kind))
295: (8)             assert_array_equal(np.invert(expected),
296: (27)                         in1d(a, b, invert=True, kind=kind))
297: (4)           @pytest.mark.parametrize("kind", [None, "sort"])
298: (4)           def test_in1d_timedelta(self, kind):
299: (8)             """Test that in1d works for timedelta input"""
300: (8)             rstate = np.random.RandomState(0)
301: (8)             a = rstate.randint(0, 100, size=10)
302: (8)             b = rstate.randint(0, 100, size=10)
303: (8)             truth = in1d(a, b)
304: (8)             a_timedelta = a.astype("timedelta64[s]")
305: (8)             b_timedelta = b.astype("timedelta64[s]")
306: (8)             assert_array_equal(truth, in1d(a_timedelta, b_timedelta, kind=kind))
307: (4)           def test_in1d_table_timedelta_fails(self):
308: (8)             a = np.array([0, 1, 2], dtype="timedelta64[s]")
309: (8)             b = a
310: (8)             with pytest.raises(ValueError):
311: (12)               in1d(a, b, kind="table")
312: (4)           @pytest.mark.parametrize(
313: (8)             "dtype1,dtype2",
314: (8)             [
315: (12)               (np.int8, np.int16),
316: (12)               (np.int16, np.int8),
317: (12)               (np.uint8, np.uint16),
318: (12)               (np.uint16, np.uint8),
319: (12)               (np.uint8, np.int16),
320: (12)               (np.int16, np.uint8),
321: (8)             ]
322: (4)           )
323: (4)           @pytest.mark.parametrize("kind", [None, "sort", "table"])
324: (4)           def test_in1d_mixed_dtype(self, dtype1, dtype2, kind):
325: (8)             """Test that in1d works as expected for mixed dtype input."""
326: (8)             is_dtype2_signed = np.issubdtype(dtype2, np.signedinteger)
327: (8)             ar1 = np.array([0, 0, 1, 1], dtype=dtype1)
328: (8)             if is_dtype2_signed:
329: (12)               ar2 = np.array([-128, 0, 127], dtype=dtype2)
330: (8)             else:
331: (12)               ar2 = np.array([127, 0, 255], dtype=dtype2)

```

```

332: (8)           expected = np.array([True, True, False, False])
333: (8)           expect_failure = kind == "table" and any((
334: (12)             dtype1 == np.int8 and dtype2 == np.int16,
335: (12)             dtype1 == np.int16 and dtype2 == np.int8
336: (8)         ))
337: (8)         if expect_failure:
338: (12)             with pytest.raises(RuntimeError, match="exceed the maximum"):
339: (16)                 in1d(ar1, ar2, kind=kind)
340: (8)         else:
341: (12)             assert_array_equal(in1d(ar1, ar2, kind=kind), expected)
342: (4) @pytest.mark.parametrize("kind", [None, "sort", "table"])
343: (4) def test_in1d_mixed_boolean(self, kind):
344: (8)     """Test that in1d works as expected for bool/int input."""
345: (8)     for dtype in np.typecodes["AllInteger"]:
346: (12)         a = np.array([True, False, False], dtype=bool)
347: (12)         b = np.array([0, 0, 0, 0], dtype=dtype)
348: (12)         expected = np.array([False, True, True], dtype=bool)
349: (12)         assert_array_equal(in1d(a, b, kind=kind), expected)
350: (12)         a, b = b, a
351: (12)         expected = np.array([True, True, True, True], dtype=bool)
352: (12)         assert_array_equal(in1d(a, b, kind=kind), expected)
353: (4) def test_in1d_first_array_is_object(self):
354: (8)     ar1 = [None]
355: (8)     ar2 = np.array([1]*10)
356: (8)     expected = np.array([False])
357: (8)     result = np.in1d(ar1, ar2)
358: (8)     assert_array_equal(result, expected)
359: (4) def test_in1d_second_array_is_object(self):
360: (8)     ar1 = 1
361: (8)     ar2 = np.array([None]*10)
362: (8)     expected = np.array([False])
363: (8)     result = np.in1d(ar1, ar2)
364: (8)     assert_array_equal(result, expected)
365: (4) def test_in1d_both_arrays_are_object(self):
366: (8)     ar1 = [None]
367: (8)     ar2 = np.array([None]*10)
368: (8)     expected = np.array([True])
369: (8)     result = np.in1d(ar1, ar2)
370: (8)     assert_array_equal(result, expected)
371: (4) def test_in1d_both_arrays_have_structured_dtype(self):
372: (8)     dt = np.dtype([('field1', int), ('field2', object)])
373: (8)     ar1 = np.array([(1, None)], dtype=dt)
374: (8)     ar2 = np.array([(1, None)]*10, dtype=dt)
375: (8)     expected = np.array([True])
376: (8)     result = np.in1d(ar1, ar2)
377: (8)     assert_array_equal(result, expected)
378: (4) def test_in1d_with_arrays_containing_tuples(self):
379: (8)     ar1 = np.array([(1,), 2], dtype=object)
380: (8)     ar2 = np.array([(1,), 2], dtype=object)
381: (8)     expected = np.array([True, True])
382: (8)     result = np.in1d(ar1, ar2)
383: (8)     assert_array_equal(result, expected)
384: (8)     result = np.in1d(ar1, ar2, invert=True)
385: (8)     assert_array_equal(result, np.invert(expected))
386: (8)     ar1 = np.array([(1,), (2, 1), 1], dtype=object)
387: (8)     ar1 = ar1[:-1]
388: (8)     ar2 = np.array([(1,), (2, 1), 1], dtype=object)
389: (8)     ar2 = ar2[:-1]
390: (8)     expected = np.array([True, True])
391: (8)     result = np.in1d(ar1, ar2)
392: (8)     assert_array_equal(result, expected)
393: (8)     result = np.in1d(ar1, ar2, invert=True)
394: (8)     assert_array_equal(result, np.invert(expected))
395: (8)     ar1 = np.array([(1,), (2, 3), 1], dtype=object)
396: (8)     ar1 = ar1[:-1]
397: (8)     ar2 = np.array([(1,), 2], dtype=object)
398: (8)     expected = np.array([True, False])
399: (8)     result = np.in1d(ar1, ar2)
400: (8)     assert_array_equal(result, expected)

```

```

401: (8)             result = np.in1d(ar1, ar2, invert=True)
402: (8)             assert_array_equal(result, np.invert(expected))
403: (4)             def test_in1d_errors(self):
404: (8)                 """Test that in1d raises expected errors."""
405: (8)                 ar1 = np.array([1, 2, 3, 4, 5])
406: (8)                 ar2 = np.array([2, 4, 6, 8, 10])
407: (8)                 assert_raises(ValueError, in1d, ar1, ar2, kind='quicksort')
408: (8)                 obj_ar1 = np.array([1, 'a', 3, 'b', 5], dtype=object)
409: (8)                 obj_ar2 = np.array([1, 'a', 3, 'b', 5], dtype=object)
410: (8)                 assert_raises(ValueError, in1d, obj_ar1, obj_ar2, kind='table')
411: (8)             for dtype in [np.int32, np.int64]:
412: (12)                 ar1 = np.array([-1, 2, 3, 4, 5], dtype=dtype)
413: (12)                 overflow_ar2 = np.array([-1, np.iinfo(dtype).max], dtype=dtype)
414: (12)                 assert_raises(
415: (16)                     RuntimeError,
416: (16)                     in1d, ar1, overflow_ar2, kind='table'
417: (12)                 )
418: (12)                 result = np.in1d(ar1, overflow_ar2, kind=None)
419: (12)                 assert_array_equal(result, [True] + [False] * 4)
420: (12)                 result = np.in1d(ar1, overflow_ar2, kind='sort')
421: (12)                 assert_array_equal(result, [True] + [False] * 4)
422: (4)             def test_union1d(self):
423: (8)                 a = np.array([5, 4, 7, 1, 2])
424: (8)                 b = np.array([2, 4, 3, 3, 2, 1, 5])
425: (8)                 ec = np.array([1, 2, 3, 4, 5, 7])
426: (8)                 c = union1d(a, b)
427: (8)                 assert_array_equal(c, ec)
428: (8)                 x = np.array([[0, 1, 2], [3, 4, 5]])
429: (8)                 y = np.array([0, 1, 2, 3, 4])
430: (8)                 ez = np.array([0, 1, 2, 3, 4, 5])
431: (8)                 z = union1d(x, y)
432: (8)                 assert_array_equal(z, ez)
433: (8)                 assert_array_equal([], union1d([], []))
434: (4)             def test_setdiff1d(self):
435: (8)                 a = np.array([6, 5, 4, 7, 1, 2, 7, 4])
436: (8)                 b = np.array([2, 4, 3, 3, 2, 1, 5])
437: (8)                 ec = np.array([6, 7])
438: (8)                 c = setdiff1d(a, b)
439: (8)                 assert_array_equal(c, ec)
440: (8)                 a = np.arange(21)
441: (8)                 b = np.arange(19)
442: (8)                 ec = np.array([19, 20])
443: (8)                 c = setdiff1d(a, b)
444: (8)                 assert_array_equal(c, ec)
445: (8)                 assert_array_equal([], setdiff1d([], []))
446: (8)                 a = np.array(((), np.uint32))
447: (8)                 assert_equal(setdiff1d(a, []).dtype, np.uint32)
448: (4)             def test_setdiff1d_unique(self):
449: (8)                 a = np.array([3, 2, 1])
450: (8)                 b = np.array([7, 5, 2])
451: (8)                 expected = np.array([3, 1])
452: (8)                 actual = setdiff1d(a, b, assume_unique=True)
453: (8)                 assert_equal(actual, expected)
454: (4)             def test_setdiff1d_char_array(self):
455: (8)                 a = np.array(['a', 'b', 'c'])
456: (8)                 b = np.array(['a', 'b', 's'])
457: (8)                 assert_array_equal(setdiff1d(a, b), np.array(['c']))
458: (4)             def test_manyways(self):
459: (8)                 a = np.array([5, 7, 1, 2, 8])
460: (8)                 b = np.array([9, 8, 2, 4, 3, 1, 5])
461: (8)                 c1 = setxor1d(a, b)
462: (8)                 aux1 = intersect1d(a, b)
463: (8)                 aux2 = union1d(a, b)
464: (8)                 c2 = setdiff1d(aux2, aux1)
465: (8)                 assert_array_equal(c1, c2)
466: (0)             class TestUnique:
467: (4)                 def test_unique_1d(self):
468: (8)                     def check_all(a, b, i1, i2, c, dt):
469: (12)                         base_msg = 'check {0} failed for type {1}'

```

```

470: (12)                         msg = base_msg.format('values', dt)
471: (12)                         v = unique(a)
472: (12)                         assert_array_equal(v, b, msg)
473: (12)                         msg = base_msg.format('return_index', dt)
474: (12)                         v, j = unique(a, True, False, False)
475: (12)                         assert_array_equal(v, b, msg)
476: (12)                         assert_array_equal(j, i1, msg)
477: (12)                         msg = base_msg.format('return_inverse', dt)
478: (12)                         v, j = unique(a, False, True, False)
479: (12)                         assert_array_equal(v, b, msg)
480: (12)                         assert_array_equal(j, i2, msg)
481: (12)                         msg = base_msg.format('return_counts', dt)
482: (12)                         v, j = unique(a, False, False, True)
483: (12)                         assert_array_equal(v, b, msg)
484: (12)                         assert_array_equal(j, c, msg)
485: (12)                         msg = base_msg.format('return_index and return_inverse', dt)
486: (12)                         v, j1, j2 = unique(a, True, True, False)
487: (12)                         assert_array_equal(v, b, msg)
488: (12)                         assert_array_equal(j1, i1, msg)
489: (12)                         assert_array_equal(j2, i2, msg)
490: (12)                         msg = base_msg.format('return_index and return_counts', dt)
491: (12)                         v, j1, j2 = unique(a, True, False, True)
492: (12)                         assert_array_equal(v, b, msg)
493: (12)                         assert_array_equal(j1, i1, msg)
494: (12)                         assert_array_equal(j2, c, msg)
495: (12)                         msg = base_msg.format('return_inverse and return_counts', dt)
496: (12)                         v, j1, j2 = unique(a, False, True, True)
497: (12)                         assert_array_equal(v, b, msg)
498: (12)                         assert_array_equal(j1, i2, msg)
499: (12)                         assert_array_equal(j2, c, msg)
500: (12)                         msg = base_msg.format(('return_index, return_inverse '
501: (35)                               'and return_counts'), dt)
502: (12)                         v, j1, j2, j3 = unique(a, True, True, True)
503: (12)                         assert_array_equal(v, b, msg)
504: (12)                         assert_array_equal(j1, i1, msg)
505: (12)                         assert_array_equal(j2, i2, msg)
506: (12)                         assert_array_equal(j3, c, msg)
507: (8)                          a = [5, 7, 1, 2, 1, 5, 7]*10
508: (8)                          b = [1, 2, 5, 7]
509: (8)                          i1 = [2, 3, 0, 1]
510: (8)                          i2 = [2, 3, 0, 1, 0, 2, 3]*10
511: (8)                          c = np.multiply([2, 1, 2, 2], 10)
512: (8)                          types = []
513: (8)                          types.extend(np.typecodes['AllInteger'])
514: (8)                          types.extend(np.typecodes['AllFloat'])
515: (8)                          types.append('datetime64[D]')
516: (8)                          types.append('timedelta64[D]')
517: (8)                          for dt in types:
518: (12)                            aa = np.array(a, dt)
519: (12)                            bb = np.array(b, dt)
520: (12)                            check_all(aa, bb, i1, i2, c, dt)
521: (8)                          dt = 'O'
522: (8)                          aa = np.empty(len(a), dt)
523: (8)                          aa[:] = a
524: (8)                          bb = np.empty(len(b), dt)
525: (8)                          bb[:] = b
526: (8)                          check_all(aa, bb, i1, i2, c, dt)
527: (8)                          dt = [('', 'i'), ('', 'i')]
528: (8)                          aa = np.array(list(zip(a, a)), dt)
529: (8)                          bb = np.array(list(zip(b, b)), dt)
530: (8)                          check_all(aa, bb, i1, i2, c, dt)
531: (8)                          aa = [1. + 0.j, 1 - 1.j, 1]
532: (8)                          assert_array_equal(np.unique(aa), [1. - 1.j, 1. + 0.j])
533: (8)                          a = [(1, 2), (1, 2), (2, 3)]
534: (8)                          unq = [1, 2, 3]
535: (8)                          inv = [0, 1, 0, 1, 1, 2]
536: (8)                          a1 = unique(a)
537: (8)                          assert_array_equal(a1, unq)
538: (8)                          a2, a2_inv = unique(a, return_inverse=True)

```

```

539: (8)             assert_array_equal(a2, unq)
540: (8)             assert_array_equal(a2_inv, inv)
541: (8)             a = np.chararray(5)
542: (8)             a[...] = ''
543: (8)             a2, a2_inv = np.unique(a, return_inverse=True)
544: (8)             assert_array_equal(a2_inv, np.zeros(5))
545: (8)             a = []
546: (8)             a1_idx = np.unique(a, return_index=True)[1]
547: (8)             a2_inv = np.unique(a, return_inverse=True)[1]
548: (8)             a3_idx, a3_inv = np.unique(a, return_index=True,
549: (35)                           return_inverse=True)[1:]
550: (8)             assert_equal(a1_idx.dtype, np.intp)
551: (8)             assert_equal(a2_inv.dtype, np.intp)
552: (8)             assert_equal(a3_idx.dtype, np.intp)
553: (8)             assert_equal(a3_inv.dtype, np.intp)
554: (8)             a = [2.0, np.nan, 1.0, np.nan]
555: (8)             ua = [1.0, 2.0, np.nan]
556: (8)             ua_idx = [2, 0, 1]
557: (8)             ua_inv = [1, 2, 0, 2]
558: (8)             ua_cnt = [1, 1, 2]
559: (8)             assert_equal(np.unique(a), ua)
560: (8)             assert_equal(np.unique(a, return_index=True), (ua, ua_idx))
561: (8)             assert_equal(np.unique(a, return_inverse=True), (ua, ua_inv))
562: (8)             assert_equal(np.unique(a, return_counts=True), (ua, ua_cnt))
563: (8)             a = [2.0-1j, np.nan, 1.0+1j, complex(0.0, np.nan), complex(1.0,
np.nan)]
564: (8)             ua = [1.0+1j, 2.0-1j, complex(0.0, np.nan)]
565: (8)             ua_idx = [2, 0, 3]
566: (8)             ua_inv = [1, 2, 0, 2, 2]
567: (8)             ua_cnt = [1, 1, 3]
568: (8)             assert_equal(np.unique(a), ua)
569: (8)             assert_equal(np.unique(a, return_index=True), (ua, ua_idx))
570: (8)             assert_equal(np.unique(a, return_inverse=True), (ua, ua_inv))
571: (8)             assert_equal(np.unique(a, return_counts=True), (ua, ua_cnt))
572: (8)             nat = np.datetime64('nat')
573: (8)             a = [np.datetime64('2020-12-26'), nat, np.datetime64('2020-12-24'),
nat]
574: (8)             ua = [np.datetime64('2020-12-24'), np.datetime64('2020-12-26'), nat]
575: (8)             ua_idx = [2, 0, 1]
576: (8)             ua_inv = [1, 2, 0, 2]
577: (8)             ua_cnt = [1, 1, 2]
578: (8)             assert_equal(np.unique(a), ua)
579: (8)             assert_equal(np.unique(a, return_index=True), (ua, ua_idx))
580: (8)             assert_equal(np.unique(a, return_inverse=True), (ua, ua_inv))
581: (8)             assert_equal(np.unique(a, return_counts=True), (ua, ua_cnt))
582: (8)             nat = np.timedelta64('nat')
583: (8)             a = [np.timedelta64(1, 'D'), nat, np.timedelta64(1, 'h'), nat]
584: (8)             ua = [np.timedelta64(1, 'h'), np.timedelta64(1, 'D'), nat]
585: (8)             ua_idx = [2, 0, 1]
586: (8)             ua_inv = [1, 2, 0, 2]
587: (8)             ua_cnt = [1, 1, 2]
588: (8)             assert_equal(np.unique(a), ua)
589: (8)             assert_equal(np.unique(a, return_index=True), (ua, ua_idx))
590: (8)             assert_equal(np.unique(a, return_inverse=True), (ua, ua_inv))
591: (8)             assert_equal(np.unique(a, return_counts=True), (ua, ua_cnt))
592: (8)             all_nans = [np.nan] * 4
593: (8)             ua = [np.nan]
594: (8)             ua_idx = [0]
595: (8)             ua_inv = [0, 0, 0, 0]
596: (8)             ua_cnt = [4]
597: (8)             assert_equal(np.unique(all_nans), ua)
598: (8)             assert_equal(np.unique(all_nans, return_index=True), (ua, ua_idx))
599: (8)             assert_equal(np.unique(all_nans, return_inverse=True), (ua, ua_inv))
600: (8)             assert_equal(np.unique(all_nans, return_counts=True), (ua, ua_cnt))
601: (4)             def test_unique_axis_errors(self):
602: (8)                 assert_raises(TypeError, self._run_axis_tests, object)
603: (8)                 assert_raises(TypeError, self._run_axis_tests,
604: (22)                               [(['a', int], ('b', object))])
605: (8)                 assert_raises(np.AxisError, unique, np.arange(10), axis=2)

```

```

606: (8)             assert_raises(np.AxisError, unique, np.arange(10), axis=-2)
607: (4)             def test_unique_axis_list(self):
608: (8)                 msg = "Unique failed on list of lists"
609: (8)                 inp = [[0, 1, 0], [0, 1, 0]]
610: (8)                 inp_arr = np.asarray(inp)
611: (8)                 assert_array_equal(unique(inp, axis=0), unique(inp_arr, axis=0), msg)
612: (8)                 assert_array_equal(unique(inp, axis=1), unique(inp_arr, axis=1), msg)
613: (4)             def test_unique_axis(self):
614: (8)                 types = []
615: (8)                 types.extend(np.typecodes['AllInteger'])
616: (8)                 types.extend(np.typecodes['AllFloat'])
617: (8)                 types.append('datetime64[D]')
618: (8)                 types.append('timedelta64[D]')
619: (8)                 types.append([('a', int), ('b', int)])
620: (8)                 types.append([('a', int), ('b', float)])
621: (8)                 for dtype in types:
622: (12)                     self._run_axis_tests(dtype)
623: (8)                 msg = 'Non-bitwise-equal booleans test failed'
624: (8)                 data = np.arange(10, dtype=np.uint8).reshape(-1, 2).view(bool)
625: (8)                 result = np.array([[False, True], [True, True]], dtype=bool)
626: (8)                 assert_array_equal(unique(data, axis=0), result, msg)
627: (8)                 msg = 'Negative zero equality test failed'
628: (8)                 data = np.array([[-0.0, 0.0], [0.0, -0.0], [-0.0, 0.0], [0.0, -0.0]])
629: (8)                 result = np.array([[-0.0, 0.0]])
630: (8)                 assert_array_equal(unique(data, axis=0), result, msg)
631: (4)             @pytest.mark.parametrize("axis", [0, -1])
632: (4)             def test_unique_1d_with_axis(self, axis):
633: (8)                 x = np.array([4, 3, 2, 3, 2, 1, 2, 2])
634: (8)                 uniq = unique(x, axis=axis)
635: (8)                 assert_array_equal(uniq, [1, 2, 3, 4])
636: (4)             def test_unique_axis_zeros(self):
637: (8)                 single_zero = np.empty(shape=(2, 0), dtype=np.int8)
638: (8)                 uniq, idx, inv, cnt = unique(single_zero, axis=0, return_index=True,
639: (37)                               return_inverse=True, return_counts=True)
640: (8)                 assert_equal(uniq.dtype, single_zero.dtype)
641: (8)                 assert_array_equal(uniq, np.empty(shape=(1, 0)))
642: (8)                 assert_array_equal(idx, np.array([0]))
643: (8)                 assert_array_equal(inv, np.array([0, 0]))
644: (8)                 assert_array_equal(cnt, np.array([2]))
645: (8)                 uniq, idx, inv, cnt = unique(single_zero, axis=1, return_index=True,
646: (37)                               return_inverse=True, return_counts=True)
647: (8)                 assert_equal(uniq.dtype, single_zero.dtype)
648: (8)                 assert_array_equal(uniq, np.empty(shape=(2, 0)))
649: (8)                 assert_array_equal(idx, np.array([]))
650: (8)                 assert_array_equal(inv, np.array([]))
651: (8)                 assert_array_equal(cnt, np.array([]))
652: (8)                 shape = (0, 2, 0, 3, 0, 4, 0)
653: (8)                 multiple_zeros = np.empty(shape=shape)
654: (8)                 for axis in range(len(shape)):
655: (12)                     expected_shape = list(shape)
656: (12)                     if shape[axis] == 0:
657: (16)                         expected_shape[axis] = 0
658: (12)                     else:
659: (16)                         expected_shape[axis] = 1
660: (12)                     assert_array_equal(unique(multiple_zeros, axis=axis),
661: (31)                           np.empty(shape=expected_shape))
662: (4)             def test_unique_masked(self):
663: (8)                 x = np.array([64, 0, 1, 2, 3, 63, 63, 0, 0, 0, 1, 2, 0, 63, 0],
664: (21)                               dtype='uint8')
665: (8)                 y = np.ma.masked_equal(x, 0)
666: (8)                 v = np.unique(y)
667: (8)                 v2, i, c = np.unique(y, return_index=True, return_counts=True)
668: (8)                 msg = 'Unique returned different results when asked for index'
669: (8)                 assert_array_equal(v.data, v2.data, msg)
670: (8)                 assert_array_equal(v.mask, v2.mask, msg)
671: (4)             def test_unique_sort_order_with_axis(self):
672: (8)                 fmt = "sort order incorrect for integer type '%s'"
673: (8)                 for dt in 'bhilq':
674: (12)                     a = np.array([-1, 0], dt)

```

```

675: (12)             b = np.unique(a, axis=0)
676: (12)             assert_array_equal(a, b, fmt % dt)
677: (4)              def _run_axis_tests(self, dtype):
678: (8)                data = np.array([[0, 1, 0, 0],
679: (25)                  [1, 0, 0, 0],
680: (25)                  [0, 1, 0, 0],
681: (25)                  [1, 0, 0, 0]]).astype(dtype)
682: (8)                msg = 'Unique with 1d array and axis=0 failed'
683: (8)                result = np.array([0, 1])
684: (8)                assert_array_equal(unique(data), result.astype(dtype), msg)
685: (8)                msg = 'Unique with 2d array and axis=0 failed'
686: (8)                result = np.array([[0, 1, 0, 0], [1, 0, 0, 0]])
687: (8)                assert_array_equal(unique(data, axis=0), result.astype(dtype), msg)
688: (8)                msg = 'Unique with 2d array and axis=1 failed'
689: (8)                result = np.array([[0, 0, 1], [0, 1, 0], [0, 0, 1], [0, 1, 0]])
690: (8)                assert_array_equal(unique(data, axis=1), result.astype(dtype), msg)
691: (8)                msg = 'Unique with 3d array and axis=2 failed'
692: (8)                data3d = np.array([[[1, 1],
693: (28)                  [1, 0]],
694: (27)                  [[0, 1],
695: (28)                  [0, 0]]]).astype(dtype)
696: (8)                result = np.take(data3d, [1, 0], axis=2)
697: (8)                assert_array_equal(unique(data3d, axis=2), result, msg)
698: (8)                uniq, idx, inv, cnt = unique(data, axis=0, return_index=True,
699: (37)                                return_inverse=True, return_counts=True)
700: (8)                msg = "Unique's return_index=True failed with axis=0"
701: (8)                assert_array_equal(data[idx], uniq, msg)
702: (8)                msg = "Unique's return_inverse=True failed with axis=0"
703: (8)                assert_array_equal(uniq[inv], data)
704: (8)                msg = "Unique's return_counts=True failed with axis=0"
705: (8)                assert_array_equal(cnt, np.array([2, 2]), msg)
706: (8)                uniq, idx, inv, cnt = unique(data, axis=1, return_index=True,
707: (37)                                return_inverse=True, return_counts=True)
708: (8)                msg = "Unique's return_index=True failed with axis=1"
709: (8)                assert_array_equal(data[:, idx], uniq)
710: (8)                msg = "Unique's return_inverse=True failed with axis=1"
711: (8)                assert_array_equal(uniq[:, inv], data)
712: (8)                msg = "Unique's return_counts=True failed with axis=1"
713: (8)                assert_array_equal(cnt, np.array([2, 1, 1]), msg)
714: (4)              def test_unique_nanequals(self):
715: (8)                a = np.array([1, 1, np.nan, np.nan, np.nan])
716: (8)                unq = np.unique(a)
717: (8)                not_unq = np.unique(a, equal_nan=False)
718: (8)                assert_array_equal(unq, np.array([1, np.nan]))
719: (8)                assert_array_equal(not_unq, np.array([1, np.nan, np.nan, np.nan]))
```

---

File 231 - test\_arrayterator.py:

```

1: (0)            from operator import mul
2: (0)            from functools import reduce
3: (0)            import numpy as np
4: (0)            from numpy.random import randint
5: (0)            from numpy.lib import Arrayterator
6: (0)            from numpy.testing import assert_
7: (0)            def test():
8: (4)              np.random.seed(np.arange(10))
9: (4)              ndims = randint(5)+1
10: (4)              shape = tuple(randint(10)+1 for dim in range(ndims))
11: (4)              els = reduce(mul, shape)
12: (4)              a = np.arange(els)
13: (4)              a.shape = shape
14: (4)              buf_size = randint(2*els)
15: (4)              b = Arrayterator(a, buf_size)
16: (4)              for block in b:
17: (8)                assert_(len(block.flat) <= (buf_size or els))
18: (4)                assert_(list(b.flat) == list(a.flat))
19: (4)                start = [randint(dim) for dim in shape]
```

```

20: (4)         stop = [randint(dim)+1 for dim in shape]
21: (4)         step = [randint(dim)+1 for dim in shape]
22: (4)         slice_ = tuple(slice(*t) for t in zip(start, stop, step))
23: (4)         c = b[slice_]
24: (4)         d = a[slice_]
25: (4)         for block in c:
26: (8)             assert_(len(block.flat) <= (buf_size or els))
27: (4)             assert_(np.all(c._array_() == d))
28: (4)             assert_(list(c.flat) == list(d.flat))
-----
```

File 232 - test\_financial\_expired.py:

```

1: (0)         import sys
2: (0)         import pytest
3: (0)         import numpy as np
4: (0)         def test_financial_expired():
5: (4)             match = 'NEP 32'
6: (4)             with pytest.warns(DeprecationWarning, match=match):
7: (8)                 func = np.fv
8: (4)             with pytest.raises(RuntimeError, match=match):
9: (8)                 func(1, 2, 3)
-----
```

File 233 - test\_format.py:

```

1: (0)         r''' Test the .npy file format.
2: (0)         Set up:
3: (4)             >>> import sys
4: (4)             >>> from io import BytesIO
5: (4)             >>> from numpy.lib import format
6: (4)
7: (4)             >>> scalars = [
8: (4)                 ...     np.uint8,
9: (4)                 ...     np.int8,
10: (4)                ...     np.uint16,
11: (4)                ...     np.int16,
12: (4)                ...     np.uint32,
13: (4)                ...     np.int32,
14: (4)                ...     np.uint64,
15: (4)                ...     np.int64,
16: (4)                ...     np.float32,
17: (4)                ...     np.float64,
18: (4)                ...     np.complex64,
19: (4)                ...     np.complex128,
20: (4)                ...     object,
21: (4)                 ... ]
22: (4)             >>>
23: (4)             >>> basic_arrays = []
24: (4)
25: (4)             >>> for scalar in scalars:
26: (4)                 for endian in '<>':
27: (4)                     dtype = np.dtype(scalar).newbyteorder(endian)
28: (4)                     basic = np.arange(15).astype(dtype)
29: (4)                     basic_arrays.extend([
30: (4)                         ...     np.array([], dtype=dtype),
31: (4)                         ...     np.array(10, dtype=dtype),
32: (4)                         ...     basic,
33: (4)                         ...     basic.reshape((3,5)),
34: (4)                         ...     basic.reshape((3,5)).T,
35: (4)                         ...     basic.reshape((3,5))[:-1,:2],
36: (4)                         ... ])
37: (4)
38: (4)             >>>
39: (4)             >>> Pdescr = [
40: (4)                 ...     ('x', 'i4', (2,)),
41: (4)                 ...     ('y', 'f8', (2, 2)),
```

```

42: (4)          ...      ('z', 'u1')])
43: (4)          >>>
44: (4)          >>>
45: (4)          >>> PbufferT = [
46: (4)          ...      ([3,2], [[6.,4.],[6.,4.]], 8),
47: (4)          ...      ([4,3], [[7.,5.],[7.,5.]], 9),
48: (4)          ...      ]
49: (4)          >>>
50: (4)          >>>
51: (4)          >>> Ndescr = [
52: (4)          ...      ('x', 'i4', (2,)),
53: (4)          ...      ('Info', [
54: (4)          ...          ('value', 'c16'),
55: (4)          ...          ('y2', 'f8'),
56: (4)          ...          ('Info2', [
57: (4)          ...              ('name', 'S2'),
58: (4)          ...              ('value', 'c16', (2,)),
59: (4)          ...              ('y3', 'f8', (2,)),
60: (4)          ...              ('z3', 'u4', (2,))),
61: (4)          ...          ('name', 'S2'),
62: (4)          ...          ('z2', 'b1'))),
63: (4)          ...          ('color', 'S2'),
64: (4)          ...          ('info', [
65: (4)          ...              ('Name', 'U8'),
66: (4)          ...              ('Value', 'c16'))),
67: (4)          ...          ('y', 'f8', (2, 2)),
68: (4)          ...          ('z', 'u1'))
69: (4)          >>>
70: (4)          >>>
71: (4)          >>> NbufferT = [
72: (4)          ...      ([3,2], (6j, 6., ('nn', [6j,4j], [6.,4.], [1,2]), 'NN', True),
'cc', ('NN', 6j), [[6.,4.],[6.,4.]], 8),
73: (4)          ...      ([4,3], (7j, 7., ('oo', [7j,5j], [7.,5.], [2,1]), 'OO', False),
'dd', ('OO', 7j), [[7.,5.],[7.,5.]], 9),
74: (4)          ...      ]
75: (4)          >>>
76: (4)          >>>
77: (4)          >>> record_arrays = [
78: (4)          ...      np.array(PbufferT, dtype=np.dtype(Pdescr).newbyteorder('<')),
79: (4)          ...      np.array(NbufferT, dtype=np.dtype(Ndescr).newbyteorder('<')),
80: (4)          ...      np.array(PbufferT, dtype=np.dtype(Pdescr).newbyteorder('>')),
81: (4)          ...      np.array(NbufferT, dtype=np.dtype(Ndescr).newbyteorder('>')),
82: (4)          ...      ]
83: (0)  Test the magic string writing.
84: (4)      >>> format.magic(1, 0)
85: (4)      '\x93NUMPY\x01\x00'
86: (4)      >>> format.magic(0, 0)
87: (4)      '\x93NUMPY\x00\x00'
88: (4)      >>> format.magic(255, 255)
89: (4)      '\x93NUMPY\xff\xff'
90: (4)      >>> format.magic(2, 5)
91: (4)      '\x93NUMPY\x02\x05'
92: (0)  Test the magic string reading.
93: (4)      >>> format.read_magic(BytesIO(format.magic(1, 0)))
94: (4)      (1, 0)
95: (4)      >>> format.read_magic(BytesIO(format.magic(0, 0)))
96: (4)      (0, 0)
97: (4)      >>> format.read_magic(BytesIO(format.magic(255, 255)))
98: (4)      (255, 255)
99: (4)      >>> format.read_magic(BytesIO(format.magic(2, 5)))
100: (4)      (2, 5)
101: (0)  Test the header writing.
102: (4)      >>> for arr in basic_arrays + record_arrays:
103: (4)          ...      f = BytesIO()
104: (4)          ...      format.write_array_header_1_0(f, arr) # XXX: arr is not a dict,
items gets called on it
105: (4)          ...      print(repr(f.getvalue()))
106: (4)          ...
107: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': (0,)}"

```

```
\n"
108: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': ()}
\n"
109: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': (15,)}
\n"
110: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': (3, 5)}
\n"
111: (4)          "F\x00{'descr': '|u1', 'fortran_order': True, 'shape': (5, 3)}
\n"
112: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': (3, 3)}
\n"
113: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': (0,)}
\n"
114: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': ()}
\n"
115: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': (15,)}
\n"
116: (4)          "F\x00{'descr': '|u1', 'fortran_order': False, 'shape': (3, 5)}
\n"
117: (4)          "F\x00{'descr': '|u1', 'fortran_order': True, 'shape': (5, 3)}
\n"
118: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': (0,)}
\n"
119: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': ()}
\n"
120: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': (15,)}
\n"
121: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': (3, 5)}
\n"
122: (4)          "F\x00{'descr': '|i1', 'fortran_order': True, 'shape': (5, 3)}
\n"
123: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': (3, 3)}
\n"
124: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': (0,)}
\n"
125: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': ()}
\n"
126: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': (15,)}
\n"
127: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': (3, 5)}
\n"
128: (4)          "F\x00{'descr': '|i1', 'fortran_order': True, 'shape': (5, 3)}
\n"
129: (4)          "F\x00{'descr': '|i1', 'fortran_order': False, 'shape': (3, 3)}
\n"
130: (4)          "F\x00{'descr': '<u2', 'fortran_order': False, 'shape': (0,)}
\n"
131: (4)          "F\x00{'descr': '<u2', 'fortran_order': False, 'shape': ()}
\n"
132: (4)          "F\x00{'descr': '<u2', 'fortran_order': False, 'shape': (15,)}
\n"
133: (4)          "F\x00{'descr': '<u2', 'fortran_order': False, 'shape': (3, 5)}
\n"
134: (4)          "F\x00{'descr': '<u2', 'fortran_order': True, 'shape': (5, 3)}
\n"
135: (4)          "F\x00{'descr': '<u2', 'fortran_order': False, 'shape': (3, 3)}
\n"
136: (4)          "F\x00{'descr': '>u2', 'fortran_order': False, 'shape': (0,)}
\n"
137: (4)          "F\x00{'descr': '>u2', 'fortran_order': False, 'shape': ()}
\n"
138: (4)          "F\x00{'descr': '>u2', 'fortran_order': False, 'shape': (15,)}
\n"
139: (4)          "F\x00{'descr': '>u2', 'fortran_order': False, 'shape': (3, 5)}
\n"
140: (4)          "F\x00{'descr': '>u2', 'fortran_order': True, 'shape': (5, 3)}
\n"
```

```
142: (4)
\n"
143: (4)
\n"
144: (4)
\n"
145: (4)
\n"
146: (4)
\n"
147: (4)
\n"
148: (4)
\n"
149: (4)
\n"
150: (4)
\n"
151: (4)
\n"
152: (4)
\n"
153: (4)
\n"
154: (4)
\n"
155: (4)
\n"
156: (4)
\n"
157: (4)
\n"
158: (4)
\n"
159: (4)
\n"
160: (4)
\n"
161: (4)
\n"
162: (4)
\n"
163: (4)
\n"
164: (4)
\n"
165: (4)
\n"
166: (4)
\n"
167: (4)
\n"
168: (4)
\n"
169: (4)
\n"
170: (4)
\n"
171: (4)
\n"
172: (4)
\n"
173: (4)
\n"
174: (4)
\n"
175: (4)
\n"
176: (4)

    "F\x00{'descr': '>u2', 'fortran_order': False, 'shape': (3, 3)}
    "F\x00{'descr': '<i2', 'fortran_order': False, 'shape': (0,)}
    "F\x00{'descr': '<i2', 'fortran_order': False, 'shape': ()}
    "F\x00{'descr': '<i2', 'fortran_order': False, 'shape': (15,)}
    "F\x00{'descr': '<i2', 'fortran_order': False, 'shape': (3, 5)}
    "F\x00{'descr': '<i2', 'fortran_order': True, 'shape': (5, 3)}
    "F\x00{'descr': '<i2', 'fortran_order': False, 'shape': (3, 3)}
    "F\x00{'descr': '>i2', 'fortran_order': False, 'shape': (0,)}
    "F\x00{'descr': '>i2', 'fortran_order': False, 'shape': (15,)}
    "F\x00{'descr': '>i2', 'fortran_order': False, 'shape': (3, 5)}
    "F\x00{'descr': '>i2', 'fortran_order': True, 'shape': (5, 3)}
    "F\x00{'descr': '>i2', 'fortran_order': False, 'shape': (3, 3)}
    "F\x00{'descr': '<u4', 'fortran_order': False, 'shape': (0,)}
    "F\x00{'descr': '<u4', 'fortran_order': False, 'shape': ()}
    "F\x00{'descr': '<u4', 'fortran_order': False, 'shape': (15,)}
    "F\x00{'descr': '<u4', 'fortran_order': False, 'shape': (3, 5)}
    "F\x00{'descr': '<u4', 'fortran_order': True, 'shape': (5, 3)}
    "F\x00{'descr': '<u4', 'fortran_order': False, 'shape': (3, 3)}
    "F\x00{'descr': '>u4', 'fortran_order': False, 'shape': (0,)}
    "F\x00{'descr': '>u4', 'fortran_order': False, 'shape': ()}
    "F\x00{'descr': '>u4', 'fortran_order': False, 'shape': (15,)}
    "F\x00{'descr': '>u4', 'fortran_order': False, 'shape': (3, 5)}
    "F\x00{'descr': '>u4', 'fortran_order': True, 'shape': (5, 3)}
    "F\x00{'descr': '>u4', 'fortran_order': False, 'shape': (3, 3)}
    "F\x00{'descr': '<i4', 'fortran_order': False, 'shape': (0,)}
    "F\x00{'descr': '<i4', 'fortran_order': False, 'shape': ()}
    "F\x00{'descr': '<i4', 'fortran_order': False, 'shape': (15,)}
    "F\x00{'descr': '<i4', 'fortran_order': False, 'shape': (3, 5)}
    "F\x00{'descr': '<i4', 'fortran_order': True, 'shape': (5, 3)}
    "F\x00{'descr': '<i4', 'fortran_order': False, 'shape': (3, 3)}
    "F\x00{'descr': '>i4', 'fortran_order': False, 'shape': (0,)}
    "F\x00{'descr': '>i4', 'fortran_order': False, 'shape': ()}
    "F\x00{'descr': '>i4', 'fortran_order': False, 'shape': (15,)}
    "F\x00{'descr': '>i4', 'fortran_order': False, 'shape': (3, 5)}
```

```
\n"
177: (4)          "F\x00{'descr': '>i4', 'fortran_order': True, 'shape': (5, 3)}
\n"
178: (4)          "F\x00{'descr': '>i4', 'fortran_order': False, 'shape': (3, 3)}
\n"
179: (4)          "F\x00{'descr': '<u8', 'fortran_order': False, 'shape': (0,)}
\n"
180: (4)          "F\x00{'descr': '<u8', 'fortran_order': False, 'shape': ()}
\n"
181: (4)          "F\x00{'descr': '<u8', 'fortran_order': False, 'shape': (15,)}
\n"
182: (4)          "F\x00{'descr': '<u8', 'fortran_order': False, 'shape': (3, 5)}
\n"
183: (4)          "F\x00{'descr': '<u8', 'fortran_order': True, 'shape': (5, 3)}
\n"
184: (4)          "F\x00{'descr': '<u8', 'fortran_order': False, 'shape': (3, 3)}
\n"
185: (4)          "F\x00{'descr': '>u8', 'fortran_order': False, 'shape': (0,)}
\n"
186: (4)          "F\x00{'descr': '>u8', 'fortran_order': False, 'shape': ()}
\n"
187: (4)          "F\x00{'descr': '>u8', 'fortran_order': False, 'shape': (15,)}
\n"
188: (4)          "F\x00{'descr': '>u8', 'fortran_order': False, 'shape': (3, 5)}
\n"
189: (4)          "F\x00{'descr': '>u8', 'fortran_order': True, 'shape': (5, 3)}
\n"
190: (4)          "F\x00{'descr': '>u8', 'fortran_order': False, 'shape': (3, 3)}
\n"
191: (4)          "F\x00{'descr': '<i8', 'fortran_order': False, 'shape': (0,)}
\n"
192: (4)          "F\x00{'descr': '<i8', 'fortran_order': False, 'shape': ()}
\n"
193: (4)          "F\x00{'descr': '<i8', 'fortran_order': False, 'shape': (15,)}
\n"
194: (4)          "F\x00{'descr': '<i8', 'fortran_order': False, 'shape': (3, 5)}
\n"
195: (4)          "F\x00{'descr': '<i8', 'fortran_order': True, 'shape': (5, 3)}
\n"
196: (4)          "F\x00{'descr': '<i8', 'fortran_order': False, 'shape': (3, 3)}
\n"
197: (4)          "F\x00{'descr': '>i8', 'fortran_order': False, 'shape': (0,)}
\n"
198: (4)          "F\x00{'descr': '>i8', 'fortran_order': False, 'shape': ()}
\n"
199: (4)          "F\x00{'descr': '>i8', 'fortran_order': False, 'shape': (15,)}
\n"
200: (4)          "F\x00{'descr': '>i8', 'fortran_order': False, 'shape': (3, 5)}
\n"
201: (4)          "F\x00{'descr': '>i8', 'fortran_order': True, 'shape': (5, 3)}
\n"
202: (4)          "F\x00{'descr': '>i8', 'fortran_order': False, 'shape': (3, 3)}
\n"
203: (4)          "F\x00{'descr': '<f4', 'fortran_order': False, 'shape': (0,)}
\n"
204: (4)          "F\x00{'descr': '<f4', 'fortran_order': False, 'shape': ()}
\n"
205: (4)          "F\x00{'descr': '<f4', 'fortran_order': False, 'shape': (15,)}
\n"
206: (4)          "F\x00{'descr': '<f4', 'fortran_order': False, 'shape': (3, 5)}
\n"
207: (4)          "F\x00{'descr': '<f4', 'fortran_order': True, 'shape': (5, 3)}
\n"
208: (4)          "F\x00{'descr': '<f4', 'fortran_order': False, 'shape': (3, 3)}
\n"
209: (4)          "F\x00{'descr': '>f4', 'fortran_order': False, 'shape': (0,)}
\n"
210: (4)          "F\x00{'descr': '>f4', 'fortran_order': False, 'shape': ()}
```

```
211: (4)
\n"
212: (4)
\n"
213: (4)
\n"
214: (4)
\n"
215: (4)
\n"
216: (4)
\n"
217: (4)
\n"
218: (4)
\n"
219: (4)
\n"
220: (4)
\n"
221: (4)
\n"
222: (4)
\n"
223: (4)
\n"
224: (4)
\n"
225: (4)
\n"
226: (4)
\n"
227: (4)
\n"
228: (4)
\n"
229: (4)
\n"
230: (4)
\n"
231: (4)
\n"
232: (4)
\n"
233: (4)
\n"
234: (4)
\n"
235: (4)
\n"
236: (4)
\n"
237: (4)
\n"
238: (4)
\n"
239: (4)
\n"
240: (4)
\n"
241: (4)
\n"
242: (4)
\n"
243: (4)
\n"
244: (4)
\n"
245: (4)
```

```
\n"
246: (4)          "F\x00{'descr': '>c16', 'fortran_order': False, 'shape': ()}"
\n"
247: (4)          "F\x00{'descr': '>c16', 'fortran_order': False, 'shape': (15,)}"
\n"
248: (4)          "F\x00{'descr': '>c16', 'fortran_order': False, 'shape': (3, 5)}"
\n"
249: (4)          "F\x00{'descr': '>c16', 'fortran_order': True, 'shape': (5, 3)}"
\n"
250: (4)          "F\x00{'descr': '>c16', 'fortran_order': False, 'shape': (3, 3)}"
\n"
251: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': (0,)}"
\n"
252: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': ()}"
\n"
253: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': (15,)}"
\n"
254: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': (3, 5)}"
\n"
255: (4)          "F\x00{'descr': '0', 'fortran_order': True, 'shape': (5, 3)}"
\n"
256: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': (3, 3)}"
\n"
257: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': (0,)}"
\n"
258: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': ()}"
\n"
259: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': (15,)}"
\n"
260: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': (3, 5)}"
\n"
261: (4)          "F\x00{'descr': '0', 'fortran_order': True, 'shape': (5, 3)}"
\n"
262: (4)          "F\x00{'descr': '0', 'fortran_order': False, 'shape': (3, 3)}"
\n"
263: (4)          "v\x00{'descr': [('x', '<i4', (2,)), ('y', '<f8', (2, 2)), ('z',
['|u1']),],\n  'fortran_order': False,\n  'shape': (2,)}\n"
264: (4)          "\x16\x02{'descr': [('x', '<i4', (2,)),\n  ('y2', '<f8'),\n  ('value', '|S2'),\n  ('z3', '<u4', (2,))],\n  ('name', '|S2'),\n  ('color', '|S2'),\n  ('info', [('Name', '<U8'), ('Value', '<c16')]),\n  ('z', '|u1')],\n  'fortran_order': False,\n  'shape': (2,)}\n"
265: (4)          "v\x00{'descr': [('x', '>i4', (2,)), ('y', '>f8', (2, 2)), ('z',
['|u1']),],\n  'fortran_order': False,\n  'shape': (2,)}\n"
266: (4)          "\x16\x02{'descr': [('x', '>i4', (2,)),\n  ('y2', '>f8'),\n  ('value', '>c16'),\n  ('z3', '>u4', (2,))],\n  ('name', '|S2'),\n  ('color', '|S2'),\n  ('info', [('Name', '>U8'), ('Value', '>c16')]),\n  ('z', '|u1')],\n  'fortran_order': False,\n  'shape': (2,)}\n"
267: (0)          ...
268: (0)          import sys
269: (0)          import os
270: (0)          import warnings
271: (0)          import pytest
272: (0)          from io import BytesIO
273: (0)          import numpy as np
274: (0)          from numpy.testing import (
275: (4)            assert_, assert_array_equal, assert_raises, assert_raises_regex,
276: (4)            assert_warns, IS_PYPY, IS_WASM
277: (4)          )
278: (0)          from numpy.testing._private.utils import requires_memory
279: (0)          from numpy.lib import format
280: (0)          scalars = [
281: (4)            np.uint8,
282: (4)            np.int8,
283: (4)            np.uint16,
284: (4)            np.int16,
```

```

285: (4)          np.uint32,
286: (4)          np.int32,
287: (4)          np.uint64,
288: (4)          np.int64,
289: (4)          np.float32,
290: (4)          np.float64,
291: (4)          np.complex64,
292: (4)          np.complex128,
293: (4)          object,
294: (0)
295: (0)          basic_arrays = []
296: (0)          for scalar in scalars:
297: (4)              for endian in '<>':
298: (8)                  dtype = np.dtype(scalar).newbyteorder(endian)
299: (8)                  basic = np.arange(1500).astype(dtype)
300: (8)                  basic_arrays.extend([
301: (12)                      np.array([], dtype=dtype),
302: (12)                      np.array(10, dtype=dtype),
303: (12)                      basic,
304: (12)                      basic.reshape((30, 50)),
305: (12)                      basic.reshape((30, 50)).T,
306: (12)                      basic.reshape((30, 50))[:-1, ::2],
307: (8)                  ])
308: (0)          Pdescr = [
309: (4)              ('x', 'i4', (2,)),
310: (4)              ('y', 'f8', (2, 2)),
311: (4)              ('z', 'u1')]
312: (0)          PbufferT = [
313: (4)              ([3, 2], [[6., 4.], [6., 4.]], 8),
314: (4)              ([4, 3], [[7., 5.], [7., 5.]], 9),
315: (4)          ]
316: (0)          Ndescr = [
317: (4)              ('x', 'i4', (2,)),
318: (4)              ('Info', [
319: (8)                  ('value', 'c16'),
320: (8)                  ('y2', 'f8'),
321: (8)                  ('Info2', [
322: (12)                      ('name', 'S2'),
323: (12)                      ('value', 'c16', (2,)),
324: (12)                      ('y3', 'f8', (2,)),
325: (12)                      ('z3', 'u4', (2,))),
326: (8)                      ('name', 'S2'),
327: (8)                      ('z2', 'b1'))),
328: (4)                  ('color', 'S2'),
329: (4)                  ('info', [
330: (8)                      ('Name', 'U8'),
331: (8)                      ('Value', 'c16'))),
332: (4)                  ('y', 'f8', (2, 2)),
333: (4)                  ('z', 'u1')]
334: (0)          NbufferT = [
335: (4)              ([3, 2], (6j, 6., ('nn', [6j, 4j], [6., 4.], [1, 2]), 'NN', True),
336: (5)              ('cc', ('NN', 6j), [[6., 4.], [6., 4.]], 8),
337: (4)              ([4, 3], (7j, 7., ('oo', [7j, 5j], [7., 5.], [2, 1]), 'OO', False),
338: (5)              ('dd', ('OO', 7j), [[7., 5.], [7., 5.]], 9),
339: (4)          ]
340: (0)          record_arrays = [
341: (4)              np.array(PbufferT, dtype=np.dtype(Pdescr).newbyteorder('<')),
342: (4)              np.array(NbufferT, dtype=np.dtype(Ndescr).newbyteorder('<')),
343: (4)              np.array(PbufferT, dtype=np.dtype(Pdescr).newbyteorder('>')),
344: (4)              np.array(NbufferT, dtype=np.dtype(Ndescr).newbyteorder('>')),
345: (4)              np.zeros(1, dtype=[('c', ('<f8', (5,)), (2,))])
346: (0)
347: (0)          class BytesIOSRandomSize(BytesIO):
348: (4)              def read(self, size=None):
349: (8)                  import random
350: (8)                  size = random.randint(1, size)
351: (8)                  return super().read(size)
352: (0)          def roundtrip(arr):
353: (4)              f = BytesIO()

```

```

354: (4)             format.write_array(f, arr)
355: (4)             f2 = BytesIO(f.getvalue())
356: (4)             arr2 = format.read_array(f2, allow_pickle=True)
357: (4)             return arr2
358: (0)             def roundtrip_randsize(arr):
359: (4)                 f = BytesIO()
360: (4)                 format.write_array(f, arr)
361: (4)                 f2 = BytesIOSRandomSize(f.getvalue())
362: (4)                 arr2 = format.read_array(f2)
363: (4)                 return arr2
364: (0)             def roundtrip_truncated(arr):
365: (4)                 f = BytesIO()
366: (4)                 format.write_array(f, arr)
367: (4)                 f2 = BytesIO(f.getvalue()[0:-1])
368: (4)                 arr2 = format.read_array(f2)
369: (4)                 return arr2
370: (0)             def assert_equal_(o1, o2):
371: (4)                 assert_(o1 == o2)
372: (0)             def test_roundtrip():
373: (4)                 for arr in basic_arrays + record_arrays:
374: (8)                     arr2 = roundtrip(arr)
375: (8)                     assert_array_equal(arr, arr2)
376: (0)             def test_roundtrip_randsize():
377: (4)                 for arr in basic_arrays + record_arrays:
378: (8)                     if arr.dtype != object:
379: (12)                         arr2 = roundtrip_randsize(arr)
380: (12)                         assert_array_equal(arr, arr2)
381: (0)             def test_roundtrip_truncated():
382: (4)                 for arr in basic_arrays:
383: (8)                     if arr.dtype != object:
384: (12)                         assert_raises(ValueError, roundtrip_truncated, arr)
385: (0)             def test_long_str():
386: (4)                 long_str_arr = np.ones(1, dtype=np.dtype((str, format.BUFFER_SIZE + 1)))
387: (4)                 long_str_arr2 = roundtrip(long_str_arr)
388: (4)                 assert_array_equal(long_str_arr, long_str_arr2)
389: (0)             @pytest.mark.skipif(IS_WASM, reason="memmap doesn't work correctly")
390: (0)             @pytest.mark.slow
391: (0)             def test_memmap_roundtrip(tmpdir):
392: (4)                 for i, arr in enumerate(basic_arrays + record_arrays):
393: (8)                     if arr.dtype.hasobject:
394: (12)                         continue
395: (8)                     nfn = os.path.join(tmpdir, f'normal{i}.npy')
396: (8)                     mfn = os.path.join(tmpdir, f'memmap{i}.npy')
397: (8)                     with open(nfn, 'wb') as fp:
398: (12)                         format.write_array(fp, arr)
399: (8)                     fortran_order = (
400: (12)                         arr.flags.f_contiguous and not arr.flags.c_contiguous)
401: (8)                     ma = format.open_memmap(mfn, mode='w+', dtype=arr.dtype,
402: (32)                                     shape=arr.shape, fortran_order=fortran_order)
403: (8)                     ma[...] = arr
404: (8)                     ma.flush()
405: (8)                     with open(nfn, 'rb') as fp:
406: (12)                         normal_bytes = fp.read()
407: (8)                     with open(mfn, 'rb') as fp:
408: (12)                         memmap_bytes = fp.read()
409: (8)                     assert_equal_(normal_bytes, memmap_bytes)
410: (8)                     ma = format.open_memmap(nfn, mode='r')
411: (8)                     ma.flush()
412: (0)             def test_compressed_roundtrip(tmpdir):
413: (4)                 arr = np.random.rand(200, 200)
414: (4)                 npz_file = os.path.join(tmpdir, 'compressed.npz')
415: (4)                 np.savez_compressed(npz_file, arr=arr)
416: (4)                 with np.load(npz_file) as npz:
417: (8)                     arr1 = npz['arr']
418: (4)                     assert_array_equal(arr, arr1)
419: (0)                     dt1 = np.dtype('i1, i4, i1', align=True)
420: (0)                     dt2 = np.dtype({'names': ['a', 'b'], 'formats': ['i4', 'i4'],
421: (16)                         'offsets': [1, 6] })
422: (0)                     dt3 = np.dtype({'names': ['c', 'd'], 'formats': ['i4', dt2] })

```

```

423: (0)
424: (0)
425: (16)
426: (0)
427: (0)
428: (0)
429: (4)
430: (4)
431: (8)
432: (4)
433: (4)
434: (4)
435: (8)
436: (4)
437: (0)
438: (0)
439: (0)
440: (4)
441: (4)
442: (4)
443: (8)
444: (4)
445: (0)
446: (4)
447: (4)
448: (25)
449: (24)
450: (4)
451: (18)
452: (8)
453: (8)
454: (12)
455: (12)
456: (16)
457: (16)
458: (12)
459: (16)
460: (12)
461: (16)
462: (16)
463: (16)
464: (12)
465: (16)
466: (16)
467: (8)
468: (12)
469: (16)
470: (16)
471: (16)
472: (16)
473: (31)
474: (16)
475: (16)
476: (12)
477: (16)
478: (30)
479: (16)
480: (30)
481: (30)
482: (0)
483: (4)
484: (4)
485: (4)
486: (18)
487: (4)
488: (4)
489: (8)
490: (4)
491: (4)

    dt4 = np.dtype({'names': ['a', '', 'b'], 'formats': ['i4']*3})
    dt5 = np.dtype({'names': ['a', 'b'], 'formats': ['i4', 'i4'],
                    'offsets': [1, 6], 'titles': ['aa', 'bb']})
    dt6 = np.dtype({'names': [], 'formats': [], 'itemsize': 8})
    @pytest.mark.parametrize("dt", [dt1, dt2, dt3, dt4, dt5, dt6])
    def test_load_padded_dtype(tmpdir, dt):
        arr = np.zeros(3, dt)
        for i in range(3):
            arr[i] = i + 5
        npz_file = os.path.join(tmpdir, 'aligned.npz')
        np.savez(npz_file, arr=arr)
        with np.load(npz_file) as npz:
            arr1 = npz['arr']
        assert_array_equal(arr, arr1)
    @pytest.mark.skipif(sys.version_info >= (3, 12), reason="see gh-23988")
    @pytest.mark.xfail(IS_WASM, reason="Emscripten NODEFS has a buggy dup")
    def test_python2_python3_interoperability():
        fname = 'win64python2.npy'
        path = os.path.join(os.path.dirname(__file__), 'data', fname)
        with pytest.warns(UserWarning, match="Reading.*this warning\\."):
            data = np.load(path)
        assert_array_equal(data, np.ones(2))
    def test_pickle_python2_python3():
        data_dir = os.path.join(os.path.dirname(__file__), 'data')
        expected = np.array([None, range, '\u0512a\u0826f',
                            b'\xe4\xb8\x8d\xe8\x89\xaf'],
                           dtype=object)
        for fname in ['py2-objarr.npy', 'py2-objarr.npz',
                      'py3-objarr.npy', 'py3-objarr.npz']:
            path = os.path.join(data_dir, fname)
            for encoding in ['bytes', 'latin1']:
                data_f = np.load(path, allow_pickle=True, encoding=encoding)
                if fname.endswith('.npz'):
                    data = data_f['x']
                    data_f.close()
                else:
                    data = data_f
                if encoding == 'latin1' and fname.startswith('py2'):
                    assert_(isinstance(data[3], str))
                    assert_array_equal(data[:-1], expected[:-1])
                    assert_array_equal(data[-1].encode(encoding), expected[-1])
                else:
                    assert_(isinstance(data[3], bytes))
                    assert_array_equal(data, expected)
            if fname.startswith('py2'):
                if fname.endswith('.npz'):
                    data = np.load(path, allow_pickle=True)
                    assert_raises(UnicodeError, data.__getitem__, 'x')
                    data.close()
                    data = np.load(path, allow_pickle=True, fix_imports=False,
                                   encoding='latin1')
                    assert_raises(ImportError, data.__getitem__, 'x')
                    data.close()
                else:
                    assert_raises(UnicodeError, np.load, path,
                                  allow_pickle=True)
            assert_raises(ImportError, np.load, path,
                          allow_pickle=True, fix_imports=False,
                          encoding='latin1')
    def test_pickle_disallow(tmpdir):
        data_dir = os.path.join(os.path.dirname(__file__), 'data')
        path = os.path.join(data_dir, 'py2-objarr.npy')
        assert_raises(ValueError, np.load, path,
                      allow_pickle=False, encoding='latin1')
        path = os.path.join(data_dir, 'py2-objarr.npz')
        with np.load(path, allow_pickle=False, encoding='latin1') as f:
            assert_raises(ValueError, f.__getitem__, 'x')
        path = os.path.join(tmpdir, 'pickle-disabled.npy')
        assert_raises(ValueError, np.save, path, np.array([None]), dtype=object),

```

```

492: (18)                                allow_pickle=False)
493: (0) @pytest.mark.parametrize('dt', [
494: (4)     np.dtype(np.dtype([('a', np.int8),
495: (23)         ('b', np.int16),
496: (23)         ('c', np.int32),
497: (22)         ], align=True),
498: (13)         (3,)),
499: (4)     np.dtype([('x', np.dtype({'names':['a','b'],
500: (30)             'formats':['i1','i1'],
501: (30)             'offsets':[0,4],
502: (30)             'itemsize':8,
503: (29)             },
504: (20)                 (3,)),
505: (15)                 (4,)),
506: (13)             )]),
507: (4)     np.dtype([('x',
508: (19)             ('<f8', (5,)),
509: (19)             (2,)),
510: (15)             )]),
511: (4)     np.dtype([('x', np.dtype(
512: (8)         np.dtype(
513: (12)             np.dtype({'names':['a','b'],
514: (22)                 'formats':['i1','i1'],
515: (22)                 'offsets':[0,4],
516: (22)                 'itemsize':8}),
517: (12)             (3,)),
518: (12)         )),
519: (8)             (4,)),
520: (8)         ))),
521: (8)     ],
522: (4)     np.dtype([
523: (8)         ('a', np.dtype(
524: (12)             np.dtype(
525: (16)                 np.dtype(
526: (20)                     np.dtype([
527: (24)                         ('a', int),
528: (24)                         ('b', np.dtype({'names':['a','b'],
529: (40)                             'formats':['i1','i1'],
530: (40)                             'offsets':[0,4],
531: (40)                             'itemsize':8}))),
532: (20)                     ])),
533: (20)                 (3,)),
534: (16)             )),
535: (16)             (4,)),
536: (12)         )),
537: (12)         (5,)),
538: (8)             ))),
539: (8)         ],
540: (4)     ]),
541: (0) def test_descr_to_dtype(dt):
542: (4)     dt1 = format.descr_to_dtype(dt.descr)
543: (4)     assert_equal(dt1, dt)
544: (4)     arr1 = np.zeros(3, dt)
545: (4)     arr2 = roundtrip(arr1)
546: (4)     assert_array_equal(arr1, arr2)
547: (0) def test_version_2_0():
548: (4)     f = BytesIO()
549: (4)     dt = [(("%d" % i) * 100, float) for i in range(500)]
550: (4)     d = np.ones(1000, dtype=dt)
551: (4)     format.write_array(f, d, version=(2, 0))
552: (4)     with warnings.catch_warnings(record=True) as w:
553: (8)         warnings.filterwarnings('always', '', UserWarning)
554: (8)         format.write_array(f, d)
555: (8)         assert_(w[0].category is UserWarning)
556: (4)     f.seek(0)
557: (4)     header = f.readline()
558: (4)     assert_(len(header) % format.ARRAY_ALIGN == 0)
559: (4)     f.seek(0)
560: (4)     n = format.read_array(f, max_header_size=200000)

```

```

561: (4)             assert_array_equal(d, n)
562: (4)             assert_raises(ValueError, format.write_array, f, d, (1, 0))
563: (0) @pytest.mark.skipif(IS_WASM, reason="memmap doesn't work correctly")
564: (0) def test_version_2_0_memmap(tmpdir):
565: (4)     dt = [(("%d" % i) * 100, float) for i in range(500)]
566: (4)     d = np.ones(1000, dtype=dt)
567: (4)     tf1 = os.path.join(tmpdir, f'version2_01.npy')
568: (4)     tf2 = os.path.join(tmpdir, f'version2_02.npy')
569: (4)     assert_raises(ValueError, format.open_memmap, tf1, mode='w+', 
570: (28)                   shape=d.shape, version=(1, 0))
571: (4)     ma = format.open_memmap(tf1, mode='w+', dtype=d.dtype,
572: (28)                   shape=d.shape, version=(2, 0))
573: (4)     ma[...] = d
574: (4)     ma.flush()
575: (4)     ma = format.open_memmap(tf1, mode='r', max_header_size=200000)
576: (4)     assert_array_equal(ma, d)
577: (4)     with warnings.catch_warnings(record=True) as w:
578: (8)         warnings.filterwarnings('always', '', UserWarning)
579: (8)         ma = format.open_memmap(tf2, mode='w+', dtype=d.dtype,
580: (32)               shape=d.shape, version=None)
581: (8)         assert_(w[0].category is UserWarning)
582: (8)         ma[...] = d
583: (8)         ma.flush()
584: (4)         ma = format.open_memmap(tf2, mode='r', max_header_size=200000)
585: (4)         assert_array_equal(ma, d)
586: (0) @pytest.mark.parametrize("mmap_mode", ["r", None])
587: (0) def test_huge_header(tmpdir, mmap_mode):
588: (4)     f = os.path.join(tmpdir, f'large_header.npy')
589: (4)     arr = np.array(1, dtype="i,*10000+i")
590: (4)     with pytest.warns(UserWarning, match=".format 2.0"):
591: (8)         np.save(f, arr)
592: (4)     with pytest.raises(ValueError, match="Header.*large"):
593: (8)         np.load(f, mmap_mode=mmap_mode)
594: (4)     with pytest.raises(ValueError, match="Header.*large"):
595: (8)         np.load(f, mmap_mode=mmap_mode, max_header_size=20000)
596: (4)         res = np.load(f, mmap_mode=mmap_mode, allow_pickle=True)
597: (4)         assert_array_equal(res, arr)
598: (4)         res = np.load(f, mmap_mode=mmap_mode, max_header_size=180000)
599: (4)         assert_array_equal(res, arr)
600: (0) def test_huge_header_npz(tmpdir):
601: (4)     f = os.path.join(tmpdir, f'large_header.npz')
602: (4)     arr = np.array(1, dtype="i,*10000+i")
603: (4)     with pytest.warns(UserWarning, match=".format 2.0"):
604: (8)         np.savez(f, arr=arr)
605: (4)     with pytest.raises(ValueError, match="Header.*large"):
606: (8)         np.load(f)["arr"]
607: (4)     with pytest.raises(ValueError, match="Header.*large"):
608: (8)         np.load(f, max_header_size=20000)["arr"]
609: (4)         res = np.load(f, allow_pickle=True)["arr"]
610: (4)         assert_array_equal(res, arr)
611: (4)         res = np.load(f, max_header_size=180000)["arr"]
612: (4)         assert_array_equal(res, arr)
613: (0) def test_write_version():
614: (4)     f = BytesIO()
615: (4)     arr = np.arange(1)
616: (4)     format.write_array(f, arr, version=(1, 0))
617: (4)     format.write_array(f, arr)
618: (4)     format.write_array(f, arr, version=None)
619: (4)     format.write_array(f, arr)
620: (4)     format.write_array(f, arr, version=(2, 0))
621: (4)     format.write_array(f, arr)
622: (4)     bad_versions = [
623: (8)         (1, 1),
624: (8)         (0, 0),
625: (8)         (0, 1),
626: (8)         (2, 2),
627: (8)         (255, 255),
628: (4)     ]

```

```

629: (4)             for version in bad_versions:
630: (8)                 with assert_raises_regex(ValueError,
631: (33)                         'we only support format version.*'):
632: (12)                         format.write_array(f, arr, version=version)
633: (0)             bad_version_magic = [
634: (4)                 b'\x93NUMPY\x01\x01',
635: (4)                 b'\x93NUMPY\x00\x00',
636: (4)                 b'\x93NUMPY\x00\x01',
637: (4)                 b'\x93NUMPY\x02\x00',
638: (4)                 b'\x93NUMPY\x02\x02',
639: (4)                 b'\x93NUMPY\xff\xff',
640: (0)
641: (0)             ]
642: (4)             malformed_magic = [
643: (4)                 b'\x92NUMPY\x01\x00',
644: (4)                 b'\x00NUMPY\x01\x00',
645: (4)                 b'\x93numpy\x01\x00',
646: (4)                 b'\x93MATLB\x01\x00',
647: (4)                 b'\x93NUMPY\x01',
648: (4)                 b'\x93NUMPY',
649: (0),
650: (0)             def test_read_magic():
651: (4)                 s1 = BytesIO()
652: (4)                 s2 = BytesIO()
653: (4)                 arr = np.ones((3, 6), dtype=float)
654: (4)                 format.write_array(s1, arr, version=(1, 0))
655: (4)                 format.write_array(s2, arr, version=(2, 0))
656: (4)                 s1.seek(0)
657: (4)                 s2.seek(0)
658: (4)                 version1 = format.read_magic(s1)
659: (4)                 version2 = format.read_magic(s2)
660: (4)                 assert_(version1 == (1, 0))
661: (4)                 assert_(version2 == (2, 0))
662: (4)                 assert_(s1.tell() == format.MAGIC_LEN)
663: (4)                 assert_(s2.tell() == format.MAGIC_LEN)
664: (0)             def test_read_magic_bad_magic():
665: (4)                 for magic in malformed_magic:
666: (8)                     f = BytesIO(magic)
667: (8)                     assert_raises(ValueError, format.read_array, f)
668: (0)             def test_read_version_1_0_bad_magic():
669: (4)                 for magic in bad_version_magic + malformed_magic:
670: (8)                     f = BytesIO(magic)
671: (8)                     assert_raises(ValueError, format.read_array, f)
672: (0)             def test_bad_magic_args():
673: (4)                 assert_raises(ValueError, format.magic, -1, 1)
674: (4)                 assert_raises(ValueError, format.magic, 256, 1)
675: (4)                 assert_raises(ValueError, format.magic, 1, -1)
676: (4)                 assert_raises(ValueError, format.magic, 1, 256)
677: (0)             def test_large_header():
678: (4)                 s = BytesIO()
679: (4)                 d = {'shape': tuple(), 'fortran_order': False, 'descr': '<i8'}
680: (4)                 format.write_array_header_1_0(s, d)
681: (4)                 s = BytesIO()
682: (4)                 d['descr'] = [('x'*256*256, '<i8')]
683: (4)                 assert_raises(ValueError, format.write_array_header_1_0, s, d)
684: (0)             def test_read_array_header_1_0():
685: (4)                 s = BytesIO()
686: (4)                 arr = np.ones((3, 6), dtype=float)
687: (4)                 format.write_array(s, arr, version=(1, 0))
688: (4)                 s.seek(format.MAGIC_LEN)
689: (4)                 shape, fortran, dtype = format.read_array_header_1_0(s)
690: (4)                 assert_(s.tell() % format.ARRAY_ALIGN == 0)
691: (4)                 assert_((shape, fortran, dtype) == ((3, 6), False, float))
692: (0)             def test_read_array_header_2_0():
693: (4)                 s = BytesIO()
694: (4)                 arr = np.ones((3, 6), dtype=float)
695: (4)                 format.write_array(s, arr, version=(2, 0))
696: (4)                 s.seek(format.MAGIC_LEN)
697: (4)                 shape, fortran, dtype = format.read_array_header_2_0(s)

```

```

698: (4)             assert_(s.tell() % format.ARRAY_ALIGN == 0)
699: (4)             assert_((shape, fortran, dtype) == ((3, 6), False, float))
700: (0) def test_bad_header():
701: (4)             s = BytesIO()
702: (4)             assert_raises(ValueError, format.read_array_header_1_0, s)
703: (4)             s = BytesIO(b'1')
704: (4)             assert_raises(ValueError, format.read_array_header_1_0, s)
705: (4)             s = BytesIO(b'\x01\x00')
706: (4)             assert_raises(ValueError, format.read_array_header_1_0, s)
707: (4)             s = BytesIO(
708: (8)                 b"\x93NUMPY\x01\x006\x00{'descr': 'x', 'shape': (1, 2), }" +
709: (8)                 b"\n"
710: (4)             )
711: (4)             assert_raises(ValueError, format.read_array_header_1_0, s)
712: (4)             d = {"shape": (1, 2),
713: (9)                 "fortran_order": False,
714: (9)                 "descr": "x",
715: (9)                 "extrakey": -1}
716: (4)             s = BytesIO()
717: (4)             format.write_array_header_1_0(s, d)
718: (4)             assert_raises(ValueError, format.read_array_header_1_0, s)
719: (0) def test_large_file_support(tmpdir):
720: (4)             if (sys.platform == 'win32' or sys.platform == 'cygwin'):
721: (8)                 pytest.skip("Unknown if Windows has sparse filesystems")
722: (4)             tf_name = os.path.join(tmpdir, 'sparse_file')
723: (4)             try:
724: (8)                 import subprocess as sp
725: (8)                 sp.check_call(["truncate", "-s", "5368709120", tf_name])
726: (4)             except Exception:
727: (8)                 pytest.skip("Could not create 5GB large file")
728: (4)             with open(tf_name, "wb") as f:
729: (8)                 f.seek(5368709120)
730: (8)                 d = np.arange(5)
731: (8)                 np.save(f, d)
732: (4)             with open(tf_name, "rb") as f:
733: (8)                 f.seek(5368709120)
734: (8)                 r = np.load(f)
735: (4)                 assert_array_equal(r, d)
736: (0) @pytest.mark.skipif(IS_PYPY, reason="flaky on PyPy")
737: (0) @pytest.mark.skipif(np.dtype(np.intp).itemsize < 8,
738: (20)                     reason="test requires 64-bit system")
739: (0) @pytest.mark.slow
740: (0) @requires_memory(free_bytes=2 * 2**30)
741: (0) def test_large_archive(tmpdir):
742: (4)             shape = (2**30, 2)
743: (4)             try:
744: (8)                 a = np.empty(shape, dtype=np.uint8)
745: (4)             except MemoryError:
746: (8)                 pytest.skip("Could not create large file")
747: (4)             fname = os.path.join(tmpdir, "large_archive")
748: (4)             with open(fname, "wb") as f:
749: (8)                 np.savez(f, arr=a)
750: (4)             del a
751: (4)             with open(fname, "rb") as f:
752: (8)                 new_a = np.load(f)["arr"]
753: (4)                 assert new_a.shape == shape
754: (0) def test_empty_npz(tmpdir):
755: (4)             fname = os.path.join(tmpdir, "nothing.npz")
756: (4)             np.savez(fname)
757: (4)             with np.load(fname) as nps:
758: (8)                 pass
759: (0) def test_unicode_field_names(tmpdir):
760: (4)             arr = np.array([
761: (8)                 (1, 3),
762: (8)                 (1, 2),
763: (8)                 (1, 3),
764: (8)                 (1, 2)
765: (4)             ], dtype=[
766: (8)                 ('int', int),

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

767: (8)          ('\\N{CJK UNIFIED IDEOGRAPH-6574}\\N{CJK UNIFIED IDEOGRAPH-5F62}', int)
768: (4)          ])
769: (4)          fname = os.path.join(tmpdir, "unicode.npy")
770: (4)          with open(fname, 'wb') as f:
771: (8)            format.write_array(f, arr, version=(3, 0))
772: (4)          with open(fname, 'rb') as f:
773: (8)            arr2 = format.read_array(f)
774: (4)          assert_array_equal(arr, arr2)
775: (4)          with open(fname, 'wb') as f:
776: (8)            with assert_warns(UserWarning):
777: (12)              format.write_array(f, arr, version=None)
778: (0)          def test_header_growth_axis():
779: (4)            for is_fortran_array, dtype_space, expected_header_length in [
780: (8)              [False, 22, 128], [False, 23, 192], [True, 23, 128], [True, 24, 192]
781: (4)            ]:
782: (8)              for size in [10**i for i in range(format.GROWTH_AXIS_MAX_DIGITS)]:
783: (12)                fp = BytesIO()
784: (12)                format.write_array_header_1_0(fp, {
785: (16)                  'shape': (2, size) if is_fortran_array else (size, 2),
786: (16)                  'fortran_order': is_fortran_array,
787: (16)                  'descr': np.dtype([(' *dtype_space, int)])})
788: (12)                })
789: (12)                assert len(fp.getvalue()) == expected_header_length
790: (0)          @pytest.mark.parametrize('dt, fail', [
791: (4)            (np.dtype({'names': ['a', 'b'], 'formats': [float, np.dtype('S3',
792: (17)              metadata={'some': 'stuff'})]}), True),
793: (4)            (np.dtype(int, metadata={'some': 'stuff'}), False),
794: (4)            (np.dtype([('subarray', (int, (2,)))]), metadata={'some': 'stuff'}),
795: (4)            (np.dtype({'names': ['a', 'b'], 'formats': [
796: (8)              float, np.dtype({'names': ['c'], 'formats': [np.dtype(int, metadata=
797: (4)                {})]})]}),
798: (4)                )),
799: (0)          @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
800: (8)            reason="PyPy bug in error formatting")
801: (0)          def test_metadata_dtype(dt, fail):
802: (4)            arr = np.ones(10, dtype=dt)
803: (4)            buf = BytesIO()
804: (4)            with assert_warns(UserWarning):
805: (8)              np.save(buf, arr)
806: (4)              buf.seek(0)
807: (4)              if fail:
808: (8)                with assert_raises(ValueError):
809: (12)                  np.load(buf)
810: (4)              else:
811: (8)                arr2 = np.load(buf)
812: (8)                from numpy.lib.utils import drop_metadata
813: (8)                assert_array_equal(arr, arr2)
814: (8)                assert drop_metadata(arr.dtype) is not arr.dtype
815: (8)                assert drop_metadata(arr2.dtype) is arr2.dtype

```

---

File 234 - test\_function\_base.py:

```

1: (0)          import operator
2: (0)          import warnings
3: (0)          import sys
4: (0)          import decimal
5: (0)          from fractions import Fraction
6: (0)          import math
7: (0)          import pytest
8: (0)          import hypothesis
9: (0)          from hypothesis.extra.numpy import arrays
10: (0)         import hypothesis.strategies as st
11: (0)         from functools import partial
12: (0)         import numpy as np
13: (0)         from numpy import ma

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

14: (0)
15: (4)
16: (4)
17: (4)
IS_WASM
18: (4)
19: (0)
20: (0)
21: (0)
22: (4)
23: (4)
24: (4)
25: (4)
26: (4)
27: (0)
28: (0)
29: (4)
30: (4)
31: (4)
32: (0)
33: (4)
34: (4)
finite
35: (4)
36: (4)
37: (4)
38: (4)
39: (4)
40: (4)
41: (0)
42: (4)
43: (8)
44: (8)
45: (8)
46: (8)
47: (8)
48: (8)
49: (13)
50: (8)
51: (14)
52: (14)
53: (8)
54: (14)
55: (8)
56: (14)
57: (14)
58: (8)
59: (14)
60: (8)
61: (12)
62: (8)
63: (12)
64: (8)
65: (12)
66: (8)
67: (12)
68: (8)
69: (8)
70: (4)
71: (8)
72: (8)
73: (8)
74: (8)
75: (4)
76: (8)
77: (8)
78: (23)
79: (22)
80: (23)

from numpy.testing import (
    assert_, assert_equal, assert_array_equal, assert_almost_equal,
    assert_array_almost_equal, assert_raises, assert_allclose, IS_PYPY,
    assert_warns, assert_raises_regex, suppress_warnings, HAS_REFCOUNT,
)
import numpy.lib.function_base as nfb
from numpy.random import rand
from numpy.lib import (
    add_newdoc_ufunc, angle, average, bartlett, blackman, corrcoef, cov,
    delete, diff, digitize, extract, flipud, gradient, hamming, hanning,
    i0, insert, interp, kaiser, meshgrid, msort, piecewise, place, rot90,
    select, setxor1d, sinc, trapz, trim_zeros, unwrap, unique, vectorize
)
from numpy.core.numeric import normalize_axis_tuple
def get_mat(n):
    data = np.arange(n)
    data = np.add.outer(data, data)
    return data
def make_complex(real, imag):
    """
    Like real + 1j * imag, but behaves as expected when imag contains non-
    finite
    values
    """
    ret = np.zeros(np.broadcast(real, imag).shape, np.complex_)
    ret.real = real
    ret.imag = imag
    return ret
class TestRot90:
    def test_basic(self):
        assert_raises(ValueError, rot90, np.ones(4))
        assert_raises(ValueError, rot90, np.ones((2,2,2)), axes=(0,1,2))
        assert_raises(ValueError, rot90, np.ones((2,2)), axes=(0,2))
        assert_raises(ValueError, rot90, np.ones((2,2)), axes=(1,1))
        assert_raises(ValueError, rot90, np.ones((2,2,2)), axes=(-2,1))
        a = [[0, 1, 2],
              [3, 4, 5]]
        b1 = [[2, 5],
              [1, 4],
              [0, 3]]
        b2 = [[5, 4, 3],
              [2, 1, 0]]
        b3 = [[3, 0],
              [4, 1],
              [5, 2]]
        b4 = [[0, 1, 2],
              [3, 4, 5]]
        for k in range(-3, 13, 4):
            assert_equal(rot90(a, k=k), b1)
        for k in range(-2, 13, 4):
            assert_equal(rot90(a, k=k), b2)
        for k in range(-1, 13, 4):
            assert_equal(rot90(a, k=k), b3)
        for k in range(0, 13, 4):
            assert_equal(rot90(a, k=k), b4)
        assert_equal(rot90(rot90(a, axes=(0,1)), axes=(1,0)), a)
        assert_equal(rot90(a, k=1, axes=(1,0)), rot90(a, k=-1, axes=(0,1)))
    def test_axes(self):
        a = np.ones((50, 40, 3))
        assert_equal(rot90(a).shape, (40, 50, 3))
        assert_equal(rot90(a, axes=(0,2)), rot90(a, axes=(0,-1)))
        assert_equal(rot90(a, axes=(1,2)), rot90(a, axes=(-2,-1)))
    def test_rotation_axes(self):
        a = np.arange(8).reshape((2,2,2))
        a_rot90_01 = [[[2, 3],
                      [6, 7]],
                     [[0, 1],
                      [4, 5]]]

```

```

81: (8)             a_rot90_12 = [[[1, 3],
82: (23)           [0, 2]],
83: (22)           [[5, 7],
84: (23)           [4, 6]]]
85: (8)             a_rot90_20 = [[[4, 0],
86: (23)           [6, 2]],
87: (22)           [[5, 1],
88: (23)           [7, 3]]]
89: (8)             a_rot90_10 = [[[4, 5],
90: (23)           [0, 1]],
91: (22)           [[6, 7],
92: (23)           [2, 3]]]
93: (8)             assert_equal(rot90(a, axes=(0, 1)), a_rot90_01)
94: (8)             assert_equal(rot90(a, axes=(1, 0)), a_rot90_10)
95: (8)             assert_equal(rot90(a, axes=(1, 2)), a_rot90_12)
96: (8)             for k in range(1,5):
97: (12)               assert_equal(rot90(a, k=k, axes=(2, 0)),
98: (25)                   rot90(a_rot90_20, k=k-1, axes=(2, 0)))
99: (0)
100: (4)            class TestFlip:
101: (8)              def test_axes(self):
102: (8)                assert_raises(np.AxisError, np.flip, np.ones(4), axis=1)
103: (8)                assert_raises(np.AxisError, np.flip, np.ones((4, 4)), axis=2)
104: (8)                assert_raises(np.AxisError, np.flip, np.ones((4, 4)), axis=-3)
105: (4)                assert_raises(np.AxisError, np.flip, np.ones((4, 4)), axis=(0, 3))
106: (8)              def test_basic_lr(self):
107: (8)                a = get_mat(4)
108: (8)                b = a[:, ::-1]
109: (8)                assert_equal(np.flip(a, 1), b)
110: (13)               a = [[0, 1, 2],
111: (8)                 [3, 4, 5]]
112: (13)               b = [[2, 1, 0],
113: (8)                 [5, 4, 3]]
114: (4)                assert_equal(np.flip(a, 1), b)
115: (8)              def test_basic_ud(self):
116: (8)                a = get_mat(4)
117: (8)                b = a[::-1, :]
118: (8)                assert_equal(np.flip(a, 0), b)
119: (13)               a = [[0, 1, 2],
120: (8)                 [3, 4, 5]]
121: (13)               b = [[3, 4, 5],
122: (8)                 [0, 1, 2]]
123: (4)                assert_equal(np.flip(a, 0), b)
124: (8)              def test_3d_swap_axis0(self):
125: (23)                a = np.array([[[0, 1],
126: (22)                  [2, 3]],
127: (23)                  [[4, 5],
128: (8)                    [6, 7]]])
129: (23)                b = np.array([[[4, 5],
130: (22)                  [6, 7]],
131: (23)                  [[0, 1],
132: (8)                    [2, 3]]])
133: (4)                assert_equal(np.flip(a, 0), b)
134: (8)              def test_3d_swap_axis1(self):
135: (23)                a = np.array([[[0, 1],
136: (22)                  [2, 3]],
137: (23)                  [[4, 5],
138: (8)                    [6, 7]]])
139: (23)                b = np.array([[[2, 3],
140: (22)                  [0, 1]],
141: (23)                  [[6, 7],
142: (8)                    [4, 5]]])
143: (4)                assert_equal(np.flip(a, 1), b)
144: (8)              def test_3d_swap_axis2(self):
145: (23)                a = np.array([[[0, 1],
146: (22)                  [2, 3]],
147: (23)                  [[4, 5],
148: (8)                    [6, 7]]])
149: (23)                b = np.array([[[1, 0],
150: (23)                  [3, 2]],
```

```

150: (22)                         [[5, 4],
151: (23)                         [7, 6]])
152: (8)                          assert_equal(np.flip(a, 2), b)
153: (4)                           def test_4d(self):
154: (8)                             a = np.arange(2 * 3 * 4 * 5).reshape(2, 3, 4, 5)
155: (8)                             for i in range(a.ndim):
156: (12)                               assert_equal(np.flip(a, i),
157: (25)                                     np.flipud(a.swapaxes(0, i)).swapaxes(i, 0))
158: (4)                           def test_default_axis(self):
159: (8)                             a = np.array([[1, 2, 3],
160: (22)                               [4, 5, 6]])
161: (8)                             b = np.array([[6, 5, 4],
162: (22)                               [3, 2, 1]])
163: (8)                             assert_equal(np.flip(a), b)
164: (4)                           def test_multiple_axes(self):
165: (8)                             a = np.array([[[0, 1],
166: (23)                               [2, 3]],
167: (22)                               [[4, 5],
168: (23)                                 [6, 7]]])
169: (8)                             assert_equal(np.flip(a, axis=()), a)
170: (8)                             b = np.array([[[5, 4],
171: (23)                               [7, 6]],
172: (22)                               [[1, 0],
173: (23)                                 [3, 2]]])
174: (8)                             assert_equal(np.flip(a, axis=(0, 2)), b)
175: (8)                             c = np.array([[[3, 2],
176: (23)                               [1, 0]],
177: (22)                               [[7, 6],
178: (23)                                 [5, 4]]])
179: (8)                             assert_equal(np.flip(a, axis=(1, 2)), c)
180: (0)                            class TestAny:
181: (4)                              def test_basic(self):
182: (8)                                y1 = [0, 0, 1, 0]
183: (8)                                y2 = [0, 0, 0, 0]
184: (8)                                y3 = [1, 0, 1, 0]
185: (8)                                assert_(np.any(y1))
186: (8)                                assert_(np.any(y3))
187: (8)                                assert_(not np.any(y2))
188: (4)                              def test_nd(self):
189: (8)                                y1 = [[0, 0, 0], [0, 1, 0], [1, 1, 0]]
190: (8)                                assert_(np.any(y1))
191: (8)                                assert_array_equal(np.any(y1, axis=0), [1, 1, 0])
192: (8)                                assert_array_equal(np.any(y1, axis=1), [0, 1, 1])
193: (0)                            class TestAll:
194: (4)                              def test_basic(self):
195: (8)                                y1 = [0, 1, 1, 0]
196: (8)                                y2 = [0, 0, 0, 0]
197: (8)                                y3 = [1, 1, 1, 1]
198: (8)                                assert_(not np.all(y1))
199: (8)                                assert_(np.all(y3))
200: (8)                                assert_(not np.all(y2))
201: (8)                                assert_(np.all(~np.array(y2)))
202: (4)                              def test_nd(self):
203: (8)                                y1 = [[0, 0, 1], [0, 1, 1], [1, 1, 1]]
204: (8)                                assert_(not np.all(y1))
205: (8)                                assert_array_equal(np.all(y1, axis=0), [0, 0, 1])
206: (8)                                assert_array_equal(np.all(y1, axis=1), [0, 0, 1])
207: (0)                            class TestCopy:
208: (4)                              def test_basic(self):
209: (8)                                a = np.array([[1, 2], [3, 4]])
210: (8)                                a_copy = np.copy(a)
211: (8)                                assert_array_equal(a, a_copy)
212: (8)                                a_copy[0, 0] = 10
213: (8)                                assert_equal(a[0, 0], 1)
214: (8)                                assert_equal(a_copy[0, 0], 10)
215: (4)                              def test_order(self):
216: (8)                                a = np.array([[1, 2], [3, 4]])
217: (8)                                assert_(a.flags.c_contiguous)
218: (8)                                assert_(not a.flags.f_contiguous)

```

```

219: (8)             a_fort = np.array([[1, 2], [3, 4]], order="F")
220: (8)             assert_(not a_fort.flags.c_contiguous)
221: (8)             assert_(a_fort.flags.f_contiguous)
222: (8)             a_copy = np.copy(a)
223: (8)             assert_(a_copy.flags.c_contiguous)
224: (8)             assert_(not a_copy.flags.f_contiguous)
225: (8)             a_fort_copy = np.copy(a_fort)
226: (8)             assert_(not a_fort_copy.flags.c_contiguous)
227: (8)             assert_(a_fort_copy.flags.f_contiguous)
228: (4)             def test_subok(self):
229: (8)                 mx = ma.ones(5)
230: (8)                 assert_(not ma.isMaskedArray(np.copy(mx, subok=False)))
231: (8)                 assert_(ma.isMaskedArray(np.copy(mx, subok=True)))
232: (8)                 assert_(not ma.isMaskedArray(np.copy(mx)))
233: (0)             class TestAverage:
234: (4)                 def test_basic(self):
235: (8)                     y1 = np.array([1, 2, 3])
236: (8)                     assert_(average(y1, axis=0) == 2.)
237: (8)                     y2 = np.array([1., 2., 3.])
238: (8)                     assert_(average(y2, axis=0) == 2.)
239: (8)                     y3 = [0., 0., 0.]
240: (8)                     assert_(average(y3, axis=0) == 0.)
241: (8)                     y4 = np.ones((4, 4))
242: (8)                     y4[0, 1] = 0
243: (8)                     y4[1, 0] = 2
244: (8)                     assert_almost_equal(y4.mean(0), average(y4, 0))
245: (8)                     assert_almost_equal(y4.mean(1), average(y4, 1))
246: (8)                     y5 = rand(5, 5)
247: (8)                     assert_almost_equal(y5.mean(0), average(y5, 0))
248: (8)                     assert_almost_equal(y5.mean(1), average(y5, 1))
249: (4)             @pytest.mark.parametrize(
250: (8)                 'x, axis, expected_avg, weights, expected_wavg, expected_wsum',
251: (8)                 '[[[1, 2, 3], None, [2.0], [3, 4, 1], [1.75], [8.0]],
252: (9)                 ([[1, 2, 5], [1, 6, 11]], 0, [[1.0, 4.0, 8.0]],
253: (10)                [1, 3], [[1.0, 5.0, 9.5]], [[4, 4, 4]]]', ),
254: (4)
255: (4)             def test_basic_keepdims(self, x, axis, expected_avg,
256: (28)                         weights, expected_wavg, expected_wsum):
257: (8)                 avg = np.average(x, axis=axis, keepdims=True)
258: (8)                 assert avg.shape == np.shape(expected_avg)
259: (8)                 assert_array_equal(avg, expected_avg)
260: (8)                 wavg = np.average(x, axis=axis, weights=weights, keepdims=True)
261: (8)                 assert wavg.shape == np.shape(expected_wavg)
262: (8)                 assert_array_equal(wavg, expected_wavg)
263: (8)                 wavg, wsum = np.average(x, axis=axis, weights=weights, returned=True,
264: (32)                           keepdims=True)
265: (8)                 assert wavg.shape == np.shape(expected_wavg)
266: (8)                 assert_array_equal(wavg, expected_wavg)
267: (8)                 assert wsum.shape == np.shape(expected_wsum)
268: (8)                 assert_array_equal(wsum, expected_wsum)
269: (4)             def test_weights(self):
270: (8)                 y = np.arange(10)
271: (8)                 w = np.arange(10)
272: (8)                 actual = average(y, weights=w)
273: (8)                 desired = (np.arange(10) ** 2).sum() * 1. / np.arange(10).sum()
274: (8)                 assert_almost_equal(actual, desired)
275: (8)                 y1 = np.array([[1, 2, 3], [4, 5, 6]])
276: (8)                 w0 = [1, 2]
277: (8)                 actual = average(y1, weights=w0, axis=0)
278: (8)                 desired = np.array([3., 4., 5.])
279: (8)                 assert_almost_equal(actual, desired)
280: (8)                 w1 = [0, 0, 1]
281: (8)                 actual = average(y1, weights=w1, axis=1)
282: (8)                 desired = np.array([3., 6.])
283: (8)                 assert_almost_equal(actual, desired)
284: (8)                 w2 = [[0, 0, 1], [0, 0, 2]]
285: (8)                 desired = np.array([3., 6.])
286: (8)                 assert_array_equal(average(y1, weights=w2, axis=1), desired)
287: (8)                 assert_equal(average(y1, weights=w2), 5.)

```

```

288: (8)          y3 = rand(5).astype(np.float32)
289: (8)          w3 = rand(5).astype(np.float64)
290: (8)          assert_(np.average(y3, weights=w3).dtype == np.result_type(y3, w3))
291: (8)          x = np.array([2, 3, 4]).reshape(3, 1)
292: (8)          w = np.array([4, 5, 6]).reshape(3, 1)
293: (8)          actual = np.average(x, weights=w, axis=1, keepdims=False)
294: (8)          desired = np.array([2., 3., 4.])
295: (8)          assert_array_equal(actual, desired)
296: (8)          actual = np.average(x, weights=w, axis=1, keepdims=True)
297: (8)          desired = np.array([[2.], [3.], [4.]])
298: (8)          assert_array_equal(actual, desired)
299: (4)          def test_returned(self):
300: (8)              y = np.array([[1, 2, 3], [4, 5, 6]])
301: (8)              avg, scl = average(y, returned=True)
302: (8)              assert_equal(scl, 6.)
303: (8)              avg, scl = average(y, 0, returned=True)
304: (8)              assert_array_equal(scl, np.array([2., 2., 2.]))
305: (8)              avg, scl = average(y, 1, returned=True)
306: (8)              assert_array_equal(scl, np.array([3., 3.]))
307: (8)              w0 = [1, 2]
308: (8)              avg, scl = average(y, weights=w0, axis=0, returned=True)
309: (8)              assert_array_equal(scl, np.array([3., 3., 3.]))
310: (8)              w1 = [1, 2, 3]
311: (8)              avg, scl = average(y, weights=w1, axis=1, returned=True)
312: (8)              assert_array_equal(scl, np.array([6., 6.]))
313: (8)              w2 = [[0, 0, 1], [1, 2, 3]]
314: (8)              avg, scl = average(y, weights=w2, axis=1, returned=True)
315: (8)              assert_array_equal(scl, np.array([1., 6.]))
316: (4)          def test_subclasses(self):
317: (8)              class subclass(np.ndarray):
318: (12)                  pass
319: (8)                  a = np.array([[1,2],[3,4]]).view(subclass)
320: (8)                  w = np.array([[1,2],[3,4]]).view(subclass)
321: (8)                  assert_equal(type(np.average(a)), subclass)
322: (8)                  assert_equal(type(np.average(a, weights=w)), subclass)
323: (4)          def test_upcasting(self):
324: (8)              typs = [('i4', 'i4', 'f8'), ('i4', 'f4', 'f8'), ('f4', 'i4', 'f8'),
325: (17)                  ('f4', 'f4', 'f4'), ('f4', 'f8', 'f8')]
326: (8)              for at, wt, rt in typs:
327: (12)                  a = np.array([[1,2],[3,4]], dtype=at)
328: (12)                  w = np.array([[1,2],[3,4]], dtype=wt)
329: (12)                  assert_equal(np.average(a, weights=w).dtype, np.dtype(rt))
330: (4)          def test_object_dtype(self):
331: (8)              a = np.array([decimal.Decimal(x) for x in range(10)])
332: (8)              w = np.array([decimal.Decimal(1) for _ in range(10)])
333: (8)              w /= w.sum()
334: (8)              assert_almost_equal(a.mean(0), average(a, weights=w))
335: (4)          def test_average_class_without_dtype(self):
336: (8)              a = np.array([Fraction(1, 5), Fraction(3, 5)])
337: (8)              assert_equal(np.average(a), Fraction(2, 5))
338: (0)          class TestSelect:
339: (4)              choices = [np.array([1, 2, 3]),
340: (15)                  np.array([4, 5, 6]),
341: (15)                  np.array([7, 8, 9])]
342: (4)              conditions = [np.array([False, False, False]),
343: (18)                  np.array([False, True, False]),
344: (18)                  np.array([False, False, True])]
345: (4)              def _select(self, cond, values, default=0):
346: (8)                  output = []
347: (8)                  for m in range(len(cond)):
348: (12)                      output += [V[m] for V, C in zip(values, cond) if C[m]] or
349: (8)                          return output
350: (4)          def test_basic(self):
351: (8)              choices = self.choices
352: (8)              conditions = self.conditions
353: (8)              assert_array_equal(select(conditions, choices, default=15),
354: (27)                  self._select(conditions, choices, default=15))
355: (8)              assert_equal(len(choices), 3)

```

```

356: (8)             assert_equal(len(conditions), 3)
357: (4)             def test_broadcasting(self):
358: (8)                 conditions = [np.array(True), np.array([False, True, False])]
359: (8)                 choices = [1, np.arange(12).reshape(4, 3)]
360: (8)                 assert_array_equal(select(conditions, choices), np.ones((4, 3)))
361: (8)                 assert_equal(select([True], [0], default=[0]).shape, (1,))
362: (4)             def test_return_dtype(self):
363: (8)                 assert_equal(select(self.conditions, self.choices, 1j).dtype,
364: (21)                         np.complex_)
365: (8)                 choices = [choice.astype(np.int8) for choice in self.choices]
366: (8)                 assert_equal(select(self.conditions, choices).dtype, np.int8)
367: (8)                 d = np.array([1, 2, 3, np.nan, 5, 7])
368: (8)                 m = np.isnan(d)
369: (8)                 assert_equal(select([m], [d]), [0, 0, 0, np.nan, 0, 0])
370: (4)             def test_DEPRECATED_empty(self):
371: (8)                 assert_raises(ValueError, select, [], [], 3j)
372: (8)                 assert_raises(ValueError, select, [], [])
373: (4)             def test_non_bool_deprecation(self):
374: (8)                 choices = self.choices
375: (8)                 conditions = self.conditions[:]
376: (8)                 conditions[0] = conditions[0].astype(np.int_)
377: (8)                 assert_raises(TypeError, select, conditions, choices)
378: (8)                 conditions[0] = conditions[0].astype(np.uint8)
379: (8)                 assert_raises(TypeError, select, conditions, choices)
380: (8)                 assert_raises(TypeError, select, conditions, choices)
381: (4)             def test_many_arguments(self):
382: (8)                 conditions = [np.array([False])] * 100
383: (8)                 choices = [np.array([1])] * 100
384: (8)                 select(conditions, choices)
385: (0)             class TestInsert:
386: (4)                 def test_basic(self):
387: (8)                     a = [1, 2, 3]
388: (8)                     assert_equal(insert(a, 0, 1), [1, 1, 2, 3])
389: (8)                     assert_equal(insert(a, 3, 1), [1, 2, 3, 1])
390: (8)                     assert_equal(insert(a, [1, 1, 1], [1, 2, 3]), [1, 1, 2, 3, 2, 3])
391: (8)                     assert_equal(insert(a, 1, [1, 2, 3]), [1, 1, 2, 3, 2, 3])
392: (8)                     assert_equal(insert(a, [1, -1, 3], 9), [1, 9, 2, 9, 3, 9])
393: (8)                     assert_equal(insert(a, slice(-1, None, -1), 9), [9, 1, 9, 2, 9, 3])
394: (8)                     assert_equal(insert(a, [-1, 1, 3], [7, 8, 9]), [1, 8, 2, 7, 3, 9])
395: (8)                     b = np.array([0, 1], dtype=np.float64)
396: (8)                     assert_equal(insert(b, 0, b[0]), [0., 0., 1.])
397: (8)                     assert_equal(insert(b, [], []), b)
398: (8)                     with warnings.catch_warnings(record=True) as w:
399: (12)                         warnings.filterwarnings('always', '', FutureWarning)
400: (12)                         assert_equal(
401: (16)                             insert(a, np.array([True] * 4), 9), [1, 9, 9, 9, 9, 2, 3])
402: (12)                         assert_(w[0].category is FutureWarning)
403: (4)             def test_multidim(self):
404: (8)                 a = [[1, 1, 1]]
405: (8)                 r = [[2, 2, 2],
406: (13)                     [1, 1, 1]]
407: (8)                 assert_equal(insert(a, 0, [1]), [1, 1, 1, 1])
408: (8)                 assert_equal(insert(a, 0, [2, 2, 2], axis=0), r)
409: (8)                 assert_equal(insert(a, 0, 2, axis=0), r)
410: (8)                 assert_equal(insert(a, 2, 2, axis=1), [[1, 1, 2, 1]])
411: (8)                 a = np.array([[1, 1], [2, 2], [3, 3]])
412: (8)                 b = np.arange(1, 4).repeat(3).reshape(3, 3)
413: (8)                 c = np.concatenate(
414: (12)                     (a[:, 0:1], np.arange(1, 4).repeat(3).reshape(3, 3).T,
415: (13)                         a[:, 1:2]), axis=1)
416: (8)                 assert_equal(insert(a, [1], [[1], [2], [3]]], axis=1), b)
417: (8)                 assert_equal(insert(a, [1], [1, 2, 3], axis=1), c)
418: (8)                 assert_equal(insert(a, 1, [1, 2, 3], axis=1), b)
419: (8)                 assert_equal(insert(a, 1, [[1], [2], [3]]], axis=1), c)
420: (8)                 a = np.arange(4).reshape(2, 2)
421: (8)                 assert_equal(insert(a[:, :1], 1, a[:, 1], axis=1), a)
422: (8)                 assert_equal(insert(a[:, :1], 1, a[1, :], axis=0), a)
423: (8)                 a = np.arange(24).reshape((2, 3, 4))
424: (8)                 assert_equal(insert(a, 1, a[:, :, 3], axis=-1),

```

```

425: (21)           insert(a, 1, a[:, :, 3], axis=2))
426: (8)            assert_equal(insert(a, 1, a[:, 2, :], axis=-2),
427: (21)                      insert(a, 1, a[:, 2, :], axis=1))
428: (8)            assert_raises(np.AxisError, insert, a, 1, a[:, 2, :, :], axis=3)
429: (8)            assert_raises(np.AxisError, insert, a, 1, a[:, 2, :, :], axis=-4)
430: (8)            a = np.arange(24).reshape((2, 3, 4))
431: (8)            assert_equal(insert(a, 1, a[:, :, 3], axis=-1),
432: (21)                      insert(a, 1, a[:, :, 3], axis=2))
433: (8)            assert_equal(insert(a, 1, a[:, 2, :], axis=-2),
434: (21)                      insert(a, 1, a[:, 2, :], axis=1))
435: (4)             def test_0d(self):
436: (8)               a = np.array(1)
437: (8)               with pytest.raises(np.AxisError):
438: (12)                 insert(a, [], 2, axis=0)
439: (8)               with pytest.raises(TypeError):
440: (12)                 insert(a, [], 2, axis="nonsense")
441: (4)             def test_subclass(self):
442: (8)               class SubClass(np.ndarray):
443: (12)                 pass
444: (8)               a = np.arange(10).view(SubClass)
445: (8)               assert_(isinstance(np.insert(a, 0, [0]), SubClass))
446: (8)               assert_(isinstance(np.insert(a, [], []), SubClass))
447: (8)               assert_(isinstance(np.insert(a, [0, 1], [1, 2]), SubClass))
448: (8)               assert_(isinstance(np.insert(a, slice(1, 2), [1, 2]), SubClass))
449: (8)               assert_(isinstance(np.insert(a, slice(1, -2, -1), []), SubClass))
450: (8)               a = np.array(1).view(SubClass)
451: (8)               assert_(isinstance(np.insert(a, 0, [0]), SubClass))
452: (4)             def test_index_array_copied(self):
453: (8)               x = np.array([1, 1, 1])
454: (8)               np.insert([0, 1, 2], x, [3, 4, 5])
455: (8)               assert_equal(x, np.array([1, 1, 1]))
456: (4)             def test_structured_array(self):
457: (8)               a = np.array([(1, 'a'), (2, 'b'), (3, 'c')], 
458: (21)                     dtype=[('foo', 'i'), ('bar', 'a1')])
459: (8)               val = (4, 'd')
460: (8)               b = np.insert(a, 0, val)
461: (8)               assert_array_equal(b[0], np.array(val, dtype=b.dtype))
462: (8)               val = [(4, 'd')] * 2
463: (8)               b = np.insert(a, [0, 2], val)
464: (8)               assert_array_equal(b[[0, 3]], np.array(val, dtype=b.dtype))
465: (4)             def test_index_floats(self):
466: (8)               with pytest.raises(IndexError):
467: (12)                 np.insert([0, 1, 2], np.array([1.0, 2.0]), [10, 20])
468: (8)               with pytest.raises(IndexError):
469: (12)                 np.insert([0, 1, 2], np.array([], dtype=float), [])
470: (4)               @pytest.mark.parametrize('idx', [4, -4])
471: (4)             def test_index_out_of_bounds(self, idx):
472: (8)               with pytest.raises(IndexError, match='out of bounds'):
473: (12)                 np.insert([0, 1, 2], [idx], [3, 4])
474: (0)             class TestAmax:
475: (4)               def test_basic(self):
476: (8)                 a = [3, 4, 5, 10, -3, -5, 6.0]
477: (8)                 assert_equal(np.amax(a), 10.0)
478: (8)                 b = [[3, 6.0, 9.0],
479: (13)                   [4, 10.0, 5.0],
480: (13)                   [8, 3.0, 2.0]]
481: (8)                 assert_equal(np.amax(b, axis=0), [8.0, 10.0, 9.0])
482: (8)                 assert_equal(np.amax(b, axis=1), [9.0, 10.0, 8.0])
483: (0)             class TestAmin:
484: (4)               def test_basic(self):
485: (8)                 a = [3, 4, 5, 10, -3, -5, 6.0]
486: (8)                 assert_equal(np.amin(a), -5.0)
487: (8)                 b = [[3, 6.0, 9.0],
488: (13)                   [4, 10.0, 5.0],
489: (13)                   [8, 3.0, 2.0]]
490: (8)                 assert_equal(np.amin(b, axis=0), [3.0, 3.0, 2.0])
491: (8)                 assert_equal(np.amin(b, axis=1), [3.0, 4.0, 2.0])
492: (0)             class TestPtp:
493: (4)               def test_basic(self):

```

```

494: (8)          a = np.array([3, 4, 5, 10, -3, -5, 6.0])
495: (8)          assert_equal(a.ptp(axis=0), 15.0)
496: (8)          b = np.array([[3, 6.0, 9.0],
497: (22)            [4, 10.0, 5.0],
498: (22)            [8, 3.0, 2.0]])
499: (8)          assert_equal(b.ptp(axis=0), [5.0, 7.0, 7.0])
500: (8)          assert_equal(b.ptp(axis=-1), [6.0, 6.0, 6.0])
501: (8)          assert_equal(b.ptp(axis=0, keepdims=True), [[5.0, 7.0, 7.0]])
502: (8)          assert_equal(b.ptp(axis=(0,1), keepdims=True), [[8.0]])
503: (0)          class TestCumsum:
504: (4)          def test_basic(self):
505: (8)              ba = [1, 2, 10, 11, 6, 5, 4]
506: (8)              ba2 = [[1, 2, 3, 4], [5, 6, 7, 9], [10, 3, 4, 5]]
507: (8)              for ctype in [np.int8, np.uint8, np.int16, np.uint16, np.int32,
508: (22)                np.uint32, np.float32, np.float64, np.complex64,
509: (22)                np.complex128]:
510: (12)              a = np.array(ba, ctype)
511: (12)              a2 = np.array(ba2, ctype)
512: (12)              tgt = np.array([1, 3, 13, 24, 30, 35, 39], ctype)
513: (12)              assert_array_equal(np.cumsum(a, axis=0), tgt)
514: (12)              tgt = np.array(
515: (16)                  [[1, 2, 3, 4], [6, 8, 10, 13], [16, 11, 14, 18]], ctype)
516: (12)              assert_array_equal(np.cumsum(a2, axis=0), tgt)
517: (12)              tgt = np.array(
518: (16)                  [[1, 3, 6, 10], [5, 11, 18, 27], [10, 13, 17, 22]], ctype)
519: (12)              assert_array_equal(np.cumsum(a2, axis=1), tgt)
520: (0)          class TestProd:
521: (4)          def test_basic(self):
522: (8)              ba = [1, 2, 10, 11, 6, 5, 4]
523: (8)              ba2 = [[1, 2, 3, 4], [5, 6, 7, 9], [10, 3, 4, 5]]
524: (8)              for ctype in [np.int16, np.uint16, np.int32, np.uint32,
525: (22)                np.float32, np.float64, np.complex64, np.complex128]:
526: (12)              a = np.array(ba, ctype)
527: (12)              a2 = np.array(ba2, ctype)
528: (12)              if ctype in ['1', 'b']:
529: (16)                  assert_raises(ArithmeticError, np.prod, a)
530: (16)                  assert_raises(ArithmeticError, np.prod, a2, 1)
531: (12)              else:
532: (16)                  assert_equal(a.prod(axis=0), 26400)
533: (16)                  assert_array_equal(a2.prod(axis=0),
534: (35)                      np.array([50, 36, 84, 180], ctype))
535: (16)                  assert_array_equal(a2.prod(axis=-1),
536: (35)                      np.array([24, 1890, 600], ctype))
537: (0)          class TestCumprod:
538: (4)          def test_basic(self):
539: (8)              ba = [1, 2, 10, 11, 6, 5, 4]
540: (8)              ba2 = [[1, 2, 3, 4], [5, 6, 7, 9], [10, 3, 4, 5]]
541: (8)              for ctype in [np.int16, np.uint16, np.int32, np.uint32,
542: (22)                np.float32, np.float64, np.complex64, np.complex128]:
543: (12)              a = np.array(ba, ctype)
544: (12)              a2 = np.array(ba2, ctype)
545: (12)              if ctype in ['1', 'b']:
546: (16)                  assert_raises(ArithmeticError, np.cumprod, a)
547: (16)                  assert_raises(ArithmeticError, np.cumprod, a2, 1)
548: (16)                  assert_raises(ArithmeticError, np.cumprod, a)
549: (12)              else:
550: (16)                  assert_array_equal(np.cumprod(a, axis=-1),
551: (35)                      np.array([1, 2, 20, 220,
552: (45)                          1320, 6600, 26400], ctype))
553: (16)                  assert_array_equal(np.cumprod(a2, axis=0),
554: (35)                      np.array([[1, 2, 3, 4],
555: (45)                          [5, 12, 21, 36],
556: (45)                          [50, 36, 84, 180]], ctype))
557: (16)                  assert_array_equal(np.cumprod(a2, axis=-1),
558: (35)                      np.array([[1, 2, 6, 24],
559: (45)                          [5, 30, 210, 1890],
560: (45)                          [10, 30, 120, 600]], ctype))
561: (0)          class TestDiff:
562: (4)          def test_basic(self):

```

```

563: (8)          x = [1, 4, 6, 7, 12]
564: (8)          out = np.array([3, 2, 1, 5])
565: (8)          out2 = np.array([-1, -1, 4])
566: (8)          out3 = np.array([0, 5])
567: (8)          assert_array_equal(diff(x), out)
568: (8)          assert_array_equal(diff(x, n=2), out2)
569: (8)          assert_array_equal(diff(x, n=3), out3)
570: (8)          x = [1.1, 2.2, 3.0, -0.2, -0.1]
571: (8)          out = np.array([1.1, 0.8, -3.2, 0.1])
572: (8)          assert_almost_equal(diff(x), out)
573: (8)          x = [True, True, False, False]
574: (8)          out = np.array([False, True, False])
575: (8)          out2 = np.array([True, True])
576: (8)          assert_array_equal(diff(x), out)
577: (8)          assert_array_equal(diff(x, n=2), out2)
578: (4)          def test_axis(self):
579: (8)          x = np.zeros((10, 20, 30))
580: (8)          x[:, 1::2, :] = 1
581: (8)          exp = np.ones((10, 19, 30))
582: (8)          exp[:, 1::2, :] = -1
583: (8)          assert_array_equal(diff(x), np.zeros((10, 20, 29)))
584: (8)          assert_array_equal(diff(x, axis=-1), np.zeros((10, 20, 29)))
585: (8)          assert_array_equal(diff(x, axis=0), np.zeros((9, 20, 30)))
586: (8)          assert_array_equal(diff(x, axis=1), exp)
587: (8)          assert_array_equal(diff(x, axis=-2), exp)
588: (8)          assert_raises(np.AxisError, diff, x, axis=3)
589: (8)          assert_raises(np.AxisError, diff, x, axis=-4)
590: (8)          x = np.array(1.111111111111111, np.float64)
591: (8)          assert_raises(ValueError, diff, x)
592: (4)          def test_nd(self):
593: (8)          x = 20 * rand(10, 20, 30)
594: (8)          out1 = x[:, :, 1:] - x[:, :, :-1]
595: (8)          out2 = out1[:, :, 1:] - out1[:, :, :-1]
596: (8)          out3 = x[1:, :, :] - x[:-1, :, :]
597: (8)          out4 = out3[1:, :, :] - out3[:-1, :, :]
598: (8)          assert_array_equal(diff(x), out1)
599: (8)          assert_array_equal(diff(x, n=2), out2)
600: (8)          assert_array_equal(diff(x, axis=0), out3)
601: (8)          assert_array_equal(diff(x, n=2, axis=0), out4)
602: (4)          def test_n(self):
603: (8)          x = list(range(3))
604: (8)          assert_raises(ValueError, diff, x, n=-1)
605: (8)          output = [diff(x, n=n) for n in range(1, 5)]
606: (8)          expected = [[1, 1], [0], [], []]
607: (8)          assert_(diff(x, n=0) is x)
608: (8)          for n, (expected, out) in enumerate(zip(expected, output), start=1):
609: (12)          assert_(type(out) is np.ndarray)
610: (12)          assert_array_equal(out, expected)
611: (12)          assert_equal(out.dtype, np.int_)
612: (12)          assert_equal(len(out), max(0, len(x) - n))
613: (4)          def test_times(self):
614: (8)          x = np.arange('1066-10-13', '1066-10-16', dtype=np.datetime64)
615: (8)          expected = [
616: (12)          np.array([1, 1], dtype='timedelta64[D']),
617: (12)          np.array([0], dtype='timedelta64[D']),
618: (8)          ]
619: (8)          expected.extend([np.array([], dtype='timedelta64[D'])] * 3)
620: (8)          for n, exp in enumerate(expected, start=1):
621: (12)          out = diff(x, n=n)
622: (12)          assert_array_equal(out, exp)
623: (12)          assert_equal(out.dtype, exp.dtype)
624: (4)          def test_subclass(self):
625: (8)          x = ma.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]],
626: (21)                      mask=[[False, False], [True, False],
627: (27)                          [False, True], [True, True], [False, False]])
628: (8)          out = diff(x)
629: (8)          assert_array_equal(out.data, [[1], [1], [1], [1], [1]])
630: (8)          assert_array_equal(out.mask, [[False], [True],
631: (38)                          [True], [True], [False]]))

```

```

632: (8)             assert_(type(out) is type(x))
633: (8)             out3 = diff(x, n=3)
634: (8)             assert_array_equal(out3.data, [[], [], [], [], []])
635: (8)             assert_array_equal(out3.mask, [[], [], [], [], []])
636: (8)             assert_(type(out3) is type(x))
637: (4)             def test_prepend(self):
638: (8)                 x = np.arange(5) + 1
639: (8)                 assert_array_equal(diff(x, prepend=0), np.ones(5))
640: (8)                 assert_array_equal(diff(x, prepend=[0]), np.ones(5))
641: (8)                 assert_array_equal(np.cumsum(np.diff(x, prepend=0)), x)
642: (8)                 assert_array_equal(diff(x, prepend=[-1, 0]), np.ones(6))
643: (8)                 x = np.arange(4).reshape(2, 2)
644: (8)                 result = np.diff(x, axis=1, prepend=0)
645: (8)                 expected = [[0, 1], [2, 1]]
646: (8)                 assert_array_equal(result, expected)
647: (8)                 result = np.diff(x, axis=1, prepend=[[0], [0]])
648: (8)                 assert_array_equal(result, expected)
649: (8)                 result = np.diff(x, axis=0, prepend=0)
650: (8)                 expected = [[0, 1], [2, 2]]
651: (8)                 assert_array_equal(result, expected)
652: (8)                 result = np.diff(x, axis=0, prepend=[[0, 0]])
653: (8)                 assert_array_equal(result, expected)
654: (8)                 assert_raises(ValueError, np.diff, x, prepend=np.zeros((3,3)))
655: (8)                 assert_raises(np.AxisError, diff, x, prepend=0, axis=3)
656: (4)             def test_append(self):
657: (8)                 x = np.arange(5)
658: (8)                 result = diff(x, append=0)
659: (8)                 expected = [1, 1, 1, 1, -4]
660: (8)                 assert_array_equal(result, expected)
661: (8)                 result = diff(x, append=[0])
662: (8)                 assert_array_equal(result, expected)
663: (8)                 result = diff(x, append=[0, 2])
664: (8)                 expected = expected + [2]
665: (8)                 assert_array_equal(result, expected)
666: (8)                 x = np.arange(4).reshape(2, 2)
667: (8)                 result = np.diff(x, axis=1, append=0)
668: (8)                 expected = [[1, -1], [1, -3]]
669: (8)                 assert_array_equal(result, expected)
670: (8)                 result = np.diff(x, axis=1, append=[[0], [0]])
671: (8)                 assert_array_equal(result, expected)
672: (8)                 result = np.diff(x, axis=0, append=0)
673: (8)                 expected = [[2, 2], [-2, -3]]
674: (8)                 assert_array_equal(result, expected)
675: (8)                 result = np.diff(x, axis=0, append=[[0, 0]])
676: (8)                 assert_array_equal(result, expected)
677: (8)                 assert_raises(ValueError, np.diff, x, append=np.zeros((3,3)))
678: (8)                 assert_raises(np.AxisError, diff, x, append=0, axis=3)
679: (0)             class TestDelete:
680: (4)                 def setup_method(self):
681: (8)                     self.a = np.arange(5)
682: (8)                     self.nd_a = np.arange(5).repeat(2).reshape(1, 5, 2)
683: (4)                 def _check_inverse_of_slicing(self, indices):
684: (8)                     a_del = delete(self.a, indices)
685: (8)                     nd_a_del = delete(self.nd_a, indices, axis=1)
686: (8)                     msg = 'Delete failed for obj: %r' % indices
687: (8)                     assert_array_equal(setxor1d(a_del, self.a[indices, ]), self.a,
688: (27)                                     err_msg=msg)
689: (8)                     xor = setxor1d(nd_a_del[0, :, 0], self.nd_a[0, indices, 0])
690: (8)                     assert_array_equal(xor, self.nd_a[0, :, 0], err_msg=msg)
691: (4)                 def test_slices(self):
692: (8)                     lims = [-6, -2, 0, 1, 2, 4, 5]
693: (8)                     steps = [-3, -1, 1, 3]
694: (8)                     for start in lims:
695: (12)                         for stop in lims:
696: (16)                             for step in steps:
697: (20)                                 s = slice(start, stop, step)
698: (20)                                 self._check_inverse_of_slicing(s)
699: (4)                 def test_fancy(self):
700: (8)                     self._check_inverse_of_slicing(np.array([[0, 1], [2, 1]]))

```

```

701: (8)           with pytest.raises(IndexError):
702: (12)          delete(self.a, [100])
703: (8)          with pytest.raises(IndexError):
704: (12)          delete(self.a, [-100])
705: (8)          self._check_inverse_of_slicing([0, -1, 2, 2])
706: (8)          self._check_inverse_of_slicing([True, False, False, True, False])
707: (8)          with pytest.raises(ValueError):
708: (12)          delete(self.a, True)
709: (8)          with pytest.raises(ValueError):
710: (12)          delete(self.a, False)
711: (8)          with pytest.raises(ValueError):
712: (12)          delete(self.a, [False]*4)
713: (4)           def test_single(self):
714: (8)             self._check_inverse_of_slicing(0)
715: (8)             self._check_inverse_of_slicing(-4)
716: (4)           def test_0d(self):
717: (8)             a = np.array(1)
718: (8)             with pytest.raises(np.AxisError):
719: (12)               delete(a, [], axis=0)
720: (8)             with pytest.raises(TypeError):
721: (12)               delete(a, [], axis="nonsense")
722: (4)           def test_subclass(self):
723: (8)             class SubClass(np.ndarray):
724: (12)               pass
725: (8)             a = self.a.view(SubClass)
726: (8)             assert_(isinstance(delete(a, 0), SubClass))
727: (8)             assert_(isinstance(delete(a, []), SubClass))
728: (8)             assert_(isinstance(delete(a, [0, 1]), SubClass))
729: (8)             assert_(isinstance(delete(a, slice(1, 2)), SubClass))
730: (8)             assert_(isinstance(delete(a, slice(1, -2)), SubClass))
731: (4)           def test_array_order_preserve(self):
732: (8)             k = np.arange(10).reshape(2, 5, order='F')
733: (8)             m = delete(k, slice(60, None), axis=1)
734: (8)             assert_equal(m.flags.c_contiguous, k.flags.c_contiguous)
735: (8)             assert_equal(m.flags.f_contiguous, k.flags.f_contiguous)
736: (4)           def test_index_floats(self):
737: (8)             with pytest.raises(IndexError):
738: (12)               np.delete([0, 1, 2], np.array([1.0, 2.0]))
739: (8)             with pytest.raises(IndexError):
740: (12)               np.delete([0, 1, 2], np.array([], dtype=float))
741: (4)           @pytest.mark.parametrize("indexer", [np.array([1]), [1]])
742: (4)           def test_single_item_array(self, indexer):
743: (8)             a_del_int = delete(self.a, 1)
744: (8)             a_del = delete(self.a, indexer)
745: (8)             assert_equal(a_del_int, a_del)
746: (8)             nd_a_del_int = delete(self.nd_a, 1, axis=1)
747: (8)             nd_a_del = delete(self.nd_a, np.array([1]), axis=1)
748: (8)             assert_equal(nd_a_del_int, nd_a_del)
749: (4)           def test_single_item_array_non_int(self):
750: (8)             res = delete(np.ones(1), np.array([False]))
751: (8)             assert_array_equal(res, np.ones(1))
752: (8)             x = np.ones((3, 1))
753: (8)             false_mask = np.array([False], dtype=bool)
754: (8)             true_mask = np.array([True], dtype=bool)
755: (8)             res = delete(x, false_mask, axis=-1)
756: (8)             assert_array_equal(res, x)
757: (8)             res = delete(x, true_mask, axis=-1)
758: (8)             assert_array_equal(res, x[:, :0])
759: (8)             with pytest.raises(IndexError):
760: (12)               delete(np.ones(2), np.array([0], dtype=object))
761: (8)             with pytest.raises(IndexError):
762: (12)               delete(np.ones(2), np.array([0], dtype="m8[ns]"))
763: (0)           class TestGradient:
764: (4)             def test_basic(self):
765: (8)               v = [[1, 1], [3, 4]]
766: (8)               x = np.array(v)
767: (8)               dx = [np.array([[2., 3.], [2., 3.]]),
768: (14)                 np.array([[0., 0.], [1., 1.]])]
769: (8)               assert_array_equal(gradient(x), dx)

```

```

770: (8)             assert_array_equal(gradient(v), dx)
771: (4) def test_args(self):
772: (8)             dx = np.cumsum(np.ones(5))
773: (8)             dx_uneven = [1., 2., 5., 9., 11.]
774: (8)             f_2d = np.arange(25).reshape(5, 5)
775: (8)             gradient(np.arange(5), 3.)
776: (8)             gradient(np.arange(5), np.array(3.))
777: (8)             gradient(np.arange(5), dx)
778: (8)             gradient(f_2d, 1.5)
779: (8)             gradient(f_2d, np.array(1.5))
780: (8)             gradient(f_2d, dx_uneven, dx_uneven)
781: (8)             gradient(f_2d, dx, 2)
782: (8)             gradient(f_2d, dx, axis=1)
783: (8)             assert_raises_regex(ValueError, '.*scalars or 1d',
784: (12)                 gradient, f_2d, np.stack([dx]*2, axis=-1), 1)
785: (4) def test_badargs(self):
786: (8)             f_2d = np.arange(25).reshape(5, 5)
787: (8)             x = np.cumsum(np.ones(5))
788: (8)             assert_raises(ValueError, gradient, f_2d, x, np.ones(2))
789: (8)             assert_raises(ValueError, gradient, f_2d, 1, np.ones(2))
790: (8)             assert_raises(ValueError, gradient, f_2d, np.ones(2), np.ones(2))
791: (8)             assert_raises(TypeError, gradient, f_2d, x)
792: (8)             assert_raises(TypeError, gradient, f_2d, x, axis=(0,1))
793: (8)             assert_raises(TypeError, gradient, f_2d, x, x, x)
794: (8)             assert_raises(TypeError, gradient, f_2d, 1, 1, 1)
795: (8)             assert_raises(TypeError, gradient, f_2d, x, x, axis=1)
796: (8)             assert_raises(TypeError, gradient, f_2d, 1, 1, axis=1)
797: (4) def test_datetime64(self):
798: (8)             x = np.array(
799: (12)                 ['1910-08-16', '1910-08-11', '1910-08-10', '1910-08-12',
800: (13)                 '1910-10-12', '1910-12-12', '1912-12-12'],
801: (12)                 dtype='datetime64[D]')
802: (8)             dx = np.array(
803: (12)                 [-5, -3, 0, 31, 61, 396, 731],
804: (12)                 dtype='timedelta64[D]')
805: (8)             assert_array_equal(gradient(x), dx)
806: (8)             assert_(dx.dtype == np.dtype('timedelta64[D]'))
807: (4) def test_masked(self):
808: (8)             x = np.ma.array([[1, 1], [3, 4]],
809: (24)                 mask=[[False, False], [False, False]])
810: (8)             out = gradient(x)[0]
811: (8)             assert_equal(type(out), type(x))
812: (8)             assert_(x._mask is not out._mask)
813: (8)             x2 = np.ma.arange(5)
814: (8)             x2[2] = np.ma.masked
815: (8)             np.gradient(x2, edge_order=2)
816: (8)             assert_array_equal(x2.mask, [False, False, True, False, False])
817: (4) def test_second_order_accurate(self):
818: (8)             x = np.linspace(0, 1, 10)
819: (8)             dx = x[1] - x[0]
820: (8)             y = 2 * x ** 3 + 4 * x ** 2 + 2 * x
821: (8)             analytical = 6 * x ** 2 + 8 * x + 2
822: (8)             num_error = np.abs((np.gradient(y, dx, edge_order=2) / analytical) -
1)
823: (8)             assert_(np.all(num_error < 0.03) == True)
824: (8)             np.random.seed(0)
825: (8)             x = np.sort(np.random.random(10))
826: (8)             y = 2 * x ** 3 + 4 * x ** 2 + 2 * x
827: (8)             analytical = 6 * x ** 2 + 8 * x + 2
828: (8)             num_error = np.abs((np.gradient(y, x, edge_order=2) / analytical) - 1)
829: (8)             assert_(np.all(num_error < 0.03) == True)
830: (4) def test_spacing(self):
831: (8)             f = np.array([0, 2., 3., 4., 5., 5.])
832: (8)             f = np.tile(f, (6,1)) + f.reshape(-1, 1)
833: (8)             x_uneven = np.array([0., 0.5, 1., 3., 5., 7.])
834: (8)             x_even = np.arange(6.)
835: (8)             fdx_even_ord1 = np.tile([2., 1.5, 1., 1., 0.5, 0.], (6,1))
836: (8)             fdx_even_ord2 = np.tile([2.5, 1.5, 1., 1., 0.5, -0.5], (6,1))
837: (8)             fdx_uneven_ord1 = np.tile([4., 3., 1.7, 0.5, 0.25, 0.], (6,1))

```

```

838: (8)             fdx_uneven_ord2 = np.tile([5., 3., 1.7, 0.5, 0.25, -0.25], (6,1))
839: (8)             for edge_order, exp_res in [(1, fdx_even_ord1), (2, fdx_uneven_ord2)]:
840: (12)                 res1 = gradient(f, 1., axis=(0,1), edge_order=edge_order)
841: (12)                 res2 = gradient(f, x_even, x_even,
842: (28)                             axis=(0,1), edge_order=edge_order)
843: (12)                 res3 = gradient(f, x_even, x_even,
844: (28)                             axis=None, edge_order=edge_order)
845: (12)             assert_array_equal(res1, res2)
846: (12)             assert_array_equal(res2, res3)
847: (12)             assert_almost_equal(res1[0], exp_res.T)
848: (12)             assert_almost_equal(res1[1], exp_res)
849: (12)             res1 = gradient(f, 1., axis=0, edge_order=edge_order)
850: (12)             res2 = gradient(f, x_even, axis=0, edge_order=edge_order)
851: (12)             assert_(res1.shape == res2.shape)
852: (12)             assert_almost_equal(res2, exp_res.T)
853: (12)             res1 = gradient(f, 1., axis=1, edge_order=edge_order)
854: (12)             res2 = gradient(f, x_even, axis=1, edge_order=edge_order)
855: (12)             assert_(res1.shape == res2.shape)
856: (12)             assert_array_equal(res2, exp_res)
857: (8)             for edge_order, exp_res in [(1, fdx_uneven_ord1), (2,
fdx_uneven_ord2)]:
858: (12)                 res1 = gradient(f, x_uneven, x_uneven,
859: (28)                             axis=(0,1), edge_order=edge_order)
860: (12)                 res2 = gradient(f, x_uneven, x_uneven,
861: (28)                             axis=None, edge_order=edge_order)
862: (12)             assert_array_equal(res1, res2)
863: (12)             assert_almost_equal(res1[0], exp_res.T)
864: (12)             assert_almost_equal(res1[1], exp_res)
865: (12)             res1 = gradient(f, x_uneven, axis=0, edge_order=edge_order)
866: (12)             assert_almost_equal(res1, exp_res.T)
867: (12)             res1 = gradient(f, x_uneven, axis=1, edge_order=edge_order)
868: (12)             assert_almost_equal(res1, exp_res)
869: (8)             res1 = gradient(f, x_even, x_uneven, axis=(0,1), edge_order=1)
870: (8)             res2 = gradient(f, x_uneven, x_even, axis=(1,0), edge_order=1)
871: (8)             assert_array_equal(res1[0], res2[1])
872: (8)             assert_array_equal(res1[1], res2[0])
873: (8)             assert_almost_equal(res1[0], fdx_even_ord1.T)
874: (8)             assert_almost_equal(res1[1], fdx_uneven_ord1)
875: (8)             res1 = gradient(f, x_even, x_uneven, axis=(0,1), edge_order=2)
876: (8)             res2 = gradient(f, x_uneven, x_even, axis=(1,0), edge_order=2)
877: (8)             assert_array_equal(res1[0], res2[1])
878: (8)             assert_array_equal(res1[1], res2[0])
879: (8)             assert_almost_equal(res1[0], fdx_even_ord2.T)
880: (8)             assert_almost_equal(res1[1], fdx_uneven_ord2)
881: (4)             def test_specific_axes(self):
882: (8)                 v = [[1, 1], [3, 4]]
883: (8)                 x = np.array(v)
884: (8)                 dx = [np.array([[2., 3.], [2., 3.]]),
885: (14)                               np.array([[0., 0.], [1., 1.]])]
886: (8)                 assert_array_equal(gradient(x, axis=0), dx[0])
887: (8)                 assert_array_equal(gradient(x, axis=1), dx[1])
888: (8)                 assert_array_equal(gradient(x, axis=-1), dx[1])
889: (8)                 assert_array_equal(gradient(x, axis=(1, 0)), [dx[1], dx[0]])
890: (8)                 assert_almost_equal(gradient(x, axis=None), [dx[0], dx[1]])
891: (8)                 assert_almost_equal(gradient(x, axis=None), gradient(x))
892: (8)                 assert_array_equal(gradient(x, 2, 3, axis=(1, 0)),
893: (27)                               [dx[1]/2.0, dx[0]/3.0])
894: (8)                 assert_raises(TypeError, gradient, x, 1, 2, axis=1)
895: (8)                 assert_raises(np.AxisError, gradient, x, axis=3)
896: (8)                 assert_raises(np.AxisError, gradient, x, axis=-3)
897: (4)             def test_timedelta64(self):
898: (8)                 x = np.array(
899: (12)                     [-5, -3, 10, 12, 61, 321, 300],
900: (12)                     dtype='timedelta64[D]')
901: (8)                 dx = np.array(
902: (12)                     [2, 7, 7, 25, 154, 119, -21],
903: (12)                     dtype='timedelta64[D]')
904: (8)                 assert_array_equal(gradient(x), dx)
905: (8)                 assert_(dx.dtype == np.dtype('timedelta64[D]'))

```

```

906: (4) def test_inexact_dtypes(self):
907: (8)     for dt in [np.float16, np.float32, np.float64]:
908: (12)         x = np.array([1, 2, 3], dtype=dt)
909: (12)         assert_equal(gradient(x).dtype, np.diff(x).dtype)
910: (4) def test_values(self):
911: (8)     gradient(np.arange(2), edge_order=1)
912: (8)     gradient(np.arange(3), edge_order=2)
913: (8)     assert_raises(ValueError, gradient, np.arange(0), edge_order=1)
914: (8)     assert_raises(ValueError, gradient, np.arange(0), edge_order=2)
915: (8)     assert_raises(ValueError, gradient, np.arange(1), edge_order=1)
916: (8)     assert_raises(ValueError, gradient, np.arange(1), edge_order=2)
917: (8)     assert_raises(ValueError, gradient, np.arange(2), edge_order=2)
918: (4) @pytest.mark.parametrize('f_dtype', [np.uint8, np.uint16,
919: (41)                           np.uint32, np.uint64])
920: (4) def test_f_decreasing_unsigned_int(self, f_dtype):
921: (8)     f = np.array([5, 4, 3, 2, 1], dtype=f_dtype)
922: (8)     g = gradient(f)
923: (8)     assert_array_equal(g, [-1]*len(f))
924: (4) @pytest.mark.parametrize('f_dtype', [np.int8, np.int16,
925: (41)                           np.int32, np.int64])
926: (4) def test_f_signed_int_big_jump(self, f_dtype):
927: (8)     maxint = np.iinfo(f_dtype).max
928: (8)     x = np.array([1, 3])
929: (8)     f = np.array([-1, maxint], dtype=f_dtype)
930: (8)     dfdx = gradient(f, x)
931: (8)     assert_array_equal(dfdx, [(maxint + 1) // 2]**2)
932: (4) @pytest.mark.parametrize('x_dtype', [np.uint8, np.uint16,
933: (41)                           np.uint32, np.uint64])
934: (4) def test_x_decreasing_unsigned(self, x_dtype):
935: (8)     x = np.array([3, 2, 1], dtype=x_dtype)
936: (8)     f = np.array([0, 2, 4])
937: (8)     dfdx = gradient(f, x)
938: (8)     assert_array_equal(dfdx, [-2]**len(x))
939: (4) @pytest.mark.parametrize('x_dtype', [np.int8, np.int16,
940: (41)                           np.int32, np.int64])
941: (4) def test_x_signed_int_big_jump(self, x_dtype):
942: (8)     minint = np.iinfo(x_dtype).min
943: (8)     maxint = np.iinfo(x_dtype).max
944: (8)     x = np.array([-1, maxint], dtype=x_dtype)
945: (8)     f = np.array([minint // 2, 0])
946: (8)     dfdx = gradient(f, x)
947: (8)     assert_array_equal(dfdx, [0.5, 0.5])
948: (4) def test_return_type(self):
949: (8)     res = np.gradient(([1, 2], [2, 3]))
950: (8)     if np._using_numpy2_behavior():
951: (12)         assert type(res) is tuple
952: (8)     else:
953: (12)         assert type(res) is list
954: (0) class TestAngle:
955: (4)     def test_basic(self):
956: (8)         x = [1 + 3j, np.sqrt(2) / 2.0 + 1j * np.sqrt(2) / 2,
957: (13)             1, 1j, -1, -1j, 1 - 3j, -1 + 3j]
958: (8)         y = angle(x)
959: (8)         yo = [
960: (12)             np.arctan(3.0 / 1.0),
961: (12)             np.arctan(1.0), 0, np.pi / 2, np.pi, -np.pi / 2.0,
962: (12)             -np.arctan(3.0 / 1.0), np.pi - np.arctan(3.0 / 1.0)]
963: (8)         z = angle(x, deg=True)
964: (8)         zo = np.array(yo) * 180 / np.pi
965: (8)         assert_array_almost_equal(y, yo, 11)
966: (8)         assert_array_almost_equal(z, zo, 11)
967: (4)     def test_subclass(self):
968: (8)         x = np.ma.array([1 + 3j, 1, np.sqrt(2)/2 * (1 + 1j)])
969: (8)         x[1] = np.ma.masked
970: (8)         expected = np.ma.array([np.arctan(3.0 / 1.0), 0, np.arctan(1.0)])
971: (8)         expected[1] = np.ma.masked
972: (8)         actual = angle(x)
973: (8)         assert_equal(type(actual), type(expected))
974: (8)         assert_equal(actual.mask, expected.mask)

```

```

975: (8)             assert_equal(actual, expected)
976: (0)
977: (4)
978: (4)
979: (4)
980: (4)
981: (4)
982: (8)
983: (8)
984: (4)
985: (8)
986: (8)
987: (12)
988: (12)
989: (4)
990: (8)
991: (8)
992: (12)
993: (12)
994: (4)
995: (8)
996: (8)
997: (12)
998: (12)
999: (4)
1000: (8)
1001: (12)
1002: (12)
1003: (12)
1004: (12)
1005: (12)
1006: (4)
1007: (8)
1008: (8)
1009: (8)
1010: (4)
1011: (8)
1012: (8)
1013: (9)
1014: (9)
1015: (4)
1016: (4)
1017: (8)
1018: (8)
1019: (8)
1020: (4)
1021: (8)
1022: (8)
1023: (8)
1024: (4)
1025: (8)
1026: (8)
1027: (0)
1028: (4)
1029: (8)
1030: (8)
1031: (8)
1032: (4)
1033: (8)
1034: (8)
1035: (8)
1036: (8)
1037: (8)
1038: (8)
1039: (8)
1040: (8)
1041: (8)
1042: (28)
1043: (8)

         assert_equal(actual, expected)
class TestTrimZeros:
    a = np.array([0, 0, 1, 0, 2, 3, 4, 0])
    b = a.astype(float)
    c = a.astype(complex)
    d = a.astype(object)
    def values(self):
        attr_names = ('a', 'b', 'c', 'd')
        return (getattr(self, name) for name in attr_names)
    def test_basic(self):
        slc = np.s_[2:-1]
        for arr in self.values():
            res = trim_zeros(arr)
            assert_array_equal(res, arr[slc])
    def test_leading_skip(self):
        slc = np.s_[:-1]
        for arr in self.values():
            res = trim_zeros(arr, trim='b')
            assert_array_equal(res, arr[slc])
    def test_trailing_skip(self):
        slc = np.s_[2:]
        for arr in self.values():
            res = trim_zeros(arr, trim='F')
            assert_array_equal(res, arr[slc])
    def test_all_zero(self):
        for _arr in self.values():
            arr = np.zeros_like(_arr, dtype=_arr.dtype)
            res1 = trim_zeros(arr, trim='B')
            assert len(res1) == 0
            res2 = trim_zeros(arr, trim='f')
            assert len(res2) == 0
    def test_size_zero(self):
        arr = np.zeros(0)
        res = trim_zeros(arr)
        assert_array_equal(arr, res)
    @pytest.mark.parametrize(
        'arr',
        [np.array([0, 2**62, 0]),
         np.array([0, 2**63, 0]),
         np.array([0, 2**64, 0])
    )
    def test_overflow(self, arr):
        slc = np.s_[1:2]
        res = trim_zeros(arr)
        assert_array_equal(res, arr[slc])
    def test_no_trim(self):
        arr = np.array([None, 1, None])
        res = trim_zeros(arr)
        assert_array_equal(arr, res)
    def test_list_to_list(self):
        res = trim_zeros(self.a.tolist())
        assert isinstance(res, list)
class TestExtins:
    def test_basic(self):
        a = np.array([1, 3, 2, 1, 2, 3, 3])
        b = extract(a > 1, a)
        assert_array_equal(b, [3, 2, 2, 3, 3])
    def test_place(self):
        assert_raises(TypeError, place, [1, 2, 3], [True, False], [0, 1])
        a = np.array([1, 4, 3, 2, 5, 8, 7])
        place(a, [0, 1, 0, 1, 0, 1, 0], [2, 4, 6])
        assert_array_equal(a, [1, 2, 3, 4, 5, 6, 7])
        place(a, np.zeros(7), [])
        assert_array_equal(a, np.arange(1, 8))
        place(a, [1, 0, 1, 0, 1, 0, 1], [8, 9])
        assert_array_equal(a, [8, 2, 9, 4, 8, 6, 9])
        assert_raises_regex(ValueError, "Cannot insert from an empty array",
                            lambda: place(a, [0, 0, 0, 0, 0, 1, 0], []))
        a = np.array(['12', '34'])

```

```

1044: (8)           place(a, [0, 1], '9')
1045: (8)           assert_array_equal(a, ['12', '9'])
1046: (4)           def test_both(self):
1047: (8)             a = rand(10)
1048: (8)             mask = a > 0.5
1049: (8)             ac = a.copy()
1050: (8)             c = extract(mask, a)
1051: (8)             place(a, mask, 0)
1052: (8)             place(a, mask, c)
1053: (8)             assert_array_equal(a, ac)
1054: (0)           def _foo1(x, y=1.0):
1055: (4)             return y*math.floor(x)
1056: (0)           def _foo2(x, y=1.0, z=0.0):
1057: (4)             return y*math.floor(x) + z
1058: (0)           class TestVectorize:
1059: (4)             def test_simple(self):
1060: (8)               def addsubtract(a, b):
1061: (12)                 if a > b:
1062: (16)                   return a - b
1063: (12)                 else:
1064: (16)                     return a + b
1065: (8)               f = vectorize(addsubtract)
1066: (8)               r = f([0, 3, 6, 9], [1, 3, 5, 7])
1067: (8)               assert_array_equal(r, [1, 6, 1, 2])
1068: (4)           def test_scalar(self):
1069: (8)             def addsubtract(a, b):
1070: (12)               if a > b:
1071: (16)                 return a - b
1072: (12)               else:
1073: (16)                 return a + b
1074: (8)             f = vectorize(addsubtract)
1075: (8)             r = f([0, 3, 6, 9], 5)
1076: (8)             assert_array_equal(r, [5, 8, 1, 4])
1077: (4)           def test_large(self):
1078: (8)             x = np.linspace(-3, 2, 10000)
1079: (8)             f = vectorize(lambda x: x)
1080: (8)             y = f(x)
1081: (8)             assert_array_equal(y, x)
1082: (4)           def test_ufunc(self):
1083: (8)             f = vectorize(math.cos)
1084: (8)             args = np.array([0, 0.5 * np.pi, np.pi, 1.5 * np.pi, 2 * np.pi])
1085: (8)             r1 = f(args)
1086: (8)             r2 = np.cos(args)
1087: (8)             assert_array_almost_equal(r1, r2)
1088: (4)           def test_keywords(self):
1089: (8)             def foo(a, b=1):
1090: (12)               return a + b
1091: (8)             f = vectorize(foo)
1092: (8)             args = np.array([1, 2, 3])
1093: (8)             r1 = f(args)
1094: (8)             r2 = np.array([2, 3, 4])
1095: (8)             assert_array_equal(r1, r2)
1096: (8)             r1 = f(args, 2)
1097: (8)             r2 = np.array([3, 4, 5])
1098: (8)             assert_array_equal(r1, r2)
1099: (4)           def test_keywords_with_otypes_order1(self):
1100: (8)             f = vectorize(_foo1, otypes=[float])
1101: (8)             r1 = f(np.arange(3.0), 1.0)
1102: (8)             r2 = f(np.arange(3.0))
1103: (8)             assert_array_equal(r1, r2)
1104: (4)           def test_keywords_with_otypes_order2(self):
1105: (8)             f = vectorize(_foo1, otypes=[float])
1106: (8)             r1 = f(np.arange(3.0))
1107: (8)             r2 = f(np.arange(3.0), 1.0)
1108: (8)             assert_array_equal(r1, r2)
1109: (4)           def test_keywords_with_otypes_order3(self):
1110: (8)             f = vectorize(_foo1, otypes=[float])
1111: (8)             r1 = f(np.arange(3.0))
1112: (8)             r2 = f(np.arange(3.0), y=1.0)

```

```

1113: (8)          r3 = f(np.arange(3.0))
1114: (8)          assert_array_equal(r1, r2)
1115: (8)          assert_array_equal(r1, r3)
1116: (4)          def test_keywords_with_otypes_several_kwds1(self):
1117: (8)              f = vectorize(_foo2, otypes=[float])
1118: (8)              r1 = f(10.4, z=100)
1119: (8)              r2 = f(10.4, y=-1)
1120: (8)              r3 = f(10.4)
1121: (8)              assert_equal(r1, _foo2(10.4, z=100))
1122: (8)              assert_equal(r2, _foo2(10.4, y=-1))
1123: (8)              assert_equal(r3, _foo2(10.4))
1124: (4)          def test_keywords_with_otypes_several_kwds2(self):
1125: (8)              f = vectorize(_foo2, otypes=[float])
1126: (8)              r1 = f(z=100, x=10.4, y=-1)
1127: (8)              r2 = f(1, 2, 3)
1128: (8)              assert_equal(r1, _foo2(z=100, x=10.4, y=-1))
1129: (8)              assert_equal(r2, _foo2(1, 2, 3))
1130: (4)          def test_keywords_no_func_code(self):
1131: (8)              import random
1132: (8)              try:
1133: (12)                  vectorize(random.randrange) # Should succeed
1134: (8)              except Exception:
1135: (12)                  raise AssertionError()
1136: (4)          def test_keywords2_ticket_2100(self):
1137: (8)              def foo(a, b=1):
1138: (12)                  return a + b
1139: (8)              f = vectorize(foo)
1140: (8)              args = np.array([1, 2, 3])
1141: (8)              r1 = f(a=args)
1142: (8)              r2 = np.array([2, 3, 4])
1143: (8)              assert_array_equal(r1, r2)
1144: (8)              r1 = f(b=1, a=args)
1145: (8)              assert_array_equal(r1, r2)
1146: (8)              r1 = f(args, b=2)
1147: (8)              r2 = np.array([3, 4, 5])
1148: (8)              assert_array_equal(r1, r2)
1149: (4)          def test_keywords3_ticket_2100(self):
1150: (8)              def mypolyval(x, p):
1151: (12)                  _p = list(p)
1152: (12)                  res = _p.pop(0)
1153: (12)                  while _p:
1154: (16)                      res = res * x + _p.pop(0)
1155: (12)                  return res
1156: (8)              vpolyval = np.vectorize(mypolyval, excluded=['p', 1])
1157: (8)              ans = [3, 6]
1158: (8)              assert_array_equal(ans, vpolyval(x=[0, 1], p=[1, 2, 3]))
1159: (8)              assert_array_equal(ans, vpolyval([0, 1], p=[1, 2, 3]))
1160: (8)              assert_array_equal(ans, vpolyval([0, 1], [1, 2, 3]))
1161: (4)          def test_keywords4_ticket_2100(self):
1162: (8)              @vectorize
1163: (8)              def f(**kw):
1164: (12)                  res = 1.0
1165: (12)                  for _k in kw:
1166: (16)                      res *= kw[_k]
1167: (12)                  return res
1168: (8)              assert_array_equal(f(a=[1, 2], b=[3, 4]), [3, 8])
1169: (4)          def test_keywords5_ticket_2100(self):
1170: (8)              @vectorize
1171: (8)              def f(*v):
1172: (12)                  return np.prod(v)
1173: (8)              assert_array_equal(f([1, 2], [3, 4]), [3, 8])
1174: (4)          def test_coverage1_ticket_2100(self):
1175: (8)              def foo():
1176: (12)                  return 1
1177: (8)              f = vectorize(foo)
1178: (8)              assert_array_equal(f(), 1)
1179: (4)          def test_assigning_docstring(self):
1180: (8)              def foo(x):
1181: (12)                  """Original documentation"""

```

```

1182: (12)
1183: (8)         return x
1184: (8)         f = vectorize(foo)
1185: (8)         assert_equal(f.__doc__, foo.__doc__)
1186: (8)         doc = "Provided documentation"
1187: (8)         f = vectorize(foo, doc=doc)
1188: (4)         assert_equal(f.__doc__, doc)
1189: (8)     def test_UnboundMethod_ticket_1156(self):
1190: (12)         class Foo:
1191: (12)             b = 2
1192: (16)             def bar(self, a):
1193: (8)                 return a ** self.b
1194: (27)             assert_array_equal(vectorize(Foo().bar)(np.arange(9)),
1195: (8)                         np.arange(9) ** 2)
1196: (27)             assert_array_equal(vectorize(Foo.bar)(Foo(), np.arange(9)),
1197: (4)                         np.arange(9) ** 2)
1198: (8)     def test_execution_order_ticket_1487(self):
1199: (8)         f1 = vectorize(lambda x: x)
1200: (8)         res1a = f1(np.arange(3))
1201: (8)         res1b = f1(np.arange(0.1, 3))
1202: (8)         f2 = vectorize(lambda x: x)
1203: (8)         res2b = f2(np.arange(0.1, 3))
1204: (8)         res2a = f2(np.arange(3))
1205: (8)         assert_equal(res1a, res2a)
1206: (4)         assert_equal(res1b, res2b)
1207: (8)     def test_string_ticket_1892(self):
1208: (8)         f = np.vectorize(lambda x: x)
1209: (8)         s = '0123456789' * 10
1210: (4)         assert_equal(s, f(s))
1211: (8)     def test_cache(self):
1212: (8)         _calls = [0]
1213: (8)         @vectorize
1214: (12)         def f(x):
1215: (12)             _calls[0] += 1
1216: (8)             return x ** 2
1217: (8)             f.cache = True
1218: (8)             x = np.arange(5)
1219: (8)             assert_array_equal(f(x), x * x)
1220: (4)             assert_equal(_calls[0], len(x))
1221: (8)     def test_otypes(self):
1222: (8)         f = np.vectorize(lambda x: x)
1223: (8)         f.otypes = 'i'
1224: (8)         x = np.arange(5)
1225: (4)         assert_array_equal(f(x), x)
1226: (8)     def test_parse_gufunc_signature(self):
1227: (8)         assert_equal(nfb._parse_gufunc_signature('(x)->()', [(['x',]), []]))
1228: (21)         assert_equal(nfb._parse_gufunc_signature('((x,y))->()', [
1229: (8)             [(['x', 'y'])], [])))
1230: (21)         assert_equal(nfb._parse_gufunc_signature('((x),(y))->()', [
1231: (8)             [(['x',]), ('y',)], []]))
1232: (21)         assert_equal(nfb._parse_gufunc_signature('((x))->(y)', [
1233: (8)             [(['x',]), [('y',)]], [])))
1234: (21)         assert_equal(nfb._parse_gufunc_signature('((x))->(y),()', [
1235: (8)             [(['x',]), [('y',), ()]], [])))
1236: (21)         assert_equal(nfb._parse_gufunc_signature('((a,b,c),(d))->(d,e)', [
1237: (8)             [(()), ('a', 'b', 'c'), ('d',)], [('d', 'e')]]))
1238: (8)         assert_equal(nfb._parse_gufunc_signature('((x,y))->()', [
1239: (21)             [(['x', 'y'])], [])))
1240: (8)         assert_equal(nfb._parse_gufunc_signature('((x),(y))->()', [
1241: (21)             [(['x',]), ('y',)], []]))
1242: (8)         assert_equal(nfb._parse_gufunc_signature('((x))->(y)', [
1243: (21)             [(['x',]), [('y',)]], [])))
1244: (8)         assert_equal(nfb._parse_gufunc_signature('((x))->(y),()', [
1245: (21)             [(['x',]), [('y',), ()]], [])))
1246: (8)         assert_equal(nfb._parse_gufunc_signature(
1247: (21)             '((a,b,c),(d))->(d,e)', [
1248: (21)                 [(()), ('a', 'b', 'c'), ('d',)], [('d', 'e')]]))
1249: (8)         with assert_raises(ValueError):

```

```

1250: (12)             nfb._parse_gufunc_signature('(x)(y)->()')
1251: (8)              with assert_raises(ValueError):
1252: (12)                nfb._parse_gufunc_signature('(x),(y)->')
1253: (8)              with assert_raises(ValueError):
1254: (12)                nfb._parse_gufunc_signature('((x))->(x)')
1255: (4)    def test_signature_simple(self):
1256: (8)      def addsubtract(a, b):
1257: (12)        if a > b:
1258: (16)          return a - b
1259: (12)        else:
1260: (16)          return a + b
1261: (8)      f = vectorize(addsubtract, signature='(),()->()')
1262: (8)      r = f([0, 3, 6, 9], [1, 3, 5, 7])
1263: (8)      assert_array_equal(r, [1, 6, 1, 2])
1264: (4)    def test_signature_mean_last(self):
1265: (8)      def mean(a):
1266: (12)        return a.mean()
1267: (8)      f = vectorize(mean, signature='(n)->()')
1268: (8)      r = f([[1, 3], [2, 4]])
1269: (8)      assert_array_equal(r, [2, 3])
1270: (4)    def test_signature_center(self):
1271: (8)      def center(a):
1272: (12)        return a - a.mean()
1273: (8)      f = vectorize(center, signature='(n)->(n)')
1274: (8)      r = f([[1, 3], [2, 4]])
1275: (8)      assert_array_equal(r, [[-1, 1], [-1, 1]])
1276: (4)    def test_signature_two_outputs(self):
1277: (8)      f = vectorize(lambda x: (x, x), signature='()->(),()')
1278: (8)      r = f([1, 2, 3])
1279: (8)      assert_(isinstance(r, tuple) and len(r) == 2)
1280: (8)      assert_array_equal(r[0], [1, 2, 3])
1281: (8)      assert_array_equal(r[1], [1, 2, 3])
1282: (4)    def test_signature_outer(self):
1283: (8)      f = vectorize(np.outer, signature='(a),(b)->(a,b)')
1284: (8)      r = f([1, 2], [1, 2, 3])
1285: (8)      assert_array_equal(r, [[1, 2, 3], [2, 4, 6]])
1286: (8)      r = f([[1, 2]]], [1, 2, 3])
1287: (8)      assert_array_equal(r, [[[1, 2, 3], [2, 4, 6]]]))
1288: (8)      r = f([[1, 0], [2, 0]], [1, 2, 3])
1289: (8)      assert_array_equal(r, [[[1, 2, 3], [0, 0, 0]],
1290: (31)                                [[2, 4, 6], [0, 0, 0]]])
1291: (8)      r = f([1, 2], [[1, 2, 3], [0, 0, 0]])
1292: (8)      assert_array_equal(r, [[[1, 2, 3], [2, 4, 6]],
1293: (31)                                [[0, 0, 0], [0, 0, 0]]])
1294: (4)    def test_signature_computed_size(self):
1295: (8)      f = vectorize(lambda x: x[:-1], signature='(n)->(m)')
1296: (8)      r = f([1, 2, 3])
1297: (8)      assert_array_equal(r, [1, 2])
1298: (8)      r = f([[1, 2, 3], [2, 3, 4]])
1299: (8)      assert_array_equal(r, [[1, 2], [2, 3]])
1300: (4)    def test_signature_excluded(self):
1301: (8)      def foo(a, b=1):
1302: (12)        return a + b
1303: (8)      f = vectorize(foo, signature='()->()', excluded={'b'})
1304: (8)      assert_array_equal(f([1, 2, 3]), [2, 3, 4])
1305: (8)      assert_array_equal(f([1, 2, 3], b=0), [1, 2, 3])
1306: (4)    def test_signature_otypes(self):
1307: (8)      f = vectorize(lambda x: x, signature='(n)->(n)', otypes=['float64'])
1308: (8)      r = f([1, 2, 3])
1309: (8)      assert_equal(r.dtype, np.dtype('float64'))
1310: (8)      assert_array_equal(r, [1, 2, 3])
1311: (4)    def test_signature_invalid_inputs(self):
1312: (8)      f = vectorize(operator.add, signature='(n),(n)->(n)')
1313: (8)      with assert_raises_regex(TypeError, 'wrong number of positional'):
1314: (12)        f([1, 2])
1315: (8)      with assert_raises_regex(
1316: (16)          ValueError, 'does not have enough dimensions'):
1317: (12)        f(1, 2)
1318: (8)      with assert_raises_regex(

```

```

1319: (16)           ValueError, 'inconsistent size for core dimension'):
1320: (12)             f([1, 2], [1, 2, 3])
1321: (8)             f = vectorize(operator.add, signature='()->()')
1322: (8)             with assert_raises_regex(TypeError, 'wrong number of positional'):
1323: (12)               f(1, 2)
1324: (4)             def test_signature_invalid_outputs(self):
1325: (8)               f = vectorize(lambda x: x[:-1], signature='(n)->(n)')
1326: (8)               with assert_raises_regex(
1327: (16)                 ValueError, 'inconsistent size for core dimension'):
1328: (12)                   f([1, 2, 3])
1329: (8)                   f = vectorize(lambda x: x, signature='()->(),()')
1330: (8)                   with assert_raises_regex(ValueError, 'wrong number of outputs'):
1331: (12)                     f(1)
1332: (8)                     f = vectorize(lambda x: (x, x), signature='()->()')
1333: (8)                     with assert_raises_regex(ValueError, 'wrong number of outputs'):
1334: (12)                       f([1, 2])
1335: (4)             def test_size_zero_output(self):
1336: (8)               f = np.vectorize(lambda x: x)
1337: (8)               x = np.zeros([0, 5], dtype=int)
1338: (8)               with assert_raises_regex(ValueError, 'otypes'):
1339: (12)                 f(x)
1340: (8)                 f.otypes = 'i'
1341: (8)                 assert_array_equal(f(x), x)
1342: (8)                 f = np.vectorize(lambda x: x, signature='()->()')
1343: (8)                 with assert_raises_regex(ValueError, 'otypes'):
1344: (12)                   f(x)
1345: (8)                   f = np.vectorize(lambda x: x, signature='()->()', otypes='i')
1346: (8)                   assert_array_equal(f(x), x)
1347: (8)                   f = np.vectorize(lambda x: x, signature='(n)->(n)', otypes='i')
1348: (8)                   assert_array_equal(f(x), x)
1349: (8)                   f = np.vectorize(lambda x: x, signature='(n)->(n)')
1350: (8)                   assert_array_equal(f(x.T), x.T)
1351: (8)                   f = np.vectorize(lambda x: [x], signature='()->(n)', otypes='i')
1352: (8)                   with assert_raises_regex(ValueError, 'new output dimensions'):
1353: (12)                     f(x)
1354: (4)             def test_subclasses(self):
1355: (8)               class subclass(np.ndarray):
1356: (12)                 pass
1357: (8)                 m = np.array([[1., 0., 0.],
1358: (22)                   [0., 0., 1.],
1359: (22)                   [0., 1., 0.]]).view(subclass)
1360: (8)                 v = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8.,
9.]]).view(subclass)
1361: (8)                 matvec = np.vectorize(np.matmul, signature='(m,m),(m)->(m)')
1362: (8)                 r = matvec(m, v)
1363: (8)                 assert_equal(type(r), subclass)
1364: (8)                 assert_equal(r, [[1., 3., 2.], [4., 6., 5.], [7., 9., 8.]])
1365: (8)                 mult = np.vectorize(lambda x, y: x*y)
1366: (8)                 r = mult(m, v)
1367: (8)                 assert_equal(type(r), subclass)
1368: (8)                 assert_equal(r, m * v)
1369: (4)             def test_name(self):
1370: (8)               @np.vectorize
1371: (8)               def f2(a, b):
1372: (12)                 return a + b
1373: (8)                 assert f2.__name__ == 'f2'
1374: (4)             def test_decorator(self):
1375: (8)               @vectorize
1376: (8)               def addsubtract(a, b):
1377: (12)                 if a > b:
1378: (16)                   return a - b
1379: (12)                 else:
1380: (16)                   return a + b
1381: (8)                 r = addsubtract([0, 3, 6, 9], [1, 3, 5, 7])
1382: (8)                 assert_array_equal(r, [1, 6, 1, 2])
1383: (4)             def test_docstring(self):
1384: (8)               @vectorize
1385: (8)               def f(x):
1386: (12)                 """Docstring"""

```

```

1387: (12)             return x
1388: (8)              if sys.flags.optimize < 2:
1389: (12)                assert f.__doc__ == "Docstring"
1390: (4)              def test_partial(self):
1391: (8)                def foo(x, y):
1392: (12)                  return x + y
1393: (8)                bar = partial(foo, 3)
1394: (8)                vbar = np.vectorize(bar)
1395: (8)                assert vbar(1) == 4
1396: (4)              def test_signature_otypes_decorator(self):
1397: (8)                @vectorize(signature='(n)->(n)', otypes=['float64'])
1398: (8)                def f(x):
1399: (12)                  return x
1400: (8)                r = f([1, 2, 3])
1401: (8)                assert_equal(r.dtype, np.dtype('float64'))
1402: (8)                assert_array_equal(r, [1, 2, 3])
1403: (8)                assert f.__name__ == 'f'
1404: (4)              def test_bad_input(self):
1405: (8)                with assert_raises(TypeError):
1406: (12)                  A = np.vectorize(pyfunc = 3)
1407: (4)              def test_no_keywords(self):
1408: (8)                with assert_raises(TypeError):
1409: (12)                  @np.vectorize("string")
1410: (12)                  def foo():
1411: (16)                      return "bar"
1412: (4)              def test_positional_regression_9477(self):
1413: (8)                f = vectorize((lambda x: x), ['float64'])
1414: (8)                r = f([2])
1415: (8)                assert_equal(r.dtype, np.dtype('float64'))
1416: (0)            class TestLeaks:
1417: (4)              class A:
1418: (8)                iters = 20
1419: (8)                def bound(self, *args):
1420: (12)                  return 0
1421: (8)                @staticmethod
1422: (8)                def unbound(*args):
1423: (12)                  return 0
1424: (4)              @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
1425: (4)              @pytest.mark.parametrize('name, incr', [
1426: (12)                  ('bound', A.iters),
1427: (12)                  ('unbound', 0),
1428: (12)                  ])
1429: (4)              def test_frompyfunc_leaks(self, name, incr):
1430: (8)                import gc
1431: (8)                A_func = getattr(self.A, name)
1432: (8)                gc.disable()
1433: (8)                try:
1434: (12)                  refcount = sys.getrefcount(A_func)
1435: (12)                  for i in range(self.A.iters):
1436: (16)                      a = self.A()
1437: (16)                      a.f = np.frompyfunc(getattr(a, name), 1, 1)
1438: (16)                      out = a.f(np.arange(10))
1439: (12)                      a = None
1440: (12)                      assert_equal(sys.getrefcount(A_func), refcount + incr)
1441: (12)                      for i in range(5):
1442: (16)                          gc.collect()
1443: (12)                      assert_equal(sys.getrefcount(A_func), refcount)
1444: (8)                finally:
1445: (12)                  gc.enable()
1446: (0)            class TestDigitize:
1447: (4)              def test_forward(self):
1448: (8)                x = np.arange(-6, 5)
1449: (8)                bins = np.arange(-5, 5)
1450: (8)                assert_array_equal(digitize(x, bins), np.arange(11))
1451: (4)              def test_reverse(self):
1452: (8)                x = np.arange(5, -6, -1)
1453: (8)                bins = np.arange(5, -5, -1)
1454: (8)                assert_array_equal(digitize(x, bins), np.arange(11))
1455: (4)              def test_random(self):

```

```

1456: (8)             x = rand(10)
1457: (8)             bin = np.linspace(x.min(), x.max(), 10)
1458: (8)             assert_(np.all(digitize(x, bin) != 0))
1459: (4)             def test_right_basic(self):
1460: (8)                 x = [1, 5, 4, 10, 8, 11, 0]
1461: (8)                 bins = [1, 5, 10]
1462: (8)                 default_answer = [1, 2, 1, 3, 2, 3, 0]
1463: (8)                 assert_array_equal(digitize(x, bins), default_answer)
1464: (8)                 right_answer = [0, 1, 1, 2, 2, 3, 0]
1465: (8)                 assert_array_equal(digitize(x, bins, True), right_answer)
1466: (4)             def test_right_open(self):
1467: (8)                 x = np.arange(-6, 5)
1468: (8)                 bins = np.arange(-6, 4)
1469: (8)                 assert_array_equal(digitize(x, bins, True), np.arange(11))
1470: (4)             def test_right_open_reverse(self):
1471: (8)                 x = np.arange(5, -6, -1)
1472: (8)                 bins = np.arange(4, -6, -1)
1473: (8)                 assert_array_equal(digitize(x, bins, True), np.arange(11))
1474: (4)             def test_right_open_random(self):
1475: (8)                 x = rand(10)
1476: (8)                 bins = np.linspace(x.min(), x.max(), 10)
1477: (8)                 assert_(np.all(digitize(x, bins, True) != 10))
1478: (4)             def test_monotonic(self):
1479: (8)                 x = [-1, 0, 1, 2]
1480: (8)                 bins = [0, 0, 1]
1481: (8)                 assert_array_equal(digitize(x, bins, False), [0, 2, 3, 3])
1482: (8)                 assert_array_equal(digitize(x, bins, True), [0, 0, 2, 3])
1483: (8)                 bins = [1, 1, 0]
1484: (8)                 assert_array_equal(digitize(x, bins, False), [3, 2, 0, 0])
1485: (8)                 assert_array_equal(digitize(x, bins, True), [3, 3, 2, 0])
1486: (8)                 bins = [1, 1, 1, 1]
1487: (8)                 assert_array_equal(digitize(x, bins, False), [0, 0, 4, 4])
1488: (8)                 assert_array_equal(digitize(x, bins, True), [0, 0, 0, 4])
1489: (8)                 bins = [0, 0, 1, 0]
1490: (8)                 assert_raises(ValueError, digitize, x, bins)
1491: (8)                 bins = [1, 1, 0, 1]
1492: (8)                 assert_raises(ValueError, digitize, x, bins)
1493: (4)             def test_casting_error(self):
1494: (8)                 x = [1, 2, 3 + 1.j]
1495: (8)                 bins = [1, 2, 3]
1496: (8)                 assert_raises(TypeError, digitize, x, bins)
1497: (8)                 x, bins = bins, x
1498: (8)                 assert_raises(TypeError, digitize, x, bins)
1499: (4)             def test_return_type(self):
1500: (8)                 class A(np.ndarray):
1501: (12)                     pass
1502: (8)                 a = np.arange(5).view(A)
1503: (8)                 b = np.arange(1, 3).view(A)
1504: (8)                 assert_(not isinstance(digitize(b, a, False), A))
1505: (8)                 assert_(not isinstance(digitize(b, a, True), A))
1506: (4)             def test_large_integers_increasing(self):
1507: (8)                 x = 2**54 # loses precision in a float
1508: (8)                 assert_equal(np.digitize(x, [x - 1, x + 1]), 1)
1509: (4)             @pytest.mark.xfail(
1510: (8)                 reason="gh-11022: np.core.multiarray._monotonicity loses precision")
1511: (4)             def test_large_integers_decreasing(self):
1512: (8)                 x = 2**54 # loses precision in a float
1513: (8)                 assert_equal(np.digitize(x, [x + 1, x - 1]), 1)
1514: (0)             class TestUnwrap:
1515: (4)                 def test_simple(self):
1516: (8)                     assert_array_equal(unwrap([1, 1 + 2 * np.pi]), [1, 1])
1517: (8)                     assert_(np.all(diff(unwrap(rand(10) * 100)) < np.pi))
1518: (4)                 def test_period(self):
1519: (8)                     assert_array_equal(unwrap([1, 1 + 256], period=255), [1, 2])
1520: (8)                     assert_(np.all(diff(unwrap(rand(10) * 1000, period=255)) < 255))
1521: (8)                     simple_seq = np.array([0, 75, 150, 225, 300])
1522: (8)                     wrap_seq = np.mod(simple_seq, 255)
1523: (8)                     assert_array_equal(unwrap(wrap_seq, period=255), simple_seq)
1524: (8)                     uneven_seq = np.array([0, 75, 150, 225, 300, 430])

```

```

1525: (8)             wrap_uneven = np.mod(uneven_seq, 250)
1526: (8)             no_discont = unwrap(wrap_uneven, period=250)
1527: (8)             assert_array_equal(no_discont, [0, 75, 150, 225, 300, 180])
1528: (8)             sm_discont = unwrap(wrap_uneven, period=250, discont=140)
1529: (8)             assert_array_equal(sm_discont, [0, 75, 150, 225, 300, 430])
1530: (8)             assert sm_discont.dtype == wrap_uneven.dtype
1531: (0) @pytest.mark.parametrize(
1532: (4)     "dtype", "0" + np.typecodes["AllInteger"] + np.typecodes["Float"]
1533: (0)
1534: (0) @pytest.mark.parametrize("M", [0, 1, 10])
1535: (0) class TestFilterwindows:
1536: (4)     def test_hanning(self, dtype: str, M: int) -> None:
1537: (8)         scalar = np.array(M, dtype=dtype)[()]
1538: (8)         w = hanning(scalar)
1539: (8)         if dtype == "0":
1540: (12)             ref_dtype = np.float64
1541: (8)         else:
1542: (12)             ref_dtype = np.result_type(scalar.dtype, np.float64)
1543: (8)         assert w.dtype == ref_dtype
1544: (8)         assert_equal(w, flipud(w))
1545: (8)         if scalar < 1:
1546: (12)             assert_array_equal(w, np.array([]))
1547: (8)         elif scalar == 1:
1548: (12)             assert_array_equal(w, np.ones(1))
1549: (8)         else:
1550: (12)             assert_almost_equal(np.sum(w, axis=0), 4.500, 4)
1551: (4)     def test_hamming(self, dtype: str, M: int) -> None:
1552: (8)         scalar = np.array(M, dtype=dtype)[()]
1553: (8)         w = hamming(scalar)
1554: (8)         if dtype == "0":
1555: (12)             ref_dtype = np.float64
1556: (8)         else:
1557: (12)             ref_dtype = np.result_type(scalar.dtype, np.float64)
1558: (8)         assert w.dtype == ref_dtype
1559: (8)         assert_equal(w, flipud(w))
1560: (8)         if scalar < 1:
1561: (12)             assert_array_equal(w, np.array([]))
1562: (8)         elif scalar == 1:
1563: (12)             assert_array_equal(w, np.ones(1))
1564: (8)         else:
1565: (12)             assert_almost_equal(np.sum(w, axis=0), 4.9400, 4)
1566: (4)     def test_bartlett(self, dtype: str, M: int) -> None:
1567: (8)         scalar = np.array(M, dtype=dtype)[()]
1568: (8)         w = bartlett(scalar)
1569: (8)         if dtype == "0":
1570: (12)             ref_dtype = np.float64
1571: (8)         else:
1572: (12)             ref_dtype = np.result_type(scalar.dtype, np.float64)
1573: (8)         assert w.dtype == ref_dtype
1574: (8)         assert_equal(w, flipud(w))
1575: (8)         if scalar < 1:
1576: (12)             assert_array_equal(w, np.array([]))
1577: (8)         elif scalar == 1:
1578: (12)             assert_array_equal(w, np.ones(1))
1579: (8)         else:
1580: (12)             assert_almost_equal(np.sum(w, axis=0), 4.4444, 4)
1581: (4)     def test_blackman(self, dtype: str, M: int) -> None:
1582: (8)         scalar = np.array(M, dtype=dtype)[()]
1583: (8)         w = blackman(scalar)
1584: (8)         if dtype == "0":
1585: (12)             ref_dtype = np.float64
1586: (8)         else:
1587: (12)             ref_dtype = np.result_type(scalar.dtype, np.float64)
1588: (8)         assert w.dtype == ref_dtype
1589: (8)         assert_equal(w, flipud(w))
1590: (8)         if scalar < 1:
1591: (12)             assert_array_equal(w, np.array([]))
1592: (8)         elif scalar == 1:
1593: (12)             assert_array_equal(w, np.ones(1))

```

```

1594: (8)
1595: (12)
1596: (4)
1597: (8)
1598: (8)
1599: (8)
1600: (12)
1601: (8)
1602: (12)
1603: (8)
1604: (8)
1605: (8)
1606: (12)
1607: (8)
1608: (12)
1609: (8)
1610: (12)
1611: (0)
1612: (4)
1613: (8)
1614: (8)
1615: (8)
1616: (4)
1617: (8)
1618: (8)
1619: (8)
1620: (8)
1621: (8)
1622: (8)
1623: (8)
1624: (8)
1625: (8)
1626: (8)
1627: (8)
1628: (8)
1629: (8)
1630: (8)
1631: (8)
1632: (8)
1633: (8)
1634: (8)
1635: (8)
1636: (8)
1637: (8)
1638: (8)
1639: (8)
1640: (8)
1641: (8)
1642: (8)
1643: (8)
1644: (8)
1645: (4)
1646: (8)
1647: (8)
1648: (8)
1649: (8)
1650: (8)
1651: (8)
1652: (8)
1653: (8)
1654: (8)
1655: (8)
1656: (0)
1657: (4)
1658: (8)
1659: (8)
1660: (8)
1661: (4)
1662: (8)

        else:
            assert_almost_equal(np.sum(w, axis=0), 3.7800, 4)
    def test_kaiser(self, dtype: str, M: int) -> None:
        scalar = np.array(M, dtype=dtype)[()]
        w = kaiser(scalar, 0)
        if dtype == "O":
            ref_dtype = np.float64
        else:
            ref_dtype = np.result_type(scalar.dtype, np.float64)
        assert w.dtype == ref_dtype
        assert_equal(w, flipud(w))
        if scalar < 1:
            assert_array_equal(w, np.array([]))
        elif scalar == 1:
            assert_array_equal(w, np.ones(1))
        else:
            assert_almost_equal(np.sum(w, axis=0), 10, 15)
    class TestTrapz:
        def test_simple(self):
            x = np.arange(-10, 10, .1)
            r = trapz(np.exp(-.5 * x ** 2) / np.sqrt(2 * np.pi), dx=0.1)
            assert_almost_equal(r, 1, 7)
        def test_ndim(self):
            x = np.linspace(0, 1, 3)
            y = np.linspace(0, 2, 8)
            z = np.linspace(0, 3, 13)
            wx = np.ones_like(x) * (x[1] - x[0])
            wx[0] /= 2
            wx[-1] /= 2
            wy = np.ones_like(y) * (y[1] - y[0])
            wy[0] /= 2
            wy[-1] /= 2
            wz = np.ones_like(z) * (z[1] - z[0])
            wz[0] /= 2
            wz[-1] /= 2
            q = x[:, None, None] + y[None, :, None] + z[None, None, :]
            qx = (q * wx[:, None, None]).sum(axis=0)
            qy = (q * wy[None, :, None]).sum(axis=1)
            qz = (q * wz[None, None, :]).sum(axis=2)
            r = trapz(q, x=x[:, None, None], axis=0)
            assert_almost_equal(r, qx)
            r = trapz(q, x=y[None, :, None], axis=1)
            assert_almost_equal(r, qy)
            r = trapz(q, x=z[None, None, :], axis=2)
            assert_almost_equal(r, qz)
            r = trapz(q, x=x, axis=0)
            assert_almost_equal(r, qx)
            r = trapz(q, x=y, axis=1)
            assert_almost_equal(r, qy)
            r = trapz(q, x=z, axis=2)
            assert_almost_equal(r, qz)
        def test_masked(self):
            x = np.arange(5)
            y = x * x
            mask = x == 2
            ym = np.ma.array(y, mask=mask)
            r = 13.0 # sum(0.5 * (0 + 1) * 1.0 + 0.5 * (9 + 16))
            assert_almost_equal(trapz(ym, x), r)
            xm = np.ma.array(x, mask=mask)
            assert_almost_equal(trapz(ym, xm), r)
            xm = np.ma.array(x, mask=mask)
            assert_almost_equal(trapz(y, xm), r)
    class TestSinc:
        def test_simple(self):
            assert_(sinc(0) == 1)
            w = sinc(np.linspace(-1, 1, 100))
            assert_array_almost_equal(w, flipud(w), 7)
        def test_array_like(self):
            x = [0, 0.5]

```

```

1663: (8)          y1 = sinc(np.array(x))
1664: (8)          y2 = sinc(list(x))
1665: (8)          y3 = sinc(tuple(x))
1666: (8)          assert_array_equal(y1, y2)
1667: (8)          assert_array_equal(y1, y3)
1668: (0)          class TestUnique:
1669: (4)            def test_simple(self):
1670: (8)              x = np.array([4, 3, 2, 1, 1, 2, 3, 4, 0])
1671: (8)              assert_(np.all(unique(x) == [0, 1, 2, 3, 4]))
1672: (8)              assert_(unique(np.array([1, 1, 1, 1, 1])) == np.array([1]))
1673: (8)              x = ['widget', 'ham', 'foo', 'bar', 'foo', 'ham']
1674: (8)              assert_(np.all(unique(x) == ['bar', 'foo', 'ham', 'widget']))
1675: (8)              x = np.array([5 + 6j, 1 + 1j, 1 + 10j, 10, 5 + 6j])
1676: (8)              assert_(np.all(unique(x) == [1 + 1j, 1 + 10j, 5 + 6j, 10]))
1677: (0)          class TestCheckFinite:
1678: (4)            def test_simple(self):
1679: (8)              a = [1, 2, 3]
1680: (8)              b = [1, 2, np.inf]
1681: (8)              c = [1, 2, np.nan]
1682: (8)              np.lib.asarray_chkfinite(a)
1683: (8)              assert_raises(ValueError, np.lib.asarray_chkfinite, b)
1684: (8)              assert_raises(ValueError, np.lib.asarray_chkfinite, c)
1685: (4)            def test_dtype_order(self):
1686: (8)              a = [1, 2, 3]
1687: (8)              a = np.lib.asarray_chkfinite(a, order='F', dtype=np.float64)
1688: (8)              assert_(a.dtype == np.float64)
1689: (0)          class TestCorrCoef:
1690: (4)            A = np.array(
1691: (8)              [[0.15391142, 0.18045767, 0.14197213],
1692: (9)              [0.70461506, 0.96474128, 0.27906989],
1693: (9)              [0.9297531, 0.32296769, 0.19267156]])
1694: (4)            B = np.array(
1695: (8)              [[0.10377691, 0.5417086, 0.49807457],
1696: (9)              [0.82872117, 0.77801674, 0.39226705],
1697: (9)              [0.9314666, 0.66800209, 0.03538394]])
1698: (4)            res1 = np.array(
1699: (8)              [[1., 0.9379533, -0.04931983],
1700: (9)              [0.9379533, 1., 0.30007991],
1701: (9)              [-0.04931983, 0.30007991, 1.]])
1702: (4)            res2 = np.array(
1703: (8)              [[1., 0.9379533, -0.04931983, 0.30151751, 0.66318558, 0.51532523],
1704: (9)              [0.9379533, 1., 0.30007991, -0.04781421, 0.88157256, 0.78052386],
1705: (9)              [-0.04931983, 0.30007991, 1., -0.96717111, 0.71483595, 0.83053601],
1706: (9)              [0.30151751, -0.04781421, -0.96717111, 1., -0.51366032, -0.66173113],
1707: (9)              [0.66318558, 0.88157256, 0.71483595, -0.51366032, 1., 0.98317823],
1708: (9)              [0.51532523, 0.78052386, 0.83053601, -0.66173113, 0.98317823, 1.]])
1709: (4)            def test_non_array(self):
1710: (8)              assert_almost_equal(np.corrcoef([0, 1, 0], [1, 0, 1]),
1711: (28)                [[1., -1.], [-1., 1.]])
1712: (4)            def test_simple(self):
1713: (8)              tgt1 = corrcoef(self.A)
1714: (8)              assert_almost_equal(tgt1, self.res1)
1715: (8)              assert_(np.all(np.abs(tgt1) <= 1.0))
1716: (8)              tgt2 = corrcoef(self.A, self.B)
1717: (8)              assert_almost_equal(tgt2, self.res2)
1718: (8)              assert_(np.all(np.abs(tgt2) <= 1.0))
1719: (4)            def test_ddof(self):
1720: (8)              with suppress_warnings() as sup:
1721: (12)                warnings.simplefilter("always")
1722: (12)                assert_warns(DeprecationWarning, corrcoef, self.A, ddof=-1)
1723: (12)                sup.filter(DeprecationWarning)
1724: (12)                assert_almost_equal(corrcoef(self.A, ddof=-1), self.res1)
1725: (12)                assert_almost_equal(corrcoef(self.A, self.B, ddof=-1), self.res2)
1726: (12)                assert_almost_equal(corrcoef(self.A, ddof=3), self.res1)
1727: (12)                assert_almost_equal(corrcoef(self.A, self.B, ddof=3), self.res2)
1728: (4)            def test_bias(self):
1729: (8)              with suppress_warnings() as sup:
1730: (12)                warnings.simplefilter("always")
1731: (12)                assert_warns(DeprecationWarning, corrcoef, self.A, self.B, 1, 0)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1732: (12) assert_warns(DeprecationWarning, corrcoef, self.A, bias=0)
1733: (12) sup.filter(DeprecationWarning)
1734: (12) assert_almost_equal(corrcoef(self.A, bias=1), self.res1)
1735: (4) def test_complex(self):
1736: (8)     x = np.array([[1, 2, 3], [1j, 2j, 3j]])
1737: (8)     res = corrcoef(x)
1738: (8)     tgt = np.array([[1., -1.j], [1.j, 1.]])
1739: (8)     assert_allclose(res, tgt)
1740: (8)     assert_(np.all(np.abs(res) <= 1.0))
1741: (4) def test_xy(self):
1742: (8)     x = np.array([[1, 2, 3]])
1743: (8)     y = np.array([[1j, 2j, 3j]])
1744: (8)     assert_allclose(np.corrcoef(x, y), np.array([[1., -1.j], [1.j, 1.]]))
1745: (4) def test_empty(self):
1746: (8)     with warnings.catch_warnings(record=True):
1747: (12)         warnings.simplefilter('always', RuntimeWarning)
1748: (12)         assert_array_equal(corrcoef(np.array([])), np.nan)
1749: (12)         assert_array_equal(corrcoef(np.array([]).reshape(0, 2)),
1750: (31)                 np.array([]).reshape(0, 0))
1751: (12)         assert_array_equal(corrcoef(np.array([]).reshape(2, 0)),
1752: (31)                 np.array([[np.nan, np.nan], [np.nan, np.nan]]))
1753: (4) def test_extreme(self):
1754: (8)     x = [[1e-100, 1e100], [1e100, 1e-100]]
1755: (8)     with np.errstate(all='raise'):
1756: (12)         c = corrcoef(x)
1757: (8)         assert_array_almost_equal(c, np.array([[1., -1.], [-1., 1.]]))
1758: (8)         assert_(np.all(np.abs(c) <= 1.0))
1759: (4) @pytest.mark.parametrize("test_type", [np.half, np.single, np.double,
np.longdouble])
1760: (4) def test_corrcoef_dtype(self, test_type):
1761: (8)     cast_A = self.A.astype(test_type)
1762: (8)     res = corrcoef(cast_A, dtype=test_type)
1763: (8)     assert test_type == res.dtype
1764: (0) class TestCov:
1765: (4)     x1 = np.array([[0, 2], [1, 1], [2, 0]]).T
1766: (4)     res1 = np.array([[1., -1.], [-1., 1.]])
1767: (4)     x2 = np.array([0.0, 1.0, 2.0], ndmin=2)
1768: (4)     frequencies = np.array([1, 4, 1])
1769: (4)     x2_repeats = np.array([[0.0], [1.0], [1.0], [1.0], [1.0], [2.0]]).T
1770: (4)     res2 = np.array([[0.4, -0.4], [-0.4, 0.4]])
1771: (4)     unit_frequencies = np.ones(3, dtype=np.int_)
1772: (4)     weights = np.array([1.0, 4.0, 1.0])
1773: (4)     res3 = np.array([[2. / 3., -2. / 3.], [-2. / 3., 2. / 3.]])
1774: (4)     unit_weights = np.ones(3)
1775: (4)     x3 = np.array([0.3942, 0.5969, 0.7730, 0.9918, 0.7964])
1776: (4)     def test_basic(self):
1777: (8)         assert_allclose(cov(self.x1), self.res1)
1778: (4)     def test_complex(self):
1779: (8)         x = np.array([[1, 2, 3], [1j, 2j, 3j]])
1780: (8)         res = np.array([[1., -1.j], [1.j, 1.]])
1781: (8)         assert_allclose(cov(x), res)
1782: (8)         assert_allclose(cov(x, aweights=np.ones(3)), res)
1783: (4)     def test_xy(self):
1784: (8)         x = np.array([[1, 2, 3]])
1785: (8)         y = np.array([[1j, 2j, 3j]])
1786: (8)         assert_allclose(cov(x, y), np.array([[1., -1.j], [1.j, 1.]]))
1787: (4)     def test_empty(self):
1788: (8)         with warnings.catch_warnings(record=True):
1789: (12)             warnings.simplefilter('always', RuntimeWarning)
1790: (12)             assert_array_equal(cov(np.array([])), np.nan)
1791: (12)             assert_array_equal(cov(np.array([]).reshape(0, 2)),
1792: (31)                     np.array([]).reshape(0, 0))
1793: (12)             assert_array_equal(cov(np.array([]).reshape(2, 0)),
1794: (31)                     np.array([[np.nan, np.nan], [np.nan, np.nan]]))
1795: (4)     def test_wrong_ddof(self):
1796: (8)         with warnings.catch_warnings(record=True):
1797: (12)             warnings.simplefilter('always', RuntimeWarning)
1798: (12)             assert_array_equal(cov(self.x1, ddof=5),
1799: (31)                     np.array([[np.inf, -np.inf],

```

```

1800: (41)
1801: (4)
1802: (8)
1803: (8)
1804: (8)
1805: (4)
1806: (8)
1807: (4)
1808: (8)
1809: (24)
1810: (8)
1811: (24)
1812: (8)
1813: (24)
1814: (8)
1815: (8)
1816: (8)
1817: (8)
1818: (8)
1819: (8)
1820: (8)
1821: (8)
1822: (4)
1823: (8)
1824: (8)
1825: (24)
1826: (8)
1827: (8)
1828: (8)
1829: (8)
1830: (8)
1831: (8)
1832: (8)
1833: (4)
1834: (8)
1835: (28)
1836: (24)
1837: (8)
1838: (28)
1839: (24)
1840: (8)
1841: (28)
1842: (24)
1843: (8)
1844: (28)
1845: (24)
1846: (8)
1847: (28)
1848: (24)
1849: (8)
1850: (28)
1851: (24)
1852: (4)
np.longdouble])
1853: (4)
1854: (8)
1855: (8)
1856: (8)
1857: (0)
1858: (4)
1859: (8)
1860: (12)
1861: (12)
1862: (8)
1863: (8)
2815.71662847])
1864: (8)
1865: (8)
1866: (8)

        [-np.inf, np.inf]]))

    def test_1D_rowvar(self):
        assert_allclose(cov(self.x3), cov(self.x3, rowvar=False))
        y = np.array([0.0780, 0.3107, 0.2111, 0.0334, 0.8501])
        assert_allclose(cov(self.x3, y), cov(self.x3, y, rowvar=False))

    def test_1D_variance(self):
        assert_allclose(cov(self.x3, ddof=1), np.var(self.x3, ddof=1))

    def test_fweights(self):
        assert_allclose(cov(self.x2, fweights=self.frequencies),
                        cov(self.x2_repeats))
        assert_allclose(cov(self.x1, fweights=self.frequencies),
                        self.res2)
        assert_allclose(cov(self.x1, fweights=self.unit_frequencies),
                        self.res1)

        nonint = self.frequencies + 0.5
        assert_raises(TypeError, cov, self.x1, fweights=nonint)

        f = np.ones((2, 3), dtype=np.int_)
        assert_raises(RuntimeError, cov, self.x1, fweights=f)
        f = np.ones(2, dtype=np.int_)
        assert_raises(RuntimeError, cov, self.x1, fweights=f)
        f = -1 * np.ones(3, dtype=np.int_)
        assert_raises(ValueError, cov, self.x1, fweights=f)

    def test_aweights(self):
        assert_allclose(cov(self.x1, aweights=self.weights), self.res3)
        assert_allclose(cov(self.x1, aweights=3.0 * self.weights),
                        cov(self.x1, aweights=self.weights))
        assert_allclose(cov(self.x1, aweights=self.unit_weights), self.res1)

        w = np.ones((2, 3))
        assert_raises(RuntimeError, cov, self.x1, aweights=w)
        w = np.ones(2)
        assert_raises(RuntimeError, cov, self.x1, aweights=w)
        w = -1.0 * np.ones(3)
        assert_raises(ValueError, cov, self.x1, aweights=w)

    def test_unit_fweights_and_aweights(self):
        assert_allclose(cov(self.x2, fweights=self.frequencies,
                            aweights=self.unit_weights),
                        cov(self.x2_repeats))
        assert_allclose(cov(self.x1, fweights=self.frequencies,
                            aweights=self.unit_weights),
                        self.res2)
        assert_allclose(cov(self.x1, fweights=self.unit_frequencies,
                            aweights=self.unit_weights),
                        self.res1)
        assert_allclose(cov(self.x1, fweights=self.unit_frequencies,
                            aweights=self.weights),
                        self.res3)

        assert_allclose(cov(self.x1, fweights=self.unit_frequencies,
                            aweights=3.0 * self.weights),
                        cov(self.x1, aweights=self.weights))
        assert_allclose(cov(self.x1, fweights=self.unit_frequencies,
                            aweights=self.unit_weights),
                        self.res1)

@ pytest.mark.parametrize("test_type", [np.half, np.single, np.double,
                                         np.longdouble])

    def test_cov_dtype(self, test_type):
        cast_x1 = self.x1.astype(test_type)
        res = cov(cast_x1, dtype=test_type)
        assert test_type == res.dtype

class Test_I0:
    def test_simple(self):
        assert_almost_equal(
            i0(0.5),
            np.array(1.0634833707413234))
        A = np.array([0.49842636, 0.6969809, 0.22011976, 0.0155549, 10.0])
        expected = np.array([1.06307822, 1.12518299, 1.01214991, 1.00006049,
                            2815.71662847])

        assert_almost_equal(i0(A), expected)
        assert_almost_equal(i0(-A), expected)
        B = np.array([[0.827002, 0.99959078],

```

```

1867: (22) [0.89694769, 0.39298162],
1868: (22) [0.37954418, 0.05206293],
1869: (22) [0.36465447, 0.72446427],
1870: (22) [0.48164949, 0.50324519]])
1871: (8) assert_almost_equal(
1872: (12)     i0(B),
1873: (12)     np.array([[1.17843223, 1.26583466],
1874: (22)         [1.21147086, 1.03898290],
1875: (22)         [1.03633899, 1.00067775],
1876: (22)         [1.03352052, 1.13557954],
1877: (22)         [1.05884290, 1.06432317]]))
1878: (8) i0_0 = np.i0([0.])
1879: (8) assert_equal(i0_0.shape, (1,))
1880: (8) assert_array_equal(np.i0([0.]), np.array([1.]))
1881: (4) def test_non_array(self):
1882: (8)     a = np.arange(4)
1883: (8)     class array_like:
1884: (12)         __array_interface__ = a.__array_interface__
1885: (12)         def __array_wrap__(self, arr):
1886: (16)             return self
1887: (8)         assert isinstance(np.abs(array_like()), array_like)
1888: (8)         exp = np.i0(a)
1889: (8)         res = np.i0(array_like())
1890: (8)         assert_array_equal(exp, res)
1891: (4)     def test_complex(self):
1892: (8)         a = np.array([0, 1 + 2j])
1893: (8)         with pytest.raises(TypeError, match="i0 not supported for complex
values"):
1894: (12)             res = i0(a)
1895: (0)     class TestKaiser:
1896: (4)         def test_simple(self):
1897: (8)             assert_(np.isfinite(kaiser(1, 1.0)))
1898: (8)             assert_almost_equal(kaiser(0, 1.0),
1899: (28)                 np.array([]))
1900: (8)             assert_almost_equal(kaiser(2, 1.0),
1901: (28)                 np.array([0.78984831, 0.78984831]))
1902: (8)             assert_almost_equal(kaiser(5, 1.0),
1903: (28)                 np.array([0.78984831, 0.94503323, 1.,
1904: (38)                     0.94503323, 0.78984831]))
1905: (8)             assert_almost_equal(kaiser(5, 1.56789),
1906: (28)                 np.array([0.58285404, 0.88409679, 1.,
1907: (38)                     0.88409679, 0.58285404]))
1908: (4)         def test_int_beta(self):
1909: (8)             kaiser(3, 4)
1910: (0)     class TestMsort:
1911: (4)         def test_simple(self):
1912: (8)             A = np.array([[0.44567325, 0.79115165, 0.54900530],
1913: (22)                 [0.36844147, 0.37325583, 0.96098397],
1914: (22)                 [0.64864341, 0.52929049, 0.39172155]])
1915: (8)             with pytest.warns(DeprecationWarning, match="msort is deprecated"):
1916: (12)                 assert_almost_equal(
1917: (16)                     msort(A),
1918: (16)                     np.array([[0.36844147, 0.37325583, 0.39172155],
1919: (26)                         [0.44567325, 0.52929049, 0.54900530],
1920: (26)                         [0.64864341, 0.79115165, 0.96098397]]))
1921: (0)     class TestMeshgrid:
1922: (4)         def test_simple(self):
1923: (8)             [X, Y] = meshgrid([1, 2, 3], [4, 5, 6, 7])
1924: (8)             assert_array_equal(X, np.array([[1, 2, 3],
1925: (40)                 [1, 2, 3],
1926: (40)                 [1, 2, 3],
1927: (40)                 [1, 2, 3]]))
1928: (8)             assert_array_equal(Y, np.array([[4, 4, 4],
1929: (40)                 [5, 5, 5],
1930: (40)                 [6, 6, 6],
1931: (40)                 [7, 7, 7]]))
1932: (4)         def test_single_input(self):
1933: (8)             [X] = meshgrid([1, 2, 3, 4])
1934: (8)             assert_array_equal(X, np.array([1, 2, 3, 4]))

```

```

1935: (4)         def test_no_input(self):
1936: (8)             args = []
1937: (8)             assert_array_equal([], meshgrid(*args))
1938: (8)             assert_array_equal([], meshgrid(*args, copy=False))
1939: (4)         def test_indexing(self):
1940: (8)             x = [1, 2, 3]
1941: (8)             y = [4, 5, 6, 7]
1942: (8)             [X, Y] = meshgrid(x, y, indexing='ij')
1943: (8)             assert_array_equal(X, np.array([[1, 1, 1, 1],
1944: (40)                             [2, 2, 2, 2],
1945: (40)                             [3, 3, 3, 3]]))
1946: (8)             assert_array_equal(Y, np.array([[4, 5, 6, 7],
1947: (40)                             [4, 5, 6, 7],
1948: (40)                             [4, 5, 6, 7]]))
1949: (8)             z = [8, 9]
1950: (8)             assert_(meshgrid(x, y)[0].shape == (4, 3))
1951: (8)             assert_(meshgrid(x, y, indexing='ij')[0].shape == (3, 4))
1952: (8)             assert_(meshgrid(x, y, z)[0].shape == (4, 3, 2))
1953: (8)             assert_(meshgrid(x, y, z, indexing='ij')[0].shape == (3, 4, 2))
1954: (8)             assert_raises(ValueError, meshgrid, x, y, indexing='notvalid')
1955: (4)         def test_sparse(self):
1956: (8)             [X, Y] = meshgrid([1, 2, 3], [4, 5, 6, 7], sparse=True)
1957: (8)             assert_array_equal(X, np.array([[1, 2, 3]]))
1958: (8)             assert_array_equal(Y, np.array([[4], [5], [6], [7]]))
1959: (4)         def test_invalid_arguments(self):
1960: (8)             assert_raises(TypeError, meshgrid,
1961: (22)                             [1, 2, 3], [4, 5, 6, 7], indices='ij')
1962: (4)         def test_return_type(self):
1963: (8)             x = np.arange(0, 10, dtype=np.float32)
1964: (8)             y = np.arange(10, 20, dtype=np.float64)
1965: (8)             X, Y = np.meshgrid(x,y)
1966: (8)             assert_(X.dtype == x.dtype)
1967: (8)             assert_(Y.dtype == y.dtype)
1968: (8)             X, Y = np.meshgrid(x,y, copy=True)
1969: (8)             assert_(X.dtype == x.dtype)
1970: (8)             assert_(Y.dtype == y.dtype)
1971: (8)             X, Y = np.meshgrid(x,y, sparse=True)
1972: (8)             assert_(X.dtype == x.dtype)
1973: (8)             assert_(Y.dtype == y.dtype)
1974: (4)         def test_writeback(self):
1975: (8)             X = np.array([1.1, 2.2])
1976: (8)             Y = np.array([3.3, 4.4])
1977: (8)             x, y = np.meshgrid(X, Y, sparse=False, copy=True)
1978: (8)             x[0, :] = 0
1979: (8)             assert_equal(x[0, :], 0)
1980: (8)             assert_equal(x[1, :], X)
1981: (4)         def test_nd_shape(self):
1982: (8)             a, b, c, d, e = np.meshgrid(*([0] * i for i in range(1, 6)))
1983: (8)             expected_shape = (2, 1, 3, 4, 5)
1984: (8)             assert_equal(a.shape, expected_shape)
1985: (8)             assert_equal(b.shape, expected_shape)
1986: (8)             assert_equal(c.shape, expected_shape)
1987: (8)             assert_equal(d.shape, expected_shape)
1988: (8)             assert_equal(e.shape, expected_shape)
1989: (4)         def test_nd_values(self):
1990: (8)             a, b, c = np.meshgrid([0], [1, 2], [3, 4, 5])
1991: (8)             assert_equal(a, [[[0, 0, 0]], [[0, 0, 0]]])
1992: (8)             assert_equal(b, [[[1, 1, 1]], [[2, 2, 2]]])
1993: (8)             assert_equal(c, [[[3, 4, 5]], [[3, 4, 5]]])
1994: (4)         def test_nd_indexing(self):
1995: (8)             a, b, c = np.meshgrid([0], [1, 2], [3, 4, 5], indexing='ij')
1996: (8)             assert_equal(a, [[[0, 0, 0], [0, 0, 0]]])
1997: (8)             assert_equal(b, [[[1, 1, 1], [2, 2, 2]]])
1998: (8)             assert_equal(c, [[[3, 4, 5], [3, 4, 5]]])
1999: (0)     class TestPiecewise:
2000: (4)         def test_simple(self):
2001: (8)             x = piecewise([0, 0], [True, False], [1])
2002: (8)             assert_array_equal(x, [1, 0])
2003: (8)             x = piecewise([0, 0], [[True, False]], [1])

```

```

2004: (8) assert_array_equal(x, [1, 0])
2005: (8) x = piecewise([0, 0], np.array([True, False]), [1])
2006: (8) assert_array_equal(x, [1, 0])
2007: (8) x = piecewise([0, 0], np.array([1, 0]), [1])
2008: (8) assert_array_equal(x, [1, 0])
2009: (8) x = piecewise([0, 0], [np.array([1, 0])], [1])
2010: (8) assert_array_equal(x, [1, 0])
2011: (8) x = piecewise([0, 0], [[False, True]], [lambda x:-1])
2012: (8) assert_array_equal(x, [0, -1])
2013: (8) assert_raises_regex(ValueError, '1 or 2 functions are expected',
2014: (12)     piecewise, [0, 0], [[False, True]], [])
2015: (8) assert_raises_regex(ValueError, '1 or 2 functions are expected',
2016: (12)     piecewise, [0, 0], [[False, True]], [1, 2, 3])
2017: (4) def test_two_conditions(self):
2018: (8) x = piecewise([1, 2], [[True, False], [False, True]], [3, 4])
2019: (8) assert_array_equal(x, [3, 4])
2020: (4) def test_scalar_domains_three_conditions(self):
2021: (8) x = piecewise(3, [True, False, False], [4, 2, 0])
2022: (8) assert_equal(x, 4)
2023: (4) def test_default(self):
2024: (8) x = piecewise([1, 2], [True, False], [2])
2025: (8) assert_array_equal(x, [2, 0])
2026: (8) x = piecewise([1, 2], [True, False], [2, 3])
2027: (8) assert_array_equal(x, [2, 3])
2028: (4) def test_0d(self):
2029: (8) x = np.array(3)
2030: (8) y = piecewise(x, x > 3, [4, 0])
2031: (8) assert_(y.ndim == 0)
2032: (8) assert_(y == 0)
2033: (8) x = 5
2034: (8) y = piecewise(x, [True, False], [1, 0])
2035: (8) assert_(y.ndim == 0)
2036: (8) assert_(y == 1)
2037: (8) y = piecewise(x, [False, False, True], [1, 2, 3])
2038: (8) assert_array_equal(y, 3)
2039: (4) def test_0d_comparison(self):
2040: (8) x = 3
2041: (8) y = piecewise(x, [x <= 3, x > 3], [4, 0]) # Should succeed.
2042: (8) assert_equal(y, 4)
2043: (8) x = 4
2044: (8) y = piecewise(x, [x <= 3, (x > 3) * (x <= 5), x > 5], [1, 2, 3])
2045: (8) assert_array_equal(y, 2)
2046: (8) assert_raises_regex(ValueError, '2 or 3 functions are expected',
2047: (12)     piecewise, x, [x <= 3, x > 3], [1])
2048: (8) assert_raises_regex(ValueError, '2 or 3 functions are expected',
2049: (12)     piecewise, x, [x <= 3, x > 3], [1, 1, 1, 1])
2050: (4) def test_0d_0d_condition(self):
2051: (8) x = np.array(3)
2052: (8) c = np.array(x > 3)
2053: (8) y = piecewise(x, [c], [1, 2])
2054: (8) assert_equal(y, 2)
2055: (4) def test_multidimensional_extrafunc(self):
2056: (8) x = np.array([[-2.5, -1.5, -0.5],
2057: (22)                 [0.5, 1.5, 2.5]])
2058: (8) y = piecewise(x, [x < 0, x >= 2], [-1, 1, 3])
2059: (8) assert_array_equal(y, np.array([-1., -1., -1.,
2060: (40)                 [3., 3., 1.]]))
2061: (4) def test_subclasses(self):
2062: (8) class subclass(np.ndarray):
2063: (12)     pass
2064: (8)     x = np.arange(5.).view(subclass)
2065: (8)     r = piecewise(x, [x<2., x>=4], [-1., 1., 0.])
2066: (8)     assert_equal(type(r), subclass)
2067: (8)     assert_equal(r, [-1., -1., 0., 0., 1.])
2068: (0) class TestBincount:
2069: (4)     def test_simple(self):
2070: (8)         y = np.bincount(np.arange(4))
2071: (8)         assert_array_equal(y, np.ones(4))
2072: (4)         def test_simple2(self):

```

```

2073: (8)             y = np.bincount(np.array([1, 5, 2, 4, 1]))
2074: (8)             assert_array_equal(y, np.array([0, 2, 1, 0, 1, 1]))
2075: (4)             def test_simple_weight(self):
2076: (8)                 x = np.arange(4)
2077: (8)                 w = np.array([0.2, 0.3, 0.5, 0.1])
2078: (8)                 y = np.bincount(x, w)
2079: (8)                 assert_array_equal(y, w)
2080: (4)             def test_simple_weight2(self):
2081: (8)                 x = np.array([1, 2, 4, 5, 2])
2082: (8)                 w = np.array([0.2, 0.3, 0.5, 0.1, 0.2])
2083: (8)                 y = np.bincount(x, w)
2084: (8)                 assert_array_equal(y, np.array([0, 0.2, 0.5, 0, 0.5, 0.1]))
2085: (4)             def test_with_minlength(self):
2086: (8)                 x = np.array([0, 1, 0, 1, 1])
2087: (8)                 y = np.bincount(x, minlength=3)
2088: (8)                 assert_array_equal(y, np.array([2, 3, 0]))
2089: (8)                 x = []
2090: (8)                 y = np.bincount(x, minlength=0)
2091: (8)                 assert_array_equal(y, np.array([]))
2092: (4)             def test_with_minlength_smaller_than_maxvalue(self):
2093: (8)                 x = np.array([0, 1, 1, 2, 2, 3, 3])
2094: (8)                 y = np.bincount(x, minlength=2)
2095: (8)                 assert_array_equal(y, np.array([1, 2, 2, 2]))
2096: (8)                 y = np.bincount(x, minlength=0)
2097: (8)                 assert_array_equal(y, np.array([1, 2, 2, 2]))
2098: (4)             def test_with_minlength_and_weights(self):
2099: (8)                 x = np.array([1, 2, 4, 5, 2])
2100: (8)                 w = np.array([0.2, 0.3, 0.5, 0.1, 0.2])
2101: (8)                 y = np.bincount(x, w, 8)
2102: (8)                 assert_array_equal(y, np.array([0, 0.2, 0.5, 0, 0.5, 0.1, 0, 0]))
2103: (4)             def test_empty(self):
2104: (8)                 x = np.array([], dtype=int)
2105: (8)                 y = np.bincount(x)
2106: (8)                 assert_array_equal(x, y)
2107: (4)             def test_empty_with_minlength(self):
2108: (8)                 x = np.array([], dtype=int)
2109: (8)                 y = np.bincount(x, minlength=5)
2110: (8)                 assert_array_equal(y, np.zeros(5, dtype=int))
2111: (4)             def test_with_incorrect_minlength(self):
2112: (8)                 x = np.array([], dtype=int)
2113: (8)                 assert_raises_regex(TypeError,
2114: (28)                             "'str' object cannot be interpreted",
2115: (28)                             lambda: np.bincount(x, minlength="foobar"))
2116: (8)                 assert_raises_regex(ValueError,
2117: (28)                             "must not be negative",
2118: (28)                             lambda: np.bincount(x, minlength=-1))
2119: (8)                 x = np.arange(5)
2120: (8)                 assert_raises_regex(TypeError,
2121: (28)                             "'str' object cannot be interpreted",
2122: (28)                             lambda: np.bincount(x, minlength="foobar"))
2123: (8)                 assert_raises_regex(ValueError,
2124: (28)                             "must not be negative",
2125: (28)                             lambda: np.bincount(x, minlength=-1))
2126: (4)             @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
2127: (4)             def test_dtype_reference_leaks(self):
2128: (8)                 intp_refcount = sys.getrefcount(np.dtype(np.intp))
2129: (8)                 double_refcount = sys.getrefcount(np.dtype(np.double))
2130: (8)                 for j in range(10):
2131: (12)                     np.bincount([1, 2, 3])
2132: (8)                     assert_equal(sys.getrefcount(np.dtype(np.intp)), intp_refcount)
2133: (8)                     assert_equal(sys.getrefcount(np.dtype(np.double)), double_refcount)
2134: (8)                 for j in range(10):
2135: (12)                     np.bincount([1, 2, 3], [4, 5, 6])
2136: (8)                     assert_equal(sys.getrefcount(np.dtype(np.intp)), intp_refcount)
2137: (8)                     assert_equal(sys.getrefcount(np.dtype(np.double)), double_refcount)
2138: (4)             @pytest.mark.parametrize("vals", [[[2, 2]], 2])
2139: (4)             def test_error_not_1d(self, vals):
2140: (8)                 vals_arr = np.asarray(vals)
2141: (8)                 with assert_raises(ValueError):

```

```

2142: (12)             np.bincount(vals_arr)
2143: (8)              with assert_raises(ValueError):
2144: (12)                np.bincount(vals)
2145: (0)   class TestInterp:
2146: (4)    def test_exceptions(self):
2147: (8)      assert_raises(ValueError, interp, 0, [], [])
2148: (8)      assert_raises(ValueError, interp, 0, [0], [1, 2])
2149: (8)      assert_raises(ValueError, interp, 0, [0, 1], [1, 2], period=0)
2150: (8)      assert_raises(ValueError, interp, 0, [], [], period=360)
2151: (8)      assert_raises(ValueError, interp, 0, [0], [1, 2], period=360)
2152: (4)    def test_basic(self):
2153: (8)      x = np.linspace(0, 1, 5)
2154: (8)      y = np.linspace(0, 1, 5)
2155: (8)      x0 = np.linspace(0, 1, 50)
2156: (8)      assert_almost_equal(np.interp(x0, x, y), x0)
2157: (4)    def test_right_left_behavior(self):
2158: (8)      for size in range(1, 10):
2159: (12)        xp = np.arange(size, dtype=np.double)
2160: (12)        yp = np.ones(size, dtype=np.double)
2161: (12)        incpts = np.array([-1, 0, size - 1, size], dtype=np.double)
2162: (12)        decpts = incpts[::-1]
2163: (12)        incres = interp(incpts, xp, yp)
2164: (12)        decres = interp(decpts, xp, yp)
2165: (12)        inctgt = np.array([1, 1, 1, 1], dtype=float)
2166: (12)        dectgt = inctgt[::-1]
2167: (12)        assert_equal(incres, inctgt)
2168: (12)        assert_equal(decres, dectgt)
2169: (12)        incres = interp(incpts, xp, yp, left=0)
2170: (12)        decres = interp(decpts, xp, yp, left=0)
2171: (12)        inctgt = np.array([0, 1, 1, 1], dtype=float)
2172: (12)        dectgt = inctgt[::-1]
2173: (12)        assert_equal(incres, inctgt)
2174: (12)        assert_equal(decres, dectgt)
2175: (12)        incres = interp(incpts, xp, yp, right=2)
2176: (12)        decres = interp(decpts, xp, yp, right=2)
2177: (12)        inctgt = np.array([1, 1, 1, 2], dtype=float)
2178: (12)        dectgt = inctgt[::-1]
2179: (12)        assert_equal(incres, inctgt)
2180: (12)        assert_equal(decres, dectgt)
2181: (12)        incres = interp(incpts, xp, yp, left=0, right=2)
2182: (12)        decres = interp(decpts, xp, yp, left=0, right=2)
2183: (12)        inctgt = np.array([0, 1, 1, 2], dtype=float)
2184: (12)        dectgt = inctgt[::-1]
2185: (12)        assert_equal(incres, inctgt)
2186: (12)        assert_equal(decres, dectgt)
2187: (4)    def test_scalar_interpolation_point(self):
2188: (8)      x = np.linspace(0, 1, 5)
2189: (8)      y = np.linspace(0, 1, 5)
2190: (8)      x0 = 0
2191: (8)      assert_almost_equal(np.interp(x0, x, y), x0)
2192: (8)      x0 = .3
2193: (8)      assert_almost_equal(np.interp(x0, x, y), x0)
2194: (8)      x0 = np.float32(.3)
2195: (8)      assert_almost_equal(np.interp(x0, x, y), x0)
2196: (8)      x0 = np.float64(.3)
2197: (8)      assert_almost_equal(np.interp(x0, x, y), x0)
2198: (8)      x0 = np.nan
2199: (8)      assert_almost_equal(np.interp(x0, x, y), x0)
2200: (4)    def test_non_finite_behavior_exact_x(self):
2201: (8)      x = [1, 2, 2.5, 3, 4]
2202: (8)      xp = [1, 2, 3, 4]
2203: (8)      fp = [1, 2, np.inf, 4]
2204: (8)      assert_almost_equal(np.interp(x, xp, fp), [1, 2, np.inf, np.inf, 4])
2205: (8)      fp = [1, 2, np.nan, 4]
2206: (8)      assert_almost_equal(np.interp(x, xp, fp), [1, 2, np.nan, np.nan, 4])
2207: (4)    @pytest.fixture(params=[
2208: (8)      lambda x: np.float_(x),
2209: (8)      lambda x: _make_complex(x, 0),
2210: (8)      lambda x: _make_complex(0, x),

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

2211: (8)           lambda x: _make_complex(x, np.multiply(x, -2))
2212: (4)           ], ids=[
2213: (8)             'real',
2214: (8)             'complex-real',
2215: (8)             'complex-imag',
2216: (8)             'complex-both'
2217: (4)
2218: (4)           def sc(self, request):
2219: (8)             """ scale function used by the below tests """
2220: (8)             return request.param
2221: (4)           def test_non_finite_any_nan(self, sc):
2222: (8)             """ test that nans are propagated """
2223: (8)             assert_equal(np.interp(0.5, [np.nan, 1], sc([0, 10])), 0)
2224: (8)             assert_equal(np.interp(0.5, [0, np.nan], sc([0, 10])), 0)
2225: (8)             assert_equal(np.interp(0.5, [0, 1], sc([np.nan, 10])), 0)
2226: (8)             assert_equal(np.interp(0.5, [0, 1], sc([0, np.nan])), 0)
2227: (4)           def test_non_finite_inf(self, sc):
2228: (8)             """ Test that interp between opposite infs gives nan """
2229: (8)             assert_equal(np.interp(0.5, [-np.inf, +np.inf], sc([-10, 0])), 0)
2230: (8)             assert_equal(np.interp(0.5, [0, 1], sc([-np.inf, +np.inf])), 0)
2231: (8)             assert_equal(np.interp(0.5, [0, 1], sc([+np.inf, -np.inf])), 0)
2232: (8)             assert_equal(np.interp(0.5, [-np.inf, +np.inf], sc([-10, 10])), 0)
2233: (4)           def test_non_finite_half_inf_xf(self, sc):
2234: (8)             """ Test that interp where both axes have a bound at inf gives nan """
2235: (8)             assert_equal(np.interp(0.5, [-np.inf, 1], sc([-np.inf, 0])), 0)
2236: (8)             assert_equal(np.interp(0.5, [-np.inf, 1], sc([+np.inf, 0])), 0)
2237: (8)             assert_equal(np.interp(0.5, [-np.inf, 1], sc([0, -np.inf])), 0)
2238: (8)             assert_equal(np.interp(0.5, [-np.inf, 1], sc([0, +np.inf])), 0)
2239: (8)             assert_equal(np.interp(0.5, [0, +np.inf], sc([-np.inf, 0])), 0)
2240: (8)             assert_equal(np.interp(0.5, [0, +np.inf], sc([+np.inf, 0])), 0)
2241: (8)             assert_equal(np.interp(0.5, [0, +np.inf], sc([0, -np.inf])), 0)
2242: (8)             assert_equal(np.interp(0.5, [0, +np.inf], sc([0, +np.inf])), 0)
2243: (4)           def test_non_finite_half_inf_x(self, sc):
2244: (8)             """ Test interp where the x axis has a bound at inf """
2245: (8)             assert_equal(np.interp(0.5, [-np.inf, -np.inf], sc([0, 10])), 0)
2246: (8)             assert_equal(np.interp(0.5, [-np.inf, 1], sc([0, 10])), 0)
2247: (8)             assert_equal(np.interp(0.5, [0, +np.inf], sc([0, 10])), 0)
2248: (8)             assert_equal(np.interp(0.5, [+np.inf, +np.inf], sc([0, 10])), 0)
2249: (4)           def test_non_finite_half_inf_f(self, sc):
2250: (8)             """ Test interp where the f axis has a bound at inf """
2251: (8)             assert_equal(np.interp(0.5, [0, 1], sc([0, -np.inf])), 0)
2252: (8)             assert_equal(np.interp(0.5, [0, 1], sc([0, +np.inf])), 0)
2253: (8)             assert_equal(np.interp(0.5, [0, 1], sc([-np.inf, 10])), 0)
2254: (8)             assert_equal(np.interp(0.5, [0, 1], sc([+np.inf, 10])), 0)
2255: (8)             assert_equal(np.interp(0.5, [0, 1], sc([-np.inf, -np.inf])), 0)
2256: (8)             assert_equal(np.interp(0.5, [0, 1], sc([+np.inf, +np.inf])), 0)
2257: (4)           def test_complex_interp(self):

```

```

2258: (8)             x = np.linspace(0, 1, 5)
2259: (8)             y = np.linspace(0, 1, 5) + (1 + np.linspace(0, 1, 5))*1.0j
2260: (8)             x0 = 0.3
2261: (8)             y0 = x0 + (1+x0)*1.0j
2262: (8)             assert_almost_equal(np.interp(x0, x, y), y0)
2263: (8)             x0 = -1
2264: (8)             left = 2 + 3.0j
2265: (8)             assert_almost_equal(np.interp(x0, x, y, left=left), left)
2266: (8)             x0 = 2.0
2267: (8)             right = 2 + 3.0j
2268: (8)             assert_almost_equal(np.interp(x0, x, y, right=right), right)
2269: (8)             x = [1, 2, 2.5, 3, 4]
2270: (8)             xp = [1, 2, 3, 4]
2271: (8)             fp = [1, 2+1j, np.inf, 4]
2272: (8)             y = [1, 2+1j, np.inf+0.5j, np.inf, 4]
2273: (8)             assert_almost_equal(np.interp(x, xp, fp), y)
2274: (8)             x = [-180, -170, -185, 185, -10, -5, 0, 365]
2275: (8)             xp = [190, -190, 350, -350]
2276: (8)             fp = [5+1.0j, 10+2j, 3+3j, 4+4j]
2277: (8)             y = [7.5+1.5j, 5.+1.0j, 8.75+1.75j, 6.25+1.25j, 3.+3j, 3.25+3.25j,
2278: (13)              3.5+3.5j, 3.75+3.75j]
2279: (8)             assert_almost_equal(np.interp(x, xp, fp, period=360), y)
2280: (4)             def test_zero_dimensional_interpolation_point(self):
2281: (8)                 x = np.linspace(0, 1, 5)
2282: (8)                 y = np.linspace(0, 1, 5)
2283: (8)                 x0 = np.array(.3)
2284: (8)                 assert_almost_equal(np.interp(x0, x, y), x0)
2285: (8)                 xp = np.array([0, 2, 4])
2286: (8)                 fp = np.array([1, -1, 1])
2287: (8)                 actual = np.interp(np.array(1), xp, fp)
2288: (8)                 assert_equal(actual, 0)
2289: (8)                 assert_(isinstance(actual, np.float64))
2290: (8)                 actual = np.interp(np.array(4.5), xp, fp, period=4)
2291: (8)                 assert_equal(actual, 0.5)
2292: (8)                 assert_(isinstance(actual, np.float64))
2293: (4)             def test_if_len_x_is_small(self):
2294: (8)                 xp = np.arange(0, 10, 0.0001)
2295: (8)                 fp = np.sin(xp)
2296: (8)                 assert_almost_equal(np.interp(np.pi, xp, fp), 0.0)
2297: (4)             def test_period(self):
2298: (8)                 x = [-180, -170, -185, 185, -10, -5, 0, 365]
2299: (8)                 xp = [190, -190, 350, -350]
2300: (8)                 fp = [5, 10, 3, 4]
2301: (8)                 y = [7.5, 5., 8.75, 6.25, 3., 3.25, 3.5, 3.75]
2302: (8)                 assert_almost_equal(np.interp(x, xp, fp, period=360), y)
2303: (8)                 x = np.array(x, order='F').reshape(2, -1)
2304: (8)                 y = np.array(y, order='C').reshape(2, -1)
2305: (8)                 assert_almost_equal(np.interp(x, xp, fp, period=360), y)
2306: (0)             class TestPercentile:
2307: (4)                 def test_basic(self):
2308: (8)                     x = np.arange(8) * 0.5
2309: (8)                     assert_equal(np.percentile(x, 0), 0.)
2310: (8)                     assert_equal(np.percentile(x, 100), 3.5)
2311: (8)                     assert_equal(np.percentile(x, 50), 1.75)
2312: (8)                     x[1] = np.nan
2313: (8)                     assert_equal(np.percentile(x, 0), np.nan)
2314: (8)                     assert_equal(np.percentile(x, 0, method='nearest'), np.nan)
2315: (4)                 def test_fraction(self):
2316: (8)                     x = [Fraction(i, 2) for i in range(8)]
2317: (8)                     p = np.percentile(x, Fraction(0))
2318: (8)                     assert_equal(p, Fraction(0))
2319: (8)                     assert_equal(type(p), Fraction)
2320: (8)                     p = np.percentile(x, Fraction(100))
2321: (8)                     assert_equal(p, Fraction(7, 2))
2322: (8)                     assert_equal(type(p), Fraction)
2323: (8)                     p = np.percentile(x, Fraction(50))
2324: (8)                     assert_equal(p, Fraction(7, 4))
2325: (8)                     assert_equal(type(p), Fraction)
2326: (8)                     p = np.percentile(x, [Fraction(50)])

```

```

2327: (8)             assert_equal(p, np.array([Fraction(7, 4)]))
2328: (8)             assert_equal(type(p), np.ndarray)
2329: (4)             def test_api(self):
2330: (8)                 d = np.ones(5)
2331: (8)                 np.percentile(d, 5, None, None, False)
2332: (8)                 np.percentile(d, 5, None, None, False, 'linear')
2333: (8)                 o = np.ones((1,))
2334: (8)                 np.percentile(d, 5, None, o, False, 'linear')
2335: (4)             def test_complex(self):
2336: (8)                 arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='G')
2337: (8)                 assert_raises(TypeError, np.percentile, arr_c, 0.5)
2338: (8)                 arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='D')
2339: (8)                 assert_raises(TypeError, np.percentile, arr_c, 0.5)
2340: (8)                 arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='F')
2341: (8)                 assert_raises(TypeError, np.percentile, arr_c, 0.5)
2342: (4)             def test_2D(self):
2343: (8)                 x = np.array([[1, 1, 1],
2344: (22)                         [1, 1, 1],
2345: (22)                         [4, 4, 3],
2346: (22)                         [1, 1, 1],
2347: (22)                         [1, 1, 1]])
2348: (8)                 assert_array_equal(np.percentile(x, 50, axis=0), [1, 1, 1])
2349: (4)             @pytest.mark.parametrize("dtype", np.typecodes["Float"])
2350: (4)             def test_linear_nan_1D(self, dtype):
2351: (8)                 arr = np.asarray([15.0, np.NAN, 35.0, 40.0, 50.0], dtype=dtype)
2352: (8)                 res = np.percentile(
2353: (12)                     arr,
2354: (12)                     40.0,
2355: (12)                     method="linear")
2356: (8)                 np.testing.assert_equal(res, np.NAN)
2357: (8)                 np.testing.assert_equal(res.dtype, arr.dtype)
2358: (4)             H_F_TYPE_CODES = [(int_type, np.float64)
2359: (22)                             for int_type in np.typecodes["AllInteger"]
2360: (22)                             ] + [(np.float16, np.float16),
2361: (27)                                 (np.float32, np.float32),
2362: (27)                                 (np.float64, np.float64),
2363: (27)                                 (np.longdouble, np.longdouble),
2364: (27)                                 (np.dtype("O"), np.float64)]
2365: (4)             @pytest.mark.parametrize(["input_dtype", "expected_dtype"],
2366: (4)             H_F_TYPE_CODES)
2367: (29)             @pytest.mark.parametrize(["method", "expected"],
2368: (30)                             [("inverted_cdf", 20),
2369: (30)                             ("averaged_inverted_cdf", 27.5),
2370: (30)                             ("closest_observation", 20),
2371: (30)                             ("interpolated_inverted_cdf", 20),
2372: (30)                             ("hazen", 27.5),
2373: (30)                             ("weibull", 26),
2374: (30)                             ("linear", 29),
2375: (30)                             ("median_unbiased", 27),
2376: (30)                             ("normal_unbiased", 27.125),
2377: (4)                             ])
2378: (34)             def test_linear_interpolation(self,
2379: (34)                             method,
2380: (34)                             expected,
2381: (34)                             input_dtype,
2382: (34)                             expected_dtype):
2383: (8)                 expected_dtype = np.dtype(expected_dtype)
2384: (12)                 if np._get_promotion_state() == "legacy":
2385: (8)                     expected_dtype = np.promote_types(expected_dtype, np.float64)
2386: (8)                 arr = np.asarray([15.0, 20.0, 35.0, 40.0, 50.0], dtype=input_dtype)
2387: (8)                 actual = np.percentile(arr, 40.0, method=method)
2388: (12)                 np.testing.assert_almost_equal(
2389: (8)                     actual, expected_dtype.type(expected), 14)
2390: (12)                 if method in ["inverted_cdf", "closest_observation"]:
2391: (16)                     if input_dtype == "O":
2392: (12)                         np.testing.assert_equal(np.asarray(actual).dtype, np.float64)
2393: (16)                     else:
2394: (40)                         np.testing.assert_equal(np.asarray(actual).dtype,
2395:                                         np.dtype(input_dtype))

```

```

2395: (8)
2396: (12)
2397: (36)
2398: (4)
2399: (4)
2400: (4)
2401: (8)
2402: (35)
2403: (8)
2404: (35)
2405: (4)
2406: (4)
2407: (8)
2408: (35)
2409: (8)
2410: (35)
2411: (8)
2412: (35)
2413: (8)
2414: (35)
2415: (4)
2416: (4)
2417: (8)
2418: (35)
2419: (8)
2420: (35)
2421: (4)
2422: (8)
2423: (8)
2424: (8)
2425: (8)
2426: (8)
2427: (4)
2428: (8)
2429: (8)
2430: (4)
2431: (8)
2432: (8)
2433: (8)
2434: (8)
2435: (8)
2436: (8)
2437: (8)
2438: (8)
2439: (8)
2440: (8)
2441: (8)
2442: (8)
2443: (8)
2444: (8)
2445: (12)
2446: (8)
2447: (35)
2448: (8)
2449: (35)
2450: (8)
2451: (35)
2452: (8)
2453: (35)
2454: (8)
2455: (35)
2456: (8)
2457: (35)
2458: (8)
2459: (35)
2460: (4)
2461: (8)
2462: (8)
2463: (8)

        else:
            np.testing.assert_equal(np.asarray(actual).dtype,
                                  np.dtype(expected_dtype))
    TYPE_CODES = np.typecodes["AllInteger"] + np.typecodes["Float"] + "O"
    @pytest.mark.parametrize("dtype", TYPE_CODES)
    def test_lower_higher(self, dtype):
        assert_equal(np.percentile(np.arange(10, dtype=dtype), 50,
                                  method='lower'), 4)
        assert_equal(np.percentile(np.arange(10, dtype=dtype), 50,
                                  method='higher'), 5)
    @pytest.mark.parametrize("dtype", TYPE_CODES)
    def test_midpoint(self, dtype):
        assert_equal(np.percentile(np.arange(10, dtype=dtype), 51,
                                  method='midpoint'), 4.5)
        assert_equal(np.percentile(np.arange(9, dtype=dtype) + 1, 50,
                                  method='midpoint'), 5)
        assert_equal(np.percentile(np.arange(11, dtype=dtype), 51,
                                  method='midpoint'), 5.5)
        assert_equal(np.percentile(np.arange(11, dtype=dtype), 50,
                                  method='midpoint'), 5)
    @pytest.mark.parametrize("dtype", TYPE_CODES)
    def test_nearest(self, dtype):
        assert_equal(np.percentile(np.arange(10, dtype=dtype), 51,
                                  method='nearest'), 5)
        assert_equal(np.percentile(np.arange(10, dtype=dtype), 49,
                                  method='nearest'), 4)
    def test_linear_interpolation_extrapolation(self):
        arr = np.random.rand(5)
        actual = np.percentile(arr, 100)
        np.testing.assert_equal(actual, arr.max())
        actual = np.percentile(arr, 0)
        np.testing.assert_equal(actual, arr.min())
    def test_sequence(self):
        x = np.arange(8) * 0.5
        assert_equal(np.percentile(x, [0, 100, 50]), [0, 3.5, 1.75])
    def test_axis(self):
        x = np.arange(12).reshape(3, 4)
        assert_equal(np.percentile(x, (25, 50, 100)), [2.75, 5.5, 11.0])
        r0 = [[2, 3, 4, 5], [4, 5, 6, 7], [8, 9, 10, 11]]
        assert_equal(np.percentile(x, (25, 50, 100), axis=0), r0)
        r1 = [[0.75, 1.5, 3], [4.75, 5.5, 7], [8.75, 9.5, 11]]
        assert_equal(np.percentile(x, (25, 50, 100), axis=1), np.array(r1).T)
        x = np.arange(3 * 4 * 5 * 6).reshape(3, 4, 5, 6)
        assert_equal(np.percentile(x, (25, 50)).shape, (2,))
        assert_equal(np.percentile(x, (25, 50, 75)).shape, (3,))
        assert_equal(np.percentile(x, (25, 50), axis=0).shape, (2, 4, 5, 6))
        assert_equal(np.percentile(x, (25, 50), axis=1).shape, (2, 3, 5, 6))
        assert_equal(np.percentile(x, (25, 50), axis=2).shape, (2, 3, 4, 6))
        assert_equal(np.percentile(x, (25, 50), axis=3).shape, (2, 3, 4, 5))
        assert_equal(
            np.percentile(x, (25, 50, 75), axis=1).shape, (3, 3, 5, 6))
        assert_equal(np.percentile(x, (25, 50),
                                  method="higher").shape, (2,))
        assert_equal(np.percentile(x, (25, 50, 75),
                                  method="higher").shape, (3,))
        assert_equal(np.percentile(x, (25, 50), axis=0,
                                  method="higher").shape, (2, 4, 5, 6))
        assert_equal(np.percentile(x, (25, 50), axis=1,
                                  method="higher").shape, (2, 3, 5, 6))
        assert_equal(np.percentile(x, (25, 50), axis=2,
                                  method="higher").shape, (2, 3, 4, 6))
        assert_equal(np.percentile(x, (25, 50), axis=3,
                                  method="higher").shape, (2, 3, 4, 5))
        assert_equal(np.percentile(x, (25, 50, 75), axis=1,
                                  method="higher").shape, (3, 3, 5, 6))
    def test_scalar_q(self):
        x = np.arange(12).reshape(3, 4)
        assert_equal(np.percentile(x, 50), 5.5)
        assert_(np.isscalar(np.percentile(x, 50)))

```

```

2464: (8)          r0 = np.array([4.,  5.,  6.,  7.])
2465: (8)          assert_equal(np.percentile(x, 50, axis=0), r0)
2466: (8)          assert_equal(np.percentile(x, 50, axis=0).shape, r0.shape)
2467: (8)          r1 = np.array([1.5,  5.5,  9.5])
2468: (8)          assert_almost_equal(np.percentile(x, 50, axis=1), r1)
2469: (8)          assert_equal(np.percentile(x, 50, axis=1).shape, r1.shape)
2470: (8)          out = np.empty(1)
2471: (8)          assert_equal(np.percentile(x, 50, out=out), 5.5)
2472: (8)          assert_equal(out, 5.5)
2473: (8)          out = np.empty(4)
2474: (8)          assert_equal(np.percentile(x, 50, axis=0, out=out), r0)
2475: (8)          assert_equal(out, r0)
2476: (8)          out = np.empty(3)
2477: (8)          assert_equal(np.percentile(x, 50, axis=1, out=out), r1)
2478: (8)          assert_equal(out, r1)
2479: (8)          x = np.arange(12).reshape(3, 4)
2480: (8)          assert_equal(np.percentile(x, 50, method='lower'), 5.)
2481: (8)          assert_(np.isscalar(np.percentile(x, 50)))
2482: (8)          r0 = np.array([4.,  5.,  6.,  7.])
2483: (8)          c0 = np.percentile(x, 50, method='lower', axis=0)
2484: (8)          assert_equal(c0, r0)
2485: (8)          assert_equal(c0.shape, r0.shape)
2486: (8)          r1 = np.array([1.,  5.,  9.])
2487: (8)          c1 = np.percentile(x, 50, method='lower', axis=1)
2488: (8)          assert_almost_equal(c1, r1)
2489: (8)          assert_equal(c1.shape, r1.shape)
2490: (8)          out = np.empty(), dtype=x.dtype)
2491: (8)          c = np.percentile(x, 50, method='lower', out=out)
2492: (8)          assert_equal(c, 5)
2493: (8)          assert_equal(out, 5)
2494: (8)          out = np.empty(4, dtype=x.dtype)
2495: (8)          c = np.percentile(x, 50, method='lower', axis=0, out=out)
2496: (8)          assert_equal(c, r0)
2497: (8)          assert_equal(out, r0)
2498: (8)          out = np.empty(3, dtype=x.dtype)
2499: (8)          c = np.percentile(x, 50, method='lower', axis=1, out=out)
2500: (8)          assert_equal(c, r1)
2501: (8)          assert_equal(out, r1)
2502: (4)          def test_exception(self):
2503: (8)              assert_raises(ValueError, np.percentile, [1, 2], 56,
2504: (22)                            method='foobar')
2505: (8)              assert_raises(ValueError, np.percentile, [1], 101)
2506: (8)              assert_raises(ValueError, np.percentile, [1], -1)
2507: (8)              assert_raises(ValueError, np.percentile, [1], list(range(50)) + [101])
2508: (8)              assert_raises(ValueError, np.percentile, [1], list(range(50)) +
[-0.1])
2509: (4)          def test_percentile_list(self):
2510: (8)              assert_equal(np.percentile([1, 2, 3], 0), 1)
2511: (4)          def test_percentile_out(self):
2512: (8)              x = np.array([1, 2, 3])
2513: (8)              y = np.zeros((3,))
2514: (8)              p = (1, 2, 3)
2515: (8)              np.percentile(x, p, out=y)
2516: (8)              assert_equal(np.percentile(x, p), y)
2517: (8)              x = np.array([[1, 2, 3],
2518: (22)                            [4, 5, 6]])
2519: (8)              y = np.zeros((3, 3))
2520: (8)              np.percentile(x, p, axis=0, out=y)
2521: (8)              assert_equal(np.percentile(x, p, axis=0), y)
2522: (8)              y = np.zeros((3, 2))
2523: (8)              np.percentile(x, p, axis=1, out=y)
2524: (8)              assert_equal(np.percentile(x, p, axis=1), y)
2525: (8)              x = np.arange(12).reshape(3, 4)
2526: (8)              r0 = np.array([[2.,  3.,  4.,  5.], [4.,  5.,  6.,  7.]])
2527: (8)              out = np.empty((2, 4))
2528: (8)              assert_equal(np.percentile(x, (25, 50), axis=0, out=out), r0)
2529: (8)              assert_equal(out, r0)
2530: (8)              r1 = np.array([[0.75,  4.75,  8.75], [1.5,   5.5,   9.5]])
2531: (8)              out = np.empty((2, 3))

```

```

2532: (8) assert_equal(np.percentile(x, (25, 50), axis=1, out=out), r1)
2533: (8) assert_equal(out, r1)
2534: (8) r0 = np.array([[0, 1, 2, 3], [4, 5, 6, 7]])
2535: (8) out = np.empty((2, 4), dtype=x.dtype)
2536: (8) c = np.percentile(x, (25, 50), method='lower', axis=0, out=out)
2537: (8) assert_equal(c, r0)
2538: (8) assert_equal(out, r0)
2539: (8) r1 = np.array([[0, 4, 8], [1, 5, 9]])
2540: (8) out = np.empty((2, 3), dtype=x.dtype)
2541: (8) c = np.percentile(x, (25, 50), method='lower', axis=1, out=out)
2542: (8) assert_equal(c, r1)
2543: (8) assert_equal(out, r1)
2544: (4) def test_percentile_empty_dim(self):
2545: (8)     d = np.arange(11 * 2).reshape(11, 1, 2, 1)
2546: (8)     assert_array_equal(np.percentile(d, 50, axis=0).shape, (1, 2, 1))
2547: (8)     assert_array_equal(np.percentile(d, 50, axis=1).shape, (11, 2, 1))
2548: (8)     assert_array_equal(np.percentile(d, 50, axis=2).shape, (11, 1, 1))
2549: (8)     assert_array_equal(np.percentile(d, 50, axis=3).shape, (11, 1, 2))
2550: (8)     assert_array_equal(np.percentile(d, 50, axis=-1).shape, (11, 1, 2))
2551: (8)     assert_array_equal(np.percentile(d, 50, axis=-2).shape, (11, 1, 1))
2552: (8)     assert_array_equal(np.percentile(d, 50, axis=-3).shape, (11, 2, 1))
2553: (8)     assert_array_equal(np.percentile(d, 50, axis=-4).shape, (1, 2, 1))
2554: (8)     assert_array_equal(np.percentile(d, 50, axis=2,
2555: (41)                                     method='midpoint').shape,
2556: (27)                               (11, 1, 1))
2557: (8)     assert_array_equal(np.percentile(d, 50, axis=-2,
2558: (41)                                     method='midpoint').shape,
2559: (27)                               (11, 1, 1))
2560: (8)     assert_array_equal(np.array(np.percentile(d, [10, 50], axis=0)).shape,
2561: (27)                               (2, 1, 2, 1))
2562: (8)     assert_array_equal(np.array(np.percentile(d, [10, 50], axis=1)).shape,
2563: (27)                               (2, 11, 2, 1))
2564: (8)     assert_array_equal(np.array(np.percentile(d, [10, 50], axis=2)).shape,
2565: (27)                               (2, 11, 1, 1))
2566: (8)     assert_array_equal(np.array(np.percentile(d, [10, 50], axis=3)).shape,
2567: (27)                               (2, 11, 1, 2))
2568: (4) def test_percentile_no_overwrite(self):
2569: (8)     a = np.array([2, 3, 4, 1])
2570: (8)     np.percentile(a, [50], overwrite_input=False)
2571: (8)     assert_equal(a, np.array([2, 3, 4, 1]))
2572: (8)     a = np.array([2, 3, 4, 1])
2573: (8)     np.percentile(a, [50])
2574: (8)     assert_equal(a, np.array([2, 3, 4, 1]))
2575: (4) def test_no_p_overwrite(self):
2576: (8)     p = np.linspace(0., 100., num=5)
2577: (8)     np.percentile(np.arange(100.), p, method="midpoint")
2578: (8)     assert_array_equal(p, np.linspace(0., 100., num=5))
2579: (8)     p = np.linspace(0., 100., num=5).tolist()
2580: (8)     np.percentile(np.arange(100.), p, method="midpoint")
2581: (8)     assert_array_equal(p, np.linspace(0., 100., num=5).tolist())
2582: (4) def test_percentile_overwrite(self):
2583: (8)     a = np.array([2, 3, 4, 1])
2584: (8)     b = np.percentile(a, [50], overwrite_input=True)
2585: (8)     assert_equal(b, np.array([2.5]))
2586: (8)     b = np.percentile([2, 3, 4, 1], [50], overwrite_input=True)
2587: (8)     assert_equal(b, np.array([2.5]))
2588: (4) def test_extended_axis(self):
2589: (8)     o = np.random.normal(size=(71, 23))
2590: (8)     x = np.dstack([o] * 10)
2591: (8)     assert_equal(np.percentile(x, 30, axis=(0, 1)), np.percentile(o, 30))
2592: (8)     x = np.moveaxis(x, -1, 0)
2593: (8)     assert_equal(np.percentile(x, 30, axis=(-2, -1)), np.percentile(o,
2594: (8)         30))
2595: (8)     x = x.swapaxes(0, 1).copy()
2596: (8)     assert_equal(np.percentile(x, 30, axis=(0, -1)), np.percentile(o, 30))
2597: (8)     x = x.swapaxes(0, 1).copy()
2598: (21)    assert_equal(np.percentile(x, [25, 60], axis=(0, 1, 2)),
2599: (8)                               np.percentile(x, [25, 60], axis=None))
2599: (8)    assert_equal(np.percentile(x, [25, 60], axis=(0,)),
```

```

2600: (21) np.percentile(x, [25, 60], axis=0))
2601: (8) d = np.arange(3 * 5 * 7 * 11).reshape((3, 5, 7, 11))
2602: (8) np.random.shuffle(d.ravel())
2603: (8) assert_equal(np.percentile(d, 25, axis=(0, 1, 2))[0],
2604: (21) np.percentile(d[:, :, :, 0].flatten(), 25))
2605: (8) assert_equal(np.percentile(d, [10, 90], axis=(0, 1, 3))[:, 1],
2606: (21) np.percentile(d[:, :, 1, :].flatten(), [10, 90]))
2607: (8) assert_equal(np.percentile(d, 25, axis=(3, 1, -4))[2],
2608: (21) np.percentile(d[:, :, 2, :].flatten(), 25))
2609: (8) assert_equal(np.percentile(d, 25, axis=(3, 1, 2))[2],
2610: (21) np.percentile(d[2, :, :, :].flatten(), 25))
2611: (8) assert_equal(np.percentile(d, 25, axis=(3, 2))[2, 1],
2612: (21) np.percentile(d[2, 1, :, :].flatten(), 25))
2613: (8) assert_equal(np.percentile(d, 25, axis=(1, -2))[2, 1],
2614: (21) np.percentile(d[2, :, :, 1].flatten(), 25))
2615: (8) assert_equal(np.percentile(d, 25, axis=(1, 3))[2, 2],
2616: (21) np.percentile(d[2, :, 2, :].flatten(), 25))
2617: (4) def test_extended_axis_invalid(self):
2618: (8) d = np.ones((3, 5, 7, 11))
2619: (8) assert_raises(np.AxisError, np.percentile, d, axis=-5, q=25)
2620: (8) assert_raises(np.AxisError, np.percentile, d, axis=(0, -5), q=25)
2621: (8) assert_raises(np.AxisError, np.percentile, d, axis=4, q=25)
2622: (8) assert_raises(np.AxisError, np.percentile, d, axis=(0, 4), q=25)
2623: (8) assert_raises(ValueError, np.percentile, d, axis=(1, 1), q=25)
2624: (8) assert_raises(ValueError, np.percentile, d, axis=(-1, -1), q=25)
2625: (8) assert_raises(ValueError, np.percentile, d, axis=(3, -1), q=25)
2626: (4) def test_keepdims(self):
2627: (8) d = np.ones((3, 5, 7, 11))
2628: (8) assert_equal(np.percentile(d, 7, axis=None, keepdims=True).shape,
2629: (21) (1, 1, 1, 1))
2630: (8) assert_equal(np.percentile(d, 7, axis=(0, 1), keepdims=True).shape,
2631: (21) (1, 1, 7, 11))
2632: (8) assert_equal(np.percentile(d, 7, axis=(0, 3), keepdims=True).shape,
2633: (21) (1, 5, 7, 1))
2634: (8) assert_equal(np.percentile(d, 7, axis=(1,), keepdims=True).shape,
2635: (21) (3, 1, 7, 11))
2636: (8) assert_equal(np.percentile(d, 7, (0, 1, 2, 3), keepdims=True).shape,
2637: (21) (1, 1, 1, 1))
2638: (8) assert_equal(np.percentile(d, 7, axis=(0, 1, 3), keepdims=True).shape,
2639: (21) (1, 1, 7, 1))
2640: (8) assert_equal(np.percentile(d, [1, 7], axis=(0, 1, 3),
2641: (35) keepdims=True).shape, (2, 1, 1, 7, 1))
2642: (8) assert_equal(np.percentile(d, [1, 7], axis=(0, 3),
2643: (35) keepdims=True).shape, (2, 1, 5, 7, 1))
2644: (4) @pytest.mark.parametrize('q', [7, [1, 7]])
2645: (4) @pytest.mark.parametrize(
2646: (8) argnames='axis',
2647: (8) argvalues=[
2648: (12) None,
2649: (12) 1,
2650: (12) (1,),
2651: (12) (0, 1),
2652: (12) (-3, -1),
2653: (8) ],
2654: (4) )
2655: (4) def test_keepdims_out(self, q, axis):
2656: (8) d = np.ones((3, 5, 7, 11))
2657: (8) if axis is None:
2658: (12) shape_out = (1,) * d.ndim
2659: (8) else:
2660: (12) axis_norm = normalize_axis_tuple(axis, d.ndim)
2661: (12) shape_out = tuple(
2662: (16) 1 if i in axis_norm else d.shape[i] for i in range(d.ndim))
2663: (8) shape_out = np.shape(q) + shape_out
2664: (8) out = np.empty(shape_out)
2665: (8) result = np.percentile(d, q, axis=axis, keepdims=True, out=out)
2666: (8) assert result is out
2667: (8) assert_equal(result.shape, shape_out)
2668: (4) def test_out(self):

```

```

2669: (8)          o = np.zeros((4,))
2670: (8)          d = np.ones((3, 4))
2671: (8)          assert_equal(np.percentile(d, 0, 0, out=o), o)
2672: (8)          assert_equal(np.percentile(d, 0, 0, method='nearest', out=o), o)
2673: (8)          o = np.zeros((3,))
2674: (8)          assert_equal(np.percentile(d, 1, 1, out=o), o)
2675: (8)          assert_equal(np.percentile(d, 1, 1, method='nearest', out=o), o)
2676: (8)          o = np.zeros(())
2677: (8)          assert_equal(np.percentile(d, 2, out=o), o)
2678: (8)          assert_equal(np.percentile(d, 2, method='nearest', out=o), o)
2679: (4)          def test_out_nan(self):
2680: (8)              with warnings.catch_warnings(record=True):
2681: (12)                  warnings.filterwarnings('always', '', RuntimeWarning)
2682: (12)                  o = np.zeros((4,))
2683: (12)                  d = np.ones((3, 4))
2684: (12)                  d[2, 1] = np.nan
2685: (12)                  assert_equal(np.percentile(d, 0, 0, out=o), o)
2686: (12)                  assert_equal(
2687: (16)                      np.percentile(d, 0, 0, method='nearest', out=o), o)
2688: (12)                  o = np.zeros((3,))
2689: (12)                  assert_equal(np.percentile(d, 1, 1, out=o), o)
2690: (12)                  assert_equal(
2691: (16)                      np.percentile(d, 1, 1, method='nearest', out=o), o)
2692: (12)                  o = np.zeros(())
2693: (12)                  assert_equal(np.percentile(d, 1, out=o), o)
2694: (12)                  assert_equal(
2695: (16)                      np.percentile(d, 1, method='nearest', out=o), o)
2696: (4)          def test_nan_behavior(self):
2697: (8)              a = np.arange(24, dtype=float)
2698: (8)              a[2] = np.nan
2699: (8)              assert_equal(np.percentile(a, 0.3), np.nan)
2700: (8)              assert_equal(np.percentile(a, 0.3, axis=0), np.nan)
2701: (8)              assert_equal(np.percentile(a, [0.3, 0.6], axis=0),
2702: (21)                  np.array([np.nan] * 2))
2703: (8)              a = np.arange(24, dtype=float).reshape(2, 3, 4)
2704: (8)              a[1, 2, 3] = np.nan
2705: (8)              a[1, 1, 2] = np.nan
2706: (8)              assert_equal(np.percentile(a, 0.3), np.nan)
2707: (8)              assert_equal(np.percentile(a, 0.3).ndim, 0)
2708: (8)              b = np.percentile(np.arange(24, dtype=float).reshape(2, 3, 4), 0.3, 0)
2709: (8)              b[2, 3] = np.nan
2710: (8)              b[1, 2] = np.nan
2711: (8)              assert_equal(np.percentile(a, 0.3, 0), b)
2712: (8)              b = np.percentile(np.arange(24, dtype=float).reshape(2, 3, 4),
2713: (26)                  [0.3, 0.6], 0)
2714: (8)              b[:, 2, 3] = np.nan
2715: (8)              b[:, 1, 2] = np.nan
2716: (8)              assert_equal(np.percentile(a, [0.3, 0.6], 0), b)
2717: (8)              b = np.percentile(np.arange(24, dtype=float).reshape(2, 3, 4), 0.3, 1)
2718: (8)              b[1, 3] = np.nan
2719: (8)              b[1, 2] = np.nan
2720: (8)              assert_equal(np.percentile(a, 0.3, 1), b)
2721: (8)              b = np.percentile(
2722: (12)                  np.arange(24, dtype=float).reshape(2, 3, 4), [0.3, 0.6], 1)
2723: (8)              b[:, 1, 3] = np.nan
2724: (8)              b[:, 1, 2] = np.nan
2725: (8)              assert_equal(np.percentile(a, [0.3, 0.6], 1), b)
2726: (8)              b = np.percentile(
2727: (12)                  np.arange(24, dtype=float).reshape(2, 3, 4), 0.3, (0, 2))
2728: (8)              b[1] = np.nan
2729: (8)              b[2] = np.nan
2730: (8)              assert_equal(np.percentile(a, 0.3, (0, 2)), b)
2731: (8)              b = np.percentile(np.arange(24, dtype=float).reshape(2, 3, 4),
2732: (26)                  [0.3, 0.6], (0, 2))
2733: (8)              b[:, 1] = np.nan
2734: (8)              b[:, 2] = np.nan
2735: (8)              assert_equal(np.percentile(a, [0.3, 0.6], (0, 2)), b)
2736: (8)              b = np.percentile(np.arange(24, dtype=float).reshape(2, 3, 4),
2737: (26)                  [0.3, 0.6], (0, 2), method='nearest')

```

```

2738: (8)             b[:, 1] = np.nan
2739: (8)             b[:, 2] = np.nan
2740: (8)             assert_equal(np.percentile(
2741: (12)               a, [0.3, 0.6], (0, 2), method='nearest'), b)
2742: (4)             def test_nan_q(self):
2743: (8)                 with pytest.raises(ValueError, match="Percentiles must be in"):
2744: (12)                   np.percentile([1, 2, 3, 4.0], np.nan)
2745: (8)                 with pytest.raises(ValueError, match="Percentiles must be in"):
2746: (12)                   np.percentile([1, 2, 3, 4.0], [np.nan])
2747: (8)                 q = np.linspace(1.0, 99.0, 16)
2748: (8)                 q[0] = np.nan
2749: (8)                 with pytest.raises(ValueError, match="Percentiles must be in"):
2750: (12)                   np.percentile([1, 2, 3, 4.0], q)
2751: (4)             @pytest.mark.parametrize("dtype", ["m8[D]", "M8[s]"])
2752: (4)             @pytest.mark.parametrize("pos", [0, 23, 10])
2753: (4)             def test_nat_basic(self, dtype, pos):
2754: (8)                 a = np.arange(0, 24, dtype=dtype)
2755: (8)                 a[pos] = "NaT"
2756: (8)                 res = np.percentile(a, 30)
2757: (8)                 assert res.dtype == dtype
2758: (8)                 assert np.isnat(res)
2759: (8)                 res = np.percentile(a, [30, 60])
2760: (8)                 assert res.dtype == dtype
2761: (8)                 assert np.isnat(res).all()
2762: (8)                 a = np.arange(0, 24*3, dtype=dtype).reshape(-1, 3)
2763: (8)                 a[pos, 1] = "NaT"
2764: (8)                 res = np.percentile(a, 30, axis=0)
2765: (8)                 assert_array_equal(np.isnat(res), [False, True, False])
2766: (0)             quantile_methods = [
2767: (4)               'inverted_cdf', 'averaged_inverted_cdf', 'closest_observation',
2768: (4)               'interpolated_inverted_cdf', 'hazen', 'weibull', 'linear',
2769: (4)               'median_unbiased', 'normal_unbiased', 'nearest', 'lower', 'higher',
2770: (4)               'midpoint']
2771: (0)             class TestQuantile:
2772: (4)                 def V(self, x, y, alpha):
2773: (8)                     return (x >= y) - alpha
2774: (4)                 def test_max_ulp(self):
2775: (8)                     x = [0.0, 0.2, 0.4]
2776: (8)                     a = np.quantile(x, 0.45)
2777: (8)                     np.testing.assert_array_max_ulp(a, 0.18, maxulp=1)
2778: (4)                 def test_basic(self):
2779: (8)                     x = np.arange(8) * 0.5
2780: (8)                     assert_equal(np.quantile(x, 0), 0.)
2781: (8)                     assert_equal(np.quantile(x, 1), 3.5)
2782: (8)                     assert_equal(np.quantile(x, 0.5), 1.75)
2783: (4)                 def test_correct_quantile_value(self):
2784: (8)                     a = np.array([True])
2785: (8)                     tf_quant = np.quantile(True, False)
2786: (8)                     assert_equal(tf_quant, a[0])
2787: (8)                     assert_equal(type(tf_quant), a.dtype)
2788: (8)                     a = np.array([False, True, True])
2789: (8)                     quant_res = np.quantile(a, a)
2790: (8)                     assert_array_equal(quant_res, a)
2791: (8)                     assert_equal(quant_res.dtype, a.dtype)
2792: (4)                 def test_fraction(self):
2793: (8)                     x = [Fraction(i, 2) for i in range(8)]
2794: (8)                     q = np.quantile(x, 0)
2795: (8)                     assert_equal(q, 0)
2796: (8)                     assert_equal(type(q), Fraction)
2797: (8)                     q = np.quantile(x, 1)
2798: (8)                     assert_equal(q, Fraction(7, 2))
2799: (8)                     assert_equal(type(q), Fraction)
2800: (8)                     q = np.quantile(x, .5)
2801: (8)                     assert_equal(q, 1.75)
2802: (8)                     assert_equal(type(q), np.float64)
2803: (8)                     q = np.quantile(x, Fraction(1, 2))
2804: (8)                     assert_equal(q, Fraction(7, 4))
2805: (8)                     assert_equal(type(q), Fraction)
2806: (8)                     q = np.quantile(x, [Fraction(1, 2)])

```

```

2807: (8) assert_equal(q, np.array([Fraction(7, 4)]))
2808: (8) assert_equal(type(q), np.ndarray)
2809: (8) q = np.quantile(x, [[Fraction(1, 2)]])
2810: (8) assert_equal(q, np.array([[Fraction(7, 4)]]))
2811: (8) assert_equal(type(q), np.ndarray)
2812: (8) x = np.arange(8)
2813: (8) assert_equal(np.quantile(x, Fraction(1, 2)), Fraction(7, 2))
2814: (4) def test_complex(self):
2815: (8) arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='G')
2816: (8) assert_raises(TypeError, np.quantile, arr_c, 0.5)
2817: (8) arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='D')
2818: (8) assert_raises(TypeError, np.quantile, arr_c, 0.5)
2819: (8) arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='F')
2820: (8) assert_raises(TypeError, np.quantile, arr_c, 0.5)
2821: (4) def test_no_p_overwrite(self):
2822: (8) p0 = np.array([0, 0.75, 0.25, 0.5, 1.0])
2823: (8) p = p0.copy()
2824: (8) np.quantile(np.arange(100.), p, method="midpoint")
2825: (8) assert_array_equal(p, p0)
2826: (8) p0 = p0.tolist()
2827: (8) p = p.tolist()
2828: (8) np.quantile(np.arange(100.), p, method="midpoint")
2829: (8) assert_array_equal(p, p0)
2830: (4) @pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
2831: (4) def test_quantile_preserve_int_type(self, dtype):
2832: (8) res = np.quantile(np.array([1, 2], dtype=dtype), [0.5],
2833: (26) method="nearest")
2834: (8) assert res.dtype == dtype
2835: (4) @pytest.mark.parametrize("method", quantile_methods)
2836: (4) def test_quantile_monotonic(self, method):
2837: (8) p0 = np.linspace(0, 1, 101)
2838: (8) quantile = np.quantile(np.array([0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 1, 1,
9, 9, 9,
2839: (41) 8, 8, 7]) * 0.1, p0, method=method)
2840: (8) assert_equal(np.sort(quantile), quantile)
2841: (8) quantile = np.quantile([0., 1., 2., 3.], p0, method=method)
2842: (8) assert_equal(np.sort(quantile), quantile)
2843: (4) @hypothesis.given(
2844: (12) arr=arrays(dtype=np.float64,
2845: (23) shape=st.integers(min_value=3, max_value=1000),
2846: (23) elements=st.floats(allow_infinity=False,
allow_nan=False,
2847: (42) min_value=-1e300, max_value=1e300)))
2848: (4) def test_quantile_monotonic_hypo(self, arr):
2849: (8) p0 = np.arange(0, 1, 0.01)
2850: (8) quantile = np.quantile(arr, p0)
2851: (8) assert_equal(np.sort(quantile), quantile)
2852: (4) def test_quantile_scalar_nan(self):
2853: (8) a = np.array([[10., 7., 4.], [3., 2., 1.]])
2854: (8) a[0][1] = np.nan
2855: (8) actual = np.quantile(a, 0.5)
2856: (8) assert np.isscalar(actual)
2857: (8) assert_equal(np.quantile(a, 0.5), np.nan)
2858: (4) @pytest.mark.parametrize("method", quantile_methods)
2859: (4) @pytest.mark.parametrize("alpha", [0.2, 0.5, 0.9])
2860: (4) def test_quantile_identification_equation(self, method, alpha):
2861: (8) rng = np.random.default_rng(4321)
2862: (8) n = 102 # n * alpha = 20.4, 51. , 91.8
2863: (8) y = rng.random(n)
2864: (8) x = np.quantile(y, alpha, method=method)
2865: (8) if method in ("higher",):
2866: (12) assert np.abs(np.mean(self.V(x, y, alpha))) > 0.1 / n
2867: (8) elif int(n * alpha) == n * alpha:
2868: (12) assert_allclose(np.mean(self.V(x, y, alpha)), 0, atol=1e-14)
2869: (8) else:
2870: (12) assert_allclose(np.mean(self.V(x, y, alpha)), 0,
2871: (16) atol=1 / n / np.amin([alpha, 1 - alpha]))
2872: (4) @pytest.mark.parametrize("method", quantile_methods)
2873: (4) @pytest.mark.parametrize("alpha", [0.2, 0.5, 0.9])

```

```

2874: (4)             def test_quantile_add_and_multiply_constant(self, method, alpha):
2875: (8)                 rng = np.random.default_rng(4321)
2876: (8)                 n = 102 # n * alpha = 20.4, 51. , 91.8
2877: (8)                 y = rng.random(n)
2878: (8)                 q = np.quantile(y, alpha, method=method)
2879: (8)                 c = 13.5
2880: (8)                 assert_allclose(np.quantile(c + y, alpha, method=method), c + q)
2881: (8)                 assert_allclose(np.quantile(c * y, alpha, method=method), c * q)
2882: (8)                 q = -np.quantile(-y, 1 - alpha, method=method)
2883: (8)             if method == "inverted_cdf":
2884: (12)                 if (
2885: (16)                     n * alpha == int(n * alpha)
2886: (16)                     or np.round(n * alpha) == int(n * alpha) + 1
2887: (12)                 ):
2888: (16)                     assert_allclose(q, np.quantile(y, alpha, method="higher"))
2889: (12)                 else:
2890: (16)                     assert_allclose(q, np.quantile(y, alpha, method="lower"))
2891: (8)             elif method == "closest_observation":
2892: (12)                 if n * alpha == int(n * alpha):
2893: (16)                     assert_allclose(q, np.quantile(y, alpha, method="higher"))
2894: (12)                 elif np.round(n * alpha) == int(n * alpha) + 1:
2895: (16)                     assert_allclose(
2896: (20)                         q, np.quantile(y, alpha + 1/n, method="higher"))
2897: (12)                 else:
2898: (16)                     assert_allclose(q, np.quantile(y, alpha, method="lower"))
2899: (8)             elif method == "interpolated_inverted_cdf":
2900: (12)                 assert_allclose(q, np.quantile(y, alpha + 1/n, method=method))
2901: (8)             elif method == "nearest":
2902: (12)                 if n * alpha == int(n * alpha):
2903: (16)                     assert_allclose(q, np.quantile(y, alpha + 1/n, method=method))
2904: (12)                 else:
2905: (16)                     assert_allclose(q, np.quantile(y, alpha, method=method))
2906: (8)             elif method == "lower":
2907: (12)                 assert_allclose(q, np.quantile(y, alpha, method="higher"))
2908: (8)             elif method == "higher":
2909: (12)                 assert_allclose(q, np.quantile(y, alpha, method="lower"))
2910: (8)             else:
2911: (12)                 assert_allclose(q, np.quantile(y, alpha, method=method))
2912: (0)         class TestLerp:
2913: (4)             @hypothesis.given(t0=st.floats(allow_nan=False, allow_infinity=False,
2914: (35)                             min_value=0, max_value=1),
2915: (22)                             t1=st.floats(allow_nan=False, allow_infinity=False,
2916: (35)                             min_value=0, max_value=1),
2917: (22)                             a = st.floats(allow_nan=False, allow_infinity=False,
2918: (36)                             min_value=-1e300, max_value=1e300),
2919: (22)                             b = st.floats(allow_nan=False, allow_infinity=False,
2920: (36)                             min_value=-1e300, max_value=1e300))
2921: (4)             def test_linear_interpolation_formula_monotonic(self, t0, t1, a, b):
2922: (8)                 l0 = nfb._lerp(a, b, t0)
2923: (8)                 l1 = nfb._lerp(a, b, t1)
2924: (8)                 if t0 == t1 or a == b:
2925: (12)                     assert l0 == l1 # uninteresting
2926: (8)                 elif (t0 < t1) == (a < b):
2927: (12)                     assert l0 <= l1
2928: (8)                 else:
2929: (12)                     assert l0 >= l1
2930: (4)             @hypothesis.given(t=st.floats(allow_nan=False, allow_infinity=False,
2931: (34)                             min_value=0, max_value=1),
2932: (22)                             a=st.floats(allow_nan=False, allow_infinity=False,
2933: (34)                             min_value=-1e300, max_value=1e300),
2934: (22)                             b=st.floats(allow_nan=False, allow_infinity=False,
2935: (34)                             min_value=-1e300, max_value=1e300))
2936: (4)             def test_linear_interpolation_formula_bounded(self, t, a, b):
2937: (8)                 if a <= b:
2938: (12)                     assert a <= nfb._lerp(a, b, t) <= b
2939: (8)                 else:
2940: (12)                     assert b <= nfb._lerp(a, b, t) <= a
2941: (4)             @hypothesis.given(t=st.floats(allow_nan=False, allow_infinity=False,
2942: (34)                             min_value=0, max_value=1),

```

```

2943: (22)
2944: (34)
2945: (22)
2946: (34)
2947: (4)
2948: (8)
2949: (8)
2950: (8)
2951: (4)
2952: (8)
2953: (8)
2954: (8)
2955: (8)
2956: (0)
2957: (4)
2958: (8)
2959: (8)
2960: (8)
2961: (8)
2962: (8)
2963: (8)
2964: (8)
2965: (8)
2966: (8)
2967: (8)
2968: (8)
2969: (8)
2970: (8)
2971: (8)
2972: (8)
2973: (8)
2974: (8)
2975: (8)
2976: (4)
2977: (8)
2978: (23)
2979: (23)
2980: (23)
2981: (8)
2982: (12)
2983: (12)
2984: (12)
2985: (16)
2986: (12)
2987: (8)
2988: (8)
2989: (8)
2990: (8)
2991: (8)
2992: (4)
2993: (8)
2994: (23)
2995: (23)
2996: (23)
2997: (8)
2998: (8)
2999: (8)
3000: (8)
3001: (8)
3002: (8)
3003: (8)
3004: (24)
3005: (8)
3006: (12)
3007: (8)
3008: (12)
3009: (8)
3010: (12)
3011: (8)

                     a=st.floats(allow_nan=False, allow_infinity=False,
                                         min_value=-1e300, max_value=1e300),
                     b=st.floats(allow_nan=False, allow_infinity=False,
                                         min_value=-1e300, max_value=1e300))

def test_linear_interpolation_formula_symmetric(self, t, a, b):
    left = nfb._lerp(a, b, 1 - (1 - t))
    right = nfb._lerp(b, a, 1 - t)
    assert_allclose(left, right)

def test_linear_interpolation_formula_0d_inputs(self):
    a = np.array(2)
    b = np.array(5)
    t = np.array(0.2)
    assert nfb._lerp(a, b, t) == 2.6

class TestMedian:
    def test_basic(self):
        a0 = np.array(1)
        a1 = np.arange(2)
        a2 = np.arange(6).reshape(2, 3)
        assert_equal(np.median(a0), 1)
        assert_allclose(np.median(a1), 0.5)
        assert_allclose(np.median(a2), 2.5)
        assert_allclose(np.median(a2, axis=0), [1.5, 2.5, 3.5])
        assert_equal(np.median(a2, axis=1), [1, 4])
        assert_allclose(np.median(a2, axis=None), 2.5)
        a = np.array([0.0444502, 0.0463301, 0.141249, 0.0606775])
        assert_almost_equal((a[1] + a[3]) / 2., np.median(a))
        a = np.array([0.0463301, 0.0444502, 0.141249])
        assert_equal(a[0], np.median(a))
        a = np.array([0.0444502, 0.141249, 0.0463301])
        assert_equal(a[-1], np.median(a))
        assert_equal(np.median(a).ndim, 0)
        a[1] = np.nan
        assert_equal(np.median(a).ndim, 0)

    def test_axis_keyword(self):
        a3 = np.array([[2, 3],
                      [0, 1],
                      [6, 7],
                      [4, 5]])
        for a in [a3, np.random.randint(0, 100, size=(2, 3, 4))]:
            orig = a.copy()
            np.median(a, axis=None)
            for ax in range(a.ndim):
                np.median(a, axis=ax)
            assert_array_equal(a, orig)
        assert_allclose(np.median(a3, axis=0), [3, 4])
        assert_allclose(np.median(a3.T, axis=1), [3, 4])
        assert_allclose(np.median(a3), 3.5)
        assert_allclose(np.median(a3, axis=None), 3.5)
        assert_allclose(np.median(a3.T), 3.5)

    def test_overwrite_keyword(self):
        a3 = np.array([[2, 3],
                      [0, 1],
                      [6, 7],
                      [4, 5]])
        a0 = np.array(1)
        a1 = np.arange(2)
        a2 = np.arange(6).reshape(2, 3)
        assert_allclose(np.median(a0.copy(), overwrite_input=True), 1)
        assert_allclose(np.median(a1.copy(), overwrite_input=True), 0.5)
        assert_allclose(np.median(a2.copy(), overwrite_input=True), 2.5)
        assert_allclose(np.median(a2.copy(), overwrite_input=True, axis=0),
                       [1.5, 2.5, 3.5])
        assert_allclose(
            np.median(a2.copy(), overwrite_input=True, axis=1), [1, 4])
        assert_allclose(
            np.median(a2.copy(), overwrite_input=True, axis=None), 2.5)
        assert_allclose(
            np.median(a3.copy(), overwrite_input=True, axis=0), [3, 4])
        assert_allclose(np.median(a3.T.copy(), overwrite_input=True, axis=1),

```

```

3012: (24)           [3, 4])
3013: (8)            a4 = np.arange(3 * 4 * 5, dtype=np.float32).reshape((3, 4, 5))
3014: (8)            np.random.shuffle(a4.ravel())
3015: (8)            assert_allclose(np.median(a4, axis=None),
3016: (24)                  np.median(a4.copy(), axis=None, overwrite_input=True))
3017: (8)            assert_allclose(np.median(a4, axis=0),
3018: (24)                  np.median(a4.copy(), axis=0, overwrite_input=True))
3019: (8)            assert_allclose(np.median(a4, axis=1),
3020: (24)                  np.median(a4.copy(), axis=1, overwrite_input=True))
3021: (8)            assert_allclose(np.median(a4, axis=2),
3022: (24)                  np.median(a4.copy(), axis=2, overwrite_input=True))
3023: (4)
3024: (8)            def test_array_like(self):
3025: (8)                x = [1, 2, 3]
3026: (8)                assert_almost_equal(np.median(x), 2)
3027: (8)                x2 = [x]
3028: (8)                assert_almost_equal(np.median(x2), 2)
3029: (4)
3030: (8)            def test_subclass(self):
3031: (12)              class MySubClass(np.ndarray):
3032: (16)                  def __new__(cls, input_array, info=None):
3033: (16)                      obj = np.asarray(input_array).view(cls)
3034: (16)                      obj.info = info
3035: (12)                      return obj
3036: (16)                  def mean(self, axis=None, dtype=None, out=None):
3037: (8)                      return -7
3038: (8)                  a = MySubClass([1, 2, 3])
3039: (4)                  assert_equal(np.median(a), -7)
3040: (29)                  @pytest.mark.parametrize('arr',
3041: (4)                      ([1., 2., 3.], [1., np.nan, 3.], np.nan, 0.))
3042: (8)                  def test_subclass2(self, arr):
3043: (8)                      """Check that we return subclasses, even if a NaN scalar."""
3044: (12)                      class MySubclass(np.ndarray):
3045: (8)                          pass
3046: (8)                      m = np.median(np.array(arr).view(MySubclass))
3047: (4)                      assert isinstance(m, MySubclass)
3048: (8)                  def test_out(self):
3049: (8)                      o = np.zeros((4,))
3050: (8)                      d = np.ones((3, 4))
3051: (8)                      assert_equal(np.median(d, 0, out=o), o)
3052: (8)                      o = np.zeros((3,))
3053: (8)                      assert_equal(np.median(d, 1, out=o), o)
3054: (8)                      o = np.zeros(())
3055: (4)                      assert_equal(np.median(d, out=o), o)
3056: (8)                  def test_out_nan(self):
3057: (12)                      with warnings.catch_warnings(record=True):
3058: (12)                          warnings.filterwarnings('always', '', RuntimeWarning)
3059: (12)                          o = np.zeros((4,))
3060: (12)                          d = np.ones((3, 4))
3061: (12)                          d[2, 1] = np.nan
3062: (12)                          assert_equal(np.median(d, 0, out=o), o)
3063: (12)                          o = np.zeros((3,))
3064: (12)                          assert_equal(np.median(d, 1, out=o), o)
3065: (12)                          o = np.zeros(())
3066: (4)                          assert_equal(np.median(d, out=o), o)
3067: (8)                  def test_nan_behavior(self):
3068: (8)                      a = np.arange(24, dtype=float)
3069: (8)                      a[2] = np.nan
3070: (8)                      assert_equal(np.median(a), np.nan)
3071: (8)                      assert_equal(np.median(a, axis=0), np.nan)
3072: (8)                      a = np.arange(24, dtype=float).reshape(2, 3, 4)
3073: (8)                      a[1, 2, 3] = np.nan
3074: (8)                      a[1, 1, 2] = np.nan
3075: (8)                      assert_equal(np.median(a), np.nan)
3076: (8)                      assert_equal(np.median(a).ndim, 0)
3077: (8)                      b = np.median(np.arange(24, dtype=float).reshape(2, 3, 4), 0)
3078: (8)                      b[2, 3] = np.nan
3079: (8)                      b[1, 2] = np.nan
3080: (8)                      assert_equal(np.median(a, 0), b)

```

```

3081: (8)             b[1, 3] = np.nan
3082: (8)             b[1, 2] = np.nan
3083: (8)             assert_equal(np.median(a, 1), b)
3084: (8)             b = np.median(np.arange(24, dtype=float).reshape(2, 3, 4), (0, 2))
3085: (8)             b[1] = np.nan
3086: (8)             b[2] = np.nan
3087: (8)             assert_equal(np.median(a, (0, 2)), b)
3088: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work correctly")
3089: (4) def test_empty(self):
3090: (8)     a = np.array([], dtype=float)
3091: (8)     with warnings.catch_warnings(record=True) as w:
3092: (12)         warnings.filterwarnings('always', '', RuntimeWarning)
3093: (12)         assert_equal(np.median(a), np.nan)
3094: (12)         assert_(w[0].category is RuntimeWarning)
3095: (12)         assert_equal(len(w), 2)
3096: (8)     a = np.array([], dtype=float, ndmin=3)
3097: (8)     with warnings.catch_warnings(record=True) as w:
3098: (12)         warnings.filterwarnings('always', '', RuntimeWarning)
3099: (12)         assert_equal(np.median(a), np.nan)
3100: (12)         assert_(w[0].category is RuntimeWarning)
3101: (8)     b = np.array([], dtype=float, ndmin=2)
3102: (8)     assert_equal(np.median(a, axis=0), b)
3103: (8)     assert_equal(np.median(a, axis=1), b)
3104: (8)     b = np.array(np.nan, dtype=float, ndmin=2)
3105: (8)     with warnings.catch_warnings(record=True) as w:
3106: (12)         warnings.filterwarnings('always', '', RuntimeWarning)
3107: (12)         assert_equal(np.median(a, axis=2), b)
3108: (12)         assert_(w[0].category is RuntimeWarning)
3109: (4) def test_object(self):
3110: (8)     o = np.arange(7.)
3111: (8)     assert_(type(np.median(o.astype(object))), float)
3112: (8)     o[2] = np.nan
3113: (8)     assert_(type(np.median(o.astype(object))), float)
3114: (4) def test_extended_axis(self):
3115: (8)     o = np.random.normal(size=(71, 23))
3116: (8)     x = np.dstack([o] * 10)
3117: (8)     assert_equal(np.median(x, axis=(0, 1)), np.median(o))
3118: (8)     x = np.moveaxis(x, -1, 0)
3119: (8)     assert_equal(np.median(x, axis=(-2, -1)), np.median(o))
3120: (8)     x = x.swapaxes(0, 1).copy()
3121: (8)     assert_equal(np.median(x, axis=(0, -1)), np.median(o))
3122: (8)     assert_equal(np.median(x, axis=(0, 1, 2)), np.median(x, axis=None))
3123: (8)     assert_equal(np.median(x, axis=(0, )), np.median(x, axis=0))
3124: (8)     assert_equal(np.median(x, axis=(-1, )), np.median(x, axis=-1))
3125: (8)     d = np.arange(3 * 5 * 7 * 11).reshape((3, 5, 7, 11))
3126: (8)     np.random.shuffle(d.ravel())
3127: (8)     assert_equal(np.median(d, axis=(0, 1, 2))[0],
3128: (21)                   np.median(d[:, :, :, 0].flatten()))
3129: (8)     assert_equal(np.median(d, axis=(0, 1, 3))[1],
3130: (21)                   np.median(d[:, :, 1, :].flatten()))
3131: (8)     assert_equal(np.median(d, axis=(3, 1, -4))[2],
3132: (21)                   np.median(d[:, :, 2, :].flatten()))
3133: (8)     assert_equal(np.median(d, axis=(3, 1, 2))[2],
3134: (21)                   np.median(d[2, :, :, :].flatten()))
3135: (8)     assert_equal(np.median(d, axis=(3, 2))[2, 1],
3136: (21)                   np.median(d[2, 1, :, :].flatten()))
3137: (8)     assert_equal(np.median(d, axis=(1, -2))[2, 1],
3138: (21)                   np.median(d[2, :, :, 1].flatten()))
3139: (8)     assert_equal(np.median(d, axis=(1, 3))[2, 2],
3140: (21)                   np.median(d[2, :, 2, :].flatten()))
3141: (4) def test_extended_axis_invalid(self):
3142: (8)     d = np.ones((3, 5, 7, 11))
3143: (8)     assert_raises(np.AxisError, np.median, d, axis=-5)
3144: (8)     assert_raises(np.AxisError, np.median, d, axis=(0, -5))
3145: (8)     assert_raises(np.AxisError, np.median, d, axis=4)
3146: (8)     assert_raises(np.AxisError, np.median, d, axis=(0, 4))
3147: (8)     assert_raises(ValueError, np.median, d, axis=(1, 1))
3148: (4) def test_keepdims(self):
3149: (8)     d = np.ones((3, 5, 7, 11))

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

3150: (8)
3151: (21)
3152: (8)
3153: (21)
3154: (8)
3155: (21)
3156: (8)
3157: (21)
3158: (8)
3159: (21)
3160: (8)
3161: (21)
3162: (4)
3163: (8)
3164: (8)
3165: (12)
3166: (12)
3167: (12)
3168: (12)
3169: (12)
3170: (8)
3171: (4)
3172: (4)
3173: (8)
3174: (8)
3175: (12)
3176: (8)
3177: (12)
3178: (12)
3179: (16)
3180: (8)
3181: (8)
3182: (8)
3183: (8)
3184: (4)
3185: (4)
3186: (4)
3187: (8)
3188: (8)
3189: (8)
3190: (8)
3191: (8)
3192: (8)
3193: (8)
3194: (8)
3195: (8)
3196: (8)
3197: (8)
3198: (8)
3199: (0)
class TestAdd_newdoc_ufunc:
    def test_ufunc_arg(self):
        assert_raises(TypeError, add_newdoc_ufunc, 2, "blah")
        assert_raises(ValueError, add_newdoc_ufunc, np.add, "blah")
    def test_string_arg(self):
        assert_raises(TypeError, add_newdoc_ufunc, np.add, 3)
class TestAdd_newdoc:
    @pytest.mark.skipif(sys.flags.optimize == 2, reason="Python running -O0")
    @pytest.mark.xfail(IS_PYPY, reason="PyPy does not modify tp_doc")
    def test_add_doc(self):
        tgt = "Current flat index into the array."
        assert_equal(np.core.flatiter.index.__doc__[:len(tgt)], tgt)
        assert_(len(np.core.ufunc.identity.__doc__) > 300)
        assert_(len(np.lib.index_tricks.mgrid.__doc__) > 300)
    @pytest.mark.skipif(sys.flags.optimize == 2, reason="Python running -O0")
    def test_errors_are_ignored(self):
        prev_doc = np.core.flatiter.index.__doc__
        np.add_newdoc("numpy.core", "flatiter", ("index", "bad docstring"))
        assert prev_doc == np.core.flatiter.index.__doc__
class TestAddDocstring():

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3219: (4)         @pytest.mark.skipif(sys.flags.optimize == 2, reason="Python running -OO")
3220: (4)         @pytest.mark.skipif(IS_PYPY, reason="PyPy does not modify tp_doc")
3221: (4)         def test_add_same_docstring(self):
3222: (8)             np.add_docstring(np.ndarray.flat, np.ndarray.flat.__doc__)
3223: (8)             def func():
3224: (12)                 """docstring"""
3225: (12)                 return
3226: (8)             np.add_docstring(func, func.__doc__)
3227: (4)         @pytest.mark.skipif(sys.flags.optimize == 2, reason="Python running -OO")
3228: (4)         def test_different_docstring_fails(self):
3229: (8)             with assert_raises(RuntimeError):
3230: (12)                 np.add_docstring(np.ndarray.flat, "different docstring")
3231: (8)             def func():
3232: (12)                 """docstring"""
3233: (12)                 return
3234: (8)             with assert_raises(RuntimeError):
3235: (12)                 np.add_docstring(func, "different docstring")
3236: (0)         class TestSortComplex:
3237: (4)             @pytest.mark.parametrize("type_in, type_out", [
3238: (8)                 ('l', 'D'),
3239: (8)                 ('h', 'F'),
3240: (8)                 ('H', 'F'),
3241: (8)                 ('b', 'F'),
3242: (8)                 ('B', 'F'),
3243: (8)                 ('g', 'G'),
3244: (8)             ])
3245: (4)             def test_sort_real(self, type_in, type_out):
3246: (8)                 a = np.array([5, 3, 6, 2, 1], dtype=type_in)
3247: (8)                 actual = np.sort_complex(a)
3248: (8)                 expected = np.sort(a).astype(type_out)
3249: (8)                 assert_equal(actual, expected)
3250: (8)                 assert_equal(actual.dtype, expected.dtype)
3251: (4)             def test_sort_complex(self):
3252: (8)                 a = np.array([2 + 3j, 1 - 2j, 1 - 3j, 2 + 1j], dtype='D')
3253: (8)                 expected = np.array([1 - 3j, 1 - 2j, 2 + 1j, 2 + 3j], dtype='D')
3254: (8)                 actual = np.sort_complex(a)
3255: (8)                 assert_equal(actual, expected)
3256: (8)                 assert_equal(actual.dtype, expected.dtype)

```

---

File 235 - test\_histograms.py:

```

1: (0)         import numpy as np
2: (0)         from numpy.lib.histograms import histogram, histogramdd, histogram_bin_edges
3: (0)         from numpy.testing import (
4: (4)             assert_, assert_equal, assert_array_equal, assert_almost_equal,
5: (4)             assert_array_almost_equal, assert_raises, assert_allclose,
6: (4)             assert_array_max_ulp, assert_raises_regex, suppress_warnings,
7: (4)         )
8: (0)         from numpy.testing._private.utils import requires_memory
9: (0)         import pytest
10: (0)         class TestHistogram:
11: (4)             def setup_method(self):
12: (8)                 pass
13: (4)             def teardown_method(self):
14: (8)                 pass
15: (4)             def test_simple(self):
16: (8)                 n = 100
17: (8)                 v = np.random.rand(n)
18: (8)                 (a, b) = histogram(v)
19: (8)                 assert_equal(np.sum(a, axis=0), n)
20: (8)                 (a, b) = histogram(np.linspace(0, 10, 100))
21: (8)                 assert_array_equal(a, 10)
22: (4)             def test_one_bin(self):
23: (8)                 hist, edges = histogram([1, 2, 3, 4], [1, 2])
24: (8)                 assert_array_equal(hist, [2, ])
25: (8)                 assert_array_equal(edges, [1, 2])
26: (8)                 assert_raises(ValueError, histogram, [1, 2], bins=0)

```

```

27: (8)             h, e = histogram([1, 2], bins=1)
28: (8)             assert_equal(h, np.array([2]))
29: (8)             assert_allclose(e, np.array([1., 2.]))
30: (4)             def test_density(self):
31: (8)                 n = 100
32: (8)                 v = np.random.rand(n)
33: (8)                 a, b = histogram(v, density=True)
34: (8)                 area = np.sum(a * np.diff(b))
35: (8)                 assert_almost_equal(area, 1)
36: (8)                 v = np.arange(10)
37: (8)                 bins = [0, 1, 3, 6, 10]
38: (8)                 a, b = histogram(v, bins, density=True)
39: (8)                 assert_array_equal(a, .1)
40: (8)                 assert_equal(np.sum(a * np.diff(b)), 1)
41: (8)                 a, b = histogram(v, bins, density=False)
42: (8)                 assert_array_equal(a, [1, 2, 3, 4])
43: (8)                 v = np.arange(10)
44: (8)                 bins = [0, 1, 3, 6, np.inf]
45: (8)                 a, b = histogram(v, bins, density=True)
46: (8)                 assert_array_equal(a, [.1, .1, .1, 0.])
47: (8)                 counts, dmy = np.histogram(
48: (12)                     [1, 2, 3, 4], [0.5, 1.5, np.inf], density=True)
49: (8)                 assert_equal(counts, [.25, 0])
50: (4)             def test_outliers(self):
51: (8)                 a = np.arange(10) + .5
52: (8)                 h, b = histogram(a, range=[0, 9])
53: (8)                 assert_equal(h.sum(), 9)
54: (8)                 h, b = histogram(a, range=[1, 10])
55: (8)                 assert_equal(h.sum(), 9)
56: (8)                 h, b = histogram(a, range=[1, 9], density=True)
57: (8)                 assert_almost_equal((h * np.diff(b)).sum(), 1, decimal=15)
58: (8)                 w = np.arange(10) + .5
59: (8)                 h, b = histogram(a, range=[1, 9], weights=w, density=True)
60: (8)                 assert_equal((h * np.diff(b)).sum(), 1)
61: (8)                 h, b = histogram(a, bins=8, range=[1, 9], weights=w)
62: (8)                 assert_equal(h, w[1:-1])
63: (4)             def test_arr_weights_mismatch(self):
64: (8)                 a = np.arange(10) + .5
65: (8)                 w = np.arange(11) + .5
66: (8)                 with assert_raises_regex(ValueError, "same shape as"):
67: (12)                     h, b = histogram(a, range=[1, 9], weights=w, density=True)
68: (4)             def test_type(self):
69: (8)                 a = np.arange(10) + .5
70: (8)                 h, b = histogram(a)
71: (8)                 assert_(np.issubdtype(h.dtype, np.integer))
72: (8)                 h, b = histogram(a, density=True)
73: (8)                 assert_(np.issubdtype(h.dtype, np.floating))
74: (8)                 h, b = histogram(a, weights=np.ones(10, int))
75: (8)                 assert_(np.issubdtype(h.dtype, np.integer))
76: (8)                 h, b = histogram(a, weights=np.ones(10, float))
77: (8)                 assert_(np.issubdtype(h.dtype, np.floating))
78: (4)             def test_f32_rounding(self):
79: (8)                 x = np.array([276.318359, -69.593948, 21.329449], dtype=np.float32)
80: (8)                 y = np.array([5005.689453, 4481.327637, 6010.369629],
81: (8)                               dtype=np.float32)
82: (8)                 counts_hist, xedges, yedges = np.histogram2d(x, y, bins=100)
83: (4)                 assert_equal(counts_hist.sum(), 3.)
84: (8)             def test_bool_conversion(self):
85: (8)                 a = np.array([1, 1, 0], dtype=np.uint8)
86: (8)                 int_hist, int_edges = np.histogram(a)
87: (12)                 with suppress_warnings() as sup:
88: (12)                     rec = sup.record(RuntimeWarning, 'Converting input from .*')
89: (12)                     hist, edges = np.histogram([True, True, False])
90: (12)                     assert_equal(len(rec), 1)
91: (12)                     assert_array_equal(hist, int_hist)
92: (4)                     assert_array_equal(edges, int_edges)
93: (8)             def test_weights(self):
94: (8)                 v = np.random.rand(100)

```

```

95: (8)             a, b = histogram(v)
96: (8)             na, nb = histogram(v, density=True)
97: (8)             wa, wb = histogram(v, weights=w)
98: (8)             nwa, nwbc = histogram(v, weights=w, density=True)
99: (8)             assert_array_almost_equal(a * 5, wa)
100: (8)            assert_array_almost_equal(na, nwa)
101: (8)            v = np.linspace(0, 10, 10)
102: (8)            w = np.concatenate((np.zeros(5), np.ones(5)))
103: (8)            wa, wb = histogram(v, bins=np.arange(11), weights=w)
104: (8)            assert_array_almost_equal(wa, w)
105: (8)            wa, wb = histogram([1, 2, 2, 4], bins=4, weights=[4, 3, 2, 1])
106: (8)            assert_array_equal(wa, [4, 5, 0, 1])
107: (8)            wa, wb = histogram(
108: (12)                [1, 2, 2, 4], bins=4, weights=[4, 3, 2, 1], density=True)
109: (8)            assert_array_almost_equal(wa, np.array([4, 5, 0, 1]) / 10. / 3. * 4)
110: (8)            a, b = histogram(
111: (12)                np.arange(9), [0, 1, 3, 6, 10],
112: (12)                weights=[2, 1, 1, 1, 1, 1, 1, 1], density=True)
113: (8)            assert_almost_equal(a, [.2, .1, .1, .075])
114: (4)             def test_exotic_weights(self):
115: (8)                 values = np.array([1.3, 2.5, 2.3])
116: (8)                 weights = np.array([1, -1, 2]) + 1j * np.array([2, 1, 2])
117: (8)                 wa, wb = histogram(values, bins=[0, 2, 3], weights=weights)
118: (8)                 assert_array_almost_equal(wa, np.array([1, 1]) + 1j * np.array([2,
3]))
119: (8)                 wa, wb = histogram(values, bins=2, range=[1, 3], weights=weights)
120: (8)                 assert_array_almost_equal(wa, np.array([1, 1]) + 1j * np.array([2,
3]))
121: (8)                 from decimal import Decimal
122: (8)                 values = np.array([1.3, 2.5, 2.3])
123: (8)                 weights = np.array([Decimal(1), Decimal(2), Decimal(3)])
124: (8)                 wa, wb = histogram(values, bins=[0, 2, 3], weights=weights)
125: (8)                 assert_array_almost_equal(wa, [Decimal(1), Decimal(5)])
126: (8)                 wa, wb = histogram(values, bins=2, range=[1, 3], weights=weights)
127: (8)                 assert_array_almost_equal(wa, [Decimal(1), Decimal(5)])
128: (4)              def test_no_side_effects(self):
129: (8)                  values = np.array([1.3, 2.5, 2.3])
130: (8)                  np.histogram(values, range=[-10, 10], bins=100)
131: (8)                  assert_array_almost_equal(values, [1.3, 2.5, 2.3])
132: (4)              def test_empty(self):
133: (8)                  a, b = histogram([], bins=[0, 1])
134: (8)                  assert_array_equal(a, np.array([0]))
135: (8)                  assert_array_equal(b, np.array([0, 1]))
136: (4)              def test_error_binnum_type (self):
137: (8)                  vals = np.linspace(0.0, 1.0, num=100)
138: (8)                  histogram(vals, 5)
139: (8)                  assert_raises(TypeError, histogram, vals, 2.4)
140: (4)              def test_finite_range(self):
141: (8)                  vals = np.linspace(0.0, 1.0, num=100)
142: (8)                  histogram(vals, range=[0.25,0.75])
143: (8)                  assert_raises(ValueError, histogram, vals, range=[np.nan,0.75])
144: (8)                  assert_raises(ValueError, histogram, vals, range=[0.25,np.inf])
145: (4)              def test_invalid_range(self):
146: (8)                  vals = np.linspace(0.0, 1.0, num=100)
147: (8)                  with assert_raises_regex(ValueError, "max must be larger than"):
148: (12)                      np.histogram(vals, range=[0.1, 0.01])
149: (4)              def test_bin_edge_cases(self):
150: (8)                  arr = np.array([337, 404, 739, 806, 1007, 1811, 2012])
151: (8)                  hist, edges = np.histogram(arr, bins=8296, range=(2, 2280))
152: (8)                  mask = hist > 0
153: (8)                  left_edges = edges[:-1][mask]
154: (8)                  right_edges = edges[1:][mask]
155: (8)                  for x, left, right in zip(arr, left_edges, right_edges):
156: (12)                      assert_(x >= left)
157: (12)                      assert_(x < right)
158: (4)              def test_last_bin_inclusive_range(self):
159: (8)                  arr = np.array([0., 0., 0., 1., 2., 3., 3., 4., 5.])
160: (8)                  hist, edges = np.histogram(arr, bins=30, range=(-0.5, 5))
161: (8)                  assert_equal(hist[-1], 1)

```

```

162: (4) def test_bin_array_dims(self):
163: (8)     vals = np.linspace(0.0, 1.0, num=100)
164: (8)     bins = np.array([[0, 0.5], [0.6, 1.0]])
165: (8)     with assert_raises_regex(ValueError, "must be 1d"):
166: (12)         np.histogram(vals, bins=bins)
167: (4) def test_unsigned_monotonicity_check(self):
168: (8)     arr = np.array([2])
169: (8)     bins = np.array([1, 3, 1], dtype='uint64')
170: (8)     with assert_raises(ValueError):
171: (12)         hist, edges = np.histogram(arr, bins=bins)
172: (4) def test_object_array_of_0d(self):
173: (8)     assert_raises(ValueError,
174: (12)         histogram, [np.array(0.4) for i in range(10)] + [-np.inf])
175: (8)     assert_raises(ValueError,
176: (12)         histogram, [np.array(0.4) for i in range(10)] + [np.inf])
177: (8)     np.histogram([np.array(0.5) for i in range(10)] + [.5000000000000001])
178: (8)     np.histogram([np.array(0.5) for i in range(10)] + [.5])
179: (4) def test_some_nan_values(self):
180: (8)     one_nan = np.array([0, 1, np.nan])
181: (8)     all_nan = np.array([np.nan, np.nan])
182: (8)     sup = suppress_warnings()
183: (8)     sup.filter(RuntimeWarning)
184: (8)     with sup:
185: (12)         assert_raises(ValueError, histogram, one_nan, bins='auto')
186: (12)         assert_raises(ValueError, histogram, all_nan, bins='auto')
187: (12)         h, b = histogram(one_nan, bins='auto', range=(0, 1))
188: (12)         assert_equal(h.sum(), 2) # nan is not counted
189: (12)         h, b = histogram(all_nan, bins='auto', range=(0, 1))
190: (12)         assert_equal(h.sum(), 0) # nan is not counted
191: (12)         h, b = histogram(one_nan, bins=[0, 1])
192: (12)         assert_equal(h.sum(), 2) # nan is not counted
193: (12)         h, b = histogram(all_nan, bins=[0, 1])
194: (12)         assert_equal(h.sum(), 0) # nan is not counted
195: (4) def test_datetime(self):
196: (8)     begin = np.datetime64('2000-01-01', 'D')
197: (8)     offsets = np.array([0, 0, 1, 1, 2, 3, 5, 10, 20])
198: (8)     bins = np.array([0, 2, 7, 20])
199: (8)     dates = begin + offsets
200: (8)     date_bins = begin + bins
201: (8)     td = np.dtype('timedelta64[D]')
202: (8)     d_count, d_edge = histogram(dates, bins=date_bins)
203: (8)     t_count, t_edge = histogram(offsets.astype(td), bins=bins.astype(td))
204: (8)     i_count, i_edge = histogram(offsets, bins=bins)
205: (8)     assert_equal(d_count, i_count)
206: (8)     assert_equal(t_count, i_count)
207: (8)     assert_equal((d_edge - begin).astype(int), i_edge)
208: (8)     assert_equal(t_edge.astype(int), i_edge)
209: (8)     assert_equal(d_edge.dtype, dates.dtype)
210: (8)     assert_equal(t_edge.dtype, td)
211: (4) def do_signed_overflow_bounds(self, dtype):
212: (8)     exponent = 8 * np.dtype(dtype).itemsize - 1
213: (8)     arr = np.array([-2**exponent + 4, 2**exponent - 4], dtype=dtype)
214: (8)     hist, e = histogram(arr, bins=2)
215: (8)     assert_equal(e, [-2**exponent + 4, 0, 2**exponent - 4])
216: (8)     assert_equal(hist, [1, 1])
217: (4) def test_signed_overflow_bounds(self):
218: (8)     self.do_signed_overflow_bounds(np.byte)
219: (8)     self.do_signed_overflow_bounds(np.short)
220: (8)     self.do_signed_overflow_bounds(np.intc)
221: (8)     self.do_signed_overflow_bounds(np.int_)
222: (8)     self.do_signed_overflow_bounds(np.longlong)
223: (4) def do_precision_lower_bound(self, float_small, float_large):
224: (8)     eps = np.finfo(float_large).eps
225: (8)     arr = np.array([1.0], float_small)
226: (8)     range = np.array([1.0 + eps, 2.0], float_large)
227: (8)     if range.astype(float_small)[0] != 1:
228: (12)         return
229: (8)     count, x_loc = np.histogram(arr, bins=1, range=range)
230: (8)     assert_equal(count, [1])

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

231: (8)             assert_equal(x_loc.dtype, float_small)
232: (4)             def do_precision_upper_bound(self, float_small, float_large):
233: (8)                 eps = np.finfo(float_large).eps
234: (8)                 arr = np.array([1.0], float_small)
235: (8)                 range = np.array([0.0, 1.0 - eps], float_large)
236: (8)                 if range.astype(float_small)[-1] != 1:
237: (12)                     return
238: (8)                 count, x_loc = np.histogram(arr, bins=1, range=range)
239: (8)                 assert_equal(count, [1])
240: (8)                 assert_equal(x_loc.dtype, float_small)
241: (4)             def do_precision(self, float_small, float_large):
242: (8)                 self.do_precision_lower_bound(float_small, float_large)
243: (8)                 self.do_precision_upper_bound(float_small, float_large)
244: (4)             def test_precision(self):
245: (8)                 self.do_precision(np.half, np.single)
246: (8)                 self.do_precision(np.half, np.double)
247: (8)                 self.do_precision(np.half, np.longdouble)
248: (8)                 self.do_precision(np.single, np.double)
249: (8)                 self.do_precision(np.single, np.longdouble)
250: (8)                 self.do_precision(np.double, np.longdouble)
251: (4)             def test_histogram_bin_edges(self):
252: (8)                 hist, e = histogram([1, 2, 3, 4], [1, 2])
253: (8)                 edges = histogram_bin_edges([1, 2, 3, 4], [1, 2])
254: (8)                 assert_array_equal(edges, e)
255: (8)                 arr = np.array([0., 0., 0., 1., 2., 3., 3., 4., 5.])
256: (8)                 hist, e = histogram(arr, bins=30, range=(-0.5, 5))
257: (8)                 edges = histogram_bin_edges(arr, bins=30, range=(-0.5, 5))
258: (8)                 assert_array_equal(edges, e)
259: (8)                 hist, e = histogram(arr, bins='auto', range=(0, 1))
260: (8)                 edges = histogram_bin_edges(arr, bins='auto', range=(0, 1))
261: (8)                 assert_array_equal(edges, e)
262: (4)             @pytest.mark.skip(reason="Bad memory reports lead to OOM in ci testing")
263: (4)             def test_big_arrays(self):
264: (8)                 sample = np.zeros([100000000, 3])
265: (8)                 xbins = 400
266: (8)                 ybins = 400
267: (8)                 zbins = np.arange(16000)
268: (8)                 hist = np.histogramdd(sample=sample, bins=(xbins, ybins, zbins))
269: (8)                 assert_equal(type(hist), type((1, 2)))
270: (4)             def test_gh_23110(self):
271: (8)                 hist, e = np.histogram(np.array([-0.9e-308]), dtype='>f8'),
272: (31)                     bins=2,
273: (31)                     range=(-1e-308, -2e-313))
274: (8)                 expected_hist = np.array([1, 0])
275: (8)                 assert_array_equal(hist, expected_hist)
276: (0)             class TestHistogramOptimBinNums:
277: (4)                 """
278: (4)                     Provide test coverage when using provided estimators for optimal number of
279: (4)                     bins
280: (4)                     """
281: (4)             def test_empty(self):
282: (8)                 estimator_list = ['fd', 'scott', 'rice', 'sturges',
283: (26)                     'doane', 'sqrt', 'auto', 'stone']
284: (8)                 for estimator in estimator_list:
285: (12)                     a, b = histogram([], bins=estimator)
286: (12)                     assert_array_equal(a, np.array([0]))
287: (12)                     assert_array_equal(b, np.array([0, 1]))
288: (4)             def test_simple(self):
289: (8)                 """
290: (8)                     Straightforward testing with a mixture of linspace data (for
291: (8)                     consistency). All test values have been precomputed and the values
292: (8)                     shouldn't change
293: (8)                     """
294: (8)             basic_test = {50: {'fd': 4, 'scott': 4, 'rice': 8, 'sturges': 7,
295: (29)                     'doane': 8, 'sqrt': 8, 'auto': 7, 'stone': 2},
296: (22)                     500: {'fd': 8, 'scott': 8, 'rice': 16, 'sturges': 10,
297: (29)                     'doane': 12, 'sqrt': 23, 'auto': 10, 'stone': 9},
298: (22)                     5000: {'fd': 17, 'scott': 17, 'rice': 35, 'sturges': 14,
299: (29)                     'doane': 17, 'sqrt': 71, 'auto': 17, 'stone': 9}

```

```

20}}}
300: (8)           for testlen, expectedResults in basic_test.items():
301: (12)          x1 = np.linspace(-10, -1, testlen // 5 * 2)
302: (12)          x2 = np.linspace(1, 10, testlen // 5 * 3)
303: (12)          x = np.concatenate((x1, x2))
304: (12)          for estimator, numbins in expectedResults.items():
305: (16)             a, b = np.histogram(x, estimator)
306: (16)             assert_equal(len(a), numbins, err_msg="For the {0} estimator "
307: (29)                   "with datasize of {1}".format(estimator,
testlen))
308: (4)           def test_small(self):
309: (8)             """
310: (8)             Smaller datasets have the potential to cause issues with the data
311: (8)             adaptive methods, especially the FD method. All bin numbers have been
312: (8)             precalculated.
313: (8)
314: (8)             small_dat = {1: {'fd': 1, 'scott': 1, 'rice': 1, 'sturges': 1,
315: (25)                 'doane': 1, 'sqrt': 1, 'stone': 1},
316: (21)                 2: {'fd': 2, 'scott': 1, 'rice': 3, 'sturges': 2,
317: (25)                 'doane': 1, 'sqrt': 2, 'stone': 1},
318: (21)                 3: {'fd': 2, 'scott': 2, 'rice': 3, 'sturges': 3,
319: (25)                 'doane': 3, 'sqrt': 2, 'stone': 1}}
320: (8)             for testlen, expectedResults in small_dat.items():
321: (12)               testdat = np.arange(testlen)
322: (12)               for estimator, expbins in expectedResults.items():
323: (16)                 a, b = np.histogram(testdat, estimator)
324: (16)                 assert_equal(len(a), expbins, err_msg="For the {0} estimator "
325: (29)                   "with datasize of {1}".format(estimator,
testlen))
326: (4)           def test_incorrect_methods(self):
327: (8)             """
328: (8)             Check a Value Error is thrown when an unknown string is passed in
329: (8)
330: (8)             check_list = ['mad', 'freeman', 'histograms', 'IQR']
331: (8)             for estimator in check_list:
332: (12)               assert_raises(ValueError, histogram, [1, 2, 3], estimator)
333: (4)           def test_no variance(self):
334: (8)
335: (8)
336: (8)             """
337: (8)             Check that methods handle no variance in data
338: (8)             Primarily for Scott and FD as the SD and IQR are both 0 in this case
339: (8)
340: (28)             novar_dataset = np.ones(100)
341: (8)             novar_resultdict = {'fd': 1, 'scott': 1, 'rice': 1, 'sturges': 1,
342: (12)                 'doane': 1, 'sqrt': 1, 'auto': 1, 'stone': 1}
343: (12)             for estimator, numbins in novar_resultdict.items():
344: (25)               a, b = np.histogram(novar_dataset, estimator)
345: (4)               assert_equal(len(a), numbins, err_msg="{0} estimator, "
346: (8)                 "No Variance test".format(estimator))
347: (8)           def test_limited_variance(self):
348: (8)
349: (8)
350: (8)             """
351: (8)             lim_var_data = np.ones(1000)
352: (8)             lim_var_data[:3] = 0
353: (8)             lim_var_data[-4:] = 100
354: (8)             edges_auto = histogram_bin_edges(lim_var_data, 'auto')
355: (8)             assert_equal(edges_auto, np.linspace(0, 100, 12))
356: (8)             edges_fd = histogram_bin_edges(lim_var_data, 'fd')
357: (8)             assert_equal(edges_fd, np.array([0, 100]))
358: (8)             edges_sturges = histogram_bin_edges(lim_var_data, 'sturges')
359: (4)             assert_equal(edges_sturges, np.linspace(0, 100, 12))
360: (8)           def test_outlier(self):
361: (8)
362: (8)             """
363: (8)             Check the FD, Scott and Doane with outliers.
364: (8)             The FD estimates a smaller binwidth since it's less affected by
365: (8)             outliers. Since the range is so (artificially) large, this means more
bins, most of which will be empty, but the data of interest usually is
unaffected. The Scott estimator is more affected and returns fewer

```

```

bins,
366: (8)             despite most of the variance being in one area of the data. The Doane
367: (8)             estimator lies somewhere between the other two.
368: (8)
369: (8)
370: (8)             xccenter = np.linspace(-10, 10, 50)
371: (8)             outlier_dataset = np.hstack((np.linspace(-110, -100, 5), xccenter))
372: (8)             outlier_resultdict = {'fd': 21, 'scott': 5, 'doane': 11, 'stone': 6}
373: (12)            for estimator, numbins in outlier_resultdict.items():
374: (12)                a, b = np.histogram(outlier_dataset, estimator)
375: (4)                  assert_equal(len(a), numbins)
376: (8)            def test_scott_vs_stone(self):
377: (8)                """Verify that Scott's rule and Stone's rule converges for normally
378: (8)                distributed data"""
379: (12)            def nbins_ratio(seed, size):
380: (12)                rng = np.random.RandomState(seed)
381: (12)                x = rng.normal(loc=0, scale=2, size=size)
382: (8)                a, b = len(np.histogram(x, 'stone')[0]), len(np.histogram(x,
383: (14)                  'scott')[0])
384: (8)                ll = [[nbins_ratio(seed, size) for size in np.geomspace(start=10,
385: (8)                  stop=100, num=4).round().astype(int)]
386: (14)                  for seed in range(10)]]
387: (8)                avg = abs(np.mean(ll, axis=0)) - 0.5
388: (8)                assert_almost_equal(avg, [0.15, 0.09, 0.08, 0.03], decimal=2)
389: (8)            def test_simple_range(self):
390: (8)                """
391: (8)                Straightforward testing with a mixture of linspace data (for
392: (8)                  consistency). Adding in a 3rd mixture that will then be
393: (8)                  completely ignored. All test values have been precomputed and
394: (8)                  the shouldn't change.
395: (8)
396: (22)            basic_test = {
397: (29)              50: {'fd': 8, 'scott': 8, 'rice': 15,
398: (22)                'sturges': 14, 'auto': 14, 'stone': 8},
399: (29)              500: {'fd': 15, 'scott': 16, 'rice': 32,
400: (21)                'sturges': 20, 'auto': 20, 'stone': 80},
401: (8)              5000: {'fd': 33, 'scott': 33, 'rice': 69,
402: (12)                'sturges': 27, 'auto': 33, 'stone': 80}
403: (12)            }
404: (12)            for testlen, expectedResults in basic_test.items():
405: (12)                x1 = np.linspace(-10, -1, testlen // 5 * 2)
406: (12)                x2 = np.linspace(1, 10, testlen // 5 * 3)
407: (12)                x3 = np.linspace(-100, -50, testlen)
408: (12)                x = np.hstack((x1, x2, x3))
409: (12)                for estimator, numbins in expectedResults.items():
410: (16)                  a, b = np.histogram(x, estimator, range = (-20, 20))
411: (16)                  msg = "For the {} estimator".format(estimator)
412: (16)                  msg += " with datasize of {}".format(testlen)
413: (4)                    assert_equal(len(a), numbins, err_msg=msg)
414: (8)            @pytest.mark.parametrize("bins", ['auto', 'fd', 'doane', 'scott',
415: (38)              'stone', 'rice', 'sturges'])
416: (4)            def test_signed_integer_data(self, bins):
417: (8)                a = np.array([-2, 0, 127], dtype=np.int8)
418: (8)                hist, edges = np.histogram(a, bins=bins)
419: (8)                hist32, edges32 = np.histogram(a.astype(np.int32), bins=bins)
420: (8)                assert_array_equal(hist, hist32)
421: (8)                assert_array_equal(edges, edges32)
422: (8)
423: (8)            def test_simple_weighted(self):
424: (8)                """
425: (8)                Check that weighted data raises a TypeError
426: (26)            estimator_list = ['fd', 'scott', 'rice', 'sturges', 'auto']
427: (0)            for estimator in estimator_list:
428: (4)                assert_raises(TypeError, histogram, [1, 2, 3],
429: (8)                  estimator, weights=[1, 2, 3])
430: (22)            class TestHistogramdd:
431: (4)                def test_simple(self):
432: (8)                    x = np.array([[-.5, .5, 1.5], [-.5, 1.5, 2.5], [-.5, 2.5, .5],
433: (8)                      [.5, .5, 1.5], [.5, 1.5, 2.5], [.5, 2.5, 2.5]])

```

```

431: (8)
432: (31)
433: (8)
434: (27)
435: (8)
436: (8)
437: (8)
438: (8)
439: (8)
440: (31)
441: (31)
442: (8)
443: (27)
444: (8)
445: (8)
446: (8)
447: (12)
448: (8)
449: (27)
450: (27)
451: (27)
452: (8)
453: (8)
454: (8)
455: (8)
456: (8)
457: (4)
458: (8)
459: (16)
460: (8)
461: (8)
462: (12)
463: (12)
464: (4)
465: (8)
466: (16)
467: (16)
468: (16)
469: (16)
470: (16)
471: (8)
472: (8)
473: (12)
474: (12)
475: (4)
476: (8)
477: (8)
478: (8)
479: (8)
480: (8)
481: (8)
482: (8)
483: (8)
484: (8)
485: (4)
486: (8)
487: (8)
488: (8)
489: (4)
490: (8)
491: (8)
492: (8)
493: (8)
494: (4)
495: (8)
496: (8)
497: (8)
498: (8)
499: (12)

        H, edges = histogramdd(x, (2, 3, 3),
                                range=[[-1, 1], [0, 3], [0, 3]])
        answer = np.array([[0, 1, 0], [0, 0, 1], [1, 0, 0],
                           [0, 1, 0], [0, 0, 1], [0, 0, 1]])
        assert_array_equal(H, answer)
        ed = [[-2, 0, 2], [0, 1, 2, 3], [0, 1, 2, 3]]
        H, edges = histogramdd(x, bins=ed, density=True)
        assert_(np.all(H == answer / 12.))
        H, edges = histogramdd(x, (2, 3, 4),
                               range=[[-1, 1], [0, 3], [0, 4]],
                               density=True)
        answer = np.array([[0, 1, 0, 0], [0, 0, 1, 0], [1, 0, 0, 0],
                           [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0]])
        assert_array_almost_equal(H, answer / 6., 4)
        z = [np.squeeze(y) for y in np.split(x, 3, axis=1)]
        H, edges = histogramdd(
            z, bins=(4, 3, 2), range=[[-2, 2], [0, 3], [0, 2]])
        answer = np.array([[0, 0, 0], [0, 0, 0], [1, 0],
                           [0, 1, 0, 0], [0, 0, 0], [0, 0],
                           [0, 0, 0, 0], [0, 0, 0, 0]])
        assert_array_equal(H, answer)
        Z = np.zeros((5, 5, 5))
        Z[list(range(5)), list(range(5)), list(range(5))] = 1.
        H, edges = histogramdd([np.arange(5), np.arange(5), np.arange(5)], 5)
        assert_array_equal(H, Z)

    def test_shape_3d(self):
        bins = ((5, 4, 6), (6, 4, 5), (5, 6, 4), (4, 6, 5), (6, 5, 4),
                (4, 5, 6))
        r = np.random.rand(10, 3)
        for b in bins:
            H, edges = histogramdd(r, b)
            assert_(H.shape == b)

    def test_shape_4d(self):
        bins = ((7, 4, 5, 6), (4, 5, 7, 6), (5, 6, 4, 7), (7, 6, 5, 4),
                (5, 7, 6, 4), (4, 6, 7, 5), (6, 5, 7, 4), (7, 5, 4, 6),
                (7, 4, 6, 5), (6, 4, 7, 5), (6, 7, 5, 4), (4, 6, 5, 7),
                (4, 7, 5, 6), (5, 4, 6, 7), (5, 7, 4, 6), (6, 7, 4, 5),
                (6, 5, 4, 7), (4, 7, 6, 5), (4, 5, 6, 7), (7, 6, 4, 5),
                (5, 4, 7, 6), (5, 6, 7, 4), (6, 4, 5, 7), (7, 5, 6, 4))
        r = np.random.rand(10, 4)
        for b in bins:
            H, edges = histogramdd(r, b)
            assert_(H.shape == b)

    def test_weights(self):
        v = np.random.rand(100, 2)
        hist, edges = histogramdd(v)
        n_hist, edges = histogramdd(v, density=True)
        w_hist, edges = histogramdd(v, weights=np.ones(100))
        assert_array_equal(w_hist, hist)
        w_hist, edges = histogramdd(v, weights=np.ones(100) * 2, density=True)
        assert_array_equal(w_hist, n_hist)
        w_hist, edges = histogramdd(v, weights=np.ones(100, int) * 2)
        assert_array_equal(w_hist, 2 * hist)

    def test_identical_samples(self):
        x = np.zeros((10, 2), int)
        hist, edges = histogramdd(x, bins=2)
        assert_array_equal(edges[0], np.array([-0.5, 0., 0.5]))

    def test_empty(self):
        a, b = histogramdd([], [], bins=([], [], []))
        assert_array_max_ulp(a, np.array([]))
        a, b = np.histogramdd([], [], bins=2)
        assert_array_max_ulp(a, np.zeros((2, 2, 2)))

    def test_bins_errors(self):
        x = np.arange(8).reshape(2, 4)
        assert_raises(ValueError, np.histogramdd, x, bins=[-1, 2, 4, 5])
        assert_raises(ValueError, np.histogramdd, x, bins=[1, 0.99, 1, 1])
        assert_raises(
            ValueError, np.histogramdd, x, bins=[1, 1, 1, [1, 2, 3, -3]])

```

```

500: (8) assert_(np.histogramdd(x, bins=[1, 1, 1, [1, 2, 3, 4]]))
501: (4) def test_inf_edges(self):
502: (8)     with np.errstate(invalid='ignore'):
503: (12)         x = np.arange(6).reshape(3, 2)
504: (12)         expected = np.array([[1, 0], [0, 1], [0, 1]])
505: (12)         h, e = np.histogramdd(x, bins=[3, [-np.inf, 2, 10]])
506: (12)         assert_allclose(h, expected)
507: (12)         h, e = np.histogramdd(x, bins=[3, np.array([-1, 2, np.inf])])
508: (12)         assert_allclose(h, expected)
509: (12)         h, e = np.histogramdd(x, bins=[3, [-np.inf, 3, np.inf]])
510: (12)         assert_allclose(h, expected)
511: (4) def test_rightmost_binedge(self):
512: (8)     x = [0.9999999995]
513: (8)     bins = [[0., 0.5, 1.0]]
514: (8)     hist, _ = histogramdd(x, bins=bins)
515: (8)     assert_(hist[0] == 0.0)
516: (8)     assert_(hist[1] == 1.)
517: (8)     x = [1.0]
518: (8)     bins = [[0., 0.5, 1.0]]
519: (8)     hist, _ = histogramdd(x, bins=bins)
520: (8)     assert_(hist[0] == 0.0)
521: (8)     assert_(hist[1] == 1.)
522: (8)     x = [1.0000000001]
523: (8)     bins = [[0., 0.5, 1.0]]
524: (8)     hist, _ = histogramdd(x, bins=bins)
525: (8)     assert_(hist[0] == 0.0)
526: (8)     assert_(hist[1] == 0.0)
527: (8)     x = [1.0001]
528: (8)     bins = [[0., 0.5, 1.0]]
529: (8)     hist, _ = histogramdd(x, bins=bins)
530: (8)     assert_(hist[0] == 0.0)
531: (8)     assert_(hist[1] == 0.0)
532: (4) def test_finite_range(self):
533: (8)     vals = np.random.random((100, 3))
534: (8)     histogramdd(vals, range=[[0.0, 1.0], [0.25, 0.75], [0.25, 0.5]])
535: (8)     assert_raises(ValueError, histogramdd, vals,
536: (22)         range=[[0.0, 1.0], [0.25, 0.75], [0.25, np.inf]])
537: (8)     assert_raises(ValueError, histogramdd, vals,
538: (22)         range=[[0.0, 1.0], [np.nan, 0.75], [0.25, 0.5]])
539: (4) def test_equal_edges(self):
540: (8)     """ Test that adjacent entries in an edge array can be equal """
541: (8)     x = np.array([0, 1, 2])
542: (8)     y = np.array([0, 1, 2])
543: (8)     x_edges = np.array([0, 2, 2])
544: (8)     y_edges = 1
545: (8)     hist, edges = histogramdd((x, y), bins=(x_edges, y_edges))
546: (8)     hist_expected = np.array([
547: (12)         [2.],
548: (12)         [1.], # x == 2 falls in the final bin
549: (8)     ])
550: (8)     assert_equal(hist, hist_expected)
551: (4) def test_edge_dtype(self):
552: (8)     """ Test that if an edge array is input, its type is preserved """
553: (8)     x = np.array([0, 10, 20])
554: (8)     y = x / 10
555: (8)     x_edges = np.array([0, 5, 15, 20])
556: (8)     y_edges = x_edges / 10
557: (8)     hist, edges = histogramdd((x, y), bins=(x_edges, y_edges))
558: (8)     assert_equal(edges[0].dtype, x_edges.dtype)
559: (8)     assert_equal(edges[1].dtype, y_edges.dtype)
560: (4) def test_large_integers(self):
561: (8)     big = 2**60 # Too large to represent with a full precision float
562: (8)     x = np.array([0], np.int64)
563: (8)     x_edges = np.array([-1, +1], np.int64)
564: (8)     y = big + x
565: (8)     y_edges = big + x_edges
566: (8)     hist, edges = histogramdd((x, y), bins=(x_edges, y_edges))
567: (8)     assert_equal(hist[0, 0], 1)
568: (4) def test_density_non_uniform_2d(self):

```

```

569: (8)          x_edges = np.array([0, 2, 8])
570: (8)          y_edges = np.array([0, 6, 8])
571: (8)          relative_areas = np.array([
572: (12)            [3, 9],
573: (12)            [1, 3]])
574: (8)          x = np.array([1] + [1]*3 + [7]*3 + [7]*9)
575: (8)          y = np.array([7] + [1]*3 + [7]*3 + [1]*9)
576: (8)          hist, edges = histogramdd((y, x), bins=(y_edges, x_edges))
577: (8)          assert_equal(hist, relative_areas)
578: (8)          hist, edges = histogramdd((y, x), bins=(y_edges, x_edges),
density=True)
579: (8)          assert_equal(hist, 1 / (8*8))
580: (4)          def test_density_non_uniform_1d(self):
581: (8)              v = np.arange(10)
582: (8)              bins = np.array([0, 1, 3, 6, 10])
583: (8)              hist, edges = histogram(v, bins, density=True)
584: (8)              hist_dd, edges_dd = histogramdd((v,), (bins,), density=True)
585: (8)              assert_equal(hist, hist_dd)
586: (8)              assert_equal(edges, edges_dd[0])
-----
```

## File 236 - test\_index\_tricks.py:

```

1: (0)          import pytest
2: (0)          import numpy as np
3: (0)          from numpy.testing import (
4: (4)            assert_, assert_equal, assert_array_equal, assert_almost_equal,
5: (4)            assert_array_almost_equal, assert_raises, assert_raises_regex,
6: (4)            )
7: (0)          from numpy.lib.index_tricks import (
8: (4)            mgrid, ogrid, ndenumerate, fill_diagonal, diag_indices, diag_indices_from,
9: (4)            index_exp, ndindex, r_, s_, ix_
10: (4)           )
11: (0)          class TestRavelUnravelIndex:
12: (4)          def test_basic(self):
13: (8)              assert_equal(np.unravel_index(2, (2, 2)), (1, 0))
14: (8)              assert_equal(np.unravel_index(indices=2,
15: (38)                                shape=(2, 2)),
16: (38)                                (1, 0))
17: (8)              with assert_raises(TypeError):
18: (12)                  np.unravel_index(indices=2, shape=(2, 2))
19: (8)              with assert_raises(TypeError):
20: (12)                  np.unravel_index(2, shape=(2, 2))
21: (8)              with assert_raises(TypeError):
22: (12)                  np.unravel_index(254, dims=(17, 94))
23: (8)              with assert_raises(TypeError):
24: (12)                  np.unravel_index(254, dims=(17, 94))
25: (8)              assert_equal(np.ravel_multi_index((1, 0), (2, 2)), 2)
26: (8)              assert_equal(np.unravel_index(254, (17, 94)), (2, 66))
27: (8)              assert_equal(np.ravel_multi_index((2, 66), (17, 94)), 254)
28: (8)              assert_raises(ValueError, np.unravel_index, -1, (2, 2))
29: (8)              assert_raises(TypeError, np.unravel_index, 0.5, (2, 2))
30: (8)              assert_raises(ValueError, np.unravel_index, 4, (2, 2))
31: (8)              assert_raises(ValueError, np.ravel_multi_index, (-3, 1), (2, 2))
32: (8)              assert_raises(ValueError, np.ravel_multi_index, (2, 1), (2, 2))
33: (8)              assert_raises(ValueError, np.ravel_multi_index, (0, -3), (2, 2))
34: (8)              assert_raises(ValueError, np.ravel_multi_index, (0, 2), (2, 2))
35: (8)              assert_raises(TypeError, np.ravel_multi_index, (0.1, 0.), (2, 2))
36: (8)              assert_equal(np.unravel_index((2*3 + 1)*6 + 4, (4, 3, 6)), [2, 1, 4])
37: (8)              assert_equal(
38: (12)                  np.ravel_multi_index([2, 1, 4], (4, 3, 6)), (2*3 + 1)*6 + 4)
39: (8)              arr = np.array([[3, 6, 6], [4, 5, 1]])
40: (8)              assert_equal(np.ravel_multi_index(arr, (7, 6)), [22, 41, 37])
41: (8)              assert_equal(
42: (12)                  np.ravel_multi_index(arr, (7, 6), order='F'), [31, 41, 13])
43: (8)              assert_equal(
44: (12)                  np.ravel_multi_index(arr, (4, 6), mode='clip'), [22, 23, 19])
45: (8)              assert_equal(np.ravel_multi_index(arr, (4, 4), mode='clip', 'wrap')),
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

46: (21) [12, 13, 13])
47: (8) assert_equal(np.ravel_multi_index((3, 1, 4, 1), (6, 7, 8, 9)), 1621)
48: (8) assert_equal(np.unravel_index(np.array([22, 41, 37]), (7, 6)),
49: (21) [[3, 6, 6], [4, 5, 1]])
50: (8) assert_equal(
51: (12) np.unravel_index(np.array([31, 41, 13]), (7, 6), order='F'),
52: (12) [[3, 6, 6], [4, 5, 1]])
53: (8) assert_equal(np.unravel_index(1621, (6, 7, 8, 9)), [3, 1, 4, 1])
54: (4) def test_empty_indices(self):
55: (8)     msg1 = 'indices must be integral: the provided empty sequence was'
56: (8)     msg2 = 'only int indices permitted'
57: (8)     assert_raises_regex(TypeError, msg1, np.unravel_index, [], (10, 3, 5))
58: (8)     assert_raises_regex(TypeError, msg1, np.unravel_index, (), (10, 3, 5))
59: (8)     assert_raises_regex(TypeError, msg2, np.unravel_index, np.array([]),
60: (28)         (10, 3, 5))
61: (8)     assert_equal(np.unravel_index(np.array([], dtype=int), (10, 3, 5)),
62: (21)         [[], [], []])
63: (8)     assert_raises_regex(TypeError, msg1, np.ravel_multi_index, ([], []),
64: (28)         (10, 3))
65: (8)     assert_raises_regex(TypeError, msg1, np.ravel_multi_index, ([]|,
66: (28)         ['abc']), (10, 3))
67: (8)     assert_raises_regex(TypeError, msg2, np.ravel_multi_index,
68: (20)         (np.array([]), np.array([])), (5, 3))
69: (8)     assert_equal(np.ravel_multi_index(
70: (16)         (np.array[], dtype=int), np.array[], dtype=int)), (5, 3)),
[])
71: (8)     assert_equal(np.ravel_multi_index(np.array([[[], []]], dtype=int),
72: (21)         (5, 3)), [])
73: (4) def test_big_indices(self):
74: (8)     if np.intp == np.int64:
75: (12)         arr = ([1, 29], [3, 5], [3, 117], [19, 2],
76: (19)             [2379, 1284], [2, 2], [0, 1])
77: (12)         assert_equal(
78: (16)             np.ravel_multi_index(arr, (41, 7, 120, 36, 2706, 8, 6)),
79: (16)             [5627771580, 117259570957])
80: (8)         assert_raises(ValueError, np.unravel_index, 1, (2**32-1, 2**31+1))
81: (8)         dummy_arr = ([0], [0])
82: (8)         half_max = np.iinfo(np.intp).max // 2
83: (8)         assert_equal(
84: (12)             np.ravel_multi_index(dummy_arr, (half_max, 2)), [0])
85: (8)         assert_raises(ValueError,
86: (12)             np.ravel_multi_index, dummy_arr, (half_max+1, 2))
87: (8)         assert_equal(
88: (12)             np.ravel_multi_index(dummy_arr, (half_max, 2), order='F'), [0])
89: (8)         assert_raises(ValueError,
90: (12)             np.ravel_multi_index, dummy_arr, (half_max+1, 2), order='F')
91: (4) def test_dtotypes(self):
92: (8)     for dtype in [np.int16, np.uint16, np.int32,
93: (22)         np.uint32, np.int64, np.uint64]:
94: (12)         coords = np.array(
95: (16)             [[1, 0, 1, 2, 3, 4], [1, 6, 1, 3, 2, 0]], dtype=dtype)
96: (12)         shape = (5, 8)
97: (12)         uncoords = 8*coords[0]+coords[1]
98: (12)         assert_equal(np.ravel_multi_index(coords, shape), uncoords)
99: (12)         assert_equal(coords, np.unravel_index(uncoords, shape))
100: (12)         uncoords = coords[0]+5*coords[1]
101: (12)         assert_equal(
102: (16)             np.ravel_multi_index(coords, shape, order='F'), uncoords)
103: (12)         assert_equal(coords, np.unravel_index(uncoords, shape, order='F'))
104: (12)         coords = np.array(
105: (16)             [[1, 0, 1, 2, 3, 4], [1, 6, 1, 3, 2, 0], [1, 3, 1, 0, 9, 5]],
106: (16)             dtype=dtype)
107: (12)         shape = (5, 8, 10)
108: (12)         uncoords = 10*(8*coords[0]+coords[1])+coords[2]
109: (12)         assert_equal(np.ravel_multi_index(coords, shape), uncoords)
110: (12)         assert_equal(coords, np.unravel_index(uncoords, shape))
111: (12)         uncoords = coords[0]+5*(coords[1]+8*coords[2])
112: (12)         assert_equal(

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

113: (16) np.ravel_multi_index(coords, shape, order='F'), uncoords)
114: (12) assert_equal(coords, np.unravel_index(uncoords, shape, order='F'))
115: (4) def test_clipmodes(self):
116: (8)     assert_equal(
117: (12)         np.ravel_multi_index([5, 1, -1, 2], (4, 3, 7, 12), mode='wrap'),
118: (12)         np.ravel_multi_index([1, 1, 6, 2], (4, 3, 7, 12)))
119: (8)     assert_equal(np.ravel_multi_index([5, 1, -1, 2], (4, 3, 7, 12),
120: (42)             mode=(
121: (46)                 'wrap', 'raise', 'clip',
122: (21)             'raise')),,
123: (8)         np.ravel_multi_index([1, 1, 0, 2], (4, 3, 7, 12)))
124: (12)     assert_raises(
125: (4)         ValueError, np.ravel_multi_index, [5, 1, -1, 2], (4, 3, 7, 12))
126: (8) def test_writeability(self):
127: (8)     x, y = np.unravel_index([1, 2, 3], (4, 5))
128: (8)     assert_(x.flags.writeable)
129: (8)     assert_(y.flags.writeable)
130: (4) def test_0d(self):
131: (8)     x = np.unravel_index(0, ())
132: (8)     assert_equal(x, ())
133: (8)     assert_raises_regex(ValueError, "0d array", np.unravel_index, [0], ())
134: (12)     assert_raises_regex(
135: (4)         ValueError, "out of bounds", np.unravel_index, [1], ())
136: (4) @pytest.mark.parametrize("mode", ["clip", "wrap", "raise"])
137: (8) def test_empty_array_ravel(self, mode):
138: (20)     res = np.ravel_multi_index(
139: (8)         np.zeros((3, 0), dtype=np.intp), (2, 1, 0), mode=mode)
140: (8)     assert(res.shape == (0,))
141: (12)     with assert_raises(ValueError):
142: (20)         np.ravel_multi_index(
143: (4)             np.zeros((3, 1), dtype=np.intp), (2, 1, 0), mode=mode)
144: (8) def test_empty_array_unravel(self):
145: (8)     res = np.unravel_index(np.zeros(0, dtype=np.intp), (2, 1, 0))
146: (8)     assert(len(res) == 3)
147: (8)     assert(all(a.shape == (0,) for a in res))
148: (12)     with assert_raises(ValueError):
149: (0)         np.unravel_index([1], (2, 1, 0))
150: (4) class TestGrid:
151: (8)     def test_basic(self):
152: (8)         a = mgrid[-1:1:10j]
153: (8)         b = mgrid[-1:1:0.1]
154: (8)         assert_(a.shape == (10,))
155: (8)         assert_(b.shape == (20,))
156: (8)         assert_(a[0] == -1)
157: (8)         assert_almost_equal(a[-1], 1)
158: (8)         assert_(b[0] == -1)
159: (8)         assert_almost_equal(b[1]-b[0], 0.1, 11)
160: (8)         assert_almost_equal(b[-1], b[0]+19*0.1, 11)
161: (4)         assert_almost_equal(a[1]-a[0], 2.0/9.0, 11)
162: (8)     def test_linspace_equivalence(self):
163: (8)         y, st = np.linspace(2, 10, retstep=True)
164: (8)         assert_almost_equal(st, 8/49.0)
165: (4)         assert_array_almost_equal(y, mgrid[2:10:50j], 13)
166: (8)     def test_nd(self):
167: (8)         c = mgrid[-1:1:10j, -2:2:10j]
168: (8)         d = mgrid[-1:1:0.1, -2:2:0.2]
169: (8)         assert_(c.shape == (2, 10, 10))
170: (8)         assert_(d.shape == (2, 20, 20))
171: (8)         assert_array_equal(c[0][0, :], -np.ones(10, 'd'))
172: (8)         assert_array_equal(c[1][:, 0], -2*np.ones(10, 'd'))
173: (8)         assert_array_almost_equal(c[0][-1, :], np.ones(10, 'd'), 11)
174: (8)         assert_array_almost_equal(c[1][:, -1], 2*np.ones(10, 'd'), 11)
175: (34)         assert_array_almost_equal(d[0, 1, :] - d[0, 0, :],
176: (8)             0.1*np.ones(20, 'd'), 11)
177: (34)         assert_array_almost_equal(d[1, :, 1] - d[1, :, 0],
178: (4)             0.2*np.ones(20, 'd'), 11)
179: (8)     def test_sparse(self):
180: (8)         grid_full = mgrid[-1:1:10j, -2:2:10j]

```

```

181: (8)             grid_broadcast = np.broadcast_arrays(*grid_sparse)
182: (8)             for f, b in zip(grid_full, grid_broadcast):
183: (12)               assert_equal(f, b)
184: (4)             @pytest.mark.parametrize("start, stop, step, expected", [
185: (8)               (None, 10, 10j, (200, 10)),
186: (8)               (-10, 20, None, (1800, 30)),
187: (8)             ])
188: (4)             def test_mgrid_size_none_handling(self, start, stop, step, expected):
189: (8)               grid = mgrid[start:stop:step, start:stop:step]
190: (8)               grid_small = mgrid[start:stop:step]
191: (8)               assert_equal(grid.size, expected[0])
192: (8)               assert_equal(grid_small.size, expected[1])
193: (4)             def test_accepts_nploating(self):
194: (8)               grid64 = mgrid[0.1:0.33:0.1, ]
195: (8)               grid32 = mgrid[np.float32(0.1):np.float32(0.33):np.float32(0.1), ]
196: (8)               assert_(grid32.dtype == np.float64)
197: (8)               assert_array_almost_equal(grid64, grid32)
198: (8)               grid64 = mgrid[0.1:0.33:0.1]
199: (8)               grid32 = mgrid[np.float32(0.1):np.float32(0.33):np.float32(0.1)]
200: (8)               assert_(grid32.dtype == np.float64)
201: (8)               assert_array_almost_equal(grid64, grid32)
202: (4)             def test_accepts_longdouble(self):
203: (8)               grid64 = mgrid[0.1:0.33:0.1, ]
204: (8)               grid128 = mgrid[
205: (12)                 np.longdouble(0.1):np.longdouble(0.33):np.longdouble(0.1),
206: (8)               ]
207: (8)               assert_(grid128.dtype == np.longdouble)
208: (8)               assert_array_almost_equal(grid64, grid128)
209: (8)               grid128c_a = mgrid[0:np.longdouble(1):3.4j]
210: (8)               grid128c_b = mgrid[0:np.longdouble(1):3.4j, ]
211: (8)               assert_(grid128c_a.dtype == grid128c_b.dtype == np.longdouble)
212: (8)               assert_array_equal(grid128c_a, grid128c_b[0])
213: (8)               grid64 = mgrid[0.1:0.33:0.1]
214: (8)               grid128 = mgrid[
215: (12)                 np.longdouble(0.1):np.longdouble(0.33):np.longdouble(0.1)
216: (8)               ]
217: (8)               assert_(grid128.dtype == np.longdouble)
218: (8)               assert_array_almost_equal(grid64, grid128)
219: (4)             def test_accepts_npcomplexfloating(self):
220: (8)               assert_array_almost_equal(
221: (12)                 mgrid[0.1:0.3:3j, ], mgrid[0.1:0.3:np.complex64(3j), ]
222: (8)               )
223: (8)               assert_array_almost_equal(
224: (12)                 mgrid[0.1:0.3:3j], mgrid[0.1:0.3:np.complex64(3j)]
225: (8)               )
226: (8)               grid64_a = mgrid[0.1:0.3:3.3j]
227: (8)               grid64_b = mgrid[0.1:0.3:3.3j, ][0]
228: (8)               assert_(grid64_a.dtype == grid64_b.dtype == np.float64)
229: (8)               assert_array_equal(grid64_a, grid64_b)
230: (8)               grid128_a = mgrid[0.1:0.3:np.clongdouble(3.3j)]
231: (8)               grid128_b = mgrid[0.1:0.3:np.clongdouble(3.3j), ][0]
232: (8)               assert_(grid128_a.dtype == grid128_b.dtype == np.longdouble)
233: (8)               assert_array_equal(grid64_a, grid64_b)
234: (0)             class TestConcatenator:
235: (4)               def test_1d(self):
236: (8)                 assert_array_equal(r_[1, 2, 3, 4, 5, 6], np.array([1, 2, 3, 4, 5, 6]))
237: (8)                 b = np.ones(5)
238: (8)                 c = r_[b, 0, 0, b]
239: (8)                 assert_array_equal(c, [1, 1, 1, 1, 0, 0, 1, 1, 1, 1])
240: (4)               def test_mixed_type(self):
241: (8)                 g = r_[10.1, 1:10]
242: (8)                 assert_(g.dtype == 'f8')
243: (4)               def test_more_mixed_type(self):
244: (8)                 g = r_[-10.1, np.array([1]), np.array([2, 3, 4]), 10.0]
245: (8)                 assert_(g.dtype == 'f8')
246: (4)               def test_complex_step(self):
247: (8)                 g = r_[0:36:100j]
248: (8)                 assert_(g.shape == (100,))
249: (8)                 g = r_[0:36:np.complex64(100j)]
```

```

250: (8)             assert_(g.shape == (100,))
251: (4)             def test_2d(self):
252: (8)                 b = np.random.rand(5, 5)
253: (8)                 c = np.random.rand(5, 5)
254: (8)                 d = r_[1, b, c] # append columns
255: (8)                 assert_(d.shape == (5, 10))
256: (8)                 assert_array_equal(d[:, :5], b)
257: (8)                 assert_array_equal(d[:, 5:], c)
258: (8)                 d = r_[b, c]
259: (8)                 assert_(d.shape == (10, 5))
260: (8)                 assert_array_equal(d[:5, :], b)
261: (8)                 assert_array_equal(d[5:, :], c)
262: (4)             def test_0d(self):
263: (8)                 assert_equal(r_[0, np.array(1), 2], [0, 1, 2])
264: (8)                 assert_equal(r_[[0, 1, 2], np.array(3)], [0, 1, 2, 3])
265: (8)                 assert_equal(r_[np.array(0), [1, 2, 3]], [0, 1, 2, 3])
266: (0)             class TestNdenumerate:
267: (4)                 def test_basic(self):
268: (8)                     a = np.array([[1, 2], [3, 4]])
269: (8)                     assert_equal(list(ndenumerate(a)),
270: (21)                         [((0, 0), 1), ((0, 1), 2), ((1, 0), 3), ((1, 1), 4)])
271: (0)             class TestIndexExpression:
272: (4)                 def test_regression_1(self):
273: (8)                     a = np.arange(2)
274: (8)                     assert_equal(a[:-1], a[s_[:-1]])
275: (8)                     assert_equal(a[:-1], a[index_exp[:-1]])
276: (4)                 def test_simple_1(self):
277: (8)                     a = np.random.rand(4, 5, 6)
278: (8)                     assert_equal(a[:, :3, [1, 2]], a[index_exp[:, :3, [1, 2]]])
279: (8)                     assert_equal(a[:, :3, [1, 2]], a[s_[:, :3, [1, 2]]])
280: (0)             class TestIx_:
281: (4)                 def test_regression_1(self):
282: (8)                     a, = np.ix_(range(0))
283: (8)                     assert_equal(a.dtype, np.intp)
284: (8)                     a, = np.ix_()
285: (8)                     assert_equal(a.dtype, np.intp)
286: (8)                     a, = np.ix_(np.array([], dtype=np.float32))
287: (8)                     assert_equal(a.dtype, np.float32)
288: (4)                 def test_shape_and_dtype(self):
289: (8)                     sizes = (4, 5, 3, 2)
290: (8)                     for func in (range, np.arange):
291: (12)                         arrays = np.ix_(*[func(sz) for sz in sizes])
292: (12)                         for k, (a, sz) in enumerate(zip(arrays, sizes)):
293: (16)                             assert_equal(a.shape[k], sz)
294: (16)                             assert_(all(sh == 1 for j, sh in enumerate(a.shape) if j != k))
295: (16)                             assert_(np.issubdtype(a.dtype, np.integer))
296: (4)                 def test_bool(self):
297: (8)                     bool_a = [True, False, True, True]
298: (8)                     int_a, = np.nonzero(bool_a)
299: (8)                     assert_equal(np.ix_(bool_a)[0], int_a)
300: (4)                 def test_1d_only(self):
301: (8)                     idx2d = [[1, 2, 3], [4, 5, 6]]
302: (8)                     assert_raises(ValueError, np.ix_, idx2d)
303: (4)                 def test_repeated_input(self):
304: (8)                     length_of_vector = 5
305: (8)                     x = np.arange(length_of_vector)
306: (8)                     out = ix_(x, x)
307: (8)                     assert_equal(out[0].shape, (length_of_vector, 1))
308: (8)                     assert_equal(out[1].shape, (1, length_of_vector))
309: (8)                     assert_equal(x.shape, (length_of_vector,))
310: (0)                 def test_c_():
311: (4)                     a = np.c_[np.array([[1, 2, 3]]), 0, 0, np.array([[4, 5, 6]])]
312: (4)                     assert_equal(a, [[1, 2, 3, 0, 0, 4, 5, 6]])
313: (0)             class TestFillDiagonal:
314: (4)                 def test_basic(self):
315: (8)                     a = np.zeros((3, 3), int)
316: (8)                     fill_diagonal(a, 5)
317: (8)                     assert_array_equal(

```

```

318: (12)             a, np.array([[5, 0, 0],
319: (25)                 [0, 5, 0],
320: (25)                 [0, 0, 5]])
321: (12)
322: (4)         def test_tall_matrix(self):
323: (8)             a = np.zeros((10, 3), int)
324: (8)             fill_diagonal(a, 5)
325: (8)             assert_array_equal(
326: (12)                 a, np.array([[5, 0, 0],
327: (25)                     [0, 5, 0],
328: (25)                     [0, 0, 5],
329: (25)                     [0, 0, 0],
330: (25)                     [0, 0, 0],
331: (25)                     [0, 0, 0],
332: (25)                     [0, 0, 0],
333: (25)                     [0, 0, 0],
334: (25)                     [0, 0, 0],
335: (25)                     [0, 0, 0]])
336: (12)
337: (4)         def test_tall_matrix_wrap(self):
338: (8)             a = np.zeros((10, 3), int)
339: (8)             fill_diagonal(a, 5, True)
340: (8)             assert_array_equal(
341: (12)                 a, np.array([[5, 0, 0],
342: (25)                     [0, 5, 0],
343: (25)                     [0, 0, 5],
344: (25)                     [0, 0, 0],
345: (25)                     [5, 0, 0],
346: (25)                     [0, 5, 0],
347: (25)                     [0, 0, 5],
348: (25)                     [0, 0, 0],
349: (25)                     [5, 0, 0],
350: (25)                     [0, 5, 0]])
351: (12)
352: (4)         def test_wide_matrix(self):
353: (8)             a = np.zeros((3, 10), int)
354: (8)             fill_diagonal(a, 5)
355: (8)             assert_array_equal(
356: (12)                 a, np.array([[5, 0, 0, 0, 0, 0, 0, 0, 0, 0],
357: (25)                     [0, 5, 0, 0, 0, 0, 0, 0, 0, 0],
358: (25)                     [0, 0, 5, 0, 0, 0, 0, 0, 0, 0]])
359: (12)
360: (4)         def test_operate_4d_array(self):
361: (8)             a = np.zeros((3, 3, 3, 3), int)
362: (8)             fill_diagonal(a, 4)
363: (8)             i = np.array([0, 1, 2])
364: (8)             assert_equal(np.where(a != 0), (i, i, i, i))
365: (4)         def test_low_dim_handling(self):
366: (8)             a = np.zeros(3, int)
367: (8)             with assert_raises_regex(ValueError, "at least 2-d"):
368: (12)                 fill_diagonal(a, 5)
369: (4)         def test_hetero_shape_handling(self):
370: (8)             a = np.zeros((3, 3, 7, 3), int)
371: (8)             with assert_raises_regex(ValueError, "equal length"):
372: (12)                 fill_diagonal(a, 2)
373: (0)         def test_diag_indices():
374: (4)             di = diag_indices(4)
375: (4)             a = np.array([[1, 2, 3, 4],
376: (18)                 [5, 6, 7, 8],
377: (18)                 [9, 10, 11, 12],
378: (18)                 [13, 14, 15, 16]])
379: (4)             a[di] = 100
380: (4)             assert_array_equal(
381: (8)                 a, np.array([[100, 2, 3, 4],
382: (21)                     [5, 100, 7, 8],
383: (21)                     [9, 10, 100, 12],
384: (21)                     [13, 14, 15, 100]]))
385: (8)
386: (4)             d3 = diag_indices(2, 3)

```

```

387: (4)             a = np.zeros((2, 2, 2), int)
388: (4)             a[d3] = 1
389: (4)             assert_array_equal(
390: (8)                 a, np.array([[[1, 0],
391: (22)                     [0, 0]],
392: (21)                     [[0, 0],
393: (22)                         [0, 1]]]))
394: (8)
395: (0)         class TestDiagIndicesFrom:
396: (4)             def test_diag_indices_from(self):
397: (8)                 x = np.random.random((4, 4))
398: (8)                 r, c = diag_indices_from(x)
399: (8)                 assert_array_equal(r, np.arange(4))
400: (8)                 assert_array_equal(c, np.arange(4))
401: (4)             def test_error_small_input(self):
402: (8)                 x = np.ones(7)
403: (8)                 with assert_raises_regex(ValueError, "at least 2-d"):
404: (12)                     diag_indices_from(x)
405: (4)             def test_error_shape_mismatch(self):
406: (8)                 x = np.zeros((3, 3, 2, 3), int)
407: (8)                 with assert_raises_regex(ValueError, "equal length"):
408: (12)                     diag_indices_from(x)
409: (0)
410: (4)             def test_ndindex():
411: (4)                 x = list(ndindex(1, 2, 3))
412: (4)                 expected = [ix for ix, e in ndenumerate(np.zeros((1, 2, 3)))]
413: (4)                 assert_array_equal(x, expected)
414: (4)                 x = list(ndindex((1, 2, 3)))
415: (4)                 assert_array_equal(x, expected)
416: (4)                 x = list(ndindex((3,)))
417: (4)                 assert_array_equal(x, list(ndindex(3)))
418: (4)                 x = list(ndindex())
419: (4)                 assert_equal(x, [()])
420: (4)                 x = list(ndindex(()))
421: (4)                 assert_equal(x, [()])
422: (4)                 x = list(ndindex(*[0]))
423: (4)                 assert_equal(x, [])

```

-----

## File 237 - test\_io.py:

```

1: (0)             import sys
2: (0)             import gc
3: (0)             import gzip
4: (0)             import os
5: (0)             import threading
6: (0)             import time
7: (0)             import warnings
8: (0)             import io
9: (0)             import re
10: (0)            import pytest
11: (0)            from pathlib import Path
12: (0)            from tempfile import NamedTemporaryFile
13: (0)            from io import BytesIO, StringIO
14: (0)            from datetime import datetime
15: (0)            import locale
16: (0)            from multiprocessing import Value, get_context
17: (0)            from ctypes import c_bool
18: (0)            import numpy as np
19: (0)            import numpy.ma as ma
20: (0)            from numpy.lib._iotools import ConverterError, ConversionWarning
21: (0)            from numpy.compat import asbytes
22: (0)            from numpy.ma.testutils import assert_equal
23: (0)            from numpy.testing import (
24: (4)                assert_warns, assert_, assert_raises_regex, assert_raises,
25: (4)                assert_allclose, assert_array_equal, temppath, tempdir, IS_PYPY,
26: (4)                HAS_REFCOUNT, suppress_warnings, assert_no_gc_cycles, assert_no_warnings,
27: (4)                break_cycles, IS_WASM
28: (4)            )

```

```

29: (0)
30: (0)
31: (4)
32: (4)
33: (4)
34: (4)
35: (4)
36: (4)
37: (4)
38: (8)
39: (4)
40: (8)
41: (4)
42: (8)
43: (0)
44: (0)
45: (4)
46: (4)
47: (0)
48: (4)
49: (0)
50: (4)
51: (4)
52: (0)
53: (4)
54: (0)
55: (4)
56: (4)
57: (4)
58: (4)
59: (4)
60: (8)
61: (4)
62: (0)
63: (4)
64: (8)
65: (8)
66: (12)
67: (8)
68: (12)
69: (12)
70: (8)
71: (12)
72: (8)
73: (12)
74: (8)
75: (12)
76: (8)
77: (8)
78: (8)
79: (8)
80: (8)
81: (12)
82: (12)
83: (8)
84: (12)
85: (12)
86: (8)
87: (12)
88: (12)
89: (12)
90: (12)
91: (12)
BytesIO):
92: (16)
93: (12)
94: (12)
95: (12)
96: (8)

from numpy.testing._private.utils import requires_memory
class TextIO(BytesIO):
    """Helper IO class.
    Writes encode strings to bytes if needed, reads return bytes.
    This makes it easier to emulate files opened in binary mode
    without needing to explicitly convert strings to bytes in
    setting up the test data.
    """
    def __init__(self, s=""):
        BytesIO.__init__(self, asbytes(s))
    def write(self, s):
        BytesIO.write(self, asbytes(s))
    def writelines(self, lines):
        BytesIO.writelines(self, [asbytes(s) for s in lines])
IS_64BIT = sys.maxsize > 2**32
try:
    import bz2
    HAS_BZ2 = True
except ImportError:
    HAS_BZ2 = False
try:
    import lzma
    HAS_LZMA = True
except ImportError:
    HAS_LZMA = False
def strftime(s, fmt=None):
    """
    This function is available in the datetime module only from Python >=
    2.5.
    """
    if type(s) == bytes:
        s = s.decode("latin1")
    return datetime.strptime(s, fmt)[:3]
class RoundtripTest:
    def roundtrip(self, save_func, *args, **kwargs):
        """
        save_func : callable
            Function used to save arrays to file.
        file_on_disk : bool
            If true, store the file on disk, instead of in a
            string buffer.
        save_kwds : dict
            Parameters passed to `save_func`.
        load_kwds : dict
            Parameters passed to `numpy.load` .
        args : tuple of arrays
            Arrays stored to file.
        """
        save_kwds = kwargs.get('save_kwds', {})
        load_kwds = kwargs.get('load_kwds', {"allow_pickle": True})
        file_on_disk = kwargs.get('file_on_disk', False)
        if file_on_disk:
            target_file = NamedTemporaryFile(delete=False)
            load_file = target_file.name
        else:
            target_file = BytesIO()
            load_file = target_file
        try:
            arr = args
            save_func(target_file, *arr, **save_kwds)
            target_file.flush()
            target_file.seek(0)
            if sys.platform == 'win32' and not isinstance(target_file,
                BytesIO):
                target_file.close()
            arr_reloaded = np.load(load_file, **load_kwds)
            self.arr = arr
            self.arr_reloaded = arr_reloaded
        finally:

```

```

97: (12)             if not isinstance(target_file, BytesIO):
98: (16)                 target_file.close()
99: (16)                 if 'arr_reloaded' in locals():
100: (20)                     if not isinstance(arr_reloaded, np.lib.npyio.NpzFile):
101: (24)                         os.remove(target_file.name)
102: (4)             def check_roundtrips(self, a):
103: (8)                 self.roundtrip(a)
104: (8)                 self.roundtrip(a, file_on_disk=True)
105: (8)                 self.roundtrip(np.asfortranarray(a))
106: (8)                 self.roundtrip(np.asfortranarray(a), file_on_disk=True)
107: (8)                 if a.shape[0] > 1:
108: (12)                     self.roundtrip(np.asfortranarray(a)[1:])
109: (12)                     self.roundtrip(np.asfortranarray(a)[1:], file_on_disk=True)
110: (4)             def test_array(self):
111: (8)                 a = np.array([], float)
112: (8)                 self.check_roundtrips(a)
113: (8)                 a = np.array([[1, 2], [3, 4]], float)
114: (8)                 self.check_roundtrips(a)
115: (8)                 a = np.array([[1, 2], [3, 4]], int)
116: (8)                 self.check_roundtrips(a)
117: (8)                 a = np.array([[1 + 5j, 2 + 6j], [3 + 7j, 4 + 8j]], dtype=np.csingle)
118: (8)                 self.check_roundtrips(a)
119: (8)                 a = np.array([[1 + 5j, 2 + 6j], [3 + 7j, 4 + 8j]], dtype=np.cdouble)
120: (8)                 self.check_roundtrips(a)
121: (4)             def test_array_object(self):
122: (8)                 a = np.array([], object)
123: (8)                 self.check_roundtrips(a)
124: (8)                 a = np.array([[1, 2], [3, 4]], object)
125: (8)                 self.check_roundtrips(a)
126: (4)             def test_1D(self):
127: (8)                 a = np.array([1, 2, 3, 4], int)
128: (8)                 self.roundtrip(a)
129: (4)             @pytest.mark.skipif(sys.platform == 'win32', reason="Fails on Win32")
130: (4)             def test_mmap(self):
131: (8)                 a = np.array([[1, 2.5], [4, 7.3]])
132: (8)                 self.roundtrip(a, file_on_disk=True, load_kwds={'mmap_mode': 'r'})
133: (8)                 a = np.asfortranarray([[1, 2.5], [4, 7.3]])
134: (8)                 self.roundtrip(a, file_on_disk=True, load_kwds={'mmap_mode': 'r'})
135: (4)             def test_record(self):
136: (8)                 a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
137: (8)                 self.check_roundtrips(a)
138: (4)             @pytest.mark.slow
139: (4)             def test_format_2_0(self):
140: (8)                 dt = [(("%d" % i) * 100, float) for i in range(500)]
141: (8)                 a = np.ones(1000, dtype=dt)
142: (8)                 with warnings.catch_warnings(record=True):
143: (12)                     warnings.filterwarnings('always', '', UserWarning)
144: (12)                     self.check_roundtrips(a)
145: (0)             class TestSaveLoad(RoundtripTest):
146: (4)                 def roundtrip(self, *args, **kwargs):
147: (8)                     RoundtripTest.roundtrip(self, np.save, *args, **kwargs)
148: (8)                     assert_equal(self.arr[0], self.arr_reloaded)
149: (8)                     assert_equal(self.arr[0].dtype, self.arr_reloaded.dtype)
150: (8)                     assert_equal(self.arr[0].flags.fnc, self.arr_reloaded.flags.fnc)
151: (0)             class TestSavezLoad(RoundtripTest):
152: (4)                 def roundtrip(self, *args, **kwargs):
153: (8)                     RoundtripTest.roundtrip(self, np.savez, *args, **kwargs)
154: (8)                     try:
155: (12)                         for n, arr in enumerate(self.arr):
156: (16)                             reloaded = self.arr_reloaded['arr_%d' % n]
157: (16)                             assert_equal(arr, reloaded)
158: (16)                             assert_equal(arr.dtype, reloaded.dtype)
159: (16)                             assert_equal(arr.flags.fnc, reloaded.flags.fnc)
160: (8)                     finally:
161: (12)                         if self.arr_reloaded.fid:
162: (16)                             self.arr_reloaded.fid.close()
163: (16)                             os.remove(self.arr_reloaded.fid.name)
164: (4)             @pytest.mark.skipif(IS_PYPY, reason="Hangs on PyPy")
165: (4)             @pytest.mark.skipif(not IS_64BIT, reason="Needs 64bit platform")

```

```

166: (4)          @pytest.mark.slow
167: (4)          def test_big_arrays(self):
168: (8)          L = (1 << 31) + 100000
169: (8)          a = np.empty(L, dtype=np.uint8)
170: (8)          with temppath(prefix="numpy_test_big_arrays_", suffix=".npz") as tmp:
171: (12)          np.savez(tmp, a=a)
172: (12)          del a
173: (12)          npfile = np.load(tmp)
174: (12)          a = npfile['a'] # Should succeed
175: (12)          npfile.close()
176: (12)          del a # Avoid pyflakes unused variable warning.
177: (4)          def test_multiple_arrays(self):
178: (8)          a = np.array([[1, 2], [3, 4]], float)
179: (8)          b = np.array([[1 + 2j, 2 + 7j], [3 - 6j, 4 + 12j]], complex)
180: (8)          self.roundtrip(a, b)
181: (4)          def test_named_arrays(self):
182: (8)          a = np.array([[1, 2], [3, 4]], float)
183: (8)          b = np.array([[1 + 2j, 2 + 7j], [3 - 6j, 4 + 12j]], complex)
184: (8)          c = BytesIO()
185: (8)          np.savez(c, file_a=a, file_b=b)
186: (8)          c.seek(0)
187: (8)          l = np.load(c)
188: (8)          assert_equal(a, l['file_a'])
189: (8)          assert_equal(b, l['file_b'])
190: (4)          def test_tuple_getitem_raises(self):
191: (8)          a = np.array([1, 2, 3])
192: (8)          f = BytesIO()
193: (8)          np.savez(f, a=a)
194: (8)          f.seek(0)
195: (8)          l = np.load(f)
196: (8)          with pytest.raises(KeyError, match="(1, 2)"):
197: (12)          l[1, 2]
198: (4)          def test_BagObj(self):
199: (8)          a = np.array([[1, 2], [3, 4]], float)
200: (8)          b = np.array([[1 + 2j, 2 + 7j], [3 - 6j, 4 + 12j]], complex)
201: (8)          c = BytesIO()
202: (8)          np.savez(c, file_a=a, file_b=b)
203: (8)          c.seek(0)
204: (8)          l = np.load(c)
205: (8)          assert_equal(sorted(dir(l.f)), ['file_a','file_b'])
206: (8)          assert_equal(a, l.f.file_a)
207: (8)          assert_equal(b, l.f.file_b)
208: (4)          @pytest.mark.skipif(IS_WASM, reason="Cannot start thread")
209: (4)          def test_savez_filename_clashes(self):
210: (8)          def writer(error_list):
211: (12)          with temppath(suffix='.npz') as tmp:
212: (16)          arr = np.random.randn(500, 500)
213: (16)
214: (20)          try:
215: (16)              np.savez(tmp, arr=arr)
216: (20)          except OSError as err:
217: (8)              error_list.append(err)
218: (8)          errors = []
219: (19)          threads = [threading.Thread(target=writer, args=(errors,))]
220: (8)          for j in range(3)]
221: (12)          for t in threads:
222: (8)              t.start()
223: (8)          for t in threads:
224: (12)              t.join()
225: (8)          if errors:
226: (12)              raise AssertionError(errors)
227: (4)          def test_not_closing_opened_fid(self):
228: (8)          with temppath(suffix='.npz') as tmp:
229: (12)          with open(tmp, 'wb') as fp:
230: (16)              np.savez(fp, data='LOVELY LOAD')
231: (12)          with open(tmp, 'rb', 10000) as fp:
232: (16)              fp.seek(0)
233: (16)              assert_(not fp.closed)
234: (16)              np.load(fp)['data']

```

```

235: (16)                      fp.seek(0)
236: (16)                      assert_(not fp.closed)
237: (4)                       @pytest.mark.slow_pypy
238: (4)                       def test_closing_fid(self):
239: (8)                        with temppath(suffix='.npz') as tmp:
240: (12)                          np.savez(tmp, data='LOVELY LOAD')
241: (12)                          with suppress_warnings() as sup:
242: (16)                            sup.filter(ResourceWarning) # TODO: specify exact message
243: (16)                            for i in range(1, 1025):
244: (20)                              try:
245: (24)                                np.load(tmp)["data"]
246: (20)                              except Exception as e:
247: (24)                                msg = "Failed to load data from a file: %s" % e
248: (24)                                raise AssertionError(msg)
249: (20)                              finally:
250: (24)                                if IS_PYPY:
251: (28)                                  gc.collect()
252: (4)                       def test_closing_zipfile_after_load(self):
253: (8)                         prefix = 'numpy_test_closing_zipfile_after_load_'
254: (8)                         with temppath(suffix='.npz', prefix=prefix) as tmp:
255: (12)                           np.savez(tmp, lab='place holder')
256: (12)                           data = np.load(tmp)
257: (12)                           fp = data.zip.fp
258: (12)                           data.close()
259: (12)                           assert_(fp.closed)
260: (4)                         @pytest.mark.parametrize("count, expected_repr", [
261: (8)                           (1, "NpzFile {fname!r} with keys: arr_0"),
262: (8)                           (5, "NpzFile {fname!r} with keys: arr_0, arr_1, arr_2, arr_3, arr_4"),
263: (8)                           (6, "NpzFile {fname!r} with keys: arr_0, arr_1, arr_2, arr_3,
arr_4..."),
264: (4)                         ])
265: (4)                         def test_repr_lists_keys(self, count, expected_repr):
266: (8)                           a = np.array([[1, 2], [3, 4]], float)
267: (8)                           with temppath(suffix='.npz') as tmp:
268: (12)                             np.savez(tmp, *[a]*count)
269: (12)                             l = np.load(tmp)
270: (12)                             assert repr(l) == expected_repr.format(fname=tmp)
271: (12)                             l.close()
272: (0)                          class TestSaveTxt:
273: (4)                            def test_array(self):
274: (8)                              a = np.array([[1, 2], [3, 4]], float)
275: (8)                              fmt = "%.18e"
276: (8)                              c = BytesIO()
277: (8)                              np.savetxt(c, a, fmt=fmt)
278: (8)                              c.seek(0)
279: (8)                              assert_equal(c.readlines(),
280: (21)                                [asbytes((fmt + ' ' + fmt + '\n') % (1, 2)),
281: (22)                                asbytes((fmt + ' ' + fmt + '\n') % (3, 4))])
282: (8)                              a = np.array([[1, 2], [3, 4]], int)
283: (8)                              c = BytesIO()
284: (8)                              np.savetxt(c, a, fmt='%d')
285: (8)                              c.seek(0)
286: (8)                              assert_equal(c.readlines(), [b'1 2\n', b'3 4\n'])
287: (4)                            def test_1D(self):
288: (8)                              a = np.array([1, 2, 3, 4], int)
289: (8)                              c = BytesIO()
290: (8)                              np.savetxt(c, a, fmt='%d')
291: (8)                              c.seek(0)
292: (8)                              lines = c.readlines()
293: (8)                              assert_equal(lines, [b'1\n', b'2\n', b'3\n', b'4\n'])
294: (4)                            def test_0D_3D(self):
295: (8)                              c = BytesIO()
296: (8)                              assert_raises(ValueError, np.savetxt, c, np.array(1))
297: (8)                              assert_raises(ValueError, np.savetxt, c, np.array([[1], [2]])))
298: (4)                            def test_structured(self):
299: (8)                              a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
300: (8)                              c = BytesIO()
301: (8)                              np.savetxt(c, a, fmt='%d')
302: (8)                              c.seek(0)

```

```

303: (8)             assert_equal(c.readlines(), [b'1 2\n', b'3 4\n'])
304: (4)             def test_structured_padded(self):
305: (8)                 a = np.array([(1, 2, 3), (4, 5, 6)], dtype=[
306: (12)                   ('foo', 'i4'), ('bar', 'i4'), ('baz', 'i4')
307: (8)               ])
308: (8)               c = BytesIO()
309: (8)               np.savetxt(c, a[['foo', 'baz']], fmt='%d')
310: (8)               c.seek(0)
311: (8)               assert_equal(c.readlines(), [b'1 3\n', b'4 6\n'])
312: (4)             def test_multifield_view(self):
313: (8)                 a = np.ones(1, dtype=[('x', 'i4'), ('y', 'i4'), ('z', 'f4')])
314: (8)                 v = a[['x', 'z']]
315: (8)                 with temppath(suffix='.npy') as path:
316: (12)                     path = Path(path)
317: (12)                     np.save(path, v)
318: (12)                     data = np.load(path)
319: (12)                     assert_array_equal(data, v)
320: (4)             def test_delimiter(self):
321: (8)                 a = np.array([[1., 2.], [3., 4.]])
322: (8)                 c = BytesIO()
323: (8)                 np.savetxt(c, a, delimiter=',', fmt='%d')
324: (8)                 c.seek(0)
325: (8)                 assert_equal(c.readlines(), [b'1,2\n', b'3,4\n'])
326: (4)             def test_format(self):
327: (8)                 a = np.array([(1, 2), (3, 4)])
328: (8)                 c = BytesIO()
329: (8)                 np.savetxt(c, a, fmt=['%02d', '%3.1f'])
330: (8)                 c.seek(0)
331: (8)                 assert_equal(c.readlines(), [b'01 2.0\n', b'03 4.0\n'])
332: (8)                 c = BytesIO()
333: (8)                 np.savetxt(c, a, fmt='%02d : %3.1f')
334: (8)                 c.seek(0)
335: (8)                 lines = c.readlines()
336: (8)                 assert_equal(lines, [b'01 : 2.0\n', b'03 : 4.0\n'])
337: (8)                 c = BytesIO()
338: (8)                 np.savetxt(c, a, fmt='%02d : %3.1f', delimiter=',')
339: (8)                 c.seek(0)
340: (8)                 lines = c.readlines()
341: (8)                 assert_equal(lines, [b'01 : 2.0\n', b'03 : 4.0\n'])
342: (8)                 c = BytesIO()
343: (8)                 assert_raises(ValueError, np.savetxt, c, a, fmt=99)
344: (4)             def test_header_footer(self):
345: (8)                 c = BytesIO()
346: (8)                 a = np.array([(1, 2), (3, 4)], dtype=int)
347: (8)                 test_header_footer = 'Test header / footer'
348: (8)                 np.savetxt(c, a, fmt='%1d', header=test_header_footer)
349: (8)                 c.seek(0)
350: (8)                 assert_equal(c.read(),
351: (21)                     asbytes('# ' + test_header_footer + '\n1 2\n3 4\n'))
352: (8)                 c = BytesIO()
353: (8)                 np.savetxt(c, a, fmt='%1d', footer=test_header_footer)
354: (8)                 c.seek(0)
355: (8)                 assert_equal(c.read(),
356: (21)                     asbytes('1 2\n3 4\n# ' + test_header_footer + '\n'))
357: (8)                 c = BytesIO()
358: (8)                 commentstr = '% '
359: (8)                 np.savetxt(c, a, fmt='%1d',
360: (19)                     header=test_header_footer, comments=commentstr)
361: (8)                 c.seek(0)
362: (8)                 assert_equal(c.read(),
363: (21)                     asbytes(commentstr + test_header_footer + '\n' + '1 2\n3
4\n'))
364: (8)                 c = BytesIO()
365: (8)                 commentstr = '% '
366: (8)                 np.savetxt(c, a, fmt='%1d',
367: (19)                     footer=test_header_footer, comments=commentstr)
368: (8)                 c.seek(0)
369: (8)                 assert_equal(c.read(),
370: (21)                     asbytes('1 2\n3 4\n' + commentstr + test_header_footer +

```

```

'\\n')))
371: (4)     def test_file_roundtrip(self):
372: (8)         with temppath() as name:
373: (12)             a = np.array([(1, 2), (3, 4)])
374: (12)             np.savetxt(name, a)
375: (12)             b = np.loadtxt(name)
376: (12)             assert_array_equal(a, b)
377: (4)     def test_complex_arrays(self):
378: (8)         ncols = 2
379: (8)         nrows = 2
380: (8)         a = np.zeros((ncols, nrows), dtype=np.complex128)
381: (8)         re = np.pi
382: (8)         im = np.e
383: (8)         a[:] = re + 1.0j * im
384: (8)         c = BytesIO()
385: (8)         np.savetxt(c, a, fmt=' %+.3e')
386: (8)         c.seek(0)
387: (8)         lines = c.readlines()
388: (8)         assert_equal(
389: (12)             lines,
390: (12)             [b' ( +3.142e+00+ +2.718e+00j) ( +3.142e+00+ +2.718e+00j)\\n',
391: (13)                 b' ( +3.142e+00+ +2.718e+00j) ( +3.142e+00+ +2.718e+00j)\\n'])
392: (8)         c = BytesIO()
393: (8)         np.savetxt(c, a, fmt=' %+.3e' * 2 * ncols)
394: (8)         c.seek(0)
395: (8)         lines = c.readlines()
396: (8)         assert_equal(
397: (12)             lines,
398: (12)             [b' +3.142e+00 +2.718e+00 +3.142e+00 +2.718e+00\\n',
399: (13)                 b' +3.142e+00 +2.718e+00 +3.142e+00 +2.718e+00\\n'])
400: (8)         c = BytesIO()
401: (8)         np.savetxt(c, a, fmt=['(%+.3e%+.3ej)' * ncols])
402: (8)         c.seek(0)
403: (8)         lines = c.readlines()
404: (8)         assert_equal(
405: (12)             lines,
406: (12)             [b'(3.142e+00+2.718e+00j) (3.142e+00+2.718e+00j)\\n',
407: (13)                 b'(3.142e+00+2.718e+00j) (3.142e+00+2.718e+00j)\\n'])
408: (4)     def test_complex_negative_exponent(self):
409: (8)         ncols = 2
410: (8)         nrows = 2
411: (8)         a = np.zeros((ncols, nrows), dtype=np.complex128)
412: (8)         re = np.pi
413: (8)         im = np.e
414: (8)         a[:] = re - 1.0j * im
415: (8)         c = BytesIO()
416: (8)         np.savetxt(c, a, fmt='%.3e')
417: (8)         c.seek(0)
418: (8)         lines = c.readlines()
419: (8)         assert_equal(
420: (12)             lines,
421: (12)             [b' (3.142e+00-2.718e+00j) (3.142e+00-2.718e+00j)\\n',
422: (13)                 b' (3.142e+00-2.718e+00j) (3.142e+00-2.718e+00j)\\n'])
423: (4)     def test_custom_writer(self):
424: (8)         class CustomWriter(list):
425: (12)             def write(self, text):
426: (16)                 self.extend(text.split(b'\\n'))
427: (8)         w = CustomWriter()
428: (8)         a = np.array([(1, 2), (3, 4)])
429: (8)         np.savetxt(w, a)
430: (8)         b = np.loadtxt(w)
431: (8)         assert_array_equal(a, b)
432: (4)     def test_unicode(self):
433: (8)         utf8 = b'\\xcf\\x96'.decode('UTF-8')
434: (8)         a = np.array([utf8], dtype=np.str_)
435: (8)         with tempdir() as tmpdir:
436: (12)             np.savetxt(os.path.join(tmpdir, 'test.csv'), a, fmt=['%s'],
437: (23)                 encoding='UTF-8')
438: (4)     def test_unicode_roundtrip(self):

```

```

439: (8)             utf8 = b'\xcf\x96'.decode('UTF-8')
440: (8)             a = np.array([utf8], dtype=np.str_)
441: (8)             suffixes = ['', '.gz']
442: (8)             if HAS_BZ2:
443: (12)                suffixes.append('.bz2')
444: (8)             if HAS_LZMA:
445: (12)                suffixes.extend(['.xz', '.lzma'])
446: (8)             with tempdir() as tmpdir:
447: (12)                 for suffix in suffixes:
448: (16)                     np.savetxt(os.path.join(tmpdir, 'test.csv' + suffix), a,
449: (27)                         fmt=['%s'], encoding='UTF-16-LE')
450: (16)                     b = np.loadtxt(os.path.join(tmpdir, 'test.csv' + suffix),
451: (31)                         encoding='UTF-16-LE', dtype=np.str_)
452: (16)                     assert_array_equal(a, b)
453: (4)             def test_unicode_bytestream(self):
454: (8)                 utf8 = b'\xcf\x96'.decode('UTF-8')
455: (8)                 a = np.array([utf8], dtype=np.str_)
456: (8)                 s = BytesIO()
457: (8)                 np.savetxt(s, a, fmt=['%s'], encoding='UTF-8')
458: (8)                 s.seek(0)
459: (8)                 assert_equal(s.read().decode('UTF-8'), utf8 + '\n')
460: (4)             def test_unicode_stringstream(self):
461: (8)                 utf8 = b'\xcf\x96'.decode('UTF-8')
462: (8)                 a = np.array([utf8], dtype=np.str_)
463: (8)                 s = StringIO()
464: (8)                 np.savetxt(s, a, fmt=['%s'], encoding='UTF-8')
465: (8)                 s.seek(0)
466: (8)                 assert_equal(s.read(), utf8 + '\n')
467: (4)             @pytest.mark.parametrize("fmt", ["%f", b"%f"])
468: (4)             @pytest.mark.parametrize("iotype", [StringIO, BytesIO])
469: (4)             def test_unicode_and_bytes_fmt(self, fmt, iotype):
470: (8)                 a = np.array([1.])
471: (8)                 s = iotype()
472: (8)                 np.savetxt(s, a, fmt=fmt)
473: (8)                 s.seek(0)
474: (8)                 if iotype is StringIO:
475: (12)                     assert_equal(s.read(), "%f\n" % 1.)
476: (8)                 else:
477: (12)                     assert_equal(s.read(), b"%f\n" % 1.)
478: (4)             @pytest.mark.skipif(sys.platform=='win32', reason="files>4GB may not
work")
479: (4)             @pytest.mark.slow
480: (4)             @requires_memory(free_bytes=7e9)
481: (4)             def test_large_zip(self):
482: (8)                 def check_large_zip(memoryerror_raised):
483: (12)                     memoryerror_raised.value = False
484: (12)                     try:
485: (16)                         test_data = np.asarray([np.random.rand(
486: (40)                             np.random.randint(50,100),4)
487: (40)                             for i in range(800000)], dtype=object)
488: (16)
489: (20)                     with tempdir() as tmpdir:
490: (29)                         np.savez(os.path.join(tmpdir, 'test.npz'),
491: (12)                             test_data=test_data)
492: (16)                     except MemoryError:
493: (16)                         memoryerror_raised.value = True
494: (16)                         raise
495: (8)                     memoryerror_raised = Value(c_bool)
496: (8)                     ctx = get_context('fork')
497: (8)                     p = ctx.Process(target=check_large_zip, args=(memoryerror_raised,))
498: (8)                     p.start()
499: (8)                     p.join()
500: (8)                     if memoryerror_raised.value:
501: (12)                         raise MemoryError("Child process raised a MemoryError exception")
502: (8)                     if p.exitcode == -9:
503: (12)                         pytest.xfail("subprocess got a SIGKILL, apparently free memory was
not sufficient")
504: (8)                         assert p.exitcode == 0
505: (0)             class LoadTxtBase:
506: (4)                 def check_compressed(self, fopen, suffixes):

```

```

506: (8)           wanted = np.arange(6).reshape((2, 3))
507: (8)           linesep = ('\n', '\r\n', '\r')
508: (8)           for sep in linesep:
509: (12)             data = '0 1 2' + sep + '3 4 5'
510: (12)             for suffix in suffixes:
511: (16)               with temppath(suffix=suffix) as name:
512: (20)                 with fopen(name, mode='wt', encoding='UTF-32-LE') as f:
513: (24)                   f.write(data)
514: (20)                 res = self.loadfunc(name, encoding='UTF-32-LE')
515: (20)                 assert_array_equal(res, wanted)
516: (20)                 with fopen(name, "rt", encoding='UTF-32-LE') as f:
517: (24)                   res = self.loadfunc(f)
518: (20)                   assert_array_equal(res, wanted)
519: (4)           def test_compressed_gzip(self):
520: (8)             self.check_compressed(gzip.open, ('.gz',))
521: (4)             @pytest.mark.skipif(not HAS_BZ2, reason="Needs bz2")
522: (4)           def test_compressed_bz2(self):
523: (8)             self.check_compressed(bz2.open, ('.bz2',))
524: (4)             @pytest.mark.skipif(not HAS_LZMA, reason="Needs lzma")
525: (4)           def test_compressed_lzma(self):
526: (8)             self.check_compressed(lzma.open, ('.xz', '.lzma'))
527: (4)           def test_encoding(self):
528: (8)             with temppath() as path:
529: (12)               with open(path, "wb") as f:
530: (16)                 f.write('0.\n1.\n2.'.encode("UTF-16"))
531: (12)               x = self.loadfunc(path, encoding="UTF-16")
532: (12)               assert_array_equal(x, [0., 1., 2.])
533: (4)           def test_stringload(self):
534: (8)             nonascii = b'\xc3\xb6\xc3\xbc\xc3\xb6'.decode("UTF-8")
535: (8)             with temppath() as path:
536: (12)               with open(path, "wb") as f:
537: (16)                 f.write(nonascii.encode("UTF-16"))
538: (12)               x = self.loadfunc(path, encoding="UTF-16", dtype=np.str_)
539: (12)               assert_array_equal(x, nonascii)
540: (4)           def test_binary_decode(self):
541: (8)             utf16 = b'\xff\xfe\x04\x00\x04\x00j\x04'
542: (8)             v = self.loadfunc(BytesIO(utf16), dtype=np.str_, encoding='UTF-16')
543: (8)             assert_array_equal(v, np.array(utf16.decode('UTF-16').split()))
544: (4)           def test_converters_decode(self):
545: (8)             c = TextIO()
546: (8)             c.write(b'\xcf\x96')
547: (8)             c.seek(0)
548: (8)             x = self.loadfunc(c, dtype=np.str_,
549: (26)                           converters={0: lambda x: x.decode('UTF-8')})
550: (8)             a = np.array([b'\xcf\x96'.decode('UTF-8')])
551: (8)             assert_array_equal(x, a)
552: (4)           def test_converters_nodecode(self):
553: (8)             utf8 = b'\xcf\x96'.decode('UTF-8')
554: (8)             with temppath() as path:
555: (12)               with io.open(path, 'wt', encoding='UTF-8') as f:
556: (16)                 f.write(utf8)
557: (12)               x = self.loadfunc(path, dtype=np.str_,
558: (30)                           converters={0: lambda x: x + 't'},
559: (30)                           encoding='UTF-8')
560: (12)               a = np.array([utf8 + 't'])
561: (12)               assert_array_equal(x, a)
562: (0)           class TestLoadTxt(LoadTxtBase):
563: (4)             loadfunc = staticmethod(np.loadtxt)
564: (4)             def setup_method(self):
565: (8)               self.orig_chunk = np.lib.npyio._loadtxt_chunksize
566: (8)               np.lib.npyio._loadtxt_chunksize = 1
567: (4)             def teardown_method(self):
568: (8)               np.lib.npyio._loadtxt_chunksize = self.orig_chunk
569: (4)             def test_record(self):
570: (8)               c = TextIO()
571: (8)               c.write('1 2\n3 4')
572: (8)               c.seek(0)
573: (8)               x = np.loadtxt(c, dtype=[('x', np.int32), ('y', np.int32)])
574: (8)               a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
```

```

575: (8)             assert_array_equal(x, a)
576: (8)             d = TextIO()
577: (8)             d.write('M 64 75.0\nF 25 60.0')
578: (8)             d.seek(0)
579: (8)             mydescriptor = {'names': ('gender', 'age', 'weight'),
580: (24)                     'formats': ('S1', 'i4', 'f4')}
581: (8)             b = np.array([('M', 64.0, 75.0),
582: (22)                         ('F', 25.0, 60.0)], dtype=mydescriptor)
583: (8)             y = np.loadtxt(d, dtype=mydescriptor)
584: (8)             assert_array_equal(y, b)
585: (4)             def test_array(self):
586: (8)                 c = TextIO()
587: (8)                 c.write('1 2\n3 4')
588: (8)                 c.seek(0)
589: (8)                 x = np.loadtxt(c, dtype=int)
590: (8)                 a = np.array([[1, 2], [3, 4]], int)
591: (8)                 assert_array_equal(x, a)
592: (8)                 c.seek(0)
593: (8)                 x = np.loadtxt(c, dtype=float)
594: (8)                 a = np.array([[1, 2], [3, 4]], float)
595: (8)                 assert_array_equal(x, a)
596: (4)             def test_1D(self):
597: (8)                 c = TextIO()
598: (8)                 c.write('1\n2\n3\n4\n')
599: (8)                 c.seek(0)
600: (8)                 x = np.loadtxt(c, dtype=int)
601: (8)                 a = np.array([1, 2, 3, 4], int)
602: (8)                 assert_array_equal(x, a)
603: (8)                 c = TextIO()
604: (8)                 c.write('1,2,3,4\n')
605: (8)                 c.seek(0)
606: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',')
607: (8)                 a = np.array([1, 2, 3, 4], int)
608: (8)                 assert_array_equal(x, a)
609: (4)             def test_missing(self):
610: (8)                 c = TextIO()
611: (8)                 c.write('1,2,3,,5\n')
612: (8)                 c.seek(0)
613: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',',
614: (23)                         converters={3: lambda s: int(s or - 999)})
615: (8)                 a = np.array([1, 2, 3, -999, 5], int)
616: (8)                 assert_array_equal(x, a)
617: (4)             def test_converters_with_usecols(self):
618: (8)                 c = TextIO()
619: (8)                 c.write('1,2,3,,5\n6,7,8,9,10\n')
620: (8)                 c.seek(0)
621: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',',
622: (23)                         converters={3: lambda s: int(s or - 999)},
623: (23)                         usecols=(1, 3,))
624: (8)                 a = np.array([[2, -999], [7, 9]], int)
625: (8)                 assert_array_equal(x, a)
626: (4)             def test_comments_unicode(self):
627: (8)                 c = TextIO()
628: (8)                 c.write('# comment\n1,2,3,5\n')
629: (8)                 c.seek(0)
630: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',',
631: (23)                         comments='#')
632: (8)                 a = np.array([1, 2, 3, 5], int)
633: (8)                 assert_array_equal(x, a)
634: (4)             def test_comments_byte(self):
635: (8)                 c = TextIO()
636: (8)                 c.write('# comment\n1,2,3,5\n')
637: (8)                 c.seek(0)
638: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',',
639: (23)                         comments=b'#')
640: (8)                 a = np.array([1, 2, 3, 5], int)
641: (8)                 assert_array_equal(x, a)
642: (4)             def test_comments_multiple(self):
643: (8)                 c = TextIO()

```

```

644: (8)           c.write('# comment\n1,2,3\n@ comment2\n4,5,6 // comment3')
645: (8)           c.seek(0)
646: (8)           x = np.loadtxt(c, dtype=int, delimiter=',',
647: (23)             comments=['#', '@', '//'])
648: (8)           a = np.array([[1, 2, 3], [4, 5, 6]], int)
649: (8)           assert_array_equal(x, a)
650: (4)           @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
651: (24)             reason="PyPy bug in error formatting")
652: (4)           def test_comments_multi_chars(self):
653: (8)             c = TextIO()
654: (8)             c.write('/* comment\n1,2,3,5\n')
655: (8)             c.seek(0)
656: (8)             x = np.loadtxt(c, dtype=int, delimiter=',',
657: (23)               comments='/*')
658: (8)             a = np.array([1, 2, 3, 5], int)
659: (8)             assert_array_equal(x, a)
660: (8)             c = TextIO()
661: (8)             c.write('/* comment\n1,2,3,5\n')
662: (8)             c.seek(0)
663: (8)             assert_raises(ValueError, np.loadtxt, c, dtype=int, delimiter=',',
664: (22)               comments='/*')
665: (4)           def test_skiprows(self):
666: (8)             c = TextIO()
667: (8)             c.write('comment\n1,2,3,5\n')
668: (8)             c.seek(0)
669: (8)             x = np.loadtxt(c, dtype=int, delimiter=',',
670: (23)               skiprows=1)
671: (8)             a = np.array([1, 2, 3, 5], int)
672: (8)             assert_array_equal(x, a)
673: (8)             c = TextIO()
674: (8)             c.write('# comment\n1,2,3,5\n')
675: (8)             c.seek(0)
676: (8)             x = np.loadtxt(c, dtype=int, delimiter=',',
677: (23)               skiprows=1)
678: (8)             a = np.array([1, 2, 3, 5], int)
679: (8)             assert_array_equal(x, a)
680: (4)           def test_usecols(self):
681: (8)             a = np.array([[1, 2], [3, 4]], float)
682: (8)             c = BytesIO()
683: (8)             np.savetxt(c, a)
684: (8)             c.seek(0)
685: (8)             x = np.loadtxt(c, dtype=float, usecols=(1,))
686: (8)             assert_array_equal(x, a[:, 1])
687: (8)             a = np.array([[1, 2, 3], [3, 4, 5]], float)
688: (8)             c = BytesIO()
689: (8)             np.savetxt(c, a)
690: (8)             c.seek(0)
691: (8)             x = np.loadtxt(c, dtype=float, usecols=(1, 2))
692: (8)             assert_array_equal(x, a[:, 1:])
693: (8)             c.seek(0)
694: (8)             x = np.loadtxt(c, dtype=float, usecols=np.array([1, 2]))
695: (8)             assert_array_equal(x, a[:, 1:])
696: (8)             for int_type in [int, np.int8, np.int16,
697: (25)               np.int32, np.int64, np.uint8, np.uint16,
698: (25)               np.uint32, np.uint64]:
699: (12)               to_read = int_type(1)
700: (12)               c.seek(0)
701: (12)               x = np.loadtxt(c, dtype=float, usecols=to_read)
702: (12)               assert_array_equal(x, a[:, 1])
703: (8)             class CrazyInt:
704: (12)               def __index__(self):
705: (16)                 return 1
706: (8)               crazy_int = CrazyInt()
707: (8)               c.seek(0)
708: (8)               x = np.loadtxt(c, dtype=float, usecols=crazy_int)
709: (8)               assert_array_equal(x, a[:, 1])
710: (8)               c.seek(0)
711: (8)               x = np.loadtxt(c, dtype=float, usecols=(crazy_int,))
712: (8)               assert_array_equal(x, a[:, 1])

```

```

713: (8)           data = '''JOE 70.1 25.3
714: (16)          BOB 60.5 27.9
715: (16)
716: (8)          ''
717: (8)          c = TextIO(data)
718: (8)          names = ['stid', 'temp']
719: (8)          dtypes = ['S4', 'f8']
720: (8)          arr = np.loadtxt(c, usecols=(0, 2), dtype=list(zip(names, dtypes)))
721: (8)          assert_equal(arr['stid'], [b"JOE", b"BOB"])
722: (8)          assert_equal(arr['temp'], [25.3, 27.9])
723: (8)          c.seek(0)
724: (8)          bogus_idx = 1.5
725: (12)         assert_raises_regex(
726: (12)             TypeError,
727: (12)             '^usecols must be.*%s' % type(bogus_idx).__name__,
728: (12)             np.loadtxt, c, usecols=bogus_idx
729: (12)
730: (12)         assert_raises_regex(
731: (12)             TypeError,
732: (12)             '^usecols must be.*%s' % type(bogus_idx).__name__,
733: (12)             np.loadtxt, c, usecols=[0, bogus_idx, 0]
734: (12)
735: (4)          def test_bad_usecols(self):
736: (8)          with pytest.raises(OverflowError):
737: (12)              np.loadtxt(["1\n"], usecols=[2**64], delimiter=",")
738: (8)          with pytest.raises((ValueError, OverflowError)):
739: (12)              np.loadtxt(["1\n"], usecols=[2**62], delimiter=",")
740: (8)          with pytest.raises(TypeError,
741: (16)              match="If a structured dtype .*. But 1 usecols were given and
742: (22)                  "the number of fields is 3."):
743: (12)                  np.loadtxt(["1,1\n"], dtype="i,(2)i", usecols=[0], delimiter=",")
744: (4)          def test_fancy_dtype(self):
745: (8)          c = TextIO()
746: (8)          c.write('1,2,3.0\n4,5,6.0\n')
747: (8)          c.seek(0)
748: (8)          dt = np.dtype([('x', int), ('y', [(('t', int), ('s', float))])])
749: (8)          x = np.loadtxt(c, dtype=dt, delimiter=',')
750: (8)          a = np.array([(1, (2, 3.0)), (4, (5, 6.0))], dt)
751: (8)          assert_array_equal(x, a)
752: (4)          def test_shaped_dtype(self):
753: (8)          c = TextIO("aaaa 1.0 8.0 1 2 3 4 5 6")
754: (23)         dt = np.dtype([('name', 'S4'), ('x', float), ('y', float),
755: (8)                      ('block', int, (2, 3))])
756: (8)          x = np.loadtxt(c, dtype=dt)
757: (21)         a = np.array([('aaaa', 1.0, 8.0, [[1, 2, 3], [4, 5, 6]])], dtype=dt)
758: (8)          assert_array_equal(x, a)
759: (4)          def test_3d_shaped_dtype(self):
760: (8)          c = TextIO("aaaa 1.0 8.0 1 2 3 4 5 6 7 8 9 10 11 12")
761: (8)          dt = np.dtype([('name', 'S4'), ('x', float), ('y', float),
762: (23)                      ('block', int, (2, 2, 3))])
763: (8)          x = np.loadtxt(c, dtype=dt)
764: (8)          a = np.array([('aaaa', 1.0, 8.0,
765: (23)                          [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]]], dtype=dt)
766: (21)         assert_array_equal(x, a)
767: (8)          def test_str_dtype(self):
768: (4)          c = ["str1", "str2"]
769: (8)          for dt in (str, np.bytes_):
770: (8)              a = np.array(["str1", "str2"], dtype=dt)
771: (12)              x = np.loadtxt(c, dtype=dt)
772: (12)              assert_array_equal(x, a)
773: (12)
774: (4)          def test_empty_file(self):
775: (8)          with pytest.warns(UserWarning, match="input contained no data"):
776: (12)              c = TextIO()
777: (12)              x = np.loadtxt(c)
778: (12)              assert_equal(x.shape, (0,))
779: (12)              x = np.loadtxt(c, dtype=np.int64)
780: (12)              assert_equal(x.shape, (0,))
```

```

781: (12)                                assert_(x.dtype == np.int64)
782: (4)       def test_unused_converter(self):
783: (8)           c = TextIO()
784: (8)           c.writelines(['1 21\n', '3 42\n'])
785: (8)           c.seek(0)
786: (8)           data = np.loadtxt(c, usecols=(1,),
787: (26)                           converters={0: lambda s: int(s, 16)})
788: (8)           assert_array_equal(data, [21, 42])
789: (8)           c.seek(0)
790: (8)           data = np.loadtxt(c, usecols=(1,),
791: (26)                           converters={1: lambda s: int(s, 16)})
792: (8)           assert_array_equal(data, [33, 66])
793: (4)       def test_dtype_with_object(self):
794: (8)           data = """ 1; 2001-01-01
795: (19)               2; 2002-01-31 """
796: (8)           ndtype = [('idx', int), ('code', object)]
797: (8)           func = lambda s: strftime(s.strip(), "%Y-%m-%d")
798: (8)           converters = {1: func}
799: (8)           test = np.loadtxt(TextIO(data), delimiter=";", dtype=ndtype,
800: (26)                           converters=converters)
801: (8)           control = np.array(
802: (12)               [(1, datetime(2001, 1, 1)), (2, datetime(2002, 1, 31))],
803: (12)               dtype=ndtype)
804: (8)           assert_equal(test, control)
805: (4)       def test_uint64_type(self):
806: (8)           tgt = (9223372043271415339, 9223372043271415853)
807: (8)           c = TextIO()
808: (8)           c.write("%s %s" % tgt)
809: (8)           c.seek(0)
810: (8)           res = np.loadtxt(c, dtype=np.uint64)
811: (8)           assert_equal(res, tgt)
812: (4)       def test_int64_type(self):
813: (8)           tgt = (-9223372036854775807, 9223372036854775807)
814: (8)           c = TextIO()
815: (8)           c.write("%s %s" % tgt)
816: (8)           c.seek(0)
817: (8)           res = np.loadtxt(c, dtype=np.int64)
818: (8)           assert_equal(res, tgt)
819: (4)       def test_from_float_hex(self):
820: (8)           tgt = np.logspace(-10, 10, 5).astype(np.float32)
821: (8)           tgt = np.hstack((tgt, -tgt)).astype(float)
822: (8)           inp = '\n'.join(map(float.hex, tgt))
823: (8)           c = TextIO()
824: (8)           c.write(inp)
825: (8)           for dt in [float, np.float32]:
826: (12)               c.seek(0)
827: (12)               res = np.loadtxt(
828: (16)                   c, dtype=dt, converters=float.fromhex, encoding="latin1")
829: (12)               assert_equal(res, tgt, err_msg="%s" % dt)
830: (4) @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
831: (24)                           reason="PyPy bug in error formatting")
832: (4)       def test_default_float_converter_no_default_hex_conversion(self):
833: (8)           """
834: (8)               Ensure that fromhex is only used for values with the correct prefix
835: (8)               and
836: (8)               is not called by default. Regression test related to gh-19598.
837: (8)           """
838: (8)           c = TextIO("a b c")
839: (16)           with pytest.raises(ValueError,
840: (12)                           match=".*convert string 'a' to float64 at row 0, column 1"):
841: (4)               np.loadtxt(c)
842: (24) @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
843: (4)                           reason="PyPy bug in error formatting")
844: (8)           def test_default_float_converter_exception(self):
845: (8)               """
846: (8)               Ensure that the exception message raised during failed floating point
847: (8)               conversion is correct. Regression test related to gh-19598.
848: (8)               """
849: (8)           c = TextIO("qrs tuv") # Invalid values for default float converter

```

```

849: (8)             with pytest.raises(ValueError,
850: (16)                 match="could not convert string 'qrs' to float64"):
851: (12)                 np.loadtxt(c)
852: (4)             def test_from_complex(self):
853: (8)                 tgt = (complex(1, 1), complex(1, -1))
854: (8)                 c = TextIO()
855: (8)                 c.write("%s %s" % tgt)
856: (8)                 c.seek(0)
857: (8)                 res = np.loadtxt(c, dtype=complex)
858: (8)                 assert_equal(res, tgt)
859: (4)             def test_complex_misformatted(self):
860: (8)                 a = np.zeros((2, 2), dtype=np.complex128)
861: (8)                 re = np.pi
862: (8)                 im = np.e
863: (8)                 a[:] = re - 1.0j * im
864: (8)                 c = BytesIO()
865: (8)                 np.savetxt(c, a, fmt='%.16e')
866: (8)                 c.seek(0)
867: (8)                 txt = c.read()
868: (8)                 c.seek(0)
869: (8)                 txt_bad = txt.replace(b'e+00-', b'e00+-')
870: (8)                 assert_(txt_bad != txt)
871: (8)                 c.write(txt_bad)
872: (8)                 c.seek(0)
873: (8)                 res = np.loadtxt(c, dtype=complex)
874: (8)                 assert_equal(res, a)
875: (4)             def test_universal_newline(self):
876: (8)                 with temppath() as name:
877: (12)                     with open(name, 'w') as f:
878: (16)                         f.write('1 21\r3 42\r')
879: (12)                     data = np.loadtxt(name)
880: (8)                     assert_array_equal(data, [[1, 21], [3, 42]])
881: (4)             def test_empty_field_after_tab(self):
882: (8)                 c = TextIO()
883: (8)                 c.write('1 \t2 \t3\tstart \n4\t5\t6\t \n7\t8\t9.5\t')
884: (8)                 c.seek(0)
885: (8)                 dt = {'names': ('x', 'y', 'z', 'comment'),
886: (14)                     'formats': ('<i4', '<i4', '<f4', '|S8')}
887: (8)                 x = np.loadtxt(c, dtype=dt, delimiter='\t')
888: (8)                 a = np.array([b'start ', b' ', b''])
889: (8)                 assert_array_equal(x['comment'], a)
890: (4)             def test_unpack_structured(self):
891: (8)                 txt = TextIO("M 21 72\nF 35 58")
892: (8)                 dt = {'names': ('a', 'b', 'c'), 'formats': ('|S1', '<i4', '<f4')}
893: (8)                 a, b, c = np.loadtxt(txt, dtype=dt, unpack=True)
894: (8)                 assert_(a.dtype.str == '|S1')
895: (8)                 assert_(b.dtype.str == '<i4')
896: (8)                 assert_(c.dtype.str == '<f4')
897: (8)                 assert_array_equal(a, np.array([b'M', b'F']))
898: (8)                 assert_array_equal(b, np.array([21, 35]))
899: (8)                 assert_array_equal(c, np.array([72., 58.]))
900: (4)             def test_ndmin_keyword(self):
901: (8)                 c = TextIO()
902: (8)                 c.write('1,2,3\n4,5,6')
903: (8)                 c.seek(0)
904: (8)                 assert_raises(ValueError, np.loadtxt, c, ndmin=3)
905: (8)                 c.seek(0)
906: (8)                 assert_raises(ValueError, np.loadtxt, c, ndmin=1.5)
907: (8)                 c.seek(0)
908: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',', ndmin=1)
909: (8)                 a = np.array([[1, 2, 3], [4, 5, 6]])
910: (8)                 assert_array_equal(x, a)
911: (8)                 d = TextIO()
912: (8)                 d.write('0,1,2')
913: (8)                 d.seek(0)
914: (8)                 x = np.loadtxt(d, dtype=int, delimiter=',', ndmin=2)
915: (8)                 assert_(x.shape == (1, 3))
916: (8)                 d.seek(0)
917: (8)                 x = np.loadtxt(d, dtype=int, delimiter=',', ndmin=1)

```

```

918: (8)             assert_(x.shape == (3,))
919: (8)             d.seek(0)
920: (8)             x = np.loadtxt(d, dtype=int, delimiter=',', ndmin=0)
921: (8)             assert_(x.shape == (3,))
922: (8)             e = TextIO()
923: (8)             e.write('0\n1\n2')
924: (8)             e.seek(0)
925: (8)             x = np.loadtxt(e, dtype=int, delimiter=',', ndmin=2)
926: (8)             assert_(x.shape == (3, 1))
927: (8)             e.seek(0)
928: (8)             x = np.loadtxt(e, dtype=int, delimiter=',', ndmin=1)
929: (8)             assert_(x.shape == (3,))
930: (8)             e.seek(0)
931: (8)             x = np.loadtxt(e, dtype=int, delimiter=',', ndmin=0)
932: (8)             assert_(x.shape == (3,))
933: (8)             with pytest.warns(UserWarning, match="input contained no data"):
934: (12)                 f = TextIO()
935: (12)                 assert_(np.loadtxt(f, ndmin=2).shape == (0, 1,))
936: (12)                 assert_(np.loadtxt(f, ndmin=1).shape == (0,))
937: (4)             def test_generator_source(self):
938: (8)                 def count():
939: (12)                     for i in range(10):
940: (16)                         yield "%d" % i
941: (8)             res = np.loadtxt(count())
942: (8)             assert_array_equal(res, np.arange(10))
943: (4)             def test_bad_line(self):
944: (8)                 c = TextIO()
945: (8)                 c.write('1 2 3\n4 5 6\n2 3')
946: (8)                 c.seek(0)
947: (8)                 assert_raises_regex(ValueError, "3", np.loadtxt, c)
948: (4)             def test_none_as_string(self):
949: (8)                 c = TextIO()
950: (8)                 c.write('100,foo,200\n300,None,400')
951: (8)                 c.seek(0)
952: (8)                 dt = np.dtype([('x', int), ('a', 'S10'), ('y', int)])
953: (8)                 np.loadtxt(c, delimiter=',', dtype=dt, comments=None) # Should
succeed
954: (4)             @pytest.mark.skipif(locale.getpreferredencoding() == 'ANSI_X3.4-1968',
955: (24)                           reason="Wrong preferred encoding")
956: (4)             def test_binary_load(self):
957: (8)                 butf8 = b"5,6,7,\xc3\x95scarscar\r\n15,2,3,hello\r\n"
958: (16)                 b"20,2,3,\xc3\x95scar\r\n"
959: (8)                 sutf8 = butf8.decode("UTF-8").replace("\r", "").splitlines()
960: (8)                 with temppath() as path:
961: (12)                     with open(path, "wb") as f:
962: (16)                         f.write(butf8)
963: (12)                     with open(path, "rb") as f:
964: (16)                         x = np.loadtxt(f, encoding="UTF-8", dtype=np.str_)
965: (12)                         assert_array_equal(x, sutf8)
966: (12)                     with open(path, "rb") as f:
967: (16)                         x = np.loadtxt(f, encoding="UTF-8", dtype="S")
968: (12)                         x = [b'5,6,7,\xc3\x95scarscar', b'15,2,3,hello',
b'20,2,3,\xc3\x95scar']
969: (12)                         assert_array_equal(x, np.array(x, dtype="S"))
970: (4)             def test_max_rows(self):
971: (8)                 c = TextIO()
972: (8)                 c.write('1,2,3,5\n4,5,7,8\n2,1,4,5')
973: (8)                 c.seek(0)
974: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',',
975: (23)                               max_rows=1)
976: (8)                 a = np.array([1, 2, 3, 5], int)
977: (8)                 assert_array_equal(x, a)
978: (4)             def test_max_rows_with_skiprows(self):
979: (8)                 c = TextIO()
980: (8)                 c.write('comments\n1,2,3,5\n4,5,7,8\n2,1,4,5')
981: (8)                 c.seek(0)
982: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',',
983: (23)                               skiprows=1, max_rows=1)
984: (8)                 a = np.array([1, 2, 3, 5], int)

```

```

985: (8)             assert_array_equal(x, a)
986: (8)             c = TextIO()
987: (8)             c.write('comment\n1,2,3,5\n4,5,7,8\n2,1,4,5')
988: (8)             c.seek(0)
989: (8)             x = np.loadtxt(c, dtype=int, delimiter=',',
990: (23)                 skiprows=1, max_rows=2)
991: (8)             a = np.array([[1, 2, 3, 5], [4, 5, 7, 8]], int)
992: (8)             assert_array_equal(x, a)
993: (4)             def test_max_rows_with_read_continuation(self):
994: (8)                 c = TextIO()
995: (8)                 c.write('1,2,3,5\n4,5,7,8\n2,1,4,5')
996: (8)                 c.seek(0)
997: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',',
998: (23)                     max_rows=2)
999: (8)                 a = np.array([[1, 2, 3, 5], [4, 5, 7, 8]], int)
1000: (8)                assert_array_equal(x, a)
1001: (8)                x = np.loadtxt(c, dtype=int, delimiter=',')
1002: (8)                a = np.array([2,1,4,5], int)
1003: (8)                assert_array_equal(x, a)
1004: (4)             def test_max_rows_larger(self):
1005: (8)                 c = TextIO()
1006: (8)                 c.write('comment\n1,2,3,5\n4,5,7,8\n2,1,4,5')
1007: (8)                 c.seek(0)
1008: (8)                 x = np.loadtxt(c, dtype=int, delimiter=',',
1009: (23)                     skiprows=1, max_rows=6)
1010: (8)                 a = np.array([[1, 2, 3, 5], [4, 5, 7, 8], [2, 1, 4, 5]], int)
1011: (8)                 assert_array_equal(x, a)
1012: (4)             @pytest.mark.parametrize(["skip", "data"], [
1013: (12)                 (1, ["ignored\n", "1,2\n", "\n", "3,4\n"]),
1014: (12)                 (1, ["ignored", "1,2", "", "3,4"]),
1015: (12)                 (1, StringIO("ignored\n1,2\n\n3,4")),
1016: (12)                 (0, ["-1,0\n", "1,2\n", "\n", "3,4\n"]),
1017: (12)                 (0, ["-1,0", "1,2", "", "3,4"]),
1018: (12)                 (0, StringIO("-1,0\n1,2\n\n3,4")))
1019: (4)             def test_max_rows_empty_lines(self, skip, data):
1020: (8)                 with pytest.warns(UserWarning,
1021: (20)                     match=f"Input line 3.*max_rows={3-skip}"):
1022: (12)                 res = np.loadtxt(data, dtype=int, skiprows=skip, delimiter=",",
1023: (29)                     max_rows=3-skip)
1024: (12)                 assert_array_equal(res, [[-1, 0], [1, 2], [3, 4]][skip:])
1025: (8)                 if isinstance(data, StringIO):
1026: (12)                     data.seek(0)
1027: (8)                     with warnings.catch_warnings():
1028: (12)                         warnings.simplefilter("error", UserWarning)
1029: (12)                         with pytest.raises(UserWarning):
1030: (16)                             np.loadtxt(data, dtype=int, skiprows=skip, delimiter=",",
1031: (27)                                 max_rows=3-skip)
1032: (0)             class Testfromregex:
1033: (4)                 def test_record(self):
1034: (8)                     c = TextIO()
1035: (8)                     c.write('1.312 foo\n1.534 bar\n4.444 qux')
1036: (8)                     c.seek(0)
1037: (8)                     dt = [('num', np.float64), ('val', 'S3')]
1038: (8)                     x = np.fromregex(c, r"([0-9.]+)\s+(...)", dt)
1039: (8)                     a = np.array([(1.312, 'foo'), (1.534, 'bar'), (4.444, 'qux')], dtype=dt)
1040: (21)                     assert_array_equal(x, a)
1041: (8)                 def test_record_2(self):
1042: (4)                     c = TextIO()
1043: (8)                     c.write('1312 foo\n1534 bar\n4444 qux')
1044: (8)                     c.seek(0)
1045: (8)                     dt = [('num', np.int32), ('val', 'S3')]
1046: (8)                     x = np.fromregex(c, r"(\d+)\s+(...)", dt)
1047: (8)                     a = np.array([(1312, 'foo'), (1534, 'bar'), (4444, 'qux')], dtype=dt)
1048: (8)                     assert_array_equal(x, a)
1049: (21)                 def test_record_3(self):
1050: (8)                     c = TextIO()
1051: (4)                     c.write('1312 foo\n1534 bar\n4444 qux')
1052: (8)
1053: (8)

```

```

1054: (8)             c.seek(0)
1055: (8)             dt = [('num', np.float64)]
1056: (8)             x = np.fromregex(c, r"(\d+)\s+...", dt)
1057: (8)             a = np.array([(1312,), (1534,), (4444,)], dtype=dt)
1058: (8)             assert_array_equal(x, a)
1059: (4)             @pytest.mark.parametrize("path_type", [str, Path])
1060: (4)             def test_record_unicode(self, path_type):
1061: (8)                 utf8 = b'\xcf\x96'
1062: (8)                 with temppath() as str_path:
1063: (12)                     path = path_type(str_path)
1064: (12)                     with open(path, 'wb') as f:
1065: (16)                         f.write(b'1.312 foo' + utf8 + b'\n1.534 bar\n4.444 qux')
1066: (12)                     dt = [('num', np.float64), ('val', 'U4')]
1067: (12)                     x = np.fromregex(path, r"(?u)[0-9.]+\s+(\w+)", dt,
encoding='UTF-8')
1068: (12)                     a = np.array([(1.312, 'foo' + utf8.decode('UTF-8')), (1.534,
1069: (27)                         'bar'),
1070: (12)                         (4.444, 'qux')], dtype=dt)
1071: (12)             assert_array_equal(x, a)
1072: (12)             regexp = re.compile(r"([0-9.]+\s+(\w+)", re.UNICODE)
1073: (12)             x = np.fromregex(path, regexp, dt, encoding='UTF-8')
1074: (12)             assert_array_equal(x, a)
1074: (4)             def test_compiled_bytes(self):
1075: (8)                 regexp = re.compile(b'(\d)')
1076: (8)                 c = BytesIO(b'123')
1077: (8)                 dt = [('num', np.float64)]
1078: (8)                 a = np.array([1, 2, 3], dtype=dt)
1079: (8)                 x = np.fromregex(c, regexp, dt)
1080: (8)                 assert_array_equal(x, a)
1081: (4)             def test_bad_dtype_not_structured(self):
1082: (8)                 regexp = re.compile(b'(\d)')
1083: (8)                 c = BytesIO(b'123')
1084: (8)                 with pytest.raises(TypeError, match='structured datatype'):
1085: (12)                     np.fromregex(c, regexp, dtype=np.float64)
1086: (0)             class TestFromTxt(LoadTxtBase):
1087: (4)                 loadfunc = staticmethod(np.genfromtxt)
1088: (4)                 def test_record(self):
1089: (8)                     data = TextIO('1 2\n3 4')
1090: (8)                     test = np.genfromtxt(data, dtype=[('x', np.int32), ('y', np.int32)])
1091: (8)                     control = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
1092: (8)                     assert_equal(test, control)
1093: (8)                     data = TextIO('M 64.0 75.0\nF 25.0 60.0')
1094: (8)                     descriptor = {'names': ('gender', 'age', 'weight'),
1095: (22)                         'formats': ('S1', 'i4', 'f4')}
1096: (8)                     control = np.array([('M', 64.0, 75.0), ('F', 25.0, 60.0)],
1097: (27)                         dtype=descriptor)
1098: (8)                     test = np.genfromtxt(data, dtype=descriptor)
1099: (8)                     assert_equal(test, control)
1100: (4)                 def test_array(self):
1101: (8)                     data = TextIO('1 2\n3 4')
1102: (8)                     control = np.array([[1, 2], [3, 4]], dtype=int)
1103: (8)                     test = np.genfromtxt(data, dtype=int)
1104: (8)                     assert_array_equal(test, control)
1105: (8)                     data.seek(0)
1106: (8)                     control = np.array([[1, 2], [3, 4]], dtype=float)
1107: (8)                     test = np.loadtxt(data, dtype=float)
1108: (8)                     assert_array_equal(test, control)
1109: (4)                 def test_1D(self):
1110: (8)                     control = np.array([1, 2, 3, 4], int)
1111: (8)                     data = TextIO('1\n2\n3\n4\n')
1112: (8)                     test = np.genfromtxt(data, dtype=int)
1113: (8)                     assert_array_equal(test, control)
1114: (8)                     data = TextIO('1,2,3,4\n')
1115: (8)                     test = np.genfromtxt(data, dtype=int, delimiter=',')
1116: (8)                     assert_array_equal(test, control)
1117: (4)                 def test_comments(self):
1118: (8)                     control = np.array([1, 2, 3, 5], int)
1119: (8)                     data = TextIO('# comment\n1,2,3,5\n')
1120: (8)                     test = np.genfromtxt(data, dtype=int, delimiter=',', comments='#')

```

```

1121: (8)             assert_equal(test, control)
1122: (8)             data = TextIO('1,2,3,5# comment\n')
1123: (8)             test = np.genfromtxt(data, dtype=int, delimiter=',', comments='#')
1124: (8)             assert_equal(test, control)
1125: (4)             def test_skiprows(self):
1126: (8)                 control = np.array([1, 2, 3, 5], int)
1127: (8)                 kwargs = dict(dtype=int, delimiter=',')
1128: (8)                 data = TextIO('comment\n1,2,3,5\n')
1129: (8)                 test = np.genfromtxt(data, skip_header=1, **kwargs)
1130: (8)                 assert_equal(test, control)
1131: (8)                 data = TextIO('# comment\n1,2,3,5\n')
1132: (8)                 test = np.loadtxt(data, skiprows=1, **kwargs)
1133: (8)                 assert_equal(test, control)
1134: (4)             def test_skip_footer(self):
1135: (8)                 data = ["# %i" % i for i in range(1, 6)]
1136: (8)                 data.append("A, B, C")
1137: (8)                 data.extend(["%i,%3.1f,%03s" % (i, i, i) for i in range(51)])
1138: (8)                 data[-1] = "99,99"
1139: (8)                 kwargs = dict(delimiter=",", names=True, skip_header=5,
skip_footer=10)
1140: (8)             test = np.genfromtxt(TextIO("\n".join(data)), **kwargs)
1141: (8)             ctrl = np.array([( "%f" % i, "%f" % i, "%f" % i) for i in range(41)],
1142: (24)                     dtype=[(_, float) for _ in "ABC"])
1143: (8)             assert_equal(test, ctrl)
1144: (4)             def test_skip_footer_with_invalid(self):
1145: (8)                 with suppress_warnings() as sup:
1146: (12)                     sup.filter(ConversionWarning)
1147: (12)                     basestr = '1 1\n2 2\n3 3\n4 4\n5 5\n6 6\n7 7\n'
1148: (12)                     assert_raises(ValueError, np.genfromtxt,
1149: (26)                         TextIO(basestr), skip_footer=1)
1150: (12)                     a = np.genfromtxt(
1151: (16)                         TextIO(basestr), skip_footer=1, invalid_raise=False)
1152: (12)                     assert_equal(a, np.array([[1., 1.], [2., 2.], [3., 3.], [4.,
4.]]))
1153: (12)             a = np.genfromtxt(TextIO(basestr), skip_footer=3)
1154: (12)             assert_equal(a, np.array([[1., 1.], [2., 2.], [3., 3.], [4.,
4.]]))
1155: (12)             basestr = '1 1\n2 2\n3 3\n4 4\n5 5\n6 6\n7 7\n'
1156: (12)             a = np.genfromtxt(
1157: (16)                 TextIO(basestr), skip_footer=1, invalid_raise=False)
1158: (12)             assert_equal(a, np.array([[1., 1.], [3., 3.], [4., 4.], [6.,
6.]]))
1159: (12)             a = np.genfromtxt(
1160: (16)                 TextIO(basestr), skip_footer=3, invalid_raise=False)
1161: (12)             assert_equal(a, np.array([[1., 1.], [3., 3.], [4., 4.]]))
1162: (4)             def test_header(self):
1163: (8)                 data = TextIO('gender age weight\nM 64.0 75.0\nF 25.0 60.0')
1164: (8)                 with warnings.catch_warnings(record=True) as w:
1165: (12)                     warnings.filterwarnings('always', ''),
np.VisibleDeprecationWarning)
1166: (12)                     test = np.genfromtxt(data, dtype=None, names=True)
1167: (12)                     assert_(w[0].category is np.VisibleDeprecationWarning)
1168: (8)                     control = {'gender': np.array(['M', 'F']),
1169: (19)                         'age': np.array([64.0, 25.0]),
1170: (19)                         'weight': np.array([75.0, 60.0])}
1171: (8)                     assert_equal(test['gender'], control['gender'])
1172: (8)                     assert_equal(test['age'], control['age'])
1173: (8)                     assert_equal(test['weight'], control['weight'])
1174: (4)             def test_auto_dtype(self):
1175: (8)                 data = TextIO('A 64 75.0 3+4j True\nBCD 25 60.0 5+6j False')
1176: (8)                 with warnings.catch_warnings(record=True) as w:
1177: (12)                     warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1178: (12)                     test = np.genfromtxt(data, dtype=None)
1179: (12)                     assert_(w[0].category is np.VisibleDeprecationWarning)
1180: (8)                     control = [np.array(['A', 'BCD']),
1181: (19)                         np.array([64, 25]),
1182: (19)                         np.array([75.0, 60.0]),
1183: (19)                         np.array([3 + 4j, 5 + 6j]),
```

```

1184: (19)             np.array([True, False]), ]
1185: (8)              assert_equal(test.dtype.names, ['f0', 'f1', 'f2', 'f3', 'f4'])
1186: (8)              for (i, ctrl) in enumerate(control):
1187: (12)                assert_equal(test['f%i' % i], ctrl)
1188: (4)              def test_auto_dtype_uniform(self):
1189: (8)                data = TextIO('1 2 3 4\n5 6 7 8\n')
1190: (8)                test = np.genfromtxt(data, dtype=None)
1191: (8)                control = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
1192: (8)                assert_equal(test, control)
1193: (4)              def test_fancy_dtype(self):
1194: (8)                data = TextIO('1,2,3.0\n4,5,6.0\n')
1195: (8)                fancydtype = np.dtype([('x', int), ('y', [('t', int), ('s', float)])])
1196: (8)                test = np.genfromtxt(data, dtype=fancydtype, delimiter=',')
1197: (8)                control = np.array([(1, (2, 3.0)), (4, (5, 6.0))], dtype=fancydtype)
1198: (8)                assert_equal(test, control)
1199: (4)              def test_names_overwrite(self):
1200: (8)                descriptor = {'names': ('g', 'a', 'w'),
1201: (22)                  'formats': ('S1', 'i4', 'f4')}
1202: (8)                data = TextIO(b'M 64.0 75.0\nF 25.0 60.0')
1203: (8)                names = ('gender', 'age', 'weight')
1204: (8)                test = np.genfromtxt(data, dtype=descriptor, names=names)
1205: (8)                descriptor['names'] = names
1206: (8)                control = np.array([('M', 64.0, 75.0),
1207: (28)                  ('F', 25.0, 60.0)], dtype=descriptor)
1208: (8)                assert_equal(test, control)
1209: (4)              def test_bad_fname(self):
1210: (8)                with pytest.raises(TypeError, match='fname must be a string,'):
1211: (12)                  np.genfromtxt(123)
1212: (4)              def test_commented_header(self):
1213: (8)                data = TextIO("""
1214: (0)                  M 21 72.100000
1215: (0)                  F 35 58.330000
1216: (0)                  M 33 21.99
1217: (8)                  """
1218: (8)                  with warnings.catch_warnings(record=True) as w:
1219: (12)                    warnings.filterwarnings('always', ''),
np.VisibleDeprecationWarning)
1220: (12)                    test = np.genfromtxt(data, names=True, dtype=None)
1221: (12)                    assert_(w[0].category is np.VisibleDeprecationWarning)
1222: (8)                    ctrl = np.array([('M', 21, 72.1), ('F', 35, 58.33), ('M', 33, 21.99)],
1223: (24)                      dtype=[('gender', '|S1'), ('age', int), ('weight',
float)])
1224: (8)                    assert_equal(test, ctrl)
1225: (8)                    data = TextIO(b"""
1226: (0)                  M 21 72.100000
1227: (0)                  F 35 58.330000
1228: (0)                  M 33 21.99
1229: (8)                  """
1230: (8)                  with warnings.catch_warnings(record=True) as w:
1231: (12)                    warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1232: (12)                    test = np.genfromtxt(data, names=True, dtype=None)
1233: (12)                    assert_(w[0].category is np.VisibleDeprecationWarning)
1234: (8)                    assert_equal(test, ctrl)
1235: (4)              def test_names_and_comments_none(self):
1236: (8)                data = TextIO('col1 col2\n1 2\n3 4')
1237: (8)                test = np.genfromtxt(data, dtype=(int, int), comments=None,
names=True)
1238: (8)                control = np.array([(1, 2), (3, 4)], dtype=[('col1', int), ('col2',
int)])
1239: (8)                assert_equal(test, control)
1240: (4)              def test_file_is_closed_on_error(self):
1241: (8)                with tempdir() as tmpdir:
1242: (12)                  fpath = os.path.join(tmpdir, "test.csv")
1243: (12)                  with open(fpath, "wb") as f:
1244: (16)                    f.write('\N{GREEK PI SYMBOL}'.encode())
1245: (12)                  with assert_no_warnings():
1246: (16)                    with pytest.raises(UnicodeDecodeError):
1247: (20)                      np.genfromtxt(fpath, encoding="ascii")

```

```

1248: (4)             def test_autonames_and_usecols(self):
1249: (8)                 data = TextIO('A B C D\naaaa 121 45 9.1')
1250: (8)                 with warnings.catch_warnings(record=True) as w:
1251: (12)                     warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1252: (12)                         test = np.genfromtxt(data, usecols=('A', 'C', 'D'),
1253: (32)                             names=True, dtype=None)
1254: (12)                         assert_(w[0].category is np.VisibleDeprecationWarning)
1255: (8)                         control = np.array(('aaaa', 45, 9.1),
1256: (27)                             dtype=[('A', '|S4'), ('C', int), ('D', float)])
1257: (8)                         assert_equal(test, control)
1258: (4)             def test_converters_with_usecols(self):
1259: (8)                 data = TextIO('1,2,3,,5\n6,7,8,9,10\n')
1260: (8)                 test = np.genfromtxt(data, dtype=int, delimiter=',',
1261: (28)                             converters={3: lambda s: int(s or - 999)},
1262: (28)                             usecols=(1, 3,))
1263: (8)                 control = np.array([[2, -999], [7, 9]], int)
1264: (8)                 assert_equal(test, control)
1265: (4)             def test_converters_with_usecols_and_names(self):
1266: (8)                 data = TextIO('A B C D\naaaa 121 45 9.1')
1267: (8)                 with warnings.catch_warnings(record=True) as w:
1268: (12)                     warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1269: (12)                         test = np.genfromtxt(data, usecols=('A', 'C', 'D'), names=True,
1270: (32)                             dtype=None,
1271: (32)                             converters={'C': lambda s: 2 * int(s)})
1272: (12)                         assert_(w[0].category is np.VisibleDeprecationWarning)
1273: (8)                         control = np.array(('aaaa', 90, 9.1),
1274: (27)                             dtype=[('A', '|S4'), ('C', int), ('D', float)])
1275: (8)                         assert_equal(test, control)
1276: (4)             def test_converters_cornercases(self):
1277: (8)                 converter = {
1278: (12)                     'date': lambda s: strptime(s, '%Y-%m-%d %H:%M:%SZ')}
1279: (8)                 data = TextIO('2009-02-03 12:00:00Z, 72214.0')
1280: (8)                 test = np.genfromtxt(data, delimiter=',', dtype=None,
1281: (28)                             names=['date', 'stid'], converters=converter)
1282: (8)                 control = np.array((datetime(2009, 2, 3), 72214.),
1283: (27)                             dtype=[('date', np.object_), ('stid', float)])
1284: (8)                 assert_equal(test, control)
1285: (4)             def test_converters_cornercases2(self):
1286: (8)                 converter = {
1287: (12)                     'date': lambda s: np.datetime64(strptime(s, '%Y-%m-%d
%H:%M:%SZ'))}
1288: (8)                 data = TextIO('2009-02-03 12:00:00Z, 72214.0')
1289: (8)                 test = np.genfromtxt(data, delimiter=',', dtype=None,
1290: (28)                             names=['date', 'stid'], converters=converter)
1291: (8)                 control = np.array((datetime(2009, 2, 3), 72214.),
1292: (27)                             dtype=[('date', 'datetime64[us]'), ('stid',
float)])
1293: (8)                 assert_equal(test, control)
1294: (4)             def test_unused_converter(self):
1295: (8)                 data = TextIO("1 21\n 3 42\n")
1296: (8)                 test = np.genfromtxt(data, usecols=(1,), converters={0: lambda s: int(s, 16)})
1297: (28)                 assert_equal(test, [21, 42])
1298: (8)                 data.seek(0)
1299: (8)                 test = np.genfromtxt(data, usecols=(1,), converters={1: lambda s: int(s, 16)})
1300: (8)                 assert_equal(test, [33, 66])
1301: (28)             def test_invalid_converter(self):
1302: (8)                 strip_rand = lambda x: float((b'r' in x.lower() and x.split()[-1]) or
1303: (4)                     (b'r' not in x.lower() and x.strip() or
1304: (8)                         0.0))
1305: (37)                         strip_per = lambda x: float((b'%' in x.lower() and x.split()[0]) or
1306: (8)                             (b'%' not in x.lower() and x.strip() or
1307: (36)                               0.0))
1308: (8)                         s = TextIO("D01N01,10/1/2003 ,1 %,R 75,400,600\r\n"
1309: (19)                           "L24U05,12/5/2003, 2 %,1,300, 150.5\r\n"
1310: (19)                           "D02N03,10/10/2004,R 1,,7,145.55")
```

```

1311: (8)
1312: (12)
1313: (12)
1314: (8)
1315: (4)
1316: (8)
1317: (8)
1318: (8)
1319: (8)
1320: (8)
1321: (4)
1322: (8)
1323: (8)
1324: (28)
1325: (8)
1326: (27)
float))
1327: (8)
1328: (8)
1329: (28)
1330: (8)
1331: (8)
1332: (4)
1333: (8)
1334: (8)
1335: (8)
1336: (8)
1337: (8)
1338: (29)
1339: (8)
dtype=dtyp)
1340: (8)
1341: (8)
1342: (8)
1343: (29)
1344: (8)
1345: (8)
1346: (4)
1347: (8)
1348: (19)
1349: (8)
1350: (8)
1351: (8)
1352: (8)
1353: (29)
1354: (8)
1355: (12)
1356: (12)
1357: (8)
1358: (8)
1359: (8)
1360: (33)
1361: (12)
1362: (33)
1363: (8)
1364: (8)
1365: (33)
1366: (12)
1367: (33)
1368: (4)
1369: (8)
1370: (8)
1371: (8)
1372: (8)
1373: (4)
1374: (8)
1375: (8)
1376: (29)
1377: (8)

        kwargs = dict(
            converters={2: strip_per, 3: strip_rand}, delimiter=",",
            dtype=None)
        assert_raises(ConverterError, np.genfromtxt, s, **kwargs)
    def test_tricky_converter_bug1666(self):
        s = TextIO('q1,2\nq3,4')
        cnv = lambda s: float(s[1:])
        test = np.genfromtxt(s, delimiter=',', converters={0: cnv})
        control = np.array([[1., 2.], [3., 4.]])
        assert_equal(test, control)
    def test_dtype_with_converters(self):
        dstr = "2009; 23; 46"
        test = np.genfromtxt(TextIO(dstr, ),
                            delimiter=";", dtype=float, converters={0: bytes})
        control = np.array([('2009', 23., 46)],
                           dtype=[('f0', '|S4'), ('f1', float), ('f2',
float)])
        assert_equal(test, control)
    test = np.genfromtxt(TextIO(dstr, ),
                        delimiter=";", dtype=float, converters={0: float})
    control = np.array([2009., 23., 46])
    assert_equal(test, control)
    def test_dtype_with_converters_and_usecols(self):
        dstr = "1,5,-1,1:1\n2,8,-1,1:n\n3,3,-2,m:n\n"
        dmap = {'1:1':0, '1:n':1, 'm:1':2, 'm:n':3}
        dtyp = [('e1','i4'),('e2','i4'),('e3','i2'),('n', 'i1')]
        conv = {0: int, 1: int, 2: int, 3: lambda r: dmap[r.decode()]}
        test = np.recfromcsv(TextIO(dstr, ), dtype=dtyp, delimiter=',',
                             names=None, converters=conv)
        control = np.rec.array([(1,5,-1,0), (2,8,-1,1), (3,3,-2,3)],
                             dtype=dtyp)
        assert_equal(test, control)
        dtyp = [('e1','i4'),('e2','i4'),('n', 'i1')]
        test = np.recfromcsv(TextIO(dstr, ), dtype=dtyp, delimiter=',',
                             usecols=(0,1,3), names=None, converters=conv)
        control = np.rec.array([(1,5,0), (2,8,1), (3,3,3)], dtype=dtyp)
        assert_equal(test, control)
    def test_dtype_with_object(self):
        data = """ 1; 2001-01-01
                    2; 2002-01-31 """
        ndtype = [('idx', int), ('code', object)]
        func = lambda s: strftime(s.strip(), "%Y-%m-%d")
        converters = {1: func}
        test = np.genfromtxt(TextIO(data), delimiter=";", dtype=ndtype,
                             converters=converters)
        control = np.array([
            (1, datetime(2001, 1, 1)), (2, datetime(2002, 1, 31))],
            dtype=ndtype)
        assert_equal(test, control)
        ndtype = [('nest', [('idx', int), ('code', object)])]
        with assert_raises_regex(NotImplementedError,
                               'Nested fields.* not supported.*'):
            test = np.genfromtxt(TextIO(data), delimiter=";",
                                 dtype=ndtype, converters=converters)
        ndtype = [('idx', int), ('code', object), ('nest', [])]
        with assert_raises_regex(NotImplementedError,
                               'Nested fields.* not supported.*'):
            test = np.genfromtxt(TextIO(data), delimiter=";",
                                 dtype=ndtype, converters=converters)
    def test_dtype_with_object_no_converter(self):
        parsed = np.genfromtxt(TextIO("1"), dtype=object)
        assert parsed[()] == b"1"
        parsed = np.genfromtxt(TextIO("string"), dtype=object)
        assert parsed[()] == b"string"
    def test_userconverters_with_explicit_dtype(self):
        data = TextIO('skip,skip,2001-01-01,1.0,skip')
        test = np.genfromtxt(data, delimiter=",", names=None, dtype=float,
                            usecols=(2, 3), converters={2: bytes})
        control = np.array([('2001-01-01', 1.)],
                           dtype=[('f0', '|S10'), ('f1', float)])
        assert_equal(test, control)

```

```

1378: (27)                               dtype=[(' ', '|S10'), (' ', float)])
1379: (8)       assert_equal(test, control)
1380: (4)       def test_utf8_userconverters_with_explicit_dtype(self):
1381: (8)           utf8 = b'\xcf\x96'
1382: (8)           with temppath() as path:
1383: (12)               with open(path, 'wb') as f:
1384: (16)                   f.write(b'skip,skip,2001-01-01' + utf8 + b',1.0,skip')
1385: (12)               test = np.genfromtxt(path, delimiter=",", names=None, dtype=float,
1386: (33)                           usecols=(2, 3), converters={2:
1387: (33)                               encoding='UTF-8')
1388: (8)               control = np.array([('2001-01-01' + utf8.decode('UTF-8'), 1.)],
1389: (27)                               dtype=[(' ', '|U11'), (' ', float)])
1390: (8)       assert_equal(test, control)
1391: (4)       def test_spacedelimiter(self):
1392: (8)           data = TextIO("1 2 3 4 5\n6 7 8 9 10")
1393: (8)           test = np.genfromtxt(data)
1394: (8)           control = np.array([[1., 2., 3., 4., 5.],
1395: (28)                           [6., 7., 8., 9., 10.]])
1396: (8)       assert_equal(test, control)
1397: (4)       def test_integer_delimiter(self):
1398: (8)           data = " 1 2 3\n 4 5 67\n890123 4"
1399: (8)           test = np.genfromtxt(TextIO(data), delimiter=3)
1400: (8)           control = np.array([[1, 2, 3], [4, 5, 67], [890, 123, 4]])
1401: (8)       assert_equal(test, control)
1402: (4)       def test_missing(self):
1403: (8)           data = TextIO('1,2,3,,5\n')
1404: (8)           test = np.genfromtxt(data, dtype=int, delimiter=',',
1405: (28)                           converters={3: lambda s: int(s or - 999)})
1406: (8)           control = np.array([1, 2, 3, -999, 5], int)
1407: (8)       assert_equal(test, control)
1408: (4)       def test_missing_with_tabs(self):
1409: (8)           txt = "1\t2\t3\n\t2\t\t1\t3"
1410: (8)           test = np.genfromtxt(TextIO(txt), delimiter="\t",
1411: (29)                           usemask=True,)
1412: (8)           ctrl_d = np.array([(1, 2, 3), (np.nan, 2, np.nan), (1, np.nan, 3)],)
1413: (8)           ctrl_m = np.array([(0, 0, 0), (1, 0, 1), (0, 1, 0)], dtype=bool)
1414: (8)       assert_equal(test.data, ctrl_d)
1415: (8)       assert_equal(test.mask, ctrl_m)
1416: (4)       def test_usecols(self):
1417: (8)           control = np.array([[1, 2], [3, 4]], float)
1418: (8)           data = TextIO()
1419: (8)           np.savetxt(data, control)
1420: (8)           data.seek(0)
1421: (8)           test = np.genfromtxt(data, dtype=float, usecols=(1,))
1422: (8)           assert_equal(test, control[:, 1])
1423: (8)           control = np.array([[1, 2, 3], [3, 4, 5]], float)
1424: (8)           data = TextIO()
1425: (8)           np.savetxt(data, control)
1426: (8)           data.seek(0)
1427: (8)           test = np.genfromtxt(data, dtype=float, usecols=(1, 2))
1428: (8)           assert_equal(test, control[:, 1:])
1429: (8)           data.seek(0)
1430: (8)           test = np.genfromtxt(data, dtype=float, usecols=np.array([1, 2]))
1431: (8)           assert_equal(test, control[:, 1:])
1432: (4)       def test_usecols_as_css(self):
1433: (8)           data = "1 2 3\n4 5 6"
1434: (8)           test = np.genfromtxt(TextIO(data),
1435: (29)                           names="a, b, c", usecols="a, c")
1436: (8)           ctrl = np.array([(1, 3), (4, 6)], dtype=[(_, float) for _ in "ac"])
1437: (8)       assert_equal(test, ctrl)
1438: (4)       def test_usecols_with_structured_dtype(self):
1439: (8)           data = TextIO("JOE 70.1 25.3\nBOB 60.5 27.9")
1440: (8)           names = ['stid', 'temp']
1441: (8)           dtypes = ['S4', 'f8']
1442: (8)           test = np.genfromtxt(
1443: (12)               data, usecols=(0, 2), dtype=list(zip(names, dtypes)))
1444: (8)           assert_equal(test['stid'], [b"JOE", b"BOB"])
1445: (8)           assert_equal(test['temp'], [25.3, 27.9])

```

```

1446: (4) def test_usecols_with_integer(self):
1447: (8)     test = np.genfromtxt(TextIO(b"1 2 3\n4 5 6"), usecols=0)
1448: (8)     assert_equal(test, np.array([1., 4.]))
1449: (4) def test_usecols_with_named_columns(self):
1450: (8)     ctrl = np.array([(1, 3), (4, 6)], dtype=[('a', float), ('c', float)])
1451: (8)     data = "1 2 3\n4 5 6"
1452: (8)     kwargs = dict(names="a, b, c")
1453: (8)     test = np.genfromtxt(TextIO(data), usecols=(0, -1), **kwargs)
1454: (8)     assert_equal(test, ctrl)
1455: (8)     test = np.genfromtxt(TextIO(data),
1456: (29)                     usecols=('a', 'c'), **kwargs)
1457: (8)     assert_equal(test, ctrl)
1458: (4) def test_empty_file(self):
1459: (8)     with suppress_warnings() as sup:
1460: (12)         sup.filter(message="genfromtxt: Empty input file:")
1461: (12)         data = TextIO()
1462: (12)         test = np.genfromtxt(data)
1463: (12)         assert_equal(test, np.array([]))
1464: (12)         test = np.genfromtxt(data, skip_header=1)
1465: (12)         assert_equal(test, np.array([]))
1466: (4) def test_fancy_dtype_alt(self):
1467: (8)     data = TextIO('1,2,3.0\n4,5,6.0\n')
1468: (8)     fancydtype = np.dtype([('x', int), ('y', [('t', int), ('s', float)])])
1469: (8)     test = np.genfromtxt(data, dtype=fancydtype, delimiter=',',
usemask=True)
1470: (8)     control = ma.array([(1, (2, 3.0)), (4, (5, 6.0))], dtype=fancydtype)
1471: (8)     assert_equal(test, control)
1472: (4) def test_shaped_dtype(self):
1473: (8)     c = TextIO("aaaa 1.0 8.0 1 2 3 4 5 6")
1474: (8)     dt = np.dtype([('name', 'S4'), ('x', float), ('y', float),
1475: (23)           ('block', int, (2, 3))])
1476: (8)     x = np.genfromtxt(c, dtype=dt)
1477: (8)     a = np.array([('aaaa', 1.0, 8.0, [[1, 2, 3], [4, 5, 6]]]),
1478: (21)           dtype=dt)
1479: (8)     assert_array_equal(x, a)
1480: (4) def test_withmissing(self):
1481: (8)     data = TextIO('A,B\n0,1\nN/A')
1482: (8)     kwargs = dict(delimiter=",", missing_values="N/A", names=True)
1483: (8)     test = np.genfromtxt(data, dtype=None, usemask=True, **kwargs)
1484: (8)     control = ma.array([(0, 1), (2, -1)],
1485: (27)           mask=[(False, False), (False, True)],
1486: (27)           dtype=[('A', int), ('B', int)])
1487: (8)     assert_equal(test, control)
1488: (8)     assert_equal(test.mask, control.mask)
1489: (8)     data.seek(0)
1490: (8)     test = np.genfromtxt(data, usemask=True, **kwargs)
1491: (8)     control = ma.array([(0, 1), (2, -1)],
1492: (27)           mask=[(False, False), (False, True)],
1493: (27)           dtype=[('A', float), ('B', float)])
1494: (8)     assert_equal(test, control)
1495: (8)     assert_equal(test.mask, control.mask)
1496: (4) def test_user_missing_values(self):
1497: (8)     data = "A, B, C\n0, 0., 0j\n1, N/A, 1j\n-9, 2.2, N/A\n3, -99, 3j"
1498: (8)     basekwargs = dict(dtype=None, delimiter=",", names=True,)
1499: (8)     mdtype = [('A', int), ('B', float), ('C', complex)]
1500: (8)     test = np.genfromtxt(TextIO(data), missing_values="N/A",
1501: (28)           **basekwargs)
1502: (8)     control = ma.array([(0, 0.0, 0j), (1, -999, 1j),
1503: (28)                   (-9, 2.2, -999j), (3, -99, 3j)],
1504: (27)                   mask=[(0, 0, 0), (0, 1, 0), (0, 0, 1), (0, 0, 0)],
1505: (27)                   dtype=mdtype)
1506: (8)     assert_equal(test, control)
1507: (8)     basekwargs['dtype'] = mdtype
1508: (8)     test = np.genfromtxt(TextIO(data),
1509: (28)                     missing_values={0: -9, 1: -99, 2: -999j},
usemask=True, **basekwargs)
1510: (8)     control = ma.array([(0, 0.0, 0j), (1, -999, 1j),
1511: (28)                   (-9, 2.2, -999j), (3, -99, 3j)],
1512: (27)                   mask=[(0, 0, 0), (0, 1, 0), (1, 0, 1), (0, 1, 0)],
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1513: (27)
1514: (8)
1515: (8)
1516: (28)
1517: (28)
1518: (28)
1519: (8)
1520: (28)
1521: (27)
1522: (27)
1523: (8)
1524: (4)
def test_user_filling_values(self):
1525: (8)
1526: (8)
1527: (8)
1528: (22)
1529: (22)
1530: (22)
1531: (22)
1532: (8)
1533: (8)
1534: (24)
1535: (8)
1536: (8)
1537: (8)
1538: (8)
1539: (8)
1540: (8)
1541: (29)
1542: (8)
1543: (8)
1544: (8)
1545: (29)
1546: (8)
1547: (8)
1548: (4)
def test_withmissing_float(self):
1549: (8)
1550: (8)
1551: (28)
1552: (8)
1553: (27)
1554: (27)
1555: (8)
1556: (8)
1557: (4)
def test_with_masked_column_uniform(self):
1558: (8)
1559: (8)
1560: (29)
1561: (8)
0])
1562: (8)
1563: (4)
1564: (8)
1565: (8)
1566: (29)
1567: (8)
1568: (27)
1569: (27)
1570: (8)
1571: (4)
1572: (8)
1573: (8)
1574: (12)
1575: (8)
1576: (8)
1577: (8)
1578: (8)
1579: (12)
1580: (8)

        dtype=mdtype)
        assert_equal(test, control)
        test = np.genfromtxt(TextIO(data),
                             missing_values={0: -9, 'B': -99, 'C': -999j},
                             usemask=True,
                             **basekwargs)
        control = ma.array([(0, 0.0, 0j), (1, -999, 1j),
                            (-9, 2.2, -999j), (3, -99, 3j)],
                           mask=[(0, 0, 0), (0, 1, 0), (1, 0, 1), (0, 1, 0)],
                           dtype=mdtype)
        assert_equal(test, control)
def test_user_filling_values(self):
    ctrl = np.array([(0, 3), (4, -999)], dtype=[('a', int), ('b', int)])
    data = "N/A, 2, 3\n4, ,???""
    kwargs = dict(delimiter=",",
                  dtype=int,
                  names="a,b,c",
                  missing_values={0: "N/A", 'b': " ", 2: "???"},
                  filling_values={0: 0, 'b': 0, 2: -999})
    test = np.genfromtxt(TextIO(data), **kwargs)
    ctrl = np.array([(0, 2, 3), (4, 0, -999)],
                   dtype=[(_, int) for _ in "abc"])
    assert_equal(test, ctrl)
    test = np.genfromtxt(TextIO(data), usecols=(0, -1), **kwargs)
    ctrl = np.array([(0, 3), (4, -999)], dtype=[(_, int) for _ in "ac"])
    assert_equal(test, ctrl)
    data2 = "1,2,*\n4\n5,*\n7,8\n"
    test = np.genfromtxt(TextIO(data2), delimiter=',', dtype=int,
                         missing_values="*", filling_values=0)
    ctrl = np.array([[1, 2, 0, 4], [5, 0, 7, 8]])
    assert_equal(test, ctrl)
    test = np.genfromtxt(TextIO(data2), delimiter=',', dtype=int,
                         missing_values="*", filling_values=-1)
    ctrl = np.array([[1, 2, -1, 4], [5, -1, 7, 8]])
    assert_equal(test, ctrl)
def test_withmissing_float(self):
    data = TextIO('A,B\n0,1.5\n2,-999.00')
    test = np.genfromtxt(data, dtype=None, delimiter=',',
                         missing_values=-999.0, names=True, usemask=True)
    control = ma.array([(0, 1.5), (2, -1.)],
                      mask=[(False, False), (False, True)],
                      dtype=[('A', int), ('B', float)])
    assert_equal(test, control)
    assert_equal(test.mask, control.mask)
def test_with_masked_column_uniform(self):
    data = TextIO('1 2 3\n4 5 6\n')
    test = np.genfromtxt(data, dtype=None,
                         missing_values='2,5', usemask=True)
    control = ma.array([[1, 2, 3], [4, 5, 6]], mask=[[0, 1, 0], [0, 1,
0]])
    assert_equal(test, control)
def test_with_masked_column_various(self):
    data = TextIO('True 2 3\nFalse 5 6\n')
    test = np.genfromtxt(data, dtype=None,
                         missing_values='2,5', usemask=True)
    control = ma.array([(1, 2, 3), (0, 5, 6)],
                      mask=[(0, 1, 0), (0, 1, 0)],
                      dtype=[('f0', bool), ('f1', bool), ('f2', int)])
    assert_equal(test, control)
def test_invalid_raise(self):
    data = ["1, 1, 1, 1, 1"] * 50
    for i in range(5):
        data[10 * i] = "2, 2, 2, 2 2"
    data.insert(0, "a, b, c, d, e")
    mdata = TextIO("\n".join(data))
    kwargs = dict(delimiter=",", dtype=None, names=True)
    def f():
        return np.genfromtxt(mdata, invalid_raise=False, **kwargs)
    mtest = assert_warns(ConversionWarning, f)

```

```

1581: (8)             assert_equal(len(mtest), 45)
1582: (8)             assert_equal(mtest, np.ones(45, dtype=[(_, int) for _ in 'abcde']))
1583: (8)             mdata.seek(0)
1584: (8)             assert_raises(ValueError, np.genfromtxt, mdata,
1585: (22)                         delimiter=",", names=True)
1586: (4)             def test_invalid_raise_with_usecols(self):
1587: (8)                 data = ["1, 1, 1, 1, 1"] * 50
1588: (8)                 for i in range(5):
1589: (12)                     data[10 * i] = "2, 2, 2, 2 2"
1590: (8)                 data.insert(0, "a, b, c, d, e")
1591: (8)                 mdata = TextIO("\n".join(data))
1592: (8)                 kwargs = dict(delimiter=",", dtype=None, names=True,
1593: (22)                         invalid_raise=False)
1594: (8)             def f():
1595: (12)                 return np.genfromtxt(mdata, usecols=(0, 4), **kwargs)
1596: (8)             mtest = assert_warns(ConversionWarning, f)
1597: (8)             assert_equal(len(mtest), 45)
1598: (8)             assert_equal(mtest, np.ones(45, dtype=[(_, int) for _ in 'ae']))
1599: (8)             mdata.seek(0)
1600: (8)             mtest = np.genfromtxt(mdata, usecols=(0, 1), **kwargs)
1601: (8)             assert_equal(len(mtest), 50)
1602: (8)             control = np.ones(50, dtype=[(_, int) for _ in 'ab'])
1603: (8)             control[[10 * _ for _ in range(5)]] = (2, 2)
1604: (8)             assert_equal(mtest, control)
1605: (4)             def test_inconsistent_dtype(self):
1606: (8)                 data = ["1, 1, 1, 1, -1.1"] * 50
1607: (8)                 mdata = TextIO("\n".join(data))
1608: (8)                 converters = {4: lambda x: "(%s)" % x.decode()}
1609: (8)                 kwargs = dict(delimiter=",", converters=converters,
1610: (22)                         dtype=[(_, int) for _ in 'abcde'],)
1611: (8)                 assert_raises(ValueError, np.genfromtxt, mdata, **kwargs)
1612: (4)             def test_default_field_format(self):
1613: (8)                 data = "0, 1, 2.3\n4, 5, 6.7"
1614: (8)                 mtest = np.genfromtxt(TextIO(data),
1615: (29)                         delimiter=",", dtype=None, defaultfmt="f%02i")
1616: (8)                 ctrl = np.array([(0, 1, 2.3), (4, 5, 6.7)],
1617: (24)                         dtype=[("f00", int), ("f01", int), ("f02", float)])
1618: (8)                 assert_equal(mtest, ctrl)
1619: (4)             def test_single_dtype_wo_names(self):
1620: (8)                 data = "0, 1, 2.3\n4, 5, 6.7"
1621: (8)                 mtest = np.genfromtxt(TextIO(data),
1622: (29)                         delimiter=",", dtype=float, defaultfmt="f%02i")
1623: (8)                 ctrl = np.array([[0., 1., 2.3], [4., 5., 6.7]], dtype=float)
1624: (8)                 assert_equal(mtest, ctrl)
1625: (4)             def test_single_dtype_w_explicit_names(self):
1626: (8)                 data = "0, 1, 2.3\n4, 5, 6.7"
1627: (8)                 mtest = np.genfromtxt(TextIO(data),
1628: (29)                         delimiter=",", dtype=float, names="a, b, c")
1629: (8)                 ctrl = np.array([(0., 1., 2.3), (4., 5., 6.7)],
1630: (24)                         dtype=[(_, float) for _ in "abc"])
1631: (8)                 assert_equal(mtest, ctrl)
1632: (4)             def test_single_dtype_w_implicit_names(self):
1633: (8)                 data = "a, b, c\n0, 1, 2.3\n4, 5, 6.7"
1634: (8)                 mtest = np.genfromtxt(TextIO(data),
1635: (29)                         delimiter=",", dtype=float, names=True)
1636: (8)                 ctrl = np.array([(0., 1., 2.3), (4., 5., 6.7)],
1637: (24)                         dtype=[(_, float) for _ in "abc"])
1638: (8)                 assert_equal(mtest, ctrl)
1639: (4)             def test_easy_structured_dtype(self):
1640: (8)                 data = "0, 1, 2.3\n4, 5, 6.7"
1641: (8)                 mtest = np.genfromtxt(TextIO(data), delimiter=",",
1642: (29)                         dtype=(int, float, float), defaultfmt="f_%02i")
1643: (8)                 ctrl = np.array([(0, 1, 2.3), (4, 5, 6.7)],
1644: (24)                         dtype=[("f_00", int), ("f_01", float), ("f_02",
float)])
1645: (8)                 assert_equal(mtest, ctrl)
1646: (4)             def test_strip(self):
1647: (8)                 data = "01/01/2003 , 1.3, abcde"
1648: (8)                 kwargs = dict(delimiter=",", dtype=None)

```

```

1649: (8)             with warnings.catch_warnings(record=True) as w:
1650: (12)             warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1651: (12)             mtest = np.genfromtxt(TextIO(data), **kwargs)
1652: (12)             assert_(w[0].category is np.VisibleDeprecationWarning)
1653: (8)             ctrl = np.array([('01/01/2003 ', 1.3, ' abcde')],
1654: (24)                         dtype=[('f0', '|S12'), ('f1', float), ('f2', '|S8')])
1655: (8)             assert_equal(mtest, ctrl)
1656: (8)             with warnings.catch_warnings(record=True) as w:
1657: (12)                 warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1658: (12)                 mtest = np.genfromtxt(TextIO(data), autostrip=True, **kwargs)
1659: (12)                 assert_(w[0].category is np.VisibleDeprecationWarning)
1660: (8)                 ctrl = np.array([('01/01/2003', 1.3, 'abcde')],
1661: (24)                         dtype=[('f0', '|S10'), ('f1', float), ('f2', '|S5')])
1662: (8)                 assert_equal(mtest, ctrl)
1663: (4)             def test_replace_space(self):
1664: (8)                 txt = "A.A, B (B), C:C\n1, 2, 3.14"
1665: (8)                 test = np.genfromtxt(TextIO(txt),
1666: (29)                         delimiter=",", names=True, dtype=None)
1667: (8)                 ctrl_dtype = [("AA", int), ("B_B", int), ("CC", float)]
1668: (8)                 ctrl = np.array((1, 2, 3.14), dtype=ctrl_dtype)
1669: (8)                 assert_equal(test, ctrl)
1670: (8)                 test = np.genfromtxt(TextIO(txt),
1671: (29)                         delimiter=",", names=True, dtype=None,
1672: (29)                         replace_space='', deletechars='')
1673: (8)                 ctrl_dtype = [("A.A", int), ("B (B)", int), ("C:C", float)]
1674: (8)                 ctrl = np.array((1, 2, 3.14), dtype=ctrl_dtype)
1675: (8)                 assert_equal(test, ctrl)
1676: (8)                 test = np.genfromtxt(TextIO(txt),
1677: (29)                         delimiter=",", names=True, dtype=None,
1678: (29)                         deletechars='')
1679: (8)                 ctrl_dtype = [("A.A", int), ("B_(B)", int), ("C:C", float)]
1680: (8)                 ctrl = np.array((1, 2, 3.14), dtype=ctrl_dtype)
1681: (8)                 assert_equal(test, ctrl)
1682: (4)             def test_replace_space_known_dtype(self):
1683: (8)                 txt = "A.A, B (B), C:C\n1, 2, 3"
1684: (8)                 test = np.genfromtxt(TextIO(txt),
1685: (29)                         delimiter=",", names=True, dtype=int)
1686: (8)                 ctrl_dtype = [("AA", int), ("B_B", int), ("CC", int)]
1687: (8)                 ctrl = np.array((1, 2, 3), dtype=ctrl_dtype)
1688: (8)                 assert_equal(test, ctrl)
1689: (8)                 test = np.genfromtxt(TextIO(txt),
1690: (29)                         delimiter=",", names=True, dtype=int,
1691: (29)                         replace_space='', deletechars='')
1692: (8)                 ctrl_dtype = [("A.A", int), ("B (B)", int), ("C:C", int)]
1693: (8)                 ctrl = np.array((1, 2, 3), dtype=ctrl_dtype)
1694: (8)                 assert_equal(test, ctrl)
1695: (8)                 test = np.genfromtxt(TextIO(txt),
1696: (29)                         delimiter=",", names=True, dtype=int,
1697: (29)                         deletechars='')
1698: (8)                 ctrl_dtype = [("A.A", int), ("B_(B)", int), ("C:C", int)]
1699: (8)                 ctrl = np.array((1, 2, 3), dtype=ctrl_dtype)
1700: (8)                 assert_equal(test, ctrl)
1701: (4)             def test_incomplete_names(self):
1702: (8)                 data = "A,,C\n0,1,2\\n3,4,5"
1703: (8)                 kwargs = dict(delimiter=",", names=True)
1704: (8)                 ctrl = np.array([(0, 1, 2), (3, 4, 5)],
1705: (24)                         dtype=[(_, int) for _ in ('A', 'f0', 'C')])
1706: (8)                 test = np.genfromtxt(TextIO(data), dtype=None, **kwargs)
1707: (8)                 assert_equal(test, ctrl)
1708: (8)                 ctrl = np.array([(0, 1, 2), (3, 4, 5)],
1709: (24)                         dtype=[(_, float) for _ in ('A', 'f0', 'C')])
1710: (8)                 test = np.genfromtxt(TextIO(data), **kwargs)
1711: (4)             def test_names_auto_completion(self):
1712: (8)                 data = "1 2 3\\n 4 5 6"
1713: (8)                 test = np.genfromtxt(TextIO(data),
1714: (29)                         dtype=(int, float, int), names="a")
1715: (8)                 ctrl = np.array([(1, 2, 3), (4, 5, 6)],
```

```

1716: (24)           dtype=[('a', int), ('f0', float), ('f1', int)])
1717: (8)             assert_equal(test, ctrl)
1718: (4)             def test_names_with_usecols_bug1636(self):
1719: (8)               data = "A,B,C,D,E\n0,1,2,3,4\n0,1,2,3,4\n0,1,2,3,4"
1720: (8)               ctrl_names = ("A", "C", "E")
1721: (8)               test = np.genfromtxt(TextIO(data),
1722: (29)                 dtype=(int, int, int), delimiter=",",
1723: (29)                 usecols=(0, 2, 4), names=True)
1724: (8)               assert_equal(test.dtype.names, ctrl_names)
1725: (8)               test = np.genfromtxt(TextIO(data),
1726: (29)                 dtype=(int, int, int), delimiter=",",
1727: (29)                 usecols=("A", "C", "E"), names=True)
1728: (8)               assert_equal(test.dtype.names, ctrl_names)
1729: (8)               test = np.genfromtxt(TextIO(data),
1730: (29)                 dtype=int, delimiter=",",
1731: (29)                 usecols=("A", "C", "E"), names=True)
1732: (8)               assert_equal(test.dtype.names, ctrl_names)
1733: (4)             def test_fixed_width_names(self):
1734: (8)               data = "     A   B   C\nn    0     1 2.3\nn    45   67   9."
1735: (8)               kwargs = dict(delimiter=(5, 5, 4), names=True, dtype=None)
1736: (8)               ctrl = np.array([(0, 1, 2.3), (45, 67, 9.)],
1737: (24)                 dtype=[('A', int), ('B', int), ('C', float)])
1738: (8)               test = np.genfromtxt(TextIO(data), **kwargs)
1739: (8)               assert_equal(test, ctrl)
1740: (8)               kwargs = dict(delimiter=5, names=True, dtype=None)
1741: (8)               ctrl = np.array([(0, 1, 2.3), (45, 67, 9.)],
1742: (24)                 dtype=[('A', int), ('B', int), ('C', float)])
1743: (8)               test = np.genfromtxt(TextIO(data), **kwargs)
1744: (8)               assert_equal(test, ctrl)
1745: (4)             def test_filling_values(self):
1746: (8)               data = b"1, 2, 3\n1, , 5\n0, 6, \n"
1747: (8)               kwargs = dict(delimiter=",", dtype=None, filling_values=-999)
1748: (8)               ctrl = np.array([[1, 2, 3], [1, -999, 5], [0, 6, -999]], dtype=int)
1749: (8)               test = np.genfromtxt(TextIO(data), **kwargs)
1750: (8)               assert_equal(test, ctrl)
1751: (4)             def test_comments_is_none(self):
1752: (8)               with warnings.catch_warnings(record=True) as w:
1753: (12)                 warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1754: (12)                   test = np.genfromtxt(TextIO("test1,testNonetherestofthedata"),
1755: (33)                     dtype=None, comments=None, delimiter=',')
1756: (12)                   assert_(w[0].category is np.VisibleDeprecationWarning)
1757: (8)                   assert_equal(test[1], b'testNonetherestofthedata')
1758: (8)                   with warnings.catch_warnings(record=True) as w:
1759: (12)                     warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1760: (12)                   test = np.genfromtxt(TextIO("test1, testNonetherestofthedata"),
1761: (33)                     dtype=None, comments=None, delimiter=',')
1762: (12)                   assert_(w[0].category is np.VisibleDeprecationWarning)
1763: (8)                   assert_equal(test[1], b' testNonetherestofthedata')
1764: (4)             def test_latin1(self):
1765: (8)               latin1 = b'\xf6\xfc\xf6'
1766: (8)               norm = b"norm1,norm2,norm3\n"
1767: (8)               enc = b"test1,testNonethe" + latin1 + b",test3\n"
1768: (8)               s = norm + enc + norm
1769: (8)               with warnings.catch_warnings(record=True) as w:
1770: (12)                 warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1771: (12)                   test = np.genfromtxt(TextIO(s),
1772: (33)                     dtype=None, comments=None, delimiter=',')
1773: (12)                   assert_(w[0].category is np.VisibleDeprecationWarning)
1774: (8)                   assert_equal(test[1, 0], b"test1")
1775: (8)                   assert_equal(test[1, 1], b"testNonethe" + latin1)
1776: (8)                   assert_equal(test[1, 2], b"test3")
1777: (8)                   test = np.genfromtxt(TextIO(s),
1778: (29)                     dtype=None, comments=None, delimiter=',',
1779: (29)                     encoding='latin1')
1780: (8)                   assert_equal(test[1, 0], "test1")
1781: (8)                   assert_equal(test[1, 1], "testNonethe" + latin1.decode('latin1'))

```

```

1782: (8)             assert_equal(test[1, 2], "test3")
1783: (8)             with warnings.catch_warnings(record=True) as w:
1784: (12)                 warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1785: (12)                 test = np.genfromtxt(TextIO(b"0,testNonethe" + latin1),
1786: (33)                               dtype=None, comments=None, delimiter=',')
1787: (12)                     assert_(w[0].category is np.VisibleDeprecationWarning)
1788: (8)                     assert_equal(test['f0'], 0)
1789: (8)                     assert_equal(test['f1'], b"testNonethe" + latin1)
1790: (4)             def test_binary_decode_autodtype(self):
1791: (8)                 utf16 = b'\xff\xfe\x04 \x00i\x04 \x00j\x04'
1792: (8)                 v = self.loadtxt(BytesIO(utf16), dtype=None, encoding='UTF-16')
1793: (8)                 assert_array_equal(v, np.array(utf16.decode('UTF-16').split()))
1794: (4)             def test_utf8_byte_encoding(self):
1795: (8)                 utf8 = b"\xcf\x96"
1796: (8)                 norm = b"norm1,norm2,norm3\n"
1797: (8)                 enc = b"test1,testNonethe" + utf8 + b",test3\n"
1798: (8)                 s = norm + enc + norm
1799: (8)                 with warnings.catch_warnings(record=True) as w:
1800: (12)                     warnings.filterwarnings('always', '',
np.VisibleDeprecationWarning)
1801: (12)                     test = np.genfromtxt(TextIO(s),
1802: (33)                           dtype=None, comments=None, delimiter=',')
1803: (12)                     assert_(w[0].category is np.VisibleDeprecationWarning)
1804: (8)             ctl = np.array([
1805: (17)                 [b'norm1', b'norm2', b'norm3'],
1806: (17)                 [b'test1', b'testNonethe' + utf8, b'test3'],
1807: (17)                 [b'norm1', b'norm2', b'norm3']])
1808: (8)             assert_array_equal(test, ctl)
1809: (4)             def test_utf8_file(self):
1810: (8)                 utf8 = b"\xcf\x96"
1811: (8)                 with temppath() as path:
1812: (12)                     with open(path, "wb") as f:
1813: (16)                         f.write((b"test1,testNonethe" + utf8 + b",test3\n") * 2)
1814: (12)                     test = np.genfromtxt(path, dtype=None, comments=None,
1815: (33)                           delimiter=',', encoding="UTF-8")
1816: (12)                     ctl = np.array([
1817: (21)                         ["test1", "testNonethe" + utf8.decode("UTF-8"), "test3"],
1818: (21)                         ["test1", "testNonethe" + utf8.decode("UTF-8"),
1819: ("test3")],
1820: (21)                         dtype=np.str_]
1821: (12)                     assert_array_equal(test, ctl)
1822: (12)                     with open(path, "wb") as f:
1823: (16)                         f.write(b"0,testNonethe" + utf8)
1824: (12)                     test = np.genfromtxt(path, dtype=None, comments=None,
1825: (33)                           delimiter=',', encoding="UTF-8")
1826: (12)                     assert_equal(test['f0'], 0)
1827: (4)             def test_utf8_file_nodtype_unicode(self):
1828: (8)                 utf8 = '\u03d6'
1829: (8)                 latin1 = '\xfc\xfc\xfc'
1830: (8)                 try:
1831: (12)                     encoding = locale.getpreferredencoding()
1832: (12)                     utf8.encode(encoding)
1833: (8)                 except (UnicodeError, ImportError):
1834: (12)                     pytest.skip('Skipping test_utf8_file_nodtype_unicode, '
1835: (24)                           'unable to encode utf8 in preferred encoding')
1836: (8)                 with temppath() as path:
1837: (12)                     with io.open(path, "wt") as f:
1838: (16)                         f.write("norm1,norm2,norm3\n")
1839: (16)                         f.write("norm1," + latin1 + ",norm3\n")
1840: (16)                         f.write("test1,testNonethe" + utf8 + ",test3\n")
1841: (12)                     with warnings.catch_warnings(record=True) as w:
1842: (16)                         warnings.filterwarnings('always', '',
1843: (40)                             np.VisibleDeprecationWarning)
1844: (16)                         test = np.genfromtxt(path, dtype=None, comments=None,
1845: (37)                               delimiter=',')
1846: (16)                         assert_(w[0].category is np.VisibleDeprecationWarning)
1847: (12)                         ctl = np.array([

```

```

1848: (21)
1849: (21)
1850: (21)
1851: (21)
1852: (12)
1853: (4)
1854: (8)
1855: (8)
1856: (8)
1857: (8)
1858: (27)
1859: (8)
1860: (8)
1861: (8)
1862: (8)
1863: (8)
1864: (27)
1865: (27)
1866: (8)
1867: (8)
1868: (8)
1869: (4)
1870: (8)
1871: (8)
1872: (8)
1873: (8)
1874: (27)
1875: (8)
1876: (8)
1877: (8)
1878: (8)
1879: (8)
1880: (27)
1881: (27)
1882: (8)
1883: (8)
1884: (8)
1885: (8)
1886: (8)
1887: (8)
1888: (27)
1889: (8)
1890: (8)
1891: (8)
1892: (8)
1893: (8)
1894: (8)
1895: (27)
1896: (8)
1897: (8)
1898: (8)
1899: (8)
1900: (8)
4)])
1901: (8)
1902: (8)
1903: (4)
1904: (8)
1905: (8)
1906: (8)
1907: (8)
1908: (8)
1909: (8)
1910: (8)
1911: (8)
1912: (8)
1913: (8)
1914: (8)
1915: (8)

        ["norm1", "norm2", "norm3"],
        ["norm1", latin1, "norm3"],
        ["test1", "testNonethe" + utf8, "test3"]],
        dtype=np.str_)
    assert_array_equal(test, ctl)
def test_recfromtxt(self):
    data = TextIO('A,B\n0,1\n2,3')
    kwargs = dict(delimiter=',', missing_values="N/A", names=True)
    test = np.recfromtxt(data, **kwargs)
    control = np.array([(0, 1), (2, 3)],
                      dtype=[('A', int), ('B', int)])
    assert_(isinstance(test, np.recarray))
    assert_equal(test, control)
    data = TextIO('A,B\n0,1\n2,N/A')
    test = np.recfromtxt(data, dtype=None, usemask=True, **kwargs)
    control = ma.array([(0, 1), (2, -1)],
                      mask=[(False, False), (False, True)],
                      dtype=[('A', int), ('B', int)])
    assert_equal(test, control)
    assert_equal(test.mask, control.mask)
    assert_equal(test.A, [0, 2])
def test_recfromcsv(self):
    data = TextIO('A,B\n0,1\n2,3')
    kwargs = dict(missing_values="N/A", names=True, case_sensitive=True)
    test = np.recfromcsv(data, dtype=None, **kwargs)
    control = np.array([(0, 1), (2, 3)],
                      dtype=[('A', int), ('B', int)])
    assert_(isinstance(test, np.recarray))
    assert_equal(test, control)
    data = TextIO('A,B\n0,1\n2,N/A')
    test = np.recfromcsv(data, dtype=None, usemask=True, **kwargs)
    control = ma.array([(0, 1), (2, -1)],
                      mask=[(False, False), (False, True)],
                      dtype=[('A', int), ('B', int)])
    assert_equal(test, control)
    assert_equal(test.mask, control.mask)
    assert_equal(test.A, [0, 2])
    data = TextIO('A,B\n0,1\n2,3')
    test = np.recfromcsv(data, missing_values='N/A')
    control = np.array([(0, 1), (2, 3)],
                      dtype=[('a', int), ('b', int)])
    assert_(isinstance(test, np.recarray))
    assert_equal(test, control)
    data = TextIO('A,B\n0,1\n2,3')
    dtype = [('a', int), ('b', float)]
    test = np.recfromcsv(data, missing_values='N/A', dtype=dtype)
    control = np.array([(0, 1), (2, 3)],
                      dtype=dtype)
    assert_(isinstance(test, np.recarray))
    assert_equal(test, control)
    data = TextIO('color\n"red"\n"blue"')
    test = np.recfromcsv(data, converters={0: lambda x: x.strip(b'\"')})
    control = np.array([('red',), ('blue',)], dtype=[('color', (bytes,
4))])
    assert_equal(test.dtype, control.dtype)
    assert_equal(test, control)
def test_max_rows(self):
    data = '1 2\n3 4\n5 6\n7 8\n9 10\n'
    txt = TextIO(data)
    a1 = np.genfromtxt(txt, max_rows=3)
    a2 = np.genfromtxt(txt)
    assert_equal(a1, [[1, 2], [3, 4], [5, 6]])
    assert_equal(a2, [[7, 8], [9, 10]])
    assert_raises(ValueError, np.genfromtxt, TextIO(data), max_rows=0)
    data = '1 1\n2 2\n0 0\n3 3\n4 4\n5 5\n6 6\n7 7\n'
    test = np.genfromtxt(TextIO(data), max_rows=2)
    control = np.array([[1., 1.], [2., 2.]])
    assert_equal(test, control)
    assert_raises(ValueError, np.genfromtxt, TextIO(data), skip_footer=1,

```

```

1916: (22)
1917: (8)
1918: (8)
1919: (12)
1920: (12)
1921: (12)
1922: (12)
1923: (12)
1924: (12)
1925: (12)
1926: (8)
1927: (8)
1928: (8)
1929: (8)
1930: (22)
1931: (8)
1932: (8)
1933: (8)
1934: (22)
1935: (8)
1936: (4)
1937: (8)
1938: (8)
1939: (8)
1940: (12)
1941: (12)
1942: (16)
1943: (20)
1944: (16)
1945: (12)
1946: (4)
1947: (8)
1948: (8)
1949: (8)
1950: (12)
1951: (12)
1952: (12)
1953: (16)
1954: (12)
1955: (16)
1956: (20)
1957: (16)
1958: (4)
1959: (8)
1960: (12)
1961: (16)
1962: (8)
1963: (8)
1964: (4)
1965: (8)
1966: (8)
1967: (8)
1968: (8)
1969: (8)
1970: (8)
1971: (8)
1972: (8)
1973: (8)
1974: (4)
1975: (8)
1976: (8)
1977: (8)
1978: (8)
1979: (8)
1980: (4)
1981: (8)
1982: (8)

        max_rows=4)
    assert_raises(ValueError, np.genfromtxt, TextIO(data), max_rows=4)
    with suppress_warnings() as sup:
        sup.filter(ConversionWarning)
        test = np.genfromtxt(TextIO(data), max_rows=4,
                             control = np.array([[1., 1.], [2., 2.], [3., 3.], [4., 4.]]))
        assert_equal(test, control)
        test = np.genfromtxt(TextIO(data), max_rows=5,
                             control = np.array([[1., 1.], [2., 2.], [3., 3.], [4., 4.]]))
        assert_equal(test, control)
data = 'a b\nc d\n1 1\n2 2\n3 3\n4 4\n5 5'
txt = TextIO(data)
test = np.genfromtxt(txt, skip_header=1, max_rows=3, names=True)
control = np.array([(1.0, 1.0), (2.0, 2.0), (3.0, 3.0)],
                  dtype=[('c', '<f8'), ('d', '<f8')])
assert_equal(test, control)
test = np.genfromtxt(txt, max_rows=None, dtype=test.dtype)
control = np.array([(4.0, 4.0), (5.0, 5.0)],
                  dtype=[('c', '<f8'), ('d', '<f8')])
assert_equal(test, control)
def test_gft_using_filename(self):
    tgt = np.arange(6).reshape((2, 3))
    linesep = ('\n', '\r\n', '\r')
    for sep in linesep:
        data = '0 1 2' + sep + '3 4 5'
        with temppath() as name:
            with open(name, 'w') as f:
                f.write(data)
            res = np.genfromtxt(name)
            assert_array_equal(res, tgt)
def test_gft_from_gzip(self):
    wanted = np.arange(6).reshape((2, 3))
    linesep = ('\n', '\r\n', '\r')
    for sep in linesep:
        data = '0 1 2' + sep + '3 4 5'
        s = BytesIO()
        with gzip.GzipFile(fileobj=s, mode='w') as g:
            g.write(asbytes(data))
        with temppath(suffix='.gz2') as name:
            with open(name, 'w') as f:
                f.write(data)
            assert_array_equal(np.genfromtxt(name), wanted)
def test_gft_using_generator(self):
    def count():
        for i in range(10):
            yield asbytes("%d" % i)
    res = np.genfromtxt(count())
    assert_array_equal(res, np.arange(10))
def test_auto_dtype_largeint(self):
    data = TextIO('73786976294838206464 17179869184 1024')
    test = np.genfromtxt(data, dtype=None)
    assert_equal(test.dtype.names, ['f0', 'f1', 'f2'])
    assert_(test.dtype['f0'] == float)
    assert_(test.dtype['f1'] == np.int64)
    assert_(test.dtype['f2'] == np.int_)
    assert_allclose(test['f0'], 73786976294838206464.)
    assert_equal(test['f1'], 17179869184)
    assert_equal(test['f2'], 1024)
def test_unpack_float_data(self):
    txt = TextIO("1,2,3\n4,5,6\n7,8,9\n0.0,1.0,2.0")
    a, b, c = np.loadtxt(txt, delimiter=",", unpack=True)
    assert_array_equal(a, np.array([1.0, 4.0, 7.0, 0.0]))
    assert_array_equal(b, np.array([2.0, 5.0, 8.0, 1.0]))
    assert_array_equal(c, np.array([3.0, 6.0, 9.0, 2.0]))
def test_unpack_structured(self):
    txt = TextIO("M 21 72\nF 35 58")
    dt = {'names': ('a', 'b', 'c'), 'formats': ('S1', 'i4', 'f4')}

```

```

1983: (8)         a, b, c = np.genfromtxt(txt, dtype=dt, unpack=True)
1984: (8)         assert_equal(a.dtype, np.dtype('S1'))
1985: (8)         assert_equal(b.dtype, np.dtype('i4'))
1986: (8)         assert_equal(c.dtype, np.dtype('f4'))
1987: (8)         assert_array_equal(a, np.array([b'M', b'F']))
1988: (8)         assert_array_equal(b, np.array([21, 35]))
1989: (8)         assert_array_equal(c, np.array([72., 58.]))
1990: (4)         def test_unpack_auto_dtype(self):
1991: (8)             txt = TextIO("M 21 72.\nF 35 58.")
1992: (8)             expected = (np.array(["M", "F"]), np.array([21, 35]), np.array([72.,
58.]))
1993: (8)             test = np.genfromtxt(txt, dtype=None, unpack=True, encoding="utf-8")
1994: (8)             for arr, result in zip(expected, test):
1995: (12)                 assert_array_equal(arr, result)
1996: (12)                 assert_equal(arr.dtype, result.dtype)
1997: (4)         def test_unpack_single_name(self):
1998: (8)             txt = TextIO("21\n35")
1999: (8)             dt = {'names': ('a',), 'formats': ('i4',)}
2000: (8)             expected = np.array([21, 35], dtype=np.int32)
2001: (8)             test = np.genfromtxt(txt, dtype=dt, unpack=True)
2002: (8)             assert_array_equal(expected, test)
2003: (8)             assert_equal(expected.dtype, test.dtype)
2004: (4)         def test_squeeze_scalar(self):
2005: (8)             txt = TextIO("1")
2006: (8)             dt = {'names': ('a',), 'formats': ('i4',)}
2007: (8)             expected = np.array((1,), dtype=np.int32)
2008: (8)             test = np.genfromtxt(txt, dtype=dt, unpack=True)
2009: (8)             assert_array_equal(expected, test)
2010: (8)             assert_equal(((), test.shape),
2011: (8)                 assert_equal(expected.dtype, test.dtype)
2012: (4)             @pytest.mark.parametrize("ndim", [0, 1, 2])
2013: (4)         def test_ndmin_keyword(self, ndim: int):
2014: (8)             txt = "42"
2015: (8)             a = np.loadtxt(StringIO(txt), ndmin=ndim)
2016: (8)             b = np.genfromtxt(StringIO(txt), ndmin=ndim)
2017: (8)             assert_array_equal(a, b)
2018: (0)         class TestPathUsage:
2019: (4)             def test_loadtxt(self):
2020: (8)                 with temppath(suffix='.txt') as path:
2021: (12)                     path = Path(path)
2022: (12)                     a = np.array([[1.1, 2], [3, 4]])
2023: (12)                     np.savetxt(path, a)
2024: (12)                     x = np.loadtxt(path)
2025: (12)                     assert_array_equal(x, a)
2026: (4)             def test_save_load(self):
2027: (8)                 with temppath(suffix='.npy') as path:
2028: (12)                     path = Path(path)
2029: (12)                     a = np.array([[1, 2], [3, 4]], int)
2030: (12)                     np.save(path, a)
2031: (12)                     data = np.load(path)
2032: (12)                     assert_array_equal(data, a)
2033: (4)             def test_save_load_memmap(self):
2034: (8)                 with temppath(suffix='.npy') as path:
2035: (12)                     path = Path(path)
2036: (12)                     a = np.array([[1, 2], [3, 4]], int)
2037: (12)                     np.save(path, a)
2038: (12)                     data = np.load(path, mmap_mode='r')
2039: (12)                     assert_array_equal(data, a)
2040: (12)                     del data
2041: (12)                     if IS_PYPY:
2042: (16)                         break_cycles()
2043: (16)                         break_cycles()
2044: (4)             @pytest.mark.xfail(IS_WASM, reason="memmap doesn't work correctly")
2045: (4)             def test_save_load_memmap_readwrite(self):
2046: (8)                 with temppath(suffix='.npy') as path:
2047: (12)                     path = Path(path)
2048: (12)                     a = np.array([[1, 2], [3, 4]], int)
2049: (12)                     np.save(path, a)
2050: (12)                     b = np.load(path, mmap_mode='r+')

```

```

2051: (12)             a[0][0] = 5
2052: (12)             b[0][0] = 5
2053: (12)             del b # closes the file
2054: (12)             if IS_PYPY:
2055: (16)                 break_cycles()
2056: (16)                 break_cycles()
2057: (12)             data = np.load(path)
2058: (12)             assert_array_equal(data, a)
2059: (4)             def test_savez_load(self):
2060: (8)                 with temppath(suffix='.npz') as path:
2061: (12)                     path = Path(path)
2062: (12)                     np.savez(path, lab='place holder')
2063: (12)                     with np.load(path) as data:
2064: (16)                         assert_array_equal(data['lab'], 'place holder')
2065: (4)             def test_savez_compressed_load(self):
2066: (8)                 with temppath(suffix='.npz') as path:
2067: (12)                     path = Path(path)
2068: (12)                     np.savez_compressed(path, lab='place holder')
2069: (12)                     data = np.load(path)
2070: (12)                     assert_array_equal(data['lab'], 'place holder')
2071: (12)                     data.close()
2072: (4)             def test_genfromtxt(self):
2073: (8)                 with temppath(suffix='.txt') as path:
2074: (12)                     path = Path(path)
2075: (12)                     a = np.array([(1, 2), (3, 4)])
2076: (12)                     np.savetxt(path, a)
2077: (12)                     data = np.genfromtxt(path)
2078: (12)                     assert_array_equal(a, data)
2079: (4)             def test_recfromtxt(self):
2080: (8)                 with temppath(suffix='.txt') as path:
2081: (12)                     path = Path(path)
2082: (12)                     with path.open('w') as f:
2083: (16)                         f.write('A,B\n0,1\n2,3')
2084: (12)                     kwargs = dict(delimiter=",", missing_values="N/A", names=True)
2085: (12)                     test = np.recfromtxt(path, **kwargs)
2086: (12)                     control = np.array([(0, 1), (2, 3)],
2087: (31)                         dtype=[('A', int), ('B', int)])
2088: (12)                     assert_(isinstance(test, np.recarray))
2089: (12)                     assert_equal(test, control)
2090: (4)             def test_recfromcsv(self):
2091: (8)                 with temppath(suffix='.txt') as path:
2092: (12)                     path = Path(path)
2093: (12)                     with path.open('w') as f:
2094: (16)                         f.write('A,B\n0,1\n2,3')
2095: (12)                     kwargs = dict(missing_values="N/A", names=True,
2096: (12)                         case_sensitive=True)
2097: (12)                     test = np.recfromcsv(path, dtype=None, **kwargs)
2098: (31)                     control = np.array([(0, 1), (2, 3)],
2099: (12)                         dtype=[('A', int), ('B', int)])
2100: (12)                     assert_(isinstance(test, np.recarray))
2101: (12)                     assert_equal(test, control)
2102: (0)             def test_gzip_load():
2103: (4)                 a = np.random.random((5, 5))
2104: (4)                 s = BytesIO()
2105: (4)                 f = gzip.GzipFile(fileobj=s, mode="w")
2106: (4)                 np.save(f, a)
2107: (4)                 f.close()
2108: (4)                 s.seek(0)
2109: (4)                 f = gzip.GzipFile(fileobj=s, mode="r")
2110: (0)                 assert_array_equal(np.load(f), a)
2111: (4)             class JustWriter:
2112: (8)                 def __init__(self, base):
2113: (4)                     self.base = base
2114: (8)                 def write(self, s):
2115: (4)                     return self.base.write(s)
2116: (8)                 def flush(self):
2117: (0)                     return self.base.flush()
2118: (4)             class JustReader:

```

```

2119: (8)             self.base = base
2120: (4)             def read(self, n):
2121: (8)                 return self.base.read(n)
2122: (4)             def seek(self, off, whence=0):
2123: (8)                 return self.base.seek(off, whence)
2124: (0)             def test_ducktyping():
2125: (4)                 a = np.random.random((5, 5))
2126: (4)                 s = BytesIO()
2127: (4)                 f = JustWriter(s)
2128: (4)                 np.save(f, a)
2129: (4)                 f.flush()
2130: (4)                 s.seek(0)
2131: (4)                 f = JustReader(s)
2132: (4)                 assert_array_equal(np.load(f), a)
2133: (0)             def test_gzip_loadtxt():
2134: (4)                 s = BytesIO()
2135: (4)                 g = gzip.GzipFile(fileobj=s, mode='w')
2136: (4)                 g.write(b'1 2 3\n')
2137: (4)                 g.close()
2138: (4)                 s.seek(0)
2139: (4)                 with temppath(suffix='.gz') as name:
2140: (8)                     with open(name, 'wb') as f:
2141: (12)                         f.write(s.read())
2142: (8)                     res = np.loadtxt(name)
2143: (4)                     s.close()
2144: (4)                     assert_array_equal(res, [1, 2, 3])
2145: (0)             def test_gzip_loadtxt_from_string():
2146: (4)                 s = BytesIO()
2147: (4)                 f = gzip.GzipFile(fileobj=s, mode="w")
2148: (4)                 f.write(b'1 2 3\n')
2149: (4)                 f.close()
2150: (4)                 s.seek(0)
2151: (4)                 f = gzip.GzipFile(fileobj=s, mode="r")
2152: (4)                 assert_array_equal(np.loadtxt(f), [1, 2, 3])
2153: (0)             def test_npzfile_dict():
2154: (4)                 s = BytesIO()
2155: (4)                 x = np.zeros((3, 3))
2156: (4)                 y = np.zeros((3, 3))
2157: (4)                 np.savetxt(s, x=x, y=y)
2158: (4)                 s.seek(0)
2159: (4)                 z = np.load(s)
2160: (4)                 assert_('x' in z)
2161: (4)                 assert_('y' in z)
2162: (4)                 assert_('x' in z.keys())
2163: (4)                 assert_('y' in z.keys())
2164: (4)                 for f, a in z.items():
2165: (8)                     assert_(f in ['x', 'y'])
2166: (8)                     assert_equal(a.shape, (3, 3))
2167: (4)                 assert_(len(z.items()) == 2)
2168: (4)                 for f in z:
2169: (8)                     assert_(f in ['x', 'y'])
2170: (4)                 assert_('x' in z.keys())
2171: (0)             @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
2172: (0)             def test_load_refcount():
2173: (4)                 f = BytesIO()
2174: (4)                 np.savetxt(f, [1, 2, 3])
2175: (4)                 f.seek(0)
2176: (4)                 with assert_no_gc_cycles():
2177: (8)                     np.load(f)
2178: (4)                 f.seek(0)
2179: (4)                 dt = [("a", 'u1', 2), ("b", 'u1', 2)]
2180: (4)                 with assert_no_gc_cycles():
2181: (8)                     x = np.loadtxt(TextIO("0 1 2 3"), dtype=dt)
2182: (8)                     assert_equal(x, np.array([(0, 1), (2, 3)]], dtype=dt))
2183: (0)             def test_load_multiple_arrays_until_eof():
2184: (4)                 f = BytesIO()
2185: (4)                 np.savetxt(f, 1)
2186: (4)                 np.savetxt(f, 2)
2187: (4)                 f.seek(0)

```

```

2188: (4)         assert np.load(f) == 1
2189: (4)         assert np.load(f) == 2
2190: (4)         with pytest.raises(EOFError):
2191: (8)             np.load(f)
-----
```

## File 238 - test\_loadtxt.py:

```

1: (0) """
2: (0)     Tests specific to `np.loadtxt` added during the move of loadtxt to be backed
3: (0)     by C code.
4: (0)     These tests complement those found in `test_io.py` .
5: (0) """
6: (0)     import sys
7: (0)     import os
8: (0)     import pytest
9: (0)     from tempfile import NamedTemporaryFile, mkstemp
10: (0)    from io import StringIO
11: (0)    import numpy as np
12: (0)    from numpy.ma.testutils import assert_equal
13: (0)    from numpy.testing import assert_array_equal, HAS_REFCOUNT, IS_PYPY
14: (0)    def test_scientific_notation():
15: (4)        """Test that both 'e' and 'E' are parsed correctly."""
16: (4)        data = StringIO(
17: (8)            (
18: (12)                "1.0e-1,2.0E1,3.0\n"
19: (12)                "4.0e-2,5.0E-1,6.0\n"
20: (12)                "7.0e-3,8.0E1,9.0\n"
21: (12)                "0.0e-4,1.0E-1,2.0"
22: (8)            )
23: (4)        )
24: (4)        expected = np.array(
25: (8)            [[0.1, 20., 3.0], [0.04, 0.5, 6], [0.007, 80., 9], [0, 0.1, 2]]
26: (4)        )
27: (4)        assert_array_equal(np.loadtxt(data, delimiter=",", comments="#"), expected)
28: (0)    @pytest.mark.parametrize("comment", ["..", "//", "@-", "this is a comment"])
29: (0)    def test_comment_multiple_chars(comment):
30: (4)        content = "# IGNORE\n1.5, 2.5# ABC\n3.0,4.0# XXX\n5.5,6.0\n"
31: (4)        txt = StringIO(content.replace("#", comment))
32: (4)        a = np.loadtxt(txt, delimiter=",", comments=comment)
33: (4)        assert_equal(a, [[1.5, 2.5], [3.0, 4.0], [5.5, 6.0]])
34: (0)    @pytest.fixture
35: (0)    def mixed_types_structured():
36: (4)        """
37: (4)            Fixture providing heterogeneous input data with a structured dtype, along
38: (4)            with the associated structured array.
39: (4) """
40: (4)        data = StringIO(
41: (8)            (
42: (12)                "1000;2.4;alpha;-34\n"
43: (12)                "2000;3.1;beta;29\n"
44: (12)                "3500;9.9;gamma;120\n"
45: (12)                "4090;8.1;delta;0\n"
46: (12)                "5001;4.4;epsilon;-99\n"
47: (12)                "6543;7.8;omega;-1\n"
48: (8)            )
49: (4)        )
50: (4)        dtype = np.dtype(
51: (8)            [('f0', np.uint16), ('f1', np.float64), ('f2', 'S7'), ('f3', np.int8)]
52: (4)        )
53: (4)        expected = np.array(
54: (8)            [
55: (12)                (1000, 2.4, "alpha", -34),
56: (12)                (2000, 3.1, "beta", 29),
57: (12)                (3500, 9.9, "gamma", 120),
58: (12)                (4090, 8.1, "delta", 0),
59: (12)                (5001, 4.4, "epsilon", -99),
60: (12)                (6543, 7.8, "omega", -1)

```

```

61: (8)                ],
62: (8)                dtype=dtype
63: (4)
64: (4)            )
65: (0)        return data, dtype, expected
66: (0)    @pytest.mark.parametrize('skiprows', [0, 1, 2, 3])
67: (8)    def test_structured_dtype_and_skiprows_no_empty_lines(
68: (8)        skiprows, mixed_types_structured):
69: (4)            data, dtype, expected = mixed_types_structured
70: (4)            a = np.loadtxt(data, dtype=dtype, delimiter=";", skiprows=skiprows)
71: (0)        assert_array_equal(a, expected[skiprows:])
72: (4)    def test_unpack_structured(mixed_types_structured):
73: (4)        data, dtype, expected = mixed_types_structured
74: (4)        a, b, c, d = np.loadtxt(data, dtype=dtype, delimiter=";", unpack=True)
75: (4)        assert_array_equal(a, expected["f0"])
76: (4)        assert_array_equal(b, expected["f1"])
77: (4)        assert_array_equal(c, expected["f2"])
78: (4)        assert_array_equal(d, expected["f3"])
79: (0)    def test_structured_dtype_with_shape():
80: (4)        dtype = np.dtype([('a', 'u1', 2), ('b', 'u1', 2)])
81: (4)        data = StringIO("0,1,2,3\n6,7,8,9\n")
82: (4)        expected = np.array([(0, 1), (2, 3), (6, 7), (8, 9)]), dtype=dtype)
83: (0)        assert_array_equal(np.loadtxt(data, delimiter=",", dtype=dtype), expected)
84: (4)    def test_structured_dtype_with_multi_shape():
85: (4)        dtype = np.dtype([('a', 'u1', (2, 2))])
86: (4)        data = StringIO("0 1 2 3\n")
87: (4)        expected = np.array([(0, 1), (2, 3)]), dtype=dtype)
88: (0)        assert_array_equal(np.loadtxt(data, dtype=dtype), expected)
89: (4)    def test_nested_structured_subarray():
90: (4)        point = np.dtype([('x', float), ('y', float)])
91: (4)        dt = np.dtype([('code', int), ('points', point, (2,))])
92: (4)        data = StringIO("100,1,2,3,4\n200,5,6,7,8\n")
93: (8)        expected = np.array(
94: (12)            [
95: (12)                (100, [(1., 2.), (3., 4.)]),
96: (8)                (200, [(5., 6.), (7., 8.)]),
97: (8)            ],
98: (4)            dtype=dt
99: (4)        )
100: (0)        assert_array_equal(np.loadtxt(data, dtype=dt, delimiter=","), expected)
101: (4)    def test_structured_dtype_offsets():
102: (4)        dt = np.dtype("i1, i4, i1, i4, i1, i4", align=True)
103: (4)        data = StringIO("1,2,3,4,5,6\n7,8,9,10,11,12\n")
104: (4)        expected = np.array([(1, 2, 3, 4, 5, 6), (7, 8, 9, 10, 11, 12)]), dtype=dt)
105: (0)        assert_array_equal(np.loadtxt(data, delimiter=",", dtype=dt), expected)
106: (0)    @pytest.mark.parametrize("param", ("skiprows", "max_rows"))
107: (0)    def test_exception_negative_row_limits(param):
108: (4)        """skiprows and max_rows should raise for negative parameters."""
109: (8)        with pytest.raises(ValueError, match="argument must be nonnegative"):
110: (0)            np.loadtxt("foo.bar", **{param: -3})
111: (0)    @pytest.mark.parametrize("param", ("skiprows", "max_rows"))
112: (4)    def test_exception_noninteger_row_limits(param):
113: (8)        with pytest.raises(TypeError, match="argument must be an integer"):
114: (0)            np.loadtxt("foo.bar", **{param: 1.0})
115: (4)    @pytest.mark.parametrize(
116: (4)        "data, shape",
117: (8)        [
118: (8)            ("1 2 3 4 5\n", (1, 5)), # Single row
119: (4)            ("1\n2\n3\n4\n5\n", (5, 1)), # Single column
120: (0)        ]
121: (0)    def test_ndmin_single_row_or_col(data, shape):
122: (4)        arr = np.array([1, 2, 3, 4, 5])
123: (4)        arr2d = arr.reshape(shape)
124: (4)        assert_array_equal(np.loadtxt(StringIO(data), dtype=int), arr)
125: (4)        assert_array_equal(np.loadtxt(StringIO(data), dtype=int, ndmin=0), arr)
126: (4)        assert_array_equal(np.loadtxt(StringIO(data), dtype=int, ndmin=1), arr)
127: (4)        assert_array_equal(np.loadtxt(StringIO(data), dtype=int, ndmin=2), arr2d)
128: (0)    @pytest.mark.parametrize("badval", [-1, 3, None, "plate of shrimp"])
129: (0)    def test_bad_ndmin(badval):

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

130: (4)
131: (8)
132: (0)
133: (4)
134: (4)
135: (12)
136: (12)
137: (12)
138: (12)
139: (12)
140: (4)
141: (0)
142: (0)
143: (4)
144: (8)
145: (8)
146: (8)
147: (8)
148: (4)
149: (4)
150: (4)
151: (8)
152: (4)
153: (0)
154: (4)
155: (4)
156: (4)
157: (8)
158: (4)
159: (0)
160: (0)
161: (4)
162: (4)
163: (4)
164: (4)
165: (0)
166: (20)
167: (0)
168: (0)
169: (4)
170: (4)
171: (4)
172: (8)
173: (0)
174: (4)
175: (4)
176: (4)
177: (4)
178: (8)
179: (4)
180: (4)
181: (0)
182: (4)
183: (4)
184: (4)
185: (4)
186: (8)
187: (8)
188: (8)
189: (8)
190: (8)
191: (8)
192: (4)
193: (4)
194: (4)
195: (21)
196: (4)
197: (0)
198: (4)

    with pytest.raises(ValueError, match="Illegal value of ndmin keyword"):
        np.loadtxt("foo.bar", ndmin=badval)

@pytest.mark.parametrize(
    "ws",
    (
        " ", # space
        "\t", # tab
        "\u2003", # em
        "\u00A0", # non-break
        "\u3000", # ideographic space
    )
)
def test_blank_lines_spaces_delimit(ws):
    txt = StringIO(
        f"1 2{ws}30\n\n{ws}\n"
        f"4 5 60{ws}\n {ws} \n"
        f"7 8 {ws} 90\n# comment\n"
        f"3 2 1"
    )
    expected = np.array([[1, 2, 30], [4, 5, 60], [7, 8, 90], [3, 2, 1]])
    assert_equal(
        np.loadtxt(txt, dtype=int, delimiter=None, comments="#"), expected
    )

def test_blank_lines_normal_delimiter():
    txt = StringIO('1,2,30\n\n4,5,60\n\n7,8,90\n# comment\n3,2,1')
    expected = np.array([[1, 2, 30], [4, 5, 60], [7, 8, 90], [3, 2, 1]])
    assert_equal(
        np.loadtxt(txt, dtype=int, delimiter=',', comments="#"), expected
    )

@pytest.mark.parametrize("dtype", (float, object))
def test_maxrows_no_blank_lines(dtype):
    txt = StringIO("1.5,2.5\n3.0,4.0\n5.5,6.0")
    res = np.loadtxt(txt, dtype=dtype, delimiter=",", max_rows=2)
    assert_equal(res.dtype, dtype)
    assert_equal(res, np.array([["1.5", "2.5"], ["3.0", "4.0"]], dtype=dtype))

@pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
                    reason="PyPy bug in error formatting")
@pytest.mark.parametrize("dtype", (np.dtype("f8"), np.dtype("i2")))
def test_exception_message_bad_values(dtype):
    txt = StringIO("1,2\n3,XXX\n5,6")
    msg = f"could not convert string 'XXX' to {dtype} at row 1, column 2"
    with pytest.raises(ValueError, match=msg):
        np.loadtxt(txt, dtype=dtype, delimiter=",")

def test_converters_negative_indices():
    txt = StringIO('1.5,2.5\n3.0,XXX\n5.5,6.0')
    conv = {-1: lambda s: np.nan if s == 'XXX' else float(s)}
    expected = np.array([[1.5, 2.5], [3.0, np.nan], [5.5, 6.0]])
    res = np.loadtxt(
        txt, dtype=np.float64, delimiter=",", converters=conv, encoding=None
    )
    assert_equal(res, expected)

def test_converters_negative_indices_with_usecols():
    txt = StringIO('1.5,2.5,3.5\n3.0,4.0,XXX\n5.5,6.0,7.5\n')
    conv = {-1: lambda s: np.nan if s == 'XXX' else float(s)}
    expected = np.array([[1.5, 2.5, 3.5], [3.0, np.nan, 7.5]])
    res = np.loadtxt(
        txt,
        dtype=np.float64,
        delimiter=",",
        converters=conv,
        usecols=[0, -1],
        encoding=None,
    )
    assert_equal(res, expected)

res = np.loadtxt(StringIO('''0,1,2\n0,1,2,3,4'''), delimiter=",",
                usecols=[0, -1], converters={-1: (lambda x: -1)})

assert_array_equal(res, [[0, -1], [0, -1]])

def test_ragged_error():
    rows = ["1,2,3", "1,2,3", "4,3,2,1"]

```

```

199: (4)           with pytest.raises(ValueError,
200: (12)             match="the number of columns changed from 3 to 4 at row 3"):
201: (8)               np.loadtxt(rows, delimiter=",")
202: (0)
203: (4)       def test_ragged_usecols():
204: (4)           txt = StringIO("0,0,XXX\n0,XXX,0,XXX\n0,XXX,XXX,0,XXX\n")
205: (4)           expected = np.array([[0, 0], [0, 0], [0, 0]])
206: (4)           res = np.loadtxt(txt, dtype=float, delimiter=",", usecols=[0, -2])
207: (4)           assert_equal(res, expected)
208: (4)           txt = StringIO("0,0,XXX\n0\n0,XXX,XXX,0,XXX\n")
209: (16)          with pytest.raises(ValueError,
210: (8)              match="invalid column index -2 at row 2 with 1 columns"):
211: (0)                np.loadtxt(txt, dtype=float, delimiter=",", usecols=[0, -2])
212: (4)       def test_empty_usecols():
213: (4)           txt = StringIO("0,0,XXX\n0,XXX,0,XXX\n0,XXX,XXX,0,XXX\n")
214: (4)           res = np.loadtxt(txt, dtype=np.dtype([]), delimiter=",", usecols=[])
215: (4)           assert res.shape == (3,)
216: (4)           assert res.dtype == np.dtype([])
217: (0)         @pytest.mark.parametrize("c1", ["a", "ä®", "ð«•"])
218: (0)         @pytest.mark.parametrize("c2", ["a", "ä®", "ð«•"])
219: (4)       def test_large_unicode_characters(c1, c2):
220: (4)           txt = StringIO(f"a,{c1},c,1.0\n{c2},2.0,g")
221: (4)           res = np.loadtxt(txt, dtype=np.dtype('U12'), delimiter=",")
222: (4)           expected = np.array(
223: (8)               [f"a,{c1},c,1.0".split(","), f"{c2},2.0,g".split(",")],
224: (8)               dtype=np.dtype('U12')
225: (4)           )
226: (0)           assert_equal(res, expected)
227: (4)       def test_unicode_with_converter():
228: (4)           txt = StringIO("cat,dog\ní±í²í³,í¹íµí¶\nabc,def\n")
229: (4)           conv = {0: lambda s: s.upper()}
230: (4)           res = np.loadtxt(
231: (8)               txt,
232: (8)               dtype=np.dtype("U12"),
233: (8)               converters=conv,
234: (8)               delimiter=",",
235: (4)               encoding=None
236: (4)           )
237: (4)           expected = np.array([['CAT', 'dog'], ['í¹íµí¶', 'í¹íµí¶'], ['ABC',
238: ('def')]])
239: (4)           assert_equal(res, expected)
240: (0)       def test_converter_with_structured_dtype():
241: (4)           txt = StringIO('1.5,2.5,Abc\n3.0,4.0,dEf\n5.5,6.0,ghI\n')
242: (4)           dt = np.dtype([('m', np.int32), ('r', np.float32), ('code', 'U8')])
243: (4)           conv = {0: lambda s: int(10*float(s)), -1: lambda s: s.upper()}
244: (4)           res = np.loadtxt(txt, dtype=dt, delimiter=",", converters=conv)
245: (4)           expected = np.array(
246: (8)               [(15, 2.5, 'ABC'), (30, 4.0, 'DEF'), (55, 6.0, 'GHI')],
247: (4)               dtype=dt
248: (0)           )
249: (4)           assert_equal(res, expected)
250: (0)       def test_converter_with_unicode_dtype():
251: (4)           """
252: (4)               With the default 'bytes' encoding, tokens are encoded prior to being
253: (4)               passed to the converter. This means that the output of the converter may
254: (4)               be bytes instead of unicode as expected by `read_rows`.
255: (4)               This test checks that outputs from the above scenario are properly decoded
256: (4)               prior to parsing by `read_rows`.
257: (4)           """
258: (4)           txt = StringIO('abc,def\nrst,xyz')
259: (4)           conv = bytes.upper
260: (4)           res = np.loadtxt(
261: (12)             txt, dtype=np.dtype("U3"), converters=conv, delimiter=",")
262: (4)           expected = np.array([['ABC', 'DEF'], ['RST', 'XYZ']])
263: (4)           assert_equal(res, expected)
264: (0)       def test_read_huge_row():
265: (4)           row = "1.5, 2.5, " * 50000
266: (4)           row = row[:-1] + "\n"

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

267: (0) @pytest.mark.parametrize("dtype", "edfgFDG")
268: (0) def test_huge_float(dtype):
269: (4)     field = "0" * 1000 + ".123456789"
270: (4)     dtype = np.dtype(dtype)
271: (4)     value = np.loadtxt([field], dtype=dtype)[()]
272: (4)     assert value == dtype.type("0.123456789")
273: (0) @pytest.mark.parametrize(
274: (4)     ("given_dtype", "expected_dtype"),
275: (4)     [
276: (8)         ("S", np.dtype("S5")),
277: (8)         ("U", np.dtype("U5")),
278: (4)     ],
279: (0) )
280: (0) def test_string_no_length_given(given_dtype, expected_dtype):
281: (4)     """
282: (4)     The given dtype is just 'S' or 'U' with no length. In these cases, the
283: (4)     length of the resulting dtype is determined by the longest string found
284: (4)     in the file.
285: (4)     """
286: (4)     txt = StringIO("AAA,5-1\nBBBBB,0-3\nC,4-9\n")
287: (4)     res = np.loadtxt(txt, dtype=given_dtype, delimiter=",")
288: (4)     expected = np.array([
289: (8)         [['AAA', '5-1'], ['BBBBB', '0-3'], ['C', '4-9']]],
290: (4)         dtype=expected_dtype
291: (4)     )
292: (4)     assert_equal(res, expected)
293: (0)     assert_equal(res.dtype, expected_dtype)
294: (4) def test_float_conversion():
295: (4)     """
296: (4)     Some tests that the conversion to float64 works as accurately as the
297: (4)     Python built-in `float` function. In a naive version of the float parser,
298: (4)     these strings resulted in values that were off by an ULP or two.
299: (4)     """
300: (8)     strings = [
301: (8)         '0.9999999999999999',
302: (8)         '9876543210.123456',
303: (8)         '5.43215432154321e+300',
304: (8)         '0.901',
305: (4)         '0.333',
306: (4)     ]
307: (4)     txt = StringIO('\n'.join(strings))
308: (4)     res = np.loadtxt(txt)
309: (4)     expected = np.array([float(s) for s in strings])
310: (0)     assert_equal(res, expected)
311: (4) def test_bool():
312: (4)     txt = StringIO("1, 0\n10, -1")
313: (4)     res = np.loadtxt(txt, dtype=bool, delimiter=",")
314: (4)     assert res.dtype == bool
315: (4)     assert_array_equal(res, [[True, False], [True, True]])
316: (0)     assert_array_equal(res.view(np.uint8), [[1, 0], [1, 1]])
317: (20)     @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
318: (0)                         reason="PyPy bug in error formatting")
319: (0)     @pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
320: (0)     @pytest.mark.filterwarnings("error:.*integer via a
float.*:DeprecationWarning")
321: (4)         def test_integer_signs(dtype):
322: (4)             dtype = np.dtype(dtype)
323: (4)             assert np.loadtxt(["+2"], dtype=dtype) == 2
324: (4)             if dtype.kind == "u":
325: (8)                 with pytest.raises(ValueError):
326: (12)                     np.loadtxt(["-1\n"], dtype=dtype)
327: (4)             else:
328: (8)                 assert np.loadtxt(["-2\n"], dtype=dtype) == -2
329: (4)             for sign in ["++", "+-", "--", "-+"]:
330: (8)                 with pytest.raises(ValueError):
331: (12)                     np.loadtxt([f"{sign}2\n"], dtype=dtype)
332: (0)             @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
333: (0)                         reason="PyPy bug in error formatting")
334: (0)             @pytest.mark.parametrize("dtype", np.typecodes["AllInteger"])
335: (0)             @pytest.mark.filterwarnings("error:.*integer via a

```

```

float.*:DeprecationWarning")
335: (0)        def test_implicit_cast_float_to_int_fails(dtype):
336: (4)            txt = StringIO("1.0, 2.1, 3.7\n4, 5, 6")
337: (4)            with pytest.raises(ValueError):
338: (8)                np.loadtxt(txt, dtype=dtype, delimiter=",")
339: (0)        @pytest.mark.parametrize("dtype", (np.complex64, np.complex128))
340: (0)        @pytest.mark.parametrize("with_parens", (False, True))
341: (0)        def test_complex_parsing(dtype, with_parens):
342: (4)            s = "(1.0-2.5j),3.75,(7+-5.0j)\n(4),(-19e2j),(0)"
343: (4)            if not with_parens:
344: (8)                s = s.replace("(", "").replace(")", "")
345: (4)            res = np.loadtxt(StringIO(s), dtype=dtype, delimiter=",")
346: (4)            expected = np.array(
347: (8)                [[1.0-2.5j, 3.75, 7-5j], [4.0, -1900j, 0]], dtype=dtype
348: (4)
349: (4)            assert_equal(res, expected)
350: (0)        def test_read_from_generator():
351: (4)            def gen():
352: (8)                for i in range(4):
353: (12)                    yield f"{i},{2*i},{i**2}"
354: (4)            res = np.loadtxt(gen(), dtype=int, delimiter=",")
355: (4)            expected = np.array([[0, 0, 0], [1, 2, 1], [2, 4, 4], [3, 6, 9]])
356: (4)            assert_equal(res, expected)
357: (0)        def test_read_from_generator_multitype():
358: (4)            def gen():
359: (8)                for i in range(3):
360: (12)                    yield f"{i} {i / 4}"
361: (4)            res = np.loadtxt(gen(), dtype="i, d", delimiter=" ")
362: (4)            expected = np.array([(0, 0.0), (1, 0.25), (2, 0.5)], dtype="i, d")
363: (4)            assert_equal(res, expected)
364: (0)        def test_read_from_bad_generator():
365: (4)            def gen():
366: (8)                for entry in ["1,2", b"3, 5", 12738]:
367: (12)                    yield entry
368: (4)            with pytest.raises(
369: (12)                TypeError, match=r"non-string returned while reading data"):
370: (8)                np.loadtxt(gen(), dtype="i, i", delimiter=",")
371: (0)        @pytest.mark.skipif(not HAS_REFCOUNT, reason="Python lacks refcounts")
372: (0)        def test_object_cleanup_on_read_error():
373: (4)            sentinel = object()
374: (4)            already_read = 0
375: (4)            def conv(x):
376: (8)                nonlocal already_read
377: (8)                if already_read > 4999:
378: (12)                    raise ValueError("failed half-way through!")
379: (8)
380: (8)
381: (4)            already_read += 1
382: (4)            return sentinel
383: (8)
384: (4)
385: (0)        @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
386: (20)                      reason="PyPy bug in error formatting")
387: (0)        def test_character_not_bytes_compatible():
388: (4)            """Test exception when a character cannot be encoded as 'S'."""
389: (4)            data = StringIO("\u2013") # == \u2013
390: (4)            with pytest.raises(ValueError):
391: (8)                np.loadtxt(data, dtype="S5")
392: (0)        @pytest.mark.parametrize("conv", (0, [float], ""))
393: (0)        def test_invalid_converter(conv):
394: (4)            msg = (
395: (8)                "converters must be a dictionary mapping columns to converter "
396: (8)                "functions or a single callable."
397: (4)
398: (4)            with pytest.raises(TypeError, match=msg):
399: (8)                np.loadtxt(StringIO("1 2\n3 4"), converters=conv)
400: (0)        @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
401: (20)                      reason="PyPy bug in error formatting")
402: (0)        def test_converters_dict_raises_non_integer_key():

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

403: (4)
404: (8)
405: (4)
406: (8)
407: (0)
408: (0)
409: (4)
410: (4)
411: (8)
412: (0)
413: (4)
414: (16)
415: (8)
416: (0)
417: (0)
418: (4)
419: (8)
420: (4)
421: (4)
422: (4)
423: (8)
424: (4)
425: (4)
426: (4)
427: (0)
428: (0)
429: (4)
430: (8)
431: (4)
432: (4)
433: (4)
434: (8)
435: (4)
436: (4)
437: (4)
438: (0)
439: (4)
440: (4)
441: (4)
442: (4)
443: (12)
444: (8)
445: (4)
446: (4)
447: (4)
448: (4)
449: (0)
450: (20)
451: (0)
452: (4)
453: (4)
454: (4)
455: (8)
456: (0)
457: (4)
458: (4)
459: (8)
460: (8)
461: (4)
462: (4)
463: (8)
464: (4)
465: (8)
466: (4)
467: (4)
468: (0)
469: (4)
470: (8)
471: (12)

        with pytest.raises(TypeError, match="keys of the converters dict"):
            np.loadtxt(StringIO("1 2\n3 4"), converters={"a": int})
        with pytest.raises(TypeError, match="keys of the converters dict"):
            np.loadtxt(StringIO("1 2\n3 4"), converters={"a": int}, usecols=0)
    @pytest.mark.parametrize("bad_col_ind", (3, -3))
    def test_converters_dict_raises_non_col_key(bad_col_ind):
        data = StringIO("1 2\n3 4")
        with pytest.raises(ValueError, match="converter specified for column"):
            np.loadtxt(data, converters={bad_col_ind: int})
    def test_converters_dict_raises_val_not_callable():
        with pytest.raises(TypeError,
                           match="values of the converters dictionary must be callable"):
            np.loadtxt(StringIO("1 2\n3 4"), converters={0: 1})
    @pytest.mark.parametrize("q", ('"', "'", '`'))
    def test_quoted_field(q):
        txt = StringIO(
            f'{q}alpha, x{q}, 2.5\n{q}beta, y{q}, 4.5\n{q}gamma, z{q}, 5.0\n'
        )
        dtype = np.dtype([('f0', 'U8'), ('f1', np.float64)])
        expected = np.array([
            ("alpha, x", 2.5), ("beta, y", 4.5), ("gamma, z", 5.0)], dtype=dtype)
        res = np.loadtxt(txt, dtype=dtype, delimiter=",", quotechar=q)
        assert_array_equal(res, expected)
    @pytest.mark.parametrize("q", ('"', "'", '`'))
    def test_quoted_field_with_whitespace_delimiter(q):
        txt = StringIO(
            f'{q}alpha, x{q}    2.5\n{q}beta, y{q} 4.5\n{q}gamma, z{q}  5.0\n'
        )
        dtype = np.dtype([('f0', 'U8'), ('f1', np.float64)])
        expected = np.array([
            ("alpha, x", 2.5), ("beta, y", 4.5), ("gamma, z", 5.0)], dtype=dtype)
        res = np.loadtxt(txt, dtype=dtype, delimiter=None, quotechar=q)
        assert_array_equal(res, expected)
    def test_quote_support_default():
        """Support for quoted fields is disabled by default."""
        txt = StringIO('lat,long', 45, 30\n')
        dtype = np.dtype([('f0', 'U24'), ('f1', np.float64), ('f2', np.float64)])
        with pytest.raises(ValueError,
                           match="the dtype passed requires 3 columns but 4 were"):
            np.loadtxt(txt, dtype=dtype, delimiter=",")
        txt.seek(0)
        expected = np.array([('lat,long', 45., 30.)], dtype=dtype)
        res = np.loadtxt(txt, dtype=dtype, delimiter=",", quotechar='''')
        assert_array_equal(res, expected)
    @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
                       reason="PyPy bug in error formatting")
    def test_quotechar_multichar_error():
        txt = StringIO("1,2\n3,4")
        msg = r".*must be a single unicode character or None"
        with pytest.raises(TypeError, match=msg):
            np.loadtxt(txt, delimiter=",", quotechar="''")
    def test_comment_multichar_error_with_quote():
        txt = StringIO("1,2\n3,4")
        msg = (
            "when multiple comments or a multi-character comment is given, "
            "quotes are not supported."
        )
        with pytest.raises(ValueError, match=msg):
            np.loadtxt(txt, delimiter=",", comments="123", quotechar="''")
        with pytest.raises(ValueError, match=msg):
            np.loadtxt(txt, delimiter=",", comments=[#, %], quotechar="''")
        res = np.loadtxt(txt, delimiter=",", comments=(#, ), quotechar="''")
        assert_equal(res, [[1, 2], [3, 4]])
    def test_structured_dtype_with_quotes():
        data = StringIO(
            (
                "1000;2.4;'alpha';-34\n"

```

```

472: (12)                                "2000;3.1;'beta';29\n"
473: (12)                                "3500;9.9;'gamma';120\n"
474: (12)                                "4090;8.1;'delta';0\n"
475: (12)                                "5001;4.4;'epsilon';-99\n"
476: (12)                                "6543;7.8;'omega';-1\n"
477: (8)                                 )
478: (4)                                 )
479: (4)      dtype = np.dtype(
480: (8)        [ ('f0', np.uint16), ('f1', np.float64), ('f2', 'S7'), ('f3', np.int8) ]
481: (4)      )
482: (4)      expected = np.array(
483: (8)        [
484: (12)          (1000, 2.4, "alpha", -34),
485: (12)          (2000, 3.1, "beta", 29),
486: (12)          (3500, 9.9, "gamma", 120),
487: (12)          (4090, 8.1, "delta", 0),
488: (12)          (5001, 4.4, "epsilon", -99),
489: (12)          (6543, 7.8, "omega", -1)
490: (8)        ],
491: (8)        dtype=dtype
492: (4)      )
493: (4)      res = np.loadtxt(data, dtype=dtype, delimiter=";", quotechar="")
494: (4)      assert_array_equal(res, expected)
495: (0)      def test_quoted_field_is_not_empty():
496: (4)        txt = StringIO('1\n\n"4"\n"')
497: (4)        expected = np.array(["1", "4", ""], dtype="U1")
498: (4)        res = np.loadtxt(txt, delimiter=";", dtype="U1", quotechar='')
499: (4)        assert_equal(res, expected)
500: (0)      def test_quoted_field_is_not_empty_nonstrict():
501: (4)        txt = StringIO('1\n\n"4"\n"')
502: (4)        expected = np.array(["1", "4", ""], dtype="U1")
503: (4)        res = np.loadtxt(txt, delimiter=";", dtype="U1", quotechar='')
504: (4)        assert_equal(res, expected)
505: (0)      def test_consecutive_quotechar_escaped():
506: (4)        txt = StringIO('Hello, my name is ""Monty""!')
507: (4)        expected = np.array('Hello, my name is "Monty"!', dtype="U40")
508: (4)        res = np.loadtxt(txt, dtype="U40", delimiter=";", quotechar='')
509: (4)        assert_equal(res, expected)
510: (0)      @pytest.mark.parametrize("data", ("", "\n\n\n", "# 1 2 3\n# 4 5 6\n"))
511: (0)      @pytest.mark.parametrize("ndmin", (0, 1, 2))
512: (0)      @pytest.mark.parametrize("usecols", [None, (1, 2, 3)])
513: (0)      def test_warn_on_no_data(data, ndmin, usecols):
514: (4)        """Check that a UserWarning is emitted when no data is read from input."""
515: (4)        if usecols is not None:
516: (8)          expected_shape = (0, 3)
517: (4)        elif ndmin == 2:
518: (8)          expected_shape = (0, 1) # guess a single column?!
519: (4)        else:
520: (8)          expected_shape = (0,)
521: (4)        txt = StringIO(data)
522: (4)        with pytest.warns(UserWarning, match="input contained no data"):
523: (8)          res = np.loadtxt(txt, ndmin=ndmin, usecols=usecols)
524: (4)        assert res.shape == expected_shape
525: (4)        with NamedTemporaryFile(mode="w") as fh:
526: (8)          fh.write(data)
527: (8)          fh.seek(0)
528: (8)          with pytest.warns(UserWarning, match="input contained no data"):
529: (12)            res = np.loadtxt(txt, ndmin=ndmin, usecols=usecols)
530: (8)            assert res.shape == expected_shape
531: (0)      @pytest.mark.parametrize("skiprows", (2, 3))
532: (0)      def test_warn_on_skipped_data(skiprows):
533: (4)        data = "1 2 3\n4 5 6"
534: (4)        txt = StringIO(data)
535: (4)        with pytest.warns(UserWarning, match="input contained no data"):
536: (8)          np.loadtxt(txt, skiprows=skiprows)
537: (0)      @pytest.mark.parametrize(["dtype", "value"], [
538: (8)        ("i2", 0x0001), ("u2", 0x0001),
539: (8)        ("i4", 0x00010203), ("u4", 0x00010203),
540: (8)        ("i8", 0x0001020304050607), ("u8", 0x0001020304050607),

```

```

541: (8) ("float16", 3.07e-05),
542: (8) ("float32", 9.2557e-41), ("complex64", 9.2557e-41+2.8622554e-29j),
543: (8) ("float64", -1.758571353180402e-24),
544: (8) ("complex128", repr(5.406409232372729e-29-1.758571353180402e-24j)),
545: (8) ("longdouble", 0x01020304050607),
546: (8) ("clongdouble", repr(0x01020304050607 + (0x00121314151617 * 1j))), ("U2", "\U00010203\U000a0b0c")])
547: (8)
548: (0) @pytest.mark.parametrize("swap", [True, False])
549: (0) def test_byteswapping_and_unaligned(dtype, value, swap):
550: (4)     dtype = np.dtype(dtype)
551: (4)     data = [f"x,{value}\n"] # repr as PyPy `str` truncates some
552: (4)     if swap:
553: (8)         dtype = dtype.newbyteorder()
554: (4)     full_dt = np.dtype([(("a", "S1"), ("b", dtype)], align=False)
555: (4)     assert full_dt.fields["b"][1] == 1
556: (4)     res = np.loadtxt(data, dtype=full_dt, delimiter=",", encoding=None,
557: (21)             max_rows=1) # max-rows prevents over-allocation
558: (4)     assert res["b"] == dtype.type(value)
559: (0) @pytest.mark.parametrize("dtype",
560: (8)     np.typecodes["AllInteger"] + "efdFD" + "?")
561: (0) def test_unicode_whitespace_striping(dtype):
562: (4)     txt = StringIO(' 3 ,"\u202F2\n"')
563: (4)     res = np.loadtxt(txt, dtype=dtype, delimiter=",", quotechar='''')
564: (4)     assert_array_equal(res, np.array([3, 2]).astype(dtype))
565: (0) @pytest.mark.parametrize("dtype", "FD")
566: (0) def test_unicode_whitespace_striping_complex(dtype):
567: (4)     line = " 1 , 2+3j , ( 4+5j ), ( 6+-7j ) , 8j , ( 9j ) \n"
568: (4)     data = [line, line.replace(" ", "\u202F")]
569: (4)     res = np.loadtxt(data, dtype=dtype, delimiter=',')
570: (4)     assert_array_equal(res, np.array([[1, 2+3j, 4+5j, 6-7j, 8j, 9j]] * 2))
571: (0) @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
572: (20)             reason="PyPy bug in error formatting")
573: (0) @pytest.mark.parametrize("dtype", "FD")
574: (0) @pytest.mark.parametrize("field",
575: (8)     ["1 +2j", "1+ 2j", "1+2 j", "1+-+3", "(1j", "(1", "(1+2j", "1+2j)"])
576: (0) def test_bad_complex(dtype, field):
577: (4)     with pytest.raises(ValueError):
578: (8)         np.loadtxt([field + "\n"], dtype=dtype, delimiter=",")
579: (0) @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
580: (20)             reason="PyPy bug in error formatting")
581: (0) @pytest.mark.parametrize("dtype",
582: (12)     np.typecodes["AllInteger"] + "efgdFDG" + "?")
583: (0) def test_nul_character_error(dtype):
584: (4)     if dtype.lower() == "g":
585: (8)         pytest.xfail("longdouble/clongdouble assignment may misbehave.")
586: (4)     with pytest.raises(ValueError):
587: (8)         np.loadtxt(["1\000"], dtype=dtype, delimiter=",", quotechar='''')
588: (0) @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
589: (20)             reason="PyPy bug in error formatting")
590: (0) @pytest.mark.parametrize("dtype",
591: (8)     np.typecodes["AllInteger"] + "efgdFDG" + "?")
592: (0) def test_no_thousands_support(dtype):
593: (4)     if dtype == "e":
594: (8)         pytest.skip("half assignment currently uses Python float converter")
595: (4)     if dtype in "eG":
596: (8)         pytest.xfail("clongdouble assignment is buggy (uses `complex`?).")
597: (4)     assert int("1_1") == float("1_1") == complex("1_1") == 11
598: (4)     with pytest.raises(ValueError):
599: (8)         np.loadtxt(["1_1\n"], dtype=dtype)
600: (0) @pytest.mark.parametrize("data", [
601: (4)     ["1,2\n", "2\n,3\n"],
602: (4)     ["1,2\n", "2\r,3\n"]])
603: (0) def test_bad_newline_in_iterator(data):
604: (4)     msg = "Found an unquoted embedded newline within a single line"
605: (4)     with pytest.raises(ValueError, match=msg):
606: (8)         np.loadtxt(data, delimiter=",")
607: (0) @pytest.mark.parametrize("data", [
608: (4)     ["1,2\n", "2,3\r\n"], # a universal newline
609: (4)     ["1,2\n", "'2\n',3\n"], # a quoted newline

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

610: (4)
611: (4)
612: (0)
613: (0)
614: (4)
615: (4)
616: (0)
617: (0)
618: (4)
619: (4)
620: (4)
621: (4)
622: (0)
623: (4)
624: (4)
625: (4)
626: (21)
627: (4)
628: (0)
629: (4)
630: (8)
631: (12)
632: (8)
633: (12)
634: (16)
635: (12)
636: (4)
637: (8)
638: (0)
639: (4)
640: (8)
641: (12)
642: (16)
643: (4)
644: (8)
645: (12)
646: (12)
647: (16)
648: (16)
649: (20)
650: (16)
651: (8)
652: (12)
653: (16)
654: (4)
655: (8)
656: (12)
657: (12)
658: (16)
659: (8)
660: (20)
661: (12)
662: (16)
663: (4)
664: (8)
665: (20)
666: (12)
667: (16)
668: (4)
669: (8)
670: (12)
671: (16)
672: (4)
673: (8)
674: (12)
675: (16)
676: (4)
677: (4)
678: (8)

        ["1,2\n", "'2\r',3\n"],  

        ["1,2\n", "'2\r\n',3\n"],  

    ])  

def test_good_newline_in_iterator(data):  

    res = np.loadtxt(data, delimiter=",", quotechar="')")  

    assert_array_equal(res, [[1., 2.], [2., 3.]])  

@pytest.mark.parametrize("newline", ["\n", "\r", "\r\n"])
def test_universal_newlines_quoted(newline):  

    data = ['1,"2\n"\n', '3,"4\n", "1"\n']  

    data = [row.replace("\n", newline) for row in data]  

    res = np.loadtxt(data, dtype=object, delimiter=",", quotechar="')")  

    assert_array_equal(res, [['1', f'2{newline}'], ['3', f'4{newline}1']])  

def test_null_character():  

    res = np.loadtxt(["1\0002\0003\n", "4\0005\0006"], delimiter="\000")  

    assert_array_equal(res, [[1, 2, 3], [4, 5, 6]])  

    res = np.loadtxt(["1\000,2\000,3\n", "4\000,5\000,6"],  

                    delimiter=",", dtype=object)  

    assert res.tolist() == [[1\000, 2\000, 3], [4\000, 5\000, 6]]  

def test_iterator_fails_getting_next_line():  

    class BadSequence:  

        def __len__(self):  

            return 100  

        def __getitem__(self, item):  

            if item == 50:  

                raise RuntimeError("Bad things happened!")  

            return f"{item}, {item+1}"  

    with pytest.raises(RuntimeError, match="Bad things happened!"):
        np.loadtxt(BadSequence(), dtype=int, delimiter=",")  

class TestCReaderUnitTests:  

    def test_not_an_filelike(self):  

        with pytest.raises(AttributeError, match=".*read"):  

            np.core._multiarray_umath._load_from_filelike(  

                object(), dtype=np.dtype("i"), filelike=True)  

    def test_filelike_read_fails(self):  

        class BadFileLike:  

            counter = 0  

            def read(self, size):  

                self.counter += 1  

                if self.counter > 20:  

                    raise RuntimeError("Bad bad bad!")  

                return "1,2,3\n"  

        with pytest.raises(RuntimeError, match="Bad bad bad!"):
            np.core._multiarray_umath._load_from_filelike(  

                BadFileLike(), dtype=np.dtype("i"), filelike=True)  

    def test_filelike_bad_read(self):  

        class BadFileLike:  

            counter = 0  

            def read(self, size):  

                return 1234 # not a string!  

        with pytest.raises(TypeError,
                           match="non-string returned while reading data"):
            np.core._multiarray_umath._load_from_filelike(  

                BadFileLike(), dtype=np.dtype("i"), filelike=True)  

    def test_not_an_iter(self):  

        with pytest.raises(TypeError,
                           match="error reading from object, expected an iterable"):  

            np.core._multiarray_umath._load_from_filelike(  

                object(), dtype=np.dtype("i"), filelike=False)  

    def test_bad_type(self):  

        with pytest.raises(TypeError, match="internal error: dtype must"):  

            np.core._multiarray_umath._load_from_filelike(  

                object(), dtype="i", filelike=False)  

    def test_bad_encoding(self):  

        with pytest.raises(TypeError, match="encoding must be a unicode"):  

            np.core._multiarray_umath._load_from_filelike(  

                object(), dtype=np.dtype("i"), filelike=False, encoding=123)  

@pytest.mark.parametrize("newline", ["\r", "\n", "\r\n"])
def test_manual_universal_newlines(self, newline):  

    data = StringIO('0\n1\n"2\n"\n3\n4 #\n'.replace("\n", newline),

```

```

679: (24)
680: (8)             newline="")
681: (12)            res = np.core._multiarray_umath._load_from_filelike(
682: (12)              data, dtype=np.dtype("U10"), filelike=True,
683: (8)                quote='', comment="#", skipnlines=1)
684: (0)              assert_array_equal(res[:, 0], ["1", f"2{newline}", "3", "4"])
def test_delimiter_comment_collision_raises():
685: (4)      with pytest.raises(TypeError, match=".control characters.*incompatible"):
686: (8)          np.loadtxt(StringIO("1, 2, 3"), delimiter=",", comments=",")
def test_delimiter_quotechar_collision_raises():
688: (4)      with pytest.raises(TypeError, match=".control characters.*incompatible"):
689: (8)          np.loadtxt(StringIO("1, 2, 3"), delimiter=",", quotechar=",")
def test_comment_quotechar_collision_raises():
691: (4)      with pytest.raises(TypeError, match=".control characters.*incompatible"):
692: (8)          np.loadtxt(StringIO("1 2 3"), comments="#", quotechar="#")
def test_delimiter_and_multiple_comments_collision_raises():
694: (4)      with pytest.raises(
695: (8)          TypeError, match="Comment characters.*cannot include the delimiter"
696: (4)      ):
697: (8)          np.loadtxt(StringIO("1, 2, 3"), delimiter=",", comments=[#, , ])
@pytest.mark.parametrize(
698: (0)
699: (4)      "ws",
700: (4)      (
701: (8)          " ", # space
702: (8)          "\t", # tab
703: (8)          "\u2003", # em
704: (8)          "\u00A0", # non-break
705: (8)          "\u3000", # ideographic space
706: (4)
707: (0)
708: (0)      def test_collision_with_default_delimiter_raises(ws):
709: (4)          with pytest.raises(TypeError, match=".control characters.*incompatible"):
710: (8)              np.loadtxt(StringIO(f"1{ws}2{ws}3{n4{ws}6\n"), comments=ws)
711: (4)          with pytest.raises(TypeError, match=".control characters.*incompatible"):
712: (8)              np.loadtxt(StringIO(f"1{ws}2{ws}3{n4{ws}5{ws}6\n"), quotechar=ws)
@pytest.mark.parametrize("nl", ("\n", "\r"))
def test_control_character_newline_raises(nl):
714: (0)      def test_control_character_newline_raises(nl):
715: (4)          txt = StringIO(f"1{nl}2{nl}3{nl}{nl}4{nl}5{nl}6{nl}{nl}")
716: (4)          msg = "control character.*cannot be a newline"
717: (4)          with pytest.raises(TypeError, match=msg):
718: (8)              np.loadtxt(txt, delimiter=nl)
719: (4)          with pytest.raises(TypeError, match=msg):
720: (8)              np.loadtxt(txt, comments=nl)
721: (4)          with pytest.raises(TypeError, match=msg):
722: (8)              np.loadtxt(txt, quotechar=nl)
@pytest.mark.parametrize(
723: (0)
724: (4)      ("generic_data", "long_datum", "unitless_dtype", "expected_dtype"),
725: (4)      [
726: (8)          ("2012-03", "2013-01-15", "M8", "M8[D]"), # Datetimes
727: (8)          ("spam-a-lot", "tis_but_a_scratch", "U", "U17"), # str
728: (4)
729: (0)
730: (0)      @pytest.mark.parametrize("nrows", (10, 50000, 60000)) # lt, eq, gt chunksizes
def test_parametric_unit_discovery(
731: (0)          generic_data, long_datum, unitless_dtype, expected_dtype, nrows
732: (4)
733: (0)
734: (4)          """Check that the correct unit (e.g. month, day, second) is discovered
from
735: (4)              the data when a user specifies a unitless datetime."""
736: (4)          data = [generic_data] * 50000 + [long_datum]
737: (4)          expected = np.array(data, dtype=expected_dtype)
738: (4)          txt = StringIO("\n".join(data))
739: (4)          a = np.loadtxt(txt, dtype=unitless_dtype)
740: (4)          assert a.dtype == expected.dtype
741: (4)          assert_equal(a, expected)
742: (4)          fd, fname = mkstemp()
743: (4)          os.close(fd)
744: (4)          with open(fname, "w") as fh:
745: (8)              fh.write("\n".join(data))
746: (4)          a = np.loadtxt(fname, dtype=unitless_dtype)

```

```

747: (4)             os.remove(fname)
748: (4)             assert a.dtype == expected.dtype
749: (4)             assert_equal(a, expected)
750: (0)             def test_str_dtype_unit_discovery_with_converter():
751: (4)                 data = ["spam-a-lot"] * 60000 + ["XXXtis_but_a_scratch"]
752: (4)                 expected = np.array(
753: (8)                     ["spam-a-lot"] * 60000 + ["tis_but_a_scratch"], dtype="U17"
754: (4)                 )
755: (4)                 conv = lambda s: s.strip("XXX")
756: (4)                 txt = StringIO("\n".join(data))
757: (4)                 a = np.loadtxt(txt, dtype="U", converters=conv, encoding=None)
758: (4)                 assert a.dtype == expected.dtype
759: (4)                 assert_equal(a, expected)
760: (4)                 fd, fname = mkstemp()
761: (4)                 os.close(fd)
762: (4)                 with open(fname, "w") as fh:
763: (8)                     fh.write("\n".join(data))
764: (4)                     a = np.loadtxt(fname, dtype="U", converters=conv, encoding=None)
765: (4)                     os.remove(fname)
766: (4)                     assert a.dtype == expected.dtype
767: (4)                     assert_equal(a, expected)
768: (0)             @pytest.mark.skipif(IS_PYPY and sys.implementation.version <= (7, 3, 8),
769: (20)                           reason="PyPy bug in error formatting")
770: (0)             def test_control_character_empty():
771: (4)                 with pytest.raises(TypeError, match="Text reading control character
must"):
772: (8)                     np.loadtxt(StringIO("1 2 3"), delimiter="")
773: (4)                     with pytest.raises(TypeError, match="Text reading control character
must"):
774: (8)                         np.loadtxt(StringIO("1 2 3"), quotechar="")
775: (4)                         with pytest.raises(ValueError, match="comments cannot be an empty
string"):
776: (8)                             np.loadtxt(StringIO("1 2 3"), comments="")
777: (4)                             with pytest.raises(ValueError, match="comments cannot be an empty
string"):
778: (8)                                 np.loadtxt(StringIO("1 2 3"), comments=["#", ""])
779: (0)             def test_control_characters_as_bytes():
780: (4)                 """Byte control characters (comments, delimiter) are supported."""
781: (4)                 a = np.loadtxt(StringIO("#header\n1,2,3"), comments=b"#", delimiter=b",")
782: (4)                 assert_equal(a, [1, 2, 3])
783: (0)             @pytest.mark.filterwarnings('ignore::UserWarning')
784: (0)             def test_field_growing_cases():
785: (4)                 res = np.loadtxt([""], delimiter=",", dtype=bytes)
786: (4)                 assert len(res) == 0
787: (4)                 for i in range(1, 1024):
788: (8)                     res = np.loadtxt(["," * i], delimiter=",", dtype=bytes)
789: (8)                     assert len(res) == i+1

```

---

## File 239 - test\_mixins.py:

```

1: (0)             import numbers
2: (0)             import operator
3: (0)             import numpy as np
4: (0)             from numpy.testing import assert_, assert_equal, assert_raises
5: (0)             class ArrayLike(np.lib.mixins.NDArrayOperatorsMixin):
6: (4)                 def __init__(self, value):
7: (8)                     self.value = np.asarray(value)
8: (4)                     _HANDLED_TYPES = (np.ndarray, numbers.Number)
9: (4)                     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
10: (8)                         out = kwargs.get('out', ())
11: (8)                         for x in inputs + out:
12: (12)                             if not isinstance(x, self._HANDLED_TYPES + (ArrayLike,)):
13: (16)                                 return NotImplemented
14: (8)                         inputs = tuple(x.value if isinstance(x, ArrayLike) else x
15: (23)                                         for x in inputs)
16: (8)                         if out:
17: (12)                             kwargs['out'] = tuple(

```

```

18: (16)                     x.value if isinstance(x, ArrayLike) else x
19: (16)                     for x in out)
20: (8)                      result = getattr(ufunc, method)(*inputs, **kwargs)
21: (8)                      if type(result) is tuple:
22: (12)                        return tuple(type(self)(x) for x in result)
23: (8)                      elif method == 'at':
24: (12)                        return None
25: (8)                      else:
26: (12)                        return type(self)(result)
27: (4)                      def __repr__(self):
28: (8)                        return '%s(%r)' % (type(self).__name__, self.value)
29: (0)                      def wrap_array_like(result):
30: (4)                        if type(result) is tuple:
31: (8)                          return tuple(ArrayLike(r) for r in result)
32: (4)                        else:
33: (8)                          return ArrayLike(result)
34: (0)                      def _assert_equal_type_and_value(result, expected, err_msg=None):
35: (4)                        assert_equal(type(result), type(expected), err_msg=err_msg)
36: (4)                        if isinstance(result, tuple):
37: (8)                          assert_equal(len(result), len(expected), err_msg=err_msg)
38: (8)                          for result_item, expected_item in zip(result, expected):
39: (12)                            _assert_equal_type_and_value(result_item, expected_item, err_msg)
40: (4)                        else:
41: (8)                          assert_equal(result.value, expected.value, err_msg=err_msg)
42: (8)                          assert_equal(getattr(result.value, 'dtype', None),
43: (21)                            getattr(expected.value, 'dtype', None), err_msg=err_msg)
44: (0)                      _ALL_BINARY_OPERATORS = [
45: (4)                        operator.lt,
46: (4)                        operator.le,
47: (4)                        operator.eq,
48: (4)                        operator.ne,
49: (4)                        operator.gt,
50: (4)                        operator.ge,
51: (4)                        operator.add,
52: (4)                        operator.sub,
53: (4)                        operator.mul,
54: (4)                        operator.truediv,
55: (4)                        operator.floordiv,
56: (4)                        operator.mod,
57: (4)                        divmod,
58: (4)                        pow,
59: (4)                        operator.lshift,
60: (4)                        operator.rshift,
61: (4)                        operator.and_,
62: (4)                        operator.xor,
63: (4)                        operator.or_,
64: (0)                    ]
65: (0)                    class TestNDArrayOperatorsMixin:
66: (4)                      def test_array_like_add(self):
67: (8)                        def check(result):
68: (12)                          _assert_equal_type_and_value(result, ArrayLike(0))
69: (8)                          check(ArrayLike(0) + 0)
70: (8)                          check(0 + ArrayLike(0))
71: (8)                          check(ArrayLike(0) + np.array(0))
72: (8)                          check(np.array(0) + ArrayLike(0))
73: (8)                          check(ArrayLike(np.array(0)) + 0)
74: (8)                          check(0 + ArrayLike(np.array(0)))
75: (8)                          check(ArrayLike(np.array(0)) + np.array(0))
76: (8)                          check(np.array(0) + ArrayLike(np.array(0)))
77: (4)                      def test_inplace(self):
78: (8)                        array_like = ArrayLike(np.array([0]))
79: (8)                        array_like += 1
80: (8)                        _assert_equal_type_and_value(array_like, ArrayLike(np.array([1])))
81: (8)                        array = np.array([0])
82: (8)                        array += ArrayLike(1)
83: (8)                        _assert_equal_type_and_value(array, ArrayLike(np.array([1])))
84: (4)                      def test_opt_out(self):
85: (8)                        class OptOut:
86: (12)                          """Object that opts out of __array_ufunc__."""

```

```

87: (12)           __array_ufunc__ = None
88: (12)           def __add__(self, other):
89: (16)             return self
90: (12)           def __radd__(self, other):
91: (16)             return self
92: (8)           array_like = ArrayLike(1)
93: (8)           opt_out = OptOut()
94: (8)           assert_(array_like + opt_out is opt_out)
95: (8)           assert_(opt_out + array_like is opt_out)
96: (8)           with assert_raises(TypeError):
97: (12)             array_like += opt_out
98: (8)           with assert_raises(TypeError):
99: (12)             array_like - opt_out
100: (8)           with assert_raises(TypeError):
101: (12)             opt_out - array_like
102: (4)           def test_subclass(self):
103: (8)             class SubArrayLike(ArrayLike):
104: (12)               """Should take precedence over ArrayLike."""
105: (8)               x = ArrayLike(0)
106: (8)               y = SubArrayLike(1)
107: (8)               _assert_equal_type_and_value(x + y, y)
108: (8)               _assert_equal_type_and_value(y + x, y)
109: (4)           def test_object(self):
110: (8)             x = ArrayLike(0)
111: (8)             obj = object()
112: (8)             with assert_raises(TypeError):
113: (12)               x + obj
114: (8)             with assert_raises(TypeError):
115: (12)               obj + x
116: (8)             with assert_raises(TypeError):
117: (12)               x += obj
118: (4)           def test_unary_methods(self):
119: (8)             array = np.array([-1, 0, 1, 2])
120: (8)             array_like = ArrayLike(array)
121: (8)             for op in [operator.neg,
122: (19)               operator.pos,
123: (19)               abs,
124: (19)               operator.invert]:
125: (12)               _assert_equal_type_and_value(op(array_like), ArrayLike(op(array)))
126: (4)           def test_forward_binary_methods(self):
127: (8)             array = np.array([-1, 0, 1, 2])
128: (8)             array_like = ArrayLike(array)
129: (8)             for op in _ALL_BINARY_OPERATORS:
130: (12)               expected = wrap_array_like(op(array, 1))
131: (12)               actual = op(array_like, 1)
132: (12)               err_msg = 'failed for operator {}'.format(op)
133: (12)               _assert_equal_type_and_value(expected, actual, err_msg=err_msg)
134: (4)           def test_reflected_binary_methods(self):
135: (8)             for op in _ALL_BINARY_OPERATORS:
136: (12)               expected = wrap_array_like(op(2, 1))
137: (12)               actual = op(2, ArrayLike(1))
138: (12)               err_msg = 'failed for operator {}'.format(op)
139: (12)               _assert_equal_type_and_value(expected, actual, err_msg=err_msg)
140: (4)           def test_matmul(self):
141: (8)             array = np.array([1, 2], dtype=np.float64)
142: (8)             array_like = ArrayLike(array)
143: (8)             expected = ArrayLike(np.float64(5))
144: (8)             _assert_equal_type_and_value(expected, np.matmul(array_like, array))
145: (8)             _assert_equal_type_and_value(
146: (12)               expected, operator.matmul(array_like, array))
147: (8)             _assert_equal_type_and_value(
148: (12)               expected, operator.matmul(array, array_like))
149: (4)           def test_ufunc_at(self):
150: (8)             array = ArrayLike(np.array([1, 2, 3, 4]))
151: (8)             assert_(np.negative.at(array, np.array([0, 1])) is None)
152: (8)             _assert_equal_type_and_value(array, ArrayLike([-1, -2, 3, 4]))
153: (4)           def test_ufunc_two_outputs(self):
154: (8)             mantissa, exponent = np.frexp(2 ** -3)
155: (8)             expected = (ArrayLike(mantissa), ArrayLike(exponent))

```

```

156: (8)             _assert_equal_type_and_value(
157: (12)            np.frexp(ArrayLike(2 ** -3)), expected)
158: (8)             _assert_equal_type_and_value(
159: (12)            np.frexp(ArrayLike(np.array(2 ** -3))), expected)
-----
```

File 240 - test\_packbits.py:

```

1: (0)          import numpy as np
2: (0)          from numpy.testing import assert_array_equal, assert_equal, assert_raises
3: (0)          import pytest
4: (0)          from itertools import chain
5: (0)          def test_packbits():
6: (4)            a = [[[1, 0, 1], [0, 1, 0]],
7: (9)              [[1, 1, 0], [0, 0, 1]]]
8: (4)            for dt in '?bBhHiIlLqQ':
9: (8)              arr = np.array(a, dtype=dt)
10: (8)             b = np.packbits(arr, axis=-1)
11: (8)             assert_equal(b.dtype, np.uint8)
12: (8)             assert_array_equal(b, np.array([[160], [64]], [[192], [32]])))
13: (4)             assert_raises(TypeError, np.packbits, np.array(a, dtype=float))
14: (0)          def test_packbits_empty():
15: (4)            shapes = [
16: (8)              (0,), (10, 20, 0), (10, 0, 20), (0, 10, 20), (20, 0, 0), (0, 20, 0),
17: (8)              (0, 0, 20), (0, 0, 0),
18: (4)            ]
19: (4)            for dt in '?bBhHiIlLqQ':
20: (8)              for shape in shapes:
21: (12)                a = np.empty(shape, dtype=dt)
22: (12)                b = np.packbits(a)
23: (12)                assert_equal(b.dtype, np.uint8)
24: (12)                assert_equal(b.shape, (0,))
25: (0)          def test_packbits_empty_with_axis():
26: (4)            shapes = [
27: (8)              ((0,), [(0,)]),
28: (8)              ((10, 20, 0), [(2, 20, 0), (10, 3, 0), (10, 20, 0)]),
29: (8)              ((10, 0, 20), [(2, 0, 20), (10, 0, 20), (10, 0, 3)]),
30: (8)              ((0, 10, 20), [(0, 10, 20), (0, 2, 20), (0, 10, 3)]),
31: (8)              ((20, 0, 0), [(3, 0, 0), (20, 0, 0), (20, 0, 0)]),
32: (8)              ((0, 20, 0), [(0, 20, 0), (0, 3, 0), (0, 20, 0)]),
33: (8)              ((0, 0, 20), [(0, 0, 20), (0, 0, 20), (0, 0, 3)]),
34: (8)              ((0, 0, 0), [(0, 0, 0), (0, 0, 0), (0, 0, 0)]),
35: (4)            ]
36: (4)            for dt in '?bBhHiIlLqQ':
37: (8)              for in_shape, out_shapes in shapes:
38: (12)                for ax, out_shape in enumerate(out_shapes):
39: (16)                  a = np.empty(in_shape, dtype=dt)
40: (16)                  b = np.packbits(a, axis=ax)
41: (16)                  assert_equal(b.dtype, np.uint8)
42: (16)                  assert_equal(b.shape, out_shape)
43: (0)          @pytest.mark.parametrize('bitorder', ('little', 'big'))
44: (0)          def test_packbits_large(bitorder):
45: (4)            a = np.array([1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0,
46: (18)              0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1,
47: (18)              1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0,
48: (18)              1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1,
49: (18)              1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1,
50: (18)              1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1,
51: (18)              1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1,
52: (18)              0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1,
53: (18)              1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0,
54: (18)              1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1,
55: (18)              1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0,
56: (18)              0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1,
57: (18)              1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0,
58: (18)              1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0,
59: (18)              1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0])
60: (4)            a = a.repeat(3)
```

```

61: (4)          for dtype in '?bBhHiIlLqQ':
62: (8)            arr = np.array(a, dtype=dtype)
63: (8)            b = np.packbits(arr, axis=None, bitorder=bitorder)
64: (8)            assert_equal(b.dtype, np.uint8)
65: (8)            r = [252, 127, 192, 3, 254, 7, 252, 0, 7, 31, 240, 0, 28, 1, 255, 252,
66: (13)          113, 248, 3, 255, 192, 28, 15, 192, 28, 126, 0, 224, 127, 255,
67: (13)          227, 142, 7, 31, 142, 63, 28, 126, 56, 227, 240, 0, 227, 128, 63,
68: (13)          224, 14, 56, 252, 112, 56, 255, 241, 248, 3, 240, 56, 224, 112,
69: (13)          63, 255, 255, 199, 224, 14, 0, 31, 143, 192, 3, 255, 199, 0, 1,
70: (13)          255, 224, 1, 255, 252, 126, 63, 0, 1, 192, 252, 14, 63, 0, 15,
71: (13)          199, 252, 113, 255, 3, 128, 56, 252, 14, 7, 0, 113, 255, 255,
72: (13)          142, 56, 227,
73: (8)          129, 248, 227, 129, 199, 31, 128]
74: (12)        if bitorder == 'big':
75: (8)          assert_array_equal(b, r)
76: (8)        assert_array_equal(np.unpackbits(b, bitorder=bitorder)[:-4], a)
77: (8)        b = [np.packbits(arr[:i], axis=None)[-1] for i in range(1, 16)]
78: (31)        assert_array_equal(b, [128, 128, 128, 31, 30, 28, 24, 16, 0, 0, 0,
79: (8)          199,
80: (8)          198, 196, 192])
81: (8)        arr = arr.reshape(36, 25)
82: (8)        b = np.packbits(arr, axis=0)
83: (32)        assert_equal(b.dtype, np.uint8)
84: (32)        assert_array_equal(b, [[190, 186, 178, 178, 150, 215, 87, 83, 83, 195,
85: (31)          199, 206, 204, 204, 140, 140, 136, 136, 8, 40,
86: (32)          105,
87: (32)          107, 75, 74, 88],
88: (31)          [72, 216, 248, 241, 227, 195, 202, 90, 90, 83,
89: (32)          83, 119, 127, 109, 73, 64, 208, 244, 189, 45,
90: (32)          41, 104, 122, 90, 18],
91: (31)          [113, 120, 248, 216, 152, 24, 60, 52, 182, 150,
92: (32)          150, 150, 146, 210, 210, 246, 255, 255, 223,
93: (32)          151, 21, 17, 17, 131, 163],
94: (31)          [214, 210, 210, 64, 68, 5, 5, 1, 72, 88, 92,
95: (32)          92, 78, 110, 39, 181, 149, 220, 222, 218, 218,
96: (32)          202, 234, 170, 168],
97: (8)          [0, 128, 128, 192, 80, 112, 48, 160, 160, 224,
98: (8)          240, 208, 144, 128, 160, 224, 240, 208, 144,
99: (8)          144, 176, 240, 224, 192, 128]])
100: (31)        b = np.packbits(arr, axis=1)
101: (31)        assert_equal(b.dtype, np.uint8)
102: (31)        assert_array_equal(b, [[252, 127, 192, 0],
103: (31)          [7, 252, 15, 128],
104: (31)          [240, 0, 28, 0],
105: (31)          [255, 128, 0, 128],
106: (31)          [192, 31, 255, 128],
107: (31)          [142, 63, 0, 0],
108: (31)          [255, 240, 7, 0],
109: (31)          [7, 224, 14, 0],
110: (31)          [126, 0, 224, 0],
111: (31)          [255, 255, 199, 0],
112: (31)          [56, 28, 126, 0],
113: (31)          [113, 248, 227, 128],
114: (31)          [227, 142, 63, 0],
115: (31)          [0, 28, 112, 0],
116: (31)          [15, 248, 3, 128],
117: (31)          [28, 126, 56, 0],
118: (31)          [56, 255, 241, 128],
119: (31)          [240, 7, 224, 0],
120: (31)          [227, 129, 192, 128],
121: (31)          [255, 255, 254, 0],
122: (31)          [126, 0, 224, 0],
123: (31)          [3, 241, 248, 0],
124: (31)          [0, 255, 241, 128],
125: (31)          [128, 0, 255, 128],
126: (31)          [224, 113, 248, 0],

```

```

127: (31) [ 0, 252, 127, 128],
128: (31) [142, 63, 224, 0],
129: (31) [224, 14, 63, 0],
130: (31) [ 7, 3, 128, 0],
131: (31) [113, 255, 255, 128],
132: (31) [ 28, 113, 199, 0],
133: (31) [ 7, 227, 142, 0],
134: (31) [ 14, 56, 252, 0]])
135: (8) arr = arr.T.copy()
136: (8) b = np.packbits(arr, axis=0)
137: (8) assert_equal(b.dtype, np.uint8)
138: (8) assert_array_equal(b, [[252, 7, 240, 255, 192, 142, 255, 7, 126, 255,
139: (32) 56, 113, 227, 0, 15, 28, 56, 240, 227, 255,
140: (32) 126, 3, 0, 128, 224, 248, 0, 224, 0, 142, 224,
141: (32) 7, 113, 28, 7, 14],,
142: (32) [127, 252, 0, 128, 31, 63, 240, 224, 0, 255,
143: (33) 28, 248, 142, 28, 248, 126, 255, 7, 129, 255,
144: (33) 0, 241, 255, 0, 1, 252, 7, 113, 252, 63, 14,
145: (33) 3, 255, 113, 227, 56],
146: (32) [192, 15, 28, 0, 255, 0, 7, 14, 224, 199, 126,
147: (33) 227, 63, 112, 3, 56, 241, 224, 192, 254, 224,
148: (33) 248, 241, 255, 255, 126, 3, 248, 127, 224,
63,
149: (33) 128, 255, 199, 142, 252],
150: (32) [0, 128, 0, 128, 128, 0, 0, 0, 0, 0, 0, 128,
0,
151: (33) 0, 128, 0, 128, 0, 128, 0, 0, 0, 128, 128,
152: (33) 128, 0, 128, 0, 128, 0, 0, 0, 128, 0, 0, 0]])
153: (8) b = np.packbits(arr, axis=1)
154: (8) assert_equal(b.dtype, np.uint8)
155: (8) assert_array_equal(b, [[190, 72, 113, 214, 0],
156: (31) [186, 216, 120, 210, 128],
157: (31) [178, 248, 248, 210, 128],
158: (31) [178, 241, 216, 64, 192],
159: (31) [150, 227, 152, 68, 80],
160: (31) [215, 195, 24, 5, 112],
161: (31) [ 87, 202, 60, 5, 48],
162: (31) [ 83, 90, 52, 1, 160],
163: (31) [ 83, 90, 182, 72, 160],
164: (31) [195, 83, 150, 88, 224],
165: (31) [199, 83, 150, 92, 240],
166: (31) [206, 119, 150, 92, 208],
167: (31) [204, 127, 146, 78, 144],
168: (31) [204, 109, 210, 110, 128],
169: (31) [140, 73, 210, 39, 160],
170: (31) [140, 64, 246, 181, 224],
171: (31) [136, 208, 255, 149, 240],
172: (31) [136, 244, 255, 220, 208],
173: (31) [ 8, 189, 223, 222, 144],
174: (31) [ 40, 45, 151, 218, 144],
175: (31) [105, 41, 21, 218, 176],
176: (31) [107, 104, 17, 202, 240],
177: (31) [ 75, 122, 17, 234, 224],
178: (31) [ 74, 90, 131, 170, 192],
179: (31) [ 88, 18, 163, 168, 128]])
180: (4) for dtype in 'bBhHiIlLqQ':
181: (8)     arr = np.array(a, dtype=dtype)
182: (8)     rnd = np.random.randint(low=np.iinfo(dtype).min,
183: (32)                                     high=np.iinfo(dtype).max, size=arr.size,
184: (32)                                     dtype=dtype)
185: (8)     rnd[rnd == 0] = 1
186: (8)     arr *= rnd.astype(dtype)
187: (8)     b = np.packbits(arr, axis=-1)
188: (8)     assert_array_equal(np.unpackbits(b)[::-4], a)
189: (4)     assert_raises(TypeError, np.packbits, np.array(a, dtype=float))
190: (0) def test_packbits_very_large():
191: (4)     for s in range(950, 1050):
192: (8)         for dt in '?bBhHiIlLqQ':
193: (12)             x = np.ones((200, s), dtype=bool)

```

```

194: (12)                      np.packbits(x, axis=1)
195: (0)   def test_unpackbits():
196: (4)     a = np.array([[2], [7], [23]], dtype=np.uint8)
197: (4)     b = np.unpackbits(a, axis=1)
198: (4)     assert_equal(b.dtype, np.uint8)
199: (4)     assert_array_equal(b, np.array([[0, 0, 0, 0, 0, 0, 1, 0],
200: (36)                                [0, 0, 0, 0, 0, 1, 1, 1],
201: (36)                                [0, 0, 0, 1, 0, 1, 1, 1]]))
202: (0)   def test_pack_unpack_order():
203: (4)     a = np.array([[2], [7], [23]], dtype=np.uint8)
204: (4)     b = np.unpackbits(a, axis=1)
205: (4)     assert_equal(b.dtype, np.uint8)
206: (4)     b_little = np.unpackbits(a, axis=1, bitorder='little')
207: (4)     b_big = np.unpackbits(a, axis=1, bitorder='big')
208: (4)     assert_array_equal(b, b_big)
209: (4)     assert_array_equal(a, np.packbits(b_little, axis=1, bitorder='little'))
210: (4)     assert_array_equal(b[:,::-1], b_little)
211: (4)     assert_array_equal(a, np.packbits(b_big, axis=1, bitorder='big'))
212: (4)     assert_raises(ValueError, np.unpackbits, a, bitorder='r')
213: (4)     assert_raises(TypeError, np.unpackbits, a, bitorder=10)
214: (0)   def test_unpackbits_empty():
215: (4)     a = np.empty((0,), dtype=np.uint8)
216: (4)     b = np.unpackbits(a)
217: (4)     assert_equal(b.dtype, np.uint8)
218: (4)     assert_array_equal(b, np.empty((0,)))
219: (0)   def test_unpackbits_empty_with_axis():
220: (4)     shapes = [
221: (8)       [[(0,)], (0,)),
222: (8)       [[[2, 24, 0), (16, 3, 0), (16, 24, 0)], (16, 24, 0)),
223: (8)       [[[2, 0, 24), (16, 0, 24), (16, 0, 3)], (16, 0, 24)),
224: (8)       [[[0, 16, 24), (0, 2, 24), (0, 16, 3)], (0, 16, 24)),
225: (8)       [[[3, 0, 0), (24, 0, 0), (24, 0, 0)], (24, 0, 0)),
226: (8)       [[[0, 24, 0), (0, 3, 0), (0, 24, 0)], (0, 24, 0)),
227: (8)       [[[0, 0, 24), (0, 0, 24), (0, 0, 3)], (0, 0, 24)),
228: (8)       [[[0, 0, 0), (0, 0, 0), (0, 0, 0)], (0, 0, 0))],
229: (4)
230: (4)     for in_shapes, out_shape in shapes:
231: (8)       for ax, in_shape in enumerate(in_shapes):
232: (12)         a = np.empty(in_shape, dtype=np.uint8)
233: (12)         b = np.unpackbits(a, axis=ax)
234: (12)         assert_equal(b.dtype, np.uint8)
235: (12)         assert_equal(b.shape, out_shape)
236: (0)   def test_unpackbits_large():
237: (4)     d = np.arange(277, dtype=np.uint8)
238: (4)     assert_array_equal(np.packbits(np.unpackbits(d)), d)
239: (4)     assert_array_equal(np.packbits(np.unpackbits(d[::2])), d[::2])
240: (4)     d = np.tile(d, (3, 1))
241: (4)     assert_array_equal(np.packbits(np.unpackbits(d, axis=1), axis=1), d)
242: (4)     d = d.T.copy()
243: (4)     assert_array_equal(np.packbits(np.unpackbits(d, axis=0), axis=0), d)
244: (0)   class TestCount():
245: (4)     x = np.array([
246: (8)       [1, 0, 1, 0, 0, 1, 0],
247: (8)       [0, 1, 1, 1, 0, 0, 0],
248: (8)       [0, 0, 1, 0, 0, 1, 1],
249: (8)       [1, 1, 0, 0, 0, 1, 1],
250: (8)       [1, 0, 1, 0, 1, 0, 1],
251: (8)       [0, 0, 1, 1, 1, 0, 0],
252: (8)       [0, 1, 0, 1, 0, 1, 0],
253: (4)     ], dtype=np.uint8)
254: (4)     padded1 = np.zeros(57, dtype=np.uint8)
255: (4)     padded1[:49] = x.ravel()
256: (4)     padded1b = np.zeros(57, dtype=np.uint8)
257: (4)     padded1b[:49] = x[::-1].copy().ravel()
258: (4)     padded2 = np.zeros((9, 9), dtype=np.uint8)
259: (4)     padded2[:7, :7] = x
260: (4)     @pytest.mark.parametrize('bitorder', ('little', 'big'))
261: (4)     @pytest.mark.parametrize('count', chain(range(58), range(-1, -57, -1)))
262: (4)     def test_roundtrip(self, bitorder, count):

```

```

263: (8)             if count < 0:
264: (12)            cutoff = count - 1
265: (8)            else:
266: (12)              cutoff = count
267: (8)            packed = np.packbits(self.x, bitorder=bitorder)
268: (8)            unpacked = np.unpackbits(packed, count=count, bitorder=bitorder)
269: (8)            assert_equal(unpacked.dtype, np.uint8)
270: (8)            assert_array_equal(unpacked, self.padded1[:cutoff])
271: (4)            @pytest.mark.parametrize('kwargs', [
272: (20)                {}, {'count': None},
273: (20)                ])
274: (4)            def test_count(self, kwargs):
275: (8)                packed = np.packbits(self.x)
276: (8)                unpacked = np.unpackbits(packed, **kwargs)
277: (8)                assert_equal(unpacked.dtype, np.uint8)
278: (8)                assert_array_equal(unpacked, self.padded1[:-1])
279: (4)                @pytest.mark.parametrize('bitorder', ('little', 'big'))
280: (4)                @pytest.mark.parametrize('count', chain(range(8), range(-1, -9, -1)))
281: (4)            def test_roundtrip_axis(self, bitorder, count):
282: (8)                if count < 0:
283: (12)                  cutoff = count - 1
284: (8)                else:
285: (12)                  cutoff = count
286: (8)                packed0 = np.packbits(self.x, axis=0, bitorder=bitorder)
287: (8)                unpacked0 = np.unpackbits(packed0, axis=0, count=count,
288: (34)                                bitorder=bitorder)
289: (8)                assert_equal(unpacked0.dtype, np.uint8)
290: (8)                assert_array_equal(unpacked0, self.padded2[:cutoff, :self.x.shape[1]])
291: (8)                packed1 = np.packbits(self.x, axis=1, bitorder=bitorder)
292: (8)                unpacked1 = np.unpackbits(packed1, axis=1, count=count,
293: (34)                                bitorder=bitorder)
294: (8)                assert_equal(unpacked1.dtype, np.uint8)
295: (8)                assert_array_equal(unpacked1, self.padded2[:self.x.shape[0], :cutoff])
296: (4)            @pytest.mark.parametrize('kwargs', [
297: (20)                {}, {'count': None},
298: (20)                {'bitorder' : 'little'},
299: (20)                {'bitorder': 'little', 'count': None},
300: (20)                {'bitorder' : 'big'},
301: (20)                {'bitorder': 'big', 'count': None},
302: (20)                ])
303: (4)            def test_axis_count(self, kwargs):
304: (8)                packed0 = np.packbits(self.x, axis=0)
305: (8)                unpacked0 = np.unpackbits(packed0, axis=0, **kwargs)
306: (8)                assert_equal(unpacked0.dtype, np.uint8)
307: (8)                if kwargs.get('bitorder', 'big') == 'big':
308: (12)                  assert_array_equal(unpacked0, self.padded2[:-1, :self.x.shape[1]])
309: (8)
310: (12)                assert_array_equal(unpacked0[::-1, :], self.padded2[-1,
311: (8)                    :self.x.shape[1]])
312: (8)
313: (8)
314: (8)
315: (12)                packed1 = np.packbits(self.x, axis=1)
316: (8)                unpacked1 = np.unpackbits(packed1, axis=1, **kwargs)
317: (12)                assert_equal(unpacked1.dtype, np.uint8)
318: (4)                if kwargs.get('bitorder', 'big') == 'big':
319: (8)                  assert_array_equal(unpacked1, self.padded2[:self.x.shape[0], :-1])
320: (8)                else:
321: (8)                  assert_array_equal(unpacked1[:, ::-1],
322: (8)                    self.padded2[:self.x.shape[0], :-1])
323: (8)
324: (8)            def test_bad_count(self):
325: (8)                packed0 = np.packbits(self.x, axis=0)
326: (8)                assert_raises(ValueError, np.unpackbits, packed0, axis=0, count=-9)
327: (8)                packed1 = np.packbits(self.x, axis=1)
328: (8)                assert_raises(ValueError, np.unpackbits, packed1, axis=1, count=-9)
329: (8)                packed = np.packbits(self.x)
330: (8)                assert_raises(ValueError, np.unpackbits, packed, count=-57)

```

-----  
File 241 - test\_nanfunctions.py:

```

1: (0) import warnings
2: (0) import pytest
3: (0) import inspect
4: (0) import numpy as np
5: (0) from numpy.core.numeric import normalize_axis_tuple
6: (0) from numpy.lib.nanfunctions import _nan_mask, _replace_nan
7: (0) from numpy.testing import (
8: (4)     assert_, assert_equal, assert_almost_equal, assert_raises,
9: (4)     assert_array_equal, suppress_warnings
10: (4) )
11: (0) _ndat = np.array([[0.6244, np.nan, 0.2692, 0.0116, np.nan, 0.1170],
12: (18)             [0.5351, -0.9403, np.nan, 0.2100, 0.4759, 0.2833],
13: (18)             [np.nan, np.nan, np.nan, 0.1042, np.nan, -0.5954],
14: (18)             [0.1610, np.nan, np.nan, 0.1859, 0.3146, np.nan]])
15: (0) _rdat = [np.array([0.6244, 0.2692, 0.0116, 0.1170]),
16: (9)     np.array([0.5351, -0.9403, 0.2100, 0.4759, 0.2833]),
17: (9)     np.array([0.1042, -0.5954]),
18: (9)     np.array([0.1610, 0.1859, 0.3146])]
19: (0) _ndat_ones = np.array([[0.6244, 1.0, 0.2692, 0.0116, 1.0, 0.1170],
20: (23)             [0.5351, -0.9403, 1.0, 0.2100, 0.4759, 0.2833],
21: (23)             [1.0, 1.0, 1.0, 0.1042, 1.0, -0.5954],
22: (23)             [0.1610, 1.0, 1.0, 0.1859, 0.3146, 1.0]])
23: (0) _ndat_zeros = np.array([[0.6244, 0.0, 0.2692, 0.0116, 0.0, 0.1170],
24: (24)             [0.5351, -0.9403, 0.0, 0.2100, 0.4759, 0.2833],
25: (24)             [0.0, 0.0, 0.0, 0.1042, 0.0, -0.5954],
26: (24)             [0.1610, 0.0, 0.0, 0.1859, 0.3146, 0.0]])
27: (0) class TestSignatureMatch:
28: (4)     NANFUNCS = {
29: (8)         np.nanmin: np.amin,
30: (8)         np.nanmax: np.amax,
31: (8)         np.nanargmin: np.argmin,
32: (8)         np.nanargmax: np.argmax,
33: (8)         np.nansum: np.sum,
34: (8)         np.nanprod: np.prod,
35: (8)         np.nancumsum: np.cumsum,
36: (8)         np.nancumprod: np.cumprod,
37: (8)         np.nanmean: np.mean,
38: (8)         np.nanmedian: np.median,
39: (8)         np.nanpercentile: np.percentile,
40: (8)         np.nanquantile: np.quantile,
41: (8)         np.nanvar: np.var,
42: (8)         np.nanstd: np.std,
43: (4)     }
44: (4)     IDS = [k.__name__ for k in NANFUNCS]
45: (4)     @staticmethod
46: (4)     def get_signature(func, default="..."):
47: (8)         """Construct a signature and replace all default parameter-values."""
48: (8)         prm_list = []
49: (8)         signature = inspect.signature(func)
50: (8)         for prm in signature.parameters.values():
51: (12)             if prm.default is inspect.Parameter.empty:
52: (16)                 prm_list.append(prm)
53: (12)             else:
54: (16)                 prm_list.append(prm.replace(default=default))
55: (8)         return inspect.Signature(prm_list)
56: (4)     @pytest.mark.parametrize("nan_func,func", NANFUNCS.items(), ids=IDS)
57: (4)     def test_signature_match(self, nan_func, func):
58: (8)         signature = self.get_signature(func)
59: (8)         nan_signature = self.get_signature(nan_func)
60: (8)         np.testing.assert_equal(signature, nan_signature)
61: (4)     def test_exhaustiveness(self):
62: (8)         """Validate that all nan functions are actually tested."""
63: (8)         np.testing.assert_equal(
64: (12)             set(self.IDS), set(np.lib.nanfunctions._all__))
65: (8)     )
66: (0) class TestNanFunctions_MinMax:
67: (4)     nanfuncs = [np.nanmin, np.nanmax]
68: (4)     stdfuncs = [np.min, np.max]
69: (4)     def test_mutation(self):

```

```

70: (8)             ndat = _ndat.copy()
71: (8)             for f in self.nanfuncs:
72: (12)                 f(ndat)
73: (12)                 assert_equal(ndat, _ndat)
74: (4)             def test_keepdims(self):
75: (8)                 mat = np.eye(3)
76: (8)                 for nf, rf in zip(self.nanfuncs, self.stdfuncs):
77: (12)                     for axis in [None, 0, 1]:
78: (16)                         tgt = rf(mat, axis=axis, keepdims=True)
79: (16)                         res = nf(mat, axis=axis, keepdims=True)
80: (16)                         assert_(res.ndim == tgt.ndim)
81: (4)             def test_out(self):
82: (8)                 mat = np.eye(3)
83: (8)                 for nf, rf in zip(self.nanfuncs, self.stdfuncs):
84: (12)                     resout = np.zeros(3)
85: (12)                     tgt = rf(mat, axis=1)
86: (12)                     res = nf(mat, axis=1, out=resout)
87: (12)                     assert_almost_equal(res, resout)
88: (12)                     assert_almost_equal(res, tgt)
89: (4)             def test_dtype_from_input(self):
90: (8)                 codes = 'efdgFDG'
91: (8)                 for nf, rf in zip(self.nanfuncs, self.stdfuncs):
92: (12)                     for c in codes:
93: (16)                         mat = np.eye(3, dtype=c)
94: (16)                         tgt = rf(mat, axis=1).dtype.type
95: (16)                         res = nf(mat, axis=1).dtype.type
96: (16)                         assert_(res is tgt)
97: (16)                         tgt = rf(mat, axis=None).dtype.type
98: (16)                         res = nf(mat, axis=None).dtype.type
99: (16)                         assert_(res is tgt)
100: (4)            def test_result_values(self):
101: (8)             for nf, rf in zip(self.nanfuncs, self.stdfuncs):
102: (12)                 tgt = [rf(d) for d in _rdat]
103: (12)                 res = nf(_ndat, axis=1)
104: (12)                 assert_almost_equal(res, tgt)
105: (4)             @pytest.mark.parametrize("axis", [None, 0, 1])
106: (4)             @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
107: (4)             @pytest.mark.parametrize("array", [
108: (8)                 np.array(np.nan),
109: (8)                 np.full((3, 3), np.nan),
110: (4)             ], ids=["0d", "2d"])
111: (4)             def test_allnans(self, axis, dtype, array):
112: (8)                 if axis is not None and array.ndim == 0:
113: (12)                     pytest.skip(f"`axis != None` not supported for 0d arrays")
114: (8)                 array = array.astype(dtype)
115: (8)                 match = "All-NaN slice encountered"
116: (8)                 for func in self.nanfuncs:
117: (12)                     with pytest.warns(RuntimeWarning, match=match):
118: (16)                         out = func(array, axis=axis)
119: (12)                         assert np.isnan(out).all()
120: (12)                         assert out.dtype == array.dtype
121: (4)             def test_masked(self):
122: (8)                 mat = np.ma.fix_invalid(_ndat)
123: (8)                 msk = mat._mask.copy()
124: (8)                 for f in [np.nanmin]:
125: (12)                     res = f(mat, axis=1)
126: (12)                     tgt = f(_ndat, axis=1)
127: (12)                     assert_equal(res, tgt)
128: (12)                     assert_equal(mat._mask, msk)
129: (12)                     assert_(not np.isinf(mat).any())
130: (4)             def test_scalar(self):
131: (8)                 for f in self.nanfuncs:
132: (12)                     assert_(f(0.) == 0.)
133: (4)             def test_subclass(self):
134: (8)                 class MyNDArray(np.ndarray):
135: (12)                     pass
136: (8)                     mine = np.eye(3).view(MyNDArray)
137: (8)                     for f in self.nanfuncs:
138: (12)                         res = f(mine, axis=0)

```

```

139: (12)             assert_(isinstance(res, MyNDArray))
140: (12)             assert_(res.shape == (3,))
141: (12)             res = f(mine, axis=1)
142: (12)             assert_(isinstance(res, MyNDArray))
143: (12)             assert_(res.shape == (3,))
144: (12)             res = f(mine)
145: (12)             assert_(res.shape == ())
146: (8)              mine[1] = np.nan
147: (8)              for f in self.nanfuncs:
148: (12)                  with warnings.catch_warnings(record=True) as w:
149: (16)                      warnings.simplefilter('always')
150: (16)                      res = f(mine, axis=0)
151: (16)                      assert_(isinstance(res, MyNDArray))
152: (16)                      assert_(not np.any(np.isnan(res)))
153: (16)                      assert_(len(w) == 0)
154: (12)                  with warnings.catch_warnings(record=True) as w:
155: (16)                      warnings.simplefilter('always')
156: (16)                      res = f(mine, axis=1)
157: (16)                      assert_(isinstance(res, MyNDArray))
158: (16)                      assert_(np.isnan(res[1]) and not np.isnan(res[0])
159: (24)                          and not np.isnan(res[2]))
160: (16)                      assert_(len(w) == 1, 'no warning raised')
161: (16)                      assert_(issubclass(w[0].category, RuntimeWarning))
162: (12)                  with warnings.catch_warnings(record=True) as w:
163: (16)                      warnings.simplefilter('always')
164: (16)                      res = f(mine)
165: (16)                      assert_(res.shape == ())
166: (16)                      assert_(res != np.nan)
167: (16)                      assert_(len(w) == 0)
168: (4)               def test_object_array(self):
169: (8)                 arr = np.array([[1.0, 2.0], [np.nan, 4.0], [np.nan, np.nan]],
dtype=object)
170: (8)                 assert_equal(np.nanmin(arr), 1.0)
171: (8)                 assert_equal(np.nanmin(arr, axis=0), [1.0, 2.0])
172: (8)                 with warnings.catch_warnings(record=True) as w:
173: (12)                     warnings.simplefilter('always')
174: (12)                     assert_equal(list(np.nanmin(arr, axis=1)), [1.0, 4.0, np.nan])
175: (12)                     assert_(len(w) == 1, 'no warning raised')
176: (12)                     assert_(issubclass(w[0].category, RuntimeWarning))
177: (4) @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
178: (4) def test_initial(self, dtype):
179: (8)             class MyNDArray(np.ndarray):
180: (12)                 pass
181: (8)                 ar = np.arange(9).astype(dtype)
182: (8)                 ar[:5] = np.nan
183: (8)                 for f in self.nanfuncs:
184: (12)                     initial = 100 if f is np.nanmax else 0
185: (12)                     ret1 = f(ar, initial=initial)
186: (12)                     assert ret1.dtype == dtype
187: (12)                     assert ret1 == initial
188: (12)                     ret2 = f(ar.view(MyNDArray), initial=initial)
189: (12)                     assert ret2.dtype == dtype
190: (12)                     assert ret2 == initial
191: (4) @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
192: (4) def test_where(self, dtype):
193: (8)             class MyNDArray(np.ndarray):
194: (12)                 pass
195: (8)                 ar = np.arange(9).reshape(3, 3).astype(dtype)
196: (8)                 ar[0, :] = np.nan
197: (8)                 where = np.ones_like(ar, dtype=np.bool_)
198: (8)                 where[:, 0] = False
199: (8)                 for f in self.nanfuncs:
200: (12)                     reference = 4 if f is np.nanmin else 8
201: (12)                     ret1 = f(ar, where=where, initial=5)
202: (12)                     assert ret1.dtype == dtype
203: (12)                     assert ret1 == reference
204: (12)                     ret2 = f(ar.view(MyNDArray), where=where, initial=5)
205: (12)                     assert ret2.dtype == dtype
206: (12)                     assert ret2 == reference

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

207: (0)
208: (4)
209: (4)
210: (8)
211: (8)
212: (12)
213: (12)
214: (4)
215: (8)
216: (12)
217: (16)
218: (20)
219: (20)
220: (20)
221: (20)
222: (20)
223: (20)
224: (4)
225: (4)
226: (4)
227: (8)
228: (8)
229: (4)
230: (4)
231: (8)
232: (12)
233: (8)
234: (8)
235: (12)
236: (16)
237: (4)
238: (8)
239: (8)
240: (12)
241: (16)
242: (12)
243: (16)
244: (16)
245: (4)
246: (8)
247: (12)
248: (4)
249: (8)
250: (12)
251: (8)
252: (8)
253: (12)
254: (12)
255: (12)
256: (12)
257: (12)
258: (12)
259: (12)
260: (12)
261: (4)
262: (4)
263: (8)
264: (8)
265: (8)
266: (12)
267: (12)
268: (12)
269: (12)
270: (4)
271: (4)
272: (8)
273: (8)
274: (8)
275: (12)

class TestNanFunctions_ArgminArgmax:
    nanfuncs = [np.nanargmin, np.nanargmax]
    def test_mutation(self):
        ndat = _ndat.copy()
        for f in self.nanfuncs:
            f(ndat)
            assert_equal(ndat, _ndat)
    def test_result_values(self):
        for f, fcmp in zip(self.nanfuncs, [np.greater, np.less]):
            for row in _ndat:
                with suppress_warnings() as sup:
                    sup.filter(RuntimeWarning, "invalid value encountered in")
                    ind = f(row)
                    val = row[ind]
                    assert_(not np.isnan(val))
                    assert_(not fcmp(val, row).any())
                    assert_(not np.equal(val, row[:ind]).any())
    @pytest.mark.parametrize("axis", [None, 0, 1])
    @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
    @pytest.mark.parametrize("array", [
        np.array(np.nan),
        np.full((3, 3), np.nan),
    ], ids=["0d", "2d"])
    def test_allnans(self, axis, dtype, array):
        if axis is not None and array.ndim == 0:
            pytest.skip(f"`axis != None` not supported for 0d arrays")
        array = array.astype(dtype)
        for func in self.nanfuncs:
            with pytest.raises(ValueError, match="All-NaN slice encountered"):
                func(array, axis=axis)
    def test_empty(self):
        mat = np.zeros((0, 3))
        for f in self.nanfuncs:
            for axis in [0, None]:
                assert_raises(ValueError, f, mat, axis=axis)
            for axis in [1]:
                res = f(mat, axis=axis)
                assert_equal(res, np.zeros(0))
    def test_scalar(self):
        for f in self.nanfuncs:
            assert_(f(0.) == 0.)
    def test_subclass(self):
        class MyNDArray(np.ndarray):
            pass
        mine = np.eye(3).view(MyNDArray)
        for f in self.nanfuncs:
            res = f(mine, axis=0)
            assert_(isinstance(res, MyNDArray))
            assert_(res.shape == (3,))
            res = f(mine, axis=1)
            assert_(isinstance(res, MyNDArray))
            assert_(res.shape == (3,))
            res = f(mine)
            assert_(res.shape == ())
    @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
    def test_keeplims(self, dtype):
        ar = np.arange(9).astype(dtype)
        ar[:5] = np.nan
        for f in self.nanfuncs:
            reference = 5 if f is np.nanargmin else 8
            ret = f(ar, keepdims=True)
            assert ret.ndim == ar.ndim
            assert ret == reference
    @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
    def test_out(self, dtype):
        ar = np.arange(9).astype(dtype)
        ar[:5] = np.nan
        for f in self.nanfuncs:
            out = np.zeros((), dtype=np.intp)

```

```

276: (12)                     reference = 5 if f is np.nanargmin else 8
277: (12)                     ret = f(ar, out=out)
278: (12)                     assert ret is out
279: (12)                     assert ret == reference
280: (0) _TEST_ARRAYS = {
281: (4)     "0d": np.array(5),
282: (4)     "1d": np.array([127, 39, 93, 87, 46])
283: (0)
284: (0)     for _v in _TEST_ARRAYS.values():
285: (4)         _v.setflags(write=False)
286: (0)     @pytest.mark.parametrize(
287: (4)         "dtype",
288: (4)         np.typecodes["AllInteger"] + np.typecodes["AllFloat"] + "0",
289: (0)     )
290: (0)     @pytest.mark.parametrize("mat", _TEST_ARRAYS.values(),
ids=_TEST_ARRAYS.keys())
291: (0)     class TestNanFunctions_NumberTypes:
292: (4)         nanfuncs = {
293: (8)             np.nanmin: np.min,
294: (8)             np.nanmax: np.max,
295: (8)             np.nanargmin: np.argmin,
296: (8)             np.nanargmax: np.argmax,
297: (8)             np.nansum: np.sum,
298: (8)             np.nanprod: np.prod,
299: (8)             np.nancumsum: np.cumsum,
300: (8)             np.nancumprod: np.cumprod,
301: (8)             np.nanmean: np.mean,
302: (8)             np.nanmedian: np.median,
303: (8)             np.nanvar: np.var,
304: (8)             np.nanstd: np.std,
305: (4)
306: (4)         nanfunc_ids = [i.__name__ for i in nanfuncs]
307: (4)         @pytest.mark.parametrize("nanfunc,func", nanfuncs.items(),
ids=nanfunc_ids)
308: (4)         @np.errstate(over="ignore")
309: (4)         def test_nanfunc(self, mat, dtype, nanfunc, func):
310: (8)             mat = mat.astype(dtype)
311: (8)             tgt = func(mat)
312: (8)             out = nanfunc(mat)
313: (8)             assert_almost_equal(out, tgt)
314: (8)             if dtype == "0":
315: (12)                 assert type(out) is type(tgt)
316: (8)             else:
317: (12)                 assert out.dtype == tgt.dtype
318: (4)             @pytest.mark.parametrize(
319: (8)                 "nanfunc,func",
320: (8)                 [(np.nanquantile, np.quantile), (np.nanpercentile, np.percentile)],
321: (8)                 ids=["nanquantile", "nanpercentile"],
322: (4)
323: (4)             def test_nanfunc_q(self, mat, dtype, nanfunc, func):
324: (8)                 mat = mat.astype(dtype)
325: (8)                 if mat.dtype.kind == "c":
326: (12)                     assert_raises(TypeError, func, mat, q=1)
327: (12)                     assert_raises(TypeError, nanfunc, mat, q=1)
328: (8)                 else:
329: (12)                     tgt = func(mat, q=1)
330: (12)                     out = nanfunc(mat, q=1)
331: (12)                     assert_almost_equal(out, tgt)
332: (12)                     if dtype == "0":
333: (16)                         assert type(out) is type(tgt)
334: (12)                     else:
335: (16)                         assert out.dtype == tgt.dtype
336: (4)             @pytest.mark.parametrize(
337: (8)                 "nanfunc,func",
338: (8)                 [(np.nanvar, np.var), (np.nanstd, np.std)],
339: (8)                 ids=["nanvar", "nanstd"],
340: (4)
341: (4)             def test_nanfunc_ddof(self, mat, dtype, nanfunc, func):
342: (8)                 mat = mat.astype(dtype)

```

```

343: (8)             tgt = func(mat, ddof=0.5)
344: (8)             out = nanfunc(mat, ddof=0.5)
345: (8)             assert_almost_equal(out, tgt)
346: (8)             if dtype == "O":
347: (12)                 assert type(out) is type(tgt)
348: (8)             else:
349: (12)                 assert out.dtype == tgt.dtype
350: (0)             class SharedNanFunctionsTestsMixin:
351: (4)                 def test_mutation(self):
352: (8)                     ndat = _ndat.copy()
353: (8)                     for f in self.nanfuncs:
354: (12)                         f(ndat)
355: (12)                         assert_equal(ndat, _ndat)
356: (4)                 def test_kepdims(self):
357: (8)                     mat = np.eye(3)
358: (8)                     for nf, rf in zip(self.nanfuncs, self.stdfuncs):
359: (12)                         for axis in [None, 0, 1]:
360: (16)                             tgt = rf(mat, axis=axis, keepdims=True)
361: (16)                             res = nf(mat, axis=axis, keepdims=True)
362: (16)                             assert_(res.ndim == tgt.ndim)
363: (4)                 def test_out(self):
364: (8)                     mat = np.eye(3)
365: (8)                     for nf, rf in zip(self.nanfuncs, self.stdfuncs):
366: (12)                         resout = np.zeros(3)
367: (12)                         tgt = rf(mat, axis=1)
368: (12)                         res = nf(mat, axis=1, out=resout)
369: (12)                         assert_almost_equal(res, resout)
370: (12)                         assert_almost_equal(res, tgt)
371: (4)                 def test_dtype_from_dtype(self):
372: (8)                     mat = np.eye(3)
373: (8)                     codes = 'efdgFDG'
374: (8)                     for nf, rf in zip(self.nanfuncs, self.stdfuncs):
375: (12)                         for c in codes:
376: (16)                             with suppress_warnings() as sup:
377: (20)                                 if nf in {np.nanstd, np.nanvar} and c in 'FDG':
378: (24)                                     sup.filter(np.ComplexWarning)
379: (20)                                     tgt = rf(mat, dtype=np.dtype(c), axis=1).dtype.type
380: (20)                                     res = nf(mat, dtype=np.dtype(c), axis=1).dtype.type
381: (20)                                     assert_(res is tgt)
382: (20)                                     tgt = rf(mat, dtype=np.dtype(c), axis=None).dtype.type
383: (20)                                     res = nf(mat, dtype=np.dtype(c), axis=None).dtype.type
384: (20)                                     assert_(res is tgt)
385: (4)                 def test_dtype_from_char(self):
386: (8)                     mat = np.eye(3)
387: (8)                     codes = 'efdgFDG'
388: (8)                     for nf, rf in zip(self.nanfuncs, self.stdfuncs):
389: (12)                         for c in codes:
390: (16)                             with suppress_warnings() as sup:
391: (20)                                 if nf in {np.nanstd, np.nanvar} and c in 'FDG':
392: (24)                                     sup.filter(np.ComplexWarning)
393: (20)                                     tgt = rf(mat, dtype=c, axis=1).dtype.type
394: (20)                                     res = nf(mat, dtype=c, axis=1).dtype.type
395: (20)                                     assert_(res is tgt)
396: (20)                                     tgt = rf(mat, dtype=c, axis=None).dtype.type
397: (20)                                     res = nf(mat, dtype=c, axis=None).dtype.type
398: (20)                                     assert_(res is tgt)
399: (4)                 def test_dtype_from_input(self):
400: (8)                     codes = 'efdgFDG'
401: (8)                     for nf, rf in zip(self.nanfuncs, self.stdfuncs):
402: (12)                         for c in codes:
403: (16)                             mat = np.eye(3, dtype=c)
404: (16)                             tgt = rf(mat, axis=1).dtype.type
405: (16)                             res = nf(mat, axis=1).dtype.type
406: (16)                             assert_(res is tgt, "res %s, tgt %s" % (res, tgt))
407: (16)                             tgt = rf(mat, axis=None).dtype.type
408: (16)                             res = nf(mat, axis=None).dtype.type
409: (16)                             assert_(res is tgt)
410: (4)                 def test_result_values(self):
411: (8)                     for nf, rf in zip(self.nanfuncs, self.stdfuncs):

```

```

412: (12)             tgt = [rf(d) for d in _rdat]
413: (12)             res = nf(_ndat, axis=1)
414: (12)             assert_almost_equal(res, tgt)
415: (4)              def test_scalar(self):
416: (8)                for f in self.nanfuncs:
417: (12)                  assert_(f(0.) == 0.)
418: (4)              def test_subclass(self):
419: (8)                class MyNDArray(np.ndarray):
420: (12)                  pass
421: (8)                array = np.eye(3)
422: (8)                mine = array.view(MyNDArray)
423: (8)                for f in self.nanfuncs:
424: (12)                  expected_shape = f(array, axis=0).shape
425: (12)                  res = f(mine, axis=0)
426: (12)                  assert_(isinstance(res, MyNDArray))
427: (12)                  assert_(res.shape == expected_shape)
428: (12)                  expected_shape = f(array, axis=1).shape
429: (12)                  res = f(mine, axis=1)
430: (12)                  assert_(isinstance(res, MyNDArray))
431: (12)                  assert_(res.shape == expected_shape)
432: (12)                  expected_shape = f(array).shape
433: (12)                  res = f(mine)
434: (12)                  assert_(isinstance(res, MyNDArray))
435: (12)                  assert_(res.shape == expected_shape)
436: (0)               class TestNanFunctions_SumProd(SharedNanFunctionsTestsMixin):
437: (4)                 nanfuncs = [np.nansum, np.nanprod]
438: (4)                 stdfuncs = [np.sum, np.prod]
439: (4)                 @pytest.mark.parametrize("axis", [None, 0, 1])
440: (4)                 @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
441: (4)                 @pytest.mark.parametrize("array", [
442: (8)                   np.array(np.nan),
443: (8)                   np.full((3, 3), np.nan),
444: (4)                 ], ids=["0d", "2d"])
445: (4)                 def test_allnans(self, axis, dtype, array):
446: (8)                   if axis is not None and array.ndim == 0:
447: (12)                     pytest.skip(f`axis != None` not supported for 0d arrays`)
448: (8)                   array = array.astype(dtype)
449: (8)                   for func, identity in zip(self.nanfuncs, [0, 1]):
450: (12)                     out = func(array, axis=axis)
451: (12)                     assert np.all(out == identity)
452: (12)                     assert out.dtype == array.dtype
453: (4)                 def test_empty(self):
454: (8)                   for f, tgt_value in zip([np.nansum, np.nanprod], [0, 1]):
455: (12)                     mat = np.zeros((0, 3))
456: (12)                     tgt = [tgt_value]*3
457: (12)                     res = f(mat, axis=0)
458: (12)                     assert_equal(res, tgt)
459: (12)                     tgt = []
460: (12)                     res = f(mat, axis=1)
461: (12)                     assert_equal(res, tgt)
462: (12)                     tgt = tgt_value
463: (12)                     res = f(mat, axis=None)
464: (12)                     assert_equal(res, tgt)
465: (4)                 @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
466: (4)                 def test_initial(self, dtype):
467: (8)                   ar = np.arange(9).astype(dtype)
468: (8)                   ar[:5] = np.nan
469: (8)                   for f in self.nanfuncs:
470: (12)                     reference = 28 if f is np.nansum else 3360
471: (12)                     ret = f(ar, initial=2)
472: (12)                     assert ret.dtype == dtype
473: (12)                     assert ret == reference
474: (4)                 @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
475: (4)                 def test_where(self, dtype):
476: (8)                   ar = np.arange(9).reshape(3, 3).astype(dtype)
477: (8)                   ar[0, :] = np.nan
478: (8)                   where = np.ones_like(ar, dtype=np.bool_)
479: (8)                   where[:, 0] = False
480: (8)                   for f in self.nanfuncs:

```

```

481: (12)                     reference = 26 if f is np.nansum else 2240
482: (12)                     ret = f(ar, where=where, initial=2)
483: (12)                     assert ret.dtype == dtype
484: (12)                     assert ret == reference
485: (0)  class TestNanFunctions_CumSumProd(SharedNanFunctionsTestsMixin):
486: (4)      nanfuncs = [np.nancumsum, np.nancumprod]
487: (4)      stdfuncs = [np.cumsum, np.cumprod]
488: (4)      @pytest.mark.parametrize("axis", [None, 0, 1])
489: (4)      @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
490: (4)      @pytest.mark.parametrize("array", [
491: (8)          np.array(np.nan),
492: (8)          np.full((3, 3), np.nan)
493: (4)      ], ids=["0d", "2d"])
494: (4)      def test_allnans(self, axis, dtype, array):
495: (8)          if axis is not None and array.ndim == 0:
496: (12)              pytest.skip(f"axis != None` not supported for 0d arrays")
497: (8)          array = array.astype(dtype)
498: (8)          for func, identity in zip(self.nanfuncs, [0, 1]):
499: (12)              out = func(array)
500: (12)              assert np.all(out == identity)
501: (12)              assert out.dtype == array.dtype
502: (4)      def test_empty(self):
503: (8)          for f, tgt_value in zip(self.nanfuncs, [0, 1]):
504: (12)              mat = np.zeros((0, 3))
505: (12)              tgt = tgt_value*np.ones((0, 3))
506: (12)              res = f(mat, axis=0)
507: (12)              assert_equal(res, tgt)
508: (12)              tgt = mat
509: (12)              res = f(mat, axis=1)
510: (12)              assert_equal(res, tgt)
511: (12)              tgt = np.zeros((0))
512: (12)              res = f(mat, axis=None)
513: (12)              assert_equal(res, tgt)
514: (4)      def test_keepdims(self):
515: (8)          for f, g in zip(self.nanfuncs, self.stdfuncs):
516: (12)              mat = np.eye(3)
517: (12)              for axis in [None, 0, 1]:
518: (16)                  tgt = f(mat, axis=axis, out=None)
519: (16)                  res = g(mat, axis=axis, out=None)
520: (16)                  assert_(res.ndim == tgt.ndim)
521: (8)          for f in self.nanfuncs:
522: (12)              d = np.ones((3, 5, 7, 11))
523: (12)              rs = np.random.RandomState(0)
524: (12)              d[rs.rand(*d.shape) < 0.5] = np.nan
525: (12)              res = f(d, axis=None)
526: (12)              assert_equal(res.shape, (1155,))
527: (12)              for axis in np.arange(4):
528: (16)                  res = f(d, axis=axis)
529: (16)                  assert_equal(res.shape, (3, 5, 7, 11))
530: (4)      def test_result_values(self):
531: (8)          for axis in (-2, -1, 0, 1, None):
532: (12)              tgt = np.cumprod(_ndat_ones, axis=axis)
533: (12)              res = np.nancumprod(_ndat, axis=axis)
534: (12)              assert_almost_equal(res, tgt)
535: (12)              tgt = np.cumsum(_ndat_zeros, axis=axis)
536: (12)              res = np.nancumsum(_ndat, axis=axis)
537: (12)              assert_almost_equal(res, tgt)
538: (4)      def test_out(self):
539: (8)          mat = np.eye(3)
540: (8)          for nf, rf in zip(self.nanfuncs, self.stdfuncs):
541: (12)              resout = np.eye(3)
542: (12)              for axis in (-2, -1, 0, 1):
543: (16)                  tgt = rf(mat, axis=axis)
544: (16)                  res = nf(mat, axis=axis, out=resout)
545: (16)                  assert_almost_equal(res, resout)
546: (16)                  assert_almost_equal(res, tgt)
547: (0)  class TestNanFunctions_MeanVarStd(SharedNanFunctionsTestsMixin):
548: (4)      nanfuncs = [np.nanmean, np.nanvar, np.nanstd]
549: (4)      stdfuncs = [np.mean, np.var, np.std]

```

```

550: (4) def test_dtype_error(self):
551: (8)     for f in self.nanfuncs:
552: (12)         for dtype in [np.bool_, np.int_, np.object_]:
553: (16)             assert_raises(TypeError, f, _ndat, axis=1, dtype=dtype)
554: (4) def test_out_dtype_error(self):
555: (8)     for f in self.nanfuncs:
556: (12)         for dtype in [np.bool_, np.int_, np.object_]:
557: (16)             out = np.empty(_ndat.shape[0], dtype=dtype)
558: (16)             assert_raises(TypeError, f, _ndat, axis=1, out=out)
559: (4) def test_ddof(self):
560: (8)     nanfuncs = [np.nanvar, np.nanstd]
561: (8)     stdfuncs = [np.var, np.std]
562: (8)     for nf, rf in zip(nanfuncs, stdfuncs):
563: (12)         for ddof in [0, 1]:
564: (16)             tgt = [rf(d, ddof=ddof) for d in _rdat]
565: (16)             res = nf(_ndat, axis=1, ddof=ddof)
566: (16)             assert_almost_equal(res, tgt)
567: (4) def test_ddof_too_big(self):
568: (8)     nanfuncs = [np.nanvar, np.nanstd]
569: (8)     stdfuncs = [np.var, np.std]
570: (8)     dsize = [len(d) for d in _rdat]
571: (8)     for nf, rf in zip(nanfuncs, stdfuncs):
572: (12)         for ddof in range(5):
573: (16)             with suppress_warnings() as sup:
574: (20)                 sup.record(RuntimeWarning)
575: (20)                 sup.filter(np.ComplexWarning)
576: (20)                 tgt = [ddof >= d for d in dsize]
577: (20)                 res = nf(_ndat, axis=1, ddof=ddof)
578: (20)                 assert_equal(np.isnan(res), tgt)
579: (20)                 if any(tgt):
580: (24)                     assert_(len(sup.log) == 1)
581: (20)                 else:
582: (24)                     assert_(len(sup.log) == 0)
583: (4) @pytest.mark.parametrize("axis", [None, 0, 1])
584: (4) @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
585: (4) @pytest.mark.parametrize("array", [
586: (8)     np.array(np.nan),
587: (8)     np.full((3, 3), np.nan),
588: (4) ], ids=["0d", "2d"])
589: (4) def test_allnans(self, axis, dtype, array):
590: (8)     if axis is not None and array.ndim == 0:
591: (12)         pytest.skip(f"`axis != None` not supported for 0d arrays")
592: (8)     array = array.astype(dtype)
593: (8)     match = "(Degrees of freedom <= 0 for slice.)|(Mean of empty slice)"
594: (8)     for func in self.nanfuncs:
595: (12)         with pytest.warns(RuntimeWarning, match=match):
596: (16)             out = func(array, axis=axis)
597: (12)             assert np.isnan(out).all()
598: (12)             if func is np.nanmean:
599: (16)                 assert out.dtype == array.dtype
600: (12)             else:
601: (16)                 assert out.dtype == np.abs(array).dtype
602: (4) def test_empty(self):
603: (8)     mat = np.zeros((0, 3))
604: (8)     for f in self.nanfuncs:
605: (12)         for axis in [0, None]:
606: (16)             with warnings.catch_warnings(record=True) as w:
607: (20)                 warnings.simplefilter('always')
608: (20)                 assert_(np.isnan(f(mat, axis=axis)).all())
609: (20)                 assert_(len(w) == 1)
610: (20)                 assert_(issubclass(w[0].category, RuntimeWarning))
611: (12)         for axis in [1]:
612: (16)             with warnings.catch_warnings(record=True) as w:
613: (20)                 warnings.simplefilter('always')
614: (20)                 assert_equal(f(mat, axis=axis), np.zeros([]))
615: (20)                 assert_(len(w) == 0)
616: (4) @pytest.mark.parametrize("dtype", np.typecodes["AllFloat"])
617: (4) def test_where(self, dtype):
618: (8)     ar = np.arange(9).reshape(3, 3).astype(dtype)

```

```

619: (8)             ar[0, :] = np.nan
620: (8)             where = np.ones_like(ar, dtype=np.bool_)
621: (8)             where[:, 0] = False
622: (8)             for f, f_std in zip(self.nanfuncs, self.stdfuncs):
623: (12)                 reference = f_std(ar[where][2:])
624: (12)                 dtype_reference = dtype if f is np.nanmean else ar.real.dtype
625: (12)                 ret = f(ar, where=where)
626: (12)                 assert ret.dtype == dtype_reference
627: (12)                 np.testing.assert_allclose(ret, reference)
628: (0)             _TIME_UNITS = (
629: (4)                 "Y", "M", "W", "D", "h", "m", "s", "ms", "us", "ns", "ps", "fs", "as"
630: (0)             )
631: (0)             _TYPE_CODES = list(np.typecodes["AllFloat"])
632: (0)             _TYPE_CODES += [f"m8[{unit}]" for unit in _TIME_UNITS]
633: (0)             class TestNanFunctions_Median:
634: (4)                 def test_mutation(self):
635: (8)                     ndat = _ndat.copy()
636: (8)                     np.nanmedian(ndat)
637: (8)                     assert_equal(ndat, _ndat)
638: (4)                 def test_keepldims(self):
639: (8)                     mat = np.eye(3)
640: (8)                     for axis in [None, 0, 1]:
641: (12)                         tgt = np.median(mat, axis=axis, out=None, overwrite_input=False)
642: (12)                         res = np.nanmedian(mat, axis=axis, out=None,
643: (12)                                         overwrite_input=False)
644: (8)                         assert_(res.ndim == tgt.ndim)
645: (8)                         d = np.ones((3, 5, 7, 11))
646: (8)                         w = np.random.random((4, 200)) * np.array(d.shape)[:, None]
647: (8)                         w = w.astype(np.intp)
648: (8)                         d[tuple(w)] = np.nan
649: (12)                         with suppress_warnings() as sup:
650: (12)                             sup.filter(RuntimeWarning)
651: (12)                             res = np.nanmedian(d, axis=None, keepdims=True)
652: (12)                             assert_equal(res.shape, (1, 1, 1, 1))
653: (12)                             res = np.nanmedian(d, axis=(0, 1), keepdims=True)
654: (12)                             assert_equal(res.shape, (1, 1, 7, 11))
655: (12)                             res = np.nanmedian(d, axis=(0, 3), keepdims=True)
656: (12)                             assert_equal(res.shape, (1, 5, 7, 1))
657: (12)                             res = np.nanmedian(d, axis=(1,), keepdims=True)
658: (12)                             assert_equal(res.shape, (3, 1, 7, 11))
659: (12)                             res = np.nanmedian(d, axis=(0, 1, 2, 3), keepdims=True)
660: (12)                             assert_equal(res.shape, (1, 1, 1, 1))
661: (12)                             res = np.nanmedian(d, axis=(0, 1, 3), keepdims=True)
662: (4)                             assert_equal(res.shape, (1, 1, 7, 1))
663: (8)             @pytest.mark.parametrize(
664: (8)                 argnames='axis',
665: (12)                 argvalues=[
666: (12)                     None,
667: (12)                     1,
668: (12)                     (1, ),
669: (12)                     (0, 1),
670: (8)                     (-3, -1),
671: (4)                 ]
672: (4)             )
673: (4)             @pytest.mark.filterwarnings("ignore>All-NaN slice:RuntimeWarning")
674: (4)             def test_keepldims_out(self, axis):
675: (8)                 d = np.ones((3, 5, 7, 11))
676: (8)                 w = np.random.random((4, 200)) * np.array(d.shape)[:, None]
677: (8)                 w = w.astype(np.intp)
678: (8)                 d[tuple(w)] = np.nan
679: (12)                 if axis is None:
680: (8)                     shape_out = (1,) * d.ndim
681: (12)                 else:
682: (12)                     axis_norm = normalize_axis_tuple(axis, d.ndim)
683: (16)                     shape_out = tuple(
684: (8)                         1 if i in axis_norm else d.shape[i] for i in range(d.ndim))
685: (8)                     out = np.empty(shape_out)
686: (8)                     result = np.nanmedian(d, axis=axis, keepdims=True, out=out)
687: (8)                     assert result is out

```

```

687: (8)             assert_equal(result.shape, shape_out)
688: (4) def test_out(self):
689: (8)     mat = np.random.rand(3, 3)
690: (8)     nan_mat = np.insert(mat, [0, 2], np.nan, axis=1)
691: (8)     resout = np.zeros(3)
692: (8)     tgt = np.median(mat, axis=1)
693: (8)     res = np.nanmedian(nan_mat, axis=1, out=resout)
694: (8)     assert_almost_equal(res, resout)
695: (8)     assert_almost_equal(res, tgt)
696: (8)     resout = np.zeros(())
697: (8)     tgt = np.median(mat, axis=None)
698: (8)     res = np.nanmedian(nan_mat, axis=None, out=resout)
699: (8)     assert_almost_equal(res, resout)
700: (8)     assert_almost_equal(res, tgt)
701: (8)     res = np.nanmedian(nan_mat, axis=(0, 1), out=resout)
702: (8)     assert_almost_equal(res, resout)
703: (8)     assert_almost_equal(res, tgt)
704: (4) def test_small_large(self):
705: (8)     for s in [5, 20, 51, 200, 1000]:
706: (12)         d = np.random.randn(4, s)
707: (12)         w = np.random.randint(0, d.size, size=d.size // 5)
708: (12)         d.ravel()[w] = np.nan
709: (12)         d[:, 0] = 1. # ensure at least one good value
710: (12)         tgt = []
711: (12)         for x in d:
712: (16)             nonan = np.compress(~np.isnan(x), x)
713: (16)             tgt.append(np.median(nanon, overwrite_input=True))
714: (12)             assert_array_equal(np.nanmedian(d, axis=-1), tgt)
715: (4) def test_result_values(self):
716: (12)     tgt = [np.median(d) for d in _rdat]
717: (12)     res = np.nanmedian(_ndat, axis=1)
718: (12)     assert_almost_equal(res, tgt)
719: (4) @pytest.mark.parametrize("axis", [None, 0, 1])
720: (4) @pytest.mark.parametrize("dtype", _TYPE_CODES)
721: (4) def test_allnans(self, dtype, axis):
722: (8)     mat = np.full((3, 3), np.nan).astype(dtype)
723: (8)     with suppress_warnings() as sup:
724: (12)         sup.record(RuntimeWarning)
725: (12)         output = np.nanmedian(mat, axis=axis)
726: (12)         assert output.dtype == mat.dtype
727: (12)         assert np.isnan(output).all()
728: (12)         if axis is None:
729: (16)             assert_(len(sup.log) == 1)
730: (12)         else:
731: (16)             assert_(len(sup.log) == 3)
732: (12)         scalar = np.array(np.nan).astype(dtype)[()]
733: (12)         output_scalar = np.nanmedian(scalar)
734: (12)         assert output_scalar.dtype == scalar.dtype
735: (12)         assert np.isnan(output_scalar)
736: (12)         if axis is None:
737: (16)             assert_(len(sup.log) == 2)
738: (12)         else:
739: (16)             assert_(len(sup.log) == 4)
740: (4) def test_empty(self):
741: (8)     mat = np.zeros((0, 3))
742: (8)     for axis in [0, None]:
743: (12)         with warnings.catch_warnings(record=True) as w:
744: (16)             warnings.simplefilter('always')
745: (16)             assert_(np.isnan(np.nanmedian(mat, axis=axis)).all())
746: (16)             assert_(len(w) == 1)
747: (16)             assert_(issubclass(w[0].category, RuntimeWarning))
748: (8)         for axis in [1]:
749: (12)             with warnings.catch_warnings(record=True) as w:
750: (16)                 warnings.simplefilter('always')
751: (16)                 assert_equal(np.nanmedian(mat, axis=axis), np.zeros([]))
752: (16)                 assert_(len(w) == 0)
753: (4) def test_scalar(self):
754: (8)     assert_(np.nanmedian(0.) == 0.)
755: (4) def test_extended_axis_invalid(self):

```

```

756: (8)             d = np.ones((3, 5, 7, 11))
757: (8)             assert_raises(np.AxisError, np.nanmedian, d, axis=-5)
758: (8)             assert_raises(np.AxisError, np.nanmedian, d, axis=(0, -5))
759: (8)             assert_raises(np.AxisError, np.nanmedian, d, axis=4)
760: (8)             assert_raises(np.AxisError, np.nanmedian, d, axis=(0, 4))
761: (8)             assert_raises(ValueError, np.nanmedian, d, axis=(1, 1))
762: (4)              def test_float_special(self):
763: (8)                  with suppress_warnings() as sup:
764: (12)                      sup.filter(RuntimeWarning)
765: (12)                      for inf in [np.inf, -np.inf]:
766: (16)                          a = np.array([[inf, np.nan], [np.nan, np.nan]])
767: (16)                          assert_equal(np.nanmedian(a, axis=0), [inf, np.nan])
768: (16)                          assert_equal(np.nanmedian(a, axis=1), [inf, np.nan])
769: (16)                          assert_equal(np.nanmedian(a), inf)
770: (16)                          a = np.array([[np.nan, np.nan, inf],
771: (29)                                [np.nan, np.nan, inf]]])
772: (16)                          assert_equal(np.nanmedian(a), inf)
773: (16)                          assert_equal(np.nanmedian(a, axis=0), [np.nan, np.nan, inf])
774: (16)                          assert_equal(np.nanmedian(a, axis=1), inf)
775: (16)                          a = np.array([[inf, inf], [inf, inf]])
776: (16)                          assert_equal(np.nanmedian(a, axis=1), inf)
777: (16)                          a = np.array([[inf, 7, -inf, -9],
778: (30)                            [-10, np.nan, np.nan, 5],
779: (30)                            [4, np.nan, np.nan, inf]], dtype=np.float32)
780: (30)
781: (16)                      if inf > 0:
782: (20)                          assert_equal(np.nanmedian(a, axis=0), [4., 7., -inf, 5.])
783: (20)                          assert_equal(np.nanmedian(a), 4.5)
784: (16)                      else:
785: (20)                          assert_equal(np.nanmedian(a, axis=0), [-10., 7., -inf,
786: (20)                            -9.])
787: (16)                          assert_equal(np.nanmedian(a), -2.5)
788: (16)                      assert_equal(np.nanmedian(a, axis=-1), [-1., -2.5, inf])
789: (20)                      for i in range(0, 10):
790: (24)                          for j in range(1, 10):
791: (24)                              a = np.array([(np.nan * i) + (inf * j)] * 2)
792: (24)                              assert_equal(np.nanmedian(a), inf)
793: (24)                              assert_equal(np.nanmedian(a, axis=1), inf)
794: (37)                              assert_equal(np.nanmedian(a, axis=0),
795: (24)                                ([np.nan * i] + [inf * j]))
796: (24)                              a = np.array([(np.nan * i) + (-inf * j)] * 2)
797: (24)                              assert_equal(np.nanmedian(a), -inf)
798: (24)                              assert_equal(np.nanmedian(a, axis=1), -inf)
799: (37)                              assert_equal(np.nanmedian(a, axis=0),
800: (0)                                ([np.nan * i] + [-inf * j]))
801: (4)              class TestNanFunctions_Percentile:
802: (8)                  def test_mutation(self):
803: (8)                      ndat = _ndat.copy()
804: (8)                      np.nanpercentile(ndat, 30)
805: (4)                      assert_equal(ndat, _ndat)
806: (8)                  def test_kepdims(self):
807: (8)                      mat = np.eye(3)
808: (12)                      for axis in [None, 0, 1]:
809: (32)                          tgt = np.percentile(mat, 70, axis=axis, out=None,
810: (12)                            overwrite_input=False)
811: (35)                          res = np.nanpercentile(mat, 70, axis=axis, out=None,
812: (12)                            overwrite_input=False)
813: (8)                          assert_(res.ndim == tgt.ndim)
814: (8)                      d = np.ones((3, 5, 7, 11))
815: (8)                      w = np.random.random((4, 200)) * np.array(d.shape)[:, None]
816: (8)                      w = w.astype(np.intp)
817: (8)                      d[tuple(w)] = np.nan
818: (12)                      with suppress_warnings() as sup:
819: (12)                          sup.filter(RuntimeWarning)
820: (12)                          res = np.nanpercentile(d, 90, axis=None, keepdims=True)
821: (12)                          assert_equal(res.shape, (1, 1, 1, 1))
822: (12)                          res = np.nanpercentile(d, 90, axis=(0, 1), keepdims=True)
823: (12)                          assert_equal(res.shape, (1, 1, 7, 11))
824: (12)                          res = np.nanpercentile(d, 90, axis=(0, 3), keepdims=True)

```

```

824: (12)             assert_equal(res.shape, (1, 5, 7, 1))
825: (12)             res = np.nanpercentile(d, 90, axis=(1,), keepdims=True)
826: (12)             assert_equal(res.shape, (3, 1, 7, 11))
827: (12)             res = np.nanpercentile(d, 90, axis=(0, 1, 2, 3), keepdims=True)
828: (12)             assert_equal(res.shape, (1, 1, 1, 1))
829: (12)             res = np.nanpercentile(d, 90, axis=(0, 1, 3), keepdims=True)
830: (12)             assert_equal(res.shape, (1, 1, 7, 1))
831: (4) @pytest.mark.parametrize('q', [7, [1, 7]])
832: (4) @pytest.mark.parametrize(
833: (8)     argnames='axis',
834: (8)     argvalues=[
835: (12)         None,
836: (12)         1,
837: (12)         (1,),
838: (12)         (0, 1),
839: (12)         (-3, -1),
840: (8)     ]
841: (4) )
842: (4) @pytest.mark.filterwarnings("ignore>All-NaN slice:RuntimeWarning")
843: (4) def test_keepdims_out(self, q, axis):
844: (8)     d = np.ones((3, 5, 7, 11))
845: (8)     w = np.random.random((4, 200)) * np.array(d.shape)[:, None]
846: (8)     w = w.astype(np.intp)
847: (8)     d[tuple(w)] = np.nan
848: (8)     if axis is None:
849: (12)         shape_out = (1,) * d.ndim
850: (8)     else:
851: (12)         axis_norm = normalize_axis_tuple(axis, d.ndim)
852: (12)         shape_out = tuple(
853: (16)             1 if i in axis_norm else d.shape[i] for i in range(d.ndim))
854: (8)     shape_out = np.shape(q) + shape_out
855: (8)     out = np.empty(shape_out)
856: (8)     result = np.nanpercentile(d, q, axis=axis, keepdims=True, out=out)
857: (8)     assert result is out
858: (8)     assert_equal(result.shape, shape_out)
859: (4) def test_out(self):
860: (8)     mat = np.random.rand(3, 3)
861: (8)     nan_mat = np.insert(mat, [0, 2], np.nan, axis=1)
862: (8)     resout = np.zeros(3)
863: (8)     tgt = np.percentile(mat, 42, axis=1)
864: (8)     res = np.nanpercentile(nan_mat, 42, axis=1, out=resout)
865: (8)     assert_almost_equal(res, resout)
866: (8)     assert_almost_equal(res, tgt)
867: (8)     resout = np.zeros(())
868: (8)     tgt = np.percentile(mat, 42, axis=None)
869: (8)     res = np.nanpercentile(nan_mat, 42, axis=None, out=resout)
870: (8)     assert_almost_equal(res, resout)
871: (8)     assert_almost_equal(res, tgt)
872: (8)     res = np.nanpercentile(nan_mat, 42, axis=(0, 1), out=resout)
873: (8)     assert_almost_equal(res, resout)
874: (8)     assert_almost_equal(res, tgt)
875: (4) def test_complex(self):
876: (8)     arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='G')
877: (8)     assert_raises(TypeError, np.nanpercentile, arr_c, 0.5)
878: (8)     arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='D')
879: (8)     assert_raises(TypeError, np.nanpercentile, arr_c, 0.5)
880: (8)     arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='F')
881: (8)     assert_raises(TypeError, np.nanpercentile, arr_c, 0.5)
882: (4) def test_result_values(self):
883: (8)     tgt = [np.percentile(d, 28) for d in _rdat]
884: (8)     res = np.nanpercentile(_ndat, 28, axis=1)
885: (8)     assert_almost_equal(res, tgt)
886: (8)     tgt = np.transpose([np.percentile(d, (28, 98)) for d in _rdat])
887: (8)     res = np.nanpercentile(_ndat, (28, 98), axis=1)
888: (8)     assert_almost_equal(res, tgt)
889: (4) @pytest.mark.parametrize("axis", [None, 0, 1])
890: (4) @pytest.mark.parametrize("dtype", np.typecodes["Float"])
891: (4) @pytest.mark.parametrize("array", [
892: (8)     np.array(np.nan),

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

893: (8) np.full((3, 3), np.nan),
894: (4) ], ids=["0d", "2d"])
895: (4) def test_allnans(self, axis, dtype, array):
896: (8)     if axis is not None and array.ndim == 0:
897: (12)         pytest.skip(f"`axis != None` not supported for 0d arrays")
898: (8)     array = array.astype(dtype)
899: (8)     with pytest.warns(RuntimeWarning, match="All-NaN slice encountered"):
900: (12)         out = np.nanpercentile(array, 60, axis=axis)
901: (8)     assert np.isnan(out).all()
902: (8)     assert out.dtype == array.dtype
903: (4) def test_empty(self):
904: (8)     mat = np.zeros((0, 3))
905: (8)     for axis in [0, None]:
906: (12)         with warnings.catch_warnings(record=True) as w:
907: (16)             warnings.simplefilter('always')
908: (16)             assert_(np.isnan(np.nanpercentile(mat, 40, axis=axis)).all())
909: (16)             assert_(len(w) == 1)
910: (16)             assert_(issubclass(w[0].category, RuntimeWarning))
911: (8)     for axis in [1]:
912: (12)         with warnings.catch_warnings(record=True) as w:
913: (16)             warnings.simplefilter('always')
914: (16)             assert_equal(np.nanpercentile(mat, 40, axis=axis),
915: (16)                         assert_(len(w) == 0)
916: (4) def test_scalar(self):
917: (8)     assert_equal(np.nanpercentile(0., 100), 0.)
918: (8)     a = np.arange(6)
919: (8)     r = np.nanpercentile(a, 50, axis=0)
920: (8)     assert_equal(r, 2.5)
921: (8)     assert_(np.isscalar(r))
922: (4) def test_extended_axis_invalid(self):
923: (8)     d = np.ones((3, 5, 7, 11))
924: (8)     assert_raises(np.AxisError, np.nanpercentile, d, q=5, axis=-5)
925: (8)     assert_raises(np.AxisError, np.nanpercentile, d, q=5, axis=(0, -5))
926: (8)     assert_raises(np.AxisError, np.nanpercentile, d, q=5, axis=4)
927: (8)     assert_raises(np.AxisError, np.nanpercentile, d, q=5, axis=(0, 4))
928: (8)     assert_raises(ValueError, np.nanpercentile, d, q=5, axis=(1, 1))
929: (4) def test_multiple_percentiles(self):
930: (8)     perc = [50, 100]
931: (8)     mat = np.ones((4, 3))
932: (8)     nan_mat = np.nan * mat
933: (8)     large_mat = np.ones((3, 4, 5))
934: (8)     large_mat[:, 0:2:4, :] = 0
935: (8)     large_mat[:, :, 3:] *= 2
936: (8)     for axis in [None, 0, 1]:
937: (12)         for keepdim in [False, True]:
938: (16)             with suppress_warnings() as sup:
939: (20)                 sup.filter(RuntimeWarning, "All-NaN slice encountered")
940: (20)                 val = np.percentile(mat, perc, axis=axis,
keepdims=keepdim)
941: (20)                 nan_val = np.nanpercentile(nan_mat, perc, axis=axis,
942: (47)                               keepdims=keepdim)
943: (20)                 assert_equal(nan_val.shape, val.shape)
944: (20)                 val = np.percentile(large_mat, perc, axis=axis,
945: (40)                               keepdims=keepdim)
946: (20)                 nan_val = np.nanpercentile(large_mat, perc, axis=axis,
947: (47)                               keepdims=keepdim)
948: (20)                 assert_equal(nan_val, val)
949: (8)                 megamat = np.ones((3, 4, 5, 6))
950: (8)                 assert_equal(np.nanpercentile(megamat, perc, axis=(1, 2)).shape, (2,
3, 6))
951: (0) class TestNanFunctions_Quantile:
952: (4)     def test_regression(self):
953: (8)         ar = np.arange(24).reshape(2, 3, 4).astype(float)
954: (8)         ar[0][1] = np.nan
955: (8)         assert_equal(np.nanquantile(ar, q=0.5), np.nanpercentile(ar, q=50))
956: (8)         assert_equal(np.nanquantile(ar, q=0.5, axis=0),
957: (21)                           np.nanpercentile(ar, q=50, axis=0))
958: (8)         assert_equal(np.nanquantile(ar, q=0.5, axis=1),

```

```

959: (21)                         np.nanpercentile(ar, q=50, axis=1))
960: (8)                          assert_equal(np.nanquantile(ar, q=[0.5], axis=1),
961: (21)                                np.nanpercentile(ar, q=[50], axis=1))
962: (8)                          assert_equal(np.nanquantile(ar, q=[0.25, 0.5, 0.75], axis=1),
963: (21)                                np.nanpercentile(ar, q=[25, 50, 75], axis=1))
964: (4)                           def test_basic(self):
965: (8)                             x = np.arange(8) * 0.5
966: (8)                             assert_equal(np.nanquantile(x, 0), 0.)
967: (8)                             assert_equal(np.nanquantile(x, 1), 3.5)
968: (8)                             assert_equal(np.nanquantile(x, 0.5), 1.75)
969: (4)                           def test_complex(self):
970: (8)                             arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='G')
971: (8)                             assert_raises(TypeError, np.nanquantile, arr_c, 0.5)
972: (8)                             arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='D')
973: (8)                             assert_raises(TypeError, np.nanquantile, arr_c, 0.5)
974: (8)                             arr_c = np.array([0.5+3.0j, 2.1+0.5j, 1.6+2.3j], dtype='F')
975: (8)                             assert_raises(TypeError, np.nanquantile, arr_c, 0.5)
976: (4)                           def test_no_p_overwrite(self):
977: (8)                             p0 = np.array([0, 0.75, 0.25, 0.5, 1.0])
978: (8)                             p = p0.copy()
979: (8)                             np.nanquantile(np.arange(100.), p, method="midpoint")
980: (8)                             assert_array_equal(p, p0)
981: (8)                             p0 = p0.tolist()
982: (8)                             p = p.tolist()
983: (8)                             np.nanquantile(np.arange(100.), p, method="midpoint")
984: (8)                             assert_array_equal(p, p0)
985: (4) @pytest.mark.parametrize("axis", [None, 0, 1])
986: (4) @pytest.mark.parametrize("dtype", np.typecodes["Float"])
987: (4) @pytest.mark.parametrize("array", [
988: (8)     np.array(np.nan),
989: (8)     np.full((3, 3), np.nan),
990: (4) ], ids=["0d", "2d"])
991: (4) def test_allnans(self, axis, dtype, array):
992: (8)     if axis is not None and array.ndim == 0:
993: (12)         pytest.skip(f"`axis != None` not supported for 0d arrays")
994: (8)     array = array.astype(dtype)
995: (8)     with pytest.warns(RuntimeWarning, match="All-NaN slice encountered"):
996: (12)         out = np.nanquantile(array, 1, axis=axis)
997: (8)         assert np.isnan(out).all()
998: (8)         assert out.dtype == array.dtype
999: (0) @pytest.mark.parametrize("arr, expected", [
1000: (4)     (np.array([np.nan, 5.0, np.nan, np.inf]),
1001: (5)       np.array([False, True, False, True])),
1002: (4)     (np.array([1, 5, 7, 9], dtype=np.int64),
1003: (5)       True),
1004: (4)     (np.array([False, True, False, True]),
1005: (5)       True),
1006: (4)     (np.array([[np.nan, 5.0],
1007: (15)           [np.nan, np.inf]], dtype=np.complex64),
1008: (5)       np.array([[False, True],
1009: (15)           [False, True]])),
1010: (4)   ])
1011: (0) def test_nan_mask(arr, expected):
1012: (4)     for out in [None, np.empty(arr.shape, dtype=np.bool_)]:
1013: (8)         actual = _nan_mask(arr, out=out)
1014: (8)         assert_equal(actual, expected)
1015: (8)         if type(expected) is not np.ndarray:
1016: (12)             assert actual is True
1017: (0) def test_replace_nan():
1018: (4)     """ Test that _replace_nan returns the original array if there are no
1019: (4)     NaNs, not a copy.
1020: (4)     """
1021: (4)     for dtype in [np.bool_, np.int32, np.int64]:
1022: (8)         arr = np.array([0, 1], dtype=dtype)
1023: (8)         result, mask = _replace_nan(arr, 0)
1024: (8)         assert mask is None
1025: (8)         assert result is arr
1026: (4)         for dtype in [np.float32, np.float64]:
1027: (8)             arr = np.array([0, 1], dtype=dtype)

```

```

1028: (8)             result, mask = _replace_nan(arr, 2)
1029: (8)             assert (mask == False).all()
1030: (8)             assert result is not arr
1031: (8)             assert_equal(result, arr)
1032: (8)             arr_nan = np.array([0, 1, np.nan], dtype=dtype)
1033: (8)             result_nan, mask_nan = _replace_nan(arr_nan, 2)
1034: (8)             assert_equal(mask_nan, np.array([False, False, True]))
1035: (8)             assert result_nan is not arr_nan
1036: (8)             assert_equal(result_nan, np.array([0, 1, 2]))
1037: (8)             assert np.isnan(arr_nan[-1])

```

-----

## File 242 - test\_polynomial.py:

```

1: (0)             import numpy as np
2: (0)             from numpy.testing import (
3: (4)                 assert_, assert_equal, assert_array_equal, assert_almost_equal,
4: (4)                 assert_array_almost_equal, assert_raises, assert_allclose
5: (4)             )
6: (0)             import pytest
7: (0)             TYPE_CODES = np.typecodes["AllInteger"] + np.typecodes["AllFloat"] + "0"
8: (0)             class TestPolynomial:
9: (4)                 def test_poly1d_str_and_repr(self):
10: (8)                   p = np.poly1d([1., 2, 3])
11: (8)                   assert_equal(repr(p), 'poly1d([1., 2., 3.])')
12: (8)                   assert_equal(str(p),
13: (21)                     '2\n'
14: (21)                     '1 x + 2 x + 3')
15: (8)                   q = np.poly1d([3., 2, 1])
16: (8)                   assert_equal(repr(q), 'poly1d([3., 2., 1.])')
17: (8)                   assert_equal(str(q),
18: (21)                     '2\n'
19: (21)                     '3 x + 2 x + 1')
20: (8)                   r = np.poly1d([1.89999 + 2j, -3j, -5.12345678, 2 + 1j])
21: (8)                   assert_equal(str(r),
22: (21)                     '3\n'
23: (21)                     '(1.9 + 2j) x - 3j x - 5.123 x + (2 + 1j)')
24: (8)                   assert_equal(str(np.poly1d([-3, -2, -1])),
25: (21)                     '2\n'
26: (21)                     '-3 x - 2 x - 1')
27: (4)                 def test_poly1d_resolution(self):
28: (8)                   p = np.poly1d([1., 2, 3])
29: (8)                   q = np.poly1d([3., 2, 1])
30: (8)                   assert_equal(p(0), 3.0)
31: (8)                   assert_equal(p(5), 38.0)
32: (8)                   assert_equal(q(0), 1.0)
33: (8)                   assert_equal(q(5), 86.0)
34: (4)                 def test_poly1d_math(self):
35: (8)                   p = np.poly1d([1., 2, 4])
36: (8)                   q = np.poly1d([4., 2, 1])
37: (8)                   assert_equal(p/q, (np.poly1d([0.25]), np.poly1d([1.5, 3.75])))
38: (8)                   assert_equal(p.integ(), np.poly1d([1/3, 1., 4., 0.]))
39: (8)                   assert_equal(p.integ(1), np.poly1d([1/3, 1., 4., 0.]))
40: (8)                   p = np.poly1d([1., 2, 3])
41: (8)                   q = np.poly1d([3., 2, 1])
42: (8)                   assert_equal(p * q, np.poly1d([3., 8., 14., 8., 3.]))
43: (8)                   assert_equal(p + q, np.poly1d([4., 4., 4.]))
44: (8)                   assert_equal(p - q, np.poly1d([-2., 0., 2.]))
45: (8)                   assert_equal(p ** 4, np.poly1d([1., 8., 36., 104., 214., 312., 324.,
216., 81.])))
46: (8)                   assert_equal(p(q), np.poly1d([9., 12., 16., 8., 6.]))
47: (8)                   assert_equal(q(p), np.poly1d([3., 12., 32., 40., 34.]))
48: (8)                   assert_equal(p.deriv(), np.poly1d([2., 2.]))
49: (8)                   assert_equal(p.deriv(2), np.poly1d([2.]))
50: (8)                   assert_equal(np.polydiv(np.poly1d([1, 0, -1]), np.poly1d([1, 1])),
51: (21)                     (np.poly1d([1., -1.]), np.poly1d([0.])))
52: (4)                   @pytest.mark.parametrize("type_code", TYPE_CODES)
53: (4)                   def test_poly1d_misc(self, type_code: str) -> None:

```

```

54: (8)          dtype = np.dtype(type_code)
55: (8)          ar = np.array([1, 2, 3], dtype=dtype)
56: (8)          p = np.poly1d(ar)
57: (8)          assert_equal(np.asarray(p), ar)
58: (8)          assert_equal(np.asarray(p).dtype, dtype)
59: (8)          assert_equal(len(p), 2)
60: (8)          comparison_dct = {-1: 0, 0: 3, 1: 2, 2: 1, 3: 0}
61: (8)          for index, ref in comparison_dct.items():
62: (12)             scalar = p[index]
63: (12)             assert_equal(scalar, ref)
64: (12)             if dtype == np.object_:
65: (16)                 assert isinstance(scalar, int)
66: (12)             else:
67: (16)                 assert_equal(scalar.dtype, dtype)
68: (4)          def test_poly1d_variable_arg(self):
69: (8)              q = np.poly1d([1., 2, 3], variable='y')
70: (8)              assert_equal(str(q),
71: (21)                  '2\n'
72: (21)                  '1 y + 2 y + 3')
73: (8)              q = np.poly1d([1., 2, 3], variable='lambda')
74: (8)              assert_equal(str(q),
75: (21)                  '2\n'
76: (21)                  '1 lambda + 2 lambda + 3')
77: (4)          def test_poly(self):
78: (8)              assert_array_almost_equal(np.poly([3, -np.sqrt(2), np.sqrt(2)]),
79: (34)                  [1, -3, -2, 6])
80: (8)              A = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
81: (8)              assert_array_almost_equal(np.poly(A), [1, -6, -72, -27])
82: (8)              assert_(np.isrealobj(np.poly([+1.082j, +2.613j, -2.613j, -1.082j])))
83: (8)              assert_(np.isrealobj(np.poly([0+1j, -0+-1j, 1+2j,
84: (38)                  1-2j, 1.+3.5j, 1-3.5j])))
85: (8)              assert_(np.isrealobj(np.poly([1j, -1j, 1+2j, 1-2j, 1+3j, 1-3.j])))
86: (8)              assert_(np.isrealobj(np.poly([1j, -1j, 1+2j, 1-2j])))
87: (8)              assert_(np.isrealobj(np.poly([1j, -1j, 2j, -2j])))
88: (8)              assert_(np.isrealobj(np.poly([1j, -1j])))
89: (8)              assert_(np.isrealobj(np.poly([1, -1])))
90: (8)              assert_(np.iscomplexobj(np.poly([1j, -1.0000001j])))
91: (8)              np.random.seed(42)
92: (8)              a = np.random.randn(100) + 1j*np.random.randn(100)
93: (8)              assert_(np.isrealobj(np.poly(np.concatenate((a, np.conjugate(a))))))
94: (4)          def test_roots(self):
95: (8)              assert_array_equal(np.roots([1, 0, 0]), [0, 0])
96: (4)          def test_str_leading_zeros(self):
97: (8)              p = np.poly1d([4, 3, 2, 1])
98: (8)              p[3] = 0
99: (8)              assert_equal(str(p),
100: (21)                  "2\n"
101: (21)                  "3 x + 2 x + 1")
102: (8)              p = np.poly1d([1, 2])
103: (8)              p[0] = 0
104: (8)              p[1] = 0
105: (8)              assert_equal(str(p), "\n0")
106: (4)          def test_polyfit(self):
107: (8)              c = np.array([3., 2., 1.])
108: (8)              x = np.linspace(0, 2, 7)
109: (8)              y = np.polyval(c, x)
110: (8)              err = [1, -1, 1, -1, 1, -1, 1]
111: (8)              weights = np.arange(8, 1, -1)**2/7.0
112: (8)              assert_raises(ValueError, np.polyfit,
113: (22)                  [1], [1], deg=0, cov=True)
114: (8)              m, cov = np.polyfit(x, y+err, 2, cov=True)
115: (8)              est = [3.8571, 0.2857, 1.619]
116: (8)              assert_almost_equal(est, m, decimal=4)
117: (8)              val0 = [[ 1.4694, -2.9388,  0.8163],
118: (16)                  [-2.9388,  6.3673, -2.1224],
119: (16)                  [ 0.8163, -2.1224,  1.161 ]]
120: (8)              assert_almost_equal(val0, cov, decimal=4)
121: (8)              m2, cov2 = np.polyfit(x, y+err, 2, w=weights, cov=True)
122: (8)              assert_almost_equal([4.8927, -1.0177, 1.7768], m2, decimal=4)

```

```

123: (8)           val = [[ 4.3964, -5.0052,  0.4878],
124: (15)          [-5.0052,  6.8067, -0.9089],
125: (15)          [ 0.4878, -0.9089,  0.3337]]
126: (8)          assert_almost_equal(val, cov2, decimal=4)
127: (8)          m3, cov3 = np.polyfit(x, y+err, 2, w=weights, cov="unscaled")
128: (8)          assert_almost_equal([4.8927, -1.0177, 1.7768], m3, decimal=4)
129: (8)          val = [[ 0.1473, -0.1677,  0.0163],
130: (15)          [-0.1677,  0.228 , -0.0304],
131: (15)          [ 0.0163, -0.0304,  0.0112]]
132: (8)          assert_almost_equal(val, cov3, decimal=4)
133: (8)          y = y[:, np.newaxis]
134: (8)          c = c[:, np.newaxis]
135: (8)          assert_almost_equal(c, np.polyfit(x, y, 2))
136: (8)          yy = np.concatenate((y, y), axis=1)
137: (8)          cc = np.concatenate((c, c), axis=1)
138: (8)          assert_almost_equal(cc, np.polyfit(x, yy, 2))
139: (8)          m, cov = np.polyfit(x, yy + np.array(err)[:, np.newaxis], 2, cov=True)
140: (8)          assert_almost_equal(est, m[:, 0], decimal=4)
141: (8)          assert_almost_equal(est, m[:, 1], decimal=4)
142: (8)          assert_almost_equal(val0, cov[:, :, 0], decimal=4)
143: (8)          assert_almost_equal(val0, cov[:, :, 1], decimal=4)
144: (8)          np.random.seed(123)
145: (8)          y = np.random.normal(size=(4, 10000))
146: (8)          mean, cov = np.polyfit(np.zeros(y.shape[0]), y, deg=0, cov=True)
147: (8)          assert_allclose(mean.std(), 0.5, atol=0.01)
148: (8)          assert_allclose(np.sqrt(cov.mean()), 0.5, atol=0.01)
149: (8)          mean, cov = np.polyfit(np.zeros(y.shape[0]), y, w=np.ones(y.shape[0]),
150: (31)              deg=0, cov="unscaled")
151: (8)          assert_allclose(mean.std(), 0.5, atol=0.01)
152: (8)          assert_almost_equal(np.sqrt(cov.mean()), 0.5)
153: (8)          w = np.full(y.shape[0], 1./0.5)
154: (8)          mean, cov = np.polyfit(np.zeros(y.shape[0]), y, w=w, deg=0, cov=True)
155: (8)          assert_allclose(mean.std(), 0.5, atol=0.01)
156: (8)          assert_allclose(np.sqrt(cov.mean()), 0.5, atol=0.01)
157: (8)          mean, cov = np.polyfit(np.zeros(y.shape[0]), y, w=w, deg=0,
cov="unscaled")
158: (8)          assert_allclose(mean.std(), 0.5, atol=0.01)
159: (8)          assert_almost_equal(np.sqrt(cov.mean()), 0.25)
160: (4)          def test_objects(self):
161: (8)              from decimal import Decimal
162: (8)              p = np.poly1d([Decimal('4.0'), Decimal('3.0'), Decimal('2.0')])
163: (8)              p2 = p * Decimal('1.333333333333333')
164: (8)              assert_(p2[1] == Decimal("3.999999999999999"))
165: (8)              p2 = p.deriv()
166: (8)              assert_(p2[1] == Decimal('8.0'))
167: (8)              p2 = p.integ()
168: (8)              assert_(p2[3] == Decimal("1.3333333333333333333333"))
169: (8)              assert_(p2[2] == Decimal('1.5'))
170: (8)              assert_(np.issubdtype(p2.coeffs.dtype, np.object_))
171: (8)              p = np.poly([Decimal(1), Decimal(2)])
172: (8)              assert_equal(np.poly([Decimal(1), Decimal(2)]),
173: (21)                  [1, Decimal(-3), Decimal(2)])
174: (4)          def test_complex(self):
175: (8)              p = np.poly1d([3j, 2j, 1j])
176: (8)              p2 = p.integ()
177: (8)              assert_((p2.coeffs == [1j, 1j, 1j, 0]).all())
178: (8)              p2 = p.deriv()
179: (8)              assert_((p2.coeffs == [6j, 2j]).all())
180: (4)          def test_integ_coeffs(self):
181: (8)              p = np.poly1d([3, 2, 1])
182: (8)              p2 = p.integ(3, k=[9, 7, 6])
183: (8)              assert_
184: (12)                  (p2.coeffs == [1/4./5., 1/3./4., 1/2./3., 9/1./2., 7, 6]).all()
185: (4)          def test_zero_dims(self):
186: (8)              try:
187: (12)                  np.poly(np.zeros((0, 0)))
188: (8)              except ValueError:
189: (12)                  pass
190: (4)          def test_poly_int_overflow(self):

```

```

191: (8)             """
192: (8)             Regression test for gh-5096.
193: (8)             """
194: (8)             v = np.arange(1, 21)
195: (8)             assert_almost_equal(np.poly(v), np.poly(np.diag(v)))
196: (4)             def test_zero_poly_dtype(self):
197: (8)                 """
198: (8)                 Regression test for gh-16354.
199: (8)                 """
200: (8)                 z = np.array([0, 0, 0])
201: (8)                 p = np.poly1d(z.astype(np.int64))
202: (8)                 assert_equal(p.coeffs.dtype, np.int64)
203: (8)                 p = np.poly1d(z.astype(np.float32))
204: (8)                 assert_equal(p.coeffs.dtype, np.float32)
205: (8)                 p = np.poly1d(z.astype(np.complex64))
206: (8)                 assert_equal(p.coeffs.dtype, np.complex64)
207: (4)             def test_poly_eq(self):
208: (8)                 p = np.poly1d([1, 2, 3])
209: (8)                 p2 = np.poly1d([1, 2, 4])
210: (8)                 assert_equal(p == None, False)
211: (8)                 assert_equal(p != None, True)
212: (8)                 assert_equal(p == p, True)
213: (8)                 assert_equal(p == p2, False)
214: (8)                 assert_equal(p != p2, True)
215: (4)             def test_polydiv(self):
216: (8)                 b = np.poly1d([2, 6, 6, 1])
217: (8)                 a = np.poly1d([-1j, (1+2j), -(2+1j), 1])
218: (8)                 q, r = np.polydiv(b, a)
219: (8)                 assert_equal(q.coeffs.dtype, np.complex128)
220: (8)                 assert_equal(r.coeffs.dtype, np.complex128)
221: (8)                 assert_equal(q*a + r, b)
222: (8)                 c = [1, 2, 3]
223: (8)                 d = np.poly1d([1, 2, 3])
224: (8)                 s, t = np.polydiv(c, d)
225: (8)                 assert isinstance(s, np.poly1d)
226: (8)                 assert isinstance(t, np.poly1d)
227: (8)                 u, v = np.polydiv(d, c)
228: (8)                 assert isinstance(u, np.poly1d)
229: (8)                 assert isinstance(v, np.poly1d)
230: (4)             def test_poly_coefficients(self):
231: (8)                 """ Coefficients should be modifiable """
232: (8)                 p = np.poly1d([1, 2, 3])
233: (8)                 p.coeffs += 1
234: (8)                 assert_equal(p.coeffs, [2, 3, 4])
235: (8)                 p.coeffs[2] += 10
236: (8)                 assert_equal(p.coeffs, [2, 3, 14])
237: (8)                 assert_raises(AttributeError, setattr, p, 'coeffs', np.array(1))

```

---

## File 243 - test\_recfunctions.py:

```

1: (0)             import pytest
2: (0)             import numpy as np
3: (0)             import numpy.ma as ma
4: (0)             from numpy.ma.mrecords import MaskedRecords
5: (0)             from numpy.ma.testutils import assert_equal
6: (0)             from numpy.testing import assert_, assert_raises
7: (0)             from numpy.lib.recfunctions import (
8: (4)                 drop_fields, rename_fields, get_fieldstructure, recursive_fill_fields,
9: (4)                 find_duplicates, merge_arrays, append_fields, stack_arrays, join_by,
10: (4)                 repack_fields, unstructured_to_structured, structured_to_unstructured,
11: (4)                 apply_along_fields, require_fields, assign_fields_by_name)
12: (0)             get_fieldspec = np.lib.recfunctions._get_fieldspec
13: (0)             get_names = np.lib.recfunctions.get_names
14: (0)             get_names_flat = np.lib.recfunctions.get_names_flat
15: (0)             zip_descr = np.lib.recfunctions._zip_descr
16: (0)             zip_dtype = np.lib.recfunctions._zip_dtype
17: (0)             class TestRecFunctions:

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

18: (4)
19: (8)
20: (8)
21: (8)
22: (21)
23: (8)
24: (21)
25: (8)
26: (4)
27: (8)
28: (8)
29: (8)
30: (21)
31: (8)
32: (8)
33: (21)
34: (8)
35: (8)
36: (21)
37: (8)
38: (8)
39: (21)
40: (31)
41: (8)
42: (8)
43: (21)
44: (31)
45: (31)
46: (8)
47: (8)
48: (21)
49: (31)
50: (37)
51: (4)
52: (8)
53: (21)
54: (8)
55: (8)
56: (27)
57: (8)
58: (8)
59: (8)
60: (8)
61: (8)
62: (8)
63: (27)
64: (8)
65: (8)
66: (8)
67: (8)
68: (8)
69: (8)
70: (8)
71: (4)
72: (8)
73: (21)
74: (28)
75: (8)
76: (8)
77: (8)
78: (8)
79: (8)
80: (4)
81: (8)
82: (8)
83: (8)
84: (8)
85: (8)
86: (8)

        def setup_method(self):
            x = np.array([1, 2, ])
            y = np.array([10, 20, 30])
            z = np.array([('A', 1.), ('B', 2.),
                         dtype=[('A', '|S3'), ('B', float)])
            w = np.array([(1, (2, 3.0)), (4, (5, 6.0))],
                         dtype=[('a', int), ('b', [('ba', float), ('bb', int)])])
            self.data = (w, x, y, z)
        def test_zip_descr(self):
            (w, x, y, z) = self.data
            test = zip_descr((x, x), flatten=True)
            assert_equal(test,
                         np.dtype([('int'), ('int')]))
            test = zip_descr((x, x), flatten=False)
            assert_equal(test,
                         np.dtype([('int'), ('int')]))
            test = zip_descr((x, z), flatten=True)
            assert_equal(test,
                         np.dtype([('int'), ('A', '|S3'), ('B', float)]))
            test = zip_descr((x, z), flatten=False)
            assert_equal(test,
                         np.dtype([('int'),
                                   ('[('A', '|S3'), ('B', float)])]))
            test = zip_descr((x, w), flatten=True)
            assert_equal(test,
                         np.dtype([('int'),
                                   ('a', int),
                                   ('ba', float), ('bb', int)]))
            test = zip_descr((x, w), flatten=False)
            assert_equal(test,
                         np.dtype([('int'),
                                   ('[('a', int),
                                   ('b', [('ba', float), ('bb', int)])]))))

        def test_drop_fields(self):
            a = np.array([(1, (2, 3.0)), (4, (5, 6.0))],
                         dtype=[('a', int), ('b', [('ba', float), ('bb', int)])])
            test = drop_fields(a, 'a')
            control = np.array([(2, 3.0), (5, 6.0)],
                         dtype=[('b', [('ba', float), ('bb', int)])])
            assert_equal(test, control)
            test = drop_fields(a, 'b')
            control = np.array([(1, ), (4, )], dtype=[('a', int)])
            assert_equal(test, control)
            test = drop_fields(a, ['ba', ])
            control = np.array([(1, (3.0, )), (4, (6.0, ))],
                         dtype=[('a', int), ('b', [('bb', int)])])
            assert_equal(test, control)
            test = drop_fields(a, ['ba', 'bb'])
            control = np.array([(1, ), (4, )], dtype=[('a', int)])
            assert_equal(test, control)
            test = drop_fields(a, ['a', 'b'])
            control = np.array([((), ), ()], dtype=[])
            assert_equal(test, control)

        def test_rename_fields(self):
            a = np.array([(1, (2, [3.0, 30.])), (4, (5, [6.0, 60.])),
                         dtype=[('a', int),
                               ('b', [('ba', float), ('bb', (float, 2))])])
            test = rename_fields(a, {'a': 'A', 'bb': 'BB'})
            newdtype = [(('A', int), ('b', [('ba', float), ('BB', (float, 2))]))]
            control = a.view(newdtype)
            assert_equal(test.dtype, newdtype)
            assert_equal(test, control)

        def test_get_names(self):
            ndtype = np.dtype([('A', '|S3'), ('B', float)])
            test = get_names(ndtype)
            assert_equal(test, ('A', 'B'))
            ndtype = np.dtype([('a', int), ('b', [('ba', float), ('bb', int)])])
            test = get_names(ndtype)
            assert_equal(test, ('a', ('b', ('ba', 'bb')))))

```

```

87: (8)          ndtype = np.dtype([('a', int), ('b', [])])
88: (8)          test = get_names(ndtype)
89: (8)          assert_equal(test, ('a', ('b', ())))
90: (8)          ndtype = np.dtype([])
91: (8)          test = get_names(ndtype)
92: (8)          assert_equal(test, ())
93: (4)          def test_get_names_flat(self):
94: (8)              ndtype = np.dtype([('A', '|S3'), ('B', float)])
95: (8)              test = get_names_flat(ndtype)
96: (8)              assert_equal(test, ('A', 'B'))
97: (8)              ndtype = np.dtype([('a', int), ('b', [('ba', float), ('bb', int)])])
98: (8)              test = get_names_flat(ndtype)
99: (8)              assert_equal(test, ('a', 'b', 'ba', 'bb'))
100: (8)             ndtype = np.dtype([('a', int), ('b', [])])
101: (8)             test = get_names_flat(ndtype)
102: (8)             assert_equal(test, ('a', 'b'))
103: (8)             ndtype = np.dtype([])
104: (8)             test = get_names_flat(ndtype)
105: (8)             assert_equal(test, ())
106: (4)          def test_get_fieldstructure(self):
107: (8)              ndtype = np.dtype([('A', '|S3'), ('B', float)])
108: (8)              test = get_fieldstructure(ndtype)
109: (8)              assert_equal(test, {'A': [], 'B': []})
110: (8)              ndtype = np.dtype([('A', int), ('B', [('BA', float), ('BB', '|S1')])])
111: (8)              test = get_fieldstructure(ndtype)
112: (8)              assert_equal(test, {'A': [], 'B': [], 'BA': ['B', ], 'BB': ['B']})
113: (8)              ndtype = np.dtype([('A', int),
114: (27)                  ('B', [('BA', int),
115: (34)                      ('BB', [('BBA', int), ('BBB', int)])])),
116: (8)                      test = get_fieldstructure(ndtype)
117: (8)                      control = {'A': [], 'B': [], 'BA': ['B'], 'BB': ['B'],
118: (19)                          'BBA': ['B', 'BB'], 'BBB': ['B', 'BB']}
119: (8)                      assert_equal(test, control)
120: (8)                      ndtype = np.dtype([])
121: (8)                      test = get_fieldstructure(ndtype)
122: (8)                      assert_equal(test, {})
123: (4)          def test_find_duplicates(self):
124: (8)              a = ma.array([(2, (2., 'B')), (1, (2., 'B')), (2, (2., 'B')),
125: (22)                  (1, (1., 'B')), (2, (2., 'B')), (2, (2., 'C'))],
126: (21)                  mask=[(0, (0, 0)), (0, (0, 0)), (0, (0, 0)),
127: (27)                      (0, (0, 0)), (1, (0, 0)), (0, (1, 0))],
128: (21)                      dtype=[('A', int), ('B', [('BA', float), ('BB',
129: '|S1')])])
130: (8)              test = find_duplicates(a, ignoremask=False, return_index=True)
131: (8)              control = [0, 2]
132: (8)              assert_equal(sorted(test[-1]), control)
133: (8)              assert_equal(test[0], a[test[-1]])
134: (8)              test = find_duplicates(a, key='A', return_index=True)
135: (8)              control = [0, 1, 2, 3, 5]
136: (8)              assert_equal(sorted(test[-1]), control)
137: (8)              assert_equal(test[0], a[test[-1]])
138: (8)              test = find_duplicates(a, key='B', return_index=True)
139: (8)              control = [0, 1, 2, 4]
140: (8)              assert_equal(sorted(test[-1]), control)
141: (8)              assert_equal(test[0], a[test[-1]])
142: (8)              test = find_duplicates(a, key='BA', return_index=True)
143: (8)              control = [0, 1, 2, 4]
144: (8)              assert_equal(sorted(test[-1]), control)
145: (8)              assert_equal(test[0], a[test[-1]])
146: (8)              test = find_duplicates(a, key='BB', return_index=True)
147: (8)              control = [0, 1, 2, 3, 4]
148: (8)              assert_equal(sorted(test[-1]), control)
149: (4)          def test_find_duplicates_ignoremask(self):
150: (8)              ndtype = [('a', int)]
151: (8)              a = ma.array([1, 1, 1, 2, 2, 3, 3],
152: (21)                  mask=[0, 0, 1, 0, 0, 0, 1]).view(ndtype)
153: (8)              test = find_duplicates(a, ignoremask=True, return_index=True)
154: (8)              control = [0, 1, 3, 4]

```

```

155: (8) assert_equal(sorted(test[-1]), control)
156: (8) assert_equal(test[0], a[test[-1]])
157: (8) test = find_duplicates(a, ignoremask=False, return_index=True)
158: (8) control = [0, 1, 2, 3, 4, 6]
159: (8) assert_equal(sorted(test[-1]), control)
160: (8) assert_equal(test[0], a[test[-1]])
161: (4) def test_repack_fields(self):
162: (8) dt = np.dtype('u1,f4,i8', align=True)
163: (8) a = np.zeros(2, dtype=dt)
164: (8) assert_equal(repack_fields(dt), np.dtype('u1,f4,i8'))
165: (8) assert_equal(repack_fields(a).itemsize, 13)
166: (8) assert_equal(repack_fields(repack_fields(dt), align=True), dt)
167: (8) dt = np.dtype((np.record, dt))
168: (8) assert_(repack_fields(dt).type is np.record)
169: (4) def test_structured_to_unstructured(self, tmp_path):
170: (8) a = np.zeros(4, dtype=[('a', 'i4'), ('b', 'f4,u2'), ('c', 'f4', 2)])
171: (8) out = structured_to_unstructured(a)
172: (8) assert_equal(out, np.zeros((4,5), dtype='f8'))
173: (8) b = np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
174: (21)           dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
175: (8) out = np.mean(structured_to_unstructured(b[['x', 'z']]), axis=-1)
176: (8) assert_equal(out, np.array([ 3., 5.5, 9., 11. ]))
177: (8) out = np.mean(structured_to_unstructured(b[['x']]), axis=-1)
178: (8) assert_equal(out, np.array([ 1., 4., 7., 10. ]))
179: (8) c = np.arange(20).reshape((4,5))
180: (8) out = unstructured_to_structured(c, a.dtype)
181: (8) want = np.array([( 0, ( 1., 2), [ 3., 4.]),
182: (25)          ( 5, ( 6., 7), [ 8., 9.]),
183: (25)          (10, (11., 12), [13., 14.]),
184: (25)          (15, (16., 17), [18., 19.])],
185: (21)         dtype=[('a', 'i4'),
186: (28)          ('b', [('f0', 'f4'), ('f1', 'u2')]),
187: (28)          ('c', 'f4', (2,)))])
188: (8) assert_equal(out, want)
189: (8) d = np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
190: (21)         dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
191: (8) assert_equal(apply_along_fields(np.mean, d),
192: (21)           np.array([ 8.0/3, 16.0/3, 26.0/3, 11. ]))
193: (8) assert_equal(apply_along_fields(np.mean, d[['x', 'z']]),
194: (21)           np.array([ 3., 5.5, 9., 11. ]))
195: (8) d = np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
196: (21)         dtype=[('x', 'i4'), ('y', 'i4'), ('z', 'i4')])
197: (8) dd = structured_to_unstructured(d)
198: (8) ddd = unstructured_to_structured(dd, d.dtype)
199: (8) assert_(np.shares_memory(dd, d))
200: (8) assert_(np.shares_memory(ddd, d))
201: (8) dd_attrib_rev = structured_to_unstructured(d[['z', 'x']])
202: (8) assert_equal(dd_attrib_rev, [[5, 1], [7, 4], [11, 7], [12, 10]])
203: (8) assert_(np.shares_memory(dd_attrib_rev, d))
204: (8) d = np.array([(1, [2, 3], [[4, 5], [6, 7]]),
205: (22)          (8, [9, 10], [[11, 12], [13, 14]]),
206: (21)          dtype=[('x0', 'i4'), ('x1', ('i4', 2)),
207: (28)          ('x2', ('i4', (2, 2)))])
208: (8) dd = structured_to_unstructured(d)
209: (8) ddd = unstructured_to_structured(dd, d.dtype)
210: (8) assert_(np.shares_memory(dd, d))
211: (8) assert_(np.shares_memory(ddd, d))
212: (8) d_rev = d[::-1]
213: (8) dd_rev = structured_to_unstructured(d_rev)
214: (8) assert_equal(dd_rev, [[8, 9, 10, 11, 12, 13, 14],
215: (30)          [1, 2, 3, 4, 5, 6, 7]])
216: (8) d_attrib_rev = d[['x2', 'x1', 'x0']]
217: (8) dd_attrib_rev = structured_to_unstructured(d_attrib_rev)
218: (8) assert_equal(dd_attrib_rev, [[4, 5, 6, 7, 2, 3, 1],
219: (37)          [11, 12, 13, 14, 9, 10, 8]])
220: (8) d = np.array([(1, [2, 3], [[4, 5], [6, 7]], 32),
221: (22)          (8, [9, 10], [[11, 12], [13, 14]], 64)],
222: (21)          dtype=[('x0', 'i4'), ('x1', ('i4', 2)),
223: (28)          ('x2', ('i4', (2, 2))), ('ignored', 'u1')]))

```

```

224: (8)           dd = structured_to_unstructured(d[['x0', 'x1', 'x2']])
225: (8)           assert_(np.shares_memory(dd, d))
226: (8)           assert_equal(dd, [[1, 2, 3, 4, 5, 6, 7],
227: (26)                 [8, 9, 10, 11, 12, 13, 14]])
228: (8)           point = np.dtype([('x', int), ('y', int)])
229: (8)           triangle = np.dtype([('a', point), ('b', point), ('c', point)])
230: (8)           arr = np.zeros(10, triangle)
231: (8)           res = structured_to_unstructured(arr, dtype=int)
232: (8)           assert_equal(res, np.zeros((10, 6), dtype=int))
233: (8)           def subarray(dt, shape):
234: (12)             return np.dtype((dt, shape))
235: (8)           def structured(*dts):
236: (12)             return np.dtype([('x{}'.format(i), dt) for i, dt in
237: (8)               enumerate(dts)])
238: (12)
239: (12)
240: (12)
241: (12)
242: (8)           dt = structured(subarray(structured(np.int32, np.int32), 3))
243: (8)           assert_equal(inspect(dt), ((6,), np.int32, dt))
244: (8)           dt = structured(subarray(subarray(np.int32, 2), 2))
245: (8)           assert_equal(inspect(dt), ((4,), np.int32, dt))
246: (8)           dt = structured(np.int32)
247: (8)           assert_equal(inspect(dt), ((1,), np.int32, dt))
248: (8)           dt = structured(np.int32, subarray(subarray(np.int32, 2), 2))
249: (8)           assert_equal(inspect(dt), ((5,), np.int32, dt))
250: (8)           dt = structured()
251: (8)           assert_raises(ValueError, structured_to_unstructured, np.zeros(3, dt))
252: (8)           assert_raises(NotImplementedError, structured_to_unstructured,
253: (43)                         np.zeros(3, dt), dtype=np.int32)
254: (8)           assert_raises(NotImplementedError, unstructured_to_structured,
255: (43)                         np.zeros((3,0), dtype=np.int32))
256: (8)           d_plain = np.array([(1, 2), (3, 4)], dtype=[('a', 'i4'), ('b', 'i4')])
257: (8)           dd_expected = structured_to_unstructured(d_plain, copy=True)
258: (8)           d = d_plain.view(np.recarray)
259: (8)           dd = structured_to_unstructured(d, copy=False)
260: (8)           ddd = structured_to_unstructured(d, copy=True)
261: (8)           assert_(np.shares_memory(d, dd))
262: (8)           assert_(type(dd) is np.recarray)
263: (8)           assert_(type(ddd) is np.recarray)
264: (8)           assert_equal(dd, dd_expected)
265: (8)           assert_equal(ddd, dd_expected)
266: (8)           d = np.memmap(tmp_path / 'memmap',
267: (22)                         mode='w+',
268: (22)                         dtype=d_plain.dtype,
269: (22)                         shape=d_plain.shape)
270: (8)           d[:] = d_plain
271: (8)           dd = structured_to_unstructured(d, copy=False)
272: (8)           ddd = structured_to_unstructured(d, copy=True)
273: (8)           assert_(np.shares_memory(d, dd))
274: (8)           assert_(type(dd) is np.memmap)
275: (8)           assert_(type(ddd) is np.memmap)
276: (8)           assert_equal(dd, dd_expected)
277: (8)           assert_equal(ddd, dd_expected)
278: (4)           def test_unstructured_to_structured(self):
279: (8)             a = np.zeros((20, 2))
280: (8)             test_dtype_args = [('x', float), ('y', float)]
281: (8)             test_dtype = np.dtype(test_dtype_args)
282: (8)             field1 = unstructured_to_structured(a, dtype=test_dtype_args) # now
283: (8)             field2 = unstructured_to_structured(a, dtype=test_dtype) # before
284: (8)             assert_equal(field1, field2)
285: (4)           def test_field_assignment_by_name(self):
286: (8)             a = np.ones(2, dtype=[('a', 'i4'), ('b', 'f8'), ('c', 'u1')])
287: (8)             newdt = [('b', 'f4'), ('c', 'u1')]
288: (8)             assert_equal(require_fields(a, newdt), np.ones(2, newdt))
289: (8)             b = np.array([(1, 2), (3, 4)], dtype=newdt)
290: (8)             assign_fields_by_name(a, b, zero_unassigned=False)
291: (8)             assert_equal(a, np.array([(1, 1, 2), (1, 3, 4)], dtype=a.dtype))

```

```

292: (8)           assign_fields_by_name(a, b)
293: (8)           assert_equal(a, np.array([(0,1,2),(0,3,4)], dtype=a.dtype))
294: (8)           a = np.ones(2, dtype=[('a', [('b', 'f8'), ('c', 'u1')])])
295: (8)           newdt = [('a', [('c', 'u1'))])
296: (8)           assert_equal(require_fields(a, newdt), np.ones(2, newdt))
297: (8)           b = np.array([(2,), (3,)], dtype=newdt)
298: (8)           assign_fields_by_name(a, b, zero_unassigned=False)
299: (8)           assert_equal(a, np.array([(1,2), (1,3)], dtype=a.dtype))
300: (8)           assign_fields_by_name(a, b)
301: (8)           assert_equal(a, np.array([(0,2), (0,3)], dtype=a.dtype))
302: (8)           a, b = np.array(3), np.array(0)
303: (8)           assign_fields_by_name(b, a)
304: (8)           assert_equal(b[()], 3)
305: (0)           class TestRecursiveFillFields:
306: (4)             def test_simple_flexible(self):
307: (8)               a = np.array([(1, 10.), (2, 20.)], dtype=[('A', int), ('B', float)])
308: (8)               b = np.zeros((3,), dtype=a.dtype)
309: (8)               test = recursive_fill_fields(a, b)
310: (8)               control = np.array([(1, 10.), (2, 20.), (0, 0.)],
311: (27)                           dtype=[('A', int), ('B', float)])
312: (8)               assert_equal(test, control)
313: (4)             def test_masked_flexible(self):
314: (8)               a = ma.array([(1, 10.), (2, 20.)], mask=[(0, 1), (1, 0)],
315: (21)                           dtype=[('A', int), ('B', float)])
316: (8)               b = ma.zeros((3,), dtype=a.dtype)
317: (8)               test = recursive_fill_fields(a, b)
318: (8)               control = ma.array([(1, 10.), (2, 20.), (0, 0.)],
319: (27)                           mask=[(0, 1), (1, 0), (0, 0)],
320: (27)                           dtype=[('A', int), ('B', float)])
321: (8)               assert_equal(test, control)
322: (0)           class TestMergeArrays:
323: (4)             def setup_method(self):
324: (8)               x = np.array([1, 2, ])
325: (8)               y = np.array([10, 20, 30])
326: (8)               z = np.array(
327: (12)                 [('A', 1.), ('B', 2.)], dtype=[('A', '|S3'), ('B', float)])
328: (8)               w = np.array(
329: (12)                 [(1, (2, 3.0, ())), (4, (5, 6.0, ()))],
330: (12)                               dtype=[('a', int), ('b', [('ba', float), ('bb', int), ('bc', [])])])
331: (8)               self.data = (w, x, y, z)
332: (4)             def test_solo(self):
333: (8)               (_, x, _, z) = self.data
334: (8)               test = merge_arrays(x)
335: (8)               control = np.array([(1,), (2,)], dtype=[('f0', int)])
336: (8)               assert_equal(test, control)
337: (8)               test = merge_arrays((x,))
338: (8)               assert_equal(test, control)
339: (8)               test = merge_arrays(z, flatten=False)
340: (8)               assert_equal(test, z)
341: (8)               test = merge_arrays(z, flatten=True)
342: (8)               assert_equal(test, z)
343: (4)             def test_solo_w_flatten(self):
344: (8)               w = self.data[0]
345: (8)               test = merge_arrays(w, flatten=False)
346: (8)               assert_equal(test, w)
347: (8)               test = merge_arrays(w, flatten=True)
348: (8)               control = np.array([(1, 2, 3.0), (4, 5, 6.0)],
349: (27)                               dtype=[('a', int), ('ba', float), ('bb', int)])
350: (8)               assert_equal(test, control)
351: (4)             def test_standard(self):
352: (8)               (_, x, y, _) = self.data
353: (8)               test = merge_arrays((x, y), usemask=False)
354: (8)               control = np.array([(1, 10), (2, 20), (-1, 30)],
355: (27)                               dtype=[('f0', int), ('f1', int)])
356: (8)               assert_equal(test, control)
357: (8)               test = merge_arrays((x, y), usemask=True)
358: (8)               control = ma.array([(1, 10), (2, 20), (-1, 30)],
359: (27)                               mask=[(0, 0), (0, 0), (1, 0)]),

```

```

360: (27)                               dtype=[('f0', int), ('f1', int)])
361: (8)       assert_equal(test, control)
362: (8)       assert_equal(test.mask, control.mask)
363: (4) def test_flatten(self):
364: (8)   (_, x, _, z) = self.data
365: (8)   test = merge_arrays((x, z), flatten=True)
366: (8)   control = np.array([(1, 'A', 1.), (2, 'B', 2.)],
367: (27)           dtype=[('f0', int), ('A', '|S3'), ('B', float)])
368: (8)   assert_equal(test, control)
369: (8)   test = merge_arrays((x, z), flatten=False)
370: (8)   control = np.array([(1, ('A', 1.)), (2, ('B', 2.))]),
371: (27)           dtype=[('f0', int),
372: (34)             ('f1', [('A', '|S3'), ('B', float)]))]
373: (8)   assert_equal(test, control)
374: (4) def test_flatten_wflexible(self):
375: (8)   (w, x, _, _) = self.data
376: (8)   test = merge_arrays((x, w), flatten=True)
377: (8)   control = np.array([(1, 1, 2, 3.0), (2, 4, 5, 6.0)],
378: (27)           dtype=[('f0', int),
379: (34)             ('a', int), ('ba', float), ('bb', int)])
380: (8)   assert_equal(test, control)
381: (8)   test = merge_arrays((x, w), flatten=False)
382: (8)   controldtype = [(('f0', int),
383: (32)               ('f1', [('a', int),
384: (40)                 ('b', [('ba', float), ('bb', int)),
385: ('bc', [])])))]
385: (8)   control = np.array([(1., (1, (2, 3.0, ()))), (2, (4, (5, 6.0, ()))),
386: (27)           dtype=controldtype)
387: (8)   assert_equal(test, control)
388: (4) def test_wmasked_arrays(self):
389: (8)   (_, x, _, _) = self.data
390: (8)   mx = ma.array([1, 2, 3], mask=[1, 0, 0])
391: (8)   test = merge_arrays((x, mx), usemask=True)
392: (8)   control = ma.array([(1, 1), (2, 2), (-1, 3)],
393: (27)           mask=[(0, 1), (0, 0), (1, 0)],
394: (27)           dtype=[('f0', int), ('f1', int)])
395: (8)   assert_equal(test, control)
396: (8)   test = merge_arrays((x, mx), usemask=True, asrecarray=True)
397: (8)   assert_equal(test, control)
398: (8)   assert_(isinstance(test, MaskedRecords))
399: (4) def test_w_singlefield(self):
400: (8)   test = merge_arrays((np.array([1, 2]).view([('a', int))),
401: (29)             np.array([10., 20., 30.])))
402: (8)   control = ma.array([(1, 10.), (2, 20.), (-1, 30.)],
403: (27)           mask=[(0, 0), (0, 0), (1, 0)],
404: (27)           dtype=[('a', int), ('f1', float)])
405: (8)   assert_equal(test, control)
406: (4) def test_w_shorter_flex(self):
407: (8)   z = self.data[-1]
408: (8)   merge_arrays((z, np.array([10, 20, 30]).view([('C', int)])))
409: (8)   np.array([(('A', 1., 10), ('B', 2., 20), ('-1', -1, 20)],
410: (17)           dtype=[('A', '|S3'), ('B', float), ('C', int)])
411: (4) def test_singlerecord(self):
412: (8)   (_, x, y, z) = self.data
413: (8)   test = merge_arrays((x[0], y[0], z[0]), usemask=False)
414: (8)   control = np.array([(1, 10, ('A', 1))],
415: (27)           dtype=[('f0', int),
416: (34)             ('f1', int),
417: (34)               ('f2', [('A', '|S3'), ('B', float)]))]
418: (8)   assert_equal(test, control)
419: (0) class TestAppendFields:
420: (4)   def setup_method(self):
421: (8)     x = np.array([1, 2, ])
422: (8)     y = np.array([10, 20, 30])
423: (8)     z = np.array(
424: (12)       [('A', 1.), ('B', 2.)], dtype=[('A', '|S3'), ('B', float)])
425: (8)     w = np.array([(1, (2, 3.0)), (4, (5, 6.0))],
426: (21)           dtype=[('a', int), ('b', [('ba', float), ('bb', int))))]
427: (8)     self.data = (w, x, y, z)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

428: (4)
429: (8)
430: (8)
431: (8)
432: (27)
433: (27)
434: (8)
435: (4)
436: (8)
437: (8)
438: (8)
439: (27)
440: (27)
441: (8)
442: (4)
443: (8)
444: (8)
445: (8)
446: (27)
447: (27)
448: (8)
449: (4)
450: (8)
451: (8)
452: (8)
453: (28)
454: (28)
455: (27)
456: (31)
457: (27)
458: (34)
459: (34)
460: (8)
461: (0)
462: (4)
463: (8)
464: (8)
465: (8)
466: (12)
467: (8)
468: (21)
469: (8)
470: (4)
471: (8)
472: (8)
473: (8)
474: (8)
475: (8)
476: (8)
477: (8)
478: (4)
479: (8)
480: (8)
481: (8)
482: (8)
483: (8)
484: (8)
485: (8)
486: (8)
487: (8)
488: (8)
489: (4)
490: (8)
491: (8)
492: (8)
493: (28)
494: (27)
495: (33)
496: (27)

        def test_append_single(self):
            (_, x, _, _) = self.data
            test = append_fields(x, 'A', data=[10, 20, 30])
            control = ma.array([(1, 10), (2, 20), (-1, 30)],
                               mask=[(0, 0), (0, 0), (1, 0)],
                               dtype=[('f0', int), ('A', int)],)
            assert_equal(test, control)
        def test_append_double(self):
            (_, x, _, _) = self.data
            test = append_fields(x, ('A', 'B'), data=[[10, 20, 30], [100, 200]])
            control = ma.array([(1, 10, 100), (2, 20, 200), (-1, 30, -1)],
                               mask=[(0, 0, 0), (0, 0, 0), (1, 0, 1)],
                               dtype=[('f0', int), ('A', int), ('B', int)],)
            assert_equal(test, control)
        def test_append_on_flex(self):
            z = self.data[-1]
            test = append_fields(z, 'C', data=[10, 20, 30])
            control = ma.array([('A', 1., 10), ('B', 2., 20), (-1, -1., 30)],
                               mask=[(0, 0, 0), (0, 0, 0), (1, 1, 0)],
                               dtype=[('A', '|S3'), ('B', float), ('C', int)],)
            assert_equal(test, control)
        def test_append_on_nested(self):
            w = self.data[0]
            test = append_fields(w, 'C', data=[10, 20, 30])
            control = ma.array([(1, (2, 3.0), 10),
                                (4, (5, 6.0), 20),
                                (-1, (-1, -1.), 30)],
                               mask=[(0, (0, 0), 0), (0, (0, 0), 0), (1, (1, 1), 0)],
                               dtype=[('a', int),
                                      ('b', [('ba', float), ('bb', int)]),
                                      ('C', int)],)
            assert_equal(test, control)
    class TestStackArrays:
        def setup_method(self):
            x = np.array([1, 2, ])
            y = np.array([10, 20, 30])
            z = np.array([
                ('A', 1.), ('B', 2.)], dtype=[('A', '|S3'), ('B', float)])
            w = np.array([(1, (2, 3.0)), (4, (5, 6.0))],
                         dtype=[('a', int), ('b', [('ba', float), ('bb', int)])])
            self.data = (w, x, y, z)
        def test_solo(self):
            (_, x, _, _) = self.data
            test = stack_arrays((x,))
            assert_equal(test, x)
            assert_(test is x)
            test = stack_arrays(x)
            assert_equal(test, x)
            assert_(test is x)
        def test_unnamed_fields(self):
            (_, x, y, _) = self.data
            test = stack_arrays((x, x), usemask=False)
            control = np.array([1, 2, 1, 2])
            assert_equal(test, control)
            test = stack_arrays((x, y), usemask=False)
            control = np.array([1, 2, 10, 20, 30])
            assert_equal(test, control)
            test = stack_arrays((y, x), usemask=False)
            control = np.array([10, 20, 30, 1, 2])
            assert_equal(test, control)
        def test_unnamed_and_named_fields(self):
            (_, x, _, z) = self.data
            test = stack_arrays((x, z))
            control = ma.array([(1, -1, -1), (2, -1, -1),
                               (-1, 'A', 1), (-1, 'B', 2)],
                               mask=[(0, 1, 1), (0, 1, 1),
                                      (1, 0, 0), (1, 0, 0)],
                               dtype=[('f0', int), ('A', '|S3'), ('B', float)])

```

```

497: (8)             assert_equal(test, control)
498: (8)             assert_equal(test.mask, control.mask)
499: (8)             test = stack_arrays((z, x))
500: (8)             control = ma.array([('A', 1, -1), ('B', 2, -1),
501: (28)                 (-1, -1, 1), (-1, -1, 2)], [
502: (27)                     mask=[(0, 0, 1), (0, 0, 1),
503: (33)                         (1, 1, 0), (1, 1, 0)],
504: (27)                     dtype=[('A', '|S3'), ('B', float), ('f2', int)])
505: (8)             assert_equal(test, control)
506: (8)             assert_equal(test.mask, control.mask)
507: (8)             test = stack_arrays((z, z, x))
508: (8)             control = ma.array([('A', 1, -1), ('B', 2, -1),
509: (28)                 ('A', 1, -1), ('B', 2, -1),
510: (28)                     (-1, -1, 1), (-1, -1, 2)], [
511: (27)                     mask=[(0, 0, 1), (0, 0, 1),
512: (33)                         (0, 0, 1), (0, 0, 1),
513: (33)                         (1, 1, 0), (1, 1, 0)],
514: (27)                     dtype=[('A', '|S3'), ('B', float), ('f2', int)])
515: (8)             assert_equal(test, control)
516: (4)             def test_matching_named_fields(self):
517: (8)                 (_, x, _, z) = self.data
518: (8)                 zz = np.array([('a', 10., 100.), ('b', 20., 200.), ('c', 30., 300.)],
519: (22)                     dtype=[('A', '|S3'), ('B', float), ('C', float)])
520: (8)                 test = stack_arrays((z, zz))
521: (8)                 control = ma.array([('A', 1, -1), ('B', 2, -1),
522: (28)                     (
523: (32)                         'a', 10., 100.), ('b', 20., 200.), ('c', 30.,
524: (300.)],
525: (27)
526: (33)
527: (8)                 assert_equal(test, control)
528: (8)                 assert_equal(test.mask, control.mask)
529: (8)                 test = stack_arrays((z, zz, x))
530: (8)                 ndtype = [('A', '|S3'), ('B', float), ('C', float), ('f3', int)]
531: (8)                 control = ma.array([('A', 1, -1, -1), ('B', 2, -1, -1),
532: (28)                     ('a', 10., 100., -1), ('b', 20., 200., -1),
533: (28)                     ('c', 30., 300., -1),
534: (28)                     (-1, -1, -1, 1), (-1, -1, -1, 2)], [
535: (27)                     dtype=ndtype,
536: (27)                     mask=[(0, 0, 1, 1), (0, 0, 1, 1),
537: (33)                         (0, 0, 0, 1), (0, 0, 0, 1), (0, 0, 0, 1),
538: (33)                         (1, 1, 1, 0), (1, 1, 1, 0)])
539: (8)                 assert_equal(test, control)
540: (8)                 assert_equal(test.mask, control.mask)
541: (4)             def test_defaults(self):
542: (8)                 (_, _, _, z) = self.data
543: (8)                 zz = np.array([('a', 10., 100.), ('b', 20., 200.), ('c', 30., 300.)],
544: (22)                     dtype=[('A', '|S3'), ('B', float), ('C', float)])
545: (8)                 defaults = {'A': '????', 'B': -999., 'C': -9999., 'D': -99999.}
546: (8)                 test = stack_arrays((z, zz), defaults=defaults)
547: (8)                 control = ma.array([('A', 1, -9999.), ('B', 2, -9999.),
548: (28)                     (
549: (32)                         'a', 10., 100.), ('b', 20., 200.), ('c', 30.,
550: (300.)],
551: (27)
552: (33)
553: (8)                 assert_equal(test, control)
554: (8)                 assert_equal(test.data, control.data)
555: (8)                 assert_equal(test.mask, control.mask)
556: (4)             def test_autoconversion(self):
557: (8)                 adtype = [('A', int), ('B', bool), ('C', float)]
558: (8)                 a = ma.array([(1, 2, 3)], mask=[(0, 1, 0)], dtype=adtype)
559: (8)                 bdtype = [('A', int), ('B', float), ('C', float)]
560: (8)                 b = ma.array([(4, 5, 6)], dtype=bdtype)
561: (8)                 control = ma.array([(1, 2, 3), (4, 5, 6)], mask=[(0, 1, 0), (0, 0,
0)],
562: (27)                               dtype=bdtype)

```

```

563: (8)             test = stack_arrays((a, b), autoconvert=True)
564: (8)             assert_equal(test, control)
565: (8)             assert_equal(test.mask, control.mask)
566: (8)             with assert_raises(TypeError):
567: (12)                 stack_arrays((a, b), autoconvert=False)
568: (4)             def test_checktitles(self):
569: (8)                 adtype = [((‘a’, ‘A’), int), ((‘b’, ‘B’), bool), ((‘c’, ‘C’), float)]
570: (8)                 a = ma.array([(1, 2, 3)], mask=[(0, 1, 0)], dtype=adtype)
571: (8)                 bdtype = [((‘a’, ‘A’), int), ((‘b’, ‘B’), bool), ((‘c’, ‘C’), float)]
572: (8)                 b = ma.array([(4, 5, 6)], dtype=bdtype)
573: (8)                 test = stack_arrays((a, b))
574: (8)                 control = ma.array([(1, 2, 3), (4, 5, 6)], mask=[(0, 1, 0), (0, 0,
0)], dtype=bdtype)
575: (27)             assert_equal(test, control)
576: (8)             assert_equal(test.mask, control.mask)
577: (8)             def test_subdtype(self):
578: (4)                 z = np.array([
579: (8)                     (‘A’, 1), (‘B’, 2)
580: (12)                 ], dtype=[(‘A’, ‘|S3’), (‘B’, float, (1,))])
581: (8)                 zz = np.array([
582: (12)                     (‘a’, [10.], 100.), (‘b’, [20.], 200.), (‘c’, [30.], 300.)
583: (8)                 ], dtype=[(‘A’, ‘|S3’), (‘B’, float, (1,)), (‘C’, float)])
584: (8)                 res = stack_arrays((z, zz))
585: (8)                 expected = ma.array(
586: (8)                     data=[
587: (12)                         (b‘A’, [1.0], 0),
588: (16)                         (b‘B’, [2.0], 0),
589: (16)                         (b‘a’, [10.0], 100.0),
590: (16)                         (b‘b’, [20.0], 200.0),
591: (16)                         (b‘c’, [30.0], 300.0)],
592: (16)                     mask=[
593: (12)                         (False, [False], True),
594: (16)                         (False, [False], True),
595: (16)                         (False, [False], False),
596: (16)                         (False, [False], False),
597: (16)                         (False, [False], False)
598: (16)                     ],
599: (12)                     dtype=zz.dtype
600: (12)                 )
601: (8)                 assert_equal(res.dtype, expected.dtype)
602: (8)                 assert_equal(res, expected)
603: (8)                 assert_equal(res.mask, expected.mask)
604: (8)             class TestJoinBy:
605: (0)                 def setup_method(self):
606: (4)                     self.a = np.array(list(zip(np.arange(10), np.arange(50, 60),
607: (8)                         np.arange(100, 110))), dtype=[(‘a’, int), (‘b’, int), (‘c’, int)])
608: (35)                     self.b = np.array(list(zip(np.arange(5, 15), np.arange(65, 75),
609: (26)                         np.arange(100, 110))), dtype=[(‘a’, int), (‘b’, int), (‘d’, int)])
610: (8)                 def test_inner_join(self):
611: (35)                     a, b = self.a, self.b
612: (26)                     test = join_by(‘a’, a, b, jointype=‘inner’)
613: (4)                     control = np.array([(5, 55, 65, 105, 100), (6, 56, 66, 106, 101),
614: (8)                         (7, 57, 67, 107, 102), (8, 58, 68, 108, 103),
615: (8)                         (9, 59, 69, 109, 104)], dtype=[(‘a’, int), (‘b1’, int), (‘b2’, int),
616: (8)                         (‘c’, int), (‘d’, int)])
617: (28)                     assert_equal(test, control)
618: (28)                 def test_join(self):
619: (27)                     a, b = self.a, self.b
620: (34)                     join_by((‘a’, ‘b’), a, b)
621: (8)                     np.array([(5, 55, 105, 100), (6, 56, 106, 101),
622: (4)                         (7, 57, 107, 102), (8, 58, 108, 103),
623: (8)                         (9, 59, 109, 104)], dtype=[(‘a’, int), (‘b’, int),
624: (8)                         (‘c’, int), (‘d’, int)])
625: (8)                 def test_join_subdtype(self):
626: (18)             
```

```

631: (8)
632: (23)
633: (8)
634: (23)
635: (8)
636: (8)
637: (4)
638: (8)
639: (8)
640: (8)
641: (28)
642: (28)
643: (28)
644: (28)
645: (28)
646: (28)
647: (28)
648: (28)
649: (28)
650: (27)
651: (33)
652: (33)
653: (33)
654: (33)
655: (33)
656: (33)
657: (33)
658: (33)
659: (33)
660: (27)
661: (34)
662: (8)
663: (4)
664: (8)
665: (8)
666: (8)
667: (28)
668: (28)
669: (28)
670: (28)
671: (27)
672: (33)
673: (33)
674: (33)
675: (33)
676: (27)
int)])
677: (8)
678: (4)
679: (8)
680: (8)
681: (8)
682: (8)
683: (4)
684: (8)
685: (8)
686: (8)
687: (4)
688: (8)
689: (8)
690: (8)
691: (12)
692: (8)
693: (8)
694: (8)
695: (8)
696: (4)
697: (8)
698: (8)

        foo = np.array([(1,)], dtype=[('key', int)])
        bar = np.array([(1, np.array([1, 2, 3]))], dtype=[('key', int), ('value', 'uint16', 3)])
        res = join_by('key', foo, bar)
        assert_equal(res, bar.view(ma.MaskedArray))

    def test_outer_join(self):
        a, b = self.a, self.b
        test = join_by(('a', 'b'), a, b, 'outer')
        control = ma.array([(0, 50, 100, -1), (1, 51, 101, -1),
                            (2, 52, 102, -1), (3, 53, 103, -1),
                            (4, 54, 104, -1), (5, 55, 105, -1),
                            (5, 65, -1, 100), (6, 56, 106, -1),
                            (6, 66, -1, 101), (7, 57, 107, -1),
                            (7, 67, -1, 102), (8, 58, 108, -1),
                            (8, 68, -1, 103), (9, 59, 109, -1),
                            (9, 69, -1, 104), (10, 70, -1, 105),
                            (11, 71, -1, 106), (12, 72, -1, 107),
                            (13, 73, -1, 108), (14, 74, -1, 109)], mask=[[0, 0, 0, 1], (0, 0, 0, 1),
                            (0, 0, 0, 1), (0, 0, 0, 1),
                            (0, 0, 1, 0), (0, 0, 0, 1),
                            (0, 0, 1, 0), (0, 0, 0, 1),
                            (0, 0, 1, 0), (0, 0, 0, 1),
                            (0, 0, 1, 0), (0, 0, 0, 1),
                            (0, 0, 1, 0), (0, 0, 1, 0),
                            (0, 0, 1, 0), (0, 0, 1, 0),
                            (0, 0, 1, 0), (0, 0, 1, 0)], dtype=[('a', int), ('b', int),
                            ('c', int), ('d', int)])
        assert_equal(test, control)

    def test_leftouter_join(self):
        a, b = self.a, self.b
        test = join_by(('a', 'b'), a, b, 'leftouter')
        control = ma.array([(0, 50, 100, -1), (1, 51, 101, -1),
                            (2, 52, 102, -1), (3, 53, 103, -1),
                            (4, 54, 104, -1), (5, 55, 105, -1),
                            (6, 56, 106, -1), (7, 57, 107, -1),
                            (8, 58, 108, -1), (9, 59, 109, -1)], mask=[[0, 0, 0, 1], (0, 0, 0, 1),
                            (0, 0, 0, 1), (0, 0, 0, 1),
                            (0, 0, 0, 1), (0, 0, 0, 1),
                            (0, 0, 0, 1), (0, 0, 0, 1)], dtype=[('a', int), ('b', int), ('c', int), ('d', int)])
        assert_equal(test, control)

    def test_different_field_order(self):
        a = np.zeros(3, dtype=[('a', 'i4'), ('b', 'f4'), ('c', 'u1')])
        b = np.ones(3, dtype=[('c', 'u1'), ('b', 'f4'), ('a', 'i4')])
        j = join_by(['c', 'b'], a, b, jointype='inner', usemask=False)
        assert_equal(j.dtype.names, ['b', 'c', 'a1', 'a2'])

    def test_duplicate_keys(self):
        a = np.zeros(3, dtype=[('a', 'i4'), ('b', 'f4'), ('c', 'u1')])
        b = np.ones(3, dtype=[('c', 'u1'), ('b', 'f4'), ('a', 'i4')])
        assert_raises(ValueError, join_by, ['a', 'b', 'b'], a, b)

    def test_same_name_different_dtotypes_key(self):
        a_dtype = np.dtype([('key', 'S5'), ('value', '<f4')])
        b_dtype = np.dtype([('key', 'S10'), ('value', '<f4')])
        expected_dtype = np.dtype([
            ('key', 'S10'), ('value1', '<f4'), ('value2', '<f4')])
        a = np.array([('Sarah', 8.0), ('John', 6.0)], dtype=a_dtype)
        b = np.array([('Sarah', 10.0), ('John', 7.0)], dtype=b_dtype)
        res = join_by('key', a, b)
        assert_equal(res.dtype, expected_dtype)

    def test_same_name_different_dtotypes(self):
        a_dtype = np.dtype([('key', 'S10'), ('value', '<f4')])
        b_dtype = np.dtype([('key', 'S10'), ('value', '<f8')])


```

```

699: (8)
700: (12)
701: (8)
702: (8)
703: (8)
704: (8)
705: (4)
706: (8)
707: (8)
708: (8)
709: (8)
710: (8)
711: (8)
712: (8)
713: (8)
714: (8)
715: (4)
716: (8)
717: (8)
718: (8)
719: (8)
720: (8)
721: (8)
722: (8)
723: (12)
724: (8)
725: (8)
726: (0)
727: (4)
728: (4)
729: (8)
730: (34)
731: (25)
732: (8)
733: (34)
734: (25)
735: (4)
736: (8)
737: (8)
738: (12)
739: (8)
740: (28)
741: (28)
742: (28)
743: (28)
744: (27)
745: (34)
746: (8)
747: (4)
748: (8)
749: (22)
750: (4)
751: (8)
752: (8)
753: (12)
754: (8)
755: (28)
756: (28)
757: (28)
758: (28)
759: (27)
760: (34)
761: (8)
762: (4)
763: (8)
764: (30)
765: (21)

    expected_dtype = np.dtype([
        ('key', '|S10'), ('value1', '<f4'), ('value2', '<f8')])
    a = np.array([('Sarah', 8.0), ('John', 6.0)], dtype=a_dtype)
    b = np.array([('Sarah', 10.0), ('John', 7.0)], dtype=b_dtype)
    res = join_by('key', a, b)
    assert_equal(res.dtype, expected_dtype)

def test_subarray_key(self):
    a_dtype = np.dtype([('pos', int, 3), ('f', '<f4')])
    a = np.array([(1, 1, 1), np.pi], [(1, 2, 3), 0.0]), dtype=a_dtype)
    b_dtype = np.dtype([('pos', int, 3), ('g', '<f4')])
    b = np.array([(1, 1, 1), 3], [(3, 2, 1), 0.0]), dtype=b_dtype)
    expected_dtype = np.dtype([('pos', int, 3), ('f', '<f4'), ('g', '<f4')])
    expected = np.array([(1, 1, 1), np.pi, 3]), dtype=expected_dtype)
    res = join_by('pos', a, b)
    assert_equal(res.dtype, expected_dtype)
    assert_equal(res, expected)

def test_padded_dtype(self):
    dt = np.dtype('i1,f4', align=True)
    dt.names = ('k', 'v')
    assert_(len(dt.descr), 3) # padding field is inserted
    a = np.array([(1, 3), (3, 2)], dt)
    b = np.array([(1, 1), (2, 2)], dt)
    res = join_by('k', a, b)
    expected_dtype = np.dtype([
        ('k', 'i1'), ('v1', 'f4'), ('v2', 'f4')])
    assert_equal(res.dtype, expected_dtype)

class TestJoinBy2:
    @classmethod
    def setup_method(cls):
        cls.a = np.array(list(zip(np.arange(10), np.arange(50, 60),
                                  np.arange(100, 110))),
                        dtype=[('a', int), ('b', int), ('c', int)])
        cls.b = np.array(list(zip(np.arange(10), np.arange(65, 75),
                                  np.arange(100, 110))),
                        dtype=[('a', int), ('b', int), ('d', int)])

    def test_no_r1postfix(self):
        a, b = self.a, self.b
        test = join_by(
            'a', a, b, r1postfix='', r2postfix='2', jointype='inner')
        control = np.array([(0, 50, 65, 100, 100), (1, 51, 66, 101, 101),
                            (2, 52, 67, 102, 102), (3, 53, 68, 103, 103),
                            (4, 54, 69, 104, 104), (5, 55, 70, 105, 105),
                            (6, 56, 71, 106, 106), (7, 57, 72, 107, 107),
                            (8, 58, 73, 108, 108), (9, 59, 74, 109, 109)],
                        dtype=[('a', int), ('b', int), ('b2', int),
                               ('c', int), ('d', int)])
        assert_equal(test, control)

    def test_no_postfix(self):
        assert_raises(ValueError, join_by, 'a', self.a, self.b,
                      r1postfix='', r2postfix='')

    def test_no_r2postfix(self):
        a, b = self.a, self.b
        test = join_by(
            'a', a, b, r1postfix='1', r2postfix='', jointype='inner')
        control = np.array([(0, 50, 65, 100, 100), (1, 51, 66, 101, 101),
                            (2, 52, 67, 102, 102), (3, 53, 68, 103, 103),
                            (4, 54, 69, 104, 104), (5, 55, 70, 105, 105),
                            (6, 56, 71, 106, 106), (7, 57, 72, 107, 107),
                            (8, 58, 73, 108, 108), (9, 59, 74, 109, 109)],
                        dtype=[('a', int), ('b1', int), ('b', int),
                               ('c', int), ('d', int)])
        assert_equal(test, control)

    def test_two_keys_two_vars(self):
        a = np.array(list(zip(np.tile([10, 11], 5), np.repeat(np.arange(5),
2),
np.arange(50, 60), np.arange(10, 20))), dtype=[('k', int), ('a', int), ('b', int), ('c', int)])

```

```

766: (8)          b = np.array(list(zip(np.tile([10, 11], 5), np.repeat(np.arange(5),
2),
767: (30)                  np.arange(65, 75), np.arange(0, 10)))),
768: (21)                  dtype=[('k', int), ('a', int), ('b', int), ('c', int)])
769: (8)          control = np.array([(10, 0, 50, 65, 10, 0), (11, 0, 51, 66, 11, 1),
770: (28)                  (10, 1, 52, 67, 12, 2), (11, 1, 53, 68, 13, 3),
771: (28)                  (10, 2, 54, 69, 14, 4), (11, 2, 55, 70, 15, 5),
772: (28)                  (10, 3, 56, 71, 16, 6), (11, 3, 57, 72, 17, 7),
773: (28)                  (10, 4, 58, 73, 18, 8), (11, 4, 59, 74, 19, 9)],
774: (27)                  dtype=[('k', int), ('a', int), ('b1', int),
775: (34)                  ('b2', int), ('c1', int), ('c2', int)])
776: (8)          test = join_by(
777: (12)              ['a', 'k'], a, b, r1postfix='1', r2postfix='2', jointype='inner')
778: (8)          assert_equal(test.dtype, control.dtype)
779: (8)          assert_equal(test, control)
780: (0)          class TestAppendFieldsObj:
781: (4)              """
782: (4)                  Test append_fields with arrays containing objects
783: (4)              """
784: (4)          def setup_method(self):
785: (8)              from datetime import date
786: (8)              self.data = dict(obj=date(2000, 1, 1))
787: (4)          def test_append_to_objects(self):
788: (8)              "Test append_fields when the base array contains objects"
789: (8)              obj = self.data['obj']
790: (8)              x = np.array([(obj, 1.), (obj, 2.)],
791: (22)                  dtype=[('A', object), ('B', float)])
792: (8)              y = np.array([10, 20], dtype=int)
793: (8)              test = append_fields(x, 'C', data=y, usemask=False)
794: (8)              control = np.array([(obj, 1.0, 10), (obj, 2.0, 20)],
795: (27)                  dtype=[('A', object), ('B', float), ('C', int)])
796: (8)              assert_equal(test, control)

```

---

## File 244 - test\_regression.py:

```

1: (0)          import os
2: (0)          import numpy as np
3: (0)          from numpy.testing import (
4: (4)              assert_, assert_equal, assert_array_equal, assert_array_almost_equal,
5: (4)              assert_raises, _assert_valid_refcount,
6: (4)          )
7: (0)          class TestRegression:
8: (4)              def test_poly1d(self):
9: (8)                  assert_equal(np.poly1d([1]) - np.poly1d([1, 0]),
10: (21)                      np.poly1d([-1, 1]))
11: (4)              def test_cov_parameters(self):
12: (8)                  x = np.random.random((3, 3))
13: (8)                  y = x.copy()
14: (8)                  np.cov(x, rowvar=True)
15: (8)                  np.cov(y, rowvar=False)
16: (8)                  assert_array_equal(x, y)
17: (4)              def test_mem_digitize(self):
18: (8)                  for i in range(100):
19: (12)                      np.digitize([1, 2, 3, 4], [1, 3])
20: (12)                      np.digitize([0, 1, 2, 3, 4], [1, 3])
21: (4)              def test_unique_zero_sized(self):
22: (8)                  assert_array_equal([], np.unique(np.array([])))
23: (4)              def test_mem_vectorise(self):
24: (8)                  vt = np.vectorize(lambda *args: args)
25: (8)                  vt(np.zeros((1, 2, 1)), np.zeros((2, 1, 1)), np.zeros((1, 1, 2)))
26: (8)                  vt(np.zeros((1, 2, 1)), np.zeros((2, 1, 1)), np.zeros((1,
27: (11)                      1, 2)), np.zeros((2, 2)))
28: (4)              def test_mgrid_single_element(self):
29: (8)                  assert_array_equal(np.mgrid[0:0:1j], [0])
30: (8)                  assert_array_equal(np.mgrid[0:0], [])
31: (4)              def testRefCount_vectorize(self):
32: (8)                  def p(x, y):

```

```

33: (12)                     return 123
34: (8)                      v = np.vectorize(p)
35: (8)                      _assert_valid_refcount(v)
36: (4) def test_poly1d_nan_roots(self):
37: (8)                      p = np.poly1d([np.nan, np.nan, 1], r=False)
38: (8)                      assert_raises(np.linalg.LinAlgError, getattr, p, "r")
39: (4) def test_mem_polymul(self):
40: (8)                      np.polymul([], [1.])
41: (4) def test_mem_string_concat(self):
42: (8)                      x = np.array([])
43: (8)                      np.append(x, 'asdasd\tasdasd')
44: (4) def test_poly_div(self):
45: (8)                      u = np.poly1d([1, 2, 3])
46: (8)                      v = np.poly1d([1, 2, 3, 4, 5])
47: (8)                      q, r = np.polydiv(u, v)
48: (8)                      assert_equal(q*v + r, u)
49: (4) def test_poly_eq(self):
50: (8)                      x = np.poly1d([1, 2, 3])
51: (8)                      y = np.poly1d([3, 4])
52: (8)                      assert_(x != y)
53: (8)                      assert_(x == x)
54: (4) def test_polyfit_build(self):
55: (8)                      ref = [-1.06123820e-06, 5.70886914e-04, -1.13822012e-01,
56: (15)                           9.95368241e+00, -3.14526520e+02]
57: (8)                      x = [90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103,
58: (13)                           104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
59: (13)                           116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 129,
60: (13)                           130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141,
61: (13)                           146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157,
62: (13)                           158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169,
63: (13)                           170, 171, 172, 173, 174, 175, 176]
64: (8)                      y = [9.0, 3.0, 7.0, 4.0, 4.0, 8.0, 6.0, 11.0, 9.0, 8.0, 11.0, 5.0,
65: (13)                           6.0, 5.0, 9.0, 8.0, 6.0, 10.0, 6.0, 10.0, 7.0, 6.0, 6.0, 6.0,
66: (13)                           13.0, 4.0, 9.0, 11.0, 4.0, 5.0, 8.0, 5.0, 7.0, 7.0, 6.0, 12.0,
67: (13)                           7.0, 7.0, 9.0, 4.0, 12.0, 6.0, 6.0, 4.0, 3.0, 9.0, 8.0, 8.0,
68: (13)                           6.0, 7.0, 9.0, 10.0, 6.0, 8.0, 4.0, 7.0, 7.0, 10.0, 8.0, 8.0,
69: (13)                           6.0, 3.0, 8.0, 4.0, 5.0, 7.0, 8.0, 6.0, 6.0, 4.0, 12.0, 9.0,
70: (13)                           8.0, 8.0, 8.0, 6.0, 7.0, 4.0, 4.0, 5.0, 7.0]
71: (8)                      tested = np.polyfit(x, y, 4)
72: (8)                      assert_array_almost_equal(ref, tested)
73: (4) def test_polydiv_type(self):
74: (8)                      msg = "Wrong type, should be complex"
75: (8)                      x = np.ones(3, dtype=complex)
76: (8)                      q, r = np.polydiv(x, x)
77: (8)                      assert_(q.dtype == complex, msg)
78: (8)                      msg = "Wrong type, should be float"
79: (8)                      x = np.ones(3, dtype=int)
80: (8)                      q, r = np.polydiv(x, x)
81: (8)                      assert_(q.dtype == float, msg)
82: (4) def test_histogramdd_too_many_bins(self):
83: (8)                      assert_raises(ValueError, np.histogramdd, np.ones((1, 10)),
84: bins=2**10)
84: (4) def test_polyint_type(self):
85: (8)                      msg = "Wrong type, should be complex"
86: (8)                      x = np.ones(3, dtype=complex)
87: (8)                      assert_(np.polyint(x).dtype == complex, msg)
88: (8)                      msg = "Wrong type, should be float"
89: (8)                      x = np.ones(3, dtype=int)
90: (8)                      assert_(np.polyint(x).dtype == float, msg)
91: (4) def test_ndenumerate_crash(self):
92: (8)                      list(np.ndenumerate(np.array([[]])))
93: (4) def test_asfarray_none(self):
94: (8)                      assert_array_equal(np.array([np.nan]), np.asarray([None]))
95: (4) def test_large_fancy_indexing(self):
96: (8)                      nbits = np.dtype(np.intp).itemsize * 8
97: (8)                      thesize = int((2**nbits)**(1.0/5.0)+1)
98: (8)                      def dp():
99: (12)                          n = 3
100: (12)                         a = np.ones((n,) * 5)

```

```

101: (12)           i = np.random.randint(0, n, size=thesize)
102: (12)           a[np.ix_(i, i, i, i, i, i)] = 0
103: (8)
104: (12)
105: (12)
106: (12)           def dp2():
107: (12)               n = 3
108: (8)               a = np.ones((n,)*5)
109: (8)               i = np.random.randint(0, n, size=thesize)
110: (4)               a[np.ix_(i, i, i, i, i, i)] = 0
111: (8)               assert_raises(ValueError, dp)
112: (8)               assert_raises(ValueError, dp2)
113: (8)
114: (4)           def test_void_coercion(self):
115: (8)               dt = np.dtype([('a', 'f4'), ('b', 'i4')])
116: (8)               x = np.zeros((1,), dt)
117: (8)               assert_(np.r_[x, x].dtype == dt)
118: (8)
119: (8)
120: (12)
121: (16)           def test_who_with_0dim_array(self):
122: (12)               import os
123: (16)               import sys
124: (8)               oldstdout = sys.stdout
125: (12)               sys.stdout = open(os.devnull, 'w')
126: (12)               try:
127: (8)                   try:
128: (16)                       np.who({'foo': np.array(1)})
129: (12)                   except Exception:
130: (16)                       raise AssertionError("ticket #1243")
131: (8)
132: (4)
133: (8)           finally:
134: (12)               sys.stdout.close()
135: (12)               sys.stdout = oldstdout
136: (8)
137: (4)           def test_include_dirs(self):
138: (8)               include_dirs = [np.get_include()]
139: (8)               for path in include_dirs:
140: (12)                   assert_(isinstance(path, str))
141: (12)                   assert_(path != '')
142: (4)           def test_polyder_return_type(self):
143: (8)               assert_(isinstance(np.polyder(np.poly1d([1]), 0), np.poly1d))
144: (8)               assert_(isinstance(np.polyder([1], 0), np.ndarray))
145: (8)               assert_(isinstance(np.polyder(np.poly1d([1]), 1), np.poly1d))
146: (8)               assert_(isinstance(np.polyder([1], 1), np.ndarray))
147: (4)           def test_append_fields_dtype_list(self):
148: (8)               from numpy.lib.recfunctions import append_fields
149: (8)               base = np.array([1, 2, 3], dtype=np.int32)
150: (8)               names = ['a', 'b', 'c']
151: (8)               data = np.eye(3).astype(np.int32)
152: (8)               dlist = [np.float64, np.int32, np.int32]
153: (8)
154: (8)               try:
155: (12)                   append_fields(base, names, data, dlist)
156: (8)               except Exception:
157: (12)                   raise AssertionError()
158: (8)
159: (4)           def test_loadtxt_fields_subarrays(self):
160: (8)               from io import StringIO
161: (8)               dt = [('a', 'u1', 2), ('b', 'u1', 2)]
162: (8)               x = np.loadtxt(StringIO("0 1 2 3"), dtype=dt)
163: (8)               assert_equal(x, np.array([(0, 1), (2, 3)]), dtype=dt)
164: (8)
165: (12)           def test_nansum_with_boolean(self):
166: (8)               a = np.zeros(2, dtype=bool)
167: (8)               try:
168: (12)                   np.nansum(a)
169: (8)               except Exception:

```

```

                     raise AssertionError()
def test_py3_compat(self):
    class C():

```

```

170: (12)          """Old-style class in python2, normal class in python3"""
171: (12)          pass
172: (8)          out = open(os.devnull, 'w')
173: (8)          try:
174: (12)              np.info(C(), output=out)
175: (8)          except AttributeError:
176: (12)              raise AssertionError()
177: (8)          finally:
178: (12)              out.close()

-----

```

## File 245 - test\_shape\_base.py:

```

1: (0)          import numpy as np
2: (0)          import functools
3: (0)          import sys
4: (0)          import pytest
5: (0)          from numpy.lib.shape_base import (
6: (4)              apply_along_axis, apply_over_axes, array_split, split, hsplit, dsplit,
7: (4)              vsplit, dstack, column_stack, kron, tile, expand_dims, take_along_axis,
8: (4)              put_along_axis
9: (4)          )
10: (0)          from numpy.testing import (
11: (4)              assert_, assert_equal, assert_array_equal, assert_raises, assert_warnings
12: (4)          )
13: (0)          IS_64BIT = sys.maxsize > 2**32
14: (0)          def _add_keepdims(func):
15: (4)              """ hack in keepdims behavior into a function taking an axis """
16: (4)              @functools.wraps(func)
17: (4)              def wrapped(a, axis, **kwargs):
18: (8)                  res = func(a, axis=axis, **kwargs)
19: (8)                  if axis is None:
20: (12)                      axis = 0 # res is now a scalar, so we can insert this anywhere
21: (8)                  return np.expand_dims(res, axis=axis)
22: (4)                  return wrapped
23: (0)          class TestTakeAlongAxis:
24: (4)              def test_argequivalent(self):
25: (8)                  """ Test it translates from arg<func> to <func> """
26: (8)                  from numpy.random import rand
27: (8)                  a = rand(3, 4, 5)
28: (8)                  funcs = [
29: (12)                      (np.sort, np.argsort, dict()),
30: (12)                      (_add_keepdims(np.min), _add_keepdims(np.argmin), dict()),
31: (12)                      (_add_keepdims(np.max), _add_keepdims(np.argmax), dict()),
32: (12)                      (np.partition, np.argpartition, dict(kth=2)),
33: (8)                  ]
34: (8)                  for func, argfunc, kwargs in funcs:
35: (12)                      for axis in list(range(a.ndim)) + [None]:
36: (16)                          a_func = func(a, axis=axis, **kwargs)
37: (16)                          ai_func = argfunc(a, axis=axis, **kwargs)
38: (16)                          assert_equal(a_func, take_along_axis(a, ai_func, axis=axis))
39: (4)              def test_invalid(self):
40: (8)                  """ Test it errors when indices has too few dimensions """
41: (8)                  a = np.ones((10, 10))
42: (8)                  ai = np.ones((10, 2), dtype=np.intp)
43: (8)                  take_along_axis(a, ai, axis=1)
44: (8)                  assert_raises(ValueError, take_along_axis, a, np.array(1), axis=1)
45: (8)                  assert_raises(IndexError, take_along_axis, a, ai.astype(bool), axis=1)
46: (8)                  assert_raises(IndexError, take_along_axis, a, ai.astype(float),
47: (8)                      axis=1)
48: (8)                  assert_raises(np.AxisError, take_along_axis, a, ai, axis=10)
49: (4)              def test_empty(self):
50: (8)                  """ Test everything is ok with empty results, even with inserted dims
51: (8)      """
52: (8)                  a = np.ones((3, 4, 5))
53: (8)                  ai = np.ones((3, 0, 5), dtype=np.intp)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

54: (4)
55: (8)
"""
56: (8)
57: (8)
58: (8)
59: (8)
60: (0)
61: (4)
62: (8)
63: (8)
64: (12)
65: (12)
66: (12)
67: (12)
68: (12)
69: (4)
70: (8)
"""
71: (8)
72: (8)
73: (8)
74: (8)
75: (0)
76: (4)
77: (8)
78: (8)
79: (12)
80: (4)
81: (8)
82: (8)
83: (12)
84: (4)
85: (8)
86: (8)
87: (27)
88: (4)
89: (8)
90: (12)
91: (8)
92: (12)
93: (8)
94: (8)
95: (8)
96: (8)
97: (8)
98: (8)
99: (8)
100: (8)
101: (4)
102: (8)
103: (12)
104: (8)
105: (12)
106: (8)
107: (8)
108: (12)
109: (8)
110: (4)
111: (8)
112: (8)
113: (8)
114: (8)
115: (4)
116: (8)
117: (12)
118: (12)
119: (12)
120: (8)

def test_broadcast(self):
    """ Test that non-indexing dimensions are broadcast in both directions

    a = np.ones((3, 4, 1))
    ai = np.ones((1, 2, 5), dtype=np.intp)
    actual = take_along_axis(a, ai, axis=1)
    assert_equal(actual.shape, (3, 2, 5))

class TestPutAlongAxis:
    def test_replace_max(self):
        a_base = np.array([[10, 30, 20], [60, 40, 50]])
        for axis in list(range(a_base.ndim)) + [None]:
            a = a_base.copy()
            i_max = _add_kepdims(np.argmax)(a, axis=axis)
            put_along_axis(a, i_max, -99, axis=axis)
            i_min = _add_kepdims(np.argmin)(a, axis=axis)
            assert_equal(i_min, i_max)

    def test_broadcast(self):
        """ Test that non-indexing dimensions are broadcast in both directions

        a = np.ones((3, 4, 1))
        ai = np.arange(10, dtype=np.intp).reshape((1, 2, 5)) % 4
        put_along_axis(a, ai, 20, axis=1)
        assert_equal(take_along_axis(a, ai, axis=1), 20)

class TestApplyAlongAxis:
    def test_simple(self):
        a = np.ones((20, 10), 'd')
        assert_array_equal(
            apply_along_axis(len, 0, a), len(a)*np.ones(a.shape[1]))

    def test_simple101(self):
        a = np.ones((10, 101), 'd')
        assert_array_equal(
            apply_along_axis(len, 0, a), len(a)*np.ones(a.shape[1]))

    def test_3d(self):
        a = np.arange(27).reshape((3, 3, 3))
        assert_array_equal(apply_along_axis(np.sum, 0, a),
                           [[27, 30, 33], [36, 39, 42], [45, 48, 51]])

    def test_preserve_subclass(self):
        def double(row):
            return row * 2

        class MyNDArray(np.ndarray):
            pass

        m = np.array([[0, 1], [2, 3]]).view(MyNDArray)
        expected = np.array([[0, 2], [4, 6]]).view(MyNDArray)
        result = apply_along_axis(double, 0, m)
        assert_(isinstance(result, MyNDArray))
        assert_array_equal(result, expected)

        result = apply_along_axis(double, 1, m)
        assert_(isinstance(result, MyNDArray))
        assert_array_equal(result, expected)

    def test_subclass(self):
        class MinimalSubclass(np.ndarray):
            data = 1

        def minimal_function(array):
            return array.data

        a = np.zeros((6, 3)).view(MinimalSubclass)
        assert_array_equal(
            apply_along_axis(minimal_function, 0, a), np.array([1, 1, 1]))
"""

def test_scalar_array(self, cls=np.ndarray):
    a = np.ones((6, 3)).view(cls)
    res = apply_along_axis(np.sum, 0, a)
    assert_(isinstance(res, cls))
    assert_array_equal(res, np.array([6, 6, 6]).view(cls))

def test_0d_array(self, cls=np.ndarray):
    def sum_to_0d(x):
        """ Sum x, returning a 0d array of the same class """
        assert_equal(x.ndim, 1)
        return np.squeeze(np.sum(x, keepdims=True))

    a = np.ones((6, 3)).view(cls)

```

```

121: (8)             res = apply_along_axis(sum_to_0d, 0, a)
122: (8)             assert_(isinstance(res, cls))
123: (8)             assert_array_equal(res, np.array([6, 6, 6]).view(cls))
124: (8)             res = apply_along_axis(sum_to_0d, 1, a)
125: (8)             assert_(isinstance(res, cls))
126: (8)             assert_array_equal(res, np.array([3, 3, 3, 3, 3, 3]).view(cls))
127: (4)             def test_axis_insertion(self, cls=np.ndarray):
128: (8)                 def f1to2(x):
129: (12)                     """produces an asymmetric non-square matrix from x"""
130: (12)                     assert_equal(x.ndim, 1)
131: (12)                     return (x[::-1] * x[1:,None]).view(cls)
132: (8)                     a2d = np.arange(6*3).reshape((6, 3))
133: (8)                     actual = apply_along_axis(f1to2, 0, a2d)
134: (8)                     expected = np.stack([
135: (12)                         f1to2(a2d[:,i]) for i in range(a2d.shape[1])
136: (8)                     ], axis=-1).view(cls)
137: (8)                     assert_equal(type(actual), type(expected))
138: (8)                     assert_equal(actual, expected)
139: (8)                     actual = apply_along_axis(f1to2, 1, a2d)
140: (8)                     expected = np.stack([
141: (12)                         f1to2(a2d[i,:]) for i in range(a2d.shape[0])
142: (8)                     ], axis=0).view(cls)
143: (8)                     assert_equal(type(actual), type(expected))
144: (8)                     assert_equal(actual, expected)
145: (8)                     a3d = np.arange(6*5*3).reshape((6, 5, 3))
146: (8)                     actual = apply_along_axis(f1to2, 1, a3d)
147: (8)                     expected = np.stack([
148: (12)                         np.stack([
149: (16)                             f1to2(a3d[i,:,j]) for i in range(a3d.shape[0])
150: (12)                         ], axis=0)
151: (12)                             for j in range(a3d.shape[2])
152: (8)                         ], axis=-1).view(cls)
153: (8)                         assert_equal(type(actual), type(expected))
154: (8)                         assert_equal(actual, expected)
155: (4)             def test_subclass_preservation(self):
156: (8)                 class MinimalSubclass(np.ndarray):
157: (12)                     pass
158: (8)                     self.test_scalar_array(MinimalSubclass)
159: (8)                     self.test_0d_array(MinimalSubclass)
160: (8)                     self.test_axis_insertion(MinimalSubclass)
161: (4)             def test_axis_insertion_ma(self):
162: (8)                 def f1to2(x):
163: (12)                     """produces an asymmetric non-square matrix from x"""
164: (12)                     assert_equal(x.ndim, 1)
165: (12)                     res = x[::-1] * x[1:,None]
166: (12)                     return np.ma.masked_where(res%5==0, res)
167: (8)                     a = np.arange(6*3).reshape((6, 3))
168: (8)                     res = apply_along_axis(f1to2, 0, a)
169: (8)                     assert_(isinstance(res, np.ma.masked_array))
170: (8)                     assert_equal(res.ndim, 3)
171: (8)                     assert_array_equal(res[:, :, 0].mask, f1to2(a[:, 0]).mask)
172: (8)                     assert_array_equal(res[:, :, 1].mask, f1to2(a[:, 1]).mask)
173: (8)                     assert_array_equal(res[:, :, 2].mask, f1to2(a[:, 2]).mask)
174: (4)             def test_tuple_func1d(self):
175: (8)                 def sample_1d(x):
176: (12)                     return x[1], x[0]
177: (8)                     res = np.apply_along_axis(sample_1d, 1, np.array([[1, 2], [3, 4]]))
178: (8)                     assert_array_equal(res, np.array([[2, 1], [4, 3]]))
179: (4)             def test_empty(self):
180: (8)                 def never_call(x):
181: (12)                     assert_(False) # should never be reached
182: (8)                     a = np.empty((0, 0))
183: (8)                     assert_raises(ValueError, np.apply_along_axis, never_call, 0, a)
184: (8)                     assert_raises(ValueError, np.apply_along_axis, never_call, 1, a)
185: (8)                     def empty_to_1(x):
186: (12)                         assert_(len(x) == 0)
187: (12)                         return 1
188: (8)                         a = np.empty((10, 0))
189: (8)                         actual = np.apply_along_axis(empty_to_1, 1, a)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

190: (8) assert_equal(actual, np.ones(10))
191: (8) assert_raises(ValueError, np.apply_along_axis, empty_to_1, 0, a)
192: (4) def test_with_iterable_object(self):
193: (8)     d = np.array([
194: (12)         [{1, 11}, {2, 22}, {3, 33}],
195: (12)         [{4, 44}, {5, 55}, {6, 66}]
196: (8)     ])
197: (8)     actual = np.apply_along_axis(lambda a: set.union(*a), 0, d)
198: (8)     expected = np.array([{1, 11, 4, 44}, {2, 22, 5, 55}, {3, 33, 6, 66}])
199: (8)     assert_equal(actual, expected)
200: (8)     for i in np.ndindex(actual.shape):
201: (12)         assert_equal(type(actual[i]), type(expected[i]))
202: (0) class TestApplyOverAxes:
203: (4)     def test_simple(self):
204: (8)         a = np.arange(24).reshape(2, 3, 4)
205: (8)         aoa_a = apply_over_axes(np.sum, a, [0, 2])
206: (8)         assert_array_equal(aoa_a, np.array([[60], [92], [124]]))
207: (0) class TestExpandDims:
208: (4)     def test_functionality(self):
209: (8)         s = (2, 3, 4, 5)
210: (8)         a = np.empty(s)
211: (8)         for axis in range(-5, 4):
212: (12)             b = expand_dims(a, axis)
213: (12)             assert_(b.shape[axis] == 1)
214: (12)             assert_(np.squeeze(b).shape == s)
215: (4)     def test_axis_tuple(self):
216: (8)         a = np.empty((3, 3, 3))
217: (8)         assert np.expand_dims(a, axis=(0, 1, 2)).shape == (1, 1, 1, 3, 3, 3)
218: (8)         assert np.expand_dims(a, axis=(0, -1, -2)).shape == (1, 3, 3, 3, 1, 1)
219: (8)         assert np.expand_dims(a, axis=(0, 3, 5)).shape == (1, 3, 3, 1, 3, 1)
220: (8)         assert np.expand_dims(a, axis=(0, -3, -5)).shape == (1, 1, 3, 1, 3, 3)
221: (4)     def test_axis_out_of_range(self):
222: (8)         s = (2, 3, 4, 5)
223: (8)         a = np.empty(s)
224: (8)         assert_raises(np.AxisError, expand_dims, a, -6)
225: (8)         assert_raises(np.AxisError, expand_dims, a, 5)
226: (8)         a = np.empty((3, 3, 3))
227: (8)         assert_raises(np.AxisError, expand_dims, a, (0, -6))
228: (8)         assert_raises(np.AxisError, expand_dims, a, (0, 5))
229: (4)     def test_repeated_axis(self):
230: (8)         a = np.empty((3, 3, 3))
231: (8)         assert_raises(ValueError, expand_dims, a, axis=(1, 1))
232: (4)     def test_subclasses(self):
233: (8)         a = np.arange(10).reshape((2, 5))
234: (8)         a = np.ma.array(a, mask=a%3 == 0)
235: (8)         expanded = np.expand_dims(a, axis=1)
236: (8)         assert_(isinstance(expanded, np.ma.MaskedArray))
237: (8)         assert_equal(expanded.shape, (2, 1, 5))
238: (8)         assert_equal(expanded.mask.shape, (2, 1, 5))
239: (0) class TestArraySplit:
240: (4)     def test_integer_0_split(self):
241: (8)         a = np.arange(10)
242: (8)         assert_raises(ValueError, array_split, a, 0)
243: (4)     def test_integer_split(self):
244: (8)         a = np.arange(10)
245: (8)         res = array_split(a, 1)
246: (8)         desired = [np.arange(10)]
247: (8)         compare_results(res, desired)
248: (8)         res = array_split(a, 2)
249: (8)         desired = [np.arange(5), np.arange(5, 10)]
250: (8)         compare_results(res, desired)
251: (8)         res = array_split(a, 3)
252: (8)         desired = [np.arange(4), np.arange(4, 7), np.arange(7, 10)]
253: (8)         compare_results(res, desired)
254: (8)         res = array_split(a, 4)
255: (8)         desired = [np.arange(3), np.arange(3, 6), np.arange(6, 8),
256: (19)             np.arange(8, 10)]
257: (8)         compare_results(res, desired)
258: (8)         res = array_split(a, 5)

```

```

259: (8)             desired = [np.arange(2), np.arange(2, 4), np.arange(4, 6),
260: (19)             np.arange(6, 8), np.arange(8, 10)]
261: (8)             compare_results(res, desired)
262: (8)             res = array_split(a, 6)
263: (8)             desired = [np.arange(2), np.arange(2, 4), np.arange(4, 6),
264: (19)             np.arange(6, 8), np.arange(8, 9), np.arange(9, 10)]
265: (8)             compare_results(res, desired)
266: (8)             res = array_split(a, 7)
267: (8)             desired = [np.arange(2), np.arange(2, 4), np.arange(4, 6),
268: (19)             np.arange(6, 7), np.arange(7, 8), np.arange(8, 9),
269: (19)             np.arange(9, 10)]
270: (8)             compare_results(res, desired)
271: (8)             res = array_split(a, 8)
272: (8)             desired = [np.arange(2), np.arange(2, 4), np.arange(4, 5),
273: (19)             np.arange(5, 6), np.arange(6, 7), np.arange(7, 8),
274: (19)             np.arange(8, 9), np.arange(9, 10)]
275: (8)             compare_results(res, desired)
276: (8)             res = array_split(a, 9)
277: (8)             desired = [np.arange(2), np.arange(2, 3), np.arange(3, 4),
278: (19)             np.arange(4, 5), np.arange(5, 6), np.arange(6, 7),
279: (19)             np.arange(7, 8), np.arange(8, 9), np.arange(9, 10)]
280: (8)             compare_results(res, desired)
281: (8)             res = array_split(a, 10)
282: (8)             desired = [np.arange(1), np.arange(1, 2), np.arange(2, 3),
283: (19)             np.arange(3, 4), np.arange(4, 5), np.arange(5, 6),
284: (19)             np.arange(6, 7), np.arange(7, 8), np.arange(8, 9),
285: (19)             np.arange(9, 10)]
286: (8)             compare_results(res, desired)
287: (8)             res = array_split(a, 11)
288: (8)             desired = [np.arange(1), np.arange(1, 2), np.arange(2, 3),
289: (19)             np.arange(3, 4), np.arange(4, 5), np.arange(5, 6),
290: (19)             np.arange(6, 7), np.arange(7, 8), np.arange(8, 9),
291: (19)             np.arange(9, 10), np.array([])]
292: (8)             compare_results(res, desired)
293: (4)             def test_integer_split_2D_rows(self):
294: (8)                 a = np.array([np.arange(10), np.arange(10)])
295: (8)                 res = array_split(a, 3, axis=0)
296: (8)                 tgt = [np.array([np.arange(10)]), np.array([np.arange(10)]),
297: (19)                   np.zeros((0, 10))]
298: (8)                 compare_results(res, tgt)
299: (8)                 assert_(a.dtype.type is res[-1].dtype.type)
300: (8)                 res = array_split(a, [0, 1], axis=0)
301: (8)                 tgt = [np.zeros((0, 10)), np.array([np.arange(10)]),
302: (15)                   np.array([np.arange(10)])]
303: (8)                 compare_results(res, tgt)
304: (8)                 assert_(a.dtype.type is res[-1].dtype.type)
305: (4)             def test_integer_split_2D_cols(self):
306: (8)                 a = np.array([np.arange(10), np.arange(10)])
307: (8)                 res = array_split(a, 3, axis=-1)
308: (8)                 desired = [np.array([np.arange(4), np.arange(4)]),
309: (19)                   np.array([np.arange(4, 7), np.arange(4, 7)]),
310: (19)                   np.array([np.arange(7, 10), np.arange(7, 10)])]
311: (8)                 compare_results(res, desired)
312: (4)             def test_integer_split_2D_default(self):
313: (8)                 """ This will fail if we change default axis
314: (8)                 """
315: (8)                 a = np.array([np.arange(10), np.arange(10)])
316: (8)                 res = array_split(a, 3)
317: (8)                 tgt = [np.array([np.arange(10)]), np.array([np.arange(10)]),
318: (19)                   np.zeros((0, 10))]
319: (8)                 compare_results(res, tgt)
320: (8)                 assert_(a.dtype.type is res[-1].dtype.type)
321: (4)             @pytest.mark.skipif(not IS_64BIT, reason="Needs 64bit platform")
322: (4)             def test_integer_split_2D_rows_greater_max_int32(self):
323: (8)                 a = np.broadcast_to([0], (1 << 32, 2))
324: (8)                 res = array_split(a, 4)
325: (8)                 chunk = np.broadcast_to([0], (1 << 30, 2))
326: (8)                 tgt = [chunk] * 4
327: (8)                 for i in range(len(tgt)):
```

```

328: (12)                                assert_equal(res[i].shape, tgt[i].shape)
329: (4)       def test_index_split_simple(self):
330: (8)           a = np.arange(10)
331: (8)           indices = [1, 5, 7]
332: (8)           res = array_split(a, indices, axis=-1)
333: (8)           desired = [np.arange(0, 1), np.arange(1, 5), np.arange(5, 7),
334: (19)                           np.arange(7, 10)]
335: (8)           compare_results(res, desired)
336: (4)       def test_index_split_low_bound(self):
337: (8)           a = np.arange(10)
338: (8)           indices = [0, 5, 7]
339: (8)           res = array_split(a, indices, axis=-1)
340: (8)           desired = [np.array([]), np.arange(0, 5), np.arange(5, 7),
341: (19)                           np.arange(7, 10)]
342: (8)           compare_results(res, desired)
343: (4)       def test_index_split_high_bound(self):
344: (8)           a = np.arange(10)
345: (8)           indices = [0, 5, 7, 10, 12]
346: (8)           res = array_split(a, indices, axis=-1)
347: (8)           desired = [np.array([]), np.arange(0, 5), np.arange(5, 7),
348: (19)                           np.arange(7, 10), np.array([]), np.array([])]
349: (8)           compare_results(res, desired)
350: (0) class TestSplit:
351: (4)     def test_equal_split(self):
352: (8)         a = np.arange(10)
353: (8)         res = split(a, 2)
354: (8)         desired = [np.arange(5), np.arange(5, 10)]
355: (8)         compare_results(res, desired)
356: (4)     def test_unequal_split(self):
357: (8)         a = np.arange(10)
358: (8)         assert_raises(ValueError, split, a, 3)
359: (0) class TestColumnStack:
360: (4)     def test_non_iterable(self):
361: (8)         assert_raises(TypeError, column_stack, 1)
362: (4)     def test_1D_arrays(self):
363: (8)         a = np.array((1, 2, 3))
364: (8)         b = np.array((2, 3, 4))
365: (8)         expected = np.array([[1, 2],
366: (29)                         [2, 3],
367: (29)                         [3, 4]])
368: (8)         actual = np.column_stack((a, b))
369: (8)         assert_equal(actual, expected)
370: (4)     def test_2D_arrays(self):
371: (8)         a = np.array([[1], [2], [3]])
372: (8)         b = np.array([[2], [3], [4]])
373: (8)         expected = np.array([[1, 2],
374: (29)                         [2, 3],
375: (29)                         [3, 4]])
376: (8)         actual = np.column_stack((a, b))
377: (8)         assert_equal(actual, expected)
378: (4)     def test_generator(self):
379: (8)         with pytest.raises(TypeError, match="arrays to stack must be"):
380: (12)             column_stack((np.arange(3) for _ in range(2)))
381: (0) class TestDstack:
382: (4)     def test_non_iterable(self):
383: (8)         assert_raises(TypeError, dstack, 1)
384: (4)     def test_0D_array(self):
385: (8)         a = np.array(1)
386: (8)         b = np.array(2)
387: (8)         res = dstack([a, b])
388: (8)         desired = np.array([[1, 2]])
389: (8)         assert_array_equal(res, desired)
390: (4)     def test_1D_array(self):
391: (8)         a = np.array([1])
392: (8)         b = np.array([2])
393: (8)         res = dstack([a, b])
394: (8)         desired = np.array([[1, 2]])
395: (8)         assert_array_equal(res, desired)
396: (4)     def test_2D_array(self):

```

```

397: (8)             a = np.array([[1], [2]])
398: (8)             b = np.array([[1], [2]])
399: (8)             res = dstack([a, b])
400: (8)             desired = np.array([[1, 1], [2, 2, ]]))
401: (8)             assert_array_equal(res, desired)
402: (4)             def test_2D_array2(self):
403: (8)                 a = np.array([1, 2])
404: (8)                 b = np.array([1, 2])
405: (8)                 res = dstack([a, b])
406: (8)                 desired = np.array([[1, 1], [2, 2]]))
407: (8)                 assert_array_equal(res, desired)
408: (4)             def test_generator(self):
409: (8)                 with pytest.raises(TypeError, match="arrays to stack must be"):
410: (12)                     dstack((np.arange(3) for _ in range(2)))
411: (0)             class TestHsplit:
412: (4)                 """Only testing for integer splits.
413: (4) """
414: (4)                 def test_non_iterable(self):
415: (8)                     assert_raises(ValueError, hsplit, 1, 1)
416: (4)                 def test_0D_array(self):
417: (8)                     a = np.array(1)
418: (8)                     try:
419: (12)                         hsplit(a, 2)
420: (12)                         assert_(0)
421: (8)                     except ValueError:
422: (12)                         pass
423: (4)                 def test_1D_array(self):
424: (8)                     a = np.array([1, 2, 3, 4])
425: (8)                     res = hsplit(a, 2)
426: (8)                     desired = [np.array([1, 2]), np.array([3, 4])]
427: (8)                     compare_results(res, desired)
428: (4)                 def test_2D_array(self):
429: (8)                     a = np.array([[1, 2, 3, 4],
430: (18)                         [1, 2, 3, 4]])
431: (8)                     res = hsplit(a, 2)
432: (8)                     desired = [np.array([[1, 2], [1, 2]]), np.array([[3, 4], [3, 4]])]
433: (8)                     compare_results(res, desired)
434: (0)             class TestVsplit:
435: (4)                 """Only testing for integer splits.
436: (4) """
437: (4)                 def test_non_iterable(self):
438: (8)                     assert_raises(ValueError, vsplit, 1, 1)
439: (4)                 def test_0D_array(self):
440: (8)                     a = np.array(1)
441: (8)                     assert_raises(ValueError, vsplit, a, 2)
442: (4)                 def test_1D_array(self):
443: (8)                     a = np.array([1, 2, 3, 4])
444: (8)                     try:
445: (12)                         vsplit(a, 2)
446: (12)                         assert_(0)
447: (8)                     except ValueError:
448: (12)                         pass
449: (4)                 def test_2D_array(self):
450: (8)                     a = np.array([[1, 2, 3, 4],
451: (18)                         [1, 2, 3, 4]])
452: (8)                     res = vsplit(a, 2)
453: (8)                     desired = [np.array([[1, 2, 3, 4]]), np.array([[1, 2, 3, 4]])]
454: (8)                     compare_results(res, desired)
455: (0)             class TestDsplit:
456: (4)                 def test_non_iterable(self):
457: (8)                     assert_raises(ValueError, dsplit, 1, 1)
458: (4)                 def test_0D_array(self):
459: (8)                     a = np.array(1)
460: (8)                     assert_raises(ValueError, dsplit, a, 2)
461: (4)                 def test_1D_array(self):
462: (8)                     a = np.array([1, 2, 3, 4])
463: (8)                     assert_raises(ValueError, dsplit, a, 2)
464: (4)                 def test_2D_array(self):
465: (8)                     a = np.array([[1, 2, 3, 4],

```

```

466: (18)                               [1, 2, 3, 4]])
467: (8)       try:
468: (12)         dsplit(a, 2)
469: (12)         assert_(0)
470: (8)       except ValueError:
471: (12)         pass
472: (4)       def test_3D_array(self):
473: (8)         a = np.array([[[1, 2, 3, 4],
474: (19)             [1, 2, 3, 4]],
475: (18)             [[1, 2, 3, 4],
476: (19)                 [1, 2, 3, 4]]])
477: (8)         res = dsplit(a, 2)
478: (8)         desired = [np.array([[1, 2], [1, 2]], [[1, 2], [1, 2]]]),
479: (19)             np.array([[3, 4], [3, 4]], [[3, 4], [3, 4]]])
480: (8)         compare_results(res, desired)
481: (0)   class TestSqueeze:
482: (4)     def test_basic(self):
483: (8)       from numpy.random import rand
484: (8)       a = rand(20, 10, 10, 1, 1)
485: (8)       b = rand(20, 1, 10, 1, 20)
486: (8)       c = rand(1, 1, 20, 10)
487: (8)       assert_array_equal(np.squeeze(a), np.reshape(a, (20, 10, 10)))
488: (8)       assert_array_equal(np.squeeze(b), np.reshape(b, (20, 10, 20)))
489: (8)       assert_array_equal(np.squeeze(c), np.reshape(c, (20, 10)))
490: (8)       a = [[[1.5]]]
491: (8)       res = np.squeeze(a)
492: (8)       assert_equal(res, 1.5)
493: (8)       assert_equal(res.ndim, 0)
494: (8)       assert_equal(type(res), np.ndarray)
495: (0)   class TestKron:
496: (4)     def test_basic(self):
497: (8)       a = np.array(1)
498: (8)       b = np.array([[1, 2], [3, 4]])
499: (8)       k = np.array([[1, 2], [3, 4]])
500: (8)       assert_array_equal(np.kron(a, b), k)
501: (8)       a = np.array([[1, 2], [3, 4]])
502: (8)       b = np.array(1)
503: (8)       assert_array_equal(np.kron(a, b), k)
504: (8)       a = np.array([3])
505: (8)       b = np.array([[1, 2], [3, 4]])
506: (8)       k = np.array([[3, 6], [9, 12]])
507: (8)       assert_array_equal(np.kron(a, b), k)
508: (8)       a = np.array([[1, 2], [3, 4]])
509: (8)       b = np.array([3])
510: (8)       assert_array_equal(np.kron(a, b), k)
511: (8)       a = np.array([[1], [[2]]])
512: (8)       b = np.array([[1, 2], [3, 4]])
513: (8)       k = np.array([[1, 2], [3, 4], [[2, 4], [6, 8]]])
514: (8)       assert_array_equal(np.kron(a, b), k)
515: (8)       a = np.array([[1, 2], [3, 4]])
516: (8)       b = np.array([[1], [[2]]])
517: (8)       k = np.array([[1, 2], [3, 4], [[2, 4], [6, 8]]])
518: (8)       assert_array_equal(np.kron(a, b), k)
519: (4)     def test_return_type(self):
520: (8)       class myarray(np.ndarray):
521: (12)         __array_priority__ = 1.0
522: (8)       a = np.ones([2, 2])
523: (8)       ma = myarray(a.shape, a.dtype, a.data)
524: (8)       assert_equal(type(kron(a, a)), np.ndarray)
525: (8)       assert_equal(type(kron(ma, ma)), myarray)
526: (8)       assert_equal(type(kron(a, ma)), myarray)
527: (8)       assert_equal(type(kron(ma, a)), myarray)
528: (4)     @pytest.mark.parametrize(
529: (8)       "array_class", [np.asarray, np.mat]
530: (4)     )
531: (4)     def test_kron_smoke(self, array_class):
532: (8)       a = array_class(np.ones([3, 3]))
533: (8)       b = array_class(np.ones([3, 3]))
534: (8)       k = array_class(np.ones([9, 9]))

```

```

535: (8)             assert_array_equal(np.kron(a, b), k)
536: (4)             def test_kron_ma(self):
537: (8)                 x = np.ma.array([[1, 2], [3, 4]], mask=[[0, 1], [1, 0]])
538: (8)                 k = np.ma.array(np.diag([1, 4, 4, 16]),
539: (16)                               mask=~np.array(np.identity(4), dtype=bool))
540: (8)                 assert_array_equal(k, np.kron(x, x))
541: (4)             @pytest.mark.parametrize(
542: (8)                 "shape_a,shape_b", [
543: (12)                   ((1, 1), (1, 1)),
544: (12)                   ((1, 2, 3), (4, 5, 6)),
545: (12)                   ((2, 2), (2, 2, 2)),
546: (12)                   ((1, 0), (1, 1)),
547: (12)                   ((2, 0, 2), (2, 2)),
548: (12)                   ((2, 0, 0, 2), (2, 0, 2)),
549: (8)               ])
550: (4)             def test_kron_shape(self, shape_a, shape_b):
551: (8)                 a = np.ones(shape_a)
552: (8)                 b = np.ones(shape_b)
553: (8)                 normalised_shape_a = (1,) * max(0, len(shape_b)-len(shape_a)) +
554: (8)                     normalised_shape_b = (1,) * max(0, len(shape_a)-len(shape_b)) +
555: (8)                     expected_shape = np.multiply(normalised_shape_a, normalised_shape_b)
556: (8)                 k = np.kron(a, b)
557: (8)                 assert np.array_equal(
558: (16)                               k.shape, expected_shape), "Unexpected shape from kron"
559: (0)             class TestTile:
560: (4)                 def test_basic(self):
561: (8)                     a = np.array([0, 1, 2])
562: (8)                     b = [[1, 2], [3, 4]]
563: (8)                     assert_equal(tile(a, 2), [0, 1, 2, 0, 1, 2])
564: (8)                     assert_equal(tile(a, (2, 2)), [[0, 1, 2, 0, 1, 2], [0, 1, 2, 0, 1,
565: (2)])
566: (8)                     assert_equal(tile(a, (1, 2)), [[0, 1, 2, 0, 1, 2]])
567: (8)                     assert_equal(tile(b, 2), [[1, 2, 1, 2], [3, 4, 3, 4]])
568: (8)                     assert_equal(tile(b, (2, 1)), [[1, 2], [3, 4], [1, 2], [3, 4]])
569: (39)                    assert_equal(tile(b, (2, 2)), [[1, 2, 1, 2], [3, 4, 3, 4],
570: (4)                        [1, 2, 1, 2], [3, 4, 3, 4]])
571: (8)             def test_tile_one_repetition_on_array_gh4679(self):
572: (8)                 a = np.arange(5)
573: (8)                 b = tile(a, 1)
574: (8)                 b += 2
575: (8)                 assert_equal(a, np.arange(5))
576: (4)             def test_empty(self):
577: (8)                 a = np.array([[[[]]]])
578: (8)                 b = np.array([[], []])
579: (8)                 c = tile(b, 2).shape
580: (8)                 d = tile(a, (3, 2, 5)).shape
581: (8)                 assert_equal(c, (2, 0))
582: (4)                 assert_equal(d, (3, 2, 0))
583: (8)             def test_kroncompare(self):
584: (8)                 from numpy.random import randint
585: (8)                 reps = [(2,), (1, 2), (2, 1), (2, 2), (2, 3, 2), (3, 2)]
586: (8)                 shape = [(3,), (2, 3), (3, 4, 3), (3, 2, 3), (4, 3, 2, 4), (2, 2)]
587: (12)                for s in shape:
588: (12)                    b = randint(0, 10, size=s)
589: (16)                    for r in reps:
590: (16)                        a = np.ones(r, b.dtype)
591: (16)                        large = tile(b, r)
592: (16)                        klarge = kron(a, b)
593: (0)                        assert_equal(large, klarge)
594: (4)             class TestMayShareMemory:
595: (8)                 def test_basic(self):
596: (8)                     d = np.ones((50, 60))
597: (8)                     d2 = np.ones((30, 60, 6))
598: (8)                     assert_(np.may_share_memory(d, d))
599: (8)                     assert_(np.may_share_memory(d, d[:-1]))
600: (8)                     assert_(np.may_share_memory(d, d[:2]))
601: (8)                     assert_(np.may_share_memory(d, d[1:, :-1]))
```

```

601: (8)             assert_(not np.may_share_memory(d[::-1], d2))
602: (8)             assert_(not np.may_share_memory(d[::-2], d2))
603: (8)             assert_(not np.may_share_memory(d[1:, ::-1], d2))
604: (8)             assert_(np.may_share_memory(d2[1:, ::-1], d2))
605: (0)  def compare_results(res, desired):
606: (4)      """Compare lists of arrays."""
607: (4)      if len(res) != len(desired):
608: (8)          raise ValueError("Iterables have different lengths")
609: (4)      for x, y in zip(res, desired):
610: (8)          assert_array_equal(x, y)

```

---

## File 246 - test\_stride\_tricks.py:

```

1: (0)          import numpy as np
2: (0)  from numpy.core._rational_tests import rational
3: (0)  from numpy.testing import (
4: (4)      assert_equal, assert_array_equal, assert_raises, assert_,
5: (4)      assert_raises_regex, assert_warnings,
6: (4)      )
7: (0)  from numpy.lib.stride_tricks import (
8: (4)      as_strided, broadcast_arrays, _broadcast_shape, broadcast_to,
9: (4)      broadcast_shapes, sliding_window_view,
10: (4)      )
11: (0) import pytest
12: (0) def assert_shapes_correct(input_shapes, expected_shape):
13: (4)     inarrays = [np.zeros(s) for s in input_shapes]
14: (4)     outarrays = broadcast_arrays(*inarrays)
15: (4)     outshapes = [a.shape for a in outarrays]
16: (4)     expected = [expected_shape] * len(inarrays)
17: (4)     assert_equal(outshapes, expected)
18: (0) def assert_incompatible_shapes_raise(input_shapes):
19: (4)     inarrays = [np.zeros(s) for s in input_shapes]
20: (4)     assert_raises(ValueError, broadcast_arrays, *inarrays)
21: (0) def assert_same_as_ufunc(shape0, shape1, transposed=False, flipped=False):
22: (4)     x0 = np.zeros(shape0, dtype=int)
23: (4)     n = int(np.multiply.reduce(shape1))
24: (4)     x1 = np.arange(n).reshape(shape1)
25: (4)     if transposed:
26: (8)         x0 = x0.T
27: (8)         x1 = x1.T
28: (4)     if flipped:
29: (8)         x0 = x0[::-1]
30: (8)         x1 = x1[::-1]
31: (4)     y = x0 + x1
32: (4)     b0, b1 = broadcast_arrays(x0, x1)
33: (4)     assert_array_equal(y, b1)
34: (0) def test_same():
35: (4)     x = np.arange(10)
36: (4)     y = np.arange(10)
37: (4)     bx, by = broadcast_arrays(x, y)
38: (4)     assert_array_equal(x, bx)
39: (4)     assert_array_equal(y, by)
40: (0) def test_broadcast_kwargs():
41: (4)     x = np.arange(10)
42: (4)     y = np.arange(10)
43: (4)     with assert_raises_regex(TypeError, 'got an unexpected keyword'):
44: (8)         broadcast_arrays(x, y, dtype='float64')
45: (0) def test_one_off():
46: (4)     x = np.array([[1, 2, 3]])
47: (4)     y = np.array([[1], [2], [3]])
48: (4)     bx, by = broadcast_arrays(x, y)
49: (4)     bx0 = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
50: (4)     by0 = bx0.T
51: (4)     assert_array_equal(bx0, bx)
52: (4)     assert_array_equal(by0, by)
53: (0) def test_same_input_shapes():
54: (4)     data = [

```

```

55: (8)                  (),
56: (8)                  (1,),
57: (8)                  (3,),
58: (8)                  (0, 1),
59: (8)                  (0, 3),
60: (8)                  (1, 0),
61: (8)                  (3, 0),
62: (8)                  (1, 3),
63: (8)                  (3, 1),
64: (8)                  (3, 3),
65: (4)
66: (4)      for shape in data:
67: (8)          input_shapes = [shape]
68: (8)          assert_shapes_correct(input_shapes, shape)
69: (8)          input_shapes2 = [shape, shape]
70: (8)          assert_shapes_correct(input_shapes2, shape)
71: (8)          input_shapes3 = [shape, shape, shape]
72: (8)          assert_shapes_correct(input_shapes3, shape)
73: (0)      def test_two_compatible_by_ones_input_shapes():
74: (4)          data = [
75: (8)              [[(1,), (3,)], (3,)],
76: (8)              [[(1, 3), (3, 3)], (3, 3)],
77: (8)              [[[3, 1), (3, 3)], (3, 3)],
78: (8)              [[[1, 3), (3, 1)], (3, 3)],
79: (8)              [[[1, 1), (3, 3)], (3, 3)],
80: (8)              [[[1, 1), (1, 3)], (1, 3)],
81: (8)              [[[1, 1), (3, 1)], (3, 1)],
82: (8)              [[[1, 0), (0, 0)], (0, 0)],
83: (8)              [[[0, 1), (0, 0)], (0, 0)],
84: (8)              [[[1, 0), (0, 1)], (0, 0)],
85: (8)              [[[1, 1), (0, 0)], (0, 0)],
86: (8)              [[[1, 1), (1, 0)], (1, 0)],
87: (8)              [[[1, 1), (0, 1)], (0, 1)],
88: (4)
89: (4)      ]
90: (8)      for input_shapes, expected_shape in data:
91: (8)          assert_shapes_correct(input_shapes, expected_shape)
92: (8)          assert_shapes_correct(input_shapes[::-1], expected_shape)
93: (0)      def test_two_compatible_by_prepending_ones_input_shapes():
94: (4)          data = [
95: (8)              [[(), (3,)], (3,)],
96: (8)              [[(3,), (3, 3)], (3, 3)],
97: (8)              [[(3,), (3, 1)], (3, 3)],
98: (8)              [[(1,), (3, 3)], (3, 3)],
99: (8)              [[((), (3, 3)], (3, 3)],
100: (8)              [[[1, 1), (3, )], (1, 3)],
101: (8)              [[[1, 1), (3, 1)], (3, 1)],
102: (8)              [[[1, 1), (1, 3)], (1, 3)],
103: (8)              [[[(), (1, 3)], (1, 3)],
104: (8)              [[[(), (3, 1)], (3, 1)],
105: (8)              [[[(), (0, )], (0, )]],
106: (8)              [[[0, ), (0, 0)], (0, 0)],
107: (8)              [[[0, ), (0, 1)], (0, 0)],
108: (8)              [[[1, ), (0, 0)], (0, 0)],
109: (8)              [[[(), (0, 0)], (0, 0)],
110: (8)              [[[1, 1), (0, )], (1, 0)],
111: (8)              [[[1, ), (0, 1)], (0, 1)],
112: (8)              [[[1, ), (1, 0)], (1, 0)],
113: (8)              [[[(), (1, 0)], (1, 0)],
114: (8)              [[[(), (0, 1)], (0, 1)],
115: (4)
116: (8)      ]
117: (8)      for input_shapes, expected_shape in data:
118: (8)          assert_shapes_correct(input_shapes, expected_shape)
119: (8)          assert_shapes_correct(input_shapes[::-1], expected_shape)
120: (0)      def test_incompatible_shapes_raise_valueerror():
121: (4)          data = [
122: (8)              [(3,), (4,)],
123: (8)              [(2, 3), (2,)],
124: (8)              [(3,), (3,), (4,)],
125: (8)              [(1, 3, 4), (2, 3, 3)]]
```

```

124: (4) ]
125: (4)     for input_shapes in data:
126: (8)         assert_incompatible_shapes_raise(input_shapes)
127: (8)         assert_incompatible_shapes_raise(input_shapes[::-1])
128: (0) def test_same_as_ufunc():
129: (4)     data = [
130: (8)         [[(1,), (3,)], (3,)],
131: (8)         [[(1, 3), (3, 3)], (3, 3)],
132: (8)         [[(3, 1), (3, 3)], (3, 3)],
133: (8)         [[(1, 3), (3, 1)], (3, 3)],
134: (8)         [[(1, 1), (3, 3)], (3, 3)],
135: (8)         [[(1, 1), (1, 3)], (1, 3)],
136: (8)         [[(1, 1), (3, 1)], (3, 1)],
137: (8)         [[(1, 0), (0, 0)], (0, 0)],
138: (8)         [[(0, 1), (0, 0)], (0, 0)],
139: (8)         [[(1, 0), (0, 1)], (0, 0)],
140: (8)         [[(1, 1), (0, 0)], (0, 0)],
141: (8)         [[(1, 1), (1, 0)], (1, 0)],
142: (8)         [[(1, 1), (0, 1)], (0, 1)],
143: (8)         [[[(), (3,)], (3,)],
144: (8)         [[(3,), (3, 3)], (3, 3)],
145: (8)         [[(3,), (3, 1)], (3, 3)],
146: (8)         [[(1,), (3, 3)], (3, 3)],
147: (8)         [[((), (3, 3)], (3, 3)],
148: (8)         [[(1, 1), (3,)], (1, 3)],
149: (8)         [[(1,), (3, 1)], (3, 1)],
150: (8)         [[(1,), (1, 3)], (1, 3)],
151: (8)         [[((), (1, 3)], (1, 3)],
152: (8)         [[((), (3, 1)], (3, 1)],
153: (8)         [[((), (0,)], (0,)],
154: (8)         [[(0,), (0, 0)], (0, 0)],
155: (8)         [[(0,), (0, 1)], (0, 0)],
156: (8)         [[(1,), (0, 0)], (0, 0)],
157: (8)         [[((), (0, 0)], (0, 0)],
158: (8)         [[(1, 1), (0,)], (1, 0)],
159: (8)         [[(1,), (0, 1)], (0, 1)],
160: (8)         [[(1,), (1, 0)], (1, 0)],
161: (8)         [[((), (1, 0)], (1, 0)],
162: (8)         [[((), (0, 1)], (0, 1)],
163: (4)     ],
164: (4) for input_shapes, expected_shape in data:
165: (8)     assert_same_as_ufunc(input_shapes[0], input_shapes[1],
166: (29)             "Shapes: %s %s" % (input_shapes[0],
input_shapes[1]))
167: (8)     assert_same_as_ufunc(input_shapes[1], input_shapes[0])
168: (8)     assert_same_as_ufunc(input_shapes[0], input_shapes[1], True)
169: (8)     if () not in input_shapes:
170: (12)         assert_same_as_ufunc(input_shapes[0], input_shapes[1], False,
True)
171: (12)         assert_same_as_ufunc(input_shapes[0], input_shapes[1], True, True)
172: (0) def test_broadcast_to_succeeds():
173: (4)     data = [
174: (8)         [np.array(0), (0,), np.array(0)],
175: (8)         [np.array(0), (1,), np.zeros(1)],
176: (8)         [np.array(0), (3,), np.zeros(3)],
177: (8)         [np.ones(1), (1,), np.ones(1)],
178: (8)         [np.ones(1), (2,), np.ones(2)],
179: (8)         [np.ones(1), (1, 2, 3), np.ones((1, 2, 3))],
180: (8)         [np.arange(3), (3,), np.arange(3)],
181: (8)         [np.arange(3), (1, 3), np.arange(3).reshape(1, -1)],
182: (8)         [np.arange(3), (2, 3), np.array([[0, 1, 2], [0, 1, 2]])],
183: (8)         [np.ones(0), 0, np.ones(0)],
184: (8)         [np.ones(1), 1, np.ones(1)],
185: (8)         [np.ones(1), 2, np.ones(2)],
186: (8)         [np.ones(1), (0,), np.ones(0)],
187: (8)         [np.ones((1, 2)), (0, 2), np.ones((0, 2))],
188: (8)         [np.ones((2, 1)), (2, 0), np.ones((2, 0))],
189: (4)     ],
190: (4)     for input_array, shape, expected in data:

```

```

191: (8)             actual = broadcast_to(input_array, shape)
192: (8)             assert_array_equal(expected, actual)
193: (0) def test_broadcast_to_raises():
194: (4)     data = [
195: (8)         [(0,), ()],
196: (8)         [(1,), ()],
197: (8)         [(3,), ()],
198: (8)         [(3,), (1,)],
199: (8)         [(3,), (2,)],
200: (8)         [(3,), (4,)],
201: (8)         [(1, 2), (2, 1)],
202: (8)         [(1, 1), (1,)],
203: (8)         [(1,), -1],
204: (8)         [(1,), (-1,)],
205: (8)         [(1, 2), (-1, 2)],
206: (4)
207: (4)     for orig_shape, target_shape in data:
208: (8)         arr = np.zeros(orig_shape)
209: (8)         assert_raises(ValueError, lambda: broadcast_to(arr, target_shape))
210: (0) def test_broadcast_shape():
211: (4)     assert_equal(_broadcast_shape(), ())
212: (4)     assert_equal(_broadcast_shape([1, 2]), (2,))
213: (4)     assert_equal(_broadcast_shape(np.ones((1, 1))), (1, 1))
214: (4)     assert_equal(_broadcast_shape(np.ones((1, 1)), np.ones((3, 4))), (3, 4))
215: (4)     assert_equal(_broadcast_shape(*([np.ones((1, 2))] * 32)), (1, 2))
216: (4)     assert_equal(_broadcast_shape(*([np.ones((1, 2))] * 100)), (1, 2))
217: (4)     assert_equal(_broadcast_shape(*([np.ones(2)] * 32 + [1])), (2,))
218: (4)     bad_args = [np.ones(2)] * 32 + [np.ones(3)] * 32
219: (4)     assert_raises(ValueError, lambda: _broadcast_shape(*bad_args))
220: (0) def test_broadcast_shapes_succeeds():
221: (4)     data = [
222: (8)         [[], ()],
223: (8)         [[(), ()]],
224: (8)         [[[7, ()], (7, ())],
225: (8)         [[[1, 2), (2, )], (1, 2)],
226: (8)         [[[1, 1)], (1, 1)],
227: (8)         [[[1, 1), (3, 4)], (3, 4)],
228: (8)         [[[6, 7), (5, 6, 1), (7, ), (5, 1, 7)], (5, 6, 7)],
229: (8)         [[[5, 6, 1)], (5, 6, 1)],
230: (8)         [[[1, 3), (3, 1)], (3, 3)],
231: (8)         [[[1, 0), (0, 0)], (0, 0)],
232: (8)         [[[0, 1), (0, 0)], (0, 0)],
233: (8)         [[[1, 0), (0, 1)], (0, 0)],
234: (8)         [[[1, 1), (0, 0)], (0, 0)],
235: (8)         [[[1, 1), (1, 0)], (1, 0)],
236: (8)         [[[1, 1), (0, 1)], (0, 1)],
237: (8)         [[[(), (0, )], (0, )]],
238: (8)         [[[0, 0), (0, 0)], (0, 0)],
239: (8)         [[[0, 0), (0, 1)], (0, 0)],
240: (8)         [[[1, 0), (0, 0)], (0, 0)],
241: (8)         [[[(), (0, 0)], (0, 0)],
242: (8)         [[[1, 1), (0, )], (1, 0)],
243: (8)         [[[1, 0), (0, 1)], (0, 1)],
244: (8)         [[[1, 0), (1, 0)], (1, 0)],
245: (8)         [[[(), (1, 0)], (1, 0)],
246: (8)         [[[(), (0, 1)], (0, 1)],
247: (8)         [[[1, 0), (3, )], (3, )],
248: (8)         [[[2, (3, 2)], (3, 2)],
249: (4)
250: (4)     for input_shapes, target_shape in data:
251: (8)         assert_equal(broadcast_shapes(*input_shapes), target_shape)
252: (4)         assert_equal(broadcast_shapes(*([1, 2]) * 32), (1, 2))
253: (4)         assert_equal(broadcast_shapes(*([1, 2]) * 100), (1, 2))
254: (4)         assert_equal(broadcast_shapes(*([(2, )] * 32)), (2,))
255: (0) def test_broadcast_shapes_raises():
256: (4)     data = [
257: (8)         [(3,), (4,)],
258: (8)         [(2, 3), (2, )],
259: (8)         [(3,), (3,), (4,)],

```

```

260: (8)          [(1, 3, 4), (2, 3, 3)],
261: (8)          [(1, 2), (3,1), (3,2), (10, 5)],
262: (8)          [2, (2, 3)],
263: (4)
264: (4)          for input_shapes in data:
265: (8)              assert_raises(ValueError, lambda: broadcast_shapes(*input_shapes))
266: (4)          bad_args = [(2,)] * 32 + [(3,)] * 32
267: (4)          assert_raises(ValueError, lambda: broadcast_shapes(*bad_args))
268: (0)          def test_as_strided():
269: (4)              a = np.array([None])
270: (4)              a_view = as_strided(a)
271: (4)              expected = np.array([None])
272: (4)              assert_array_equal(a_view, np.array([None]))
273: (4)              a = np.array([1, 2, 3, 4])
274: (4)              a_view = as_strided(a, shape=(2,), strides=(2 * a.itemsize,))
275: (4)              expected = np.array([1, 3])
276: (4)              assert_array_equal(a_view, expected)
277: (4)              a = np.array([1, 2, 3, 4])
278: (4)              a_view = as_strided(a, shape=(3, 4), strides=(0, 1 * a.itemsize))
279: (4)              expected = np.array([[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]])
280: (4)              assert_array_equal(a_view, expected)
281: (4)              dt = np.dtype([('num', 'i4'), ('obj', 'O')])
282: (4)              a = np.empty((4,), dtype=dt)
283: (4)              a['num'] = np.arange(1, 5)
284: (4)              a_view = as_strided(a, shape=(3, 4), strides=(0, a.itemsize))
285: (4)              expected_num = [[1, 2, 3, 4]] * 3
286: (4)              expected_obj = [[None]*4]*3
287: (4)              assert_equal(a_view.dtype, dt)
288: (4)              assert_array_equal(expected_num, a_view['num'])
289: (4)              assert_array_equal(expected_obj, a_view['obj'])
290: (4)              a = np.empty((4,), dtype='V4')
291: (4)              a_view = as_strided(a, shape=(3, 4), strides=(0, a.itemsize))
292: (4)              assert_equal(a.dtype, a_view.dtype)
293: (4)              dt = np.dtype({'names': [''], 'formats': ['V4']})
294: (4)              a = np.empty((4,), dtype=dt)
295: (4)              a_view = as_strided(a, shape=(3, 4), strides=(0, a.itemsize))
296: (4)              assert_equal(a.dtype, a_view.dtype)
297: (4)              r = [rational(i) for i in range(4)]
298: (4)              a = np.array(r, dtype=rational)
299: (4)              a_view = as_strided(a, shape=(3, 4), strides=(0, a.itemsize))
300: (4)              assert_equal(a.dtype, a_view.dtype)
301: (4)              assert_array_equal([r] * 3, a_view)
302: (0)          class TestSlidingWindowView:
303: (4)              def test_1d(self):
304: (8)                  arr = np.arange(5)
305: (8)                  arr_view = sliding_window_view(arr, 2)
306: (8)                  expected = np.array([[0, 1],
307: (29)                      [1, 2],
308: (29)                      [2, 3],
309: (29)                      [3, 4]])
310: (8)                  assert_array_equal(arr_view, expected)
311: (4)              def test_2d(self):
312: (8)                  i, j = np.ogrid[:3, :4]
313: (8)                  arr = 10*i + j
314: (8)                  shape = (2, 2)
315: (8)                  arr_view = sliding_window_view(arr, shape)
316: (8)                  expected = np.array([[[[0, 1], [10, 11]],
317: (30)                      [[[1, 2], [11, 12]],
318: (30)                          [[2, 3], [12, 13]]],
319: (29)                          [[[10, 11], [20, 21]],
320: (30)                              [[11, 12], [21, 22]],
321: (30)                              [[12, 13], [22, 23]]]])
322: (8)                  assert_array_equal(arr_view, expected)
323: (4)              def test_2d_with_axis(self):
324: (8)                  i, j = np.ogrid[:3, :4]
325: (8)                  arr = 10*i + j
326: (8)                  arr_view = sliding_window_view(arr, 3, 0)
327: (8)                  expected = np.array([[0, 10, 20],
328: (30)                      [1, 11, 21],

```

```

329: (30) [2, 12, 22],
330: (30) [3, 13, 23]]])
331: (8) assert_array_equal(arr_view, expected)
332: (4) def test_2d_repeated_axis(self):
333: (8) i, j = np.ogrid[:3, :4]
334: (8) arr = 10*i + j
335: (8) arr_view = sliding_window_view(arr, (2, 3), (1, 1))
336: (8) expected = np.array([[[[0, 1, 2],
337: (31) [1, 2, 3]]], [[[10, 11, 12],
338: (29) [11, 12, 13]]], [[[20, 21, 22],
339: (31) [21, 22, 23]]]])
340: (29) assert_array_equal(arr_view, expected)
341: (31) def test_2d_without_axis(self):
342: (8) i, j = np.ogrid[:4, :4]
343: (4) arr = 10*i + j
344: (8) shape = (2, 3)
345: (8) arr_view = sliding_window_view(arr, shape)
346: (8) expected = np.array([[[[0, 1, 2], [10, 11, 12],
347: (30) [[1, 2, 3], [11, 12, 13]]], [[[10, 11, 12], [20, 21, 22]],
348: (29) [[11, 12, 13], [21, 22, 23]]], [[[20, 21, 22], [30, 31, 32]],
349: (30) [[21, 22, 23], [31, 32, 33]]]])
350: (29) assert_array_equal(arr_view, expected)
351: (30) def test_errors(self):
352: (29) i, j = np.ogrid[:4, :4]
353: (30) arr = 10*i + j
354: (8) with pytest.raises(ValueError, match='cannot contain negative
values'):
355: (4) 356: (8) sliding_window_view(arr, (-1, 3))
357: (8) with pytest.raises(
358: (8) ValueError,
359: (12) match='must provide window_shape for all dimensions of `x`'):
360: (8) 361: (16) sliding_window_view(arr, (1,))
362: (16) 363: (12) with pytest.raises(
364: (8) ValueError,
365: (16) match='Must provide matching length window_shape and axis'):
366: (16) 367: (12) sliding_window_view(arr, (1, 3, 4), axis=(0, 1))
368: (8) with pytest.raises(
369: (16) ValueError,
370: (16) match='window shape cannot be larger than input array'):
371: (12) 372: (4) sliding_window_view(arr, (5, 5))
373: (8) def test_writeable(self):
374: (8) arr = np.arange(5)
375: (8) view = sliding_window_view(arr, 2, writeable=False)
376: (8) assert_(not view.flags.writeable)
377: (16) with pytest.raises(
378: (16) ValueError,
379: (12) match='assignment destination is read-only'):
380: (8) view[0, 0] = 3
381: (8) view = sliding_window_view(arr, 2, writeable=True)
382: (8) assert_(view.flags.writeable)
383: (8) view[0, 1] = 3
384: (4) assert_array_equal(arr, np.array([0, 3, 2, 3, 4]))
385: (8) def test_subok(self):
386: (12) class MyArray(np.ndarray):
387: (8) pass
388: (8) arr = np.arange(5).view(MyArray)
389: (51) assert_(not isinstance(sliding_window_view(arr, 2,
390: (31) subok=False),
391: (8) MyArray))
392: (8) assert_(isinstance(sliding_window_view(arr, 2, subok=True), MyArray))
393: (0) assert_(not isinstance(sliding_window_view(arr, 2), MyArray))
394: (4) def as_strided_writeable():
395: (4) arr = np.ones(10)
396: (4) view = as_strided(arr, writeable=False)

```

```

397: (4)           view = as_strided(arr, writeable=True)
398: (4)           assert_(view.flags.writeable)
399: (4)           view[...] = 3
400: (4)           assert_array_equal(arr, np.full_like(arr, 3))
401: (4)           arr.flags.writeable = False
402: (4)           view = as_strided(arr, writeable=False)
403: (4)           view = as_strided(arr, writeable=True)
404: (4)           assert_(not view.flags.writeable)
405: (0)           class VerySimpleSubClass(np.ndarray):
406: (4)             def __new__(cls, *args, **kwargs):
407: (8)               return np.array(*args, subok=True, **kwargs).view(cls)
408: (0)           class SimpleSubClass(VerySimpleSubClass):
409: (4)             def __new__(cls, *args, **kwargs):
410: (8)               self = np.array(*args, subok=True, **kwargs).view(cls)
411: (8)               self.info = 'simple'
412: (8)               return self
413: (4)             def __array_finalize__(self, obj):
414: (8)               self.info = getattr(obj, 'info', '') + ' finalized'
415: (0)           def test_subclasses():
416: (4)             a = VerySimpleSubClass([1, 2, 3, 4])
417: (4)             assert_(type(a) is VerySimpleSubClass)
418: (4)             a_view = as_strided(a, shape=(2,), strides=(2 * a.itemsize,))
419: (4)             assert_(type(a_view) is np.ndarray)
420: (4)             a_view = as_strided(a, shape=(2,), strides=(2 * a.itemsize,), subok=True)
421: (4)             assert_(type(a_view) is VerySimpleSubClass)
422: (4)             a = SimpleSubClass([1, 2, 3, 4])
423: (4)             a_view = as_strided(a, shape=(2,), strides=(2 * a.itemsize,), subok=True)
424: (4)             assert_(type(a_view) is SimpleSubClass)
425: (4)             assert_(a_view.info == 'simple finalized')
426: (4)             b = np.arange(len(a)).reshape(-1, 1)
427: (4)             a_view, b_view = broadcast_arrays(a, b)
428: (4)             assert_(type(a_view) is np.ndarray)
429: (4)             assert_(type(b_view) is np.ndarray)
430: (4)             assert_(a_view.shape == b_view.shape)
431: (4)             a_view, b_view = broadcast_arrays(a, b, subok=True)
432: (4)             assert_(type(a_view) is SimpleSubClass)
433: (4)             assert_(a_view.info == 'simple finalized')
434: (4)             assert_(type(b_view) is np.ndarray)
435: (4)             assert_(a_view.shape == b_view.shape)
436: (4)             shape = (2, 4)
437: (4)             a_view = broadcast_to(a, shape)
438: (4)             assert_(type(a_view) is np.ndarray)
439: (4)             assert_(a_view.shape == shape)
440: (4)             a_view = broadcast_to(a, shape, subok=True)
441: (4)             assert_(type(a_view) is SimpleSubClass)
442: (4)             assert_(a_view.info == 'simple finalized')
443: (4)             assert_(a_view.shape == shape)
444: (0)           def test_writeable():
445: (4)             original = np.array([1, 2, 3])
446: (4)             result = broadcast_to(original, (2, 3))
447: (4)             assert_equal(result.flags.writeable, False)
448: (4)             assert_raises(ValueError, result.__setitem__, slice(None), 0)
449: (4)             for is_broadcast, results in [(False, broadcast_arrays(original,)),
450: (34)                           (True, broadcast_arrays(0, original))]:
451: (8)               for result in results:
452: (12)                 if is_broadcast:
453: (16)                   with assert_warns(FutureWarning):
454: (20)                     assert_equal(result.flags.writeable, True)
455: (16)                   with assert_warns(DeprecationWarning):
456: (20)                     result[:] = 0
457: (16)                     assert_equal(result.flags.writeable, True)
458: (12)                 else:
459: (16)                     assert_equal(result.flags.writeable, True)
460: (4)             for results in [broadcast_arrays(original),
461: (20)                           broadcast_arrays(0, original)]:
462: (8)               for result in results:
463: (12)                 result.flags.writeable = True
464: (12)                 assert_equal(result.flags.writeable, True)
465: (12)                 result[:] = 0

```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

466: (4)             original.flags.writeable = False
467: (4)             _, result = broadcast_arrays(0, original)
468: (4)             assert_equal(result.flags.writeable, False)
469: (4)             shape = (2,)
470: (4)             strides = [0]
471: (4)             tricky_array = as_strided(np.array(0), shape, strides)
472: (4)             other = np.zeros((1,))
473: (4)             first, second = broadcast_arrays(tricky_array, other)
474: (4)             assert_(first.shape == second.shape)
475: (0)             def test_writeable_memoryview():
476: (4)                 original = np.array([1, 2, 3])
477: (4)                 for is_broadcast, results in [(False, broadcast_arrays(original,)),
478: (34)                               (True, broadcast_arrays(0, original))]:
479: (8)                     for result in results:
480: (12)                         if is_broadcast:
481: (16)                             assert memoryview(result).readonly
482: (12)                         else:
483: (16)                             assert not memoryview(result).readonly
484: (0)             def test_reference_types():
485: (4)                 input_array = np.array('a', dtype=object)
486: (4)                 expected = np.array(['a'] * 3, dtype=object)
487: (4)                 actual = broadcast_to(input_array, (3,))
488: (4)                 assert_array_equal(expected, actual)
489: (4)                 actual, _ = broadcast_arrays(input_array, np.ones(3))
490: (4)                 assert_array_equal(expected, actual)
```

-----  
File 247 - test\_twodim\_base.py:

```
1: (0)             """Test functions for matrix module
2: (0)             """
3: (0)             from numpy.testing import (
4: (4)                 assert_equal, assert_array_equal, assert_array_max_ulp,
5: (4)                 assert_array_almost_equal, assert_raises, assert_
6: (0)
7: (0)             )
8: (4)             from numpy import (
9: (4)                 arange, add, fliplr, flipud, zeros, ones, eye, array, diag, histogram2d,
10: (4)                tri, mask_indices, triu_indices, triu_indices_from, tril_indices,
11: (4)                tril_indices_from, vander,
12: (0)
13: (0)
14: (0)             import numpy as np
15: (0)             import pytest
16: (0)             def get_mat(n):
17: (4)                 data = arange(n)
18: (4)                 data = add.outer(data, data)
19: (0)             return data
20: (0)             class TestEye:
21: (4)                 def test_basic(self):
22: (8)                     assert_equal(eye(4),
23: (21)                         array([[1, 0, 0, 0],
24: (28)                           [0, 1, 0, 0],
25: (28)                           [0, 0, 1, 0],
26: (28)                           [0, 0, 0, 1]]))
27: (8)                     assert_equal(eye(4, dtype='f'),
28: (21)                         array([[1, 0, 0, 0],
29: (28)                           [0, 1, 0, 0],
30: (28)                           [0, 0, 1, 0],
31: (28)                           [0, 0, 0, 1]], 'f'))
32: (8)                     assert_equal(eye(3) == 1,
33: (21)                         eye(3, dtype=bool))
34: (0)             def test_uint64(self):
35: (8)                 assert_equal(eye(np.uint64(2), dtype=int), array([[1, 0], [0, 1]]))
36: (8)                 assert_equal(eye(np.uint64(2), M=np.uint64(4), k=np.uint64(1)),
37: (21)                         array([[0, 1, 0, 0], [0, 0, 1, 0]]))
38: (4)             def test_diag(self):
39: (8)                 assert_equal(eye(4, k=1),
40: (21)                         array([[0, 1, 0, 0],
41: (28)                           [0, 0, 1, 0],
```

```

40: (28)                                [0, 0, 0, 1],
41: (28)                                [0, 0, 0, 0]]))
42: (8)         assert_equal(eye(4, k=-1),
43: (21)                         array([[0, 0, 0, 0],
44: (28)                             [1, 0, 0, 0],
45: (28)                             [0, 1, 0, 0],
46: (28)                             [0, 0, 1, 0]]))
47: (4)         def test_2d(self):
48: (8)             assert_equal(eye(4, 3),
49: (21)                 array([[1, 0, 0],
50: (28)                     [0, 1, 0],
51: (28)                     [0, 0, 1],
52: (28)                     [0, 0, 0]]))
53: (8)             assert_equal(eye(3, 4),
54: (21)                 array([[1, 0, 0, 0],
55: (28)                     [0, 1, 0, 0],
56: (28)                     [0, 0, 1, 0]]))
57: (4)         def test_diag2d(self):
58: (8)             assert_equal(eye(3, 4, k=2),
59: (21)                 array([[0, 0, 1, 0],
60: (28)                     [0, 0, 0, 1],
61: (28)                     [0, 0, 0, 0]]))
62: (8)             assert_equal(eye(4, 3, k=-2),
63: (21)                 array([[0, 0, 0],
64: (28)                     [0, 0, 0],
65: (28)                     [1, 0, 0],
66: (28)                     [0, 1, 0]]))
67: (4)         def test_eye_bounds(self):
68: (8)             assert_equal(eye(2, 2, 1), [[0, 1], [0, 0]])
69: (8)             assert_equal(eye(2, 2, -1), [[0, 0], [1, 0]])
70: (8)             assert_equal(eye(2, 2, 2), [[0, 0], [0, 0]])
71: (8)             assert_equal(eye(2, 2, -2), [[0, 0], [0, 0]])
72: (8)             assert_equal(eye(3, 2, 2), [[0, 0], [0, 0], [0, 0]])
73: (8)             assert_equal(eye(3, 2, 1), [[0, 1], [0, 0], [0, 0]])
74: (8)             assert_equal(eye(3, 2, -1), [[0, 0], [1, 0], [0, 1]])
75: (8)             assert_equal(eye(3, 2, -2), [[0, 0], [0, 0], [1, 0]])
76: (8)             assert_equal(eye(3, 2, -3), [[0, 0], [0, 0], [0, 0]])
77: (4)         def test_strings(self):
78: (8)             assert_equal(eye(2, 2, dtype='S3'),
79: (21)                 [[b'1', b''], [b'', b'1']])
80: (4)         def test_bool(self):
81: (8)             assert_equal(eye(2, 2, dtype=bool), [[True, False], [False, True]])
82: (4)         def test_order(self):
83: (8)             mat_c = eye(4, 3, k=-1)
84: (8)             mat_f = eye(4, 3, k=-1, order='F')
85: (8)             assert_equal(mat_c, mat_f)
86: (8)             assert mat_c.flags.c_contiguous
87: (8)             assert not mat_c.flags.f_contiguous
88: (8)             assert not mat_f.flags.c_contiguous
89: (8)             assert mat_f.flags.f_contiguous
90: (0)         class TestDiag:
91: (4)             def test_vector(self):
92: (8)                 vals = (100 * arange(5)).astype('l')
93: (8)                 b = zeros((5, 5))
94: (8)                 for k in range(5):
95: (12)                     b[k, k] = vals[k]
96: (8)                 assert_equal(diag(vals), b)
97: (8)                 b = zeros((7, 7))
98: (8)                 c = b.copy()
99: (8)                 for k in range(5):
100: (12)                     b[k, k + 2] = vals[k]
101: (12)                     c[k + 2, k] = vals[k]
102: (8)                 assert_equal(diag(vals, k=2), b)
103: (8)                 assert_equal(diag(vals, k=-2), c)
104: (4)             def test_matrix(self, vals=None):
105: (8)                 if vals is None:
106: (12)                     vals = (100 * get_mat(5) + 1).astype('l')
107: (8)                     b = zeros((5,))
108: (8)                     for k in range(5):

```

```

109: (12)             b[k] = vals[k, k]
110: (8)              assert_equal(diag(vals), b)
111: (8)              b = b * 0
112: (8)              for k in range(3):
113: (12)                b[k] = vals[k, k + 2]
114: (8)              assert_equal(diag(vals, 2), b[:3])
115: (8)              for k in range(3):
116: (12)                b[k] = vals[k + 2, k]
117: (8)              assert_equal(diag(vals, -2), b[:3])
118: (4)               def test_fortran_order(self):
119: (8)                 vals = array((100 * get_mat(5) + 1), order='F', dtype='l')
120: (8)                 self.test_matrix(vals)
121: (4)               def test_diag_bounds(self):
122: (8)                 A = [[1, 2], [3, 4], [5, 6]]
123: (8)                 assert_equal(diag(A, k=2), [])
124: (8)                 assert_equal(diag(A, k=1), [2])
125: (8)                 assert_equal(diag(A, k=0), [1, 4])
126: (8)                 assert_equal(diag(A, k=-1), [3, 6])
127: (8)                 assert_equal(diag(A, k=-2), [5])
128: (8)                 assert_equal(diag(A, k=-3), [])
129: (4)               def test_failure(self):
130: (8)                 assert_raises(ValueError, diag, [[[1]]])
131: (0)               class TestFliplr:
132: (4)                 def test_basic(self):
133: (8)                   assert_raises(ValueError, fliplr, ones(4))
134: (8)                   a = get_mat(4)
135: (8)                   b = a[:, ::-1]
136: (8)                   assert_equal(fliplr(a), b)
137: (8)                   a = [[0, 1, 2],
138: (13)                     [3, 4, 5]]
139: (8)                   b = [[2, 1, 0],
140: (13)                     [5, 4, 3]]
141: (8)                   assert_equal(fliplr(a), b)
142: (0)               class TestFlipud:
143: (4)                 def test_basic(self):
144: (8)                   a = get_mat(4)
145: (8)                   b = a[::-1, :]
146: (8)                   assert_equal(flipud(a), b)
147: (8)                   a = [[0, 1, 2],
148: (13)                     [3, 4, 5]]
149: (8)                   b = [[3, 4, 5],
150: (13)                     [0, 1, 2]]
151: (8)                   assert_equal(flipud(a), b)
152: (0)               class TestHistogram2d:
153: (4)                 def test_simple(self):
154: (8)                   x = array(
155: (12)                     [0.41702200, 0.72032449, 1.1437481e-4, 0.302332573, 0.146755891])
156: (8)                   y = array(
157: (12)                     [0.09233859, 0.18626021, 0.34556073, 0.39676747, 0.53881673])
158: (8)                   xedges = np.linspace(0, 1, 10)
159: (8)                   yedges = np.linspace(0, 1, 10)
160: (8)                   H = histogram2d(x, y, (xedges, yedges))[0]
161: (8)                   answer = array(
162: (12)                     [[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
163: (13)                       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
164: (13)                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
165: (13)                       [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
166: (13)                       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
167: (13)                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
168: (13)                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
169: (13)                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
170: (13)                       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
171: (8)                   assert_array_equal(H.T, answer)
172: (8)                   H = histogram2d(x, y, xedges)[0]
173: (8)                   assert_array_equal(H.T, answer)
174: (8)                   H, xedges, yedges = histogram2d(list(range(10)), list(range(10)))
175: (8)                   assert_array_equal(H, eye(10, 10))
176: (8)                   assert_array_equal(xedges, np.linspace(0, 9, 11))
177: (8)                   assert_array_equal(yedges, np.linspace(0, 9, 11))

```

```

178: (4)
179: (8)
180: (8)
181: (8)
182: (12)
183: (8)
184: (12)
185: (13)
186: (13)
187: (13)
188: (13)
189: (13)
190: (8)
191: (8)
192: (8)
193: (4)
194: (8)
195: (8)
196: (8)
197: (12)
198: (8)
199: (24)
200: (24)
201: (8)
202: (4)
203: (8)
204: (8)
205: (8)
206: (4)
207: (8)
208: (8)
209: (8)
210: (8)
211: (4)
212: (8)
213: (12)
214: (13)
215: (8)
216: (12)
217: (13)
218: (8)
219: (8)
220: (8)
221: (12)
222: (13)
223: (13)
224: (13)
225: (13)
226: (13)
227: (13)
228: (13)
229: (13)
230: (13)
231: (8)
232: (8)
233: (8)
234: (8)
235: (12)
236: (13)
237: (13)
238: (13)
239: (8)
240: (8)
241: (4)
242: (8)
243: (12)
244: (16)
245: (8)
246: (8)

    def test_asym(self):
        x = array([1, 1, 2, 3, 4, 4, 4, 5])
        y = array([1, 3, 2, 0, 1, 2, 3, 4])
        H, xed, yed = histogram2d(
            x, y, (6, 5), range=[[0, 6], [0, 5]], density=True)
        answer = array([
            [0., 0, 0, 0, 0],
            [0, 1, 0, 1, 0],
            [0, 0, 1, 0, 0],
            [1, 0, 0, 0, 0],
            [0, 1, 1, 1, 0],
            [0, 0, 0, 0, 1]])
        assert_array_almost_equal(H, answer/8., 3)
        assert_array_equal(xed, np.linspace(0, 6, 7))
        assert_array_equal(yed, np.linspace(0, 5, 6))

    def test_density(self):
        x = array([1, 2, 3, 1, 2, 3, 1, 2, 3])
        y = array([1, 1, 1, 2, 2, 2, 3, 3, 3])
        H, xed, yed = histogram2d(
            x, y, [[1, 2, 3, 5], [1, 2, 3, 5]], density=True)
        answer = array([[1, 1, .5],
                      [1, 1, .5],
                      [.5, .5, .25]])/9.
        assert_array_almost_equal(H, answer, 3)

    def test_all_outliers(self):
        r = np.random.rand(100) + 1. + 1e6 # histogramdd rounds by decimal=6
        H, xed, yed = histogram2d(r, r, (4, 5), range=[[0, 1], [0, 1]])
        assert_array_equal(H, 0)

    def test_empty(self):
        a, edge1, edge2 = histogram2d([], [], bins=[[0, 1], [0, 1]])
        assert_array_max_ulp(a, array([[0.]]))
        a, edge1, edge2 = histogram2d([], [], bins=4)
        assert_array_max_ulp(a, np.zeros((4, 4)))

    def test_binparameter_combination(self):
        x = array([
            [0, 0.09207008, 0.64575234, 0.12875982, 0.47390599,
             0.59944483, 1])
        y = array([
            [0, 0.14344267, 0.48988575, 0.30558665, 0.44700682,
             0.15886423, 1])
        edges = (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1)
        H, xe, ye = histogram2d(x, y, (edges, 4))
        answer = array([
            [[2., 0., 0., 0.],
             [0., 1., 0., 0.],
             [0., 0., 0., 0.],
             [0., 0., 0., 0.],
             [0., 1., 0., 0.],
             [1., 0., 0., 0.],
             [0., 1., 0., 0.],
             [0., 0., 0., 0.],
             [0., 0., 0., 0.],
             [0., 0., 0., 1.]])
        assert_array_equal(H, answer)
        assert_array_equal(ye, array([0., 0.25, 0.5, 0.75, 1]))
        H, xe, ye = histogram2d(x, y, (4, edges))
        answer = array([
            [[1., 1., 0., 1., 0., 0., 0., 0., 0., 0.],
             [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
             [0., 1., 0., 0., 1., 0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
             [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
             [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
             [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
        assert_array_equal(H, answer)
        assert_array_equal(xe, array([0., 0.25, 0.5, 0.75, 1]))

    def test_dispatch(self):
        class ShouldDispatch:
            def __array_function__(self, function, types, args, kwargs):
                return types, args, kwargs
            xy = [1, 2]
            s_d = ShouldDispatch()

```

```

247: (8)             r = histogram2d(s_d, xy)
248: (8)             assert_(r == ((ShouldDispatch,), (s_d, xy), {}))
249: (8)             r = histogram2d(xy, s_d)
250: (8)             assert_(r == ((ShouldDispatch,), (xy, s_d), {}))
251: (8)             r = histogram2d(xy, xy, bins=s_d)
252: (8)             assert_(r, ((ShouldDispatch,), (xy, xy), dict(bins=s_d)))
253: (8)             r = histogram2d(xy, xy, bins=[s_d, 5])
254: (8)             assert_(r, ((ShouldDispatch,), (xy, xy), dict(bins=[s_d, 5])))
255: (8)             assert_raises(Exception, histogram2d, xy, xy, bins=[s_d])
256: (8)             r = histogram2d(xy, xy, weights=s_d)
257: (8)             assert_(r, ((ShouldDispatch,), (xy, xy), dict(weights=s_d)))
258: (4) @pytest.mark.parametrize(("x_len", "y_len"), [(10, 11), (20, 19)])
259: (4) def test_bad_length(self, x_len, y_len):
260: (8)     x, y = np.ones(x_len), np.ones(y_len)
261: (8)     with pytest.raises(ValueError,
262: (27)             match='x and y must have the same length.'):
263: (12)         histogram2d(x, y)
264: (0) class TestTri:
265: (4)     def test_dtype(self):
266: (8)         out = array([[1, 0, 0],
267: (21)                 [1, 1, 0],
268: (21)                 [1, 1, 1]])
269: (8)         assert_array_equal(tri(3), out)
270: (8)         assert_array_equal(tri(3, dtype=bool), out.astype(bool))
271: (0)     def test_tril_triu_ndim2():
272: (4)         for dtype in np.typecodes['AllFloat'] + np.typecodes['AllInteger']:
273: (8)             a = np.ones((2, 2), dtype=dtype)
274: (8)             b = np.tril(a)
275: (8)             c = np.triu(a)
276: (8)             assert_array_equal(b, [[1, 0], [1, 1]])
277: (8)             assert_array_equal(c, b.T)
278: (8)             assert_equal(b.dtype, a.dtype)
279: (8)             assert_equal(c.dtype, a.dtype)
280: (0)     def test_tril_triu_ndim3():
281: (4)         for dtype in np.typecodes['AllFloat'] + np.typecodes['AllInteger']:
282: (8)             a = np.array([
283: (12)                 [[1, 1], [1, 1]],
284: (12)                 [[1, 1], [1, 0]],
285: (12)                 [[1, 1], [0, 0]],
286: (12)                 ], dtype=dtype)
287: (8)             a_tril_desired = np.array([
288: (12)                 [[1, 0], [1, 1]],
289: (12)                 [[1, 0], [1, 0]],
290: (12)                 [[1, 0], [0, 0]],
291: (12)                 ], dtype=dtype)
292: (8)             a_triu_desired = np.array([
293: (12)                 [[1, 1], [0, 1]],
294: (12)                 [[1, 1], [0, 0]],
295: (12)                 [[1, 1], [0, 0]],
296: (12)                 ], dtype=dtype)
297: (8)             a_triu_observed = np.triu(a)
298: (8)             a_tril_observed = np.tril(a)
299: (8)             assert_array_equal(a_triu_observed, a_triu_desired)
300: (8)             assert_array_equal(a_tril_observed, a_tril_desired)
301: (8)             assert_equal(a_triu_observed.dtype, a.dtype)
302: (8)             assert_equal(a_tril_observed.dtype, a.dtype)
303: (0)     def test_tril_triu_with_inf():
304: (4)         arr = np.array([[1, 1, np.inf],
305: (20)                     [1, 1, 1],
306: (20)                     [np.inf, 1, 1]])
307: (4)         out_tril = np.array([[1, 0, 0],
308: (25)                     [1, 1, 0],
309: (25)                     [np.inf, 1, 1]])
310: (4)         out_triu = out_tril.T
311: (4)         assert_array_equal(np.triu(arr), out_triu)
312: (4)         assert_array_equal(np.tril(arr), out_tril)
313: (0)     def test_tril_triu_dtype():
314: (4)         for c in np.typecodes['All']:
315: (8)             if c == 'V':

```

```

316: (12)           continue
317: (8)            arr = np.zeros((3, 3), dtype=c)
318: (8)            assert_equal(np.triu(arr).dtype, arr.dtype)
319: (8)            assert_equal(np.tril(arr).dtype, arr.dtype)
320: (4)            arr = np.array(['2001-01-01T12:00', '2002-02-03T13:56'],
321: (20)           ['2004-01-01T12:00', '2003-01-03T13:45']],
322: (19)           dtype='datetime64')
323: (4)            assert_equal(np.triu(arr).dtype, arr.dtype)
324: (4)            assert_equal(np.tril(arr).dtype, arr.dtype)
325: (4)            arr = np.zeros((3, 3), dtype='f4,f4')
326: (4)            assert_equal(np.triu(arr).dtype, arr.dtype)
327: (4)            assert_equal(np.tril(arr).dtype, arr.dtype)
328: (0)             def test_mask_indices():
329: (4)               iu = mask_indices(3, np.triu)
330: (4)               a = np.arange(9).reshape(3, 3)
331: (4)               assert_array_equal(a[iu], array([0, 1, 2, 4, 5, 8]))
332: (4)               iu1 = mask_indices(3, np.triu, 1)
333: (4)               assert_array_equal(a[iu1], array([1, 2, 5]))
334: (0)             def test_tril_indices():
335: (4)               il1 = tril_indices(4)
336: (4)               il2 = tril_indices(4, k=2)
337: (4)               il3 = tril_indices(4, m=5)
338: (4)               il4 = tril_indices(4, k=2, m=5)
339: (4)               a = np.array([[1, 2, 3, 4],
340: (18)                 [5, 6, 7, 8],
341: (18)                 [9, 10, 11, 12],
342: (18)                 [13, 14, 15, 16]])
343: (4)               b = np.arange(1, 21).reshape(4, 5)
344: (4)               assert_array_equal(a[il1],
345: (23)                 array([1, 5, 6, 9, 10, 11, 13, 14, 15, 16]))
346: (4)               assert_array_equal(b[il3],
347: (23)                 array([1, 6, 7, 11, 12, 13, 16, 17, 18, 19]))
348: (4)               a[il1] = -1
349: (4)               assert_array_equal(a,
350: (23)                 array([[-1, 2, 3, 4],
351: (30)                   [-1, -1, 7, 8],
352: (30)                   [-1, -1, -1, 12],
353: (30)                   [-1, -1, -1, -1]]))
354: (4)               b[il3] = -1
355: (4)               assert_array_equal(b,
356: (23)                 array([[-1, 2, 3, 4, 5],
357: (30)                   [-1, -1, 8, 9, 10],
358: (30)                   [-1, -1, -1, 14, 15],
359: (30)                   [-1, -1, -1, -1, 20]]))
360: (4)               a[il2] = -10
361: (4)               assert_array_equal(a,
362: (23)                 array([[-10, -10, -10, 4],
363: (30)                   [-10, -10, -10, -10],
364: (30)                   [-10, -10, -10, -10],
365: (30)                   [-10, -10, -10, -10]]))
366: (4)               b[il4] = -10
367: (4)               assert_array_equal(b,
368: (23)                 array([[-10, -10, -10, 4, 5],
369: (30)                   [-10, -10, -10, -10, 10],
370: (30)                   [-10, -10, -10, -10, -10],
371: (30)                   [-10, -10, -10, -10, -10]]))
372: (0)             class TestTriuIndices:
373: (4)               def test_triu_indices(self):
374: (8)                 iu1 = triu_indices(4)
375: (8)                 iu2 = triu_indices(4, k=2)
376: (8)                 iu3 = triu_indices(4, m=5)
377: (8)                 iu4 = triu_indices(4, k=2, m=5)
378: (8)                 a = np.array([[1, 2, 3, 4],
379: (22)                   [5, 6, 7, 8],
380: (22)                   [9, 10, 11, 12],
381: (22)                   [13, 14, 15, 16]])
382: (8)                 b = np.arange(1, 21).reshape(4, 5)
383: (8)                 assert_array_equal(a[iu1],
384: (27)                   array([1, 2, 3, 4, 6, 7, 8, 11, 12, 16]))
```

```

385: (8)             assert_array_equal(b[iu3],
386: (27)                 array([1, 2, 3, 4, 5, 7, 8, 9,
387: (34)                             10, 13, 14, 15, 19, 20]))
388: (8)             a[iu1] = -1
389: (8)             assert_array_equal(a,
390: (27)                 array([[-1, -1, -1, -1],
391: (34)                     [5, -1, -1, -1],
392: (34)                     [9, 10, -1, -1],
393: (34)                     [13, 14, 15, -1]]))
394: (8)             b[iu3] = -1
395: (8)             assert_array_equal(b,
396: (27)                 array([-1, -1, -1, -1, -1],
397: (34)                     [6, -1, -1, -1, -1],
398: (34)                     [11, 12, -1, -1, -1],
399: (34)                     [16, 17, 18, -1, -1]]))
400: (8)             a[iu2] = -10
401: (8)             assert_array_equal(a,
402: (27)                 array([-1, -1, -10, -10],
403: (34)                     [5, -1, -1, -10],
404: (34)                     [9, 10, -1, -1],
405: (34)                     [13, 14, 15, -1]]))
406: (8)             b[iu4] = -10
407: (8)             assert_array_equal(b,
408: (27)                 array([-1, -1, -10, -10, -10],
409: (34)                     [6, -1, -1, -10, -10],
410: (34)                     [11, 12, -1, -1, -10],
411: (34)                     [16, 17, 18, -1, -1]]))
412: (0)             class TestTrilIndicesFrom:
413: (4)                 def test_exceptions(self):
414: (8)                     assert_raises(ValueError, tril_indices_from, np.ones((2,)))
415: (8)                     assert_raises(ValueError, tril_indices_from, np.ones((2, 2, 2)))
416: (0)             class TestTriuIndicesFrom:
417: (4)                 def test_exceptions(self):
418: (8)                     assert_raises(ValueError, triu_indices_from, np.ones((2,)))
419: (8)                     assert_raises(ValueError, triu_indices_from, np.ones((2, 2, 2)))
420: (0)             class TestVander:
421: (4)                 def test_basic(self):
422: (8)                     c = np.array([0, 1, -2, 3])
423: (8)                     v = vander(c)
424: (8)                     powers = np.array([[0, 0, 0, 0, 1],
425: (27)                         [1, 1, 1, 1, 1],
426: (27)                         [16, -8, 4, -2, 1],
427: (27)                         [81, 27, 9, 3, 1]])
428: (8)                     assert_array_equal(v, powers[:, 1:])
429: (8)                     m = powers.shape[1]
430: (8)                     for n in range(6):
431: (12)                         v = vander(c, N=n)
432: (12)                         assert_array_equal(v, powers[:, m-n:m])
433: (4)             def test_dtotypes(self):
434: (8)                 c = array([11, -12, 13], dtype=np.int8)
435: (8)                 v = vander(c)
436: (8)                 expected = np.array([[121, 11, 1],
437: (29)                     [144, -12, 1],
438: (29)                     [169, 13, 1]])
439: (8)                 assert_array_equal(v, expected)
440: (8)                 c = array([1.0+1j, 1.0-1j])
441: (8)                 v = vander(c, N=3)
442: (8)                 expected = np.array([[2j, 1+1j, 1],
443: (29)                     [-2j, 1-1j, 1]])
444: (8)                 assert_array_equal(v, expected)

```

-----

File 248 - test\_ufunclike.py:

```

1: (0)             import numpy as np
2: (0)             import numpy.core as nx
3: (0)             import numpy.lib.ufunclike as ufl
4: (0)             from numpy.testing import (

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

5: (4)             assert_, assert_equal, assert_array_equal, assert_warns, assert_raises
6: (0)
7: (0)
8: (4)
9: (8)
10: (8)
11: (8)
12: (8)
13: (8)
14: (8)
15: (8)
16: (8)
17: (8)
18: (8)
19: (12)
20: (4)
21: (8)
22: (8)
23: (8)
24: (8)
25: (8)
26: (8)
27: (8)
28: (8)
29: (8)
30: (8)
31: (12)
32: (4)
33: (8)
34: (8)
35: (8)
36: (8)
37: (8)
38: (8)
39: (8)
40: (8)
41: (8)
42: (4)
43: (8)
44: (12)
45: (16)
46: (16)
47: (16)
48: (12)
49: (16)
50: (20)
51: (16)
52: (12)
53: (16)
54: (16)
55: (8)
56: (8)
57: (8)
58: (8)
59: (8)
60: (8)
61: (8)
62: (8)
63: (8)
64: (8)
65: (8)
66: (4)
67: (8)
68: (8)
69: (8)
70: (8)
71: (8)
72: (8)
73: (8)

    class TestUfunclike:
        def test_isposinf(self):
            a = nx.array([nx.inf, -nx.inf, nx.nan, 0.0, 3.0, -3.0])
            out = nx.zeros(a.shape, bool)
            tgt = nx.array([True, False, False, False, False, False])
            res = ufl.isposinf(a)
            assert_equal(res, tgt)
            res = ufl.isposinf(a, out)
            assert_equal(res, tgt)
            assert_equal(out, tgt)
            a = a.astype(np.complex_)
            with assert_raises(TypeError):
                ufl.isposinf(a)
        def test_isneginf(self):
            a = nx.array([nx.inf, -nx.inf, nx.nan, 0.0, 3.0, -3.0])
            out = nx.zeros(a.shape, bool)
            tgt = nx.array([False, True, False, False, False, False])
            res = ufl.isneginf(a)
            assert_equal(res, tgt)
            res = ufl.isneginf(a, out)
            assert_equal(res, tgt)
            assert_equal(out, tgt)
            a = a.astype(np.complex_)
            with assert_raises(TypeError):
                ufl.isneginf(a)
        def test_fix(self):
            a = nx.array([[1.0, 1.1, 1.5, 1.8], [-1.0, -1.1, -1.5, -1.8]])
            out = nx.zeros(a.shape, float)
            tgt = nx.array([[1., 1., 1., 1.], [-1., -1., -1., -1.]])
            res = ufl.fix(a)
            assert_equal(res, tgt)
            res = ufl.fix(a, out)
            assert_equal(res, tgt)
            assert_equal(out, tgt)
            assert_equal(ufl.fix(3.14), 3)
        def test_fix_with_subclass(self):
            class MyArray(nx.ndarray):
                def __new__(cls, data, metadata=None):
                    res = nx.array(data, copy=True).view(cls)
                    res.metadata = metadata
                    return res
                def __array_wrap__(self, obj, context=None):
                    if isinstance(obj, MyArray):
                        obj.metadata = self.metadata
                    return obj
                def __array_finalize__(self, obj):
                    self.metadata = getattr(obj, 'metadata', None)
                    return self
            a = nx.array([1.1, -1.1])
            m = MyArray(a, metadata='foo')
            f = ufl.fix(m)
            assert_array_equal(f, nx.array([1, -1]))
            assert_(isinstance(f, MyArray))
            assert_equal(f.metadata, 'foo')
            m0d = m[0,...]
            m0d.metadata = 'bar'
            f0d = ufl.fix(m0d)
            assert_(isinstance(f0d, MyArray))
            assert_equal(f0d.metadata, 'bar')
        def test_scalar(self):
            x = np.inf
            actual = np.isposinf(x)
            expected = np.True_
            assert_equal(actual, expected)
            assert_equal(type(actual), type(expected))
            x = -3.4
            actual = np.fix(x)

```

```

74: (8)             expected = np.float64(-3.0)
75: (8)             assert_equal(actual, expected)
76: (8)             assert_equal(type(actual), type(expected))
77: (8)             out = np.array(0.0)
78: (8)             actual = np.fix(x, out=out)
79: (8)             assert_(actual is out)

```

---

## File 249 - test\_type\_check.py:

```

1: (0)             import numpy as np
2: (0)             from numpy.testing import (
3: (4)                 assert_, assert_equal, assert_array_equal, assert_raises
4: (4)             )
5: (0)             from numpy.lib.type_check import (
6: (4)                 common_type, mintypecode, isreal, iscomplex, isposinf, isneginf,
7: (4)                 nan_to_num, isrealobj, iscomplexobj, asfarray, real_if_close
8: (4)             )
9: (0)             def assert_all(x):
10: (4)                 assert_(np.all(x), x)
11: (0)             class TestCommonType:
12: (4)                 def test_basic(self):
13: (8)                     ai32 = np.array([[1, 2], [3, 4]], dtype=np.int32)
14: (8)                     af16 = np.array([[1, 2], [3, 4]], dtype=np.float16)
15: (8)                     af32 = np.array([[1, 2], [3, 4]], dtype=np.float32)
16: (8)                     af64 = np.array([[1, 2], [3, 4]], dtype=np.float64)
17: (8)                     acs = np.array([[1+5j, 2+6j], [3+7j, 4+8j]], dtype=np.csingle)
18: (8)                     acd = np.array([[1+5j, 2+6j], [3+7j, 4+8j]], dtype=np.cdouble)
19: (8)                     assert_(common_type(ai32) == np.float64)
20: (8)                     assert_(common_type(af16) == np.float16)
21: (8)                     assert_(common_type(af32) == np.float32)
22: (8)                     assert_(common_type(af64) == np.float64)
23: (8)                     assert_(common_type(acs) == np.csingle)
24: (8)                     assert_(common_type(acd) == np.cdouble)
25: (0)             class TestMintypecode:
26: (4)                 def test_default_1(self):
27: (8)                     for itype in '1bcswil':
28: (12)                         assert_equal(mintypecode(itype), 'd')
29: (8)                         assert_equal(mintypecode('f'), 'f')
30: (8)                         assert_equal(mintypecode('d'), 'd')
31: (8)                         assert_equal(mintypecode('F'), 'F')
32: (8)                         assert_equal(mintypecode('D'), 'D')
33: (4)                 def test_default_2(self):
34: (8)                     for itype in '1bcswil':
35: (12)                         assert_equal(mintypecode(itype+'f'), 'f')
36: (12)                         assert_equal(mintypecode(itype+'d'), 'd')
37: (12)                         assert_equal(mintypecode(itype+'F'), 'F')
38: (12)                         assert_equal(mintypecode(itype+'D'), 'D')
39: (8)                         assert_equal(mintypecode('ff'), 'f')
40: (8)                         assert_equal(mintypecode('fd'), 'd')
41: (8)                         assert_equal(mintypecode('fF'), 'F')
42: (8)                         assert_equal(mintypecode('fD'), 'D')
43: (8)                         assert_equal(mintypecode('df'), 'd')
44: (8)                         assert_equal(mintypecode('dd'), 'd')
45: (8)                         assert_equal(mintypecode('dF'), 'D')
46: (8)                         assert_equal(mintypecode('dD'), 'D')
47: (8)                         assert_equal(mintypecode('Ff'), 'F')
48: (8)                         assert_equal(mintypecode('Fd'), 'D')
49: (8)                         assert_equal(mintypecode('FF'), 'F')
50: (8)                         assert_equal(mintypecode('FD'), 'D')
51: (8)                         assert_equal(mintypecode('Df'), 'D')
52: (8)                         assert_equal(mintypecode('Dd'), 'D')
53: (8)                         assert_equal(mintypecode('DF'), 'D')
54: (8)                         assert_equal(mintypecode('DD'), 'D')
55: (4)                 def test_default_3(self):
56: (8)                     assert_equal(mintypecode('fdf'), 'D')
57: (8)                     assert_equal(mintypecode('fdD'), 'D')
58: (8)                     assert_equal(mintypecode('ffd'), 'D')

```

```

59: (8)             assert_equal(mintypecode('dFD'), 'D')
60: (8)             assert_equal(mintypecode('ifd'), 'd')
61: (8)             assert_equal(mintypecode('iff'), 'F')
62: (8)             assert_equal(mintypecode('ifD'), 'D')
63: (8)             assert_equal(mintypecode('idF'), 'D')
64: (8)             assert_equal(mintypecode('idD'), 'D')
65: (0)             class TestIsscalar:
66: (4)                 def test_basic(self):
67: (8)                     assert_(np.isscalar(3))
68: (8)                     assert_(not np.isscalar([3]))
69: (8)                     assert_(not np.isscalar((3,)))
70: (8)                     assert_(np.isscalar(3j))
71: (8)                     assert_(np.isscalar(4.0))
72: (0)             class TestReal:
73: (4)                 def test_real(self):
74: (8)                     y = np.random.rand(10,)
75: (8)                     assert_array_equal(y, np.real(y))
76: (8)                     y = np.array(1)
77: (8)                     out = np.real(y)
78: (8)                     assert_array_equal(y, out)
79: (8)                     assert_(isinstance(out, np.ndarray))
80: (8)                     y = 1
81: (8)                     out = np.real(y)
82: (8)                     assert_equal(y, out)
83: (8)                     assert_(not isinstance(out, np.ndarray))
84: (4)             def test_cmplx(self):
85: (8)                 y = np.random.rand(10,)+1j*np.random.rand(10,)
86: (8)                 assert_array_equal(y.real, np.real(y))
87: (8)                 y = np.array(1 + 1j)
88: (8)                 out = np.real(y)
89: (8)                 assert_array_equal(y.real, out)
90: (8)                 assert_(isinstance(out, np.ndarray))
91: (8)                 y = 1 + 1j
92: (8)                 out = np.real(y)
93: (8)                 assert_equal(1.0, out)
94: (8)                 assert_(not isinstance(out, np.ndarray))
95: (0)             class TestImag:
96: (4)                 def test_real(self):
97: (8)                     y = np.random.rand(10,)
98: (8)                     assert_array_equal(0, np.imag(y))
99: (8)                     y = np.array(1)
100: (8)                    out = np.imag(y)
101: (8)                    assert_array_equal(0, out)
102: (8)                    assert_(isinstance(out, np.ndarray))
103: (8)                    y = 1
104: (8)                    out = np.imag(y)
105: (8)                    assert_equal(0, out)
106: (8)                    assert_(not isinstance(out, np.ndarray))
107: (4)             def test_cmplx(self):
108: (8)                 y = np.random.rand(10,)+1j*np.random.rand(10,)
109: (8)                 assert_array_equal(y.imag, np.imag(y))
110: (8)                 y = np.array(1 + 1j)
111: (8)                 out = np.imag(y)
112: (8)                 assert_array_equal(y.imag, out)
113: (8)                 assert_(isinstance(out, np.ndarray))
114: (8)                 y = 1 + 1j
115: (8)                 out = np.imag(y)
116: (8)                 assert_equal(1.0, out)
117: (8)                 assert_(not isinstance(out, np.ndarray))
118: (0)             class TestIscomplex:
119: (4)                 def test_fail(self):
120: (8)                     z = np.array([-1, 0, 1])
121: (8)                     res = iscomplex(z)
122: (8)                     assert_(not np.any(res, axis=0))
123: (4)             def test_pass(self):
124: (8)                     z = np.array([-1j, 1, 0])
125: (8)                     res = iscomplex(z)
126: (8)                     assert_array_equal(res, [1, 0, 0])
127: (0)             class TestIsreal:

```

```

128: (4)             def test_pass(self):
129: (8)               z = np.array([-1, 0, 1j])
130: (8)               res = isreal(z)
131: (8)               assert_array_equal(res, [1, 1, 0])
132: (4)             def test_fail(self):
133: (8)               z = np.array([-1j, 1, 0])
134: (8)               res = isreal(z)
135: (8)               assert_array_equal(res, [0, 1, 1])
136: (0)             class TestIscomplexobj:
137: (4)               def test_basic(self):
138: (8)                 z = np.array([-1, 0, 1])
139: (8)                 assert_(not iscomplexobj(z))
140: (8)                 z = np.array([-1j, 0, -1])
141: (8)                 assert_(iscomplexobj(z))
142: (4)               def test_scalar(self):
143: (8)                 assert_(not iscomplexobj(1.0))
144: (8)                 assert_(iscomplexobj(1+0j))
145: (4)               def test_list(self):
146: (8)                 assert_(iscomplexobj([3, 1+0j, True]))
147: (8)                 assert_(not iscomplexobj([3, 1, True]))
148: (4)               def test_duck(self):
149: (8)                 class DummyComplexArray:
150: (12)                   @property
151: (12)                   def dtype(self):
152: (16)                     return np.dtype(complex)
153: (8)                     dummy = DummyComplexArray()
154: (8)                     assert_(iscomplexobj(dummy))
155: (4)               def test_pandas_duck(self):
156: (8)                 class PdComplex(np.complex128):
157: (12)                   pass
158: (8)                 class PdDtype:
159: (12)                   name = 'category'
160: (12)                   names = None
161: (12)                   type = PdComplex
162: (12)                   kind = 'c'
163: (12)                   str = '<c16'
164: (12)                   base = np.dtype('complex128')
165: (8)                 class DummyPd:
166: (12)                   @property
167: (12)                   def dtype(self):
168: (16)                     return PdDtype
169: (8)                     dummy = DummyPd()
170: (8)                     assert_(iscomplexobj(dummy))
171: (4)               def test_custom_dtype_duck(self):
172: (8)                 class MyArray(list):
173: (12)                   @property
174: (12)                   def dtype(self):
175: (16)                     return complex
176: (8)                     a = MyArray([1+0j, 2+0j, 3+0j])
177: (8)                     assert_(iscomplexobj(a))
178: (0)             class TestIsrealobj:
179: (4)               def test_basic(self):
180: (8)                 z = np.array([-1, 0, 1])
181: (8)                 assert_(isrealobj(z))
182: (8)                 z = np.array([-1j, 0, -1])
183: (8)                 assert_(not isrealobj(z))
184: (0)             class Testisnan:
185: (4)               def test_goodvalues(self):
186: (8)                 z = np.array((-1., 0., 1.))
187: (8)                 res = np.isnan(z) == 0
188: (8)                 assert_all(np.all(res, axis=0))
189: (4)               def test_posinf(self):
190: (8)                 with np.errstate(divide='ignore'):
191: (12)                   assert_all(np.isnan(np.array((1.,))/0.) == 0)
192: (4)               def test_neginf(self):
193: (8)                 with np.errstate(divide='ignore'):
194: (12)                   assert_all(np.isnan(np.array((-1.,))/0.) == 0)
195: (4)               def test_ind(self):
196: (8)                 with np.errstate(divide='ignore', invalid='ignore'):

```

```

197: (12)                                assert_all(np.isnan(np.array((0.,))/0.) == 1)
198: (4)       def test_integer(self):
199: (8)           assert_all(np.isnan(1) == 0)
200: (4)       def test_complex(self):
201: (8)           assert_all(np.isnan(1+1j) == 0)
202: (4)       def test_complex1(self):
203: (8)           with np.errstate(divide='ignore', invalid='ignore'):
204: (12)               assert_all(np.isnan(np.array(0+0j)/0.) == 1)
205: (0) class TestIsfinite:
206: (4)     def test_goodvalues(self):
207: (8)         z = np.array((-1., 0., 1.))
208: (8)         res = np.isfinite(z) == 1
209: (8)         assert_all(np.all(res, axis=0))
210: (4)     def test_posinf(self):
211: (8)         with np.errstate(divide='ignore', invalid='ignore'):
212: (12)             assert_all(np.isfinite(np.array((1.,))/0.) == 0)
213: (4)     def test_neginf(self):
214: (8)         with np.errstate(divide='ignore', invalid='ignore'):
215: (12)             assert_all(np.isfinite(np.array((-1.,))/0.) == 0)
216: (4)     def test_ind(self):
217: (8)         with np.errstate(divide='ignore', invalid='ignore'):
218: (12)             assert_all(np.isfinite(np.array((0.,))/0.) == 0)
219: (4)     def test_integer(self):
220: (8)         assert_all(np.isfinite(1) == 1)
221: (4)     def test_complex(self):
222: (8)         assert_all(np.isfinite(1+1j) == 1)
223: (4)     def test_complex1(self):
224: (8)         with np.errstate(divide='ignore', invalid='ignore'):
225: (12)             assert_all(np.isfinite(np.array(1+1j)/0.) == 0)
226: (0) class TestIsinf:
227: (4)     def test_goodvalues(self):
228: (8)         z = np.array((-1., 0., 1.))
229: (8)         res = np.isinf(z) == 0
230: (8)         assert_all(np.all(res, axis=0))
231: (4)     def test_posinf(self):
232: (8)         with np.errstate(divide='ignore', invalid='ignore'):
233: (12)             assert_all(np.isinf(np.array((1.,))/0.) == 1)
234: (4)     def test_posinf_scalar(self):
235: (8)         with np.errstate(divide='ignore', invalid='ignore'):
236: (12)             assert_all(np.isinf(np.array(1.))/0.) == 1)
237: (4)     def test_neginf(self):
238: (8)         with np.errstate(divide='ignore', invalid='ignore'):
239: (12)             assert_all(np.isinf(np.array((-1.,))/0.) == 1)
240: (4)     def test_neginf_scalar(self):
241: (8)         with np.errstate(divide='ignore', invalid='ignore'):
242: (12)             assert_all(np.isinf(np.array(-1.))/0.) == 1)
243: (4)     def test_ind(self):
244: (8)         with np.errstate(divide='ignore', invalid='ignore'):
245: (12)             assert_all(np.isinf(np.array((0.,))/0.) == 0)
246: (0) class TestIsposinf:
247: (4)     def test_generic(self):
248: (8)         with np.errstate(divide='ignore', invalid='ignore'):
249: (12)             vals = isposinf(np.array((-1., 0, 1))/0.)
250: (8)             assert_(vals[0] == 0)
251: (8)             assert_(vals[1] == 0)
252: (8)             assert_(vals[2] == 1)
253: (0) class TestIsneginf:
254: (4)     def test_generic(self):
255: (8)         with np.errstate(divide='ignore', invalid='ignore'):
256: (12)             vals = isneginf(np.array((-1., 0, 1))/0.)
257: (8)             assert_(vals[0] == 1)
258: (8)             assert_(vals[1] == 0)
259: (8)             assert_(vals[2] == 0)
260: (0) class TestNanToNum:
261: (4)     def test_generic(self):
262: (8)         with np.errstate(divide='ignore', invalid='ignore'):
263: (12)             vals = nan_to_num(np.array((-1., 0, 1))/0.)
264: (8)             assert_all(vals[0] < -1e10) and assert_all(np.isfinite(vals[0]))
265: (8)             assert_(vals[1] == 0)

```

```

266: (8) assert_all(vals[2] > 1e10) and assert_all(np.isfinite(vals[2]))
267: (8) assert_equal(type(vals), np.ndarray)
268: (8) with np.errstate(divide='ignore', invalid='ignore'):
269: (12)     vals = nan_to_num(np.array((-1., 0, 1))/0.,
270: (30)             nan=10, posinf=20, neginf=30)
271: (8) assert_equal(vals, [30, 10, 20])
272: (8) assert_all(np.isfinite(vals[[0, 2]]))
273: (8) assert_equal(type(vals), np.ndarray)
274: (8) with np.errstate(divide='ignore', invalid='ignore'):
275: (12)     vals = np.array((-1., 0, 1))/0.
276: (8) result = nan_to_num(vals, copy=False)
277: (8) assert_(result is vals)
278: (8) assert_all(vals[0] < -1e10) and assert_all(np.isfinite(vals[0]))
279: (8) assert_(vals[1] == 0)
280: (8) assert_all(vals[2] > 1e10) and assert_all(np.isfinite(vals[2]))
281: (8) assert_equal(type(vals), np.ndarray)
282: (8) with np.errstate(divide='ignore', invalid='ignore'):
283: (12)     vals = np.array((-1., 0, 1))/0.
284: (8) result = nan_to_num(vals, copy=False, nan=10, posinf=20, neginf=30)
285: (8) assert_(result is vals)
286: (8) assert_equal(vals, [30, 10, 20])
287: (8) assert_all(np.isfinite(vals[[0, 2]]))
288: (8) assert_equal(type(vals), np.ndarray)
289: (4) def test_array(self):
290: (8)     vals = nan_to_num([1])
291: (8)     assert_array_equal(vals, np.array([1], int))
292: (8)     assert_equal(type(vals), np.ndarray)
293: (8)     vals = nan_to_num([1], nan=10, posinf=20, neginf=30)
294: (8)     assert_array_equal(vals, np.array([1], int))
295: (8)     assert_equal(type(vals), np.ndarray)
296: (4) def test_integer(self):
297: (8)     vals = nan_to_num(1)
298: (8)     assert_all(vals == 1)
299: (8)     assert_equal(type(vals), np.int_)
300: (8)     vals = nan_to_num(1, nan=10, posinf=20, neginf=30)
301: (8)     assert_all(vals == 1)
302: (8)     assert_equal(type(vals), np.int_)
303: (4) def test_float(self):
304: (8)     vals = nan_to_num(1.0)
305: (8)     assert_all(vals == 1.0)
306: (8)     assert_equal(type(vals), np.float_)
307: (8)     vals = nan_to_num(1.1, nan=10, posinf=20, neginf=30)
308: (8)     assert_all(vals == 1.1)
309: (8)     assert_equal(type(vals), np.float_)
310: (4) def test_complex_good(self):
311: (8)     vals = nan_to_num(1+1j)
312: (8)     assert_all(vals == 1+1j)
313: (8)     assert_equal(type(vals), np.complex_)
314: (8)     vals = nan_to_num(1+1j, nan=10, posinf=20, neginf=30)
315: (8)     assert_all(vals == 1+1j)
316: (8)     assert_equal(type(vals), np.complex_)
317: (4) def test_complex_bad(self):
318: (8)     with np.errstate(divide='ignore', invalid='ignore'):
319: (12)         v = 1 + 1j
320: (12)         v += np.array(0+1.j)/0.
321: (8)         vals = nan_to_num(v)
322: (8)         assert_all(np.isfinite(vals))
323: (8)         assert_equal(type(vals), np.complex_)
324: (4) def test_complex_bad2(self):
325: (8)     with np.errstate(divide='ignore', invalid='ignore'):
326: (12)         v = 1 + 1j
327: (12)         v += np.array(-1+1.j)/0.
328: (8)         vals = nan_to_num(v)
329: (8)         assert_all(np.isfinite(vals))
330: (8)         assert_equal(type(vals), np.complex_)
331: (4) def test_do_not_rewrite_previous_keyword(self):
332: (8)     with np.errstate(divide='ignore', invalid='ignore'):
333: (12)         vals = nan_to_num(np.array((-1., 0, 1))/0., nan=np.inf,
posinf=999)

```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

334: (8)             assert_all(np.isfinite(vals[[0, 2]]))
335: (8)             assert_all(vals[0] < -1e10)
336: (8)             assert_equal(vals[[1, 2]], [np.inf, 999])
337: (8)             assert_equal(type(vals), np.ndarray)
338: (0)              class TestRealIfClose:
339: (4)                def test_basic(self):
340: (8)                  a = np.random.rand(10)
341: (8)                  b = real_if_close(a+1e-15j)
342: (8)                  assert_all(isrealobj(b))
343: (8)                  assert_array_equal(a, b)
344: (8)                  b = real_if_close(a+1e-7j)
345: (8)                  assert_all(iscomplexobj(b))
346: (8)                  b = real_if_close(a+1e-7j, tol=1e-6)
347: (8)                  assert_all(isrealobj(b))
348: (0)              class TestArrayConversion:
349: (4)                def test_asfarray(self):
350: (8)                  a = asfarray(np.array([1, 2, 3]))
351: (8)                  assert_equal(a.__class__, np.ndarray)
352: (8)                  assert_(np.issubdtype(a.dtype, np.floating))
353: (8)                  assert_raises(TypeError,
354: (12)                      asfarray, np.array([1, 2, 3]), dtype=np.array(1.0))

-----
```

## File 250 - test\_utils.py:

```
1: (0)          import inspect
2: (0)          import sys
3: (0)          import pytest
4: (0)          import numpy as np
5: (0)          from numpy.core import arange
6: (0)          from numpy.testing import assert_, assert_equal, assert_raises_regex
7: (0)          from numpy.lib import deprecate, deprecate_with_doc
8: (0)          import numpy.lib.utils as utils
9: (0)          from io import StringIO
10: (0)         @pytest.mark.skipif(sys.flags.optimize == 2, reason="Python running -OO")
11: (0)         @pytest.mark.skipif(
12: (4)           sys.version_info == (3, 10, 0, "candidate", 1),
13: (4)           reason="Broken as of bpo-44524",
14: (0)       )
15: (0)         def test_lookfor():
16: (4)           out = StringIO()
17: (4)           utils.lookfor('eigenvalue', module='numpy', output=out,
18: (18)               import_modules=False)
19: (4)           out = out.getvalue()
20: (4)           assert_('numpy.linalg.eig' in out)
21: (0)           @deprecate
22: (0)           def old_func(self, x):
23: (4)             return x
24: (0)           @deprecate(message="Rather use new_func2")
25: (0)           def old_func2(self, x):
26: (4)             return x
27: (0)           def old_func3(self, x):
28: (4)             return x
29: (0)           new_func3 = deprecate(old_func3, old_name="old_func3", new_name="new_func3")
30: (0)           def old_func4(self, x):
31: (4)             """Summary.
32: (4)             Further info.
33: (4)             """
34: (4)             return x
35: (0)           new_func4 = deprecate(old_func4)
36: (0)           def old_func5(self, x):
37: (4)             """Summary.
38: (8)               Bizarre indentation.
39: (4)             """
40: (4)             return x
41: (0)           new_func5 = deprecate(old_func5, message="This function is\\ndeprecated.")
42: (0)           def old_func6(self, x):
43: (4)             """
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

44: (4)           Also in PEP-257.
45: (4)
46: (4)           """
47: (0)           return x
48: (0)           new_func6 = deprecate(old_func6)
49: (0)           @deprecate_with_doc(msg="Rather use new_func7")
50: (4)           def old_func7(self,x):
51: (0)               return x
52: (4)           def test_deprecate_decorator():
53: (0)               assert_('deprecated' in old_func.__doc__)
54: (4)           def test_deprecate_decorator_message():
55: (0)               assert_('Rather use new_func2' in old_func2.__doc__)
56: (4)           def test_deprecate_fn():
57: (4)               assert_('old_func3' in new_func3.__doc__)
58: (0)               assert_('new_func3' in new_func3.__doc__)
59: (4)           def test_deprecate_with_doc_decorator_message():
60: (0)               assert_('Rather use new_func7' in old_func7.__doc__)
61: (0)           @pytest.mark.skipif(sys.flags.optimize == 2, reason="-O0 discards docstrings")
62: (4)           @pytest.mark.parametrize('old_func, new_func', [
63: (4)               (old_func4, new_func4),
64: (4)               (old_func5, new_func5),
65: (4)               (old_func6, new_func6),
66: (0)           ])
67: (4)           def test_deprecate_help_indentation(old_func, new_func):
68: (4)               _compare_docs(old_func, new_func)
69: (8)               for knd, func in (('old', old_func), ('new', new_func)):
70: (12)                 for li, line in enumerate(func.__doc__.split('\n')):
71: (16)                   if li == 0:
72: (12)                       assert line.startswith(' ') or not line.startswith(' ')
73: (16)                   elif line:
74: (0)                       assert line.startswith(' ', knd)
75: (4)           def _compare_docs(old_func, new_func):
76: (4)               old_doc = inspect.getdoc(old_func)
77: (4)               new_doc = inspect.getdoc(new_func)
78: (4)               index = new_doc.index('\n\n') + 2
79: (0)               assert_equal(new_doc[index:], old_doc)
80: (0)           @pytest.mark.skipif(sys.flags.optimize == 2, reason="-O0 discards docstrings")
81: (4)           def test_deprecate_preserve_whitespace():
82: (0)               assert_('\n      Bizarre' in new_func5.__doc__)
83: (4)           def test_deprecate_module():
84: (0)               assert_(old_func.__module__ == __name__)
85: (4)           def test_safe_eval_nameconstant():
86: (0)               utils.safe_eval('None')
87: (4)           class TestByteBounds:
88: (8)               def test_byte_bounds(self):
89: (8)                   a = arange(12).reshape(3, 4)
90: (8)                   low, high = utils.byte_bounds(a)
91: (4)                   assert_equal(high - low, a.size * a.itemsize)
92: (8)               def test_unusual_order_positive_stride(self):
93: (8)                   a = arange(12).reshape(3, 4)
94: (8)                   b = a.T
95: (8)                   low, high = utils.byte_bounds(b)
96: (4)                   assert_equal(high - low, b.size * b.itemsize)
97: (8)               def test_unusual_order_negative_stride(self):
98: (8)                   a = arange(12).reshape(3, 4)
99: (8)                   b = a.T[::-1]
100: (8)                  low, high = utils.byte_bounds(b)
101: (4)                  assert_equal(high - low, b.size * b.itemsize)
102: (8)               def test_strided(self):
103: (8)                   a = arange(12)
104: (8)                   b = a[::2]
105: (8)                   low, high = utils.byte_bounds(b)
106: (0)                   assert_equal(high - low, b.size * 2 * b.itemsize - b.itemsize)
107: (4)               def test_assert_raises_regex_context_manager():
108: (8)                   with assert_raises_regex(ValueError, 'no deprecation warning'):
109: (8)                       raise ValueError('no deprecation warning')
110: (4)               def test_info_method_heading():
111: (8)                   class NoPublicMethods:

```

```

112: (4)             class WithPublicMethods:
113: (8)                 def first_method():
114: (12)                     pass
115: (4)             def _has_method_heading(cls):
116: (8)                 out = StringIO()
117: (8)                 utils.info(cls, output=out)
118: (8)                 return 'Methods:' in out.getvalue()
119: (4)             assert _has_method_heading(WithPublicMethods)
120: (4)             assert not _has_method_heading(NoPublicMethods)
121: (0)         def test_drop_metadata():
122: (4)             def _compare_dtotypes(dt1, dt2):
123: (8)                 return np.can_cast(dt1, dt2, casting='no')
124: (4)             dt = np.dtype([('l1', [('l2', np.dtype('S8', metadata={'msg':
125: ('toto')))]),),
126: (18)                         metadata={'msg': 'titi'})]
127: (4)             dt_m = utils.drop_metadata(dt)
128: (4)             assert _compare_dtotypes(dt, dt_m) is True
129: (4)             assert dt_m.metadata is None
130: (4)             assert dt_m['l1'].metadata is None
131: (4)             assert dt_m['l1']['l2'].metadata is None
132: (18)             dt = np.dtype([('x', '<f8'), ('y', '<i4')],
133: (18)                         align=True,
134: (4)                         metadata={'msg': 'toto'})
135: (4)             dt_m = utils.drop_metadata(dt)
136: (4)             assert _compare_dtotypes(dt, dt_m) is True
137: (4)             assert dt_m.metadata is None
138: (18)             dt = np.dtype('8f',
139: (4)                         metadata={'msg': 'toto'})
140: (4)             dt_m = utils.drop_metadata(dt)
141: (4)             assert _compare_dtotypes(dt, dt_m) is True
142: (4)             assert dt_m.metadata is None
143: (18)             dt = np.dtype('uint32',
144: (4)                         metadata={'msg': 'toto'})
145: (4)             dt_m = utils.drop_metadata(dt)
146: (4)             assert _compare_dtotypes(dt, dt_m) is True
147: (0)             assert dt_m.metadata is None
148: (8)         @pytest.mark.parametrize("dtype",
149: (8)             [np.dtype("i,i,i,i")[[ "f1", "f3"]],
150: (8)             np.dtype("f8"),
151: (8)             np.dtype("10i")])
152: (0)         def test_drop_metadata_identity_and_copy(dtype):
153: (4)             assert utils.drop_metadata(dtype) is dtype
154: (4)             dtype = np.dtype(dtype, metadata={1: 2})
155: (4)             assert utils.drop_metadata(dtype).metadata is None

```

---

## File 251 - test\_datasource.py:

```

1: (0)             import os
2: (0)             import pytest
3: (0)             from tempfile import mkdtemp, mkstemp, NamedTemporaryFile
4: (0)             from shutil import rmtree
5: (0)             import numpy.lib._datasource as datasource
6: (0)             from numpy.testing import assert_, assert_equal, assert_raises
7: (0)             import urllib.request as urllib_request
8: (0)             from urllib.parse import urlparse
9: (0)             from urllib.error import URLError
10: (0)             def urlopen_stub(url, data=None):
11: (4)                 '''Stub to replace urlopen for testing.'''
12: (4)                 if url == valid_httpurl():
13: (8)                     tmpfile = NamedTemporaryFile(prefix='urltmp_')
14: (8)                     return tmpfile
15: (4)                 else:
16: (8)                     raise URLError('Name or service not known')
17: (0)             old_urlopen = None
18: (0)             def setup_module():
19: (4)                 global old_urlopen
20: (4)                 old_urlopen = urllib_request.urlopen

```

```

21: (4)             urllib_request.urlopen = urlopen_stub
22: (0)
23: (4)             def teardown_module():
24: (0)                 urllib_request.urlopen = old_urlopen
25: (0)                 http_path = 'http://www.google.com/'
26: (0)                 http_file = 'index.html'
27: (0)                 http_fakepath = 'http://fake.abc.web/site/'
28: (0)                 http_fakefile = 'fake.txt'
29: (19)                malicious_files = ['/etc/shadow', '../..shadow',
30: (0)                                '..\\system.dat', 'c:\\windows\\system.dat']
31: (0)                magic_line = b'three is the magic number'
32: (4)                def valid_textfile(filedir):
33: (4)                    fd, path = mkstemp(suffix='.txt', prefix='dstmp_', dir=filedir, text=True)
34: (4)                    os.close(fd)
35: (4)                    return path
36: (4)                def invalid_textfile(filedir):
37: (4)                    fd, path = mkstemp(suffix='.txt', prefix='dstmp_', dir=filedir)
38: (4)                    os.close(fd)
39: (4)                    os.remove(path)
40: (0)                    return path
41: (4)                def valid_httpurl():
42: (0)                    return http_path+http_file
43: (4)                def invalid_httpurl():
44: (0)                    return http_fakepath+http_fakefile
45: (4)                def valid_baseurl():
46: (0)                    return http_path
47: (4)                def invalid_baseurl():
48: (0)                    return http_fakepath
49: (4)                def valid_httpfile():
50: (0)                    return http_file
51: (4)                def invalid_httpfile():
52: (0)                    return http_fakefile
53: (4)                class TestDataSourceOpen:
54: (8)                    def setup_method(self):
55: (8)                        self.tmpdir = mkdtemp()
56: (8)                        self.ds = datasource.DataSource(self.tmpdir)
57: (8)                    def teardown_method(self):
58: (8)                        rmtree(self.tmpdir)
59: (8)                        del self.ds
60: (8)                    def test_ValidHTTP(self):
61: (8)                        fh = self.ds.open(valid_httpurl())
62: (8)                        assert_(fh)
63: (8)                        fh.close()
64: (8)                    def test_InvalidHTTP(self):
65: (8)                        url = invalid_httpurl()
66: (8)                        assert_raises(OSError, self.ds.open, url)
67: (12)                        try:
68: (8)                            self.ds.open(url)
69: (12)                        except OSError as e:
70: (4)                            assert_(e.errno is None)
71: (8)                    def test_InvalidHTTPCacheURLError(self):
72: (4)                        assert_raises(URLError, self.ds._cache, invalid_httpurl())
73: (8)                    def test_ValidFile(self):
74: (8)                        local_file = valid_textfile(self.tmpdir)
75: (8)                        fh = self.ds.open(local_file)
76: (8)                        assert_(fh)
77: (8)                        fh.close()
78: (8)                    def test_InvalidFile(self):
79: (8)                        invalid_file = invalid_textfile(self.tmpdir)
80: (8)                        assert_raises(OSError, self.ds.open, invalid_file)
81: (8)                    def test_ValidGzipFile(self):
82: (12)                        try:
83: (8)                            import gzip
84: (12)                        except ImportError:
85: (8)                            pytest.skip()
86: (8)                            filepath = os.path.join(self.tmpdir, 'foobar.txt.gz')
87: (8)                            fp = gzip.open(filepath, 'w')
88: (8)                            fp.write(magic_line)
89: (8)                            fp.close()

```

```

90: (8)             result = fp.readline()
91: (8)             fp.close()
92: (8)             assert_equal(magic_line, result)
93: (4)         def test_ValidBz2File(self):
94: (8)             try:
95: (12)                 import bz2
96: (8)             except ImportError:
97: (12)                 pytest.skip()
98: (8)             filepath = os.path.join(self.tmpdir, 'foobar.txt.bz2')
99: (8)             fp = bz2.BZ2File(filepath, 'w')
100: (8)             fp.write(magic_line)
101: (8)             fp.close()
102: (8)             fp = self.ds.open(filepath)
103: (8)             result = fp.readline()
104: (8)             fp.close()
105: (8)             assert_equal(magic_line, result)
106: (0)         class TestDataSourceExists:
107: (4)             def setup_method(self):
108: (8)                 self.tmpdir = mkdtemp()
109: (8)                 self.ds = datasource.DataSource(self.tmpdir)
110: (4)             def teardown_method(self):
111: (8)                 rmtree(self.tmpdir)
112: (8)                 del self.ds
113: (4)             def test_ValidHTTP(self):
114: (8)                 assert_(self.ds.exists(valid_httpurl()))
115: (4)             def test_InvalidHTTP(self):
116: (8)                 assert_equal(self.ds.exists(invalid_httpurl()), False)
117: (4)             def test_ValidFile(self):
118: (8)                 tmpfile = valid_textfile(self.tmpdir)
119: (8)                 assert_(self.ds.exists(tmpfile))
120: (8)                 localdir = mkdtemp()
121: (8)                 tmpfile = valid_textfile(locaaldir)
122: (8)                 assert_(self.ds.exists(tmpfile))
123: (8)                 rmtree(locaaldir)
124: (4)             def test_InvalidFile(self):
125: (8)                 tmpfile = invalid_textfile(self.tmpdir)
126: (8)                 assert_equal(self.ds.exists(tmpfile), False)
127: (0)         class TestDataSourceAbspath:
128: (4)             def setup_method(self):
129: (8)                 self.tmpdir = os.path.abspath(mkdtemp())
130: (8)                 self.ds = datasource.DataSource(self.tmpdir)
131: (4)             def teardown_method(self):
132: (8)                 rmtree(self.tmpdir)
133: (8)                 del self.ds
134: (4)             def test_ValidHTTP(self):
135: (8)                 scheme, netloc, upath, pms, qry, frg = urlparse(valid_httpurl())
136: (8)                 local_path = os.path.join(self.tmpdir, netloc,
137: (34)                               upath.strip(os.sep).strip('/'))
138: (8)                 assert_equal(local_path, self.ds.abspath(valid_httpurl()))
139: (4)             def test_ValidFile(self):
140: (8)                 tmpfile = valid_textfile(self.tmpdir)
141: (8)                 tmpfilename = os.path.split(tmpfile)[-1]
142: (8)                 assert_equal(tmpfile, self.ds.abspath(tmpfilename))
143: (8)                 assert_equal(tmpfile, self.ds.abspath(tmpfile))
144: (4)             def test_InvalidHTTP(self):
145: (8)                 scheme, netloc, upath, pms, qry, frg = urlparse(invalid_httpurl())
146: (8)                 invalidhttp = os.path.join(self.tmpdir, netloc,
147: (35)                               upath.strip(os.sep).strip('/'))
148: (8)                 assert_(invalidhttp != self.ds.abspath(valid_httpurl()))
149: (4)             def test_InvalidFile(self):
150: (8)                 invalidfile = valid_textfile(self.tmpdir)
151: (8)                 tmpfile = valid_textfile(self.tmpdir)
152: (8)                 tmpfilename = os.path.split(tmpfile)[-1]
153: (8)                 assert_(invalidfile != self.ds.abspath(tmpfilename))
154: (8)                 assert_(invalidfile != self.ds.abspath(tmpfile))
155: (4)             def test_sandboxing(self):
156: (8)                 tmpfile = valid_textfile(self.tmpdir)
157: (8)                 tmpfilename = os.path.split(tmpfile)[-1]
158: (8)                 tmp_path = lambda x: os.path.abspath(self.ds.abspath(x))

```

```

159: (8)             assert_(tmp_path(valid_httpurl()).startswith(self.tmpdir))
160: (8)             assert_(tmp_path(invalid_httpurl()).startswith(self.tmpdir))
161: (8)             assert_(tmp_path(tmpfile).startswith(self.tmpdir))
162: (8)             assert_(tmp_path(tmpfilename).startswith(self.tmpdir))
163: (8)             for fn in malicious_files:
164: (12)                 assert_(tmp_path(http_path+fn).startswith(self.tmpdir))
165: (12)                 assert_(tmp_path(fn).startswith(self.tmpdir))
166: (4)             def test_windows_os_sep(self):
167: (8)                 orig_os_sep = os.sep
168: (8)                 try:
169: (12)                     os.sep = '\\\\'
170: (12)                     self.test_ValidHTTP()
171: (12)                     self.test_ValidFile()
172: (12)                     self.test_InvalidHTTP()
173: (12)                     self.test_InvalidFile()
174: (12)                     self.test_sandboxing()
175: (8)                 finally:
176: (12)                     os.sep = orig_os_sep
177: (0)             class TestRepositoryAbspath:
178: (4)                 def setup_method(self):
179: (8)                     self.tmpdir = os.path.abspath(mkdtemp())
180: (8)                     self.repos = datasource.Repository(valid_baseurl(), self.tmpdir)
181: (4)                 def teardown_method(self):
182: (8)                     rmtree(self.tmpdir)
183: (8)                     del self.repos
184: (4)                 def test_ValidHTTP(self):
185: (8)                     scheme, netloc, upath, pms, qry, frg = urlparse(valid_httpurl())
186: (8)                     local_path = os.path.join(self.repos._destpath, netloc,
187: (34)                                     upath.strip(os.sep).strip('/'))
188: (8)                     filepath = self.repos.abspath(valid_httpfile())
189: (8)                     assert_equal(local_path, filepath)
190: (4)                 def test_sandboxing(self):
191: (8)                     tmp_path = lambda x: os.path.abspath(self.repos.abspath(x))
192: (8)                     assert_(tmp_path(valid_httpfile()).startswith(self.tmpdir))
193: (8)                     for fn in malicious_files:
194: (12)                         assert_(tmp_path(http_path+fn).startswith(self.tmpdir))
195: (12)                         assert_(tmp_path(fn).startswith(self.tmpdir))
196: (4)                 def test_windows_os_sep(self):
197: (8)                     orig_os_sep = os.sep
198: (8)                     try:
199: (12)                         os.sep = '\\\\'
200: (12)                         self.test_ValidHTTP()
201: (12)                         self.test_sandboxing()
202: (8)                     finally:
203: (12)                         os.sep = orig_os_sep
204: (0)             class TestRepositoryExists:
205: (4)                 def setup_method(self):
206: (8)                     self.tmpdir = makedirs()
207: (8)                     self.repos = datasource.Repository(valid_baseurl(), self.tmpdir)
208: (4)                 def teardown_method(self):
209: (8)                     rmtree(self.tmpdir)
210: (8)                     del self.repos
211: (4)                 def test_ValidFile(self):
212: (8)                     tmpfile = valid_textfile(self.tmpdir)
213: (8)                     assert_(self.repos.exists(tmpfile))
214: (4)                 def test_InvalidFile(self):
215: (8)                     tmpfile = invalid_textfile(self.tmpdir)
216: (8)                     assert_equal(self.repos.exists(tmpfile), False)
217: (4)                 def test_RemoveHTTPFile(self):
218: (8)                     assert_(self.repos.exists(valid_httpurl()))
219: (4)                 def test_CachedHTTPFile(self):
220: (8)                     localfile = valid_httpurl()
221: (8)                     scheme, netloc, upath, pms, qry, frg = urlparse(localfile)
222: (8)                     local_path = os.path.join(self.repos._destpath, netloc)
223: (8)                     os.mkdir(local_path, 0o0700)
224: (8)                     tmpfile = valid_textfile(local_path)
225: (8)                     assert_(self.repos.exists(tmpfile))
226: (0)             class TestOpenFunc:
227: (4)                 def setup_method(self):

```

```

228: (8)             self.tmpdir = mkdtemp()
229: (4)         def teardown_method(self):
230: (8)             rmtree(self.tmpdir)
231: (4)         def test_DataSourceOpen(self):
232: (8)             local_file = valid_textfile(self.tmpdir)
233: (8)             fp = datasource.open(local_file, destpath=self.tmpdir)
234: (8)             assert_(fp)
235: (8)             fp.close()
236: (8)             fp = datasource.open(local_file)
237: (8)             assert_(fp)
238: (8)             fp.close()
239: (0)         def test_del_attr_handling():
240: (4)             ds = datasource.DataSource()
241: (4)             del ds._istmpdest
242: (4)             ds.__del__()

-----

```

## File 252 - test\_iotools.py:

```

1: (0)             import time
2: (0)             from datetime import date
3: (0)             import numpy as np
4: (0)             from numpy.testing import (
5: (4)                 assert_, assert_equal, assert_allclose, assert_raises,
6: (4)             )
7: (0)             from numpy.lib._iotools import (
8: (4)                 LineSplitter, NameValidator, StringConverter,
9: (4)                 has_nested_fields, easy_dtype, flatten_dtype
10: (4)             )
11: (0)         class TestLineSplitter:
12: (4)             "Tests the LineSplitter class."
13: (4)             def test_no_delimiter(self):
14: (8)                 "Test LineSplitter w/o delimiter"
15: (8)                 strg = " 1 2 3 4 5 # test"
16: (8)                 test = LineSplitter()(strg)
17: (8)                 assert_equal(test, ['1', '2', '3', '4', '5'])
18: (8)                 test = LineSplitter('')(strg)
19: (8)                 assert_equal(test, ['1', '2', '3', '4', '5'])
20: (4)             def test_space_delimiter(self):
21: (8)                 "Test space delimiter"
22: (8)                 strg = " 1 2 3 4 5 # test"
23: (8)                 test = LineSplitter(' ')(strg)
24: (8)                 assert_equal(test, ['1', '2', '3', '4', '', '5'])
25: (8)                 test = LineSplitter('  ')(strg)
26: (8)                 assert_equal(test, ['1 2 3 4', '5'])
27: (4)             def test_tab_delimiter(self):
28: (8)                 "Test tab delimiter"
29: (8)                 strg = " 1\t2\t3\t4\t5\t6"
30: (8)                 test = LineSplitter('\t')(strg)
31: (8)                 assert_equal(test, ['1', '2', '3', '4', '5\t6'])
32: (8)                 strg = " 1 2\t3 4\t5 6"
33: (8)                 test = LineSplitter('\t')(strg)
34: (8)                 assert_equal(test, ['1 2', '3 4', '5 6'])
35: (4)             def test_other_delimiter(self):
36: (8)                 "Test LineSplitter on delimiter"
37: (8)                 strg = "1,2,3,4,,5"
38: (8)                 test = LineSplitter(',')(strg)
39: (8)                 assert_equal(test, ['1', '2', '3', '4', '', '5'])
40: (8)                 strg = " 1,2,3,4,,5 # test"
41: (8)                 test = LineSplitter(',')(strg)
42: (8)                 assert_equal(test, ['1', '2', '3', '4', '', '5'])
43: (8)                 strg = b" 1,2,3,4,,5 % test"
44: (8)                 test = LineSplitter(delimiter=b',', comments=b'%')(strg)
45: (8)                 assert_equal(test, ['1', '2', '3', '4', '', '5'])
46: (4)             def test_constant_fixed_width(self):
47: (8)                 "Test LineSplitter w/ fixed-width fields"
48: (8)                 strg = " 1 2 3 4 5 # test"
49: (8)                 test = LineSplitter(3)(strg)

```

```

50: (8)             assert_equal(test, ['1', '2', '3', '4', '', '5', ''])
51: (8)             strg = " 1   3 4 5 6# test"
52: (8)             test = LineSplitter(20)(strg)
53: (8)             assert_equal(test, ['1   3 4 5 6'])
54: (8)             strg = " 1   3 4 5 6# test"
55: (8)             test = LineSplitter(30)(strg)
56: (8)             assert_equal(test, ['1   3 4 5 6'])
57: (4) def test_variable_fixed_width(self):
58: (8)             strg = " 1   3 4 5 6# test"
59: (8)             test = LineSplitter((3, 6, 6, 3))(strg)
60: (8)             assert_equal(test, ['1', '3', '4 5', '6'])
61: (8)             strg = " 1   3 4 5 6# test"
62: (8)             test = LineSplitter((6, 6, 9))(strg)
63: (8)             assert_equal(test, ['1', '3 4', '5 6'])
64: (0) class TestNameValidator:
65: (4)     def test_case_sensitivity(self):
66: (8)         "Test case sensitivity"
67: (8)         names = ['A', 'a', 'b', 'c']
68: (8)         test = NameValidator().validate(names)
69: (8)         assert_equal(test, ['A', 'a', 'b', 'c'])
70: (8)         test = NameValidator(case_sensitive=False).validate(names)
71: (8)         assert_equal(test, ['A', 'A_1', 'B', 'C'])
72: (8)         test = NameValidator(case_sensitive='upper').validate(names)
73: (8)         assert_equal(test, ['A', 'A_1', 'B', 'C'])
74: (8)         test = NameValidator(case_sensitive='lower').validate(names)
75: (8)         assert_equal(test, ['a', 'a_1', 'b', 'c'])
76: (8)         assert_raises(ValueError, NameValidator, case_sensitive='foobar')
77: (4)     def test_excludelist(self):
78: (8)         "Test excludelist"
79: (8)         names = ['dates', 'data', 'Other Data', 'mask']
80: (8)         validator = NameValidator(excludelist=['dates', 'data', 'mask'])
81: (8)         test = validator.validate(names)
82: (8)         assert_equal(test, ['dates_', 'data_', 'Other_Data', 'mask_'])
83: (4)     def test_missing_names(self):
84: (8)         "Test validate missing names"
85: (8)         namelist = ('a', 'b', 'c')
86: (8)         validator = NameValidator()
87: (8)         assert_equal(validator(namelist), ['a', 'b', 'c'])
88: (8)         namelist = ('', 'b', 'c')
89: (8)         assert_equal(validator(namelist), ['f0', 'b', 'c'])
90: (8)         namelist = ('a', 'b', '')
91: (8)         assert_equal(validator(namelist), ['a', 'b', 'f0'])
92: (8)         namelist = ('', 'f0', '')
93: (8)         assert_equal(validator(namelist), ['f1', 'f0', 'f2'])
94: (4)     def test_validate_nb_names(self):
95: (8)         "Test validate nb names"
96: (8)         namelist = ('a', 'b', 'c')
97: (8)         validator = NameValidator()
98: (8)         assert_equal(validator(namelist, nbfields=1), ('a',))
99: (8)         assert_equal(validator(namelist, nbfields=5, defaultfmt="g%i"),
100: (21)           ['a', 'b', 'c', 'g0', 'g1'])
101: (4)     def test_validate_wo_names(self):
102: (8)         "Test validate no names"
103: (8)         namelist = None
104: (8)         validator = NameValidator()
105: (8)         assert_(validator(namelist) is None)
106: (8)         assert_equal(validator(namelist, nbfields=3), ['f0', 'f1', 'f2'])
107: (0)     def _bytes_to_date(s):
108: (4)         return date(*time.strptime(s, "%Y-%m-%d")[:3])
109: (0) class TestStringConverter:
110: (4)     "Test StringConverter"
111: (4)     def test_creation(self):
112: (8)         "Test creation of a StringConverter"
113: (8)         converter = StringConverter(int, -99999)
114: (8)         assert_equal(converter._status, 1)
115: (8)         assert_equal(converter.default, -99999)
116: (4)     def test_upgrade(self):
117: (8)         "Tests the upgrade method."
118: (8)         converter = StringConverter()

```

```

119: (8) assert_equal(converter._status, 0)
120: (8) assert_equal(converter.upgrade('0'), 0)
121: (8) assert_equal(converter._status, 1)
122: (8) import numpy.core.numeric as nx
123: (8) status_offset = int(nx.dtype(nx.int_).itemsize <
nx.dtype(nx.int64).itemsize)
124: (8) assert_equal(converter.upgrade('17179869184'), 17179869184)
125: (8) assert_equal(converter._status, 1 + status_offset)
126: (8) assert_allclose(converter.upgrade('0.'), 0.0)
127: (8) assert_equal(converter._status, 2 + status_offset)
128: (8) assert_equal(converter.upgrade('0j'), complex('0j'))
129: (8) assert_equal(converter._status, 3 + status_offset)
130: (8) for s in ['a', b'a']:
131: (12)     res = converter.upgrade(s)
132: (12)     assert_(type(res) is str)
133: (12)     assert_equal(res, 'a')
134: (12)     assert_equal(converter._status, 8 + status_offset)
135: (4) def test_missing(self):
136: (8)     "Tests the use of missing values."
137: (8)     converter = StringConverter(missing_values=('missing',
138: (52)                           'missed'))
139: (8)     converter.upgrade('0')
140: (8)     assert_equal(converter('0'), 0)
141: (8)     assert_equal(converter(''), converter.default)
142: (8)     assert_equal(converter('missing'), converter.default)
143: (8)     assert_equal(converter('missed'), converter.default)
144: (8)     try:
145: (12)         converter('miss')
146: (8)     except ValueError:
147: (12)         pass
148: (4) def test_upgrademapper(self):
149: (8)     "Tests upgrademapper"
150: (8)     dateparser = _bytes_to_date
151: (8)     _original_mapper = StringConverter._mapper[:]
152: (8)     try:
153: (12)         StringConverter.upgrade_mapper(dateparser, date(2000, 1, 1))
154: (12)         convert = StringConverter(dateparser, date(2000, 1, 1))
155: (12)         test = convert('2001-01-01')
156: (12)         assert_equal(test, date(2001, 1, 1))
157: (12)         test = convert('2009-01-01')
158: (12)         assert_equal(test, date(2009, 1, 1))
159: (12)         test = convert('')
160: (12)         assert_equal(test, date(2000, 1, 1))
161: (8)     finally:
162: (12)         StringConverter._mapper = _original_mapper
163: (4) def test_string_to_object(self):
164: (8)     "Make sure that string-to-object functions are properly recognized"
165: (8)     old_mapper = StringConverter._mapper[:] # copy of list
166: (8)     conv = StringConverter(_bytes_to_date)
167: (8)     assert_equal(conv._mapper, old_mapper)
168: (8)     assert_(hasattr(conv, 'default'))
169: (4) def test_keep_default(self):
170: (8)     "Make sure we don't lose an explicit default"
171: (8)     converter = StringConverter(None, missing_values='',
172: (36)                           default=-999)
173: (8)     converter.upgrade('3.14159265')
174: (8)     assert_equal(converter.default, -999)
175: (8)     assert_equal(converter.type, np.dtype(float))
176: (8)     converter = StringConverter(
177: (12)         None, missing_values='', default=0)
178: (8)     converter.upgrade('3.14159265')
179: (8)     assert_equal(converter.default, 0)
180: (8)     assert_equal(converter.type, np.dtype(float))
181: (4) def test_keep_default_zero(self):
182: (8)     "Check that we don't lose a default of 0"
183: (8)     converter = StringConverter(int, default=0,
184: (36)                           missing_values="N/A")
185: (8)     assert_equal(converter.default, 0)
186: (4) def test_keep_missing_values(self):

```

```

187: (8)           "Check that we're not losing missing values"
188: (8)           converter = StringConverter(int, default=0,
189: (36)             missing_values="N/A")
190: (8)
191: (12)           assert_equal(
192: (4)             converter.missing_values, {'', 'N/A'})
193: (8)           def test_int64_dtype(self):
194: (8)             "Check that int64 integer types can be specified"
195: (8)             converter = StringConverter(np.int64, default=0)
196: (8)             val = "-9223372036854775807"
197: (8)             assert_(converter(val)) == -9223372036854775807
198: (8)             val = "9223372036854775807"
199: (8)             assert_(converter(val)) == 9223372036854775807
200: (4)           def test_uint64_dtype(self):
201: (8)             "Check that uint64 integer types can be specified"
202: (8)             converter = StringConverter(np.uint64, default=0)
203: (8)             val = "9223372043271415339"
204: (0)             assert_(converter(val)) == 9223372043271415339
205: (4)           class TestMiscFunctions:
206: (8)             def test_has_nested_dtype(self):
207: (8)               "Test has_nested_dtype"
208: (8)               ndtype = np.dtype(float)
209: (8)               assert_equal(has_nested_fields(ndtype), False)
210: (8)               ndtype = np.dtype([('A', '|S3'), ('B', float)])
211: (8)               assert_equal(has_nested_fields(ndtype), False)
212: (8)               ndtype = np.dtype([('A', int), ('B', [('BA', float), ('BB', '|S1')])])
213: (4)             assert_equal(has_nested_fields(ndtype), True)
214: (8)           def test_easy_dtype(self):
215: (8)             "Test ndarray on dtypes"
216: (8)             ndtype = float
217: (8)             assert_equal(easy_dtype(ndtype), np.dtype(float))
218: (8)             ndtype = "i4, f8"
219: (21)             assert_equal(easy_dtype(ndtype),
220: (8)                           np.dtype([('f0', 'i4'), ('f1', 'f8')]))
221: (21)             assert_equal(easy_dtype(ndtype, defaultfmt="field_%03i"),
222: (8)                           np.dtype([('field_000', 'i4'), ('field_001', 'f8')]))
223: (8)
224: (21)             ndtype = "i4, f8"
225: (8)             assert_equal(easy_dtype(ndtype, names="a, b"),
226: (8)                           np.dtype([('a', 'i4'), ('b', 'f8')]))
227: (21)             ndtype = "i4, f8"
228: (8)             assert_equal(easy_dtype(ndtype, names="a, b, c"),
229: (8)                           np.dtype([('a', 'i4'), ('b', 'f8')]))
230: (21)             ndtype = "i4, f8"
231: (8)             assert_equal(easy_dtype(ndtype, names="a", defaultfmt="f%02i"),
232: (21)                           np.dtype([('a', 'i4'), ('f00', 'f8')]))
233: (8)             ndtype = [('A', int), ('B', float)]
234: (8)             assert_equal(easy_dtype(ndtype), np.dtype([('A', int), ('B', float)]))
235: (8)
236: (21)             assert_equal(easy_dtype(ndtype, names="a,b"),
237: (8)                           np.dtype([('a', int), ('b', float)]))
238: (21)             assert_equal(easy_dtype(ndtype, names="a"), np.dtype([('a', int), ('f0', float)]))
239: (8)
240: (21)             assert_equal(easy_dtype(ndtype, names="a,b,c"),
241: (8)                           np.dtype([('a', int), ('b', float)]))
242: (8)             ndtype = (int, float, float)
243: (21)             assert_equal(easy_dtype(ndtype),
244: (8)                           np.dtype([('f0', int), ('f1', float), ('f2', float)]))
245: (8)             ndtype = (int, float, float)
246: (21)             assert_equal(easy_dtype(ndtype, names="a, b, c"),
247: (8)                           np.dtype([('a', int), ('b', float), ('c', float)]))
248: (8)
249: (21)             ndtype = np.dtype(float)
250: (8)
251: (8)             assert_equal(
252: (12)               easy_dtype(ndtype, names=[' ', ' ', ' '], defaultfmt="f%02i"),
253: (12)               np.dtype([(' ', float) for _ in ('a', 'b', 'c')]))
254: (4)           def test_flatten_dtype(self):
255: (8)             "Testing flatten_dtype"

```

```

256: (8)           dt = np.dtype([("a", "f8"), ("b", "f8")])
257: (8)           dt_flat = flatten_dtype(dt)
258: (8)           assert_equal(dt_flat, [float, float])
259: (8)           dt = np.dtype([("a", [( "aa", '|S1'), ("ab", '|S2')]), ("b", int)])
260: (8)           dt_flat = flatten_dtype(dt)
261: (8)           assert_equal(dt_flat, [np.dtype('|S1'), np.dtype('|S2'), int])
262: (8)           dt = np.dtype([( "a", (float, 2)), ("b", (int, 3))])
263: (8)           dt_flat = flatten_dtype(dt)
264: (8)           assert_equal(dt_flat, [float, int])
265: (8)           dt_flat = flatten_dtype(dt, True)
266: (8)           assert_equal(dt_flat, [float] * 2 + [int] * 3)
267: (8)           dt = np.dtype([( ("a", "A"), "f8"), (( "b", "B"), "f8")])
268: (8)           dt_flat = flatten_dtype(dt)
269: (8)           assert_equal(dt_flat, [float, float])

```

-----

## File 253 - test\_version.py:

```

1: (0)           """Tests for the NumpyVersion class.
2: (0)           """
3: (0)           from numpy.testing import assert_, assert_raises
4: (0)           from numpy.lib import NumpyVersion
5: (0)           def test_main_versions():
6: (4)             assert_(NumpyVersion('1.8.0') == '1.8.0')
7: (4)             for ver in ['1.9.0', '2.0.0', '1.8.1', '10.0.1']:
8: (8)               assert_(NumpyVersion('1.8.0') < ver)
9: (4)               for ver in ['1.7.0', '1.7.1', '0.9.9']:
10: (8)                 assert_(NumpyVersion('1.8.0') > ver)
11: (0)           def test_version_1_point_10():
12: (4)             assert_(NumpyVersion('1.9.0') < '1.10.0')
13: (4)             assert_(NumpyVersion('1.11.0') < '1.11.1')
14: (4)             assert_(NumpyVersion('1.11.0') == '1.11.0')
15: (4)             assert_(NumpyVersion('1.99.11') < '1.99.12')
16: (0)           def test_alpha_beta_rc():
17: (4)             assert_(NumpyVersion('1.8.0rc1') == '1.8.0rc1')
18: (4)             for ver in ['1.8.0', '1.8.0rc2']:
19: (8)               assert_(NumpyVersion('1.8.0rc1') < ver)
20: (4)               for ver in ['1.8.0a2', '1.8.0b3', '1.7.2rc4']:
21: (8)                 assert_(NumpyVersion('1.8.0rc1') > ver)
22: (4)                 assert_(NumpyVersion('1.8.0b1') > '1.8.0a2')
23: (0)           def test_dev_version():
24: (4)             assert_(NumpyVersion('1.9.0.dev-Unknown') < '1.9.0')
25: (4)             for ver in ['1.9.0', '1.9.0a1', '1.9.0b2', '1.9.0b2.dev-ffffffff']:
26: (8)               assert_(NumpyVersion('1.9.0.dev-f16acvda') < ver)
27: (4)               assert_(NumpyVersion('1.9.0.dev-f16acvda') == '1.9.0.dev-11111111')
28: (0)           def test_dev_a_b_rc_mixed():
29: (4)             assert_(NumpyVersion('1.9.0a2.dev-f16acvda') == '1.9.0a2.dev-11111111')
30: (4)             assert_(NumpyVersion('1.9.0a2.dev-6acvda54') < '1.9.0a2')
31: (0)           def test_dev0_version():
32: (4)             assert_(NumpyVersion('1.9.0.dev0+Unknown') < '1.9.0')
33: (4)             for ver in ['1.9.0', '1.9.0a1', '1.9.0b2', '1.9.0b2.dev0+ffffffffff']:
34: (8)               assert_(NumpyVersion('1.9.0.dev0+f16acvda') < ver)
35: (4)               assert_(NumpyVersion('1.9.0.dev0+f16acvda') == '1.9.0.dev0+11111111')
36: (0)           def test_dev0_a_b_rc_mixed():
37: (4)             assert_(NumpyVersion('1.9.0a2.dev0+f16acvda') == '1.9.0a2.dev0+11111111')
38: (4)             assert_(NumpyVersion('1.9.0a2.dev0+6acvda54') < '1.9.0a2')
39: (0)           def test_raises():
40: (4)             for ver in ['1.9', '1,9.0', '1.7.x']:
41: (8)               assert_raises(ValueError, NumpyVersion, ver)

```

-----

## File 254 - \_\_init\_\_.py:

```
1: (0)
```

-----

## File 255 - linalg.py:

```

1: (0)         """Lite version of scipy.linalg.
2: (0)         Notes
3: (0)         -----
4: (0)         This module is a lite version of the linalg.py module in SciPy which
5: (0)         contains high-level Python interface to the LAPACK library. The lite
6: (0)         version only accesses the following LAPACK functions: dgesv, zgesv,
7: (0)         dgeev, zgeev, dgesdd, zgesdd, dgelsd, zgelsd, dsyevd, zheevd, dgetrf,
8: (0)         zgetrf, dpotrf, zpotrf, dgeqrf, zgeqrf, zungqr, dorgqr.
9: (0)
10: (0)        """
11: (11)       __all__ = ['matrix_power', 'solve', 'tensorsolve', 'tensorinv', 'inv',
12: (11)             'cholesky', 'eigvals', 'eigvalsh', 'pinv', 'slogdet', 'det',
13: (11)             'svd', 'eig', 'eigh', 'lstsq', 'norm', 'qr', 'cond', 'matrix_rank',
14: (11)             'LinAlgError', 'multi_dot']
15: (0)       import functools
16: (0)       import operator
17: (0)       import warnings
18: (0)       from typing import NamedTuple, Any
19: (0)       from .._utils import set_module
20: (0)       from numpy.core import (
21: (4)           array, asarray, zeros, empty, empty_like, intc, single, double,
22: (4)           csingle, cdouble, inexact, complexfloating, newaxis, all, Inf, dot,
23: (4)           add, multiply, sqrt, sum, isfinite,
24: (4)           finfo, errstate, geterrobj, moveaxis, amin, amax, prod, abs,
25: (4)           atleast_2d, intp, asanyarray, object_, matmul,
26: (4)           swapaxes, divide, count_nonzero, isnan, sign, argsort, sort,
27: (4)           reciprocal
28: (0)       )
29: (0)       from numpy.core.multiarray import normalize_axis_index
30: (0)       from numpy.core import overrides
31: (0)       from numpy.lib.twodim_base import triu, eye
32: (0)       from numpy.linalg import _umath_linalg
33: (0)       from numpy._typing import NDArray
34: (0)       class EigResult(NamedTuple):
35: (4)           eigenvalues: NDArray[Any]
36: (4)           eigenvectors: NDArray[Any]
37: (0)       class EighResult(NamedTuple):
38: (4)           eigenvalues: NDArray[Any]
39: (4)           eigenvectors: NDArray[Any]
40: (0)       class QRResult(NamedTuple):
41: (4)           Q: NDArray[Any]
42: (4)           R: NDArray[Any]
43: (0)       class SlogdetResult(NamedTuple):
44: (4)           sign: NDArray[Any]
45: (4)           logabsdet: NDArray[Any]
46: (0)       class SVDResult(NamedTuple):
47: (4)           U: NDArray[Any]
48: (4)           S: NDArray[Any]
49: (4)           Vh: NDArray[Any]
50: (0)       array_function_dispatch = functools.partial(
51: (4)           overrides.array_function_dispatch, module='numpy.linalg')
52: (0)       fortran_int = intc
53: (0)       @set_module('numpy.linalg')
54: (0)       class LinAlgError(ValueError):
55: (4)           """
56: (4)               Generic Python-exception-derived object raised by linalg functions.
57: (4)               General purpose exception class, derived from Python's ValueError
58: (4)               class, programmatically raised in linalg functions when a Linear
59: (4)               Algebra-related condition would prevent further correct execution of the
60: (4)               function.
61: (4)               Parameters
62: (4)               -----
63: (4)               None
64: (4)               Examples
65: (4)               -----
66: (4)               >>> from numpy import linalg as LA
67: (4)               >>> LA.inv(np.zeros((2,2)))

```

Traceback (most recent call last):

```

68: (6)          File "<stdin>", line 1, in <module>
69: (6)          File "...linalg.py", line 350,
70: (8)            in inv return wrap(solve(a, identity(a.shape[0], dtype=a.dtype)))
71: (6)          File "...linalg.py", line 249,
72: (8)            in solve
73: (8)              raise LinAlgError('Singular matrix')
74: (4)      numpy.linalg.LinAlgError: Singular matrix
75: (4)
76: (0)      def _determine_error_states():
77: (4)          errobj = geterrobj()
78: (4)          bufsize = errobj[0]
79: (4)          with errstate(invalid='call', over='ignore',
80: (18)              divide='ignore', under='ignore'):
81: (8)              invalid_call_errmask = geterrobj()[1]
82: (4)          return [bufsize, invalid_call_errmask, None]
83: (0)      _linalg_error_extobj = _determine_error_states()
84: (0)      del _determine_error_states
85: (0)      def _raise_linalgerror_singular(err, flag):
86: (4)          raise LinAlgError("Singular matrix")
87: (0)      def _raise_linalgerror_nonposdef(err, flag):
88: (4)          raise LinAlgError("Matrix is not positive definite")
89: (0)      def _raise_linalgerror_eigenvalues_nonconvergence(err, flag):
90: (4)          raise LinAlgError("Eigenvalues did not converge")
91: (0)      def _raise_linalgerror_svd_nonconvergence(err, flag):
92: (4)          raise LinAlgError("SVD did not converge")
93: (0)      def _raise_linalgerror_lstsq(err, flag):
94: (4)          raise LinAlgError("SVD did not converge in Linear Least Squares")
95: (0)      def _raise_linalgerror_qr(err, flag):
96: (4)          raise LinAlgError("Incorrect argument found while performing "
97: (22)              "QR factorization")
98: (0)      def get_linalg_error_extobj(callback):
99: (4)          extobj = list(_linalg_error_extobj) # make a copy
100: (4)         extobj[2] = callback
101: (4)         return extobj
102: (0)      def _makearray(a):
103: (4)          new = asarray(a)
104: (4)          wrap = getattr(a, "__array_prepare__", new.__array_wrap__)
105: (4)          return new, wrap
106: (0)      def isComplexType(t):
107: (4)          return issubclass(t, complexfloating)
108: (0)      _real_types_map = {single : single,
109: (19)          double : double,
110: (19)          csingle : single,
111: (19)          cdouble : double}
112: (0)      _complex_types_map = {single : csingle,
113: (22)          double : cdouble,
114: (22)          csingle : csingle,
115: (22)          cdouble : cdouble}
116: (0)      def _realType(t, default=double):
117: (4)          return _real_types_map.get(t, default)
118: (0)      def _complexType(t, default=cdouble):
119: (4)          return _complex_types_map.get(t, default)
120: (0)      def _commonType(*arrays):
121: (4)          result_type = single
122: (4)          is_complex = False
123: (4)          for a in arrays:
124: (8)              type_ = a.dtype.type
125: (8)              if issubclass(type_, inexact):
126: (12)                  if isComplexType(type_):
127: (16)                      is_complex = True
128: (12)                  rt = _realType(type_, default=None)
129: (12)                  if rt is double:
130: (16)                      result_type = double
131: (12)                  elif rt is None:
132: (16)                      raise TypeError("array type %s is unsupported in linalg" %
133: (24)                          (a.dtype.name,))
134: (8)
135: (12)                  result_type = double
136: (4)          if is_complex:

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

137: (8)             result_type = _complex_types_map[result_type]
138: (8)             return cdouble, result_type
139: (4)         else:
140: (8)             return double, result_type
141: (0)     def _to_native_byte_order(*arrays):
142: (4)         ret = []
143: (4)         for arr in arrays:
144: (8)             if arr.dtype.byteorder not in ('=', '|'):
145: (12)                 ret.append(asarray(arr, dtype=arr.dtype.newbyteorder('=')))
146: (8)             else:
147: (12)                 ret.append(arr)
148: (4)         if len(ret) == 1:
149: (8)             return ret[0]
150: (4)         else:
151: (8)             return ret
152: (0)     def _assert_2d(*arrays):
153: (4)         for a in arrays:
154: (8)             if a.ndim != 2:
155: (12)                 raise LinAlgError('%d-dimensional array given. Array must be '
156: (20)                               'two-dimensional' % a.ndim)
157: (0)     def _assert_stacked_2d(*arrays):
158: (4)         for a in arrays:
159: (8)             if a.ndim < 2:
160: (12)                 raise LinAlgError('%d-dimensional array given. Array must be '
161: (20)                               'at least two-dimensional' % a.ndim)
162: (0)     def _assert_stacked_square(*arrays):
163: (4)         for a in arrays:
164: (8)             m, n = a.shape[-2:]
165: (8)             if m != n:
166: (12)                 raise LinAlgError('Last 2 dimensions of the array must be square')
167: (0)     def _assert_finite(*arrays):
168: (4)         for a in arrays:
169: (8)             if not isfinite(a).all():
170: (12)                 raise LinAlgError("Array must not contain infs or NaNs")
171: (0)     def _is_empty_2d(arr):
172: (4)         return arr.size == 0 and prod(arr.shape[-2:]) == 0
173: (0)     def transpose(a):
174: (4)         """
175: (4)             Transpose each matrix in a stack of matrices.
176: (4)             Unlike np.transpose, this only swaps the last two axes, rather than all of
177: (4)             them
178: (4)             Parameters
179: (4)             -----
180: (4)             a : (... , M, N) array_like
181: (4)             Returns
182: (4)             -----
183: (4)             aT : (... , N, M) ndarray
184: (4)             """
185: (4)             return swapaxes(a, -1, -2)
186: (0)     def _tensorsolve_dispatcher(a, b, axes=None):
187: (4)         return (a, b)
188: (0)     @array_function_dispatch(_tensorsolve_dispatcher)
189: (0)     def tensorsolve(a, b, axes=None):
190: (4)         """
191: (4)             Solve the tensor equation ``a x = b`` for x.
192: (4)             It is assumed that all indices of `x` are summed over in the product,
193: (4)             together with the rightmost indices of `a`, as is done in, for example,
194: (4)             ``tensordot(a, x, axes=x.ndim)``.
195: (4)             Parameters
196: (4)             -----
197: (4)             a : array_like
198: (8)                 Coefficient tensor, of shape ``b.shape + Q``. `Q`, a tuple, equals
199: (8)                 the shape of that sub-tensor of `a` consisting of the appropriate
200: (8)                 number of its rightmost indices, and must be such that
201: (8)                 ``prod(Q) == prod(b.shape)`` (in which sense `a` is said to be
202: (8)                   'square').
203: (4)             b : array_like
204: (8)                 Right-hand tensor, which can be of any shape.
205: (4)             axes : tuple of ints, optional

```

```

206: (8)           Axes in `a` to reorder to the right, before inversion.
207: (8)           If None (default), no reordering is done.
208: (4)           Returns
209: (4)
210: (4)           -----
211: (4)           x : ndarray, shape Q
212: (4)           Raises
213: (4)
214: (4)           LinAlgError
215: (4)           If `a` is singular or not 'square' (in the above sense).
216: (4)           See Also
217: (4)           -----
218: (4)           numpy.tensordot, tensorinv, numpy.einsum
219: (4)           Examples
220: (4)
221: (4)           >>> a = np.eye(2*3*4)
222: (4)           >>> a.shape = (2*3, 4, 2, 3, 4)
223: (4)           >>> b = np.random.randn(2*3, 4)
224: (4)           >>> x = np.linalg.tensorsolve(a, b)
225: (4)           >>> x.shape
226: (4)           (2, 3, 4)
227: (4)           >>> np.allclose(np.tensordot(a, x, axes=3), b)
228: (4)           True
229: (4)           """
230: (4)           a, wrap = _makearray(a)
231: (4)           b = asarray(b)
232: (4)           an = a.ndim
233: (8)           if axes is not None:
234: (8)               allaxes = list(range(0, an))
235: (12)             for k in axes:
236: (12)                 allaxes.remove(k)
237: (8)                 allaxes.insert(an, k)
238: (4)                 a = a.transpose(allaxes)
239: (4)           oldshape = a.shape[-(an-b.ndim):]
240: (4)           prod = 1
241: (8)           for k in oldshape:
242: (8)               prod *= k
243: (8)           if a.size != prod ** 2:
244: (12)             raise LinAlgError(
245: (12)               "Input arrays must satisfy the requirement \
246: (8)                 prod(a.shape[b.ndim:]) == prod(a.shape[:b.ndim])"
247: (4)             )
248: (4)           a = a.reshape(prod, prod)
249: (4)           b = b.ravel()
250: (4)           res = wrap(solve(a, b))
251: (4)           res.shape = oldshape
252: (0)           return res
253: (4)           def _solve_dispatcher(a, b):
254: (0)               return (a, b)
255: (0)           @array_function_dispatch(_solve_dispatcher)
256: (4)           def solve(a, b):
257: (4)               """
258: (4)               Solve a linear matrix equation, or system of linear scalar equations.
259: (4)               Computes the "exact" solution, `x`, of the well-determined, i.e., full
260: (4)               rank, linear matrix equation `ax = b`.
261: (4)               Parameters
262: (4)               -----
263: (8)               a : (... , M, M) array_like
264: (8)                   Coefficient matrix.
265: (4)               b : {(... , M,) , (... , M, K)}, array_like
266: (8)                   Ordinate or "dependent variable" values.
267: (4)               Returns
268: (4)               -----
269: (8)               x : {(... , M,) , (... , M, K)} ndarray
270: (8)                   Solution to the system a x = b. Returned shape is identical to `b`.
271: (4)               Raises
272: (4)
273: (8)               LinAlgError
274: (8)               If `a` is singular or not square.

```

See Also

```

275: (4)      -----
276: (4)      scipy.linalg.solve : Similar function in SciPy.
277: (4)      Notes
278: (4)      -----
279: (4)      .. versionadded:: 1.8.0
280: (4)      Broadcasting rules apply, see the `numpy.linalg` documentation for
281: (4)      details.
282: (4)      The solutions are computed using LAPACK routine ``_gesv``.
283: (4)      `a` must be square and of full-rank, i.e., all rows (or, equivalently,
284: (4)      columns) must be linearly independent; if either is not true, use
285: (4)      `lstsq` for the least-squares best "solution" of the
286: (4)      system/equation.
287: (4)      References
288: (4)      -----
289: (4)      .. [1] G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando,
290: (11)          FL, Academic Press, Inc., 1980, pg. 22.
291: (4)      Examples
292: (4)      -----
293: (4)      Solve the system of equations ``x0 + 2 * x1 = 1`` and ``3 * x0 + 5 * x1 =
294: (4)      2``:
295: (4)      >>> a = np.array([[1, 2], [3, 5]])
296: (4)      >>> b = np.array([1, 2])
297: (4)      >>> x = np.linalg.solve(a, b)
298: (4)      >>> x
299: (4)      array([-1.,  1.])
300: (4)      Check that the solution is correct:
301: (4)      >>> np.allclose(np.dot(a, x), b)
302: (4)      True
303: (4)      """
304: (4)      a, _ = _makearray(a)
305: (4)      _assert_stacked_2d(a)
306: (4)      _assert_stacked_square(a)
307: (4)      b, wrap = _makearray(b)
308: (4)      t, result_t = _commonType(a, b)
309: (8)      if b.ndim == a.ndim - 1:
310: (4)          gufunc = _umath_linalg.solve1
311: (8)      else:
312: (4)          gufunc = _umath_linalg.solve
313: (4)      signature = 'DD->D' if isComplexType(t) else 'dd->d'
314: (4)      extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
315: (4)      r = gufunc(a, b, signature=signature, extobj=extobj)
316: (0)      return wrap(r.astype(result_t, copy=False))
317: (4)      def _tensorinv_dispatcher(a, ind=None):
318: (0)          return (a,)
319: (0)      @array_function_dispatch(_tensorinv_dispatcher)
320: (4)      def tensorinv(a, ind=2):
321: (4)          """
322: (4)          Compute the 'inverse' of an N-dimensional array.
323: (4)          The result is an inverse for `a` relative to the tensordot operation
324: (4)          ``tensordot(a, b, ind)``, i. e., up to floating-point accuracy,
325: (4)          ``tensordot(tensorinv(a), a, ind)`` is the "identity" tensor for the
326: (4)          tensordot operation.
327: (4)          Parameters
328: (4)          -----
329: (8)          a : array_like
330: (8)              Tensor to 'invert'. Its shape must be 'square', i. e.,
331: (4)              ``prod(a.shape[:ind]) == prod(a.shape[ind:])``.
332: (8)          ind : int, optional
333: (8)              Number of first indices that are involved in the inverse sum.
334: (4)              Must be a positive integer, default is 2.
335: (4)          Returns
336: (4)          -----
337: (8)          b : ndarray
338: (4)              `a`'s tensordot inverse, shape ``a.shape[ind:] + a.shape[:ind]``.
339: (4)          Raises
340: (4)          -----
341: (8)          LinAlgError
342: (4)              If `a` is singular or not 'square' (in the above sense).

```

See Also

```

343: (4)      -----
344: (4)      numpy.tensordot, tensorsolve
345: (4)      Examples
346: (4)      -----
347: (4)      >>> a = np.eye(4*6)
348: (4)      >>> a.shape = (4, 6, 8, 3)
349: (4)      >>> ainv = np.linalg.tensorinv(a, ind=2)
350: (4)      >>> ainv.shape
351: (4)      (8, 3, 4, 6)
352: (4)      >>> b = np.random.randn(4, 6)
353: (4)      >>> np.allclose(np.tensordot(ainv, b), np.linalg.tensorsolve(a, b))
354: (4)      True
355: (4)      >>> a = np.eye(4*6)
356: (4)      >>> a.shape = (24, 8, 3)
357: (4)      >>> ainv = np.linalg.tensorinv(a, ind=1)
358: (4)      >>> ainv.shape
359: (4)      (8, 3, 24)
360: (4)      >>> b = np.random.randn(24)
361: (4)      >>> np.allclose(np.tensordot(ainv, b, 1), np.linalg.tensorsolve(a, b))
362: (4)      True
363: (4)      """
364: (4)      a = asarray(a)
365: (4)      oldshape = a.shape
366: (4)      prod = 1
367: (4)      if ind > 0:
368: (8)          invshape = oldshape[ind:] + oldshape[:ind]
369: (8)          for k in oldshape[ind:]:
370: (12)              prod *= k
371: (4)      else:
372: (8)          raise ValueError("Invalid ind argument.")
373: (4)      a = a.reshape(prod, -1)
374: (4)      ia = inv(a)
375: (4)      return ia.reshape(*invshape)
376: (0)      def _unary_dispatcher(a):
377: (4)          return (a,)
378: (0)      @array_function_dispatch(_unary_dispatcher)
379: (0)      def inv(a):
380: (4)          """
381: (4)          Compute the (multiplicative) inverse of a matrix.
382: (4)          Given a square matrix `a`, return the matrix `ainv` satisfying
383: (4)          ``dot(a, ainv) = dot(ainv, a) = eye(a.shape[0])``.
384: (4)          Parameters
385: (4)          -----
386: (4)          a : (... , M, M) array_like
387: (8)              Matrix to be inverted.
388: (4)          Returns
389: (4)          -----
390: (4)          ainv : (... , M, M) ndarray or matrix
391: (8)              (Multiplicative) inverse of the matrix `a`.
392: (4)          Raises
393: (4)          -----
394: (4)          LinAlgError
395: (8)              If `a` is not square or inversion fails.
396: (4)          See Also
397: (4)          -----
398: (4)          scipy.linalg.inv : Similar function in SciPy.
399: (4)          Notes
400: (4)          -----
401: (4)          .. versionadded:: 1.8.0
402: (4)          Broadcasting rules apply, see the `numpy.linalg` documentation for
403: (4)          details.
404: (4)          Examples
405: (4)          -----
406: (4)          >>> from numpy.linalg import inv
407: (4)          >>> a = np.array([[1., 2.], [3., 4.]])
408: (4)          >>> ainv = inv(a)
409: (4)          >>> np.allclose(np.dot(a, ainv), np.eye(2))
410: (4)          True
411: (4)          >>> np.allclose(np.dot(ainv, a), np.eye(2))

```

```

412: (4)          True
413: (4)          If a is a matrix object, then the return value is a matrix as well:
414: (4)          >>> ainv = inv(np.matrix(a))
415: (4)          >>> ainv
416: (4)          matrix([[-2. ,  1. ],
417: (12)             [ 1.5, -0.5]])
418: (4)          Inverses of several matrices can be computed at once:
419: (4)          >>> a = np.array([[1., 2.], [3., 4.]], [[1, 3], [3, 5]]])
420: (4)          >>> inv(a)
421: (4)          array([[-2. ,  1. ],
422: (12)             [ 1.5 , -0.5 ]],
423: (11)             [[-1.25,  0.75],
424: (12)               [ 0.75, -0.25]]])
425: (4)          """
426: (4)          a, wrap = _makearray(a)
427: (4)          _assert_stacked_2d(a)
428: (4)          _assert_stacked_square(a)
429: (4)          t, result_t = _commonType(a)
430: (4)          signature = 'D->D' if isComplexType(t) else 'd->d'
431: (4)          extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
432: (4)          ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
433: (4)          return wrap(ainv.astype(result_t, copy=False))
434: (0)          def _matrix_power_dispatcher(a, n):
435: (4)              return (a,)
436: (0)          @array_function_dispatch(_matrix_power_dispatcher)
437: (0)          def matrix_power(a, n):
438: (4)              """
439: (4)              Raise a square matrix to the (integer) power `n`.
440: (4)              For positive integers `n`, the power is computed by repeated matrix
441: (4)              squarings and matrix multiplications. If ``n == 0``, the identity matrix
442: (4)              of the same shape as M is returned. If ``n < 0``, the inverse
443: (4)              is computed and then raised to the ``abs(n)``.
444: (4)              .. note:: Stacks of object matrices are not currently supported.
445: (4)              Parameters
446: (4)              -----
447: (4)              a : (..., M, M) array_like
448: (8)                  Matrix to be "powered".
449: (4)              n : int
450: (8)                  The exponent can be any integer or long integer, positive,
451: (8)                  negative, or zero.
452: (4)              Returns
453: (4)              -----
454: (4)              a**n : (..., M, M) ndarray or matrix object
455: (8)                  The return value is the same shape and type as `M`;
456: (8)                  if the exponent is positive or zero then the type of the
457: (8)                  elements is the same as those of `M`. If the exponent is
458: (8)                  negative the elements are floating-point.
459: (4)              Raises
460: (4)              -----
461: (4)              LinAlgError
462: (8)                  For matrices that are not square or that (for negative powers) cannot
463: (8)                  be inverted numerically.
464: (4)              Examples
465: (4)              -----
466: (4)              >>> from numpy.linalg import matrix_power
467: (4)              >>> i = np.array([[0, 1], [-1, 0]]) # matrix equiv. of the imaginary unit
468: (4)              >>> matrix_power(i, 3) # should = -i
469: (4)              array([[ 0, -1],
470: (11)                [ 1,  0]])
471: (4)              >>> matrix_power(i, 0)
472: (4)              array([[1, 0],
473: (11)                [0, 1]])
474: (4)              >>> matrix_power(i, -3) # should = 1/(-i) = i, but w/ f.p. elements
475: (4)              array([[ 0.,  1.],
476: (11)                [-1.,  0.]])
477: (4)              Somewhat more sophisticated example
478: (4)              >>> q = np.zeros((4, 4))
479: (4)              >>> q[0:2, 0:2] = -i
480: (4)              >>> q[2:4, 2:4] = i

```

```

481: (4)          >>> q # one of the three quaternion units not equal to 1
482: (4)          array([[ 0., -1.,  0.,  0.],
483: (11)           [ 1.,  0.,  0.,  0.],
484: (11)           [ 0.,  0.,  0.,  1.],
485: (11)           [ 0.,  0., -1.,  0.]])
486: (4)          >>> matrix_power(q, 2) # = -np.eye(4)
487: (4)          array([[[-1.,  0.,  0.,  0.],
488: (11)           [ 0., -1.,  0.,  0.],
489: (11)           [ 0.,  0., -1.,  0.],
490: (11)           [ 0.,  0.,  0., -1.]])
491: (4)          """
492: (4)          a = asanyarray(a)
493: (4)          _assert_stacked_2d(a)
494: (4)          _assert_stacked_square(a)
495: (4)          try:
496: (8)              n = operator.index(n)
497: (4)          except TypeError as e:
498: (8)              raise TypeError("exponent must be an integer") from e
499: (4)          if a.dtype != object:
500: (8)              fmatmul = matmul
501: (4)          elif a.ndim == 2:
502: (8)              fmatmul = dot
503: (4)          else:
504: (8)              raise NotImplementedError(
505: (12)                  "matrix_power not supported for stacks of object arrays")
506: (4)          if n == 0:
507: (8)              a = empty_like(a)
508: (8)              a[...] = eye(a.shape[-2], dtype=a.dtype)
509: (8)              return a
510: (4)          elif n < 0:
511: (8)              a = inv(a)
512: (8)              n = abs(n)
513: (4)          if n == 1:
514: (8)              return a
515: (4)          elif n == 2:
516: (8)              return fmatmul(a, a)
517: (4)          elif n == 3:
518: (8)              return fmatmul(fmatmul(a, a), a)
519: (4)          z = result = None
520: (4)          while n > 0:
521: (8)              z = a if z is None else fmatmul(z, z)
522: (8)              n, bit = divmod(n, 2)
523: (8)              if bit:
524: (12)                  result = z if result is None else fmatmul(result, z)
525: (4)          return result
526: (0)          @array_function_dispatch(_unary_dispatcher)
527: (0)          def cholesky(a):
528: (4)          """
529: (4)              Cholesky decomposition.
530: (4)              Return the Cholesky decomposition, `L * L.H`, of the square matrix `a`,
531: (4)              where `L` is lower-triangular and .H is the conjugate transpose operator
532: (4)              (which is the ordinary transpose if `a` is real-valued). `a` must be
533: (4)              Hermitian (symmetric if real-valued) and positive-definite. No
534: (4)              checking is performed to verify whether `a` is Hermitian or not.
535: (4)              In addition, only the lower-triangular and diagonal elements of `a`
536: (4)              are used. Only `L` is actually returned.
537: (4)              Parameters
538: (4)              -----
539: (4)              a : (... , M, M) array_like
540: (8)                  Hermitian (symmetric if all elements are real), positive-definite
541: (8)                  input matrix.
542: (4)              Returns
543: (4)              -----
544: (4)              L : (... , M, M) array_like
545: (8)                  Lower-triangular Cholesky factor of `a`. Returns a matrix object if
546: (8)                  `a` is a matrix object.
547: (4)              Raises
548: (4)              -----
549: (4)              LinAlgError

```

```

550: (7)           If the decomposition fails, for example, if `a` is not
551: (7)           positive-definite.
552: (4)           See Also
553: (4)
554: (4)           scipy.linalg.cholesky : Similar function in SciPy.
555: (4)           scipy.linalg.cholesky_banded : Cholesky decompose a banded Hermitian
556: (35)          positive-definite matrix.
557: (4)           scipy.linalg.cho_factor : Cholesky decomposition of a matrix, to use in
558: (30)          `scipy.linalg.cho_solve`.
559: (4)           Notes
560: (4)
561: (4)           .. versionadded:: 1.8.0
562: (4)           Broadcasting rules apply, see the `numpy.linalg` documentation for
563: (4)           details.
564: (4)           The Cholesky decomposition is often used as a fast way of solving
565: (4)           .. math:: A \mathbf{x} = \mathbf{b}
566: (4)           (when `A` is both Hermitian/symmetric and positive-definite).
567: (4)           First, we solve for :math:`\mathbf{y}` in
568: (4)           .. math:: L \mathbf{y} = \mathbf{b},
569: (4)           and then for :math:`\mathbf{x}` in
570: (4)           .. math:: L \mathbf{H} \mathbf{x} = \mathbf{y}.
571: (4)           Examples
572: (4)
573: (4)           >>> A = np.array([[1,-2j],[2j,5]])
574: (4)           >>> A
575: (4)           array([[ 1.+0.j, -0.-2.j],
576: (11)             [ 0.+2.j,  5.+0.j]])
577: (4)           >>> L = np.linalg.cholesky(A)
578: (4)           >>> L
579: (4)           array([[1.+0.j,  0.+0.j],
580: (11)             [0.+2.j,  1.+0.j]])
581: (4)           >>> np.dot(L, L.T.conj()) # verify that L * L.H = A
582: (4)           array([[1.+0.j,  0.-2.j],
583: (11)             [0.+2.j,  5.+0.j]])
584: (4)           >>> A = [[1,-2j],[2j,5]] # what happens if A is only array_like?
585: (4)           >>> np.linalg.cholesky(A) # an ndarray object is returned
586: (4)           array([[1.+0.j,  0.+0.j],
587: (11)             [0.+2.j,  1.+0.j]])
588: (4)           >>> # But a matrix object is returned if A is a matrix object
589: (4)           >>> np.linalg.cholesky(np.matrix(A))
590: (4)           matrix([[ 1.+0.j,  0.+0.j],
591: (12)             [ 0.+2.j,  1.+0.j]])
592: (4)
593: (4)           """
594: (4)           extobj = get_linalg_error_extobj(_raise_linalgerror_nonposdef)
595: (4)           gufunc = _umath_linalg.cholesky_lo
596: (4)           a, wrap = _makearray(a)
597: (4)           _assert_stacked_2d(a)
598: (4)           _assert_stacked_square(a)
599: (4)           t, result_t = _commonType(a)
600: (4)           signature = 'D->D' if isComplexType(t) else 'd->d'
601: (4)           r = gufunc(a, signature=signature, extobj=extobj)
602: (0)           return wrap(r.astype(result_t, copy=False))
603: (4)           def _qr_dispatcher(a, mode=None):
604: (0)               return (a,)
605: (0)           @array_function_dispatch(_qr_dispatcher)
606: (4)           def qr(a, mode='reduced'):
607: (4)               """
608: (4)               Compute the qr factorization of a matrix.
609: (4)               Factor the matrix `a` as *qr*, where `q` is orthonormal and `r` is
610: (4)               upper-triangular.
611: (4)               Parameters
612: (4)
613: (8)               a : array_like, shape (... , M, N)
614: (4)                   An array-like object with the dimensionality of at least 2.
615: (8)               mode : {'reduced', 'complete', 'r', 'raw'}, optional
616: (8)                   If K = min(M, N), then
617: (8)                     * 'reduced' : returns Q, R with dimensions (... , M, K), (... , K, N)
617: (8)                     * 'complete' : returns Q, R with dimensions (... , M, M), (... , M, N)

```

```

618: (8)          * 'r'           : returns R only with dimensions (... , K, N)
619: (8)          * 'raw'         : returns h, tau with dimensions (... , N, M), (... , K, )
620: (8)          The options 'reduced', 'complete', and 'raw' are new in numpy 1.8,
621: (8)          see the notes for more information. The default is 'reduced', and to
622: (8)          maintain backward compatibility with earlier versions of numpy both
623: (8)          it and the old default 'full' can be omitted. Note that array h
624: (8)          returned in 'raw' mode is transposed for calling Fortran. The
625: (8)          'economic' mode is deprecated. The modes 'full' and 'economic' may
626: (8)          be passed using only the first letter for backwards compatibility,
627: (8)          but all others must be spelled out. See the Notes for more
628: (8)          explanation.
629: (4)          Returns
630: (4)          -----
631: (4)          When mode is 'reduced' or 'complete', the result will be a namedtuple with
632: (4)          the attributes `Q` and `R`.
633: (4)          Q : ndarray of float or complex, optional
634: (8)          A matrix with orthonormal columns. When mode = 'complete' the
635: (8)          result is an orthogonal/unitary matrix depending on whether or not
636: (8)          a is real/complex. The determinant may be either +/- 1 in that
637: (8)          case. In case the number of dimensions in the input array is
638: (8)          greater than 2 then a stack of the matrices with above properties
639: (8)          is returned.
640: (4)          R : ndarray of float or complex, optional
641: (8)          The upper-triangular matrix or a stack of upper-triangular
642: (8)          matrices if the number of dimensions in the input array is greater
643: (8)          than 2.
644: (4)          (h, tau) : ndarrays of np.double or np.cdouble, optional
645: (8)          The array h contains the Householder reflectors that generate q
646: (8)          along with r. The tau array contains scaling factors for the
647: (8)          reflectors. In the deprecated 'economic' mode only h is returned.
648: (4)          Raises
649: (4)          -----
650: (4)          LinAlgError
651: (8)          If factoring fails.
652: (4)          See Also
653: (4)          -----
654: (4)          scipy.linalg.qr : Similar function in SciPy.
655: (4)          scipy.linalg.rq : Compute RQ decomposition of a matrix.
656: (4)
657: (4)
658: (4)          Notes
659: (4)          -----
660: (4)          This is an interface to the LAPACK routines ``dgeqr`` , ``zgeqr`` ,
661: (4)          ``dorgqr`` , and ``zungqr`` .
662: (4)          For more information on the qr factorization, see for example:
663: (4)          https://en.wikipedia.org/wiki/QR\_factorization
664: (4)          Subclasses of `ndarray` are preserved except for the 'raw' mode. So if
665: (4)          `a` is of type `matrix`, all the return values will be matrices too.
666: (4)          New 'reduced', 'complete', and 'raw' options for mode were added in
667: (4)          NumPy 1.8.0 and the old option 'full' was made an alias of 'reduced'. In
668: (4)          addition the options 'full' and 'economic' were deprecated. Because
669: (4)          'full' was the previous default and 'reduced' is the new default,
670: (4)          backward compatibility can be maintained by letting `mode` default.
671: (4)          The 'raw' option was added so that LAPACK routines that can multiply
672: (4)          arrays by q using the Householder reflectors can be used. Note that in
673: (4)          this case the returned arrays are of type np.double or np.cdouble and
674: (4)          the h array is transposed to be FORTRAN compatible. No routines using
675: (4)          the 'raw' return are currently exposed by numpy, but some are available
676: (4)          in lapack_lite and just await the necessary work.
677: (4)          Examples
678: (4)          -----
679: (4)          >>> a = np.random.randn(9, 6)
680: (4)          >>> Q, R = np.linalg.qr(a)
681: (4)          >>> np.allclose(a, np.dot(Q, R)) # a does equal QR
682: (4)          True
683: (4)          >>> R2 = np.linalg.qr(a, mode='r')
684: (4)          >>> np.allclose(R, R2) # mode='r' returns the same R as mode='full'
685: (4)          True
686: (4)          >>> a = np.random.normal(size=(3, 2, 2)) # Stack of 2 x 2 matrices as
687: (4)          input
688: (4)          >>> Q, R = np.linalg.qr(a)

```

```

686: (4)          >>> Q.shape
687: (4)          (3, 2, 2)
688: (4)          >>> R.shape
689: (4)          (3, 2, 2)
690: (4)          >>> np.allclose(a, np.matmul(Q, R))
691: (4)          True
692: (4)          Example illustrating a common use of `qr`: solving of least squares
693: (4)          problems
694: (4)          What are the least-squares-best `m` and `y0` in ``y = y0 + mx`` for
695: (4)          the following data: {(0,1), (1,0), (1,2), (2,1)}. (Graph the points
696: (4)          and you'll see that it should be  $y_0 = 0$ ,  $m = 1$ .) The answer is provided
697: (4)          by solving the over-determined matrix equation ``Ax = b``, where::
698: (6)          A = array([[0, 1], [1, 1], [1, 1], [2, 1]])
699: (6)          x = array([[y0], [m]])
700: (6)          b = array([[1], [0], [2], [1]])
701: (4)          If  $A = QR$  such that  $Q$  is orthonormal (which is always possible via
702: (4)          Gram-Schmidt), then `` $x = \text{inv}(R)^T \cdot (Q^T \cdot b)$ 
```

```

754: (8)             return wrap(a)
755: (4)             if mode == 'complete' and m > n:
756: (8)                 mc = m
757: (8)                 gufunc = _umath_linalg.qr_complete
758: (4)             else:
759: (8)                 mc = mn
760: (8)                 gufunc = _umath_linalg.qr_reduced
761: (4)                 signature = 'DD->D' if isComplexType(t) else 'dd->d'
762: (4)                 extobj = get_linalg_error_extobj(_raise_linalgerror_qr)
763: (4)                 q = gufunc(a, tau, signature=signature, extobj=extobj)
764: (4)                 r = triu(a[..., :mc, :])
765: (4)                 q = q.astype(result_t, copy=False)
766: (4)                 r = r.astype(result_t, copy=False)
767: (4)             return QRResult(wrap(q), wrap(r))
768: (0) @array_function_dispatch(_unary_dispatcher)
769: (0) def eigvals(a):
770: (4)     """
771: (4)         Compute the eigenvalues of a general matrix.
772: (4)         Main difference between `eigvals` and `eig`: the eigenvectors aren't
773: (4)         returned.
774: (4)         Parameters
775: (4)         -----
776: (4)         a : (... , M, M) array_like
777: (8)             A complex- or real-valued matrix whose eigenvalues will be computed.
778: (4)         Returns
779: (4)         -----
780: (4)         w : (... , M,) ndarray
781: (8)             The eigenvalues, each repeated according to its multiplicity.
782: (8)             They are not necessarily ordered, nor are they necessarily
783: (8)             real for real matrices.
784: (4)         Raises
785: (4)         -----
786: (4)         LinAlgError
787: (8)             If the eigenvalue computation does not converge.
788: (4)         See Also
789: (4)         -----
790: (4)         eig : eigenvalues and right eigenvectors of general arrays
791: (4)         eigvalsh : eigenvalues of real symmetric or complex Hermitian
792: (15)             (conjugate symmetric) arrays.
793: (4)         eigh : eigenvalues and eigenvectors of real symmetric or complex
794: (11)             Hermitian (conjugate symmetric) arrays.
795: (4)         scipy.linalg.eigvals : Similar function in SciPy.
796: (4)         Notes
797: (4)         -----
798: (4)         .. versionadded:: 1.8.0
799: (4)         Broadcasting rules apply, see the `numpy.linalg` documentation for
800: (4)         details.
801: (4)         This is implemented using the ``_geev`` LAPACK routines which compute
802: (4)         the eigenvalues and eigenvectors of general square arrays.
803: (4)         Examples
804: (4)         -----
805: (4)         Illustration, using the fact that the eigenvalues of a diagonal matrix
806: (4)         are its diagonal elements, that multiplying a matrix on the left
807: (4)         by an orthogonal matrix, `Q`, and on the right by `Q.T` (the transpose
808: (4)         of `Q`), preserves the eigenvalues of the "middle" matrix. In other
words,
809: (4)         if `Q` is orthogonal, then ``Q * A * Q.T`` has the same eigenvalues as
810: (4)         ``A``:
811: (4)         >>> from numpy import linalg as LA
812: (4)         >>> x = np.random.random()
813: (4)         >>> Q = np.array([[np.cos(x), -np.sin(x)], [np.sin(x), np.cos(x)]])
814: (4)         >>> LA.norm(Q[0, :]), LA.norm(Q[1, :]), np.dot(Q[0, :], Q[1, :])
815: (4)         (1.0, 1.0, 0.0)
816: (4)         Now multiply a diagonal matrix by ``Q`` on one side and by ``Q.T`` on the
other:
817: (4)         >>> D = np.diag((-1,1))
818: (4)         >>> LA.eigvals(D)
819: (4)         array([-1.,  1.])
820: (4)         >>> A = np.dot(Q, D)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

821: (4)          >>> A = np.dot(A, Q.T)
822: (4)          >>> LA.eigvals(A)
823: (4)          array([ 1., -1.]) # random
824: (4)          """
825: (4)          a, wrap = _makearray(a)
826: (4)          _assert_stacked_2d(a)
827: (4)          _assert_stacked_square(a)
828: (4)          _assert_finite(a)
829: (4)          t, result_t = _commonType(a)
830: (4)          extobj = get_linalg_error_extobj(
831: (8)              _raise_linalgerror_eigenvalues_nonconvergence)
832: (4)          signature = 'D->D' if isComplexType(t) else 'd->D'
833: (4)          w = _umath_linalg.eigvals(a, signature=signature, extobj=extobj)
834: (4)          if not isComplexType(t):
835: (8)              if all(w.imag == 0):
836: (12)                  w = w.real
837: (12)                  result_t = _realType(result_t)
838: (8)              else:
839: (12)                  result_t = _complexType(result_t)
840: (4)          return w.astype(result_t, copy=False)
841: (0)          def _eigvalsh_dispatcher(a, UPLO=None):
842: (4)              return (a,)
843: (0)          @array_function_dispatch(_eigvalsh_dispatcher)
844: (0)          def eigvalsh(a, UPLO='L'):
845: (4)          """
846: (4)          Compute the eigenvalues of a complex Hermitian or real symmetric matrix.
847: (4)          Main difference from eigh: the eigenvectors are not computed.
848: (4)          Parameters
849: (4)          -----
850: (4)          a : (..., M, M) array_like
851: (8)              A complex- or real-valued matrix whose eigenvalues are to be
852: (8)              computed.
853: (4)          UPLO : {'L', 'U'}, optional
854: (8)              Specifies whether the calculation is done with the lower triangular
855: (8)              part of `a` ('L', default) or the upper triangular part ('U').
856: (8)              Irrespective of this value only the real parts of the diagonal will
857: (8)              be considered in the computation to preserve the notion of a Hermitian
858: (8)              matrix. It therefore follows that the imaginary part of the diagonal
859: (8)              will always be treated as zero.
860: (4)          Returns
861: (4)          -----
862: (4)          w : (..., M,) ndarray
863: (8)              The eigenvalues in ascending order, each repeated according to
864: (8)              its multiplicity.
865: (4)          Raises
866: (4)          -----
867: (4)          LinAlgError
868: (8)              If the eigenvalue computation does not converge.
869: (4)          See Also
870: (4)          -----
871: (4)          eigh : eigenvalues and eigenvectors of real symmetric or complex Hermitian
872: (11)             (conjugate symmetric) arrays.
873: (4)          eigvals : eigenvalues of general real or complex arrays.
874: (4)          eig : eigenvalues and right eigenvectors of general real or complex
875: (10)             arrays.
876: (4)          scipy.linalg.eigvalsh : Similar function in SciPy.
877: (4)          Notes
878: (4)          -----
879: (4)          .. versionadded:: 1.8.0
880: (4)          Broadcasting rules apply, see the `numpy.linalg` documentation for
881: (4)          details.
882: (4)          The eigenvalues are computed using LAPACK routines ``_syevd``, ``_heevd``.
883: (4)          Examples
884: (4)          -----
885: (4)          >>> from numpy import linalg as LA
886: (4)          >>> a = np.array([[1, -2j], [2j, 5]])
887: (4)          >>> LA.eigvalsh(a)
888: (4)          array([ 0.17157288,  5.82842712]) # may vary
889: (4)          >>> # demonstrate the treatment of the imaginary part of the diagonal

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

890: (4)
891: (4)
892: (4)
893: (11)
894: (4)
895: (4)
896: (4)
897: (4)
898: (4)
899: (11)
900: (4)
901: (4)
902: (4)
903: (4)
904: (4)
905: (4)
906: (4)
907: (4)
908: (8)
909: (4)
910: (8)
911: (4)
912: (8)
913: (4)
914: (8)
915: (4)
916: (4)
917: (4)
918: (4)
919: (4)
920: (4)
921: (4)
922: (0)
923: (4)
924: (4)
925: (4)
926: (0)
927: (0)
928: (4)
929: (4)
930: (4)
931: (4)
932: (4)
933: (8)
934: (8)
935: (4)
936: (4)
937: (4)
938: (4)
939: (8)
940: (8)
941: (8)
942: (8)
943: (8)
944: (8)
945: (4)
946: (8)
947: (8)
948: (8)
949: (4)
950: (4)
951: (4)
952: (8)
953: (4)
954: (4)
955: (4)
956: (4)
957: (11)
958: (4)

>>> a = np.array([[5+2j, 9-2j], [0+2j, 2-1j]])
>>> a
array([[5.+2.j, 9.-2.j],
       [0.+2.j, 2.-1.j]])
>>> # with UPL0='L' this is numerically equivalent to using LA.eigvals()
>>> # with:
>>> b = np.array([[5.+0.j, 0.-2.j], [0.+2.j, 2.-0.j]])
>>> b
array([[5.+0.j, 0.-2.j],
       [0.+2.j, 2.+0.j]])
>>> wa = LA.eigvalsh(a)
>>> wb = LA.eigvals(b)
>>> wa; wb
array([1., 6.])
array([6.+0.j, 1.+0.j])
"""

UPLO = UPL0.upper()
if UPLO not in ('L', 'U'):
    raise ValueError("UPLO argument must be 'L' or 'U''")
extobj = get_linalg_error_extobj(
    _raise_linalgerror_eigenvalues_nonconvergence)
if UPLO == 'L':
    gufunc = _umath_linalg.eigvalsh_lo
else:
    gufunc = _umath_linalg.eigvalsh_up
a, wrap = _makearray(a)
_assert_stacked_2d(a)
_assert_stacked_square(a)
t, result_t = _commonType(a)
signature = 'D->d' if isComplexType(t) else 'd->d'
w = gufunc(a, signature=signature, extobj=extobj)
return w.astype(_realType(result_t), copy=False)

def _convertarray(a):
    t, result_t = _commonType(a)
    a = a.astype(t).T.copy()
    return a, t, result_t

@array_function_dispatch(_unary_dispatcher)
def eig(a):
    """
    Compute the eigenvalues and right eigenvectors of a square array.

    Parameters
    -----
    a : (..., M, M) array
        Matrices for which the eigenvalues and right eigenvectors will
        be computed

    Returns
    -----
    A namedtuple with the following attributes:
    eigenvalues : (..., M) array
        The eigenvalues, each repeated according to its multiplicity.
        The eigenvalues are not necessarily ordered. The resulting
        array will be of complex type, unless the imaginary part is
        zero in which case it will be cast to a real type. When `a`
        is real the resulting eigenvalues will be real (0 imaginary
        part) or occur in conjugate pairs
    eigenvectors : (..., M, M) array
        The normalized (unit "length") eigenvectors, such that the
        column ``eigenvectors[:,i]`` is the eigenvector corresponding to the
        eigenvalue ``eigenvalues[i]``.

    Raises
    -----
    LinAlgError
        If the eigenvalue computation does not converge.

    See Also
    -----
    eigvals : eigenvalues of a non-symmetric array.
    eigh : eigenvalues and eigenvectors of a real symmetric or complex
           Hermitian (conjugate symmetric) array.
    eigvalsh : eigenvalues of a real symmetric or complex Hermitian
    """

    return _convertarray(a)

```

```

959: (15)                               (conjugate symmetric) array.
960: (4)      scipy.linalg.eig : Similar function in SciPy that also solves the
961: (23)      generalized eigenvalue problem.
962: (4)      scipy.linalg.schur : Best choice for unitary and other non-Hermitian
963: (25)      normal matrices.

964: (4)      Notes
965: -----
966: (4)      .. versionadded:: 1.8.0
967: (4)      Broadcasting rules apply, see the `numpy.linalg` documentation for
968: (4)      details.
969: (4)      This is implemented using the ``_geev`` LAPACK routines which compute
970: (4)      the eigenvalues and eigenvectors of general square arrays.
971: (4)      The number `w` is an eigenvalue of `a` if there exists a vector `v` such
972: (4)      that ``a @ v = w * v``. Thus, the arrays `a`, `eigenvalues`, and
973: (4)      `eigenvectors` satisfy the equations ``a @ eigenvectors[:,i] =
974: (4)      eigenvalues[i] * eigenvectors[:,i]`` for :math:`i \in \{0, \dots, M-1\}`.
975: (4)      The array `eigenvectors` may not be of maximum rank, that is, some of the
976: (4)      columns may be linearly dependent, although round-off error may obscure
977: (4)      that fact. If the eigenvalues are all different, then theoretically the
978: (4)      eigenvectors are linearly independent and `a` can be diagonalized by a
979: (4)      similarity transformation using `eigenvectors`, i.e., ``inv(eigenvectors) @
980: (4)      a @ eigenvectors`` is diagonal.
981: (4)      For non-Hermitian normal matrices the SciPy function `scipy.linalg.schur`
982: (4)      is preferred because the matrix `eigenvectors` is guaranteed to be
983: (4)      unitary, which is not the case when using `eig`. The Schur factorization
984: (4)      produces an upper triangular matrix rather than a diagonal matrix, but for
985: (4)      normal matrices only the diagonal of the upper triangular matrix is
986: (4)      needed, the rest is roundoff error.
987: (4)      Finally, it is emphasized that `eigenvectors` consists of the *right* (as
988: (4)      in right-hand side) eigenvectors of `a`. A vector `y` satisfying ``y.T @ a
989: (4)      = z * y.T`` for some number `z` is called a *left* eigenvector of `a`,
990: (4)      and, in general, the left and right eigenvectors of a matrix are not
991: (4)      necessarily the (perhaps conjugate) transposes of each other.
992: (4)      References
993: -----
994: (4)      G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando, FL,
995: (4)      Academic Press, Inc., 1980, Various pp.
996: (4)      Examples
997: -----
998: (4)      >>> from numpy import linalg as LA
999: (4)      (Almost) trivial example with real eigenvalues and eigenvectors.
1000: (4)      >>> eigenvalues, eigenvectors = LA.eig(np.diag((1, 2, 3)))
1001: (4)      >>> eigenvalues
1002: (4)      array([1., 2., 3.])
1003: (4)      >>> eigenvectors
1004: (4)      array([[1., 0., 0.],
1005: (11)          [0., 1., 0.],
1006: (11)          [0., 0., 1.]])
1007: (4)      Real matrix possessing complex eigenvalues and eigenvectors; note that the
1008: (4)      eigenvalues are complex conjugates of each other.
1009: (4)      >>> eigenvalues, eigenvectors = LA.eig(np.array([[1, -1], [1, 1]]))
1010: (4)      >>> eigenvalues
1011: (4)      array([1.+1.j, 1.-1.j])
1012: (4)      >>> eigenvectors
1013: (4)      array([[0.70710678+0.j, 0.70710678-0.j],
1014: (11)          [0.          -0.70710678j, 0.          +0.70710678j]])
1015: (4)      Complex-valued matrix with real eigenvalues (but complex-valued
eigenvectors);
1016: (4)      note that ``a.conj().T == a``, i.e., `a` is Hermitian.
1017: (4)      >>> a = np.array([[1, 1j], [-1j, 1]])
1018: (4)      >>> eigenvalues, eigenvectors = LA.eig(a)
1019: (4)      >>> eigenvalues
1020: (4)      array([2.+0.j, 0.+0.j])
1021: (4)      >>> eigenvectors
1022: (4)      array([[ 0.          +0.70710678j, 0.70710678+0.j      ],
1023: (11)          [ 0.70710678+0.j      , -0.          +0.70710678j]])]
1024: (4)      Be careful about round-off error!
1025: (4)      >>> a = np.array([[1 + 1e-9, 0], [0, 1 - 1e-9]])
1026: (4)      >>> # Theor. eigenvalues are 1 +/- 1e-9

```

```

1027: (4)             >>> eigenvalues, eigenvectors = LA.eig(a)
1028: (4)             >>> eigenvalues
1029: (4)             array([1., 1.])
1030: (4)             >>> eigenvectors
1031: (4)             array([[1., 0.],
1032: (11)                 [0., 1.]])
1033: (4)             """
1034: (4)             a, wrap = _makearray(a)
1035: (4)             _assert_stacked_2d(a)
1036: (4)             _assert_stacked_square(a)
1037: (4)             _assert_finite(a)
1038: (4)             t, result_t = _commonType(a)
1039: (4)             extobj = get_linalg_error_extobj(
1040: (8)                 _raise_linalgerror_eigenvalues_nonconvergence)
1041: (4)             signature = 'D->DD' if isComplexType(t) else 'd->DD'
1042: (4)             w, vt = _umath_linalg.eig(a, signature=signature, extobj=extobj)
1043: (4)             if not isComplexType(t) and all(w.imag == 0.0):
1044: (8)                 w = w.real
1045: (8)                 vt = vt.real
1046: (8)                 result_t = _realType(result_t)
1047: (4)             else:
1048: (8)                 result_t = _complexType(result_t)
1049: (4)             vt = vt.astype(result_t, copy=False)
1050: (4)             return EigResult(w.astype(result_t, copy=False), wrap(vt))
1051: (0)             @array_function_dispatch(_eigvalsh_dispatcher)
1052: (0)             def eigh(a, UPLO='L'):
1053: (4)             """
1054: (4)             Return the eigenvalues and eigenvectors of a complex Hermitian
1055: (4)             (conjugate symmetric) or a real symmetric matrix.
1056: (4)             Returns two objects, a 1-D array containing the eigenvalues of `a`, and
1057: (4)             a 2-D square array or matrix (depending on the input type) of the
1058: (4)             corresponding eigenvectors (in columns).
1059: (4)             Parameters
1060: (4)             -----
1061: (4)             a : (..., M, M) array
1062: (8)                 Hermitian or real symmetric matrices whose eigenvalues and
1063: (8)                 eigenvectors are to be computed.
1064: (4)             UPLO : {'L', 'U'}, optional
1065: (8)                 Specifies whether the calculation is done with the lower triangular
1066: (8)                 part of `a` ('L', default) or the upper triangular part ('U').
1067: (8)                 Irrespective of this value only the real parts of the diagonal will
1068: (8)                 be considered in the computation to preserve the notion of a Hermitian
1069: (8)                 matrix. It therefore follows that the imaginary part of the diagonal
1070: (8)                 will always be treated as zero.
1071: (4)             Returns
1072: (4)             -----
1073: (4)             A namedtuple with the following attributes:
1074: (4)             eigenvalues : (..., M) ndarray
1075: (8)                 The eigenvalues in ascending order, each repeated according to
1076: (8)                 its multiplicity.
1077: (4)             eigenvectors : {(..., M, M) ndarray, (... , M, M) matrix}
1078: (8)                 The column ``eigenvectors[:, i]`` is the normalized eigenvector
1079: (8)                 corresponding to the eigenvalue ``eigenvalues[i]``. Will return a
1080: (8)                 matrix object if `a` is a matrix object.
1081: (4)             Raises
1082: (4)             -----
1083: (4)             LinAlgError
1084: (8)                 If the eigenvalue computation does not converge.
1085: (4)             See Also
1086: (4)             -----
1087: (4)             eigvalsh : eigenvalues of real symmetric or complex Hermitian
1088: (15)                 (conjugate symmetric) arrays.
1089: (4)             eig : eigenvalues and right eigenvectors for non-symmetric arrays.
1090: (4)             eigvals : eigenvalues of non-symmetric arrays.
1091: (4)             scipy.linalg.eigh : Similar function in SciPy (but also solves the
1092: (24)                 generalized eigenvalue problem).
1093: (4)             Notes
1094: (4)             -----
1095: (4)             .. versionadded:: 1.8.0

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1096: (4) Broadcasting rules apply, see the `numpy.linalg` documentation for
1097: (4) details.
1098: (4) The eigenvalues/eigenvectors are computed using LAPACK routines
``_syevd``,
1099: (4)
1100: (4) ``_heevd``.
1101: (4) The eigenvalues of real symmetric or complex Hermitian matrices are always
1102: (4) real. [1]_ The array `eigenvalues` of (column) eigenvectors is unitary and
1103: (4) `a`, `eigenvalues`, and `eigenvectors` satisfy the equations ``dot(a,
1104: (4) eigenvectors[:, i]) = eigenvalues[i] * eigenvectors[:, i]``.
1105: (4) References
1106: (4) -----
1107: (11) .. [1] G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando,
1108: (4) FL, Academic Press, Inc., 1980, pg. 222.
1109: (4) Examples
1110: (4) -----
1111: (4) >>> from numpy import linalg as LA
1112: (4) >>> a = np.array([[1, -2j], [2j, 5]])
1113: (4) >>> a
1114: (11) array([[ 1.+0.j, -0.-2.j],
1115: (4)           [ 0.+2.j,  5.+0.j]])
1116: (4) >>> eigenvalues, eigenvectors = LA.eigh(a)
1117: (4) >>> eigenvalues
1118: (4) array([0.17157288, 5.82842712])
1119: (4) >>> eigenvectors
1120: (11) array([[-0.92387953+0.j      , -0.38268343+0.j      ], # may vary
1121: (4)           [ 0.          +0.38268343j,  0.          -0.92387953j]])
1122: (4) >>> np.dot(a, eigenvectors[:, 0]) - eigenvalues[0] * eigenvectors[:, 0] #
verify 1st eigenval/vec pair
1123: (4) array([5.55111512e-17+0.0000000e+00j, 0.00000000e+00+1.2490009e-16j])
1124: (4) verify 2nd eigenval/vec pair
1125: (4) >>> np.dot(a, eigenvectors[:, 1]) - eigenvalues[1] * eigenvectors[:, 1] #
1126: (4)
1127: (4) >>> A = np.matrix(a) # what happens if input is a matrix object
1128: (12) >>> A
1129: (4) matrix([[ 1.+0.j, -0.-2.j],
1130: (4)           [ 0.+2.j,  5.+0.j]])
1131: (4) >>> eigenvalues, eigenvectors = LA.eigh(A)
1132: (4) >>> eigenvalues
1133: (4) array([0.17157288, 5.82842712])
1134: (12) >>> eigenvectors
1135: (4) matrix([[-0.92387953+0.j      , -0.38268343+0.j      ], # may vary
1136: (4)           [ 0.          +0.38268343j,  0.          -0.92387953j]])
1137: (4) >>> # demonstrate the treatment of the imaginary part of the diagonal
1138: (4) >>> a = np.array([[5+2j, 9-2j], [0+2j, 2-1j]])
1139: (11) >>> a
1140: (4) array([[5.+2.j, 9.-2.j],
1141: (4)           [0.+2.j, 2.-1.j]])
1142: (4) >>> # with UPL0='L' this is numerically equivalent to using LA.eig() with:
1143: (4) >>> b = np.array([[5.+0.j, 0.-2.j], [0.+2.j, 2.-0.j]])
1144: (11) >>> b
1145: (4) array([[5.+0.j, 0.-2.j],
1146: (4)           [0.+2.j, 2.+0.j]])
1147: (4) >>> wa, va = LA.eigh(a)
1148: (4) >>> wb, vb = LA.eig(b)
1149: (4) >>> wa; wb
1150: (4) array([1., 6.])
1151: (4) array([6.+0.j, 1.+0.j])
1152: (11) >>> va; vb
1153: (4) array([[-0.4472136 +0.j      , -0.89442719+0.j      ], # may vary
1154: (11)           [ 0.          +0.89442719j,  0.          -0.4472136j ]])
1155: (4) array([[ 0.89442719+0.j      , -0.          +0.4472136j],
1156: (4)           [-0.          +0.4472136j,  0.89442719+0.j      ]])
1157: (4) """
1158: (8)     UPL0 = UPL0.upper()
1159: (4)     if UPL0 not in ('L', 'U'):
1160: (4)         raise ValueError("UPL0 argument must be 'L' or 'U'")
1161: (4)     a, wrap = _makearray(a)
1162: (4)     _assert_stacked_2d(a)
1163: (4)     _assert_stacked_square(a)

```

```

1162: (4) t, result_t = _commonType(a)
1163: (4) extobj = get_linalg_error_extobj(
1164: (8)     _raise_linalgerror_eigenvalues_nonconvergence)
1165: (4) if UPLO == 'L':
1166: (8)     gufunc = _umath_linalg.eigh_lo
1167: (4) else:
1168: (8)     gufunc = _umath_linalg.eigh_up
1169: (4) signature = 'D->dD' if isComplexType(t) else 'd->dd'
1170: (4) w, vt = gufunc(a, signature=signature, extobj=extobj)
1171: (4) w = w.astype(_realType(result_t), copy=False)
1172: (4) vt = vt.astype(result_t, copy=False)
1173: (4) return EighResult(w, wrap(vt))
1174: (0) def _svd_dispatcher(a, full_matrices=None, compute_uv=None, hermitian=None):
1175: (4)     return (a,)
1176: (0) @array_function_dispatch(_svd_dispatcher)
1177: (0) def svd(a, full_matrices=True, compute_uv=True, hermitian=False):
1178: (4) """
1179: (4)     Singular Value Decomposition.
1180: (4)     When `a` is a 2D array, and ``full_matrices=False``, then it is
1181: (4)     factorized as ``u @ np.diag(s) @ vh = (u * s) @ vh``, where
1182: (4)     `u` and the Hermitian transpose of `vh` are 2D arrays with
1183: (4)     orthonormal columns and `s` is a 1D array of `a`'s singular
1184: (4)     values. When `a` is higher-dimensional, SVD is applied in
1185: (4)     stacked mode as explained below.
1186: (4)     Parameters
1187: (4)     -----
1188: (4)     a : (... , M, N) array_like
1189: (8)         A real or complex array with ``a.ndim >= 2``.
1190: (4)     full_matrices : bool, optional
1191: (8)         If True (default), `u` and `vh` have the shapes ``(... , M, M)`` and
1192: (8)         ``(... , N, N)``, respectively. Otherwise, the shapes are
1193: (8)         ``(... , M, K)`` and ``(... , K, N)``, respectively, where
1194: (8)         ``K = min(M, N)``.
1195: (4)     compute_uv : bool, optional
1196: (8)         Whether or not to compute `u` and `vh` in addition to `s`. True
1197: (8)         by default.
1198: (4)     hermitian : bool, optional
1199: (8)         If True, `a` is assumed to be Hermitian (symmetric if real-valued),
1200: (8)         enabling a more efficient method for finding singular values.
1201: (8)         Defaults to False.
1202: (8)         .. versionadded:: 1.17.0
1203: (4)     Returns
1204: (4)     -----
1205: (4)     When `compute_uv` is True, the result is a namedtuple with the following
1206: (4)     attribute names:
1207: (4)     U : { (... , M, M), (... , M, K) } array
1208: (8)         Unitary array(s). The first ``a.ndim - 2`` dimensions have the same
1209: (8)         size as those of the input `a`. The size of the last two dimensions
1210: (8)         depends on the value of `full_matrices`. Only returned when
1211: (8)         `compute_uv` is True.
1212: (4)     S : (... , K) array
1213: (8)         Vector(s) with the singular values, within each vector sorted in
1214: (8)         descending order. The first ``a.ndim - 2`` dimensions have the same
1215: (8)         size as those of the input `a`.
1216: (4)     Vh : { (... , N, N), (... , K, N) } array
1217: (8)         Unitary array(s). The first ``a.ndim - 2`` dimensions have the same
1218: (8)         size as those of the input `a`. The size of the last two dimensions
1219: (8)         depends on the value of `full_matrices`. Only returned when
1220: (8)         `compute_uv` is True.
1221: (4)     Raises
1222: (4)     -----
1223: (4)     LinAlgError
1224: (8)         If SVD computation does not converge.
1225: (4)     See Also
1226: (4)     -----
1227: (4)     scipy.linalg.svd : Similar function in SciPy.
1228: (4)     scipy.linalg.svdvals : Compute singular values of a matrix.
1229: (4)     Notes
1230: (4)     -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1231: (4) .. versionchanged:: 1.8.0
1232: (7) Broadcasting rules apply, see the `numpy.linalg` documentation for
1233: (7) details.
1234: (4) The decomposition is performed using LAPACK routine ``_gesdd``.
1235: (4) SVD is usually described for the factorization of a 2D matrix :math:`A`.
1236: (4) The higher-dimensional case will be discussed below. In the 2D case, SVD
is
1237: (4) written as :math:`A = U S V^H`, where :math:`A = a`, :math:`U= u`,
1238: (4) :math:`S= \sqrt{\text{np.diag}}(s)` and :math:`V^H = vh`. The 1D array `s`
1239: (4) contains the singular values of `a` and `u` and `vh` are unitary. The rows
1240: (4) of `vh` are the eigenvectors of :math:`A^H A` and the columns of `u` are
1241: (4) the eigenvectors of :math:`A A^H`. In both cases the corresponding
1242: (4) (possibly non-zero) eigenvalues are given by ``s**2``.
1243: (4) If `a` has more than two dimensions, then broadcasting rules apply, as
1244: (4) explained in :ref:`routines.linalg-broadcasting`. This means that SVD is
1245: (4) working in "stacked" mode: it iterates over all indices of the first
1246: (4) ``a.ndim - 2`` dimensions and for each combination SVD is applied to the
1247: (4) last two indices. The matrix `a` can be reconstructed from the
1248: (4) decomposition with either ``((u * s[..., None, :]) @ vh`` or
1249: (4) ``u @ (s[..., None] * vh)``. (The ``@`` operator can be replaced by the
1250: (4) function ``np.matmul`` for python versions below 3.5.)
1251: (4) If `a` is a ``matrix`` object (as opposed to an ``ndarray``), then so are
1252: (4) all the return values.
1253: (4) Examples
1254: (4) -----
1255: (4) >>> a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
1256: (4) >>> b = np.random.randn(2, 7, 8, 3) + 1j*np.random.randn(2, 7, 8, 3)
1257: (4) Reconstruction based on full SVD, 2D case:
1258: (4) >>> U, S, Vh = np.linalg.svd(a, full_matrices=True)
1259: (4) >>> U.shape, S.shape, Vh.shape
1260: (4) ((9, 9), (6,), (6, 6))
1261: (4) >>> np.allclose(a, np.dot(U[:, :6] * S, Vh))
1262: (4) True
1263: (4) >>> smat = np.zeros((9, 6), dtype=complex)
1264: (4) >>> smat[:6, :6] = np.diag(S)
1265: (4) >>> np.allclose(a, np.dot(U, np.dot(smat, Vh)))
1266: (4) True
1267: (4) Reconstruction based on reduced SVD, 2D case:
1268: (4) >>> U, S, Vh = np.linalg.svd(a, full_matrices=False)
1269: (4) >>> U.shape, S.shape, Vh.shape
1270: (4) ((9, 6), (6,), (6, 6))
1271: (4) >>> np.allclose(a, np.dot(U * S, Vh))
1272: (4) True
1273: (4) >>> smat = np.diag(S)
1274: (4) >>> np.allclose(a, np.dot(U, np.dot(smat, Vh)))
1275: (4) True
1276: (4) Reconstruction based on full SVD, 4D case:
1277: (4) >>> U, S, Vh = np.linalg.svd(b, full_matrices=True)
1278: (4) >>> U.shape, S.shape, Vh.shape
1279: (4) ((2, 7, 8, 8), (2, 7, 3), (2, 7, 3, 3))
1280: (4) >>> np.allclose(b, np.matmul(U[..., :3] * S[..., None, :], Vh))
1281: (4) True
1282: (4) >>> np.allclose(b, np.matmul(U[..., :3], S[..., None] * Vh))
1283: (4) True
1284: (4) Reconstruction based on reduced SVD, 4D case:
1285: (4) >>> U, S, Vh = np.linalg.svd(b, full_matrices=False)
1286: (4) >>> U.shape, S.shape, Vh.shape
1287: (4) ((2, 7, 8, 3), (2, 7, 3), (2, 7, 3, 3))
1288: (4) >>> np.allclose(b, np.matmul(U * S[..., None, :], Vh))
1289: (4) True
1290: (4) >>> np.allclose(b, np.matmul(U, S[..., None] * Vh))
1291: (4) True
1292: (4) """
1293: (4) import numpy as _nx
1294: (4) a, wrap = _makearray(a)
1295: (4) if hermitian:
1296: (8)     if compute_uv:
1297: (12)         s, u = eigh(a)
1298: (12)         sgn = sign(s)

```

```

1299: (12)             s = abs(s)
1300: (12)             sidx = argsort(s)[..., ::-1]
1301: (12)             sgn = _nx.take_along_axis(sgn, sidx, axis=-1)
1302: (12)             s = _nx.take_along_axis(s, sidx, axis=-1)
1303: (12)             u = _nx.take_along_axis(u, sidx[..., None, :], axis=-1)
1304: (12)             vt = transpose(u * sgn[..., None, :]).conjugate()
1305: (12)             return SVDResult(wrap(u), s, wrap(vt))
1306: (8)              else:
1307: (12)                  s = eigvalsh(a)
1308: (12)                  s = abs(s)
1309: (12)                  return sort(s)[..., ::-1]
1310: (4) _assert_stacked_2d(a)
1311: (4) t, result_t = _commonType(a)
1312: (4) extobj = get_linalg_error_extobj(_raise_linalgerror_svd_nonconvergence)
1313: (4) m, n = a.shape[-2:]
1314: (4) if compute_uv:
1315: (8)     if full_matrices:
1316: (12)         if m < n:
1317: (16)             gufunc = _umath_linalg.svd_m_f
1318: (12)         else:
1319: (16)             gufunc = _umath_linalg.svd_n_f
1320: (8)     else:
1321: (12)         if m < n:
1322: (16)             gufunc = _umath_linalg.svd_m_s
1323: (12)         else:
1324: (16)             gufunc = _umath_linalg.svd_n_s
1325: (8)     signature = 'D->DdD' if isComplexType(t) else 'd->ddd'
1326: (8)     u, s, vh = gufunc(a, signature=signature, extobj=extobj)
1327: (8)     u = u.astype(result_t, copy=False)
1328: (8)     s = s.astype(_realType(result_t), copy=False)
1329: (8)     vh = vh.astype(result_t, copy=False)
1330: (8)     return SVDResult(wrap(u), s, wrap(vh))
1331: (4) else:
1332: (8)     if m < n:
1333: (12)         gufunc = _umath_linalg.svd_m
1334: (8)     else:
1335: (12)         gufunc = _umath_linalg.svd_n
1336: (8)     signature = 'D->d' if isComplexType(t) else 'd->d'
1337: (8)     s = gufunc(a, signature=signature, extobj=extobj)
1338: (8)     s = s.astype(_realType(result_t), copy=False)
1339: (8)     return s
1340: (0) def _cond_dispatcher(x, p=None):
1341: (4)     return (x,)
1342: (0) @array_function_dispatch(_cond_dispatcher)
1343: (0) def cond(x, p=None):
1344: (4) """
1345: (4)     Compute the condition number of a matrix.
1346: (4)     This function is capable of returning the condition number using
1347: (4)     one of seven different norms, depending on the value of `p` (see
1348: (4)     Parameters below).
1349: (4)     Parameters
1350: (4) -----
1351: (4)     x : (..., M, N) array_like
1352: (8)         The matrix whose condition number is sought.
1353: (4)     p : {None, 1, -1, 2, -2, inf, -inf, 'fro'}, optional
1354: (8)         Order of the norm used in the condition number computation:
1355: (8)         =====
1356: (8)         p      norm for matrices
1357: (8)         =====
1358: (8)         None   2-norm, computed directly using the ``SVD``
1359: (8)         'fro'  Frobenius norm
1360: (8)         inf    max(sum(abs(x), axis=1))
1361: (8)         -inf   min(sum(abs(x), axis=1))
1362: (8)         1      max(sum(abs(x), axis=0))
1363: (8)         -1    min(sum(abs(x), axis=0))
1364: (8)         2      2-norm (largest sing. value)
1365: (8)         -2    smallest singular value
1366: (8)         =====
1367: (8)         inf means the `numpy.inf` object, and the Frobenius norm is

```

```

1368: (8)           the root-of-sum-of-squares norm.
1369: (4)           Returns
1370: (4)
1371: (4)           c : {float, inf}
1372: (8)           The condition number of the matrix. May be infinite.
1373: (4)           See Also
1374: (4)
1375: (4)           numpy.linalg.norm
1376: (4)
1377: (4)
1378: (4)           The condition number of `x` is defined as the norm of `x` times the
1379: (4)           norm of the inverse of `x` [1]; the norm can be the usual L2-norm
1380: (4)           (root-of-sum-of-squares) or one of a number of other matrix norms.
1381: (4)           References
1382: (4)
1383: (4)           .. [1] G. Strang, *Linear Algebra and Its Applications*, Orlando, FL,
1384: (11)             Academic Press, Inc., 1980, pg. 285.
1385: (4)
1386: (4)
1387: (4)           Examples
1388: (4)
1389: (4)           >>> from numpy import linalg as LA
1390: (4)           >>> a = np.array([[1, 0, -1], [0, 1, 0], [1, 0, 1]])
1391: (11)             >>> a
1392: (11)             array([[ 1,  0, -1],
1393: (4)               [ 0,  1,  0],
1394: (4)               [ 1,  0,  1]])
1395: (4)           >>> LA.cond(a)
1396: (4)           1.4142135623730951
1397: (4)           >>> LA.cond(a, 'fro')
1398: (4)           3.1622776601683795
1399: (4)           >>> LA.cond(a, np.inf)
1400: (4)           2.0
1401: (4)           >>> LA.cond(a, -np.inf)
1402: (4)           1.0
1403: (4)           >>> LA.cond(a, 1)
1404: (4)           2.0
1405: (4)           >>> LA.cond(a, 2)
1406: (4)           1.4142135623730951
1407: (4)           >>> LA.cond(a, -2)
1408: (4)           0.70710678118654746 # may vary
1409: (4)           >>> min(LA.svd(a, compute_uv=False))*min(LA.svd(LA.inv(a),
compute_uv=False))
1410: (4)           0.70710678118654746 # may vary
1411: (4)
1412: (4)
1413: (4)
1414: (8)           """
1415: (4)           x = asarray(x) # in case we have a matrix
1416: (4)           if _is_empty_2d(x):
1417: (4)               raise LinAlgError("cond is not defined on empty arrays")
1418: (12)
1419: (16)
1420: (12)
1421: (16)
1422: (4)
1423: (8)
1424: (8)
1425: (8)
1426: (8)
1427: (8)
1428: (12)
1429: (12)
1430: (8)
1431: (4)
1432: (4)
1433: (4)
1434: (8)
1435: (8)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1436: (12)             r[nan_mask] = Inf
1437: (8)              elif nan_mask:
1438: (12)                r[()] = Inf
1439: (4)              if r.ndim == 0:
1440: (8)                r = r[()]
1441: (4)              return r
1442: (0)          def _matrix_rank_dispatcher(A, tol=None, hermitian=None):
1443: (4)            return (A,)
1444: (0)          @array_function_dispatch(_matrix_rank_dispatcher)
1445: (0)          def matrix_rank(A, tol=None, hermitian=False):
1446: (4)            """
1447: (4)              Return matrix rank of array using SVD method
1448: (4)              Rank of the array is the number of singular values of the array that are
1449: (4)              greater than `tol`.
1450: (4)              .. versionchanged:: 1.14
1451: (7)                Can now operate on stacks of matrices
1452: (4)            Parameters
1453: (4)            -----
1454: (4)            A : {(M,), (... , M, N)} array_like
1455: (8)              Input vector or stack of matrices.
1456: (4)            tol : (...) array_like, float, optional
1457: (8)              Threshold below which SVD values are considered zero. If `tol` is
1458: (8)              None, and ``S`` is an array with singular values for `M`, and
1459: (8)              ``eps`` is the epsilon value for datatype of ``S``, then `tol` is
1460: (8)              set to ``S.max() * max(M, N) * eps``.
1461: (8)              .. versionchanged:: 1.14
1462: (11)                Broadcasted against the stack of matrices
1463: (4)            hermitian : bool, optional
1464: (8)              If True, `A` is assumed to be Hermitian (symmetric if real-valued),
1465: (8)              enabling a more efficient method for finding singular values.
1466: (8)              Defaults to False.
1467: (8)              .. versionadded:: 1.14
1468: (4)            Returns
1469: (4)            -----
1470: (4)            rank : (...) array_like
1471: (8)              Rank of A.
1472: (4)            Notes
1473: (4)            -----
1474: (4)            The default threshold to detect rank deficiency is a test on the magnitude
1475: (4)            of the singular values of `A`. By default, we identify singular values
less
1476: (4)            than ``S.max() * max(M, N) * eps`` as indicating rank deficiency (with
1477: (4)            the symbols defined above). This is the algorithm MATLAB uses [1]. It
also
1478: (4)            appears in *Numerical recipes* in the discussion of SVD solutions for
linear
1479: (4)            least squares [2].
1480: (4)            This default threshold is designed to detect rank deficiency accounting
for
1481: (4)            the numerical errors of the SVD computation. Imagine that there is a
column
1482: (4)            in `A` that is an exact (in floating point) linear combination of other
1483: (4)            columns in `A`. Computing the SVD on `A` will not produce a singular value
1484: (4)            exactly equal to 0 in general: any difference of the smallest SVD value
from
1485: (4)            0 will be caused by numerical imprecision in the calculation of the SVD.
1486: (4)            Our threshold for small SVD values takes this numerical imprecision into
1487: (4)            account, and the default threshold will detect such numerical rank
deficiency. The threshold may declare a matrix `A` rank deficient even if
1488: (4)            the linear combination of some columns of `A` is not exactly equal to
1489: (4)            another column of `A` but only numerically very close to another column of
`A`.
1490: (4)            We chose our default threshold because it is in wide use. Other
thresholds
1491: (4)            are possible. For example, elsewhere in the 2007 edition of *Numerical
1492: (4)            recipes* there is an alternative threshold of ``S.max() *
1493: (4)            np.finfo(A.dtype).eps / 2. * np.sqrt(m + n + 1.)``. The authors describe
1494: (4)            this threshold as being based on "expected roundoff error" (p 71).
1495: (4)            The thresholds above deal with floating point roundoff error in the
1496: (4)
1497: (4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

1498: (4) calculation of the SVD. However, you may have more information about the  
 1499: (4) sources of error in `A` that would make you consider other tolerance  
 values

1500: (4) to detect \*effective\* rank deficiency. The most useful measure of the  
 1501: (4) tolerance depends on the operations you intend to use on your matrix. For  
 1502: (4) example, if your data come from uncertain measurements with uncertainties  
 1503: (4) greater than floating point epsilon, choosing a tolerance near that  
 1504: (4) uncertainty may be preferable. The tolerance may be absolute if the  
 1505: (4) uncertainties are absolute rather than relative.

1506: (4) References  
 1507: (4) -----  
 1508: (4) .. [1] MATLAB reference documentation, "Rank"  
 1509: (11) https://www.mathworks.com/help/techdoc/ref/rank.html  
 1510: (4) .. [2] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery,  
 1511: (11) "Numerical Recipes (3rd edition)", Cambridge University Press,  
 2007,  
 1512: (11) page 795.

1513: (4) Examples  
 1514: (4) -----  
 1515: (4) >>> from numpy.linalg import matrix\_rank  
 1516: (4) >>> matrix\_rank(np.eye(4)) # Full rank matrix  
 1517: (4) 4  
 1518: (4) >>> I=np.eye(4); I[-1,-1] = 0. # rank deficient matrix  
 1519: (4) >>> matrix\_rank(I)  
 1520: (4) 3  
 1521: (4) >>> matrix\_rank(np.ones((4,))) # 1 dimension - rank 1 unless all 0  
 1522: (4) 1  
 1523: (4) >>> matrix\_rank(np.zeros((4,)))  
 1524: (4) 0  
 1525: (4) """"  
 1526: (4) A = asarray(A)  
 1527: (4) if A.ndim < 2:  
 1528: (8) return int(not all(A==0))  
 1529: (4) S = svd(A, compute\_uv=False, hermitian=hermitian)  
 1530: (4) if tol is None:  
 1531: (8) tol = S.max(axis=-1, keepdims=True) \* max(A.shape[-2:]) \*  
 1532: (4) else:  
 1533: (8) tol = asarray(tol)[..., newaxis]  
 1534: (4) return count\_nonzero(S > tol, axis=-1)  
 1535: (0) def \_pinv\_dispatcher(a, rcond=None, hermitian=None):  
 1536: (4) return (a,)  
 1537: (0) @array\_function\_dispatch(\_pinv\_dispatcher)  
 1538: (0) def pinv(a, rcond=1e-15, hermitian=False):  
 1539: (4) """  
 1540: (4) Compute the (Moore-Penrose) pseudo-inverse of a matrix.  
 1541: (4) Calculate the generalized inverse of a matrix using its  
 1542: (4) singular-value decomposition (SVD) and including all  
 1543: (4) \*large\* singular values.  
 1544: (4) .. versionchanged:: 1.14  
 1545: (7) Can now operate on stacks of matrices  
 1546: (4) Parameters  
 1547: (4) -----  
 1548: (4) a : (..., M, N) array\_like  
 1549: (8) Matrix or stack of matrices to be pseudo-inverted.  
 1550: (4) rcond : (...) array\_like of float  
 1551: (8) Cutoff for small singular values.  
 1552: (8) Singular values less than or equal to  
 1553: (8) ``rcond \* largest\_singular\_value`` are set to zero.  
 1554: (8) Broadcasts against the stack of matrices.  
 1555: (4) hermitian : bool, optional  
 1556: (8) If True, `a` is assumed to be Hermitian (symmetric if real-valued),  
 1557: (8) enabling a more efficient method for finding singular values.  
 1558: (8) Defaults to False.  
 1559: (8) .. versionadded:: 1.17.0  
 1560: (4) Returns  
 1561: (4) -----  
 1562: (4) B : (..., N, M) ndarray  
 1563: (8) The pseudo-inverse of `a`. If `a` is a `matrix` instance, then so

```

1564: (8)           is `B`.
1565: (4)           Raises
1566: (4)           -----
1567: (4)           LinAlgError
1568: (8)           If the SVD computation does not converge.
1569: (4)           See Also
1570: (4)           -----
1571: (4)           scipy.linalg.pinv : Similar function in SciPy.
1572: (4)           scipy.linalg.pinvh : Compute the (Moore-Penrose) pseudo-inverse of a
1573: (25)           Hermitian matrix.
1574: (4)           Notes
1575: (4)           -----
1576: (4)           The pseudo-inverse of a matrix A, denoted :math:`A^+`, is
1577: (4)           defined as: "the matrix that 'solves' [the least-squares problem]
1578: (4)           :math:`Ax = b`," i.e., if :math:`\bar{x}` is said solution, then
1579: (4)           :math:`A^+` is that matrix such that :math:`\bar{x} = A^+b`.
1580: (4)           It can be shown that if :math:`Q_1 \Sigma Q_2^T = A` is the singular
1581: (4)           value decomposition of A, then
1582: (4)           :math:`A^+ = Q_2 \Sigma^+ Q_1^T`, where :math:`Q_{1,2}` are
1583: (4)           orthogonal matrices, :math:`\Sigma` is a diagonal matrix consisting
1584: (4)           of A's so-called singular values, (followed, typically, by
1585: (4)           zeros), and then :math:`\Sigma^+` is simply the diagonal matrix
1586: (4)           consisting of the reciprocals of A's singular values
1587: (4)           (again, followed by zeros). [1]_
1588: (4)           References
1589: (4)           -----
1590: (4)           .. [1] G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando,
1591: (11)             FL, Academic Press, Inc., 1980, pp. 139-142.
1592: (4)           Examples
1593: (4)           -----
1594: (4)           The following example checks that ``a * a+ * a == a`` and
1595: (4)           ``a+ * a * a+ == a+``:
1596: (4)           >>> a = np.random.randn(9, 6)
1597: (4)           >>> B = np.linalg.pinv(a)
1598: (4)           >>> np.allclose(a, np.dot(a, np.dot(B, a)))
1599: (4)           True
1600: (4)           >>> np.allclose(B, np.dot(B, np.dot(a, B)))
1601: (4)           True
1602: (4)           """
1603: (4)           a, wrap = _makearray(a)
1604: (4)           rcond = asarray(rcond)
1605: (4)           if _is_empty_2d(a):
1606: (8)             m, n = a.shape[-2:]
1607: (8)             res = empty(a.shape[:-2] + (n, m), dtype=a.dtype)
1608: (8)             return wrap(res)
1609: (4)           a = a.conjugate()
1610: (4)           u, s, vt = svd(a, full_matrices=False, hermitian=hermitian)
1611: (4)           cutoff = rcond[..., newaxis] * amax(s, axis=-1, keepdims=True)
1612: (4)           large = s > cutoff
1613: (4)           s = divide(1, s, where=large, out=s)
1614: (4)           s[~large] = 0
1615: (4)           res = matmul(transpose(vt), multiply(s[..., newaxis], transpose(u)))
1616: (4)           return wrap(res)
1617: (0)           @array_function_dispatch(_unary_dispatcher)
1618: (0)           def slogdet(a):
1619: (4)             """
1620: (4)             Compute the sign and (natural) logarithm of the determinant of an array.
1621: (4)             If an array has a very small or very large determinant, then a call to
1622: (4)             `det` may overflow or underflow. This routine is more robust against such
1623: (4)             issues, because it computes the logarithm of the determinant rather than
1624: (4)             the determinant itself.
1625: (4)             Parameters
1626: (4)             -----
1627: (4)             a : (... , M, M) array_like
1628: (8)               Input array, has to be a square 2-D array.
1629: (4)             Returns
1630: (4)             -----
1631: (4)             A namedtuple with the following attributes:
1632: (4)               sign : (...) array_like

```

```

1633: (8) A number representing the sign of the determinant. For a real matrix,
1634: (8) this is 1, 0, or -1. For a complex matrix, this is a complex number
1635: (8) with absolute value 1 (i.e., it is on the unit circle), or else 0.
1636: (4) logabsdet : (...) array_like
1637: (8) The natural log of the absolute value of the determinant.
1638: (4) If the determinant is zero, then `sign` will be 0 and `logabsdet` will be
1639: (4) -Inf. In all cases, the determinant is equal to ``sign *
np.exp(logabsdet)``
1640: (4) See Also
1641: (4) -----
1642: (4) det
1643: (4) Notes
1644: (4) -----
1645: (4) .. versionadded:: 1.8.0
1646: (4) Broadcasting rules apply, see the `numpy.linalg` documentation for
1647: (4) details.
1648: (4) .. versionadded:: 1.6.0
1649: (4) The determinant is computed via LU factorization using the LAPACK
1650: (4) routine ``z/dgetrf``.
1651: (4) Examples
1652: (4) -----
1653: (4) The determinant of a 2-D array ``[[a, b], [c, d]]`` is ``ad - bc``:
1654: (4) >>> a = np.array([[1, 2], [3, 4]])
1655: (4) >>> (sign, logabsdet) = np.linalg.slogdet(a)
1656: (4) >>> (sign, logabsdet)
1657: (4) (-1, 0.69314718055994529) # may vary
1658: (4) >>> sign * np.exp(logabsdet)
1659: (4) -2.0
1660: (4) Computing log-determinants for a stack of matrices:
1661: (4) >>> a = np.array([ [[1, 2], [3, 4]], [[1, 2], [2, 1]], [[1, 3], [3, 1]] ])
1662: (4) >>> a.shape
1663: (4) (3, 2, 2)
1664: (4) >>> sign, logabsdet = np.linalg.slogdet(a)
1665: (4) >>> (sign, logabsdet)
1666: (4) (array([-1., -1., -1.]), array([ 0.69314718,  1.09861229,  2.07944154]))
1667: (4) >>> sign * np.exp(logabsdet)
1668: (4) array([-2., -3., -8.])
1669: (4) This routine succeeds where ordinary `det` does not:
1670: (4) >>> np.linalg.det(np.eye(500) * 0.1)
1671: (4) 0.0
1672: (4) >>> np.linalg.slogdet(np.eye(500) * 0.1)
1673: (4) (1, -1151.2925464970228)
1674: (4) """
1675: (4) a = asarray(a)
1676: (4) _assert_stacked_2d(a)
1677: (4) _assert_stacked_square(a)
1678: (4) t, result_t = _commonType(a)
1679: (4) real_t = _realType(result_t)
1680: (4) signature = 'D->Dd' if isComplexType(t) else 'd->dd'
1681: (4) sign, logdet = _umath_linalg.slogdet(a, signature=signature)
1682: (4) sign = sign.astype(result_t, copy=False)
1683: (4) logdet = logdet.astype(real_t, copy=False)
1684: (4) return SlogdetResult(sign, logdet)
1685: (0) @array_function_dispatch(_unary_dispatcher)
1686: (0) def det(a):
1687: (4) """
1688: (4) Compute the determinant of an array.
1689: (4) Parameters
1690: (4) -----
1691: (4) a : (... , M, M) array_like
1692: (8) Input array to compute determinants for.
1693: (4) Returns
1694: (4) -----
1695: (4) det : (...) array_like
1696: (8) Determinant of `a`.
1697: (4) See Also
1698: (4) -----
1699: (4) slogdet : Another way to represent the determinant, more suitable
1700: (6) for large matrices where underflow/overflow may occur.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1701: (4)          scipy.linalg.det : Similar function in SciPy.
1702: (4)          Notes
1703: (4)          -----
1704: (4)          .. versionadded:: 1.8.0
1705: (4)          Broadcasting rules apply, see the `numpy.linalg` documentation for
1706: (4)          details.
1707: (4)          The determinant is computed via LU factorization using the LAPACK
1708: (4)          routine ``z/dgetrf``.
1709: (4)          Examples
1710: (4)          -----
1711: (4)          The determinant of a 2-D array [[a, b], [c, d]] is ad - bc:
1712: (4)          >>> a = np.array([[1, 2], [3, 4]])
1713: (4)          >>> np.linalg.det(a)
1714: (4)          -2.0 # may vary
1715: (4)          Computing determinants for a stack of matrices:
1716: (4)          >>> a = np.array([ [[1, 2], [3, 4]], [[1, 2], [2, 1]], [[1, 3], [3, 1]] ])
1717: (4)          >>> a.shape
1718: (4)          (3, 2, 2)
1719: (4)          >>> np.linalg.det(a)
1720: (4)          array([-2., -3., -8.])
1721: (4)          """
1722: (4)          a = asarray(a)
1723: (4)          _assert_stacked_2d(a)
1724: (4)          _assert_stacked_square(a)
1725: (4)          t, result_t = _commonType(a)
1726: (4)          signature = 'D->D' if isComplexType(t) else 'd->d'
1727: (4)          r = _umath_linalg.det(a, signature=signature)
1728: (4)          r = r.astype(result_t, copy=False)
1729: (4)          return r
1730: (0)          def _lstsq_dispatcher(a, b, rcond=None):
1731: (4)          return (a, b)
1732: (0)          @array_function_dispatch(_lstsq_dispatcher)
1733: (0)          def lstsq(a, b, rcond="warn"):
1734: (4)          """
1735: (4)          Return the least-squares solution to a linear matrix equation.
1736: (4)          Computes the vector `x` that approximately solves the equation
1737: (4)          ``a @ x = b``. The equation may be under-, well-, or over-determined
1738: (4)          (i.e., the number of linearly independent rows of `a` can be less than,
1739: (4)          equal to, or greater than its number of linearly independent columns).
1740: (4)          If `a` is square and of full rank, then `x` (but for round-off error)
1741: (4)          is the "exact" solution of the equation. Else, `x` minimizes the
1742: (4)          Euclidean 2-norm :math:`||b - ax||`. If there are multiple minimizing
1743: (4)          solutions, the one with the smallest 2-norm :math:`||x||` is returned.
1744: (4)          Parameters
1745: (4)          -----
1746: (4)          a : (M, N) array_like
1747: (8)          "Coefficient" matrix.
1748: (4)          b : {(M,), (M, K)} array_like
1749: (8)          Ordinate or "dependent variable" values. If `b` is two-dimensional,
1750: (8)          the least-squares solution is calculated for each of the `K` columns
1751: (8)          of `b`.
1752: (4)          rcond : float, optional
1753: (8)          Cut-off ratio for small singular values of `a`.
1754: (8)          For the purposes of rank determination, singular values are treated
1755: (8)          as zero if they are smaller than `rcond` times the largest singular
1756: (8)          value of `a`.
1757: (8)          .. versionchanged:: 1.14.0
1758: (11)          If not set, a FutureWarning is given. The previous default
1759: (11)          of ``-1`` will use the machine precision as `rcond` parameter,
1760: (11)          the new default will use the machine precision times `max(M, N)` .
1761: (11)          To silence the warning and use the new default, use ``rcond=None`` ,
1762: (11)          to keep using the old behavior, use ``rcond=-1`` .
1763: (4)          Returns
1764: (4)          -----
1765: (4)          x : {(N,), (N, K)} ndarray
1766: (8)          Least-squares solution. If `b` is two-dimensional,
1767: (8)          the solutions are in the `K` columns of `x`.
1768: (4)          residuals : {(1,), (K,), (0,)} ndarray
1769: (8)          Sums of squared residuals: Squared Euclidean 2-norm for each column in

```

```

1770: (8)          ``b - a @ x``.
1771: (8)          If the rank of `a` is < N or M <= N, this is an empty array.
1772: (8)          If `b` is 1-dimensional, this is a (1,) shape array.
1773: (8)          Otherwise the shape is (K,).
1774: (4)          rank : int
1775: (8)          Rank of matrix `a`.
1776: (4)          s : (min(M, N),) ndarray
1777: (8)          Singular values of `a`.
1778: (4)          Raises
1779: (4)          -----
1780: (4)          LinAlgError
1781: (8)          If computation does not converge.
1782: (4)          See Also
1783: (4)          -----
1784: (4)          scipy.linalg.lstsq : Similar function in SciPy.
1785: (4)          Notes
1786: (4)          -----
1787: (4)          If `b` is a matrix, then all array results are returned as matrices.
1788: (4)          Examples
1789: (4)          -----
1790: (4)          Fit a line, ``y = mx + c``, through some noisy data-points:
1791: (4)          >>> x = np.array([0, 1, 2, 3])
1792: (4)          >>> y = np.array([-1, 0.2, 0.9, 2.1])
1793: (4)          By examining the coefficients, we see that the line should have a
1794: (4)          gradient of roughly 1 and cut the y-axis at, more or less, -1.
1795: (4)          We can rewrite the line equation as ``y = Ap``, where ``A = [[x 1]]``
1796: (4)          and ``p = [[m], [c]]``. Now use `lstsq` to solve for `p`:
1797: (4)          >>> A = np.vstack([x, np.ones(len(x))]).T
1798: (4)          >>> A
1799: (4)          array([[ 0.,  1.],
1800: (11)            [ 1.,  1.],
1801: (11)            [ 2.,  1.],
1802: (11)            [ 3.,  1.]])
1803: (4)          >>> m, c = np.linalg.lstsq(A, y, rcond=None)[0]
1804: (4)
1805: (4)          (1.0 -0.95) # may vary
1806: (4)          Plot the data along with the fitted line:
1807: (4)          >>> import matplotlib.pyplot as plt
1808: (4)          >>> _ = plt.plot(x, y, 'o', label='Original data', markersize=10)
1809: (4)          >>> _ = plt.plot(x, m*x + c, 'r', label='Fitted line')
1810: (4)          >>> _ = plt.legend()
1811: (4)          >>> plt.show()
1812: (4)          """
1813: (4)          a, _ = _makearray(a)
1814: (4)          b, wrap = _makearray(b)
1815: (4)          is_1d = b.ndim == 1
1816: (4)          if is_1d:
1817: (8)              b = b[:, newaxis]
1818: (4)          _assert_2d(a, b)
1819: (4)          m, n = a.shape[-2:]
1820: (4)          m2, n_rhs = b.shape[-2:]
1821: (4)          if m != m2:
1822: (8)              raise LinAlgError('Incompatible dimensions')
1823: (4)          t, result_t = _commonType(a, b)
1824: (4)          result_real_t = _realType(result_t)
1825: (4)          if rcond == "warn":
1826: (8)              warnings.warn(`rcond` parameter will change to the default of "
1827: (22)                  "machine precision times ``max(M, N)`` where M and N "
1828: (22)                  "are the input matrix dimensions.\n"
1829: (22)                  "To use the future default and silence this warning "
1830: (22)                  "we advise to pass `rcond=None`, to keep using the old,
"
1831: (22)                  "explicitly pass `rcond=-1`.",
1832: (22)                  FutureWarning, stacklevel=2)
1833: (8)          rcond = -1
1834: (4)          if rcond is None:
1835: (8)              rcond = finfo(t).eps * max(n, m)
1836: (4)          if m <= n:
1837: (8)              gufunc = _umath_linalg.lstsq_m

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1838: (4)
1839: (8)
1840: (4)
1841: (4)
1842: (4)
1843: (8)
1844: (4)
extobj=extobj)
1845: (4)
1846: (8)
1847: (4)
1848: (8)
1849: (8)
1850: (4)
1851: (8)
1852: (4)
1853: (8)
1854: (4)
1855: (4)
1856: (4)
freed
1857: (4)
1858: (0)
def _multi_svd_norm(x, row_axis, col_axis, op):
    """Compute a function of the singular values of the 2-D matrices in `x`.
    This is a private utility function used by `numpy.linalg.norm()`.

    Parameters
    -----
    x : ndarray
        The axes of `x` that hold the 2-D matrices.
    row_axis, col_axis : int
        The axes of `x` that hold the 2-D matrices.
    op : callable
        This should be either `numpy.amin` or `numpy.amax` or `numpy.sum`.

    Returns
    -----
    result : float or ndarray
        If `x` is 2-D, the return values is a float.
        Otherwise, it is an array with ``x.ndim - 2`` dimensions.
        The return values are either the minimum or maximum or sum of the
        singular values of the matrices, depending on whether `op`
        is `numpy.amin` or `numpy.amax` or `numpy.sum`.

    """
    y = moveaxis(x, (row_axis, col_axis), (-2, -1))
    result = op(svd(y, compute_uv=False), axis=-1)
    return result
def _norm_dispatcher(x, ord=None, axis=None, keepdims=None):
    return (x,)
@array_function_dispatch(_norm_dispatcher)
def norm(x, ord=None, axis=None, keepdims=False):
    """
    Matrix or vector norm.

    This function is able to return one of eight different matrix norms,
    or one of an infinite number of vector norms (described below), depending
    on the value of the ``ord`` parameter.

    Parameters
    -----
    x : array_like
        Input array. If `axis` is None, `x` must be 1-D or 2-D, unless `ord`
        is None. If both `axis` and `ord` are None, the 2-norm of
        ``x.ravel`` will be returned.
    ord : {non-zero int, inf, -inf, 'fro', 'nuc'}, optional
        Order of the norm (see table under ``Notes``). inf means numpy's
        `inf` object. The default is None.
    axis : {None, int, 2-tuple of ints}, optional.
        If `axis` is an integer, it specifies the axis of `x` along which to
        compute the vector norms. If `axis` is a 2-tuple, it specifies the
        axes that hold 2-D matrices, and the matrix norms of these matrices
        are computed. If `axis` is None then either a vector norm (when `x`
        is 1-D) or a matrix norm (when `x` is 2-D) is returned. The default
        is None.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1905: (8) .. versionadded:: 1.8.0
1906: (4) keepdims : bool, optional
1907: (8) If this is set to True, the axes which are normed over are left in the
1908: (8) result as dimensions with size one. With this option the result will
1909: (8) broadcast correctly against the original `x`.
1910: (8) .. versionadded:: 1.10.0
1911: (4) Returns
1912: (4)
1913: (4) n : float or ndarray
1914: (8) Norm of the matrix or vector(s).
1915: (4) See Also
1916: (4)
1917: (4) scipy.linalg.norm : Similar function in SciPy.
1918: (4) Notes
1919: (4)
1920: (4) -----
1921: (4) For values of ``ord < 1``, the result is, strictly speaking, not a
1922: (4) mathematical 'norm', but it may still be useful for various numerical
1923: (4) purposes.
1924: (4) The following norms can be calculated:
1925: (4) ===== norm for matrices norm for vectors
1926: (4) =====
1927: (4) None Frobenius norm 2-norm
1928: (4) 'fro' Frobenius norm --
1929: (4) 'nuc' nuclear norm --
1930: (4) inf max(sum(abs(x), axis=1)) max(abs(x))
1931: (4) -inf min(sum(abs(x), axis=1)) min(abs(x))
1932: (4) 0 -- sum(x != 0)
1933: (4) 1 max(sum(abs(x), axis=0)) as below
1934: (4) -1 min(sum(abs(x), axis=0)) as below
1935: (4) 2 2-norm (largest sing. value) as below
1936: (4) -2 smallest singular value as below
1937: (4) other -- sum(abs(x)**ord)**(1./ord)
1938: (4) =====
1939: (4) The Frobenius norm is given by [1]_:
1940: (8) :math: `||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}`
1941: (4) The nuclear norm is the sum of the singular values.
1942: (4) Both the Frobenius and nuclear norm orders are only defined for
1943: (4) matrices and raise a ValueError when ``x.ndim != 2``.
1944: (4) References
1945: (4)
1946: (4) .. [1] G. H. Golub and C. F. Van Loan, *Matrix Computations*,  

1947: (11) Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15
1948: (4) Examples
1949: (4)
1950: (4) >>> from numpy import linalg as LA
1951: (4) >>> a = np.arange(9) - 4
1952: (4) >>> a
1953: (4) array([-4, -3, -2, ..., 2, 3, 4])
1954: (4) >>> b = a.reshape((3, 3))
1955: (4) >>> b
1956: (4) array([[-4, -3, -2],  

1957: (11) [-1, 0, 1],  

1958: (11) [2, 3, 4]])
1959: (4) >>> LA.norm(a)
1960: (4) 7.745966692414834
1961: (4) >>> LA.norm(b)
1962: (4) 7.745966692414834
1963: (4) >>> LA.norm(b, 'fro')
1964: (4) 7.745966692414834
1965: (4) >>> LA.norm(a, np.inf)
1966: (4) 4.0
1967: (4) >>> LA.norm(b, np.inf)
1968: (4) 9.0
1969: (4) >>> LA.norm(a, -np.inf)
1970: (4) 0.0
1971: (4) >>> LA.norm(b, -np.inf)
1972: (4) 2.0
1973: (4) >>> LA.norm(a, 1)

```

```

1974: (4)                                20.0
1975: (4)                                >>> LA.norm(b, 1)
1976: (4)                                7.0
1977: (4)                                >>> LA.norm(a, -1)
1978: (4)                                -4.6566128774142013e-010
1979: (4)                                >>> LA.norm(b, -1)
1980: (4)                                6.0
1981: (4)                                >>> LA.norm(a, 2)
1982: (4)                                7.745966692414834
1983: (4)                                >>> LA.norm(b, 2)
1984: (4)                                7.3484692283495345
1985: (4)                                >>> LA.norm(a, -2)
1986: (4)                                0.0
1987: (4)                                >>> LA.norm(b, -2)
1988: (4)                                1.8570331885190563e-016 # may vary
1989: (4)                                >>> LA.norm(a, 3)
1990: (4)                                5.8480354764257312 # may vary
1991: (4)                                >>> LA.norm(a, -3)
1992: (4)                                0.0
1993: (4)                                Using the `axis` argument to compute vector norms:
1994: (4)                                >>> c = np.array([[ 1, 2, 3],
1995: (4)                                     ...                   [-1, 1, 4]])
1996: (4)                                >>> LA.norm(c, axis=0)
1997: (4)                                array([ 1.41421356,  2.23606798,  5.           ])
1998: (4)                                >>> LA.norm(c, axis=1)
1999: (4)                                array([ 3.74165739,  4.24264069])
2000: (4)                                >>> LA.norm(c, ord=1, axis=1)
2001: (4)                                array([ 6.,  6.])
2002: (4)                                Using the `axis` argument to compute matrix norms:
2003: (4)                                >>> m = np.arange(8).reshape(2,2,2)
2004: (4)                                >>> LA.norm(m, axis=(1,2))
2005: (4)                                array([ 3.74165739, 11.22497216])
2006: (4)                                >>> LA.norm(m[0, :, :], LA.norm(m[1, :, :]))
2007: (4)                                (3.7416573867739413, 11.224972160321824)
2008: (4)                                """
2009: (4)                                x = asarray(x)
2010: (4)                                if not issubclass(x.dtype.type, (inexact, object_)):
2011: (8)                                    x = x.astype(float)
2012: (4)                                if axis is None:
2013: (8)                                    ndim = x.ndim
2014: (8)                                    if ((ord is None) or
2015: (12)                                         (ord in ('f', 'fro') and ndim == 2) or
2016: (12)                                         (ord == 2 and ndim == 1)):
2017: (12)                                        x = x.ravel(order='K')
2018: (12)                                        if isComplexType(x.dtype.type):
2019: (16)                                            x_real = x.real
2020: (16)                                            x_imag = x.imag
2021: (16)                                            sqnorm = x_real.dot(x_real) + x_imag.dot(x_imag)
2022: (12)                                        else:
2023: (16)                                            sqnorm = x.dot(x)
2024: (12)                                            ret = sqrt(sqnorm)
2025: (12)                                            if keepdims:
2026: (16)                                                ret = ret.reshape(ndim*[1])
2027: (12)                                            return ret
2028: (4)                                nd = x.ndim
2029: (4)                                if axis is None:
2030: (8)                                    axis = tuple(range(nd))
2031: (4)                                elif not isinstance(axis, tuple):
2032: (8)                                    try:
2033: (12)                                        axis = int(axis)
2034: (8)                                    except Exception as e:
2035: (12)                                        raise TypeError("'axis' must be None, an integer or a tuple of
integers") from e
2036: (8)                                axis = (axis,)
2037: (4)                                if len(axis) == 1:
2038: (8)                                    if ord == Inf:
2039: (12)                                        return abs(x).max(axis=axis, keepdims=keepdims)
2040: (8)                                    elif ord == -Inf:
2041: (12)                                        return abs(x).min(axis=axis, keepdims=keepdims)

```

```

2042: (8)           elif ord == 0:
2043: (12)             return (x != 0).astype(x.real.dtype).sum(axis=axis,
keepdims=keepdims)
2044: (8)           elif ord == 1:
2045: (12)             return add.reduce(abs(x), axis=axis, keepdims=keepdims)
2046: (8)           elif ord is None or ord == 2:
2047: (12)             s = (x.conj() * x).real
2048: (12)             return sqrt(add.reduce(s, axis=axis, keepdims=keepdims))
2049: (8)           elif isinstance(ord, str):
2050: (12)             raise ValueError(f"Invalid norm order '{ord}' for vectors")
2051: (8)           else:
2052: (12)             absx = abs(x)
2053: (12)             absx **= ord
2054: (12)             ret = add.reduce(absx, axis=axis, keepdims=keepdims)
2055: (12)             ret **= reciprocal(ord, dtype=ret.dtype)
2056: (12)             return ret
2057: (4)           elif len(axis) == 2:
2058: (8)             row_axis, col_axis = axis
2059: (8)             row_axis = normalize_axis_index(row_axis, nd)
2060: (8)             col_axis = normalize_axis_index(col_axis, nd)
2061: (8)             if row_axis == col_axis:
2062: (12)               raise ValueError('Duplicate axes given.')
2063: (8)           if ord == 2:
2064: (12)             ret = _multi_svd_norm(x, row_axis, col_axis, amax)
2065: (8)           elif ord == -2:
2066: (12)             ret = _multi_svd_norm(x, row_axis, col_axis, amin)
2067: (8)           elif ord == 1:
2068: (12)             if col_axis > row_axis:
2069: (16)               col_axis -= 1
2070: (12)             ret = add.reduce(abs(x), axis=row_axis).max(axis=col_axis)
2071: (8)           elif ord == Inf:
2072: (12)             if row_axis > col_axis:
2073: (16)               row_axis -= 1
2074: (12)             ret = add.reduce(abs(x), axis=col_axis).max(axis=row_axis)
2075: (8)           elif ord == -1:
2076: (12)             if col_axis > row_axis:
2077: (16)               col_axis -= 1
2078: (12)             ret = add.reduce(abs(x), axis=row_axis).min(axis=col_axis)
2079: (8)           elif ord == -Inf:
2080: (12)             if row_axis > col_axis:
2081: (16)               row_axis -= 1
2082: (12)             ret = add.reduce(abs(x), axis=col_axis).min(axis=row_axis)
2083: (8)           elif ord in [None, 'fro', 'f']:
2084: (12)             ret = sqrt(add.reduce((x.conj() * x).real, axis=axis))
2085: (8)           elif ord == 'nuc':
2086: (12)             ret = _multi_svd_norm(x, row_axis, col_axis, sum)
2087: (8)           else:
2088: (12)             raise ValueError("Invalid norm order for matrices.")
2089: (8)           if keepdims:
2090: (12)             ret_shape = list(x.shape)
2091: (12)             ret_shape[axis[0]] = 1
2092: (12)             ret_shape[axis[1]] = 1
2093: (12)             ret = ret.reshape(ret_shape)
2094: (8)           return ret
2095: (4)           else:
2096: (8)             raise ValueError("Improper number of dimensions to norm.")
2097: (0)           def _multidot_dispatcher(arrays, *, out=None):
2098: (4)             yield from arrays
2099: (4)             yield out
2100: (0)           @array_function_dispatch(_multidot_dispatcher)
2101: (0)           def multi_dot(arrays, *, out=None):
2102: (4)             """
2103: (4)             Compute the dot product of two or more arrays in a single function call,
2104: (4)             while automatically selecting the fastest evaluation order.
2105: (4)             `multi_dot` chains `numpy.dot` and uses optimal parenthesization
2106: (4)             of the matrices [1]_ [2]_. Depending on the shapes of the matrices,
2107: (4)             this can speed up the multiplication a lot.
2108: (4)             If the first argument is 1-D it is treated as a row vector.
2109: (4)             If the last argument is 1-D it is treated as a column vector.

```

```

2110: (4)           The other arguments must be 2-D.
2111: (4)           Think of `multi_dot` as::
2112: (8)             def multi_dot(arrays): return functools.reduce(np.dot, arrays)
2113: (4)           Parameters
2114: (4)             -----
2115: (4)             arrays : sequence of array_like
2116: (8)               If the first argument is 1-D it is treated as row vector.
2117: (8)               If the last argument is 1-D it is treated as column vector.
2118: (8)               The other arguments must be 2-D.
2119: (4)             out : ndarray, optional
2120: (8)               Output argument. This must have the exact kind that would be returned
2121: (8)               if it was not used. In particular, it must have the right type, must
be
2122: (8)               C-contiguous, and its dtype must be the dtype that would be returned
2123: (8)               for `dot(a, b)`. This is a performance feature. Therefore, if these
2124: (8)               conditions are not met, an exception is raised, instead of attempting
2125: (8)               to be flexible.
2126: (8)               .. versionadded:: 1.19.0
2127: (4)           Returns
2128: (4)             -----
2129: (4)             output : ndarray
2130: (8)               Returns the dot product of the supplied arrays.
2131: (4)           See Also
2132: (4)             -----
2133: (4)             numpy.dot : dot multiplication with two arguments.
2134: (4)           References
2135: (4)             -----
2136: (4)               .. [1] Cormen, "Introduction to Algorithms", Chapter 15.2, p. 370-378
2137: (4)               .. [2] https://en.wikipedia.org/wiki/Matrix\_chain\_multiplication
2138: (4)           Examples
2139: (4)             -----
2140: (4)               `multi_dot` allows you to write::
2141: (4)               >>> from numpy.linalg import multi_dot
2142: (4)               >>> # Prepare some data
2143: (4)               >>> A = np.random.random((10000, 100))
2144: (4)               >>> B = np.random.random((100, 1000))
2145: (4)               >>> C = np.random.random((1000, 5))
2146: (4)               >>> D = np.random.random((5, 333))
2147: (4)               >>> # the actual dot multiplication
2148: (4)               >>> _ = multi_dot([A, B, C, D])
2149: (4)               instead of::
2150: (4)               >>> _ = np.dot(np.dot(np.dot(A, B), C), D)
2151: (4)               >>> # or
2152: (4)               >>> _ = A.dot(B).dot(C).dot(D)
2153: (4)           Notes
2154: (4)             -----
2155: (4)               The cost for a matrix multiplication can be calculated with the
2156: (4)               following function::
2157: (8)               def cost(A, B):
2158: (12)                 return A.shape[0] * A.shape[1] * B.shape[1]
2159: (4)           Assume we have three matrices
2160: (4)             :math:`A_{10x100}, B_{100x5}, C_{5x50}`.
2161: (4)           The costs for the two different parenthesizations are as follows::
2162: (8)             cost((AB)C) = 10*100*5 + 10*5*50 = 5000 + 2500 = 7500
2163: (8)             cost(A(BC)) = 10*100*50 + 100*5*50 = 50000 + 25000 = 75000
2164: (4)             """
2165: (4)             n = len(arrays)
2166: (4)             if n < 2:
2167: (8)                 raise ValueError("Expecting at least two arrays.")
2168: (4)             elif n == 2:
2169: (8)                 return dot(arrays[0], arrays[1], out=out)
2170: (4)             arrays = [asanyarray(a) for a in arrays]
2171: (4)             ndim_first, ndim_last = arrays[0].ndim, arrays[-1].ndim
2172: (4)             if arrays[0].ndim == 1:
2173: (8)                 arrays[0] = atleast_2d(arrays[0])
2174: (4)             if arrays[-1].ndim == 1:
2175: (8)                 arrays[-1] = atleast_2d(arrays[-1]).T
2176: (4)             _assert_2d(*arrays)
2177: (4)             if n == 3:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2178: (8)             result = _multi_dot_three(arrays[0], arrays[1], arrays[2], out=out)
2179: (4)         else:
2180: (8)             order = _multi_dot_matrix_chain_order(arrays)
2181: (8)             result = _multi_dot(arrays, order, 0, n - 1, out=out)
2182: (4)         if ndim_first == 1 and ndim_last == 1:
2183: (8)             return result[0, 0] # scalar
2184: (4)         elif ndim_first == 1 or ndim_last == 1:
2185: (8)             return result.ravel() # 1-D
2186: (4)         else:
2187: (8)             return result
2188: (0)     def _multi_dot_three(A, B, C, out=None):
2189: (4)         """
2190: (4)             Find the best order for three arrays and do the multiplication.
2191: (4)             For three arguments `_multi_dot_three` is approximately 15 times faster
2192: (4)             than `_multi_dot_matrix_chain_order`
2193: (4)         """
2194: (4)         a0, a1b0 = A.shape
2195: (4)         b1c0, c1 = C.shape
2196: (4)         cost1 = a0 * b1c0 * (a1b0 + c1)
2197: (4)         cost2 = a1b0 * c1 * (a0 + b1c0)
2198: (4)         if cost1 < cost2:
2199: (8)             return dot(dot(A, B), C, out=out)
2200: (4)         else:
2201: (8)             return dot(A, dot(B, C), out=out)
2202: (0)     def _multi_dot_matrix_chain_order(arrays, return_costs=False):
2203: (4)         """
2204: (4)             Return a np.array that encodes the optimal order of mutiplications.
2205: (4)             The optimal order array is then used by `_multi_dot()` to do the
2206: (4)             multiplication.
2207: (4)             Also return the cost matrix if `return_costs` is `True`
2208: (4)             The implementation CLOSELY follows Cormen, "Introduction to Algorithms",
2209: (4)             Chapter 15.2, p. 370-378. Note that Cormen uses 1-based indices.
2210: (8)             cost[i, j] = min([
2211: (12)                 cost[prefix] + cost[suffix] + cost_mult(prefix, suffix)
2212: (12)                 for k in range(i, j)])
2213: (4)         """
2214: (4)         n = len(arrays)
2215: (4)         p = [a.shape[0] for a in arrays] + [arrays[-1].shape[1]]
2216: (4)         m = zeros((n, n), dtype=double)
2217: (4)         s = empty((n, n), dtype=intp)
2218: (4)         for l in range(1, n):
2219: (8)             for i in range(n - 1):
2220: (12)                 j = i + 1
2221: (12)                 m[i, j] = Inf
2222: (12)                 for k in range(i, j):
2223: (16)                     q = m[i, k] + m[k+1, j] + p[i]*p[k+1]*p[j+1]
2224: (16)                     if q < m[i, j]:
2225: (20)                         m[i, j] = q
2226: (20)                         s[i, j] = k # Note that Cormen uses 1-based index
2227: (4)         return (s, m) if return_costs else s
2228: (0)     def _multi_dot(arrays, order, i, j, out=None):
2229: (4)         """Actually do the multiplication with the given order."""
2230: (4)         if i == j:
2231: (8)             assert out is None
2232: (8)             return arrays[i]
2233: (4)         else:
2234: (8)             return dot(_multi_dot(arrays, order, i, order[i, j]),
2235: (19)                             _multi_dot(arrays, order, order[i, j] + 1, j),
2236: (19)                             out=out)

```

---

File 256 - \_\_init\_\_.py:

```

1: (0)         """
2: (0)         ``numpy.linalg``
3: (0)         =====
4: (0)         The NumPy linear algebra functions rely on BLAS and LAPACK to provide
efficient

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

5: (0)           low level implementations of standard linear algebra algorithms. Those
6: (0)           libraries may be provided by NumPy itself using C versions of a subset of
their
7: (0)           reference implementations but, when possible, highly optimized libraries that
8: (0)           take advantage of specialized processor functionality are preferred. Examples
9: (0)           of such libraries are OpenBLAS, MKL (TM), and ATLAS. Because those libraries
10: (0)          are multithreaded and processor dependent, environmental variables and
external
11: (0)          packages such as threadpoolctl may be needed to control the number of threads
12: (0)          or specify the processor architecture.
- OpenBLAS: https://www.openblas.net/
- threadpoolctl: https://github.com/joblib/threadpoolctl
13: (0)          Please note that the most-used linear algebra functions in NumPy are present
in
14: (0)          the main ``numpy`` namespace rather than in ``numpy.linalg``. There are:
15: (0)          ``dot``, ``vdot``, ``inner``, ``outer``, ``matmul``, ``tensordot``,
``einsum``,
16: (0)          ``einsum_path`` and ``kron``.
17: (0)          Functions present in numpy.linalg are listed below.
18: (0)          Matrix and vector products
19: (0)          -----
20: (0)          multi_dot
21: (0)          matrix_power
22: (3)          Decompositions
23: (3)          -----
24: (0)          cholesky
25: (0)          qr
26: (3)          svd
27: (3)          Matrix eigenvalues
28: (3)          -----
29: (0)          eig
30: (0)          eigh
31: (3)          eigvals
32: (3)          eigvalsh
33: (3)          Norms and other numbers
34: (3)          -----
35: (0)          norm
36: (0)          cond
37: (3)          det
38: (3)          matrix_rank
39: (3)          slogdet
40: (3)          Solving equations and inverting matrices
41: (3)          -----
42: (0)          solve
43: (0)          tensorsolve
44: (3)          lstsq
45: (3)          inv
46: (3)          pinv
47: (3)          tensorinv
48: (3)          Exceptions
49: (3)          -----
50: (0)          LinAlgError
51: (0)          """
52: (3)          from . import linalg
53: (0)          from .linalg import *
54: (0)          __all__ = linalg.__all__.copy()
55: (0)          from numpy._pytesttester import PytestTester
56: (0)          test = PytestTester(__name__)
57: (0)          del PytestTester
58: (0)
59: (0)
-----
```

## File 257 - test\_deprecations.py:

```

1: (0)          """Test deprecation and future warnings.
2: (0)          """
3: (0)          import numpy as np
4: (0)          from numpy.testing import assert_warns
5: (0)          def test_qr_mode_full_future_warning():
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

6: (4)         """Check mode='full' FutureWarning.
7: (4)         In numpy 1.8 the mode options 'full' and 'economic' in linalg.qr were
8: (4)         deprecated. The release date will probably be sometime in the summer
9: (4)         of 2013.
10: (4)        """
11: (4)        a = np.eye(2)
12: (4)        assert_warnings(DeprecationWarning, np.linalg.qr, a, mode='full')
13: (4)        assert_warnings(DeprecationWarning, np.linalg.qr, a, mode='f')
14: (4)        assert_warnings(DeprecationWarning, np.linalg.qr, a, mode='economic')
15: (4)        assert_warnings(DeprecationWarning, np.linalg.qr, a, mode='e')

```

---

File 258 - test\_linalg.py:

```

1: (0)         """ Test functions for linalg module
2: (0)         """
3: (0)         import os
4: (0)         import sys
5: (0)         import itertools
6: (0)         import traceback
7: (0)         import textwrap
8: (0)         import subprocess
9: (0)         import pytest
10: (0)         import numpy as np
11: (0)         from numpy import array, single, double, csingle, cdouble, dot, identity,
matmul
12: (0)         from numpy.core import swapaxes
13: (0)         from numpy import multiply, atleast_2d, inf, asarray
14: (0)         from numpy import linalg
15: (0)         from numpy.linalg import matrix_power, norm, matrix_rank, multi_dot,
LinAlgError
16: (0)         from numpy.linalg.linalg import _multi_dot_matrix_chain_order
17: (0)         from numpy.testing import (
18: (4)             assert_, assert_equal, assert_raises, assert_array_equal,
19: (4)             assert_almost_equal, assert_allclose, suppress_warnings,
20: (4)             assert_raises_regex, HAS_LAPACK64, IS_WASM
21: (4)         )
22: (0)         try:
23: (4)             import numpy.linalg.lapack_lite
24: (0)         except ImportError:
25: (4)             pass
26: (0)         def consistent_subclass(out, in_):
27: (4)             return type(out) is (type(in_) if isinstance(in_, np.ndarray)
28: (25)                           else np.ndarray)
29: (0)         old_assert_almost_equal = assert_almost_equal
30: (0)         def assert_almost_equal(a, b, single_decimal=6, double_decimal=12, **kw):
31: (4)             if asarray(a).dtype.type in (single, csingle):
32: (8)                 decimal = single_decimal
33: (4)             else:
34: (8)                 decimal = double_decimal
35: (4)             old_assert_almost_equal(a, b, decimal=decimal, **kw)
36: (0)         def get_real_dtype(dtype):
37: (4)             return {single: single, double: double,
38: (12)                 csingle: single, cdouble: double}[dtype]
39: (0)         def get_complex_dtype(dtype):
40: (4)             return {single: csingle, double: cdouble,
41: (12)                 csingle: csingle, cdouble: cdouble}[dtype]
42: (0)         def get_rtol(dtype):
43: (4)             if dtype in (single, csingle):
44: (8)                 return 1e-5
45: (4)             else:
46: (8)                 return 1e-11
47: (0)         all_tags = {
48: (2)             'square', 'nonsquare', 'hermitian', # mutually exclusive
49: (2)             'generalized', 'size-0', 'strided' # optional additions
50: (0)
51: (0)
52: (4)             class LinalgCase:

```

```
                def __init__(self, name, a, b, tags=set()):
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

53: (8)           """
54: (8)             A bundle of arguments to be passed to a test case, with an identifying
55: (8)             name, the operands a and b, and a set of tags to filter the tests
56: (8)             """
57: (8)             assert_(isinstance(name, str))
58: (8)             self.name = name
59: (8)             self.a = a
60: (8)             self.b = b
61: (8)             self.tags = frozenset(tags) # prevent shared tags
62: (4)             def check(self, do):
63: (8)                 """
64: (8)                     Run the function `do` on this test case, expanding arguments
65: (8)                 """
66: (8)                     do(self.a, self.b, tags=self.tags)
67: (4)                     def __repr__(self):
68: (8)                         return f'LinalgCase: {self.name}''
69: (0)             def apply_tag(tag, cases):
70: (4)                 """
71: (4)                     Add the given tag (a string) to each of the cases (a list of LinalgCase
72: (4)                     objects)
73: (4)                     """
74: (4)                     assert tag in all_tags, "Invalid tag"
75: (4)                     for case in cases:
76: (8)                         case.tags = case.tags | {tag}
77: (4)                     return cases
78: (0)             np.random.seed(1234)
79: (0)             CASES = []
80: (0)             CASES += apply_tag('square', [
81: (4)                 LinalgCase("single",
82: (15)                   array([[1., 2.], [3., 4.]], dtype=single),
83: (15)                   array([2., 1.], dtype=single)),
84: (4)                 LinalgCase("double",
85: (15)                   array([[1., 2.], [3., 4.]], dtype=double),
86: (15)                   array([2., 1.], dtype=double)),
87: (4)                 LinalgCase("double_2",
88: (15)                   array([[1., 2.], [3., 4.]], dtype=double),
89: (15)                   array([[2., 1., 4.], [3., 4., 6.]], dtype=double)),
90: (4)                 LinalgCase("csingle",
91: (15)                   array([[1. + 2j, 2 + 3j], [3 + 4j, 4 + 5j]], dtype=csingle),
92: (15)                   array([2. + 1j, 1. + 2j], dtype=csingle)),
93: (4)                 LinalgCase("cdouble",
94: (15)                   array([[1. + 2j, 2 + 3j], [3 + 4j, 4 + 5j]], dtype=cdouble),
95: (15)                   array([2. + 1j, 1. + 2j], dtype=cdouble)),
96: (4)                 LinalgCase("cdouble_2",
97: (15)                   array([[1. + 2j, 2 + 3j], [3 + 4j, 4 + 5j]], dtype=cdouble),
98: (15)                   array([[2. + 1j, 1. + 2j, 1 + 3j], [1 - 2j, 1 - 3j, 1 - 6j]],

dtype=cdouble)),
99: (4)                 LinalgCase("0x0",
100: (15)                   np.empty((0, 0), dtype=double),
101: (15)                   np.empty((0,), dtype=double),
102: (15)                   tags={'size-0'}),
103: (4)                 LinalgCase("8x8",
104: (15)                   np.random.rand(8, 8),
105: (15)                   np.random.rand(8)),
106: (4)                 LinalgCase("1x1",
107: (15)                   np.random.rand(1, 1),
108: (15)                   np.random.rand(1)),
109: (4)                 LinalgCase("nonarray",
110: (15)                   [[1, 2], [3, 4]],
111: (15)                   [2, 1]),
112: (0)             ])
113: (0)             CASES += apply_tag('nonsquare', [
114: (4)                 LinalgCase("single_nsq_1",
115: (15)                   array([[1., 2., 3.], [3., 4., 6.]], dtype=single),
116: (15)                   array([2., 1.], dtype=single)),
117: (4)                 LinalgCase("single_nsq_2",
118: (15)                   array([[1., 2.], [3., 4.], [5., 6.]], dtype=single),
119: (15)                   array([2., 1., 3.], dtype=single)),
120: (4)                 LinalgCase("double_nsq_1",

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

121: (15)                                array([[1., 2., 3.], [3., 4., 6.]], dtype=double),
122: (15)                                array([2., 1.], dtype=double)),
123: (4)       LinalgCase("double_nsq_2",
124: (15)                                array([[1., 2.], [3., 4.], [5., 6.]], dtype=double),
125: (15)                                array([2., 1., 3.], dtype=double)),
126: (4)       LinalgCase("csingle_nsq_1",
127: (15)                                array(
128: (19)                                  [[1. + 1j, 2. + 2j, 3. - 3j], [3. - 5j, 4. + 9j, 6. + 2j]],
129: (15)                                dtype=csingle),
130: (4)       LinalgCase("csingle_nsq_2",
131: (15)                                array(
132: (19)                                  [[1. + 1j, 2. + 2j], [3. - 3j, 4. - 9j], [5. - 4j, 6. +
8j]], dtype=csingle),
133: (15)                                array([2. + 1j, 1. + 2j], dtype=csingle)),
134: (4)       LinalgCase("cdouble_nsq_1",
135: (15)                                array(
136: (19)                                  [[1. + 1j, 2. + 2j, 3. - 3j], [3. - 5j, 4. + 9j, 6. + 2j]],
137: (15)                                array([2. + 1j, 1. + 2j], dtype=cdouble)),
138: (4)       LinalgCase("cdouble_nsq_2",
139: (15)                                array(
140: (19)                                  [[1. + 1j, 2. + 2j], [3. - 3j, 4. - 9j], [5. - 4j, 6. +
8j]], dtype=cdouble),
141: (15)                                array([2. + 1j, 1. + 2j, 3. - 3j], dtype=cdouble)),
142: (4)       LinalgCase("cdouble_nsq_1_2",
143: (15)                                array(
144: (19)                                  [[1. + 1j, 2. + 2j, 3. - 3j], [3. - 5j, 4. + 9j, 6. + 2j]],
145: (15)                                array([[2. + 1j, 1. + 2j], [1 - 1j, 2 - 2j]], dtype=cdouble)),
146: (4)       LinalgCase("cdouble_nsq_2_2",
147: (15)                                array(
148: (19)                                  [[1. + 1j, 2. + 2j], [3. - 3j, 4. - 9j], [5. - 4j, 6. +
8j]], dtype=cdouble),
149: (15)                                array([[2. + 1j, 1. + 2j], [1 - 1j, 2 - 2j], [1 - 1j, 2 - 2j]],
150: (4)       LinalgCase("8x11",
151: (15)                                np.random.rand(8, 11),
152: (15)                                np.random.rand(8)),
153: (4)       LinalgCase("1x5",
154: (15)                                np.random.rand(1, 5),
155: (15)                                np.random.rand(1)),
156: (4)       LinalgCase("5x1",
157: (15)                                np.random.rand(5, 1),
158: (15)                                np.random.rand(5)),
159: (4)       LinalgCase("0x4",
160: (15)                                np.random.rand(0, 4),
161: (15)                                np.random.rand(0),
162: (15)                                tags={'size-0'}),
163: (4)       LinalgCase("4x0",
164: (15)                                np.random.rand(4, 0),
165: (15)                                np.random.rand(4),
166: (15)                                tags={'size-0'}),
167: (0)    ])
168: (0)    CASES += apply_tag('hermitian', [
169: (4)      LinalgCase("hsingle",
170: (15)                                array([[1., 2.], [2., 1.]], dtype=single),
171: (15)                                None),
172: (4)      LinalgCase("hdouble",
173: (15)                                array([[1., 2.], [2., 1.]], dtype=double),
174: (15)                                None),
175: (4)      LinalgCase("hcsingle",
176: (15)                                array([[1., 2 + 3j], [2 - 3j, 1]], dtype=csingle),
177: (15)                                None),
178: (4)      LinalgCase("hcdouble",
179: (15)                                array([[1., 2 + 3j], [2 - 3j, 1]], dtype=cdouble),
180: (15)                                None),
181: (4)      LinalgCase("hempty",
182: (15)                                np.empty((0, 0), dtype=double),

```

```

183: (15)           None,
184: (15)           tags={'size-0'}),
185: (4)            LinalgCase("hnonarray",
186: (15)             [[1, 2], [2, 1]],
187: (15)             None),
188: (4)            LinalgCase("matrix_b_only",
189: (15)             array([[1., 2.], [2., 1.]]),
190: (15)             None),
191: (4)            LinalgCase("hmatrix_1x1",
192: (15)             np.random.rand(1, 1),
193: (15)             None),
194: (0)
195: (0) def _make_generalized_cases():
196: (4)     new_cases = []
197: (4)     for case in CASES:
198: (8)         if not isinstance(case.a, np.ndarray):
199: (12)             continue
200: (8)         a = np.array([case.a, 2 * case.a, 3 * case.a])
201: (8)         if case.b is None:
202: (12)             b = None
203: (8)         else:
204: (12)             b = np.array([case.b, 7 * case.b, 6 * case.b])
205: (8)         new_case = LinalgCase(case.name + "_tile3", a, b,
206: (30)                         tags=case.tags | {'generalized'})
207: (8)         new_cases.append(new_case)
208: (8)         a = np.array([case.a] * 2 * 3).reshape((3, 2) + case.a.shape)
209: (8)         if case.b is None:
210: (12)             b = None
211: (8)         else:
212: (12)             b = np.array([case.b] * 2 * 3).reshape((3, 2) + case.b.shape)
213: (8)         new_case = LinalgCase(case.name + "_tile213", a, b,
214: (30)                         tags=case.tags | {'generalized'})
215: (8)         new_cases.append(new_case)
216: (4)     return new_cases
217: (0) CASES += _make_generalized_cases()
218: (0) def _stride_comb_iter(x):
219: (4)     """
220: (4)     Generate cartesian product of strides for all axes
221: (4)     """
222: (4)     if not isinstance(x, np.ndarray):
223: (8)         yield x, "nop"
224: (8)         return
225: (4)     stride_set = [(1,)] * x.ndim
226: (4)     stride_set[-1] = (1, 3, -4)
227: (4)     if x.ndim > 1:
228: (8)         stride_set[-2] = (1, 3, -4)
229: (4)     if x.ndim > 2:
230: (8)         stride_set[-3] = (1, -4)
231: (4)     for repeats in itertools.product(*tuple(stride_set)):
232: (8)         new_shape = [abs(a * b) for a, b in zip(x.shape, repeats)]
233: (8)         slices = tuple([slice(None, None, repeat) for repeat in repeats])
234: (8)         xi = np.empty(new_shape, dtype=x.dtype)
235: (8)         xi.view(np.uint32).fill(0xdeadbeef)
236: (8)         xi = xi[slices]
237: (8)         xi[...] = x
238: (8)         xi = xi.view(x.__class__)
239: (8)         assert_(np.all(xi == x))
240: (8)         yield xi, "stride_" + "_" .join(["%+d" % j for j in repeats])
241: (8)         if x.ndim >= 1 and x.shape[-1] == 1:
242: (12)             s = list(x.strides)
243: (12)             s[-1] = 0
244: (12)             xi = np.lib.stride_tricks.as_strided(x, strides=s)
245: (12)             yield xi, "stride_xxx_0"
246: (8)             if x.ndim >= 2 and x.shape[-2] == 1:
247: (12)                 s = list(x.strides)
248: (12)                 s[-2] = 0
249: (12)                 xi = np.lib.stride_tricks.as_strided(x, strides=s)
250: (12)                 yield xi, "stride_xxx_0_x"
251: (8)                 if x.ndim >= 2 and x.shape[:-2] == (1, 1):

```

```

252: (12)                     s = list(x.strides)
253: (12)                     s[-1] = 0
254: (12)                     s[-2] = 0
255: (12)                     xi = np.lib.stride_tricks.as_strided(x, strides=s)
256: (12)                     yield xi, "stride_xxx_0_0"
257: (0)  def _make_strided_cases():
258: (4)      new_cases = []
259: (4)      for case in CASES:
260: (8)          for a, a_label in _stride_comb_iter(case.a):
261: (12)              for b, b_label in _stride_comb_iter(case.b):
262: (16)                  new_case = LinalgCase(case.name + "_" + a_label + "_" +
b_label, a, b,
263: (38)                                         tags=case.tags | {'strided'})
264: (16)                  new_cases.append(new_case)
265: (4)  return new_cases
266: (0)  CASES += _make_strided_cases()
267: (0)  class LinalgTestCase:
268: (4)      TEST_CASES = CASES
269: (4)      def check_cases(self, require=set(), exclude=set()):
270: (8)          """
271: (8)              Run func on each of the cases with all of the tags in require, and
none
272: (8)              of the tags in exclude
273: (8)          """
274: (8)          for case in self.TEST_CASES:
275: (12)              if case.tags & require != require:
276: (16)                  continue
277: (12)              if case.tags & exclude:
278: (16)                  continue
279: (12)              try:
280: (16)                  case.check(self.do)
281: (12)              except Exception as e:
282: (16)                  msg = f'In test case: {case!r}\n\n'
283: (16)                  msg += traceback.format_exc()
284: (16)                  raise AssertionError(msg) from e
285: (0)  class LinalgSquareTestCase(LinalgTestCase):
286: (4)      def test_sq_cases(self):
287: (8)          self.check_cases(require={'square'},
288: (25)                           exclude={'generalized', 'size-0'})
289: (4)      def test_empty_sq_cases(self):
290: (8)          self.check_cases(require={'square', 'size-0'},
291: (25)                           exclude={'generalized'})
292: (0)  class LinalgNonsquareTestCase(LinalgTestCase):
293: (4)      def test_nonsq_cases(self):
294: (8)          self.check_cases(require={'nonsquare'},
295: (25)                           exclude={'generalized', 'size-0'})
296: (4)      def test_empty_nonsq_cases(self):
297: (8)          self.check_cases(require={'nonsquare', 'size-0'},
298: (25)                           exclude={'generalized'})
299: (0)  class HermitianTestCase(LinalgTestCase):
300: (4)      def test_herm_cases(self):
301: (8)          self.check_cases(require={'hermitian'},
302: (25)                           exclude={'generalized', 'size-0'})
303: (4)      def test_empty_herm_cases(self):
304: (8)          self.check_cases(require={'hermitian', 'size-0'},
305: (25)                           exclude={'generalized'})
306: (0)  class LinalgGeneralizedSquareTestCase(LinalgTestCase):
307: (4)      @pytest.mark.slow
308: (4)      def test_generalized_sq_cases(self):
309: (8)          self.check_cases(require={'generalized', 'square'},
310: (25)                           exclude={'size-0'})
311: (4)      @pytest.mark.slow
312: (4)      def test_generalized_empty_sq_cases(self):
313: (8)          self.check_cases(require={'generalized', 'square', 'size-0'})
314: (0)  class LinalgGeneralizedNonsquareTestCase(LinalgTestCase):
315: (4)      @pytest.mark.slow
316: (4)      def test_generalized_nonsq_cases(self):
317: (8)          self.check_cases(require={'generalized', 'nonsquare'},
318: (25)                           exclude={'size-0'})

```

```

319: (4)          @pytest.mark.slow
320: (4)          def test_generalized_empty_nonsq_cases(self):
321: (8)            self.check_cases(require={'generalized', 'nonsquare', 'size-0'})
322: (0)          class HermitianGeneralizedTestCase(LinalgTestCase):
323: (4)            @pytest.mark.slow
324: (4)            def test_generalized_herm_cases(self):
325: (8)              self.check_cases(require={'generalized', 'hermitian'},
326: (25)                  exclude={'size-0'})
327: (4)            @pytest.mark.slow
328: (4)            def test_generalized_empty_herm_cases(self):
329: (8)              self.check_cases(require={'generalized', 'hermitian', 'size-0'},
330: (25)                  exclude={'none'})
331: (0)          def dot_generalized(a, b):
332: (4)            a = asarray(a)
333: (4)            if a.ndim >= 3:
334: (8)              if a.ndim == b.ndim:
335: (12)                new_shape = a.shape[:-1] + b.shape[-1:]
336: (8)              elif a.ndim == b.ndim + 1:
337: (12)                new_shape = a.shape[:-1]
338: (8)              else:
339: (12)                raise ValueError("Not implemented...")
340: (8)            r = np.empty(new_shape, dtype=np.common_type(a, b))
341: (8)            for c in itertools.product(*map(range, a.shape[:-2])):
342: (12)              r[c] = dot(a[c], b[c])
343: (8)            return r
344: (4)          else:
345: (8)            return dot(a, b)
346: (0)          def identity_like_generalized(a):
347: (4)            a = asarray(a)
348: (4)            if a.ndim >= 3:
349: (8)              r = np.empty(a.shape, dtype=a.dtype)
350: (8)              r[...] = identity(a.shape[-2])
351: (8)              return r
352: (4)            else:
353: (8)              return identity(a.shape[0])
354: (0)          class SolveCases(LinalgSquareTestCase, LinalgGeneralizedSquareTestCase):
355: (4)            def do(self, a, b, tags):
356: (8)              x = linalg.solve(a, b)
357: (8)              assert_almost_equal(b, dot_generalized(a, x))
358: (8)              assert_(consistent_subclass(x, b))
359: (0)          class TestSolve(SolveCases):
360: (4)            @pytest.mark.parametrize('dtype', [single, double, csingle, cdouble])
361: (4)            def test_types(self, dtype):
362: (8)              x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
363: (8)              assert_equal(linalg.solve(x, x).dtype, dtype)
364: (4)            def test_0_size(self):
365: (8)              class ArraySubclass(np.ndarray):
366: (12)                pass
367: (8)              a = np.arange(8).reshape(2, 2, 2)
368: (8)              b = np.arange(6).reshape(1, 2, 3).view(ArraySubclass)
369: (8)              expected = linalg.solve(a, b)[:, 0:0, :]
370: (8)              result = linalg.solve(a[:, 0:0, 0:0], b[:, 0:0, :])
371: (8)              assert_array_equal(result, expected)
372: (8)              assert_(isinstance(result, ArraySubclass))
373: (8)              assert_raises(linalg.LinAlgError, linalg.solve, a[:, 0:0, 0:1], b)
374: (8)              assert_raises(ValueError, linalg.solve, a, b[:, 0:0, :])
375: (8)              b = np.arange(6).reshape(1, 3, 2) # broadcasting error
376: (8)              assert_raises(ValueError, linalg.solve, a, b)
377: (8)              assert_raises(ValueError, linalg.solve, a[0:0], b[0:0])
378: (8)              b = np.arange(2).reshape(1, 2).view(ArraySubclass)
379: (8)              expected = linalg.solve(a, b)[:, 0:0]
380: (8)              result = linalg.solve(a[:, 0:0, 0:0], b[:, 0:0])
381: (8)              assert_array_equal(result, expected)
382: (8)              assert_(isinstance(result, ArraySubclass))
383: (8)              b = np.arange(3).reshape(1, 3)
384: (8)              assert_raises(ValueError, linalg.solve, a, b)
385: (8)              assert_raises(ValueError, linalg.solve, a[0:0], b[0:0])
386: (8)              assert_raises(ValueError, linalg.solve, a[:, 0:0, 0:0], b)
387: (4)            def test_0_size_k(self):

```

```

388: (8)
389: (12)
390: (8)
391: (8)
392: (8)
393: (8)
394: (8)
395: (8)
396: (8)
397: (8)
398: (8)
399: (8)
400: (0)
401: (4)
402: (8)
403: (8)
404: (28)
405: (8)
406: (0)
407: (4)
408: (4)
409: (8)
410: (8)
411: (4)
412: (8)
413: (12)
414: (8)
415: (8)
416: (8)
417: (8)
418: (8)
419: (8)
420: (8)
421: (8)
422: (8)
423: (8)
424: (0)
425: (4)
426: (8)
427: (8)
428: (8)
429: (0)
430: (4)
431: (4)
432: (8)
433: (8)
434: (8)
435: (8)
436: (4)
437: (8)
438: (12)
439: (8)
440: (8)
441: (8)
442: (8)
443: (8)
444: (8)
445: (8)
446: (8)
447: (8)
448: (8)
449: (0)
450: (4)
451: (8)
452: (8)
453: (8)
454: (24)
[..., None, :, ],
455: (24)

```

```

            class ArraySubclass(np.ndarray):
                pass
                a = np.arange(4).reshape(1, 2, 2)
                b = np.arange(6).reshape(3, 2, 1).view(ArraySubclass)
                expected = linalg.solve(a, b)[:, :, 0:0]
                result = linalg.solve(a, b[:, :, 0:0])
                assert_array_equal(result, expected)
                assert_(isinstance(result, ArraySubclass))
                expected = linalg.solve(a, b)[:, 0:0, 0:0]
                result = linalg.solve(a[:, 0:0, 0:0], b[:, 0:0, 0:0])
                assert_array_equal(result, expected)
                assert_(isinstance(result, ArraySubclass))
        class InvCases(LinalgSquareTestCase, LinalgGeneralizedSquareTestCase):
            def do(self, a, b, tags):
                a_inv = linalg.inv(a)
                assert_almost_equal(dot_generalized(a, a_inv),
                                   identity_like_generalized(a))
                assert_(consistent_subclass(a_inv, a))
        class TestInv(InvCases):
            @pytest.mark.parametrize('dtype', [single, double, csingle, cdouble])
            def test_types(self, dtype):
                x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
                assert_equal(linalg.inv(x).dtype, dtype)
            def test_0_size(self):
                class ArraySubclass(np.ndarray):
                    pass
                    a = np.zeros((0, 1, 1), dtype=np.int_).view(ArraySubclass)
                    res = linalg.inv(a)
                    assert_(res.dtype.type is np.float64)
                    assert_equal(a.shape, res.shape)
                    assert_(isinstance(res, ArraySubclass))
                    a = np.zeros((0, 0), dtype=np.complex64).view(ArraySubclass)
                    res = linalg.inv(a)
                    assert_(res.dtype.type is np.complex64)
                    assert_equal(a.shape, res.shape)
                    assert_(isinstance(res, ArraySubclass))
        class EigvalsCases(LinalgSquareTestCase, LinalgGeneralizedSquareTestCase):
            def do(self, a, b, tags):
                ev = linalg.eigvals(a)
                evals, eigectors = linalg.eig(a)
                assert_almost_equal(ev, evals)
        class TestEigvals(EigvalsCases):
            @pytest.mark.parametrize('dtype', [single, double, csingle, cdouble])
            def test_types(self, dtype):
                x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
                assert_equal(linalg.eigvals(x).dtype, dtype)
                x = np.array([[1, 0.5], [-1, 1]], dtype=dtype)
                assert_equal(linalg.eigvals(x).dtype, get_complex_dtype(dtype))
            def test_0_size(self):
                class ArraySubclass(np.ndarray):
                    pass
                    a = np.zeros((0, 1, 1), dtype=np.int_).view(ArraySubclass)
                    res = linalg.eigvals(a)
                    assert_(res.dtype.type is np.float64)
                    assert_equal((0, 1), res.shape)
                    assert_(isinstance(res, np.ndarray))
                    a = np.zeros((0, 0), dtype=np.complex64).view(ArraySubclass)
                    res = linalg.eigvals(a)
                    assert_(res.dtype.type is np.complex64)
                    assert_equal((0,), res.shape)
                    assert_(isinstance(res, np.ndarray))
        class EigCases(LinalgSquareTestCase, LinalgGeneralizedSquareTestCase):
            def do(self, a, b, tags):
                res = linalg.eig(a)
                eigenvalues, eigenvectors = res.eigenvalues, res.eigenvectors
                assert_allclose(dot_generalized(a, eigenvectors),
                               np.asarray(eigenvectors) * np.asarray(eigenvalues),
                               rtol=get_rtol(eigenvalues.dtype))

```

```

456: (8)             assert_(consistent_subclass(eigenvectors, a))
457: (0)
458: (4)         class TestEig(EigCases):
459: (4)             @pytest.mark.parametrize('dtype', [single, double, csingle, cdouble])
460: (8)                 def test_types(self, dtype):
461: (8)                     x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
462: (8)                     w, v = np.linalg.eig(x)
463: (8)                     assert_equal(w.dtype, dtype)
464: (8)                     assert_equal(v.dtype, dtype)
465: (8)                     x = np.array([[1, 0.5], [-1, 1]], dtype=dtype)
466: (8)                     w, v = np.linalg.eig(x)
467: (8)                     assert_equal(w.dtype, get_complex_dtype(dtype))
468: (8)                     assert_equal(v.dtype, get_complex_dtype(dtype))
469: (8)             def test_0_size(self):
470: (12)                 class ArraySubclass(np.ndarray):
471: (8)                     pass
472: (8)                     a = np.zeros((0, 1, 1), dtype=np.int_).view(ArraySubclass)
473: (8)                     res, res_v = linalg.eig(a)
474: (8)                     assert_(res_v.dtype.type is np.float64)
475: (8)                     assert_(res.dtype.type is np.float64)
476: (8)                     assert_equal(a.shape, res_v.shape)
477: (8)                     assert_equal((0, 1), res.shape)
478: (8)                     assert_(isinstance(a, np.ndarray))
479: (8)                     a = np.zeros((0, 0), dtype=np.complex64).view(ArraySubclass)
480: (8)                     res, res_v = linalg.eig(a)
481: (8)                     assert_(res_v.dtype.type is np.complex64)
482: (8)                     assert_(res.dtype.type is np.complex64)
483: (8)                     assert_equal(a.shape, res_v.shape)
484: (8)                     assert_equal((0,), res.shape)
485: (0)                     assert_(isinstance(a, np.ndarray))
486: (4)         class SVDBaseTests:
487: (4)             hermitian = False
488: (4)             @pytest.mark.parametrize('dtype', [single, double, csingle, cdouble])
489: (8)                 def test_types(self, dtype):
490: (8)                     x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
491: (8)                     res = linalg.svd(x)
492: (8)                     U, S, Vh = res.U, res.S, res.Vh
493: (8)                     assert_equal(U.dtype, dtype)
494: (8)                     assert_equal(S.dtype, get_real_dtype(dtype))
495: (8)                     assert_equal(Vh.dtype, dtype)
496: (8)                     s = linalg.svd(x, compute_uv=False, hermitian=self.hermitian)
497: (0)                     assert_equal(s.dtype, get_real_dtype(dtype))
498: (4)         class SVDCases(LinalgSquareTestCase, LinalgGeneralizedSquareTestCase):
499: (8)             def do(self, a, b, tags):
500: (8)                 u, s, vt = linalg.svd(a, False)
501: (43)                 assert_allclose(a, dot_generalized(np.asarray(u) * np.asarray(s)[..., None, :], np.asarray(vt)),
502: (24)                                     rtol=get_rtol(u.dtype))
503: (8)                 assert_(consistent_subclass(u, a))
504: (8)                 assert_(consistent_subclass(vt, a))
505: (0)         class TestSVD(SVDCases, SVDBaseTests):
506: (4)             def test_empty_identity(self):
507: (8)                 """ Empty input should put an identity matrix in u or vh """
508: (8)                 x = np.empty((4, 0))
509: (8)                 u, s, vh = linalg.svd(x, compute_uv=True, hermitian=self.hermitian)
510: (8)                 assert_equal(u.shape, (4, 4))
511: (8)                 assert_equal(vh.shape, (0, 0))
512: (8)                 assert_equal(u, np.eye(4))
513: (8)                 x = np.empty((0, 4))
514: (8)                 u, s, vh = linalg.svd(x, compute_uv=True, hermitian=self.hermitian)
515: (8)                 assert_equal(u.shape, (0, 0))
516: (8)                 assert_equal(vh.shape, (4, 4))
517: (8)                 assert_equal(vh, np.eye(4))
518: (0)         class SVDHermitianCases(HermitianTestCase, HermitianGeneralizedTestCase):
519: (4)             def do(self, a, b, tags):
520: (8)                 u, s, vt = linalg.svd(a, False, hermitian=True)
521: (8)                 assert_allclose(a, dot_generalized(np.asarray(u) * np.asarray(s)[..., None, :], np.asarray(vt))),
```

```

523: (24)                                rtol=get_rtol(u.dtype))
524: (8)       def hermitian(mat):
525: (12)         axes = list(range(mat.ndim))
526: (12)         axes[-1], axes[-2] = axes[-2], axes[-1]
527: (12)         return np.conj(np.transpose(mat, axes=axes))
528: (8)           assert_almost_equal(np.matmul(u, hermitian(u)),
529: (8)             np.broadcast_to(np.eye(u.shape[-1]), u.shape))
529: (8)           assert_almost_equal(np.matmul(vt, hermitian(vt)),
530: (8)             np.broadcast_to(np.eye(vt.shape[-1]), vt.shape))
530: (8)             assert_equal(np.sort(s)[..., ::-1], s)
531: (8)             assert_(consistent_subclass(u, a))
532: (8)             assert_(consistent_subclass(vt, a))
533: (0)       class TestSVDHermitian(SVDDermitianCases, SVDBaseTests):
534: (4)         hermitian = True
535: (0)       class CondCases(LinalgSquareTestCase, LinalgGeneralizedSquareTestCase):
536: (4)         def do(self, a, b, tags):
537: (8)           c = asarray(a) # a might be a matrix
538: (8)           if 'size-0' in tags:
539: (12)             assert_raises(LinAlgError, linalg.cond, c)
540: (12)             return
541: (8)           s = linalg.svd(c, compute_uv=False)
542: (8)           assert_almost_equal(
543: (12)             linalg.cond(a), s[..., 0] / s[..., -1],
544: (12)               single_decimal=5, double_decimal=11)
545: (8)           assert_almost_equal(
546: (12)             linalg.cond(a, 2), s[..., 0] / s[..., -1],
547: (12)               single_decimal=5, double_decimal=11)
548: (8)           assert_almost_equal(
549: (12)             linalg.cond(a, -2), s[..., -1] / s[..., 0],
550: (12)               single_decimal=5, double_decimal=11)
551: (8)           cinv = np.linalg.inv(c)
552: (8)           assert_almost_equal(
553: (12)             linalg.cond(a, 1),
554: (12)               abs(c).sum(-2).max(-1) * abs(cinv).sum(-2).max(-1),
555: (12)               single_decimal=5, double_decimal=11)
556: (8)           assert_almost_equal(
557: (12)             linalg.cond(a, -1),
558: (12)               abs(c).sum(-2).min(-1) * abs(cinv).sum(-2).min(-1),
559: (12)               single_decimal=5, double_decimal=11)
560: (8)           assert_almost_equal(
561: (12)             linalg.cond(a, np.inf),
562: (12)               abs(c).sum(-1).max(-1) * abs(cinv).sum(-1).max(-1),
563: (12)               single_decimal=5, double_decimal=11)
564: (8)           assert_almost_equal(
565: (12)             linalg.cond(a, -np.inf),
566: (12)               abs(c).sum(-1).min(-1) * abs(cinv).sum(-1).min(-1),
567: (12)               single_decimal=5, double_decimal=11)
568: (8)           assert_almost_equal(
569: (12)             linalg.cond(a, 'fro'),
570: (12)               np.sqrt((abs(c)**2).sum(-1).sum(-1)
571: (20)                 * (abs(cinv)**2).sum(-1).sum(-1)),
572: (12)               single_decimal=5, double_decimal=11)
573: (0)       class TestCond(CondCases):
574: (4)         def test_basic_nonsvd(self):
575: (8)           A = array([[1., 0, 1], [0, -2., 0], [0, 0, 3.]])
576: (8)           assert_almost_equal(linalg.cond(A, inf), 4)
577: (8)           assert_almost_equal(linalg.cond(A, -inf), 2/3)
578: (8)           assert_almost_equal(linalg.cond(A, 1), 4)
579: (8)           assert_almost_equal(linalg.cond(A, -1), 0.5)
580: (8)           assert_almost_equal(linalg.cond(A, 'fro'), np.sqrt(265 / 12))
581: (4)         def test_singular(self):
582: (8)           As = [np.zeros((2, 2)), np.ones((2, 2))]
583: (8)           p_pos = [None, 1, 2, 'fro']
584: (8)           p_neg = [-1, -2]
585: (8)           for A, p in itertools.product(As, p_pos):
586: (12)             assert_(linalg.cond(A, p) > 1e15)
587: (8)             for A, p in itertools.product(As, p_neg):
588: (12)               linalg.cond(A, p)
589: (4)             @pytest.mark.xfail(True, run=False,

```

```

590: (23)                                     reason="Platform/LAPACK-dependent failure, "
591: (30)                                     "see gh-18914")
592: (4)
593: (8)
594: (8)
595: (8)
596: (8)
597: (8)
598: (12)
599: (12)
600: (12)
601: (8)
602: (8)
603: (8)
604: (12)
605: (12)
606: (12)
607: (16)
608: (16)
609: (12)
610: (16)
611: (16)
612: (4)
613: (8)
614: (8)
615: (8)
616: (8)
617: (8)
618: (12)
619: (12)
620: (12)
621: (12)
622: (12)
623: (0)
624: (16)
625: (16)
626: (16)
627: (4)
628: (8)
629: (8)
630: (8)
double_decimal=11)
631: (8)
632: (0)
633: (4)
634: (0)
635: (4)
636: (8)
637: (8)
638: (8)
double_decimal=11)
639: (8)
640: (0)
641: (4)
642: (0)
643: (4)
644: (8)
645: (8)
646: (8)
647: (8)
648: (12)
649: (8)
650: (12)
651: (8)
652: (8)
653: (8)
654: (8)
655: (8)
656: (8)

def test_nan(self):
    ps = [None, 1, -1, 2, -2, 'fro']
    p_pos = [None, 1, 2, 'fro']
    A = np.ones((2, 2))
    A[0,1] = np.nan
    for p in ps:
        c = linalg.cond(A, p)
        assert_(isinstance(c, np.float_))
        assert_(np.isnan(c))
    A = np.ones((3, 2, 2))
    A[1,0,1] = np.nan
    for p in ps:
        c = linalg.cond(A, p)
        assert_(np.isnan(c[1]))
        if p in p_pos:
            assert_(c[0] > 1e15)
            assert_(c[2] > 1e15)
        else:
            assert_(not np.isnan(c[0]))
            assert_(not np.isnan(c[2]))


def test_stacked_singular(self):
    np.random.seed(1234)
    A = np.random.rand(2, 2, 2)
    A[0,0] = 0
    A[1,1] = 0
    for p in (None, 1, 2, 'fro', -1, -2):
        c = linalg.cond(A, p)
        assert_equal(c[0,0], np.inf)
        assert_equal(c[1,1], np.inf)
        assert_(np.isfinite(c[0,1]))
        assert_(np.isfinite(c[1,0]))


class PinvCases(LinalgSquareTestCase,
                 LinalgNonsquareTestCase,
                 LinalgGeneralizedSquareTestCase,
                 LinalgGeneralizedNonsquareTestCase):
    def do(self, a, b, tags):
        a_ginv = linalg.pinv(a)
        dot = dot_generalized
        assert_almost_equal(dot(dot(a, a_ginv), a), a, single_decimal=5,
                           assert_(consistent_subclass(a_ginv, a))

class TestPinv(PinvCases):
    pass

class PinvHermitianCases(HermitianTestCase, HermitianGeneralizedTestCase):
    def do(self, a, b, tags):
        a_ginv = linalg.pinv(a, hermitian=True)
        dot = dot_generalized
        assert_almost_equal(dot(dot(a, a_ginv), a), a, single_decimal=5,
                           assert_(consistent_subclass(a_ginv, a))

class TestPinvHermitian(PinvHermitianCases):
    pass

class DetCases(LinalgSquareTestCase, LinalgGeneralizedSquareTestCase):
    def do(self, a, b, tags):
        d = linalg.det(a)
        res = linalg.slogdet(a)
        s, ld = res.sign, res.logabsdet
        if asarray(a).dtype.type in (single, double):
            ad = asarray(a).astype(double)
        else:
            ad = asarray(a).astype(cdouble)
        ev = linalg.eigvals(ad)
        assert_almost_equal(d, multiply.reduce(ev, axis=-1))
        assert_almost_equal(s * np.exp(ld), multiply.reduce(ev, axis=-1))
        s = np.atleast_1d(s)
        ld = np.atleast_1d(ld)
        m = (s != 0)

```

```

657: (8)             assert_almost_equal(np.abs(s[m]), 1)
658: (8)             assert_equal(ld[~m], -inf)
659: (0) class TestDet(DetCases):
660: (4)             def test_zero(self):
661: (8)                 assert_equal(linalg.det([[0.0]]), 0.0)
662: (8)                 assert_equal(type(linalg.det([[0.0]])), double)
663: (8)                 assert_equal(linalg.det([[0.0j]]), 0.0)
664: (8)                 assert_equal(type(linalg.det([[0.0j]])), cdouble)
665: (8)                 assert_equal(linalg.slogdet([[0.0]]), (0.0, -inf))
666: (8)                 assert_equal(type(linalg.slogdet([[0.0]])[0]), double)
667: (8)                 assert_equal(type(linalg.slogdet([[0.0]])[1]), double)
668: (8)                 assert_equal(linalg.slogdet([[0.0j]]), (0.0j, -inf))
669: (8)                 assert_equal(type(linalg.slogdet([[0.0j]])[0]), cdouble)
670: (8)                 assert_equal(type(linalg.slogdet([[0.0j]])[1]), double)
671: (4) @pytest.mark.parametrize('dtype', [single, double, csingle, cdouble])
672: (4) def test_types(self, dtype):
673: (8)     x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
674: (8)     assert_equal(np.linalg.det(x).dtype, dtype)
675: (8)     ph, s = np.linalg.slogdet(x)
676: (8)     assert_equal(s.dtype, get_real_dtype(dtype))
677: (8)     assert_equal(ph.dtype, dtype)
678: (4) def test_0_size(self):
679: (8)     a = np.zeros((0, 0), dtype=np.complex64)
680: (8)     res = linalg.det(a)
681: (8)     assert_equal(res, 1.)
682: (8)     assert_(res.dtype.type is np.complex64)
683: (8)     res = linalg.slogdet(a)
684: (8)     assert_equal(res, (1, 0))
685: (8)     assert_(res[0].dtype.type is np.complex64)
686: (8)     assert_(res[1].dtype.type is np.float32)
687: (8)     a = np.zeros((0, 0), dtype=np.float64)
688: (8)     res = linalg.det(a)
689: (8)     assert_equal(res, 1.)
690: (8)     assert_(res.dtype.type is np.float64)
691: (8)     res = linalg.slogdet(a)
692: (8)     assert_equal(res, (1, 0))
693: (8)     assert_(res[0].dtype.type is np.float64)
694: (8)     assert_(res[1].dtype.type is np.float64)
695: (0) class LstsqCases(LinalgSquareTestCase, LinalgNonsquareTestCase):
696: (4)     def do(self, a, b, tags):
697: (8)         arr = np.asarray(a)
698: (8)         m, n = arr.shape
699: (8)         u, s, vt = linalg.svd(a, False)
700: (8)         x, residuals, rank, sv = linalg.lstsq(a, b, rcond=-1)
701: (8)         if m == 0:
702: (12)             assert_((x == 0).all())
703: (8)         if m <= n:
704: (12)             assert_almost_equal(b, dot(a, x))
705: (12)             assert_equal(rank, m)
706: (8)         else:
707: (12)             assert_equal(rank, n)
708: (8)         assert_almost_equal(sv, sv.__array_wrap__(s))
709: (8)         if rank == n and m > n:
710: (12)             expect_resids = (
711: (16)                 np.asarray(abs(np.dot(a, x) - b)) ** 2).sum(axis=0)
712: (12)             expect_resids = np.asarray(expect_resids)
713: (12)             if np.asarray(b).ndim == 1:
714: (16)                 expect_resids.shape = (1,)
715: (16)                 assert_equal(residuals.shape, expect_resids.shape)
716: (8)             else:
717: (12)                 expect_resids = np.array([]).view(type(x))
718: (8)             assert_almost_equal(residuals, expect_resids)
719: (8)             assert_(np.issubdtype(residuals.dtype, np.floating))
720: (8)             assert_(consistent_subclass(x, b))
721: (8)             assert_(consistent_subclass(residuals, b))
722: (0) class TestLstsq(LstsqCases):
723: (4)             def test_future_rcond(self):
724: (8)                 a = np.array([[0., 1., 0., 1., 2., 0.],
725: (22)                               [0., 2., 0., 0., 1., 0.],
```

```

726: (22) [1., 0., 1., 0., 0., 4.],
727: (22) [0., 0., 0., 2., 3., 0.]]).T
728: (8) b = np.array([1, 0, 0, 0, 0, 0])
729: (8) with suppress_warnings() as sup:
730: (12)     w = sup.record(FutureWarning, "`rcond` parameter will change")
731: (12)     x, residuals, rank, s = linalg.lstsq(a, b)
732: (12)     assert_(rank == 4)
733: (12)     x, residuals, rank, s = linalg.lstsq(a, b, rcond=-1)
734: (12)     assert_(rank == 4)
735: (12)     x, residuals, rank, s = linalg.lstsq(a, b, rcond=None)
736: (12)     assert_(rank == 3)
737: (12)     assert_(len(w) == 1)
738: (4) @pytest.mark.parametrize(["m", "n", "n_rhs"], [
739: (8)     (4, 2, 2),
740: (8)     (0, 4, 1),
741: (8)     (0, 4, 2),
742: (8)     (4, 0, 1),
743: (8)     (4, 0, 2),
744: (8)     (4, 2, 0),
745: (8)     (0, 0, 0)
746: (4) ])
747: (4) def test_empty_a_b(self, m, n, n_rhs):
748: (8)     a = np.arange(m * n).reshape(m, n)
749: (8)     b = np.ones((m, n_rhs))
750: (8)     x, residuals, rank, s = linalg.lstsq(a, b, rcond=None)
751: (8)     if m == 0:
752: (12)         assert_((x == 0).all())
753: (8)         assert_equal(x.shape, (n, n_rhs))
754: (8)         assert_equal(residuals.shape, ((n_rhs,) if m > n else (0,)))
755: (8)     if m > n and n_rhs > 0:
756: (12)         r = b - np.dot(a, x)
757: (12)         assert_almost_equal(residuals, (r * r).sum(axis=-2))
758: (8)         assert_equal(rank, min(m, n))
759: (8)         assert_equal(s.shape, (min(m, n),))
760: (4) def test_incompatible_dims(self):
761: (8)     x = np.array([0, 1, 2, 3])
762: (8)     y = np.array([-1, 0.2, 0.9, 2.1, 3.3])
763: (8)     A = np.vstack([x, np.ones(len(x))]).T
764: (8)     with assert_raises_regex(LinAlgError, "Incompatible dimensions"):
765: (12)         linalg.lstsq(A, y, rcond=None)
766: (0) @pytest.mark.parametrize('dt', [np.dtype(c) for c in '?bBhHiIqQefdgFDGO'])
767: (0) class TestMatrixPower:
768: (4)     rshft_0 = np.eye(4)
769: (4)     rshft_1 = rshft_0[[3, 0, 1, 2]]
770: (4)     rshft_2 = rshft_0[[2, 3, 0, 1]]
771: (4)     rshft_3 = rshft_0[[1, 2, 3, 0]]
772: (4)     rshft_all = [rshft_0, rshft_1, rshft_2, rshft_3]
773: (4)     noninv = array([[1, 0], [0, 0]])
774: (4)     stacked = np.block([[rshft_0]]**2)
775: (4)     dtinv = [object, np.dtype('e'), np.dtype('g'), np.dtype('G')]
776: (4) def test_large_power(self, dt):
777: (8)     rshft = self.rshft_1.astype(dt)
778: (8)     assert_equal(
779: (12)         matrix_power(rshft, 2**100 + 2**10 + 2**5 + 0), self.rshft_0)
780: (8)     assert_equal(
781: (12)         matrix_power(rshft, 2**100 + 2**10 + 2**5 + 1), self.rshft_1)
782: (8)     assert_equal(
783: (12)         matrix_power(rshft, 2**100 + 2**10 + 2**5 + 2), self.rshft_2)
784: (8)     assert_equal(
785: (12)         matrix_power(rshft, 2**100 + 2**10 + 2**5 + 3), self.rshft_3)
786: (4) def test_power_is_zero(self, dt):
787: (8)     def tz(M):
788: (12)         mz = matrix_power(M, 0)
789: (12)         assert_equal(mz, identity_like_generalized(M))
790: (12)         assert_equal(mz.dtype, M.dtype)
791: (8)         for mat in self.rshft_all:
792: (12)             tz(mat.astype(dt))
793: (12)             if dt != object:
794: (16)                 tz(self.stacked.astype(dt))

```

```

795: (4)
796: (8)
797: (12)
798: (12)
799: (12)
800: (8)
801: (12)
802: (12)
803: (16)
804: (4)
805: (8)
806: (12)
807: (12)
808: (12)
809: (12)
810: (8)
811: (12)
812: (12)
813: (16)
814: (4)
815: (8)
816: (12)
817: (12)
818: (12)
819: (16)
820: (8)
821: (12)
822: (16)
823: (4)
824: (8)
825: (8)
826: (8)
827: (4)
828: (8)
829: (8)
830: (8)
831: (4)
832: (4)
833: (8)
834: (12)
835: (8)
836: (8)
837: (0)
838: (4)
839: (8)
840: (8)
841: (8)
842: (8)
843: (8)
844: (8)
845: (0)
846: (4)
847: (4)
848: (8)
849: (8)
850: (8)
851: (4)
852: (8)
853: (8)
854: (8)
855: (8)
856: (4)
857: (8)
858: (8)
859: (8)
860: (8)
861: (8)
862: (8)
863: (8)

    def test_power_is_one(self, dt):
        def tz(mat):
            mz = matrix_power(mat, 1)
            assert_equal(mz, mat)
            assert_equal(mz.dtype, mat.dtype)
        for mat in self.rshft_all:
            tz(mat.astype(dt))
            if dt != object:
                tz(self.stacked.astype(dt))

    def test_power_is_two(self, dt):
        def tz(mat):
            mz = matrix_power(mat, 2)
            mmul = matmul if mat.dtype != object else dot
            assert_equal(mz, mmul(mat, mat))
            assert_equal(mz.dtype, mat.dtype)
        for mat in self.rshft_all:
            tz(mat.astype(dt))
            if dt != object:
                tz(self.stacked.astype(dt))

    def test_power_is_minus_one(self, dt):
        def tz(mat):
            invmat = matrix_power(mat, -1)
            mmul = matmul if mat.dtype != object else dot
            assert_almost_equal(
                mmul(invmat, mat), identity_like_generalized(mat))
        for mat in self.rshft_all:
            if dt not in self.dtnoinv:
                tz(mat.astype(dt))

    def test_exceptions_bad_power(self, dt):
        mat = self.rshft_0.astype(dt)
        assert_raises(TypeError, matrix_power, mat, 1.5)
        assert_raises(TypeError, matrix_power, mat, [1])

    def test_exceptions_non_square(self, dt):
        assert_raises(LinAlgError, matrix_power, np.array([1], dt), 1)
        assert_raises(LinAlgError, matrix_power, np.array([[1], [2]], dt), 1)
        assert_raises(LinAlgError, matrix_power, np.ones((4, 3, 2), dt), 1)
@pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
    def test_exceptions_not_invertible(self, dt):
        if dt in self.dtnoinv:
            return
        mat = self.noninv.astype(dt)
        assert_raises(LinAlgError, matrix_power, mat, -1)

class TestEigvalshCases(HermitianTestCase, HermitianGeneralizedTestCase):
    def do(self, a, b, tags):
        ev = linalg.eigvalsh(a, 'L')
        evals, evecs = linalg.eig(a)
        evals.sort(axis=-1)
        assert_allclose(ev, evals, rtol=get_rtol(ev.dtype))
        ev2 = linalg.eigvalsh(a, 'U')
        assert_allclose(ev2, evals, rtol=get_rtol(ev.dtype))

    class TestEigvalsh:
        @pytest.mark.parametrize('dtype', [single, double, csingle, cdouble])
        def test_types(self, dtype):
            x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
            w = np.linalg.eigvalsh(x)
            assert_equal(w.dtype, get_real_dtype(dtype))

        def test_invalid(self):
            x = np.array([[1, 0.5], [0.5, 1]], dtype=np.float32)
            assert_raises(ValueError, np.linalg.eigvalsh, x, UPLO="lrong")
            assert_raises(ValueError, np.linalg.eigvalsh, x, "lower")
            assert_raises(ValueError, np.linalg.eigvalsh, x, "upper")

        def test_UPLO(self):
            Klo = np.array([[0, 0], [1, 0]], dtype=np.double)
            Kup = np.array([[0, 1], [0, 0]], dtype=np.double)
            tgt = np.array([-1, 1], dtype=np.double)
            rtol = get_rtol(np.double)
            w = np.linalg.eigvalsh(Klo)
            assert_allclose(w, tgt, rtol=rtol)
            w = np.linalg.eigvalsh(Klo, UPLO='L')

```

```

864: (8)             assert_allclose(w, tgt, rtol=rtol)
865: (8)             w = np.linalg.eigvalsh(Klo, UPLO='L')
866: (8)             assert_allclose(w, tgt, rtol=rtol)
867: (8)             w = np.linalg.eigvalsh(Kup, UPLO='U')
868: (8)             assert_allclose(w, tgt, rtol=rtol)
869: (8)             w = np.linalg.eigvalsh(Kup, UPLO='u')
870: (8)             assert_allclose(w, tgt, rtol=rtol)
871: (4)             def test_0_size(self):
872: (8)                 class ArraySubclass(np.ndarray):
873: (12)                     pass
874: (8)                     a = np.zeros((0, 1, 1), dtype=np.int_).view(ArraySubclass)
875: (8)                     res = linalg.eigvalsh(a)
876: (8)                     assert_(res.dtype.type is np.float64)
877: (8)                     assert_equal((0, 1), res.shape)
878: (8)                     assert_(isinstance(res, np.ndarray))
879: (8)                     a = np.zeros((0, 0), dtype=np.complex64).view(ArraySubclass)
880: (8)                     res = linalg.eigvalsh(a)
881: (8)                     assert_(res.dtype.type is np.float32)
882: (8)                     assert_equal((0,), res.shape)
883: (8)                     assert_(isinstance(res, np.ndarray))
884: (0)             class TestEighCases(HermitianTestCase, HermitianGeneralizedTestCase):
885: (4)                 def do(self, a, b, tags):
886: (8)                     res = linalg.eigh(a)
887: (8)                     ev, evc = res.eigenvalues, res.eigenvectors
888: (8)                     evals, eigctors = linalg.eig(a)
889: (8)                     evals.sort(axis=-1)
890: (8)                     assert_almost_equal(ev, evals)
891: (8)                     assert_allclose(dot_generalized(a, evc),
892: (24)                         np.asarray(ev)[..., None, :] * np.asarray(evc),
893: (24)                         rtol=get_rtol(ev.dtype))
894: (8)                     ev2, evc2 = linalg.eigh(a, 'U')
895: (8)                     assert_almost_equal(ev2, evals)
896: (8)                     assert_allclose(dot_generalized(a, evc2),
897: (24)                         np.asarray(ev2)[..., None, :] * np.asarray(evc2),
898: (24)                         rtol=get_rtol(ev.dtype), err_msg=repr(a))
899: (0)             class TestEigh:
900: (4)                 @pytest.mark.parametrize('dtype', [single, double, csingle, cdouble])
901: (4)                 def test_types(self, dtype):
902: (8)                     x = np.array([[1, 0.5], [0.5, 1]], dtype=dtype)
903: (8)                     w, v = np.linalg.eigh(x)
904: (8)                     assert_equal(w.dtype, get_real_dtype(dtype))
905: (8)                     assert_equal(v.dtype, dtype)
906: (4)                     def test_invalid(self):
907: (8)                         x = np.array([[1, 0.5], [0.5, 1]], dtype=np.float32)
908: (8)                         assert_raises(ValueError, np.linalg.eigh, x, UPLO="lron")
909: (8)                         assert_raises(ValueError, np.linalg.eigh, x, "lower")
910: (8)                         assert_raises(ValueError, np.linalg.eigh, x, "upper")
911: (4)                     def test_UPLO(self):
912: (8)                         Klo = np.array([[0, 0], [1, 0]], dtype=np.double)
913: (8)                         Kup = np.array([[0, 1], [0, 0]], dtype=np.double)
914: (8)                         tgt = np.array([-1, 1], dtype=np.double)
915: (8)                         rtol = get_rtol(np.double)
916: (8)                         w, v = np.linalg.eigh(Klo)
917: (8)                         assert_allclose(w, tgt, rtol=rtol)
918: (8)                         w, v = np.linalg.eigh(Klo, UPLO='L')
919: (8)                         assert_allclose(w, tgt, rtol=rtol)
920: (8)                         w, v = np.linalg.eigh(Klo, UPLO='l')
921: (8)                         assert_allclose(w, tgt, rtol=rtol)
922: (8)                         w, v = np.linalg.eigh(Kup, UPLO='U')
923: (8)                         assert_allclose(w, tgt, rtol=rtol)
924: (8)                         w, v = np.linalg.eigh(Kup, UPLO='u')
925: (8)                         assert_allclose(w, tgt, rtol=rtol)
926: (4)                     def test_0_size(self):
927: (8)                         class ArraySubclass(np.ndarray):
928: (12)                             pass
929: (8)                             a = np.zeros((0, 1, 1), dtype=np.int_).view(ArraySubclass)
930: (8)                             res, res_v = linalg.eigh(a)
931: (8)                             assert_(res_v.dtype.type is np.float64)
932: (8)                             assert_(res.dtype.type is np.float64)

```

```

933: (8)             assert_equal(a.shape, res_v.shape)
934: (8)             assert_equal((0, 1), res.shape)
935: (8)             assert_(isinstance(a, np.ndarray))
936: (8)             a = np.zeros((0, 0), dtype=np.complex64).view(ArraySubclass)
937: (8)             res, res_v = linalg.eigh(a)
938: (8)             assert_(res_v.dtype.type is np.complex64)
939: (8)             assert_(res.dtype.type is np.float32)
940: (8)             assert_equal(a.shape, res_v.shape)
941: (8)             assert_equal((0,), res.shape)
942: (8)             assert_(isinstance(a, np.ndarray))
943: (0)             class _TestNormBase:
944: (4)                 dt = None
945: (4)                 dec = None
946: (4)                 @staticmethod
947: (4)                 def check_dtype(x, res):
948: (8)                     if issubclass(x.dtype.type, np.inexact):
949: (12)                         assert_equal(res.dtype, x.real.dtype)
950: (8)
951: (12)                         assert_(issubclass(res.dtype.type, np.floating))
952: (0)             class _TestNormGeneral(_TestNormBase):
953: (4)                 def test_empty(self):
954: (8)                     assert_equal(norm([]), 0.0)
955: (8)                     assert_equal(norm(array([], dtype=self.dt)), 0.0)
956: (8)                     assert_equal(norm(atleast_2d(array([], dtype=self.dt))), 0.0)
957: (4)                 def test_vector_return_type(self):
958: (8)                     a = np.array([1, 0, 1])
959: (8)                     exact_types = np.typecodes['AllInteger']
960: (8)                     inexact_types = np.typecodes['AllFloat']
961: (8)                     all_types = exact_types + inexact_types
962: (8)                     for each_type in all_types:
963: (12)                         at = a.astype(each_type)
964: (12)                         an = norm(at, -np.inf)
965: (12)                         self.check_dtype(at, an)
966: (12)                         assert_almost_equal(an, 0.0)
967: (12)                         with suppress_warnings() as sup:
968: (16)                             sup.filter(RuntimeWarning, "divide by zero encountered")
969: (16)                             an = norm(at, -1)
970: (16)                             self.check_dtype(at, an)
971: (16)                             assert_almost_equal(an, 0.0)
972: (12)                         an = norm(at, 0)
973: (12)                         self.check_dtype(at, an)
974: (12)                         assert_almost_equal(an, 2)
975: (12)                         an = norm(at, 1)
976: (12)                         self.check_dtype(at, an)
977: (12)                         assert_almost_equal(an, 2.0)
978: (12)                         an = norm(at, 2)
979: (12)                         self.check_dtype(at, an)
980: (12)                         assert_almost_equal(an,
an.dtype.type(2.0)**an.dtype.type(1.0/2.0)))
981: (12)                         an = norm(at, 4)
982: (12)                         self.check_dtype(at, an)
983: (12)                         assert_almost_equal(an,
an.dtype.type(2.0)**an.dtype.type(1.0/4.0)))
984: (12)                         an = norm(at, np.inf)
985: (12)                         self.check_dtype(at, an)
986: (12)                         assert_almost_equal(an, 1.0)
987: (4)                 def test_vector(self):
988: (8)                     a = [1, 2, 3, 4]
989: (8)                     b = [-1, -2, -3, -4]
990: (8)                     c = [-1, 2, -3, 4]
991: (8)                     def _test(v):
992: (12)                         np.testing.assert_almost_equal(norm(v), 30 ** 0.5,
993: (43)                                         decimal=self.dec)
994: (12)                         np.testing.assert_almost_equal(norm(v, inf), 4.0,
995: (43)                                         decimal=self.dec)
996: (12)                         np.testing.assert_almost_equal(norm(v, -inf), 1.0,
997: (43)                                         decimal=self.dec)
998: (12)                         np.testing.assert_almost_equal(norm(v, 1), 10.0,
999: (43)                                         decimal=self.dec)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1000: (12) np.testing.assert_almost_equal(norm(v, -1), 12.0 / 25,
1001: (43)                                     decimal=self.dec)
1002: (12) np.testing.assert_almost_equal(norm(v, 2), 30 ** 0.5,
1003: (43)                                     decimal=self.dec)
1004: (12) np.testing.assert_almost_equal(norm(v, -2), ((205. / 144) ** -0.5),
1005: (43)                                     decimal=self.dec)
1006: (12) np.testing.assert_almost_equal(norm(v, 0), 4,
1007: (43)                                     decimal=self.dec)
1008: (8)  for v in (a, b, c,):
1009: (12)      _test(v)
1010: (8)  for v in (array(a, dtype=self.dt), array(b, dtype=self.dt),
1011: (18)          array(c, dtype=self.dt)):
1012: (12)      _test(v)
1013: (4)  def test_axis(self):
1014: (8)    A = array([[1, 2, 3], [4, 5, 6]], dtype=self.dt)
1015: (8)    for order in [None, -1, 0, 1, 2, 3, np.Inf, -np.Inf]:
1016: (12)      expected0 = [norm(A[:, k], ord=order) for k in range(A.shape[1])]
1017: (12)      assert_almost_equal(norm(A, ord=order, axis=0), expected0)
1018: (12)      expected1 = [norm(A[k, :], ord=order) for k in range(A.shape[0])]
1019: (12)      assert_almost_equal(norm(A, ord=order, axis=1), expected1)
1020: (8)    B = np.arange(1, 25, dtype=self.dt).reshape(2, 3, 4)
1021: (8)    nd = B.ndim
1022: (8)    for order in [None, -2, 2, -1, 1, np.Inf, -np.Inf, 'fro']:
1023: (12)      for axis in itertools.combinations(range(-nd, nd), 2):
1024: (16)        row_axis, col_axis = axis
1025: (16)        if row_axis < 0:
1026: (20)          row_axis += nd
1027: (16)        if col_axis < 0:
1028: (20)          col_axis += nd
1029: (16)        if row_axis == col_axis:
1030: (20)          assert_raises(ValueError, norm, B, ord=order, axis=axis)
1031: (16)        else:
1032: (20)          n = norm(B, ord=order, axis=axis)
1033: (20)          k_index = nd - (row_axis + col_axis)
1034: (20)          if row_axis < col_axis:
1035: (24)            expected = [norm(B[:,].take(k, axis=k_index),
1036: (36)                                         for k in range(B.shape[k_index]))]
1037: (20)          else:
1038: (24)            expected = [norm(B[:,].take(k, axis=k_index).T,
1039: (36)                                         for k in range(B.shape[k_index]))]
1040: (20)          assert_almost_equal(n, expected)
1041: (4)  def test_kepdims(self):
1042: (8)    A = np.arange(1, 25, dtype=self.dt).reshape(2, 3, 4)
1043: (8)    allclose_err = 'order {0}, axis = {1}'
1044: (8)    shape_err = 'Shape mismatch found {0}, expected {1}, order={2}, axis={3}'
1045: (8)    expected = norm(A, ord=None, axis=None)
1046: (8)    found = norm(A, ord=None, axis=None, keepdims=True)
1047: (8)    assert_allclose(np.squeeze(found), expected,
1048: (24)          err_msg=allclose_err.format(None, None))
1049: (8)    expected_shape = (1, 1, 1)
1050: (8)    assert_(found.shape == expected_shape,
1051: (16)          shape_err.format(found.shape, expected_shape, None, None))
1052: (8)    for order in [None, -1, 0, 1, 2, 3, np.Inf, -np.Inf]:
1053: (12)      for k in range(A.ndim):
1054: (16)        expected = norm(A, ord=order, axis=k)
1055: (16)        found = norm(A, ord=order, axis=k, keepdims=True)
1056: (16)        assert_allclose(np.squeeze(found), expected,
1057: (32)          err_msg=allclose_err.format(order, k))
1058: (16)        expected_shape = list(A.shape)
1059: (16)        expected_shape[k] = 1
1060: (16)        expected_shape = tuple(expected_shape)
1061: (16)        assert_(found.shape == expected_shape,
1062: (24)          shape_err.format(found.shape, expected_shape, order,
1063: (8)            k))
1063: (8)          for order in [None, -2, 2, -1, 1, np.Inf, -np.Inf, 'fro', 'nuc']:

```

```

1064: (12)             for k in itertools.permutations(range(A.ndim), 2):
1065: (16)                 expected = norm(A, ord=order, axis=k)
1066: (16)                 found = norm(A, ord=order, axis=k, keepdims=True)
1067: (16)                 assert_allclose(np.squeeze(found), expected,
1068: (32)                               err_msg=allclose_err.format(order, k))
1069: (16)                 expected_shape = list(A.shape)
1070: (16)                 expected_shape[k[0]] = 1
1071: (16)                 expected_shape[k[1]] = 1
1072: (16)                 expected_shape = tuple(expected_shape)
1073: (16)                 assert_(found.shape == expected_shape,
1074: (24)                               shape_err.format(found.shape, expected_shape, order,
k))
1075: (0)                  class _TestNorm2D(_TestNormBase):
1076: (4)                      array = np.array
1077: (4)                      def test_matrix_empty(self):
1078: (8)                          assert_equal(norm(self.array([[]], dtype=self.dt)), 0.0)
1079: (4)                      def test_matrix_return_type(self):
1080: (8)                          a = self.array([[1, 0, 1], [0, 1, 1]])
1081: (8)                          exact_types = np.typecodes['AllInteger']
1082: (8)                          inexact_types = 'fdFD'
1083: (8)                          all_types = exact_types + inexact_types
1084: (8)                          for each_type in all_types:
1085: (12)                              at = a.astype(each_type)
1086: (12)                              an = norm(at, -np.inf)
1087: (12)                              self.check_dtype(at, an)
1088: (12)                              assert_almost_equal(an, 2.0)
1089: (12)                              with suppress_warnings() as sup:
1090: (16)                                  sup.filter(RuntimeWarning, "divide by zero encountered")
1091: (16)                                  an = norm(at, -1)
1092: (16)                                  self.check_dtype(at, an)
1093: (16)                                  assert_almost_equal(an, 1.0)
1094: (12)                                  an = norm(at, 1)
1095: (12)                                  self.check_dtype(at, an)
1096: (12)                                  assert_almost_equal(an, 2.0)
1097: (12)                                  an = norm(at, 2)
1098: (12)                                  self.check_dtype(at, an)
1099: (12)                                  assert_almost_equal(an, 3.0**(1.0/2.0))
1100: (12)                                  an = norm(at, -2)
1101: (12)                                  self.check_dtype(at, an)
1102: (12)                                  assert_almost_equal(an, 1.0)
1103: (12)                                  an = norm(at, np.inf)
1104: (12)                                  self.check_dtype(at, an)
1105: (12)                                  assert_almost_equal(an, 2.0)
1106: (12)                                  an = norm(at, 'fro')
1107: (12)                                  self.check_dtype(at, an)
1108: (12)                                  assert_almost_equal(an, 2.0)
1109: (12)                                  an = norm(at, 'nuc')
1110: (12)                                  self.check_dtype(at, an)
1111: (12)                                  np.testing.assert_almost_equal(an, 2.7320508075688772, decimal=6)
1112: (4)                      def test_matrix_2x2(self):
1113: (8)                          A = self.array([[1, 3], [5, 7]], dtype=self.dt)
1114: (8)                          assert_almost_equal(norm(A), 84 ** 0.5)
1115: (8)                          assert_almost_equal(norm(A, 'fro'), 84 ** 0.5)
1116: (8)                          assert_almost_equal(norm(A, 'nuc'), 10.0)
1117: (8)                          assert_almost_equal(norm(A, inf), 12.0)
1118: (8)                          assert_almost_equal(norm(A, -inf), 4.0)
1119: (8)                          assert_almost_equal(norm(A, 1), 10.0)
1120: (8)                          assert_almost_equal(norm(A, -1), 6.0)
1121: (8)                          assert_almost_equal(norm(A, 2), 9.1231056256176615)
1122: (8)                          assert_almost_equal(norm(A, -2), 0.87689437438234041)
1123: (8)                          assert_raises(ValueError, norm, A, 'nofro')
1124: (8)                          assert_raises(ValueError, norm, A, -3)
1125: (8)                          assert_raises(ValueError, norm, A, 0)
1126: (4)                      def test_matrix_3x3(self):
1127: (8)                          A = (1 / 10) * \
1128: (12)                                self.array([[1, 2, 3], [6, 0, 5], [3, 2, 1]]], dtype=self.dt)
1129: (8)                          assert_almost_equal(norm(A), (1 / 10) * 89 ** 0.5)
1130: (8)                          assert_almost_equal(norm(A, 'fro'), (1 / 10) * 89 ** 0.5)
1131: (8)                          assert_almost_equal(norm(A, 'nuc'), 1.3366836911774836)

```

```

1132: (8)             assert_almost_equal(norm(A, inf), 1.1)
1133: (8)             assert_almost_equal(norm(A, -inf), 0.6)
1134: (8)             assert_almost_equal(norm(A, 1), 1.0)
1135: (8)             assert_almost_equal(norm(A, -1), 0.4)
1136: (8)             assert_almost_equal(norm(A, 2), 0.88722940323461277)
1137: (8)             assert_almost_equal(norm(A, -2), 0.19456584790481812)
1138: (4)             def test_bad_args(self):
1139: (8)                 A = self.array([[1, 2, 3], [4, 5, 6]], dtype=self.dt)
1140: (8)                 B = np.arange(1, 25, dtype=self.dt).reshape(2, 3, 4)
1141: (8)                 assert_raises(ValueError, norm, A, 'fro', 0)
1142: (8)                 assert_raises(ValueError, norm, A, 'nuc', 0)
1143: (8)                 assert_raises(ValueError, norm, [3, 4], 'fro', None)
1144: (8)                 assert_raises(ValueError, norm, [3, 4], 'nuc', None)
1145: (8)                 assert_raises(ValueError, norm, [3, 4], 'test', None)
1146: (8)                 for order in [0, 3]:
1147: (12)                     assert_raises(ValueError, norm, A, order, None)
1148: (12)                     assert_raises(ValueError, norm, A, order, (0, 1))
1149: (12)                     assert_raises(ValueError, norm, B, order, (1, 2))
1150: (8)                     assert_raises(np.AxisError, norm, B, None, 3)
1151: (8)                     assert_raises(np.AxisError, norm, B, None, (2, 3))
1152: (8)                     assert_raises(ValueError, norm, B, None, (0, 1, 2))
1153: (0)             class _TestNorm(_TestNorm2D, _TestNormGeneral):
1154: (4)                 pass
1155: (0)             class TestNorm_NonSystematic:
1156: (4)                 def test_longdouble_norm(self):
1157: (8)                     x = np.arange(10, dtype=np.longdouble)
1158: (8)                     old_assert_almost_equal(norm(x, ord=3), 12.65, decimal=2)
1159: (4)                 def test_intmin(self):
1160: (8)                     x = np.array([-2 ** 31], dtype=np.int32)
1161: (8)                     old_assert_almost_equal(norm(x, ord=3), 2 ** 31, decimal=5)
1162: (4)                 def test_complex_high_ord(self):
1163: (8)                     d = np.empty((2,), dtype=np.clongdouble)
1164: (8)                     d[0] = 6 + 7j
1165: (8)                     d[1] = -6 + 7j
1166: (8)                     res = 11.615898132184
1167: (8)                     old_assert_almost_equal(np.linalg.norm(d, ord=3), res, decimal=10)
1168: (8)                     d = d.astype(np.complex128)
1169: (8)                     old_assert_almost_equal(np.linalg.norm(d, ord=3), res, decimal=9)
1170: (8)                     d = d.astype(np.complex64)
1171: (8)                     old_assert_almost_equal(np.linalg.norm(d, ord=3), res, decimal=5)
1172: (0)             class _TestNormDoubleBase(_TestNormBase):
1173: (4)                 dt = np.double
1174: (4)                 dec = 12
1175: (0)             class _TestNormSingleBase(_TestNormBase):
1176: (4)                 dt = np.float32
1177: (4)                 dec = 6
1178: (0)             class _TestNormInt64Base(_TestNormBase):
1179: (4)                 dt = np.int64
1180: (4)                 dec = 12
1181: (0)             class TestNormDouble(_TestNorm, _TestNormDoubleBase):
1182: (4)                 pass
1183: (0)             class TestNormSingle(_TestNorm, _TestNormSingleBase):
1184: (4)                 pass
1185: (0)             class TestNormInt64(_TestNorm, _TestNormInt64Base):
1186: (4)                 pass
1187: (0)             class TestMatrixRank:
1188: (4)                 def test_matrix_rank(self):
1189: (8)                     assert_equal(4, matrix_rank(np.eye(4)))
1190: (8)                     I = np.eye(4)
1191: (8)                     I[-1, -1] = 0.
1192: (8)                     assert_equal(matrix_rank(I), 3)
1193: (8)                     assert_equal(matrix_rank(np.zeros((4, 4))), 0)
1194: (8)                     assert_equal(matrix_rank([1, 0, 0, 0]), 1)
1195: (8)                     assert_equal(matrix_rank(np.zeros((4,))), 0)
1196: (8)                     assert_equal(matrix_rank([1]), 1)
1197: (8)                     ms = np.array([I, np.eye(4), np.zeros((4, 4))])
1198: (8)                     assert_equal(matrix_rank(ms), np.array([3, 4, 0]))
1199: (8)                     assert_equal(matrix_rank(1), 1)
1200: (4)                 def test_symmetric_rank(self):

```

```

1201: (8)             assert_equal(4, matrix_rank(np.eye(4), hermitian=True))
1202: (8)             assert_equal(1, matrix_rank(np.ones((4, 4)), hermitian=True))
1203: (8)             assert_equal(0, matrix_rank(np.zeros((4, 4)), hermitian=True))
1204: (8)             I = np.eye(4)
1205: (8)             I[-1, -1] = 0.
1206: (8)             assert_equal(3, matrix_rank(I, hermitian=True))
1207: (8)             I[-1, -1] = 1e-8
1208: (8)             assert_equal(4, matrix_rank(I, hermitian=True, tol=0.99e-8))
1209: (8)             assert_equal(3, matrix_rank(I, hermitian=True, tol=1.01e-8))
1210: (0)             def test_reduced_rank():
1211: (4)                 rng = np.random.RandomState(20120714)
1212: (4)                 for i in range(100):
1213: (8)                     X = rng.normal(size=(40, 10))
1214: (8)                     X[:, 0] = X[:, 1] + X[:, 2]
1215: (8)                     assert_equal(matrix_rank(X), 9)
1216: (8)                     X[:, 3] = X[:, 4] + X[:, 5]
1217: (8)                     assert_equal(matrix_rank(X), 8)
1218: (0)             class TestQR:
1219: (4)                 array = np.array
1220: (4)                 def check_qr(self, a):
1221: (8)                     a_type = type(a)
1222: (8)                     a_dtype = a.dtype
1223: (8)                     m, n = a.shape
1224: (8)                     k = min(m, n)
1225: (8)                     res = linalg.qr(a, mode='complete')
1226: (8)                     Q, R = res.Q, res.R
1227: (8)                     assert_(Q.dtype == a_dtype)
1228: (8)                     assert_(R.dtype == a_dtype)
1229: (8)                     assert_(isinstance(Q, a_type))
1230: (8)                     assert_(isinstance(R, a_type))
1231: (8)                     assert_(Q.shape == (m, m))
1232: (8)                     assert_(R.shape == (m, n))
1233: (8)                     assert_almost_equal(dot(Q, R), a)
1234: (8)                     assert_almost_equal(dot(Q.T.conj(), Q), np.eye(m))
1235: (8)                     assert_almost_equal(np.triu(R), R)
1236: (8)                     q1, r1 = linalg.qr(a, mode='reduced')
1237: (8)                     assert_(q1.dtype == a_dtype)
1238: (8)                     assert_(r1.dtype == a_dtype)
1239: (8)                     assert_(isinstance(q1, a_type))
1240: (8)                     assert_(isinstance(r1, a_type))
1241: (8)                     assert_(q1.shape == (m, k))
1242: (8)                     assert_(r1.shape == (k, n))
1243: (8)                     assert_almost_equal(dot(q1, r1), a)
1244: (8)                     assert_almost_equal(dot(q1.T.conj(), q1), np.eye(k))
1245: (8)                     assert_almost_equal(np.triu(r1), r1)
1246: (8)                     r2 = linalg.qr(a, mode='r')
1247: (8)                     assert_(r2.dtype == a_dtype)
1248: (8)                     assert_(isinstance(r2, a_type))
1249: (8)                     assert_almost_equal(r2, r1)
1250: (4)             @pytest.mark.parametrize(["m", "n"], [
1251: (8)                 (3, 0),
1252: (8)                 (0, 3),
1253: (8)                 (0, 0)
1254: (4)             ])
1255: (4)             def test_qr_empty(self, m, n):
1256: (8)                 k = min(m, n)
1257: (8)                 a = np.empty((m, n))
1258: (8)                 self.check_qr(a)
1259: (8)                 h, tau = np.linalg.qr(a, mode='raw')
1260: (8)                 assert_equal(h.dtype, np.double)
1261: (8)                 assert_equal(tau.dtype, np.double)
1262: (8)                 assert_equal(h.shape, (n, m))
1263: (8)                 assert_equal(tau.shape, (k,))
1264: (4)             def test_mode_raw(self):
1265: (8)                 a = self.array([[1, 2], [3, 4], [5, 6]], dtype=np.double)
1266: (8)                 h, tau = linalg.qr(a, mode='raw')
1267: (8)                 assert_(h.dtype == np.double)
1268: (8)                 assert_(tau.dtype == np.double)
1269: (8)                 assert_(h.shape == (2, 3))

```

```

1270: (8)             assert_(tau.shape == (2,))
1271: (8)             h, tau = linalg.qr(a.T, mode='raw')
1272: (8)             assert_(h.dtype == np.double)
1273: (8)             assert_(tau.dtype == np.double)
1274: (8)             assert_(h.shape == (3, 2))
1275: (8)             assert_(tau.shape == (2,))
1276: (4)             def test_mode_all_but_economic(self):
1277: (8)                 a = self.array([[1, 2], [3, 4]])
1278: (8)                 b = self.array([[1, 2], [3, 4], [5, 6]])
1279: (8)                 for dt in "fd":
1280: (12)                     m1 = a.astype(dt)
1281: (12)                     m2 = b.astype(dt)
1282: (12)                     self.check_qr(m1)
1283: (12)                     self.check_qr(m2)
1284: (12)                     self.check_qr(m2.T)
1285: (8)                     for dt in "fd":
1286: (12)                         m1 = 1 + 1j * a.astype(dt)
1287: (12)                         m2 = 1 + 1j * b.astype(dt)
1288: (12)                         self.check_qr(m1)
1289: (12)                         self.check_qr(m2)
1290: (12)                         self.check_qr(m2.T)
1291: (4)             def check_qr_stacked(self, a):
1292: (8)                 a_type = type(a)
1293: (8)                 a_dtype = a.dtype
1294: (8)                 m, n = a.shape[-2:]
1295: (8)                 k = min(m, n)
1296: (8)                 q, r = linalg.qr(a, mode='complete')
1297: (8)                 assert_(q.dtype == a_dtype)
1298: (8)                 assert_(r.dtype == a_dtype)
1299: (8)                 assert_(isinstance(q, a_type))
1300: (8)                 assert_(isinstance(r, a_type))
1301: (8)                 assert_(q.shape[-2:] == (m, m))
1302: (8)                 assert_(r.shape[-2:] == (m, n))
1303: (8)                 assert_almost_equal(matmul(q, r), a)
1304: (8)                 I_mat = np.identity(q.shape[-1])
1305: (8)                 stack_I_mat = np.broadcast_to(I_mat,
1306: (24)                               q.shape[:-2] + (q.shape[-1],)*2)
1307: (8)                 assert_almost_equal(matmul(swapaxes(q, -1, -2).conj(), q),
1308: (8)                               stack_I_mat)
1309: (8)             assert_almost_equal(np.triu(r[..., :, :]), r)
1310: (8)             q1, r1 = linalg.qr(a, mode='reduced')
1311: (8)             assert_(q1.dtype == a_dtype)
1312: (8)             assert_(r1.dtype == a_dtype)
1313: (8)             assert_(isinstance(q1, a_type))
1314: (8)             assert_(isinstance(r1, a_type))
1315: (8)             assert_(q1.shape[-2:] == (m, k))
1316: (8)             assert_(r1.shape[-2:] == (k, n))
1317: (8)             assert_almost_equal(matmul(q1, r1), a)
1318: (8)             I_mat = np.identity(q1.shape[-1])
1319: (24)             stack_I_mat = np.broadcast_to(I_mat,
1320: (8)                               q1.shape[:-2] + (q1.shape[-1],)*2)
1321: (28)             assert_almost_equal(matmul(swapaxes(q1, -1, -2).conj(), q1),
1322: (8)                               stack_I_mat)
1323: (8)             assert_almost_equal(np.triu(r1[..., :, :]), r1)
1324: (8)             r2 = linalg.qr(a, mode='r')
1325: (8)             assert_(r2.dtype == a_dtype)
1326: (8)             assert_(isinstance(r2, a_type))
1327: (4)             assert_almost_equal(r2, r1)
1328: (8)             @pytest.mark.parametrize("size", [
1329: (8)                 (3, 4), (4, 3), (4, 4),
1330: (4)                 (3, 0), (0, 3)])
1331: (8)             @pytest.mark.parametrize("outer_size", [
1332: (4)                 (2, 2), (2,), (2, 3, 4)])
1333: (4)             @pytest.mark.parametrize("dt", [
1334: (8)                 np.single, np.double,
1335: (4)                 np.csingle, np.cdouble])
1336: (8)             def test_stacked_inputs(self, outer_size, size, dt):
1337: (8)                 A = np.random.normal(size=outer_size + size).astype(dt)

```

```

1338: (8)             self.check_qr_stacked(A)
1339: (8)             self.check_qr_stacked(A + 1.j*B)
1340: (0) class TestCholesky:
1341: (4)     @pytest.mark.parametrize(
1342: (8)         'shape', [(1, 1), (2, 2), (3, 3), (50, 50), (3, 10, 10)])
1343: (4)     ) @pytest.mark.parametrize(
1344: (8)         'dtype', (np.float32, np.float64, np.complex64, np.complex128)
1345: (4)     )
1346: (4) def test_basic_property(self, shape, dtype):
1347: (8)     np.random.seed(1)
1348: (8)     a = np.random.randn(*shape)
1349: (8)     if np.issubdtype(dtype, np.complexfloating):
1350: (8)         a = a + 1j*np.random.randn(*shape)
1351: (12)     t = list(range(len(shape)))
1352: (8)     t[-2:] = -1, -2
1353: (8)     a = np.matmul(a.transpose(t).conj(), a)
1354: (8)     a = np.asarray(a, dtype=dtype)
1355: (8)     c = np.linalg.cholesky(a)
1356: (8)     b = np.matmul(c, c.transpose(t).conj())
1357: (8)     with np._no_nep50_warning():
1358: (8)         atol = 500 * a.shape[0] * np.finfo(dtype).eps
1359: (12)     assert_allclose(b, a, atol=atol, err_msg=f'{shape} {dtype}\n{a}\n{c}')
1360: (8)
1361: (4) def test_0_size(self):
1362: (8)     class ArraySubclass(np.ndarray):
1363: (12)         pass
1364: (8)         a = np.zeros((0, 1, 1), dtype=np.int_).view(ArraySubclass)
1365: (8)         res = linalg.cholesky(a)
1366: (8)         assert_equal(a.shape, res.shape)
1367: (8)         assert_(res.dtype.type is np.float64)
1368: (8)         assert_(isinstance(res, np.ndarray))
1369: (8)         a = np.zeros((1, 0, 0), dtype=np.complex64).view(ArraySubclass)
1370: (8)         res = linalg.cholesky(a)
1371: (8)         assert_equal(a.shape, res.shape)
1372: (8)         assert_(res.dtype.type is np.complex64)
1373: (8)         assert_(isinstance(res, np.ndarray))
1374: (0) def test_byteorder_check():
1375: (4)     if sys.byteorder == 'little':
1376: (8)         native = '<'
1377: (4)     else:
1378: (8)         native = '>'
1379: (4)     for dtt in (np.float32, np.float64):
1380: (8)         arr = np.eye(4, dtype=dtt)
1381: (8)         n_arr = arr.newbyteorder(native)
1382: (8)         sw_arr = arr.newbyteorder('S').byteswap()
1383: (8)         assert_equal(arr.dtype.byteorder, '=')
1384: (8)         for routine in (linalg.inv, linalg.det, linalg.pinv):
1385: (12)             res = routine(arr)
1386: (12)             assert_array_equal(res, routine(n_arr))
1387: (12)             assert_array_equal(res, routine(sw_arr))
1388: (0) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1389: (0) def test_generalized_raise_multiloop():
1390: (4)     invertible = np.array([[1, 2], [3, 4]])
1391: (4)     non_invertible = np.array([[1, 1], [1, 1]])
1392: (4)     x = np.zeros([4, 4, 2])[1::2]
1393: (4)     x[...] = invertible
1394: (4)     x[0, 0] = non_invertible
1395: (4)     assert_raises(np.linalg.LinAlgError, np.linalg.inv, x)
1396: (0) def test_xerbla_override():
1397: (4)     XERBLA_OK = 255
1398: (4)     try:
1399: (8)         pid = os.fork()
1400: (4)     except (OSError, AttributeError):
1401: (8)         pytest.skip("Not POSIX or fork failed.")
1402: (4)     if pid == 0:
1403: (8)         os.close(1)
1404: (8)         os.close(0)
1405: (8)         import resource
1406: (8)         resource.setrlimit(resource.RLIMIT_CORE, (0, 0))

```

```

1407: (8)          try:
1408: (12)            np.linalg.lapack_lite.xerbla()
1409: (8)          except ValueError:
1410: (12)            pass
1411: (8)          except Exception:
1412: (12)            os._exit(os.EX_CONFIG)
1413: (8)          try:
1414: (12)            a = np.array([[1.]])
1415: (12)            np.linalg.lapack_lite.dorgqr(
1416: (16)              1, 1, 1, a,
1417: (16)              0, # <- invalid value
1418: (16)              a, a, 0, 0)
1419: (8)          except ValueError as e:
1420: (12)            if "DORGQR parameter number 5" in str(e):
1421: (16)              os._exit(XERBLA_OK)
1422: (8)            os._exit(os.EX_CONFIG)
1423: (4)          else:
1424: (8)            pid, status = os.wait()
1425: (8)            if os.WEXITSTATUS(status) != XERBLA_OK:
1426: (12)              pytest.skip('Numpy xerbla not linked in.')
1427: (0)          @pytest.mark.skipif(IS_WASM, reason="Cannot start subprocess")
1428: (0)          @pytest.mark.slow
1429: (0)          def test_sdot_bug_8577():
1430: (4)            bad_libs = ['PyQt5.QtWidgets', 'IPython']
1431: (4)            template = textwrap.dedent("""
1432: (4)              import sys
1433: (4)              {before}
1434: (4)              try:
1435: (8)                import {bad_lib}
1436: (4)              except ImportError:
1437: (8)                sys.exit(0)
1438: (4)              {after}
1439: (4)              x = np.ones(2, dtype=np.float32)
1440: (4)              sys.exit(0 if np.allclose(x.dot(x), 2.0) else 1)
1441: (4)              """
1442: (4)            for bad_lib in bad_libs:
1443: (8)              code = template.format(before="import numpy as np", after="",
1444: (31)                bad_lib=bad_lib)
1445: (8)              subprocess.check_call([sys.executable, "-c", code])
1446: (8)              code = template.format(after="import numpy as np", before="",
1447: (31)                bad_lib=bad_lib)
1448: (8)              subprocess.check_call([sys.executable, "-c", code])
1449: (0)          class TestMultiDot:
1450: (4)            def test_basic_function_with_three_arguments(self):
1451: (8)              A = np.random.random((6, 2))
1452: (8)              B = np.random.random((2, 6))
1453: (8)              C = np.random.random((6, 2))
1454: (8)              assert_almost_equal(multi_dot([A, B, C]), A.dot(B).dot(C))
1455: (8)              assert_almost_equal(multi_dot([A, B, C]), np.dot(A, np.dot(B, C)))
1456: (4)            def test_basic_function_with_two_arguments(self):
1457: (8)              A = np.random.random((6, 2))
1458: (8)              B = np.random.random((2, 6))
1459: (8)              assert_almost_equal(multi_dot([A, B]), A.dot(B))
1460: (8)              assert_almost_equal(multi_dot([A, B]), np.dot(A, B))
1461: (4)            def test_basic_function_with_dynamic_programming_optimization(self):
1462: (8)              A = np.random.random((6, 2))
1463: (8)              B = np.random.random((2, 6))
1464: (8)              C = np.random.random((6, 2))
1465: (8)              D = np.random.random((2, 1))
1466: (8)              assert_almost_equal(multi_dot([A, B, C, D]), A.dot(B).dot(C).dot(D))
1467: (4)            def test_vector_as_first_argument(self):
1468: (8)              A1d = np.random.random(2) # 1-D
1469: (8)              B = np.random.random((2, 6))
1470: (8)              C = np.random.random((6, 2))
1471: (8)              D = np.random.random((2, 2))
1472: (8)              assert_equal(multi_dot([A1d, B, C, D]).shape, (2,))
1473: (4)            def test_vector_as_last_argument(self):
1474: (8)              A = np.random.random((6, 2))
1475: (8)              B = np.random.random((2, 6))

```

```

1476: (8)             C = np.random.random((6, 2))
1477: (8)             D1d = np.random.random(2) # 1-D
1478: (8)             assert_equal(multi_dot([A, B, C, D1d]).shape, (6,))
1479: (4)             def test_vector_as_first_and_last_argument(self):
1480: (8)                 A1d = np.random.random(2) # 1-D
1481: (8)                 B = np.random.random((2, 6))
1482: (8)                 C = np.random.random((6, 2))
1483: (8)                 D1d = np.random.random(2) # 1-D
1484: (8)                 assert_equal(multi_dot([A1d, B, C, D1d]).shape, ())
1485: (4)             def test_three_arguments_and_out(self):
1486: (8)                 A = np.random.random((6, 2))
1487: (8)                 B = np.random.random((2, 6))
1488: (8)                 C = np.random.random((6, 2))
1489: (8)                 out = np.zeros((6, 2))
1490: (8)                 ret = multi_dot([A, B, C], out=out)
1491: (8)                 assert out is ret
1492: (8)                 assert_almost_equal(out, A.dot(B).dot(C))
1493: (8)                 assert_almost_equal(out, np.dot(A, np.dot(B, C)))
1494: (4)             def test_two_arguments_and_out(self):
1495: (8)                 A = np.random.random((6, 2))
1496: (8)                 B = np.random.random((2, 6))
1497: (8)                 out = np.zeros((6, 6))
1498: (8)                 ret = multi_dot([A, B], out=out)
1499: (8)                 assert out is ret
1500: (8)                 assert_almost_equal(out, A.dot(B))
1501: (8)                 assert_almost_equal(out, np.dot(A, B))
1502: (4)             def test_dynamic_programming_optimization_and_out(self):
1503: (8)                 A = np.random.random((6, 2))
1504: (8)                 B = np.random.random((2, 6))
1505: (8)                 C = np.random.random((6, 2))
1506: (8)                 D = np.random.random((2, 1))
1507: (8)                 out = np.zeros((6, 1))
1508: (8)                 ret = multi_dot([A, B, C, D], out=out)
1509: (8)                 assert out is ret
1510: (8)                 assert_almost_equal(out, A.dot(B).dot(C).dot(D))
1511: (4)             def test_dynamic_programming_logic(self):
1512: (8)                 arrays = [np.random.random((30, 35)),
1513: (18)                     np.random.random((35, 15)),
1514: (18)                     np.random.random((15, 5)),
1515: (18)                     np.random.random((5, 10)),
1516: (18)                     np.random.random((10, 20)),
1517: (18)                     np.random.random((20, 25))]
1518: (8)                 m_expected = np.array([[0., 15750., 7875., 9375., 11875., 15125.],
1519: (31)                     [0., 0., 2625., 4375., 7125., 10500.],
1520: (31)                     [0., 0., 0., 750., 2500., 5375.],
1521: (31)                     [0., 0., 0., 0., 1000., 3500.],
1522: (31)                     [0., 0., 0., 0., 0., 5000.],
1523: (31)                     [0., 0., 0., 0., 0., 0.]]])
1524: (8)                 s_expected = np.array([[0, 1, 1, 3, 3, 3],
1525: (31)                     [0, 0, 2, 3, 3, 3],
1526: (31)                     [0, 0, 0, 3, 3, 3],
1527: (31)                     [0, 0, 0, 0, 4, 5],
1528: (31)                     [0, 0, 0, 0, 0, 5],
1529: (31)                     [0, 0, 0, 0, 0, 0]], dtype=int)
1530: (8)                 s_expected -= 1 # Cormen uses 1-based index, python does not.
1531: (8)                 s, m = _multi_dot_matrix_chain_order(arrays, return_costs=True)
1532: (8)                 assert_almost_equal(np.triu(s[:-1, 1:]),
1533: (28)                     np.triu(s_expected[:-1, 1:]))
1534: (8)                 assert_almost_equal(np.triu(m), np.triu(m_expected))
1535: (4)             def test_too_few_input_arrays(self):
1536: (8)                 assert_raises(ValueError, multi_dot, [])
1537: (8)                 assert_raises(ValueError, multi_dot, [np.random.random((3, 3))])
1538: (0)             class TestTensorinv:
1539: (4)                 @pytest.mark.parametrize("arr, ind", [
1540: (8)                     (np.ones((4, 6, 8, 2)), 2),
1541: (8)                     (np.ones((3, 3, 2)), 1),
1542: (8)                     ])
1543: (4)                 def test_non_square_handling(self, arr, ind):
1544: (8)                     with assert_raises(LinAlgError):

```

```

1545: (12)
1546: (4)
1547: (8)
1548: (8)
1549: (8)
1550: (4)
1551: (8)
1552: (8)
1553: (8)
1554: (8)
1555: (8)
1556: (8)
1557: (4)
1558: (8)
1559: (8)
1560: (4)
1561: (8)
1562: (8)
1563: (8)
1564: (12)
1565: (4)
1566: (8)
1567: (8)
1568: (8)
1569: (8)
1570: (8)
1571: (0)
1572: (4)
1573: (8)
1574: (8)
1575: (8)
1576: (4)
1577: (8)
1578: (12)
1579: (12)
1580: (4)
1581: (8)
1582: (4)
1583: (4)
1584: (8)
1585: (8)
1586: (8)
1587: (8)
1588: (0)
1589: (4)
1590: (4)
1591: (8)
1592: (0)
1593: (0)
1594: (4)
1595: (4)
1596: (4)
1597: (4)
1598: (4)
1599: (4)
1600: (0)
1601: (19)
1602: (0)
1603: (4)
1604: (4)
1605: (4)
1606: (4)
1607: (4)
1608: (4)
1609: (4)
1610: (4)
1611: (4)
1612: (4)
1613: (4)

        linalg.tensorinv(arr, ind=ind)
    @pytest.mark.parametrize("shape, ind", [
        ((4, 6, 8, 3), 2),
        ((24, 8, 3), 1),
    ])
    def test_tensorinv_shape(self, shape, ind):
        a = np.eye(24)
        a.shape = shape
        ainv = linalg.tensorinv(a=a, ind=ind)
        expected = a.shape[ind:] + a.shape[:ind]
        actual = ainv.shape
        assert_equal(actual, expected)
    @pytest.mark.parametrize("ind", [
        0, -2,
    ])
    def test_tensorinv_ind_limit(self, ind):
        a = np.eye(24)
        a.shape = (4, 6, 8, 3)
        with assert_raises(ValueError):
            linalg.tensorinv(a=a, ind=ind)
    def test_tensorinv_result(self):
        a = np.eye(24)
        a.shape = (24, 8, 3)
        ainv = linalg.tensorinv(a, ind=1)
        b = np.ones(24)
        assert_allclose(np.tensordot(ainv, b, 1), np.linalg.tensorsolve(a, b))
class TestTensorsolve:
    @pytest.mark.parametrize("a, axes", [
        (np.ones((4, 6, 8, 2)), None),
        (np.ones((3, 3, 2)), (0, 2)),
    ])
    def test_non_square_handling(self, a, axes):
        with assert_raises(LinAlgError):
            b = np.ones(a.shape[:2])
            linalg.tensorsolve(a, b, axes=axes)
    @pytest.mark.parametrize("shape",
                           [(2, 3, 6), (3, 4, 4, 3), (0, 3, 3, 0)],
                           )
    def test_tensorsolve_result(self, shape):
        a = np.random.randn(*shape)
        b = np.ones(a.shape[:2])
        x = np.linalg.tensorsolve(a, b)
        assert_allclose(np.tensordot(a, x, axes=len(x.shape)), b)
    def test_unsupported_commonctype():
        arr = np.array([[1, -2], [2, 5]], dtype='float16')
        with assert_raises_regex(TypeError, "unsupported in linalg"):
            linalg.cholesky(arr)
    @pytest.mark.skip(reason="Bad memory reports lead to OOM in ci testing")
    def test_blas64_dot():
        n = 2**32
        a = np.zeros([1, n], dtype=np.float32)
        b = np.ones([1, 1], dtype=np.float32)
        a[0, -1] = 1
        c = np.dot(b, a)
        assert_equal(c[0, -1], 1)
    @pytest.mark.xfail(not HAS_LAPACK64,
                       reason="Numpy not compiled with 64-bit BLAS/LAPACK")
    def test_blas64_geqrf_lwork_smoketest():
        dtype = np.float64
        lapack_routine = np.linalg.lapack_lite.dgeqrf
        m = 2**32 + 1
        n = 2**32 + 1
        lda = m
        a = np.zeros([1, 1], dtype=dtype)
        work = np.zeros([1], dtype=dtype)
        tau = np.zeros([1], dtype=dtype)
        results = lapack_routine(m, n, a, lda, tau, work, -1, 0)
        assert_equal(results['info'], 0)
        assert_equal(results['m'], m)

```

```

1614: (4)         assert_equal(results['n'], m)
1615: (4)         lwork = int(work.item())
1616: (4)         assert_(2**32 < lwork < 2**42)
-----
```

## File 259 - test\_regression.py:

```

1: (0)             """ Test functions for linalg module
2: (0)             """
3: (0)             import warnings
4: (0)             import numpy as np
5: (0)             from numpy import linalg, arange, float64, array, dot, transpose
6: (0)             from numpy.testing import (
7: (4)                 assert_, assert_raises, assert_equal, assert_array_equal,
8: (4)                 assert_array_almost_equal, assert_array_less
9: (0)             )
10: (0)            class TestRegression:
11: (4)                def test_eig_build(self):
12: (8)                    rva = array([1.03221168e+02 + 0.j,
13: (21)                      -1.91843603e+01 + 0.j,
14: (21)                      -6.04004526e-01 + 15.84422474j,
15: (21)                      -6.04004526e-01 - 15.84422474j,
16: (21)                      -1.13692929e+01 + 0.j,
17: (21)                      -6.57612485e-01 + 10.41755503j,
18: (21)                      -6.57612485e-01 - 10.41755503j,
19: (21)                      1.82126812e+01 + 0.j,
20: (21)                      1.06011014e+01 + 0.j,
21: (21)                      7.80732773e+00 + 0.j,
22: (21)                      -7.65390898e-01 + 0.j,
23: (21)                      1.51971555e-15 + 0.j,
24: (21)                      -1.51308713e-15 + 0.j])
25: (8)                    a = arange(13 * 13, dtype=float64)
26: (8)                    a.shape = (13, 13)
27: (8)                    a = a % 17
28: (8)                    va, ve = linalg.eig(a)
29: (8)                    va.sort()
30: (8)                    rva.sort()
31: (8)                    assert_array_almost_equal(va, rva)
32: (4)                def test_eigh_build(self):
33: (8)                    rvals = [68.60568999, 89.57756725, 106.67185574]
34: (8)                    cov = array([[77.70273908, 3.51489954, 15.64602427],
35: (21)                      [3.51489954, 88.97013878, -1.07431931],
36: (21)                      [15.64602427, -1.07431931, 98.18223512]])
37: (8)                    vals, vecs = linalg.eigh(cov)
38: (8)                    assert_array_almost_equal(vals, rvals)
39: (4)                def test_svd_build(self):
40: (8)                    a = array([[0., 1.], [1., 1.], [2., 1.], [3., 1.]])
41: (8)                    m, n = a.shape
42: (8)                    u, s, vh = linalg.svd(a)
43: (8)                    b = dot(transpose(u[:, n:]), a)
44: (8)                    assert_array_almost_equal(b, np.zeros((2, 2)))
45: (4)                def test_norm_vector_badarg(self):
46: (8)                    assert_raises(ValueError, linalg.norm, array([1., 2., 3.]), 'fro')
47: (4)                def test_lapack_endian(self):
48: (8)                    a = array([[5.7998084, -2.1825367],
49: (19)                      [-2.1825367, 9.85910595]], dtype='>f8')
50: (8)                    b = array(a, dtype='<f8')
51: (8)                    ap = linalg.cholesky(a)
52: (8)                    bp = linalg.cholesky(b)
53: (8)                    assert_array_equal(ap, bp)
54: (4)                def test_large_svd_32bit(self):
55: (8)                    x = np.eye(1000, 66)
56: (8)                    np.linalg.svd(x)
57: (4)                def test_svd_no_uv(self):
58: (8)                    for shape in (3, 4), (4, 4), (4, 3):
59: (12)                        for t in float, complex:
60: (16)                            a = np.ones(shape, dtype=t)
61: (16)                            w = linalg.svd(a, compute_uv=False)
```

```

62: (16)             c = np.count_nonzero(np.absolute(w) > 0.5)
63: (16)             assert_equal(c, 1)
64: (16)             assert_equal(np.linalg.matrix_rank(a), 1)
65: (16)             assert_array_less(1, np.linalg.norm(a, ord=2))
66: (4)              def test_norm_object_array(self):
67: (8)                testvector = np.array([np.array([0, 1]), 0, 0], dtype=object)
68: (8)                norm = linalg.norm(testvector)
69: (8)                assert_array_equal(norm, [0, 1])
70: (8)                assert_(norm.dtype == np.dtype('float64'))
71: (8)                norm = linalg.norm(testvector, ord=1)
72: (8)                assert_array_equal(norm, [0, 1])
73: (8)                assert_(norm.dtype != np.dtype('float64'))
74: (8)                norm = linalg.norm(testvector, ord=2)
75: (8)                assert_array_equal(norm, [0, 1])
76: (8)                assert_(norm.dtype == np.dtype('float64'))
77: (8)                assert_raises(ValueError, linalg.norm, testvector, ord='fro')
78: (8)                assert_raises(ValueError, linalg.norm, testvector, ord='nuc')
79: (8)                assert_raises(ValueError, linalg.norm, testvector, ord=np.inf)
80: (8)                assert_raises(ValueError, linalg.norm, testvector, ord=-np.inf)
81: (8)                assert_raises(ValueError, linalg.norm, testvector, ord=0)
82: (8)                assert_raises(ValueError, linalg.norm, testvector, ord=-1)
83: (8)                assert_raises(ValueError, linalg.norm, testvector, ord=-2)
84: (8)                testmatrix = np.array([[np.array([0, 1]), 0, 0],
85: (31)                               [0, 0, 0]], dtype=object)
86: (8)                norm = linalg.norm(testmatrix)
87: (8)                assert_array_equal(norm, [0, 1])
88: (8)                assert_(norm.dtype == np.dtype('float64'))
89: (8)                norm = linalg.norm(testmatrix, ord='fro')
90: (8)                assert_array_equal(norm, [0, 1])
91: (8)                assert_(norm.dtype == np.dtype('float64'))
92: (8)                assert_raises(TypeError, linalg.norm, testmatrix, ord='nuc')
93: (8)                assert_raises(ValueError, linalg.norm, testmatrix, ord=np.inf)
94: (8)                assert_raises(ValueError, linalg.norm, testmatrix, ord=-np.inf)
95: (8)                assert_raises(ValueError, linalg.norm, testmatrix, ord=0)
96: (8)                assert_raises(ValueError, linalg.norm, testmatrix, ord=1)
97: (8)                assert_raises(ValueError, linalg.norm, testmatrix, ord=-1)
98: (8)                assert_raises(TypeError, linalg.norm, testmatrix, ord=2)
99: (8)                assert_raises(TypeError, linalg.norm, testmatrix, ord=-2)
100: (8)               assert_raises(ValueError, linalg.norm, testmatrix, ord=3)
101: (4)              def test_lstsq_complex_larger_rhs(self):
102: (8)                size = 20
103: (8)                n_rhs = 70
104: (8)                G = np.random.randn(size, size) + 1j * np.random.randn(size, size)
105: (8)                u = np.random.randn(size, n_rhs) + 1j * np.random.randn(size, n_rhs)
106: (8)                b = G.dot(u)
107: (8)                u_lstsq, res, rank, sv = linalg.lstsq(G, b, rcond=None)
108: (8)                assert_array_almost_equal(u_lstsq, u)

```

-----  
File 260 - \_\_init\_\_.py:

1: (0)

-----  
File 261 - core.py:

```

1: (0)      """
2: (0)      numpy.ma : a package to handle missing or invalid values.
3: (0)      This package was initially written for numarray by Paul F. Dubois
4: (0)      at Lawrence Livermore National Laboratory.
5: (0)      In 2006, the package was completely rewritten by Pierre Gerard-Marchant
6: (0)      (University of Georgia) to make the MaskedArray class a subclass of ndarray,
7: (0)      and to improve support of structured arrays.
8: (0)      Copyright 1999, 2000, 2001 Regents of the University of California.
9: (0)      Released for unlimited redistribution.
10: (0)      * Adapted for numpy_core 2005 by Travis Oliphant and (mainly) Paul Dubois.
11: (0)      * Subclassing of the base `ndarray` 2006 by Pierre Gerard-Marchant

```

```

12: (2)             (pgmdevlist_AT_gmail_DOT_com)
13: (0)             * Improvements suggested by Reggie Dugard (reggie_AT_merfinllc_DOT_com)
14: (0)             .. moduleauthor:: Pierre Gerard-Marchant
15: (0)
16: (0)             """
17: (0)             import builtins
18: (0)             import inspect
19: (0)             import operator
20: (0)             import warnings
21: (0)             import textwrap
22: (0)             import re
23: (0)             from functools import reduce
24: (0)             import numpy as np
25: (0)             import numpy.core.umath as umath
26: (0)             import numpy.core.numericatypes as ntypes
27: (0)             from numpy.core import multiarray as mu
28: (0)             from numpy import ndarray, amax, amin, iscomplexobj, bool_, _NoValue
29: (0)             from numpy import array as narray
30: (0)             from numpy.lib.function_base import angle
31: (0)             from numpy.compat import (
32: (4)                 getargspec, formatargspec, long, unicode, bytes
33: (4)             )
34: (0)             from numpy import expand_dims
35: (0)             from numpy.core.numeric import normalize_axis_tuple
36: (0)             __all__ = [
37: (4)                 'MAError', 'MaskError', 'MaskType', 'MaskedArray', 'abs', 'absolute',
38: (4)                 'add', 'all', 'allclose', 'allequal', 'alltrue', 'amax', 'amin',
39: (4)                 'angle', 'anom', 'anomalies', 'any', 'append', 'arange', 'arccos',
40: (4)                 'arccosh', 'arcsin', 'arcsinh', 'arctan', 'arctan2', 'arctanh',
41: (4)                 'argmax', 'argmin', 'argsort', 'around', 'array', 'asanyarray',
42: (4)                 'asarray', 'bitwise_and', 'bitwise_or', 'bitwise_xor', 'bool_', 'ceil',
43: (4)                 'choose', 'clip', 'common_fill_value', 'compress', 'compressed',
44: (4)                 'concatenate', 'conjugate', 'convolve', 'copy', 'correlate', 'cos',
45: (4)                 'cosh',
46: (4)                 'count', 'cumprod', 'cumsum', 'default_fill_value', 'diag', 'diagonal',
47: (4)                 'diff', 'divide', 'empty', 'empty_like', 'equal', 'exp',
48: (4)                 'expand_dims', 'fabs', 'filled', 'fix_invalid', 'flatten_mask',
49: (4)                 'flatten_structured_array', 'floor', 'floor_divide', 'fmod',
50: (4)                 'frombuffer', 'fromflex', 'fromfunction', 'getdata', 'getmask',
51: (4)                 'getmaskarray', 'greater', 'greater_equal', 'harden_mask', 'hypot',
52: (4)                 'identity', 'ids', 'indices', 'inner', 'innerproduct', 'isMA',
53: (4)                 'isMaskedArray', 'is_mask', 'is_masked', 'isarray', 'left_shift',
54: (4)                 'less', 'less_equal', 'log', 'log10', 'log2',
55: (4)                 'logical_and', 'logical_not', 'logical_or', 'logical_xor', 'make_mask',
56: (4)                 'make_mask_descr', 'make_mask_none', 'mask_or', 'masked',
57: (4)                 'masked_array', 'masked_equal', 'masked_greater',
58: (4)                 'masked_greater_equal', 'masked_inside', 'masked_invalid',
59: (4)                 'masked_less', 'masked_less_equal', 'masked_not_equal',
60: (4)                 'masked_object', 'masked_outside', 'masked_print_option',
61: (4)                 'masked_singleton', 'masked_values', 'masked_where', 'max', 'maximum',
62: (4)                 'maximum_fill_value', 'mean', 'min', 'minimum', 'minimum_fill_value',
63: (4)                 'mod', 'multiply', 'mvoid', 'ndim', 'negative', 'nomask', 'nonzero',
64: (4)                 'not_equal', 'ones', 'ones_like', 'outer', 'outerproduct', 'power',
65: (4)                 'prod',
66: (4)                 'product', 'ptp', 'put', 'putmask', 'ravel', 'remainder',
67: (4)                 'repeat', 'reshape', 'resize', 'right_shift', 'round', 'round_',
68: (4)                 'set_fill_value', 'shape', 'sin', 'sinh', 'size', 'soften_mask',
69: (4)                 'sometrue', 'sort', 'sqrt', 'squeeze', 'std', 'subtract', 'sum',
70: (0)                 'swapaxes', 'take', 'tan', 'tanh', 'trace', 'transpose', 'true_divide',
71: (0)                 'var', 'where', 'zeros', 'zeros_like',
72: (0)             ]
73: (4)             MaskType = np.bool_
74: (0)             nomask = MaskType(0)
75: (0)             class MaskedArrayFutureWarning(FutureWarning):
76: (4)                 pass
77: (4)             def __deprecate_argsort_axis(arr):
78: (4)                 """

```

Adjust the axis passed to argsort, warning if necessary

Parameters

-----

```

79: (4) arr
80: (8)     The array which argsort was called on
81: (4) np.ma.argsort has a long-term bug where the default of the axis argument
82: (4) is wrong (gh-8701), which now must be kept for backwards compatibility.
83: (4) Thankfully, this only makes a difference when arrays are 2- or more-
84: (4) dimensional, so we only need a warning then.
85: (4) """
86: (4) if arr.ndim <= 1:
87: (8)     return -1
88: (4) else:
89: (8)     warnings.warn(
90: (12)         "In the future the default for argsort will be axis=-1, not the "
91: (12)         "current None, to match its documentation and np.argsort. "
92: (12)         "Explicitly pass -1 or None to silence this warning.",
93: (12)         MaskedArrayFutureWarning, stacklevel=3)
94: (8)     return None
95: (0) def doc_note(initialdoc, note):
96: (4) """
97: (4)     Adds a Notes section to an existing docstring.
98: (4) """
99: (4)     if initialdoc is None:
100: (8)         return
101: (4)     if note is None:
102: (8)         return initialdoc
103: (4)     notesplit = re.split(r'\n\s*?Notes\n\s*?----',
inspect.cleandoc(initialdoc))
104: (4)     notedoc = "\n\nNotes\n----\n%s\n" % inspect.cleandoc(note)
105: (4)     return ''.join(notesplit[:1] + [notedoc] + notesplit[1:])
106: (0) def get_object_signature(obj):
107: (4) """
108: (4)     Get the signature from obj
109: (4) """
110: (4)     try:
111: (8)         sig = formatargspec(*getargspec(obj))
112: (4)     except TypeError:
113: (8)         sig = ''
114: (4)     return sig
115: (0) class MAError(Exception):
116: (4) """
117: (4)     Class for masked array related errors.
118: (4) """
119: (4)     pass
120: (0) class MaskError(MAError):
121: (4) """
122: (4)     Class for mask related errors.
123: (4) """
124: (4)     pass
125: (0) default.filler = {'b': True,
126: (18)             'c': 1.e20 + 0.0j,
127: (18)             'f': 1.e20,
128: (18)             'i': 999999,
129: (18)             'O': '?',
130: (18)             'S': b'N/A',
131: (18)             'u': 999999,
132: (18)             'V': b'????',
133: (18)             'U': 'N/A'
134: (18)             }
135: (0)     for v in ["Y", "M", "W", "D", "h", "m", "s", "ms", "us", "ns", "ps",
136: (10)             "fs", "as"]:
137: (4)         default.filler["M8[" + v + "]"] = np.datetime64("NaT", v)
138: (4)         default.filler["m8[" + v + "]"] = np.timedelta64("NaT", v)
139: (0)         float_types_list = [np.half, np.single, np.double, np.longdouble,
140: (20)             np.csingle, np.cdouble, np.clongdouble]
141: (0)         max.filler = ntypes._minvals
142: (0)         max.filler.update([(k, -np.inf) for k in float_types_list[:4]])
143: (0)         max.filler.update([(k, complex(-np.inf, -np.inf)) for k in
float_types_list[-3:]])
144: (0)         min.filler = ntypes._maxvals
145: (0)         min.filler.update([(k, +np.inf) for k in float_types_list[:4]])

```

```

146: (0)         minFiller.update([(k, complex(+np.inf, +np.inf)) for k in
147: (0)             float_types_list[-3:]])
148: (0)             del float_types_list
149: (4)             def _recursive_fill_value(dtype, f):
150: (4)                 """"
151: (4)                 Recursively produce a fill value for `dtype`, calling f on scalar dtypes
152: (4)                 """
153: (8)                 if dtype.names is not None:
154: (16)                     vals = tuple(
155: (16)                         np.array(_recursive_fill_value(dtype[name], f))
156: (16)                         for name in dtype.names)
157: (8)                     return np.array(vals, dtype=dtype)[()] # decay to void scalar from 0d
158: (4)                 elif dtype.subdtype:
159: (8)                     subtype, shape = dtype.subdtype
160: (8)                     subval = _recursive_fill_value(subtype, f)
161: (8)                     return np.full(shape, subval)
162: (4)                 else:
163: (8)                     return f(dtype)
164: (0)             def _get_dtype_of(obj):
165: (4)                 """ Convert the argument for *_fill_value into a dtype """
166: (8)                 if isinstance(obj, np.dtype):
167: (8)                     return obj
168: (8)                 elif hasattr(obj, 'dtype'):
169: (8)                     return obj.dtype
170: (4)                 else:
171: (8)                     return np.asarray(obj).dtype
172: (0)             def default_fill_value(obj):
173: (4)                 """
174: (4)                 Return the default fill value for the argument object.
175: (4)                 The default filling value depends on the datatype of the input
176: (7)                     array or the type of the input scalar:
177: (7)                     ===== =====
178: (7)                     datatype default
179: (7)                     ===== =====
180: (7)                     bool      True
181: (7)                     int       999999
182: (7)                     float     1.e20
183: (7)                     complex   1.e20+0j
184: (7)                     object    '?'
185: (7)                     string    'N/A'
186: (7)                     ===== =====
187: (4)                 For structured types, a structured scalar is returned, with each field the
188: (4)                     default fill value for its type.
189: (4)                 For subarray types, the fill value is an array of the same size containing
190: (4)                     the default scalar fill value.
191: (4)             Parameters
192: (4)             -----
193: (8)             obj : ndarray, dtype or scalar
194: (8)                 The array data-type or scalar for which the default fill value
195: (8)                     is returned.
196: (4)             Returns
197: (4)             -----
198: (8)             fill_value : scalar
199: (8)                 The default fill value.
200: (4)             Examples
201: (4)             -----
202: (4)             >>> np.ma.default_fill_value(1)
203: (4)             999999
204: (4)             >>> np.ma.default_fill_value(np.array([1.1, 2., np.pi]))
205: (4)             1e+20
206: (4)             >>> np.ma.default_fill_value(np.dtype(complex))
207: (4)             (1e+20+0j)
208: (4)             """
209: (8)             def _scalar_fill_value(dtype):
210: (12)                 if dtype.kind in 'Mm':
211: (8)                     return defaultFiller.get(dtype.str[1:], '?')
212: (12)                 else:
213: (8)                     return defaultFiller.get(dtype.kind, '?')
214: (4)             dtype = _get_dtype_of(obj)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

214: (4)           return _recursive_fill_value(dtype, _scalar_fill_value)
215: (0)           def _extremum_fill_value(obj, extremum, extremum_name):
216: (4)             def _scalar_fill_value(dtype):
217: (8)               try:
218: (12)                 return extremum[dtype]
219: (8)             except KeyError as e:
220: (12)               raise TypeError(
221: (16)                 f"Unsuitable type {dtype} for calculating {extremum_name}."
222: (12)             ) from None
223: (4)             dtype = _get_dtype_of(obj)
224: (4)             return _recursive_fill_value(dtype, _scalar_fill_value)
225: (0)           def minimum_fill_value(obj):
226: (4)             """
227: (4)             Return the maximum value that can be represented by the dtype of an
object.
228: (4)             This function is useful for calculating a fill value suitable for
taking the minimum of an array with a given dtype.
229: (4)             Parameters
230: (4)               -----
231: (4)               obj : ndarray, dtype or scalar
232: (4)                 An object that can be queried for it's numeric type.
233: (8)             Returns
234: (4)               -----
235: (4)               val : scalar
236: (4)                 The maximum representable value.
237: (8)             Raises
238: (4)               -----
239: (4)               TypeError
240: (4)                 If `obj` isn't a suitable numeric type.
241: (8)             See Also
242: (4)               -----
243: (4)               maximum_fill_value : The inverse function.
244: (4)               set_fill_value : Set the filling value of a masked array.
245: (4)               MaskedArray.fill_value : Return current fill value.
246: (4)             Examples
247: (4)               -----
248: (4)               >>> import numpy.ma as ma
249: (4)               >>> a = np.int8()
250: (4)               >>> ma.minimum_fill_value(a)
251: (4)               127
252: (4)               >>> a = np.int32()
253: (4)               >>> ma.minimum_fill_value(a)
254: (4)               2147483647
255: (4)               An array of numeric data can also be passed.
256: (4)               >>> a = np.array([1, 2, 3], dtype=np.int8)
257: (4)               >>> ma.minimum_fill_value(a)
258: (4)               127
259: (4)               >>> a = np.array([1, 2, 3], dtype=np.float32)
260: (4)               >>> ma.minimum_fill_value(a)
261: (4)               inf
262: (4)               """
263: (4)               return _extremum_fill_value(obj, minFiller, "minimum")
264: (4)           def maximum_fill_value(obj):
265: (0)             """
266: (4)             Return the minimum value that can be represented by the dtype of an
object.
267: (4)             This function is useful for calculating a fill value suitable for
taking the maximum of an array with a given dtype.
268: (4)             Parameters
269: (4)               -----
270: (4)               obj : ndarray, dtype or scalar
271: (4)                 An object that can be queried for it's numeric type.
272: (4)             Returns
273: (4)               -----
274: (4)               val : scalar
275: (4)                 The minimum representable value.
276: (4)             Raises
277: (4)               -----
278: (4)               TypeError
279: (4)             """

```

```

281: (8)             If `obj` isn't a suitable numeric type.
282: (4)             See Also
283: (4)
284: (4)             -----
285: (4)             minimum_fill_value : The inverse function.
286: (4)             set_fill_value : Set the filling value of a masked array.
287: (4)             MaskedArray.fill_value : Return current fill value.
288: (4)
289: (4)             Examples
290: (4)             -----
291: (4)             >>> import numpy.ma as ma
292: (4)             >>> a = np.int8()
293: (4)             >>> ma.maximum_fill_value(a)
294: (4)             -128
295: (4)             >>> a = np.int32()
296: (4)             >>> ma.maximum_fill_value(a)
297: (4)             -2147483648
298: (4)             An array of numeric data can also be passed.
299: (4)             >>> a = np.array([1, 2, 3], dtype=np.int8)
300: (4)             >>> ma.maximum_fill_value(a)
301: (4)             -128
302: (4)             >>> a = np.array([1, 2, 3], dtype=np.float32)
303: (4)             >>> ma.maximum_fill_value(a)
304: (4)             -inf
305: (0)             """
306: (4)             return _extremum_fill_value(obj, max_filler, "maximum")
307: (4)         def _recursive_set_fill_value(fillvalue, dt):
308: (4)             """
309: (4)             Create a fill value for a structured dtype.
310: (4)             Parameters
311: (4)             -----
312: (4)             fillvalue : scalar or array_like
313: (4)             Scalar or array representing the fill value. If it is of shorter
314: (4)             length than the number of fields in dt, it will be resized.
315: (4)             dt : dtype
316: (4)             The structured dtype for which to create the fill value.
317: (4)             Returns
318: (4)             -----
319: (4)             val : tuple
320: (4)             A tuple of values corresponding to the structured fill value.
321: (4)             """
322: (4)             fillvalue = np.resize(fillvalue, len(dt.names))
323: (8)             output_value = []
324: (8)             for (fval, name) in zip(fillvalue, dt.names):
325: (12)               ctype = dt[name]
326: (8)               if ctype.subdtype:
327: (12)                 ctype = ctype.subdtype[0]
328: (8)               if ctype.names is not None:
329: (12)                 output_value.append(tuple(_recursive_set_fill_value(fval,
330: (4)                               ctype)))
331: (0)             else:
332: (4)               output_value.append(np.array(fval, dtype=ctype).item())
333: (4)             return tuple(output_value)
334: (4)         def _check_fill_value(fill_value, ndtype):
335: (4)             """
336: (4)             Private function validating the given `fill_value` for the given dtype.
337: (4)             If fill_value is None, it is set to the default corresponding to the
338: (4)             dtype.
339: (4)             If fill_value is not None, its value is forced to the given dtype.
340: (4)             The result is always a 0d array.
341: (4)             """
342: (4)             ndtype = np.dtype(ndtype)
343: (4)             if fill_value is None:
344: (8)               fill_value = default_fill_value(ndtype)
345: (12)             elif ndtype.names is not None:
346: (8)               if isinstance(fill_value, (ndarray, np.void)):
347: (12)                 try:
348: (16)                   fill_value = np.array(fill_value, copy=False, dtype=ndtype)
349: (12)                 except ValueError as e:
350: (16)                   err_msg = "Unable to transform %s to dtype %s"
351: (16)                   raise ValueError(err_msg % (fill_value, ndtype)) from e

```

```

348: (8)           else:
349: (12)          fill_value = np.asarray(fill_value, dtype=object)
350: (12)          fill_value = np.array(_recursive_set_fill_value(fill_value,
351: (34)                         dtype=ndtype))
352: (4)           else:
353: (8)             if isinstance(fill_value, str) and (ndtype.char not in 'OSVU'):
354: (12)               err_msg = "Cannot set fill value of string with array of dtype %s"
355: (12)               raise TypeError(err_msg % ndtype)
356: (8)             else:
357: (12)               try:
358: (16)                 fill_value = np.array(fill_value, copy=False, dtype=ndtype)
359: (12)               except (OverflowError, ValueError) as e:
360: (16)                 err_msg = "Cannot convert fill_value %s to dtype %s"
361: (16)                 raise TypeError(err_msg % (fill_value, ndtype)) from e
362: (4)             return np.array(fill_value)
363: (0)           def set_fill_value(a, fill_value):
364: (4)             """
365: (4)               Set the filling value of a, if a is a masked array.
366: (4)               This function changes the fill value of the masked array `a` in place.
367: (4)               If `a` is not a masked array, the function returns silently, without
368: (4)               doing anything.
369: (4)               Parameters
370: (4)               -----
371: (4)                 a : array_like
372: (8)                   Input array.
373: (4)                 fill_value : dtype
374: (8)                   Filling value. A consistency test is performed to make sure
375: (8)                   the value is compatible with the dtype of `a`.
376: (4)               Returns
377: (4)               -----
378: (4)               None
379: (8)                 Nothing returned by this function.
380: (4)               See Also
381: (4)               -----
382: (4)                 maximum_fill_value : Return the default fill value for a dtype.
383: (4)                 MaskedArray.fill_value : Return current fill value.
384: (4)                 MaskedArray.set_fill_value : Equivalent method.
385: (4)               Examples
386: (4)               -----
387: (4)                 >>> import numpy.ma as ma
388: (4)                 >>> a = np.arange(5)
389: (4)                 >>> a
390: (4)                 array([0, 1, 2, 3, 4])
391: (4)                 >>> a = ma.masked_where(a < 3, a)
392: (4)                 >>> a
393: (4)                 masked_array(data=[--, --, --, 3, 4],
394: (17)                   mask=[ True,  True,  True, False, False],
395: (11)                   fill_value=999999)
396: (4)                 >>> ma.set_fill_value(a, -999)
397: (4)                 >>> a
398: (4)                 masked_array(data=[--, --, --, 3, 4],
399: (17)                   mask=[ True,  True,  True, False, False],
400: (11)                   fill_value=-999)
401: (4)                 Nothing happens if `a` is not a masked array.
402: (4)                 >>> a = list(range(5))
403: (4)                 >>> a
404: (4)                 [0, 1, 2, 3, 4]
405: (4)                 >>> ma.set_fill_value(a, 100)
406: (4)                 >>> a
407: (4)                 [0, 1, 2, 3, 4]
408: (4)                 >>> a = np.arange(5)
409: (4)                 >>> a
410: (4)                 array([0, 1, 2, 3, 4])
411: (4)                 >>> ma.set_fill_value(a, 100)
412: (4)                 >>> a
413: (4)                 array([0, 1, 2, 3, 4])
414: (4)                 """
415: (4)                 if isinstance(a, MaskedArray):

```

```

416: (8)             a.set_fill_value(fill_value)
417: (4)             return
418: (0)             def get_fill_value(a):
419: (4)                 """
420: (4)                     Return the filling value of a, if any. Otherwise, returns the
421: (4)                     default filling value for that type.
422: (4)                 """
423: (4)                 if isinstance(a, MaskedArray):
424: (8)                     result = a.fill_value
425: (4)                 else:
426: (8)                     result = default_fill_value(a)
427: (4)                 return result
428: (0)             def common_fill_value(a, b):
429: (4)                 """
430: (4)                     Return the common filling value of two masked arrays, if any.
431: (4)                     If ``a.fill_value == b.fill_value``, return the fill value,
432: (4)                     otherwise return None.
433: (4)             Parameters
434: (4)             -----
435: (4)             a, b : MaskedArray
436: (8)                 The masked arrays for which to compare fill values.
437: (4)             Returns
438: (4)             -----
439: (4)             fill_value : scalar or None
440: (8)                 The common fill value, or None.
441: (4)             Examples
442: (4)             -----
443: (4)             >>> x = np.ma.array([0, 1.], fill_value=3)
444: (4)             >>> y = np.ma.array([0, 1.], fill_value=3)
445: (4)             >>> np.ma.common_fill_value(x, y)
446: (4)                 3.0
447: (4)                 """
448: (4)             t1 = get_fill_value(a)
449: (4)             t2 = get_fill_value(b)
450: (4)             if t1 == t2:
451: (8)                 return t1
452: (4)             return None
453: (0)             def filled(a, fill_value=None):
454: (4)                 """
455: (4)                     Return input as an array with masked data replaced by a fill value.
456: (4)                     If `a` is not a `MaskedArray`, `a` itself is returned.
457: (4)                     If `a` is a `MaskedArray` and `fill_value` is None, `fill_value` is set to
458: (4)                     ``a.fill_value``.
459: (4)             Parameters
460: (4)             -----
461: (4)             a : MaskedArray or array_like
462: (8)                 An input object.
463: (4)             fill_value : array_like, optional.
464: (8)                 Can be scalar or non-scalar. If non-scalar, the
465: (8)                 resulting filled array should be broadcastable
466: (8)                 over input array. Default is None.
467: (4)             Returns
468: (4)             -----
469: (4)             a : ndarray
470: (8)                 The filled array.
471: (4)             See Also
472: (4)             -----
473: (4)             compressed
474: (4)             Examples
475: (4)             -----
476: (4)             >>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
477: (4)                               ...,
478: (4)                               ...,
479: (4)                               >>> x.filled()
480: (4)                               array([[999999,      1,      2],
481: (11)                                 [999999,      4,      5],
482: (11)                                 [      6,      7,      8]])
483: (4)                               >>> x.filled(fill_value=333)
484: (4)                               array([[333,    1,    2],

```

```

485: (11)           [333,   4,   5],
486: (11)           [ 6,   7,   8]])
487: (4)           >>> x.filled(fill_value=np.arange(3))
488: (4)           array([[0, 1, 2],
489: (11)           [ 0, 4, 5],
490: (11)           [6, 7, 8]])
491: (4)           """
492: (4)           if hasattr(a, 'filled'):
493: (8)               return a.filled(fill_value)
494: (4)           elif isinstance(a, ndarray):
495: (8)               return a
496: (4)           elif isinstance(a, dict):
497: (8)               return np.array(a, 'O')
498: (4)           else:
499: (8)               return np.array(a)
500: (0)           def get_masked_subclass(*arrays):
501: (4)           """
502: (4)           Return the youngest subclass of MaskedArray from a list of (masked)
arrays.
503: (4)           In case of siblings, the first listed takes over.
504: (4)           """
505: (4)           if len(arrays) == 1:
506: (8)               arr = arrays[0]
507: (8)               if isinstance(arr, MaskedArray):
508: (12)                   rcls = type(arr)
509: (8)               else:
510: (12)                   rcls = MaskedArray
511: (4)           else:
512: (8)               arrcls = [type(a) for a in arrays]
513: (8)               rcls = arrcls[0]
514: (8)               if not issubclass(rcls, MaskedArray):
515: (12)                   rcls = MaskedArray
516: (8)               for cls in arrcls[1:]:
517: (12)                   if issubclass(cls, rcls):
518: (16)                       rcls = cls
519: (4)               if rcls.__name__ == 'MaskedConstant':
520: (8)                   return MaskedArray
521: (4)               return rcls
522: (0)           def getdata(a, subok=True):
523: (4)           """
524: (4)           Return the data of a masked array as an ndarray.
525: (4)           Return the data of `a` (if any) as an ndarray if `a` is a ``MaskedArray``,
526: (4)           else return `a` as a ndarray or subclass (depending on `subok`) if not.
Parameters
-----
529: (4)           a : array_like
530: (8)               Input ``MaskedArray``, alternatively a ndarray or a subclass thereof.
531: (4)           subok : bool
532: (8)               Whether to force the output to be a `pure` ndarray (False) or to
533: (8)               return a subclass of ndarray if appropriate (True, default).
See Also
-----
536: (4)           getmask : Return the mask of a masked array, or nomask.
537: (4)           getmaskarray : Return the mask of a masked array, or full array of False.
Examples
-----
540: (4)           >>> import numpy.ma as ma
541: (4)           >>> a = ma.masked_equal([[1,2],[3,4]], 2)
542: (4)           >>> a
543: (4)           masked_array(
544: (6)               data=[[1, --],
545: (12)                   [3, 4]],
546: (6)               mask=[[False,  True],
547: (12)                   [False, False]],
548: (6)               fill_value=2)
549: (4)           >>> ma.getdata(a)
550: (4)           array([[1, 2],
551: (11)               [3, 4]])
552: (4)           Equivalently use the ``MaskedArray`` `data` attribute.

```

```

553: (4)             >>> a.data
554: (4)             array([[1, 2],
555: (11)            [3, 4]])
556: (4)
557: (4)             """
558: (8)             try:
559: (4)                 data = a._data
560: (8)             except AttributeError:
561: (4)                 data = np.array(a, copy=False, subok=subok)
562: (8)             if not subok:
563: (4)                 return data.view(ndarray)
564: (0)             return data
565: (0)             get_data = getdata
566: (4)             def fix_invalid(a, mask=nomask, copy=True, fill_value=None):
567: (4)                 """
568: (4)                 Return input with invalid data masked and replaced by a fill value.
569: (4)                 Invalid data means values of `nan`, `inf`, etc.
570: (4)                 Parameters
571: (4)                 -----
572: (8)                 a : array_like
573: (4)                     Input array, a (subclass of) ndarray.
574: (8)                 mask : sequence, optional
575: (4)                     Mask. Must be convertible to an array of booleans with the same
576: (4)                     shape as `data`. True indicates a masked (i.e. invalid) data.
577: (8)                 copy : bool, optional
578: (4)                     Whether to use a copy of `a` (True) or to fix `a` in place (False).
579: (4)                     Default is True.
580: (8)                 fill_value : scalar, optional
581: (4)                     Value used for fixing invalid data. Default is None, in which case
582: (4)                     the ``a.fill_value`` is used.
583: (4)             Returns
584: (4)             -----
585: (8)             b : MaskedArray
586: (4)                 The input array with invalid entries fixed.
587: (4)             Notes
588: (4)             -----
589: (4)             A copy is performed by default.
590: (4)             Examples
591: (4)             -----
592: (4)             >>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
593: (4)             >>> x
594: (17)            masked_array(data=[--, -1.0, nan, inf],
595: (4)                         mask=[ True, False, False, False],
596: (11)                         fill_value=1e+20)
597: (4)             >>> np.ma.fix_invalid(x)
598: (17)            masked_array(data=[--, -1.0, --, --],
599: (4)                         mask=[ True, False, True, True],
600: (11)                         fill_value=1e+20)
601: (4)             >>> fixed = np.ma.fix_invalid(x)
602: (4)             >>> fixed.data
603: (4)                 array([ 1.e+00, -1.e+00,  1.e+20,  1.e+20])
604: (4)             >>> x.data
605: (4)                 array([ 1., -1., nan, inf])
606: (4)             """
607: (4)             a = masked_array(a, copy=copy, mask=mask, subok=True)
608: (4)             invalid = np.logical_not(np.isfinite(a._data))
609: (4)             if not invalid.any():
610: (8)                 return a
611: (4)             a._mask |= invalid
612: (4)             if fill_value is None:
613: (8)                 fill_value = a.fill_value
614: (4)             a._data[invalid] = fill_value
615: (0)             return a
616: (4)             def is_string_or_list_of_strings(val):
617: (12)                 return (isinstance(val, str) or
618: (13)                     (isinstance(val, list) and val and
619: (0)                         builtins.all(isinstance(s, str) for s in val)))
620: (0)             ufunc_domain = {}
621: (0)             ufunc_fills = {}
622: (0)             class _DomainCheckInterval:

```

```

622: (4)
623: (4)
624: (4)
625: (4)
626: (4)
627: (4)
628: (8)
629: (8)
630: (12)
631: (8)
632: (8)
633: (4)
634: (8)
635: (8)
636: (12)
637: (36)
638: (0)
639: (4)
640: (4)
641: (4)
642: (4)
643: (4)
644: (8)
645: (8)
646: (4)
647: (8)
648: (8)
649: (12)
650: (0)
651: (4)
652: (4)
653: (4)
654: (4)
655: (8)
656: (4)
657: (8)
658: (12)
659: (8)
660: (8)
661: (12)
662: (0)
663: (4)
664: (4)
665: (4)
666: (4)
667: (8)
668: (8)
669: (4)
670: (8)
671: (8)
672: (12)
673: (0)
674: (4)
675: (4)
676: (4)
677: (4)
678: (8)
679: (8)
680: (4)
681: (8)
682: (8)
683: (12)
684: (0)
685: (4)
686: (8)
687: (8)
688: (8)
689: (4)
690: (8)

        """
        Define a valid interval, so that :
        ``domain_check_interval(a,b)(x) == True`` where
        ``x < a`` or ``x > b``.
        """
        def __init__(self, a, b):
            "domain_check_interval(a,b)(x) = true where x < a or y > b"
            if a > b:
                (a, b) = (b, a)
            self.a = a
            self.b = b
        def __call__(self, x):
            "Execute the call behavior."
            with np.errstate(invalid='ignore'):
                return umath.logical_or(umath.greater(x, self.b),
                                        umath.less(x, self.a))
    class _DomainTan:
        """
        Define a valid interval for the `tan` function, so that:
        ``domain_tan(eps) = True`` where ``abs(cos(x)) < eps``
        """
        def __init__(self, eps):
            "domain_tan(eps) = true where abs(cos(x)) < eps"
            self.eps = eps
        def __call__(self, x):
            "Executes the call behavior."
            with np.errstate(invalid='ignore'):
                return umath.less(umath.absolute(umath.cos(x)), self.eps)
    class _DomainSafeDivide:
        """
        Define a domain for safe division.
        """
        def __init__(self, tolerance=None):
            self.tolerance = tolerance
        def __call__(self, a, b):
            if self.tolerance is None:
                self.tolerance = np.finfo(float).tiny
            a, b = np.asarray(a), np.asarray(b)
            with np.errstate(invalid='ignore'):
                return umath.absolute(a) * self.tolerance >= umath.absolute(b)
    class _DomainGreater:
        """
        DomainGreater(v)(x) is True where x <= v.
        """
        def __init__(self, critical_value):
            "DomainGreater(v)(x) = true where x <= v"
            self.critical_value = critical_value
        def __call__(self, x):
            "Executes the call behavior."
            with np.errstate(invalid='ignore'):
                return umath.less_equal(x, self.critical_value)
    class _DomainGreaterEqual:
        """
        DomainGreaterEqual(v)(x) is True where x < v.
        """
        def __init__(self, critical_value):
            "DomainGreaterEqual(v)(x) = true where x < v"
            self.critical_value = critical_value
        def __call__(self, x):
            "Executes the call behavior."
            with np.errstate(invalid='ignore'):
                return umath.less(x, self.critical_value)
    class _MaskedUFunc:
        def __init__(self, ufunc):
            self.f = ufunc
            self.__doc__ = ufunc.__doc__
            self.__name__ = ufunc.__name__
        def __str__(self):
            return f"Masked version of {self.f}"

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

691: (0)
692: (4)
693: (4)
694: (4)
695: (4)
696: (4)
697: (4)
698: (8)
699: (8)
700: (4)
701: (8)
702: (4)
703: (8)
704: (8)
705: (4)
706: (4)
707: (8)
708: (8)
709: (8)
710: (8)
711: (8)
712: (4)
713: (8)
714: (8)
715: (8)
716: (8)
717: (8)
718: (12)
719: (16)
720: (12)
721: (12)
722: (12)
723: (8)
724: (12)
725: (16)
726: (12)
727: (8)
728: (12)
729: (16)
730: (12)
731: (8)
732: (12)
733: (16)
734: (12)
735: (16)
736: (8)
737: (8)
738: (8)
739: (8)
740: (0)
741: (4)
742: (4)
743: (4)
744: (4)
745: (4)
746: (4)
747: (8)
748: (8)
749: (4)
750: (8)
751: (8)
752: (4)
753: (8)
754: (4)
755: (8)
756: (4)
757: (4)
758: (8)
759: (8)

class _MaskedUnaryOperation(_MaskedUFunc):
    """
        Defines masked version of unary operations, where invalid values are
        pre-masked.
        Parameters
        -----
        mufunc : callable
            The function for which to define a masked version. Made available
            as ``_MaskedUnaryOperation.f``.
        fill : scalar, optional
            Filling value, default is 0.
        domain : class instance
            Domain for the function. Should be one of the ``_Domain*``
            classes. Default is None.
    """
    def __init__(self, mufunc, fill=0, domain=None):
        super().__init__(mufunc)
        self.fill = fill
        self.domain = domain
        ufunc_domain[mufunc] = domain
        ufunc_fills[mufunc] = fill
    def __call__(self, a, *args, **kwargs):
        """
            Execute the call behavior.
        """
        d = getdata(a)
        if self.domain is not None:
            with np.errstate(divide='ignore', invalid='ignore'):
                result = self.f(d, *args, **kwargs)
            m = ~umath.isfinite(result)
            m |= self.domain(d)
            m |= getmask(a)
        else:
            with np.errstate(divide='ignore', invalid='ignore'):
                result = self.f(d, *args, **kwargs)
            m = getmask(a)
        if not result.ndim:
            if m:
                return masked
            return result
        if m is not nomask:
            try:
                np.copyto(result, d, where=m)
            except TypeError:
                pass
        masked_result = result.view(get_masked_subclass(a))
        masked_result._mask = m
        masked_result._update_from(a)
        return masked_result

class _MaskedBinaryOperation(_MaskedUFunc):
    """
        Define masked version of binary operations, where invalid
        values are pre-masked.
        Parameters
        -----
        mbfunc : function
            The function for which to define a masked version. Made available
            as ``_MaskedBinaryOperation.f``.
        domain : class instance
            Default domain for the function. Should be one of the ``_Domain*``
            classes. Default is None.
        fillx : scalar, optional
            Filling value for the first argument, default is 0.
        filly : scalar, optional
            Filling value for the second argument, default is 0.
    """
    def __init__(self, mbfunc, fillx=0, filly=0):
        """
            abfunc(fillx, filly) must be defined.
        """

```

```

760: (8)           abfunc(x, filly) = x for all x to enable reduce.
761: (8)
762: (8)           """
763: (8)           super().__init__(mbfunc)
764: (8)           self.fillx = fillx
765: (8)           self.filly = filly
766: (8)           ufunc_domain[mbfunc] = None
767: (4)           ufunc_fills[mbfunc] = (fillx, filly)
768: (8)           def __call__(self, a, b, *args, **kwargs):
769: (8)               """
770: (8)               Execute the call behavior.
771: (8)               """
772: (8)               (da, db) = (getdata(a), getdata(b))
773: (12)              with np.errstate():
774: (12)                  np.seterr(divide='ignore', invalid='ignore')
775: (8)                  result = self.f(da, db, *args, **kwargs)
776: (8)                  (ma, mb) = (getmask(a), getmask(b))
777: (12)                  if ma is nomask:
778: (16)                      if mb is nomask:
779: (12)                          m = nomask
780: (16)                      else:
781: (8)                          m = umath.logical_or(getmaskarray(a), mb)
782: (12)                  elif mb is nomask:
783: (8)                      m = umath.logical_or(ma, getmaskarray(b))
784: (12)                  else:
785: (8)                      m = umath.logical_or(ma, mb)
786: (12)                  if not result.ndim:
787: (16)                      if m:
788: (12)                          return masked
789: (8)                      return result
790: (12)                  if m is not nomask and m.any():
791: (16)                      try:
792: (12)                          np.copyto(result, da, casting='unsafe', where=m)
793: (16)                      except Exception:
794: (8)                          pass
795: (8)                      masked_result = result.view(get_masked_subclass(a, b))
796: (8)                      masked_result._mask = m
797: (12)                      if isinstance(a, MaskedArray):
798: (8)                          masked_result._update_from(a)
799: (12)                      elif isinstance(b, MaskedArray):
800: (8)                          masked_result._update_from(b)
801: (4)                      return masked_result
802: (8)
803: (8)          def reduce(self, target, axis=0, dtype=None):
804: (8)              """
805: (8)              Reduce `target` along the given `axis`.
806: (8)              """
807: (8)              tclass = get_masked_subclass(target)
808: (8)              m = getmask(target)
809: (12)              t = filled(target, self.filly)
810: (12)              if t.shape == ():
811: (16)                  t = t.reshape(1)
812: (16)                  if m is not nomask:
813: (8)                      m = make_mask(m, copy=True)
814: (12)                      m.shape = (1,)
815: (12)                  if m is nomask:
816: (8)                      tr = self.f.reduce(t, axis)
817: (12)                      mr = nomask
818: (12)                  else:
819: (8)                      tr = self.f.reduce(t, axis, dtype=dtype)
820: (12)                      mr = umath.logical_and.reduce(m, axis)
821: (16)                  if not tr.shape:
822: (12)                      if mr:
823: (16)                          return masked
824: (8)                      return tr
825: (8)                  masked_tr = tr.view(tclass)
826: (8)                  masked_tr._mask = mr
827: (4)                  return masked_tr
828: (8)          def outer(self, a, b):
829: (8)

```

```

829: (8)             Return the function applied to the outer product of a and b.
830: (8)
831: (8)             """
832: (8)             (da, db) = (getdata(a), getdata(b))
833: (8)             d = self.f.outer(da, db)
834: (8)             ma = getmask(a)
835: (8)             mb = getmask(b)
836: (12)            if ma is nomask and mb is nomask:
837: (8)                m = nomask
838: (12)            else:
839: (12)                ma = getmaskarray(a)
840: (12)                mb = getmaskarray(b)
841: (12)                m = umath.logical_or.outer(ma, mb)
842: (8)                if (not m.ndim) and m:
843: (8)                    return masked
844: (12)                if m is not nomask:
845: (8)                    np.copyto(d, da, where=m)
846: (8)                if not d.shape:
847: (12)                    return d
848: (8)                masked_d = d.view(get_masked_subclass(a, b))
849: (8)                masked_d._mask = m
850: (8)                return masked_d
851: (4)            def accumulate(self, target, axis=0):
852: (8)                """Accumulate `target` along `axis` after filling with y fill
853: (8)                value.
854: (8)                """
855: (8)                tclass = get_masked_subclass(target)
856: (8)                t = filled(target, self.filly)
857: (8)                result = self.f.accumulate(t, axis)
858: (8)                masked_result = result.view(tclass)
859: (8)                return masked_result
860: (0)            class _DomainBinaryOperation(_MaskedUFunc):
861: (4)                """
862: (4)                    Define binary operations that have a domain, like divide.
863: (4)                    They have no reduce, outer or accumulate.
864: (4)                    Parameters
865: (4)                    -----
866: (4)                    mbfunc : function
867: (8)                    The function for which to define a masked version. Made available
868: (8)                    as ``_DomainBinaryOperation.f``.
869: (8)                    domain : class instance
870: (8)                    Default domain for the function. Should be one of the ``_Domain*``
871: (8)                    classes.
872: (8)                    fillx : scalar, optional
873: (8)                    Filling value for the first argument, default is 0.
874: (8)                    filly : scalar, optional
875: (8)                    Filling value for the second argument, default is 0.
876: (4)                def __init__(self, dbfunc, domain, fillx=0, filly=0):
877: (8)                    """abfunc(fillx, filly) must be defined.
878: (11)                    abfunc(x, filly) = x for all x to enable reduce.
879: (8)                    """
880: (8)                    super().__init__(dbfunc)
881: (8)                    self.domain = domain
882: (8)                    self.fillx = fillx
883: (8)                    self.filly = filly
884: (8)                    ufunc_domain[dbfunc] = domain
885: (8)                    ufunc_fills[dbfunc] = (fillx, filly)
886: (4)                def __call__(self, a, b, *args, **kwargs):
887: (8)                    "Execute the call behavior."
888: (8)                    (da, db) = (getdata(a), getdata(b))
889: (8)                    with np.errstate(divide='ignore', invalid='ignore'):
890: (12)                        result = self.f(da, db, *args, **kwargs)
891: (8)                        m = ~umath.isfinite(result)
892: (8)                        m |= getmask(a)
893: (8)                        m |= getmask(b)
894: (8)                        domain = ufunc_domain.get(self.f, None)
895: (8)                        if domain is not None:
896: (12)                            m |= domain(da, db)
897: (8)                        if not m.ndim:

```

```

898: (12)           if m:
899: (16)             return masked
900: (12)           else:
901: (16)             return result
902: (8)           try:
903: (12)             np.copyto(result, 0, casting='unsafe', where=m)
904: (12)             masked_da = umath.multiply(m, da)
905: (12)             if np.can_cast(masked_da.dtype, result.dtype, casting='safe'):
906: (16)               result += masked_da
907: (8)           except Exception:
908: (12)             pass
909: (8)           masked_result = result.view(get_masked_subclass(a, b))
910: (8)           masked_result._mask = m
911: (8)           if isinstance(a, MaskedArray):
912: (12)             masked_result._update_from(a)
913: (8)           elif isinstance(b, MaskedArray):
914: (12)             masked_result._update_from(b)
915: (8)           return masked_result
916: (0)           exp = _MaskedUnaryOperation(umath.exp)
917: (0)           conjugate = _MaskedUnaryOperation(umath.conjugate)
918: (0)           sin = _MaskedUnaryOperation(umath.sin)
919: (0)           cos = _MaskedUnaryOperation(umath.cos)
920: (0)           arctan = _MaskedUnaryOperation(umath.arctan)
921: (0)           arcsinh = _MaskedUnaryOperation(umath.arcsinh)
922: (0)           sinh = _MaskedUnaryOperation(umath.sinh)
923: (0)           cosh = _MaskedUnaryOperation(umath.cosh)
924: (0)           tanh = _MaskedUnaryOperation(umath.tanh)
925: (0)           abs = absolute = _MaskedUnaryOperation(umath.absolute)
926: (0)           angle = _MaskedUnaryOperation(angle) # from numpy.lib.function_base
927: (0)           fabs = _MaskedUnaryOperation(umathfabs)
928: (0)           negative = _MaskedUnaryOperation(umath.negative)
929: (0)           floor = _MaskedUnaryOperation(umath.floor)
930: (0)           ceil = _MaskedUnaryOperation(umath.ceil)
931: (0)           around = _MaskedUnaryOperation(np.round_)
932: (0)           logical_not = _MaskedUnaryOperation(umath.logical_not)
933: (0)           sqrt = _MaskedUnaryOperation(umath.sqrt, 0.0,
934: (29)                         _DomainGreaterEqual(0.0))
935: (0)           log = _MaskedUnaryOperation(umath.log, 1.0,
936: (28)                         _DomainGreater(0.0))
937: (0)           log2 = _MaskedUnaryOperation(umath.log2, 1.0,
938: (29)                         _DomainGreater(0.0))
939: (0)           log10 = _MaskedUnaryOperation(umath.log10, 1.0,
940: (30)                         _DomainGreater(0.0))
941: (0)           tan = _MaskedUnaryOperation(umath.tan, 0.0,
942: (28)                         _DomainTan(1e-35))
943: (0)           arcsin = _MaskedUnaryOperation(umath.arcsin, 0.0,
944: (31)                         _DomainCheckInterval(-1.0, 1.0))
945: (0)           arccos = _MaskedUnaryOperation(umath.arccos, 0.0,
946: (31)                         _DomainCheckInterval(-1.0, 1.0))
947: (0)           arccosh = _MaskedUnaryOperation(umath.arccosh, 1.0,
948: (32)                         _DomainGreaterEqual(1.0))
949: (0)           arctanh = _MaskedUnaryOperation(umath.arctanh, 0.0,
950: (32)                         _DomainCheckInterval(-1.0 + 1e-15, 1.0 - 1e-
15))
951: (0)           add = _MaskedBinaryOperation(umath.add)
952: (0)           subtract = _MaskedBinaryOperation(umath.subtract)
953: (0)           multiply = _MaskedBinaryOperation(umath.multiply, 1, 1)
954: (0)           arctan2 = _MaskedBinaryOperation(umath.arctan2, 0.0, 1.0)
955: (0)           equal = _MaskedBinaryOperation(umath.equal)
956: (0)           equal.reduce = None
957: (0)           not_equal = _MaskedBinaryOperation(umath.not_equal)
958: (0)           not_equal.reduce = None
959: (0)           less_equal = _MaskedBinaryOperation(umath.less_equal)
960: (0)           less_equal.reduce = None
961: (0)           greater_equal = _MaskedBinaryOperation(umath.greater_equal)
962: (0)           greater_equal.reduce = None
963: (0)           less = _MaskedBinaryOperation(umath.less)
964: (0)           less.reduce = None
965: (0)           greater = _MaskedBinaryOperation(umath.greater)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

966: (0) greater.reduce = None
967: (0) logical_and = _MaskedBinaryOperation(umath.logical_and)
968: (0) alltrue = _MaskedBinaryOperation(umath.logical_and, 1, 1).reduce
969: (0) logical_or = _MaskedBinaryOperation(umath.logical_or)
970: (0) sometrue = logical_or.reduce
971: (0) logical_xor = _MaskedBinaryOperation(umath.logical_xor)
972: (0) bitwise_and = _MaskedBinaryOperation(umath.bitwise_and)
973: (0) bitwise_or = _MaskedBinaryOperation(umath.bitwise_or)
974: (0) bitwise_xor = _MaskedBinaryOperation(umath.bitwise_xor)
975: (0) hypot = _MaskedBinaryOperation(umath.hypot)
976: (0) divide = _DomainBinaryOperation(umath.divide, _DomainSafeDivide(), 0, 1)
977: (0) true_divide = _DomainBinaryOperation(umath.true_divide,
978: (39)                                     _DomainSafeDivide(), 0, 1)
979: (0) floor_divide = _DomainBinaryOperation(umath.floor_divide,
980: (40)                                     _DomainSafeDivide(), 0, 1)
981: (0) remainder = _DomainBinaryOperation(umath.remainder,
982: (37)                                     _DomainSafeDivide(), 0, 1)
983: (0) fmod = _DomainBinaryOperation(umath.fmod, _DomainSafeDivide(), 0, 1)
984: (0) mod = _DomainBinaryOperation(umath.mod, _DomainSafeDivide(), 0, 1)
985: (0) def _replace_dtype_fields_recursive(dtype, primitive_dtype):
986: (4)     """Private function allowing recursion in _replace_dtype_fields."""
987: (4)     _recurve = _replace_dtype_fields_recursive
988: (4)     if dtype.names is not None:
989: (8)         descr = []
990: (8)         for name in dtype.names:
991: (12)             field = dtype.fields[name]
992: (12)             if len(field) == 3:
993: (16)                 name = (field[-1], name)
994: (12)                 descr.append((name, _recurve(field[0], primitive_dtype)))
995: (8)             new_dtype = np.dtype(descr)
996: (4)         elif dtype.subdtype:
997: (8)             descr = list(dtype.subdtype)
998: (8)             descr[0] = _recurve(dtype.subdtype[0], primitive_dtype)
999: (8)             new_dtype = np.dtype(tuple(descr))
1000: (4)     else:
1001: (8)         new_dtype = primitive_dtype
1002: (4)     if new_dtype == dtype:
1003: (8)         new_dtype = dtype
1004: (4)     return new_dtype
1005: (0) def _replace_dtype_fields(dtype, primitive_dtype):
1006: (4)     """
1007: (4)     Construct a dtype description list from a given dtype.
1008: (4)     Returns a new dtype object, with all fields and subtypes in the given type
1009: (4)     recursively replaced with `primitive_dtype`.
1010: (4)     Arguments are coerced to dtypes first.
1011: (4)     """
1012: (4)     dtype = np.dtype(dtype)
1013: (4)     primitive_dtype = np.dtype(primitive_dtype)
1014: (4)     return _replace_dtype_fields_recursive(dtype, primitive_dtype)
1015: (0) def make_mask_descr(ndtype):
1016: (4)     """
1017: (4)     Construct a dtype description list from a given dtype.
1018: (4)     Returns a new dtype object, with the type of all fields in `ndtype` to a
1019: (4)     boolean type. Field names are not altered.
1020: (4)     Parameters
1021: (4)     -----
1022: (4)     ndtype : dtype
1023: (8)         The dtype to convert.
1024: (4)     Returns
1025: (4)     -----
1026: (4)     result : dtype
1027: (8)         A dtype that looks like `ndtype`, the type of all fields is boolean.
1028: (4)     Examples
1029: (4)     -----
1030: (4)     >>> import numpy.ma as ma
1031: (4)     >>> dtype = np.dtype({'names':['foo', 'bar'],
1032: (4)                           'formats':[np.float32, np.int64]})
1033: (4)     >>> dtype
1034: (4)     dtype([('foo', '<f4'), ('bar', '<i8')])
```

```

1035: (4)          >>> ma.make_mask_descr(dtype)
1036: (4)          dtype([('foo', '|b1'), ('bar', '|b1')])
1037: (4)          >>> ma.make_mask_descr(np.float32)
1038: (4)          dtype('bool')
1039: (4)          """
1040: (4)          return _replace_dtype_fields(ndtype, MaskType)
1041: (0)      def getmask(a):
1042: (4)          """
1043: (4)          Return the mask of a masked array, or nomask.
1044: (4)          Return the mask of `a` as an ndarray if `a` is a `MaskedArray` and the
1045: (4)          mask is not `nomask`, else return `nomask`. To guarantee a full array
1046: (4)          of booleans of the same shape as a, use `getmaskarray`.
1047: (4)          Parameters
1048: (4)          -----
1049: (4)          a : array_like
1050: (8)              Input `MaskedArray` for which the mask is required.
1051: (4)          See Also
1052: (4)          -----
1053: (4)          getdata : Return the data of a masked array as an ndarray.
1054: (4)          getmaskarray : Return the mask of a masked array, or full array of False.
1055: (4)          Examples
1056: (4)          -----
1057: (4)          >>> import numpy.ma as ma
1058: (4)          >>> a = ma.masked_equal([[1,2],[3,4]], 2)
1059: (4)          >>> a
1060: (4)          masked_array(
1061: (6)              data=[[1, --],
1062: (12)                  [3, 4]],
1063: (6)              mask=[[False, True],
1064: (12)                  [False, False]],
1065: (6)              fill_value=2)
1066: (4)          >>> ma.getmask(a)
1067: (4)          array([[False, True],
1068: (11)                  [False, False]])
1069: (4)          Equivalently use the `MaskedArray` `mask` attribute.
1070: (4)          >>> a.mask
1071: (4)          array([[False, True],
1072: (11)                  [False, False]])
1073: (4)          Result when mask == `nomask`
1074: (4)          >>> b = ma.masked_array([[1,2],[3,4]])
1075: (4)          >>> b
1076: (4)          masked_array(
1077: (6)              data=[[1, 2],
1078: (12)                  [3, 4]],
1079: (6)              mask=False,
1080: (6)              fill_value=999999)
1081: (4)          >>> ma.nomask
1082: (4)          False
1083: (4)          >>> ma.getmask(b) == ma.nomask
1084: (4)          True
1085: (4)          >>> b.mask == ma.nomask
1086: (4)          True
1087: (4)          """
1088: (4)          return getattr(a, '_mask', nomask)
1089: (0)      get_mask = getmask
1090: (0)      def getmaskarray(arr):
1091: (4)          """
1092: (4)          Return the mask of a masked array, or full boolean array of False.
1093: (4)          Return the mask of `arr` as an ndarray if `arr` is a `MaskedArray` and the
1094: (4)          mask is not `nomask`, else return a full boolean array of False of
1095: (4)          the same shape as `arr`.
1096: (4)          Parameters
1097: (4)          -----
1098: (4)          arr : array_like
1099: (8)              Input `MaskedArray` for which the mask is required.
1100: (4)          See Also
1101: (4)          -----
1102: (4)          getmask : Return the mask of a masked array, or nomask.
1103: (4)          getdata : Return the data of a masked array as an ndarray.

```

```

1104: (4) Examples
1105: (4)
1106: (4)
1107: (4)
1108: (4)
1109: (4)
1110: (6)
1111: (12)
1112: (6)
1113: (12)
1114: (6)
1115: (4)
1116: (4)
1117: (11)
1118: (4)
1119: (4)
1120: (4)
1121: (4)
1122: (6)
1123: (12)
1124: (6)
1125: (6)
1126: (4)
1127: (4)
1128: (11)
1129: (4)
1130: (4)
1131: (4)
1132: (8)
1133: (4)
1134: (0)
1135: (4)
1136: (4) def is_mask(m):
1137: (4)     """Return True if m is a valid, standard mask.
1138: (4)     This function does not check the contents of the input, only that the
1139: (4)     type is MaskType. In particular, this function returns False if the
1140: (4)     mask has a flexible dtype.
1141: (4)     Parameters
1142: (4)     -----
1143: (8)         m : array_like
1144: (4)             Array to test.
1145: (4)     Returns
1146: (4)     -----
1147: (8)         result : bool
1148: (4)             True if `m.dtype.type` is MaskType, False otherwise.
1149: (4)     See Also
1150: (4)     -----
1151: (4)         ma.isMaskedArray : Test whether input is an instance of MaskedArray.
1152: (4)     Examples
1153: (4)
1154: (4)
1155: (4)
1156: (4)
1157: (17)
1158: (11)
1159: (4)
1160: (4)
1161: (4)
1162: (4)
1163: (4)
1164: (4)
1165: (4)
1166: (4)
1167: (4)
1168: (4)
1169: (4)
1170: (4)
1171: (4)
1172: (4)

-----
```

>>> import numpy.ma as ma

>>> a = ma.masked\_equal([[1,2],[3,4]], 2)

>>> a

masked\_array(

  data=[[1, --],

        [3, 4]],

  mask=[[False, True],

        [False, False]],

  fill\_value=2)

>>> ma.getmaskarray(a)

array([[False, True],

      [False, False]])

Result when mask == ``nomask``

>>> b = ma.masked\_array([[1,2],[3,4]])

>>> b

masked\_array(

  data=[[1, 2],

        [3, 4]],

  mask=False,

  fill\_value=999999)

>>> ma.getmaskarray(b)

array([[False, False],

      [False, False]])

"""

mask = getmask(arr)

if mask is nomask:

    mask = make\_mask\_none(np.shape(arr), getattr(arr, 'dtype', None))

return mask

def is\_mask(m):

    """

    Return True if m is a valid, standard mask.

This function does not check the contents of the input, only that the

type is MaskType. In particular, this function returns False if the

mask has a flexible dtype.

Parameters

-----

m : array\_like

    Array to test.

Returns

-----

result : bool

    True if `m.dtype.type` is MaskType, False otherwise.

See Also

-----

ma.isMaskedArray : Test whether input is an instance of MaskedArray.

Examples

-----

>>> import numpy.ma as ma

>>> m = ma.masked\_equal([0, 1, 0, 2, 3], 0)

>>> m

masked\_array(data=[--, 1, --, 2, 3],

                  mask=[ True, False, True, False, False],

                  fill\_value=0)

>>> ma.is\_mask(m)

False

>>> ma.is\_mask(m.mask)

True

Input must be an ndarray (or have similar attributes)

for it to be considered a valid mask.

>>> m = [False, True, False]

>>> ma.is\_mask(m)

False

>>> m = np.array([False, True, False])

>>> m

array([False, True, False])

>>> ma.is\_mask(m)

True

```

1173: (4)          Arrays with complex dtypes don't return True.
1174: (4)          >>> dtype = np.dtype({'names':['monty', 'python'],
1175: (4)                      ...
1176: (4)                      'formats':[bool, bool]})  

1177: (4)          >>> dtype
1178: (4)          dtype([('monty', '|b1'), ('python', '|b1')])  

1179: (4)          >>> m = np.array([(True, False), (False, True), (True, False)],
1180: (4)                      ...
1181: (4)                      dtype=dtype)  

1182: (4)          >>> m
1183: (4)          array([( True, False), (False, True), ( True, False)],
1184: (4)                      ...
1185: (4)                      dtype=[('monty', '?'), ('python', '?')])  

1186: (4)          >>> ma.is_mask(m)
1187: (4)          False
1188: (4)          """  

1189: (4)          try:  

1190: (0)              return m.dtype.type is MaskType  

1191: (4)          except AttributeError:  

1192: (4)              return False
1193: (4)          def _shrink_mask(m):
1194: (4)              """  

1195: (4)              Shrink a mask to nomask if possible
1196: (4)              """  

1197: (8)              if m.dtype.names is None and not m.any():
1198: (0)                  return nomask
1199: (4)              else:
1200: (4)                  return m
1201: (4)          def make_mask(m, copy=False, shrink=True, dtype=MaskType):
1202: (4)              """  

1203: (4)              Create a boolean mask from an array.  

1204: (4)              Return `m` as a boolean mask, creating a copy if necessary or requested.  

1205: (4)              The function can accept any sequence that is convertible to integers,  

1206: (4)              or ``nomask``. Does not require that contents must be 0s and 1s, values  

1207: (4)              of 0 are interpreted as False, everything else as True.  

1208: (4)              Parameters
1209: (4)              -----
1210: (8)              m : array_like
1211: (4)                  Potential mask.
1212: (4)              copy : bool, optional
1213: (8)                  Whether to return a copy of `m` (True) or `m` itself (False).
1214: (4)              shrink : bool, optional
1215: (8)                  Whether to shrink `m` to ``nomask`` if all its values are False.
1216: (4)              dtype : dtype, optional
1217: (8)                  Data-type of the output mask. By default, the output mask has a
1218: (4)                  dtype of MaskType (bool). If the dtype is flexible, each field has
1219: (4)                  a boolean dtype. This is ignored when `m` is ``nomask``, in which
1220: (4)                  case ``nomask`` is always returned.
1221: (4)          Returns
1222: (4)          -----
1223: (4)          result : ndarray
1224: (4)              A boolean mask derived from `m`.
1225: (4)          Examples
1226: (4)          -----
1227: (4)          >>> import numpy.ma as ma
1228: (4)          >>> m = [True, False, True, True]
1229: (4)          >>> ma.make_mask(m)
1230: (4)          array([ True, False,  True,  True])
1231: (4)          >>> m = [1, 0, 1, 1]
1232: (4)          >>> ma.make_mask(m)
1233: (4)          array([ True, False,  True,  True])
1234: (4)          Effect of the `shrink` parameter.
1235: (4)          >>> m = np.zeros(4)
1236: (4)          >>> m
1237: (4)          array([0., 0., 0., 0.])
1238: (4)          >>> ma.make_mask(m)
1239: (4)          False
1240: (4)          >>> ma.make_mask(m, shrink=False)
1241: (4)          array([False, False, False, False])

```

```

1242: (4)             Using a flexible `dtype`.
1243: (4)             >>> m = [1, 0, 1, 1]
1244: (4)             >>> n = [0, 1, 0, 0]
1245: (4)             >>> arr = []
1246: (4)             >>> for man, mouse in zip(m, n):
1247: (4)                 ...     arr.append((man, mouse))
1248: (4)             >>> arr
1249: (4)             [(1, 0), (0, 1), (1, 0), (1, 0)]
1250: (4)             >>> dtype = np.dtype({'names':['man', 'mouse'],
1251: (4)                           'formats':[np.int64, np.int64]}) 
1252: (4)             >>> arr = np.array(arr, dtype=dtype)
1253: (4)             >>> arr
1254: (4)             array([(1, 0), (0, 1), (1, 0), (1, 0)],
1255: (10)                dtype=[('man', '<i8'), ('mouse', '<i8')])
1256: (4)             >>> ma.make_mask(arr, dtype=dtype)
1257: (4)             array([(True, False), (False, True), (True, False), (True, False)],
1258: (10)                dtype=[('man', '|b1'), ('mouse', '|b1')])
1259: (4)
1260: (4)
1261: (8)             if m is nomask:
1262: (4)                 return nomask
1263: (4)             dtype = make_mask_descr(dtype)
1264: (8)             if isinstance(m, ndarray) and m.dtype.fields and dtype == np.bool_:
1265: (4)                 return np.ones(m.shape, dtype=dtype)
1266: (4)             result = np.array(filled(m, True), copy=copy, dtype=dtype, subok=True)
1267: (8)             if shrink:
1268: (4)                 result = _shrink_mask(result)
1269: (0)             return result
1270: (4)
1271: (4)             def make_mask_none(newshape, dtype=None):
1272: (4)                 """
1273: (4)                 Return a boolean mask of the given shape, filled with False.
1274: (4)                 This function returns a boolean ndarray with all entries False, that can
1275: (4)                 be used in common mask manipulations. If a complex dtype is specified, the
1276: (4)                 type of each field is converted to a boolean type.
1277: (4)                 Parameters
1278: (4)                     newshape : tuple
1279: (4)                         A tuple indicating the shape of the mask.
1280: (8)                     dtype : {None, dtype}, optional
1281: (8)                         If None, use a MaskType instance. Otherwise, use a new datatype with
1282: (4)                         the same fields as `dtype`, converted to boolean types.
1283: (4)             Returns
1284: (4)                     result : ndarray
1285: (8)                         An ndarray of appropriate shape and dtype, filled with False.
1286: (4)             See Also
1287: (4)
1288: (4)                     make_mask : Create a boolean mask from an array.
1289: (4)                     make_mask_descr : Construct a dtype description list from a given dtype.
1290: (4)
1291: (4)
1292: (4)                     >>> import numpy.ma as ma
1293: (4)                     >>> ma.make_mask_none((3,))
1294: (4)                     array([False, False, False])
1295: (4)                     Defining a more complex dtype.
1296: (4)                     >>> dtype = np.dtype({'names':['foo', 'bar'],
1297: (4)                                   'formats':[np.float32, np.int64]}) 
1298: (4)
1299: (4)                     >>> dtype
1300: (4)                     dtype([('foo', '<f4'), ('bar', '<i8')])
1301: (4)                     >>> ma.make_mask_none((3,), dtype=dtype)
1302: (4)                     array([(False, False), (False, False), (False, False)],
1303: (10)                       dtype=[('foo', '|b1'), ('bar', '|b1')])
1304: (4)
1305: (8)                     if dtype is None:
1306: (4)                         result = np.zeros(newshape, dtype=MaskType)
1307: (4)                     else:
1308: (4)                         result = np.zeros(newshape, dtype=make_mask_descr(dtype))
1309: (0)             return result
1310: (4)
1311: (4)             def _recursive_mask_or(m1, m2, newmask):
1312: (4)                 names = m1.dtype.names

```

```

1311: (4)             for name in names:
1312: (8)                 current1 = m1[name]
1313: (8)                 if current1.dtype.names is not None:
1314: (12)                     _recursive_mask_or(current1, m2[name], newmask[name])
1315: (8)                 else:
1316: (12)                     umath.logical_or(current1, m2[name], newmask[name])
1317: (0)             def mask_or(m1, m2, copy=False, shrink=True):
1318: (4)                 """
1319: (4)                     Combine two masks with the ``logical_or`` operator.
1320: (4)                     The result may be a view on `m1` or `m2` if the other is `nomask`
1321: (4)                     (i.e. False).
1322: (4)             Parameters
1323: (4)             -----
1324: (4)                 m1, m2 : array_like
1325: (8)                     Input masks.
1326: (4)                 copy : bool, optional
1327: (8)                     If copy is False and one of the inputs is `nomask`, return a view
1328: (8)                     of the other input mask. Defaults to False.
1329: (4)                 shrink : bool, optional
1330: (8)                     Whether to shrink the output to `nomask` if all its values are
1331: (8)                     False. Defaults to True.
1332: (4)             Returns
1333: (4)             -----
1334: (4)                 mask : output mask
1335: (8)                     The result masks values that are masked in either `m1` or `m2`.
1336: (4)             Raises
1337: (4)             -----
1338: (4)             ValueError
1339: (8)                     If `m1` and `m2` have different flexible dtypes.
1340: (4)             Examples
1341: (4)             -----
1342: (4)                 >>> m1 = np.ma.make_mask([0, 1, 1, 0])
1343: (4)                 >>> m2 = np.ma.make_mask([1, 0, 0, 0])
1344: (4)                 >>> np.ma.mask_or(m1, m2)
1345: (4)                 array([ True,  True,  True, False])
1346: (4)                 """
1347: (4)                 if (m1 is nomask) or (m1 is False):
1348: (8)                     dtype = getattr(m2, 'dtype', MaskType)
1349: (8)                     return make_mask(m2, copy=copy, shrink=shrink, dtype=dtype)
1350: (4)                 if (m2 is nomask) or (m2 is False):
1351: (8)                     dtype = getattr(m1, 'dtype', MaskType)
1352: (8)                     return make_mask(m1, copy=copy, shrink=shrink, dtype=dtype)
1353: (4)                 if m1 is m2 and is_mask(m1):
1354: (8)                     return m1
1355: (4)                 (dtype1, dtype2) = (getattr(m1, 'dtype', None), getattr(m2, 'dtype',
None))
1356: (4)                 if dtype1 != dtype2:
1357: (8)                     raise ValueError("Incompatible dtypes '%s'<>'%s'" % (dtype1, dtype2))
1358: (4)                 if dtype1.names is not None:
1359: (8)                     newmask = np.empty(np.broadcast(m1, m2).shape, dtype1)
1360: (8)                     _recursive_mask_or(m1, m2, newmask)
1361: (8)                     return newmask
1362: (4)                 return make_mask(umath.logical_or(m1, m2), copy=copy, shrink=shrink)
1363: (0)             def flatten_mask(mask):
1364: (4)                 """
1365: (4)                     Returns a completely flattened version of the mask, where nested fields
1366: (4)                     are collapsed.
1367: (4)             Parameters
1368: (4)             -----
1369: (4)                 mask : array_like
1370: (8)                     Input array, which will be interpreted as booleans.
1371: (4)             Returns
1372: (4)             -----
1373: (4)                 flattened_mask : ndarray of bools
1374: (8)                     The flattened input.
1375: (4)             Examples
1376: (4)             -----
1377: (4)                 >>> mask = np.array([0, 0, 1])
1378: (4)                 >>> np.ma.flatten_mask(mask)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1379: (4)     array([False, False,  True])
1380: (4)     >>> mask = np.array([(0, 0), (0, 1)], dtype=[('a', bool), ('b', bool)])
1381: (4)     >>> np.ma.flatten_mask(mask)
1382: (4)     array([False, False, False,  True])
1383: (4)     >>> mdtype = [(('a', bool), ('b', [('ba', bool), ('bb', bool)]))]
1384: (4)     >>> mask = np.array([(0, (0, 0)), (0, (0, 1))], dtype=mdtype)
1385: (4)     >>> np.ma.flatten_mask(mask)
1386: (4)     array([False, False, False, False,  True])
1387: (4)     """
1388: (4) def _flatmask(mask):
1389: (8)     "Flatten the mask and returns a (maybe nested) sequence of booleans."
1390: (8)     mnames = mask.dtype.names
1391: (8)     if mnames is not None:
1392: (12)         return [flatten_mask(mask[name]) for name in mnames]
1393: (8)     else:
1394: (12)         return mask
1395: (4) def _flatsequence(sequence):
1396: (8)     "Generates a flattened version of the sequence."
1397: (8)     try:
1398: (12)         for element in sequence:
1399: (16)             if hasattr(element, '__iter__'):
1400: (20)                 yield from _flatsequence(element)
1401: (16)             else:
1402: (20)                 yield element
1403: (8)     except TypeError:
1404: (12)         yield sequence
1405: (4) mask = np.asarray(mask)
1406: (4) flattened = _flatsequence(_flatmask(mask))
1407: (4) return np.array([_ for _ in flattened], dtype=bool)
1408: (0) def _check_mask_axis(mask, axis, keepdims=np._NoValue):
1409: (4)     "Check whether there are masked values along the given axis"
1410: (4)     kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
1411: (4)     if mask is not nomask:
1412: (8)         return mask.all(axis=axis, **kwargs)
1413: (4)     return nomask
1414: (0) def masked_where(condition, a, copy=True):
1415: (4)     """
1416: (4)     Mask an array where a condition is met.
1417: (4)     Return `a` as an array masked where `condition` is True.
1418: (4)     Any masked values of `a` or `condition` are also masked in the output.
1419: (4)     Parameters
1420: (4)     -----
1421: (4)     condition : array_like
1422: (8)         Masking condition. When `condition` tests floating point values for
1423: (8)         equality, consider using ``masked_values`` instead.
1424: (4)     a : array_like
1425: (8)         Array to mask.
1426: (4)     copy : bool
1427: (8)         If True (default) make a copy of `a` in the result. If False modify
1428: (8)         `a` in place and return a view.
1429: (4)     Returns
1430: (4)     -----
1431: (4)     result : MaskedArray
1432: (8)         The result of masking `a` where `condition` is True.
1433: (4)     See Also
1434: (4)     -----
1435: (4)     masked_values : Mask using floating point equality.
1436: (4)     masked_equal : Mask where equal to a given value.
1437: (4)     masked_not_equal : Mask where `not` equal to a given value.
1438: (4)     masked_less_equal : Mask where less than or equal to a given value.
1439: (4)     masked_greater_equal : Mask where greater than or equal to a given value.
1440: (4)     masked_less : Mask where less than a given value.
1441: (4)     masked_greater : Mask where greater than a given value.
1442: (4)     masked_inside : Mask inside a given interval.
1443: (4)     masked_outside : Mask outside a given interval.
1444: (4)     masked_invalid : Mask invalid values (NaNs or infs).
1445: (4)     Examples
1446: (4)     -----
1447: (4)     >>> import numpy.ma as ma

```

```

1448: (4)          >>> a = np.arange(4)
1449: (4)          >>> a
1450: (4)          array([0, 1, 2, 3])
1451: (4)          >>> ma.masked_where(a <= 2, a)
1452: (4)          masked_array(data=[--, --, --, 3],
1453: (17)             mask=[ True,  True,  True, False],
1454: (11)             fill_value=999999)
1455: (4)          Mask array `b` conditional on `a`.
1456: (4)          >>> b = ['a', 'b', 'c', 'd']
1457: (4)          >>> ma.masked_where(a == 2, b)
1458: (4)          masked_array(data=['a', 'b', --, 'd'],
1459: (17)             mask=[False, False, True, False],
1460: (11)             fill_value='N/A',
1461: (16)             dtype='<U1')
1462: (4)          Effect of the `copy` argument.
1463: (4)          >>> c = ma.masked_where(a <= 2, a)
1464: (4)          >>> c
1465: (4)          masked_array(data=[--, --, --, 3],
1466: (17)             mask=[ True,  True,  True, False],
1467: (11)             fill_value=999999)
1468: (4)          >>> c[0] = 99
1469: (4)          >>> c
1470: (4)          masked_array(data=[99, --, --, 3],
1471: (17)             mask=[False,  True,  True, False],
1472: (11)             fill_value=999999)
1473: (4)          >>> a
1474: (4)          array([0, 1, 2, 3])
1475: (4)          >>> c = ma.masked_where(a <= 2, a, copy=False)
1476: (4)          >>> c[0] = 99
1477: (4)          >>> c
1478: (4)          masked_array(data=[99, --, --, 3],
1479: (17)             mask=[False,  True,  True, False],
1480: (11)             fill_value=999999)
1481: (4)          >>> a
1482: (4)          array([99,  1,  2,  3])
1483: (4)          When `condition` or `a` contain masked values.
1484: (4)          >>> a = np.arange(4)
1485: (4)          >>> a = ma.masked_where(a == 2, a)
1486: (4)          >>> a
1487: (4)          masked_array(data=[0, 1, --, 3],
1488: (17)             mask=[False, False, True, False],
1489: (11)             fill_value=999999)
1490: (4)          >>> b = np.arange(4)
1491: (4)          >>> b = ma.masked_where(b == 0, b)
1492: (4)          >>> b
1493: (4)          masked_array(data=[--, 1, 2, 3],
1494: (17)             mask=[ True, False, False, False],
1495: (11)             fill_value=999999)
1496: (4)          >>> ma.masked_where(a == 3, b)
1497: (4)          masked_array(data=[--, 1, --, --],
1498: (17)             mask=[ True, False, True,  True],
1499: (11)             fill_value=999999)
1500: (4)          """
1501: (4)          cond = make_mask(condition, shrink=False)
1502: (4)          a = np.array(a, copy=copy, subok=True)
1503: (4)          (cshape, ashape) = (cond.shape, a.shape)
1504: (4)          if cshape and ashape != ashape:
1505: (8)              raise IndexError("Inconsistent shape between the condition and the
input"
1506: (25)                  " (got %s and %s)" % (cshape, ashape))
1507: (4)          if hasattr(a, '_mask'):
1508: (8)              cond = mask_or(cond, a._mask)
1509: (8)              cls = type(a)
1510: (4)          else:
1511: (8)              cls = MaskedArray
1512: (4)          result = a.view(cls)
1513: (4)          result.mask = _shrink_mask(cond)
1514: (4)          if not copy and hasattr(a, '_mask') and getmask(a) is nomask:
1515: (8)              a._mask = result._mask.view()

```

```

1516: (4)             return result
1517: (0)             def masked_greater(x, value, copy=True):
1518: (4)               """
1519: (4)                 Mask an array where greater than a given value.
1520: (4)                 This function is a shortcut to ``masked_where`` , with
1521: (4)                 `condition` = (x > value).
1522: (4)                 See Also
1523: (4)                   -----
1524: (4)                   masked_where : Mask where a condition is met.
1525: (4)                 Examples
1526: (4)                   -----
1527: (4)                   >>> import numpy.ma as ma
1528: (4)                   >>> a = np.arange(4)
1529: (4)                   >>> a
1530: (4)                   array([0, 1, 2, 3])
1531: (4)                   >>> ma.masked_greater(a, 2)
1532: (4)                   masked_array(data=[0, 1, 2, --],
1533: (17)                         mask=[False, False, False, True],
1534: (11)                         fill_value=999999)
1535: (4)               """
1536: (4)             return masked_where(greater(x, value), x, copy=copy)
1537: (0)             def masked_greater_equal(x, value, copy=True):
1538: (4)               """
1539: (4)                 Mask an array where greater than or equal to a given value.
1540: (4)                 This function is a shortcut to ``masked_where`` , with
1541: (4)                 `condition` = (x >= value).
1542: (4)                 See Also
1543: (4)                   -----
1544: (4)                   masked_where : Mask where a condition is met.
1545: (4)                 Examples
1546: (4)                   -----
1547: (4)                   >>> import numpy.ma as ma
1548: (4)                   >>> a = np.arange(4)
1549: (4)                   >>> a
1550: (4)                   array([0, 1, 2, 3])
1551: (4)                   >>> ma.masked_greater_equal(a, 2)
1552: (4)                   masked_array(data=[0, 1, --, --],
1553: (17)                         mask=[False, False, True, True],
1554: (11)                         fill_value=999999)
1555: (4)               """
1556: (4)             return masked_where(greater_equal(x, value), x, copy=copy)
1557: (0)             def masked_less(x, value, copy=True):
1558: (4)               """
1559: (4)                 Mask an array where less than a given value.
1560: (4)                 This function is a shortcut to ``masked_where`` , with
1561: (4)                 `condition` = (x < value).
1562: (4)                 See Also
1563: (4)                   -----
1564: (4)                   masked_where : Mask where a condition is met.
1565: (4)                 Examples
1566: (4)                   -----
1567: (4)                   >>> import numpy.ma as ma
1568: (4)                   >>> a = np.arange(4)
1569: (4)                   >>> a
1570: (4)                   array([0, 1, 2, 3])
1571: (4)                   >>> ma.masked_less(a, 2)
1572: (4)                   masked_array(data=[--, --, 2, 3],
1573: (17)                         mask=[ True, True, False, False],
1574: (11)                         fill_value=999999)
1575: (4)               """
1576: (4)             return masked_where(less(x, value), x, copy=copy)
1577: (0)             def masked_less_equal(x, value, copy=True):
1578: (4)               """
1579: (4)                 Mask an array where less than or equal to a given value.
1580: (4)                 This function is a shortcut to ``masked_where`` , with
1581: (4)                 `condition` = (x <= value).
1582: (4)                 See Also
1583: (4)                   -----
1584: (4)                   masked_where : Mask where a condition is met.

```

```

1585: (4)          Examples
1586: (4)          -----
1587: (4)          >>> import numpy.ma as ma
1588: (4)          >>> a = np.arange(4)
1589: (4)          >>> a
1590: (4)          array([0, 1, 2, 3])
1591: (4)          >>> ma.masked_less_equal(a, 2)
1592: (4)          masked_array(data=[--, --, --, 3],
1593: (17)             mask=[ True,  True,  True, False],
1594: (11)             fill_value=999999)
1595: (4)
1596: (4)          """
1597: (0)          return masked_where(less_equal(x, value), x, copy=copy)
def masked_not_equal(x, value, copy=True):
    """
1598: (4)          Mask an array where `not` equal to a given value.
1599: (4)          This function is a shortcut to ``masked_where``, with
1600: (4)          `condition` = (x != value).
1601: (4)          See Also
1602: (4)          -----
1603: (4)          masked_where : Mask where a condition is met.
1604: (4)          Examples
1605: (4)          -----
1606: (4)          >>> import numpy.ma as ma
1607: (4)          >>> a = np.arange(4)
1608: (4)          >>> a
1609: (4)          array([0, 1, 2, 3])
1610: (4)          >>> ma.masked_not_equal(a, 2)
1611: (4)          masked_array(data=[--, --, 2, --],
1612: (4)             mask=[ True,  True, False,  True],
1613: (17)             fill_value=999999)
1614: (11)
1615: (4)
1616: (4)          """
1617: (0)          return masked_where(not_equal(x, value), x, copy=copy)
def masked_equal(x, value, copy=True):
    """
1618: (4)          Mask an array where equal to a given value.
1619: (4)          Return a MaskedArray, masked where the data in array `x` are
1620: (4)          equal to `value`. The fill_value of the returned MaskedArray
1621: (4)          is set to `value`.
1622: (4)          For floating point arrays, consider using ``masked_values(x, value)``.
1623: (4)          See Also
1624: (4)          -----
1625: (4)          masked_where : Mask where a condition is met.
1626: (4)          masked_values : Mask using floating point equality.
1627: (4)          Examples
1628: (4)          -----
1629: (4)          >>> import numpy.ma as ma
1630: (4)          >>> a = np.arange(4)
1631: (4)          >>> a
1632: (4)          array([0, 1, 2, 3])
1633: (4)          >>> ma.masked_equal(a, 2)
1634: (4)          masked_array(data=[0, 1, --, 3],
1635: (4)             mask=[False, False,  True, False],
1636: (17)             fill_value=2)
1637: (11)
1638: (4)
1639: (4)          """
1640: (4)          output = masked_where(equal(x, value), x, copy=copy)
1641: (4)          output.fill_value = value
1642: (4)          return output
def masked_inside(x, v1, v2, copy=True):
    """
1643: (4)          Mask an array inside a given interval.
1644: (4)          Shortcut to ``masked_where``, where `condition` is True for `x` inside
1645: (4)          the interval [v1,v2] (v1 <= x <= v2). The boundaries `v1` and `v2`
1646: (4)          can be given in either order.
1647: (4)          See Also
1648: (4)          -----
1649: (4)          masked_where : Mask where a condition is met.
1650: (4)          Notes
1651: (4)          -----
1652: (4)          The array `x` is prefilled with its filling value.
1653: (4)

```

```

1654: (4) Examples
1655: (4)
1656: (4)
1657: (4)
1658: (4)
1659: (4)
1660: (17)
1661: (11)
1662: (4)
1663: (4)
1664: (4)
1665: (17)
1666: (11)
1667: (4)
1668: (4)
1669: (8)
1670: (4)
1671: (4)
1672: (4)
1673: (0)
1674: (4)
1675: (4)
1676: (4)
1677: (4)
1678: (4)
1679: (4)
1680: (4)
1681: (4)
1682: (4)
1683: (4)
1684: (4)
1685: (4)
1686: (4)
1687: (4)
1688: (4)
1689: (4)
1690: (4)
1691: (17)
1692: (11)
1693: (4)
1694: (4)
1695: (4)
1696: (17)
1697: (11)
1698: (4)
1699: (4)
1700: (8)
1701: (4)
1702: (4)
1703: (4)
1704: (0)
1705: (4)
1706: (4)
1707: (4)
1708: (4)
1709: (4)
1710: (4)
1711: (4)
1712: (8)
1713: (4)
1714: (8)
1715: (4)
1716: (8)
1717: (4)
1718: (8)
1719: (4)
1720: (4)
1721: (4)
1722: (8)

Examples
-----
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
masked_array(data=[0.31, 1.2, --, --, -0.4, -1.1],
             mask=[False, False, True, True, False, False],
             fill_value=1e+20)
The order of `v1` and `v2` doesn't matter.
>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data=[0.31, 1.2, --, --, -0.4, -1.1],
             mask=[False, False, True, True, False, False],
             fill_value=1e+20)
"""
if v2 < v1:
    (v1, v2) = (v2, v1)
xf = filled(x)
condition = (xf >= v1) & (xf <= v2)
return masked_where(condition, x, copy=copy)
def masked_outside(x, v1, v2, copy=True):
"""
Mask an array outside a given interval.
Shortcut to ``masked_where``, where `condition` is True for `x` outside
the interval [v1,v2] ( $x < v1 \mid x > v2$ ).
The boundaries `v1` and `v2` can be given in either order.
See Also
-----
masked_where : Mask where a condition is met.
Notes
-----
The array `x` is prefilled with its filling value.
Examples
-----
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
masked_array(data=[--, --, 0.01, 0.2, --, --],
             mask=[ True,  True, False, False,  True,  True],
             fill_value=1e+20)
The order of `v1` and `v2` doesn't matter.
>>> ma.masked_outside(x, 0.3, -0.3)
masked_array(data=[--, --, 0.01, 0.2, --, --],
             mask=[ True,  True, False, False,  True,  True],
             fill_value=1e+20)
"""
if v2 < v1:
    (v1, v2) = (v2, v1)
xf = filled(x)
condition = (xf < v1) | (xf > v2)
return masked_where(condition, x, copy=copy)
def masked_object(x, value, copy=True, shrink=True):
"""
Mask the array `x` where the data are exactly equal to value.
This function is similar to `masked_values`, but only suitable
for object arrays: for floating point, use `masked_values` instead.
Parameters
-----
x : array_like
    Array to mask
value : object
    Comparison value
copy : {True, False}, optional
    Whether to return a copy of `x`.
shrink : {True, False}, optional
    Whether to collapse a mask full of False to nomask
Returns
-----
result : MaskedArray
    The result of masking `x` where equal to `value`.

```

```

1723: (4) See Also
1724: (4)
1725: (4)
1726: (4)
1727: (4)
1728: (4)
1729: (4)
1730: (4)
1731: (4)
1732: (4)
1733: (4)
1734: (4)
1735: (4)
1736: (17)
1737: (11)
1738: (16)
1739: (4)
1740: (4)
1741: (4)
1742: (4)
1743: (4)
1744: (17)
1745: (11)
1746: (16)
1747: (4)
1748: (4)
1749: (4)
1750: (17)
1751: (11)
1752: (16)
1753: (4)
1754: (4)
1755: (8)
1756: (8)
1757: (4)
1758: (8)
1759: (8)
1760: (4)
1761: (4)
1762: (0)
1763: (4)
1764: (4)
1765: (4)
1766: (4)
1767: (4)
1768: (4)
1769: (4)
1770: (4)
1771: (4)
1772: (4)
1773: (4)
1774: (4)
1775: (8)
1776: (4)
1777: (8)
1778: (4)
1779: (8)
1780: (4)
1781: (8)
1782: (4)
1783: (8)
1784: (4)
1785: (4)
1786: (4)
1787: (8)
1788: (4)
1789: (4)
1790: (4)
1791: (4)

See Also
-----
masked_where : Mask where a condition is met.
masked_equal : Mask where equal to a given value (integers).
masked_values : Mask using floating point equality.
Examples
-----
>>> import numpy.ma as ma
>>> food = np.array(['green_eggs', 'ham'], dtype=object)
>>> # don't eat spoiled food
>>> eat = ma.masked_object(food, 'green_eggs')
>>> eat
masked_array(data=[--, 'ham'],
              mask=[ True, False],
              fill_value='green_eggs',
              dtype=object)
>>> # plain ol` ham is boring
>>> fresh_food = np.array(['cheese', 'ham', 'pineapple'], dtype=object)
>>> eat = ma.masked_object(fresh_food, 'green_eggs')
>>> eat
masked_array(data=['cheese', 'ham', 'pineapple'],
              mask=False,
              fill_value='green_eggs',
              dtype=object)
Note that `mask` is set to ``nomask`` if possible.
>>> eat
masked_array(data=['cheese', 'ham', 'pineapple'],
              mask=False,
              fill_value='green_eggs',
              dtype=object)
"""
if isMaskedArray(x):
    condition = umath.equal(x._data, value)
    mask = x._mask
else:
    condition = umath.equal(np.asarray(x), value)
    mask = nomask
mask = mask_or(mask, make_mask(condition, shrink=shrink))
return masked_array(x, mask=mask, copy=copy, fill_value=value)
def masked_values(x, value, rtol=1e-5, atol=1e-8, copy=True, shrink=True):
"""
Mask using floating point equality.
Return a MaskedArray, masked where the data in array `x` are approximately
equal to `value`, determined using `isclose`. The default tolerances for
`masked_values` are the same as those for `isclose`.
For integer types, exact equality is used, in the same way as
`masked_equal`.
The fill_value is set to `value` and the mask is set to ``nomask`` if
possible.
Parameters
-----
x : array_like
    Array to mask.
value : float
    Masking value.
rtol, atol : float, optional
    Tolerance parameters passed on to `isclose`
copy : bool, optional
    Whether to return a copy of `x`.
shrink : bool, optional
    Whether to collapse a mask full of False to ``nomask``.
Returns
-----
result : MaskedArray
    The result of masking `x` where approximately equal to `value`.
See Also
-----
masked_where : Mask where a condition is met.
masked_equal : Mask where equal to a given value (integers).

```

```

1792: (4) Examples
1793: (4)
1794: (4)
1795: (4)
1796: (4)
1797: (4)
1798: (17)
1799: (11)
1800: (4)
1801: (4)
1802: (4)
1803: (17)
1804: (11)
1805: (4)
1806: (4)
1807: (4)
1808: (17)
1809: (11)
1810: (4)
1811: (4)
1812: (4)
1813: (8)
1814: (4)
1815: (8)
1816: (4)
1817: (4)
1818: (8)
1819: (4)
1820: (0)
1821: (4)
1822: (4)
1823: (4)
1824: (4)
1825: (4)
1826: (4)
1827: (4)
1828: (4)
1829: (4)
1830: (4)
1831: (4)
1832: (4)
1833: (4)
1834: (4)
1835: (4)
1836: (4)
1837: (4)
1838: (4)
1839: (4)
1840: (17)
1841: (11)
1842: (4)
1843: (4)
1844: (4)
1845: (4)
1846: (8)
1847: (4)
1848: (0)
1849: (4)
1850: (4)
1851: (4)
1852: (4)
1853: (8)
1854: (8)
1855: (8)
1856: (8)
1857: (8)
1858: (4)
1859: (8)
1860: (8)

    Examples
    -----
    >>> import numpy.ma as ma
    >>> x = np.array([1, 1.1, 2, 1.1, 3])
    >>> ma.masked_values(x, 1.1)
    masked_array(data=[1.0, --, 2.0, --, 3.0],
                  mask=[False, True, False, True, False],
                  fill_value=1.1)
    Note that `mask` is set to ``nomask`` if possible.
    >>> ma.masked_values(x, 2.1)
    masked_array(data=[1. , 1.1, 2. , 1.1, 3. ],
                  mask=False,
                  fill_value=2.1)
    Unlike `masked_equal`, `masked_values` can perform approximate equalities.
    >>> ma.masked_values(x, 2.1, atol=1e-1)
    masked_array(data=[1.0, 1.1, --, 1.1, 3.0],
                  mask=[False, False, True, False, False],
                  fill_value=2.1)
    """
    xnew = filled(x, value)
    if np.issubdtype(xnew.dtype, np.floating):
        mask = np.isclose(xnew, value, atol=atol, rtol=rtol)
    else:
        mask = umath.equal(xnew, value)
    ret = masked_array(xnew, mask=mask, copy=copy, fill_value=value)
    if shrink:
        ret.shrink_mask()
    return ret
def masked_invalid(a, copy=True):
    """
    Mask an array where invalid values occur (NaNs or infs).
    This function is a shortcut to ``masked_where``, with
    `condition` = ~(np.isfinite(a)). Any pre-existing mask is conserved.
    Only applies to arrays with a dtype where NaNs or infs make sense
    (i.e. floating point types), but accepts any array_like object.
    See Also
    -----
    masked_where : Mask where a condition is met.
    Examples
    -----
    >>> import numpy.ma as ma
    >>> a = np.arange(5, dtype=float)
    >>> a[2] = np.NAN
    >>> a[3] = np.PINF
    >>> a
    array([ 0.,  1., nan, inf,  4.])
    >>> ma.masked_invalid(a)
    masked_array(data=[0.0, 1.0, --, --, 4.0],
                  mask=[False, False, True, True, False],
                  fill_value=1e+20)
    """
    a = np.array(a, copy=False, subok=True)
    res = masked_where(~(np.isfinite(a)), a, copy=copy)
    if res._mask is nomask:
        res._mask = make_mask_none(res.shape, res.dtype)
    return res
class _MaskedPrintOption:
    """
    Handle the string used to represent missing data in a masked array.
    """
    def __init__(self, display):
        """
        Create the masked_print_option object.
        """
        self._display = display
        self._enabled = True
    def display(self):
        """
        Display the string to print for masked values.

```

```

1861: (8)             """
1862: (8)         return self._display
1863: (4)     def set_display(self, s):
1864: (8)             """
1865: (8)         Set the string to print for masked values.
1866: (8)             """
1867: (8)         self._display = s
1868: (4)     def enabled(self):
1869: (8)             """
1870: (8)         Is the use of the display value enabled?
1871: (8)             """
1872: (8)         return self._enabled
1873: (4)     def enable(self, shrink=1):
1874: (8)             """
1875: (8)         Set the enabling shrink to `shrink`.
1876: (8)             """
1877: (8)         self._enabled = shrink
1878: (4)     def __str__(self):
1879: (8)         return str(self._display)
1880: (4)     __repr__ = __str__
1881: (0)     masked_print_option = _MaskedPrintOption('---')
1882: (0)     def _recursive_printoption(result, mask, printopt):
1883: (4)             """
1884: (4)         Puts printoptions in result where mask is True.
1885: (4)         Private function allowing for recursion
1886: (4)             """
1887: (4)         names = result.dtype.names
1888: (4)         if names is not None:
1889: (8)             for name in names:
1890: (12)                 curdata = result[name]
1891: (12)                 curmask = mask[name]
1892: (12)                 _recursive_printoption(curdata, curmask, printopt)
1893: (4)         else:
1894: (8)             np.copyto(result, printopt, where=mask)
1895: (4)         return
1896: (0)     _legacy_print_templates = dict(
1897: (4)         long_std=textwrap.dedent("""\
1898: (8)             masked_%(name)s(data =
1899: (9)                 %(data)s,
1900: (8)                 %(nlen)s      mask =
1901: (9)                 %(mask)s,
1902: (8)                 %(nlen)s  fill_value = %(fill)s)
1903: (8)                 """),
1904: (4)         long_flx=textwrap.dedent("""\
1905: (8)             masked_%(name)s(data =
1906: (9)                 %(data)s,
1907: (8)                 %(nlen)s      mask =
1908: (9)                 %(mask)s,
1909: (8)                 %(nlen)s  fill_value = %(fill)s,
1910: (8)                 %(nlen)s      dtype = %(dtype)s)
1911: (8)                 """),
1912: (4)         short_std=textwrap.dedent("""\
1913: (8)             masked_%(name)s(data = %(data)s,
1914: (8)                 %(nlen)s      mask = %(mask)s,
1915: (8)                 %(nlen)s  fill_value = %(fill)s)
1916: (8)                 """),
1917: (4)         short_flx=textwrap.dedent("""\
1918: (8)             masked_%(name)s(data = %(data)s,
1919: (8)                 %(nlen)s      mask = %(mask)s,
1920: (8)                 %(nlen)s  fill_value = %(fill)s,
1921: (8)                 %(nlen)s      dtype = %(dtype)s)
1922: (8)                 """)
1923: (0)
1924: (0)     def _recursive_filled(a, mask, fill_value):
1925: (4)             """
1926: (4)         Recursively fill `a` with `fill_value`.
1927: (4)             """
1928: (4)         names = a.dtype.names
1929: (4)         for name in names:

```

```

1930: (8)             current = a[name]
1931: (8)             if current.dtype.names is not None:
1932: (12)             _recursive_filled(current, mask[name], fill_value[name])
1933: (8)             else:
1934: (12)                 np.copyto(current, fill_value[name], where=mask[name])
1935: (0)             def flatten_structured_array(a):
1936: (4)                 """
1937: (4)                 Flatten a structured array.
1938: (4)                 The data type of the output is chosen such that it can represent all of
the
1939: (4)                 (nested) fields.
1940: (4)             Parameters
1941: (4)                 -----
1942: (4)                 a : structured array
1943: (4)             Returns
1944: (4)                 -----
1945: (4)                 output : masked array or ndarray
1946: (8)                 A flattened masked array if the input is a masked array, otherwise a
standard ndarray.
1947: (8)
1948: (4)             Examples
1949: (4)                 -----
1950: (4)                 >>> ndtype = [('a', int), ('b', float)]
1951: (4)                 >>> a = np.array([(1, 1), (2, 2)], dtype=ndtype)
1952: (4)                 >>> np.ma.flatten_structured_array(a)
1953: (4)                 array([[1., 1.],
1954: (11)                     [2., 2.]])
1955: (4)                 """
1956: (4)             def flatten_sequence(iterable):
1957: (8)                 """
1958: (8)                 Flattens a compound of nested iterables.
1959: (8)                 """
1960: (8)                 for elm in iter(iterable):
1961: (12)                     if hasattr(elm, '__iter__'):
1962: (16)                         yield from flatten_sequence(elm)
1963: (12)                     else:
1964: (16)                         yield elm
1965: (4)             a = np.asarray(a)
1966: (4)             inishape = a.shape
1967: (4)             a = a.ravel()
1968: (4)             if isinstance(a, MaskedArray):
1969: (8)                 out = np.array([tuple(flatten_sequence(d.item())) for d in a._data])
1970: (8)                 out = out.view(MaskedArray)
1971: (8)                 out._mask = np.array([tuple(flatten_sequence(d.item()))
1972: (30)                                 for d in getmaskarray(a)])
1973: (4)
1974: (8)             else:
1975: (4)                 out = np.array([tuple(flatten_sequence(d.item())) for d in a])
1976: (4)             if len(inishape) > 1:
1977: (8)                 newshape = list(out.shape)
1978: (8)                 newshape[0] = inishape
1979: (4)                 out.shape = tuple(flatten_sequence(newshape))
1980: (0)             return out
1981: (4)             def __arraymethod(funcname, onmask=True):
1982: (4)                 """
1983: (4)                 Return a class method wrapper around a basic array method.
1984: (4)                 Creates a class method which returns a masked array, where the new
`__data__` array is the output of the corresponding basic method called
on the original `__data__`.
1985: (4)                 If `onmask` is True, the new mask is the output of the method called
on the initial mask. Otherwise, the new mask is just a reference
to the initial mask.
1986: (4)
1987: (4)
1988: (4)
1989: (4)
1990: (4)
1991: (4)
1992: (8)             Parameters
1993: (4)                 -----
1994: (8)                 funcname : str
1995: (8)                     Name of the function to apply on data.
1996: (8)                 onmask : bool
1997: (4)                     Whether the mask must be processed also (True) or left
alone (False). Default is True. Make available as `__onmask__`
attribute.

```

```

1998: (4)
1999: (4)
2000: (8)
2001: (4)
2002: (4)
2003: (8)
2004: (8)
2005: (8)
2006: (8)
2007: (8)
2008: (12)
2009: (8)
2010: (12)
2011: (8)
2012: (4)
2013: (4)
2014: (8)
2015: (4)
2016: (4)
2017: (0)
2018: (4)
2019: (4)
2020: (4)
2021: (4)
2022: (4)
2023: (4)
2024: (4)
2025: (4)
2026: (4)
2027: (4)
2028: (4)
2029: (4)
2030: (4)
2031: (4)
2032: (4)
2033: (4)
2034: (4)
2035: (4)
2036: (4)
2037: (4)
2038: (4)
2039: (4)
2040: (4)
2041: (4)
2042: (4)
2043: (4)
2044: (4)
2045: (4)
2046: (4)
2047: (4)
2048: (4)
2049: (4)
2050: (4)
2051: (4)
2052: (4)
2053: (17)
2054: (11)
2055: (4)
2056: (4)
2057: (8)
2058: (8)
2059: (8)
2060: (12)
2061: (8)
2062: (12)
2063: (4)
2064: (8)
2065: (4)
2066: (8)

-----  

method : instancemethod  

    Class method wrapper of the specified basic array method.  

"""  

def wrapped_method(self, *args, **params):  

    result = getattr(self._data, funcname)(*args, **params)  

    result = result.view(type(self))  

    result._update_from(self)  

    mask = self._mask  

    if not onmask:  

        result.__setmask__(mask)  

    elif mask is not nomask:  

        result._mask = getattr(mask, funcname)(*args, **params)  

    return result  

methdoc = getattr(ndarray, funcname, None) or getattr(np, funcname, None)  

if methdoc is not None:  

    wrapped_method.__doc__ = methdoc.__doc__  

wrapped_method.__name__ = funcname  

return wrapped_method  

class MaskedIterator:  

"""  

Flat iterator object to iterate over masked arrays.  

A `MaskedIterator` iterator is returned by ``x.flat`` for any masked array  

`x`. It allows iterating over the array as if it were a 1-D array,  

either in a for-loop or by calling its `next` method.  

Iteration is done in C-contiguous style, with the last index varying the  

fastest. The iterator can also be indexed using basic slicing or  

advanced indexing.  

See Also  

-----  

MaskedArray.flat : Return a flat iterator over an array.  

MaskedArray.flatten : Returns a flattened copy of an array.  

Notes  

-----  

`MaskedIterator` is not exported by the `ma` module. Instead of  

instantiating a `MaskedIterator` directly, use `MaskedArray.flat`.  

Examples  

-----  

>>> x = np.ma.array(arange(6).reshape(2, 3))  

>>> f1 = x.flat  

>>> type(f1)  

<class 'numpy.ma.core.MaskedIterator'>  

>>> for item in f1:  

...     print(item)  

...  

0  

1  

2  

3  

4  

5  

Extracting more than a single element b indexing the `MaskedIterator`  

returns a masked array:  

>>> f1[2:4]  

masked_array(data = [2 3],  

             mask = False,  

             fill_value = 999999)  

"""  

def __init__(self, ma):  

    self.ma = ma  

    self.dataiter = ma._data.flat  

    if ma._mask is nomask:  

        self.maskiter = None  

    else:  

        self.maskiter = ma._mask.flat  

def __iter__(self):  

    return self  

def __getitem__(self, indx):  

    result = self.dataiter.__getitem__(indx).view(type(self.ma))

```

```

2067: (8)             if self.maskiter is not None:
2068: (12)             _mask = self.maskiter.__getitem__(indx)
2069: (12)             if isinstance(_mask, ndarray):
2070: (16)                 _mask.shape = result.shape
2071: (16)                 result._mask = _mask
2072: (12)             elif isinstance(_mask, np.void):
2073: (16)                 return mvoid(result, mask=_mask, hardmask=self.ma._hardmask)
2074: (12)             elif _mask: # Just a scalar, masked
2075: (16)                 return masked
2076: (8)             return result
2077: (4)         def __setitem__(self, index, value):
2078: (8)             self.dataiter[index] = getdata(value)
2079: (8)             if self.maskiter is not None:
2080: (12)                 self.maskiter[index] = getmaskarray(value)
2081: (4)         def __next__(self):
2082: (8)             """
2083: (8)             Return the next value, or raise StopIteration.
2084: (8)         Examples
2085: (8)         -----
2086: (8)         >>> x = np.ma.array([3, 2], mask=[0, 1])
2087: (8)         >>> f1 = x.flat
2088: (8)         >>> next(f1)
2089: (8)         3
2090: (8)         >>> next(f1)
2091: (8)         masked
2092: (8)         >>> next(f1)
2093: (8)         Traceback (most recent call last):
2094: (10)             ...
2095: (8)             StopIteration
2096: (8)             """
2097: (8)             d = next(self.dataiter)
2098: (8)             if self.maskiter is not None:
2099: (12)                 m = next(self.maskiter)
2100: (12)                 if isinstance(m, np.void):
2101: (16)                     return mvoid(d, mask=m, hardmask=self.ma._hardmask)
2102: (12)                 elif m: # Just a scalar, masked
2103: (16)                     return masked
2104: (8)             return d
2105: (0)         class MaskedArray(ndarray):
2106: (4)             """
2107: (4)             An array class with possibly masked values.
2108: (4)             Masked values of True exclude the corresponding element from any
2109: (4)             computation.
2110: (4)             Construction::
2111: (6)                 x = MaskedArray(data, mask=nomask, dtype=None, copy=False, subok=True,
2112: (22)                               ndmin=0, fill_value=None, keep_mask=True,
2113: (22)                               shrink=True, order=None)
2114: (4)             Parameters
2115: (4)             -----
2116: (4)             data : array_like
2117: (8)                 Input data.
2118: (4)             mask : sequence, optional
2119: (8)                 Mask. Must be convertible to an array of booleans with the same
2120: (8)                 shape as `data`. True indicates a masked (i.e. invalid) data.
2121: (4)             dtype : dtype, optional
2122: (8)                 Data type of the output.
2123: (8)                 If `dtype` is None, the type of the data argument (`data.dtype`)
2124: (8)                 is used. If `dtype` is not None and different from `data.dtype`,
2125: (8)                 a copy is performed.
2126: (4)             copy : bool, optional
2127: (8)                 Whether to copy the input data (True), or to use a reference instead.
2128: (8)                 Default is False.
2129: (4)             subok : bool, optional
2130: (8)                 Whether to return a subclass of `MaskedArray` if possible (True) or a
2131: (8)                 plain `MaskedArray`. Default is True.
2132: (4)             ndmin : int, optional
2133: (8)                 Minimum number of dimensions. Default is 0.
2134: (4)             fill_value : scalar, optional

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2135: (8)           Value used to fill in the masked values when necessary.
2136: (8)           If None, a default based on the data-type is used.
2137: (4)           keep_mask : bool, optional
2138: (8)             Whether to combine `mask` with the mask of the input data, if any
2139: (8)             (True), or to use only `mask` for the output (False). Default is True.
2140: (4)           hard_mask : bool, optional
2141: (8)             Whether to use a hard mask or not. With a hard mask, masked values
2142: (8)             cannot be unmasked. Default is False.
2143: (4)           shrink : bool, optional
2144: (8)             Whether to force compression of an empty mask. Default is True.
2145: (4)           order : {'C', 'F', 'A'}, optional
2146: (8)             Specify the order of the array. If order is 'C', then the array
2147: (8)             will be in C-contiguous order (last-index varies the fastest).
2148: (8)             If order is 'F', then the returned array will be in
2149: (8)             Fortran-contiguous order (first-index varies the fastest).
2150: (8)             If order is 'A' (default), then the returned array may be
2151: (8)             in any order (either C-, Fortran-contiguous, or even discontiguous),
2152: (8)             unless a copy is required, in which case it will be C-contiguous.
2153: (4)           Examples
2154: (4)
2155: (4)           -----
2156: (4)             The ``mask`` can be initialized with an array of boolean values
2157: (4)             with the same shape as ``data``.
2158: (4)             >>> data = np.arange(6).reshape((2, 3))
2159: (4)             >>> np.ma.MaskedArray(data, mask=[[False, True, False],
2160: (4)                           ...                               [False, False, True]])
2161: (6)             masked_array(
2162: (12)               data=[[0, --, 2],
2163: (6)                 [3, 4, --]],
2164: (12)               mask=[[False, True, False],
2165: (6)                 [False, False, True]],
2166: (4)               fill_value=999999)
2167: (4)             Alternatively, the ``mask`` can be initialized to homogeneous boolean
2168: (4)             array with the same shape as ``data`` by passing in a scalar
2169: (4)             boolean value:
2170: (4)             >>> np.ma.MaskedArray(data, mask=False)
2171: (6)             masked_array(
2172: (12)               data=[[0, 1, 2],
2173: (6)                 [3, 4, 5]],
2174: (12)               mask=[[False, False, False],
2175: (6)                 [False, False, False]],
2176: (4)               fill_value=999999)
2177: (4)             >>> np.ma.MaskedArray(data, mask=True)
2178: (6)             masked_array(
2179: (12)               data=[[--, --, --],
2180: (6)                 [--, --, --]],
2181: (12)               mask=[[True, True, True],
2182: (6)                 [True, True, True]],
2183: (6)               fill_value=999999,
2184: (4)               dtype=int64)
2185: (4)             .. note::
2186: (4)               The recommended practice for initializing ``mask`` with a scalar
2187: (4)               boolean value is to use ``True``/``False`` rather than
2188: (4)               ``np.True_``/``np.False_``. The reason is :attr:`nomask`
2189: (4)               is represented internally as ``np.False_``.
2190: (4)               >>> np.False_ is np.ma.nomask
2191: (4)               True
2192: (4)
2193: (4)             __array_priority__ = 15
2194: (4)             _defaultmask = nomask
2195: (4)             _defaulthardmask = False
2196: (4)             _baseclass = ndarray
2197: (4)             _print_width = 100
2198: (4)             _print_width_1d = 1500
2199: (16)            def __new__(cls, data=None, mask=nomask, dtype=None, copy=False,
2200: (16)                          subok=True, ndmin=0, fill_value=None, keep_mask=True,
2201: (8)                            hard_mask=None, shrink=True, order=None):
2202: (8)
2203: (8)               Create a new masked array from scratch.
2204: (8)               Notes

```

```

2204: (8)          -----
2205: (8)          A masked array can also be created by taking a .view(MaskedArray).
2206: (8)
2207: (8)
2208: (25)         """
2209: (8)         _data = np.array(data, dtype=dtype, copy=copy,
2210: (8)                 order=order, subok=True, ndmin=ndmin)
2211: (12)         _baseclass = getattr(data, '_baseclass', type(_data))
2212: (8)         if isinstance(data, MaskedArray) and (data.shape != _data.shape):
2213: (12)             copy = True
2214: (8)         if isinstance(data, cls) and subok and not isinstance(data,
2215: (12)             _data = ndarray.view(_data, type(data)))
2216: (8)         else:
2217: (12)             _data = ndarray.view(_data, cls)
2218: (8)         if hasattr(data, '_mask') and not isinstance(data, ndarray):
2219: (8)             _data._mask = data._mask
2220: (12)         mdtype = make_mask_descr(_data.dtype)
2221: (16)         if mask is nomask:
2222: (20)             if not keep_mask:
2223: (16)                 if shrink:
2224: (20)                     _data._mask = nomask
2225: (12)                 else:
2226: (16)                     _data._mask = np.zeros(_data.shape, dtype=mdtype)
2227: (20)             elif isinstance(data, (tuple, list)):
2228: (24)                 try:
2229: (25)                     mask = np.array(
2230: (16)                         [getmaskarray(np.asanyarray(m, dtype=_data.dtype))
2231: (20)                             for m in data], dtype=mdtype)
2232: (16)                 except (ValueError, TypeError):
2233: (20)                     mask = nomask
2234: (20)                 if (mdtype == MaskType) and mask.any():
2235: (12)                     _data._mask = mask
2236: (16)                     _data._sharedmask = False
2237: (16)                 else:
2238: (20)                     _data._sharedmask = not copy
2239: (20)                     if copy:
2240: (24)                         _data._mask = _data._mask.copy()
2241: (28)                         if getmask(data) is not nomask:
2242: (8)                             if data._mask.shape != data.shape:
2243: (12)                                 data._mask.shape = data.shape
2244: (16)             else:
2245: (12)                 if mask is None:
2246: (16)                     mask = False
2247: (12)                 if mask is True and mdtype == MaskType:
2248: (16)                     mask = np.ones(_data.shape, dtype=mdtype)
2249: (12)                 elif mask is False and mdtype == MaskType:
2250: (16)                     mask = np.zeros(_data.shape, dtype=mdtype)
2251: (20)                 else:
2252: (16)                     try:
2253: (20)                         mask = np.array(mask, copy=copy, dtype=mdtype)
2254: (36)                     except TypeError:
2255: (12)                         mask = np.array([tuple([m] * len(mdtype)) for m in mask],
2256: (16)                                         dtype=mdtype)
2257: (16)                     if mask.shape != _data.shape:
2258: (20)                         (nd, nm) = (_data.size, mask.size)
2259: (16)                         if nm == 1:
2260: (20)                             mask = np.resize(mask, _data.shape)
2261: (16)                         elif nm == nd:
2262: (20)                             mask = np.reshape(mask, _data.shape)
2263: (26)                         else:
2264: (20)                             msg = "Mask and data not compatible: data size is %i, " +
2265: (16)                                         "mask size is %i."
2266: (12)                             raise MaskError(msg % (nd, nm))
2267: (16)                             copy = True
2268: (16)                             if _data._mask is nomask:
2269: (12)                                 _data._mask = mask
2270: (16)                                 _data._sharedmask = not copy
2271: (16)                             else:
2272: (16)                                 if not keep_mask:

```

```

2271: (20)             _data._mask = mask
2272: (20)             _data._sharedmask = not copy
2273: (16)         else:
2274: (20)             if _data.dtype.names is not None:
2275: (24)                 def _recursive_or(a, b):
2276: (28)                     "do a|b on each field of a, recursively"
2277: (28)                     for name in a.dtype.names:
2278: (32)                         (af, bf) = (a[name], b[name])
2279: (32)                         if af.dtype.names is not None:
2280: (36)                             _recursive_or(af, bf)
2281: (32)                         else:
2282: (36)                             af |= bf
2283: (24)                     _recursive_or(_data._mask, mask)
2284: (20)             else:
2285: (24)                 _data._mask = np.logical_or(mask, _data._mask)
2286: (20)             _data._sharedmask = False
2287: (8)         if fill_value is None:
2288: (12)             fill_value = getattr(data, '_fill_value', None)
2289: (8)         if fill_value is not None:
2290: (12)             _data._fill_value = _check_fill_value(fill_value, _data.dtype)
2291: (8)         if hard_mask is None:
2292: (12)             _data._hardmask = getattr(data, '_hardmask', False)
2293: (8)         else:
2294: (12)             _data._hardmask = hard_mask
2295: (8)         _data._baseclass = _baseclass
2296: (8)         return _data
2297: (4)     def __update_from(self, obj):
2298: (8)         """
2299: (8)             Copies some attributes of obj to self.
2300: (8)         """
2301: (8)         if isinstance(obj, ndarray):
2302: (12)             _baseclass = type(obj)
2303: (8)         else:
2304: (12)             _baseclass = ndarray
2305: (8)         _optinfo = {}
2306: (8)         _optinfo.update(getattr(obj, '_optinfo', {}))
2307: (8)         _optinfo.update(getattr(obj, '_basedict', {}))
2308: (8)         if not isinstance(obj, MaskedArray):
2309: (12)             _optinfo.update(getattr(obj, '__dict__', {}))
2310: (8)         _dict = dict(_fill_value=getattr(obj, '_fill_value', None),
2311: (21)             _hardmask=getattr(obj, '_hardmask', False),
2312: (21)             _sharedmask=getattr(obj, '_sharedmask', False),
2313: (21)             _isfield=getattr(obj, '_isfield', False),
2314: (21)             _baseclass=getattr(obj, '_baseclass', _baseclass),
2315: (21)             _optinfo=_optinfo,
2316: (21)             _basedict=_optinfo)
2317: (8)         self.__dict__.update(_dict)
2318: (8)         self.__dict__.update(_optinfo)
2319: (8)         return
2320: (4)     def __array_finalize__(self, obj):
2321: (8)         """
2322: (8)             Finalizes the masked array.
2323: (8)         """
2324: (8)         self.__update_from(obj)
2325: (8)         if isinstance(obj, ndarray):
2326: (12)             if obj.dtype.names is not None:
2327: (16)                 _mask = getmaskarray(obj)
2328: (12)             else:
2329: (16)                 _mask = getmask(obj)
2330: (12)             if (_mask is not nomask and obj.__array_interface__["data"][0]
2331: (20)                 != self.__array_interface__["data"][0]):
2332: (16)                 if self.dtype == obj.dtype:
2333: (20)                     _mask_dtype = _mask.dtype
2334: (16)                 else:
2335: (20)                     _mask_dtype = make_mask_descr(self.dtype)
2336: (16)                 if self.flags.c_contiguous:
2337: (20)                     order = "C"
2338: (16)                 elif self.flags.f_contiguous:
2339: (20)                     order = "F"

```

```

2340: (16)
2341: (20)
2342: (16)
2343: (12)
2344: (16)
2345: (8)
2346: (12)
2347: (8)
2348: (8)
2349: (12)
2350: (16)
2351: (12)
2352: (16)
2353: (12)
2354: (16)
2355: (8)
2356: (12)
2357: (8)
2358: (12)
2359: (4)
2360: (8)
2361: (8)
2362: (8)
2363: (8)
2364: (8)
2365: (12)
2366: (8)
2367: (12)
2368: (12)
2369: (8)
2370: (12)
2371: (12)
2372: (12)
2373: (12)
2374: (12)
2375: (12)
2376: (16)
2377: (20)
2378: (16)
2379: (20)
2380: (24)
2381: (20)
2382: (24)
2383: (20)
2384: (24)
2385: (20)
2386: (20)
2387: (24)
2388: (20)
2389: (24)
2390: (12)
2391: (16)
2392: (12)
2393: (16)
2394: (16)
2395: (8)
2396: (4)
2397: (8)
2398: (8)
2399: (8)
2400: (8)
2401: (8)
2402: (12)
2403: (12)
2404: (12)
2405: (12)
2406: (12)
2407: (12)
2408: (8)

        else:
            order = "K"
            _mask = _mask.astype(_mask_dtype, order)
        else:
            _mask = _mask.view()
    else:
        _mask = nomask
    self._mask = _mask
    if self._mask is not nomask:
        try:
            self._mask.shape = self.shape
        except ValueError:
            self._mask = nomask
        except (TypeError, AttributeError):
            pass
    if self._fill_value is not None:
        self._fill_value = _check_fill_value(self._fill_value, self.dtype)
    elif self.dtype.names is not None:
        self._fill_value = _check_fill_value(None, self.dtype)
def __array_wrap__(self, obj, context=None):
"""
Special hook for ufuncs.
Wraps the numpy array and sets the mask according to context.
"""
if obj is self: # for in-place operations
    result = obj
else:
    result = obj.view(type(self))
    result._update_from(self)
if context is not None:
    result._mask = result._mask.copy()
    func, args, out_i = context
    input_args = args[:func.nin]
    m = reduce(mask_or, [getmaskarray(arg) for arg in input_args])
    domain = ufunc_domain.get(func, None)
    if domain is not None:
        with np.errstate(divide='ignore', invalid='ignore'):
            d = filled(domain(*input_args), True)
        if d.any():
            try:
                fill_value = ufunc_fills[func][-1]
            except TypeError:
                fill_value = ufunc_fills[func]
            except KeyError:
                fill_value = self.fill_value
            np.copyto(result, fill_value, where=d)
            if m is nomask:
                m = d
            else:
                m = (m | d)
    if result is not self and result.shape == () and m:
        return masked
    else:
        result._mask = m
        result._sharedmask = False
return result
def view(self, dtype=None, type=None, fill_value=None):
"""
Return a view of the MaskedArray data.
Parameters
-----
dtype : data-type or ndarray sub-class, optional
    Data-type descriptor of the returned view, e.g., float32 or int16.
    The default, None, results in the view having the same data-type
    as `a`. As with ``ndarray.view``, dtype can also be specified as
    an ndarray sub-class, which then specifies the type of the
    returned object (this is equivalent to setting the ``type``
    parameter).
type : Python type, optional
"""

```

```

2409: (12)                                         Type of the returned view, either ndarray or a subclass. The
2410: (12)                                         default None results in type preservation.
2411: (8)                                          fill_value : scalar, optional
2412: (12)                                         The value to use for invalid entries (None by default).
2413: (12)                                         If None, then this argument is inferred from the passed `dtype`,
or
2414: (12)                                         in its absence the original array, as discussed in the notes
below.
2415: (8)                                          See Also
2416: (8)                                         -----
2417: (8)                                         numpy.ndarray.view : Equivalent method on ndarray object.
2418: (8)                                         Notes
2419: (8)                                         -----
2420: (8)                                         ``a.view()`` is used two different ways:
2421: (8)                                         ``a.view(some_dtype)`` or ``a.view(dtype=some_dtype)`` constructs a
view
2422: (8)                                         of the array's memory with a different data-type. This can cause a
2423: (8)                                         reinterpretation of the bytes of memory.
2424: (8)                                         ``a.view(ndarray_subclass)`` or ``a.view(type=ndarray_subclass)`` just
2425: (8)                                         returns an instance of `ndarray_subclass` that looks at the same array
2426: (8)                                         (same shape, dtype, etc.) This does not cause a reinterpretation of
the
2427: (8)                                         memory.
2428: (8)                                         If `fill_value` is not specified, but `dtype` is specified (and is not
2429: (8)                                         an ndarray sub-class), the `fill_value` of the MaskedArray will be
2430: (8)                                         reset. If neither `fill_value` nor `dtype` are specified (or if
2431: (8)                                         `dtype` is an ndarray sub-class), then the fill value is preserved.
2432: (8)                                         Finally, if `fill_value` is specified, but `dtype` is not, the fill
2433: (8)                                         value is set to the specified value.
2434: (8)                                         For ``a.view(some_dtype)``, if ``some_dtype`` has a different number
of
2435: (8)                                         bytes per entry than the previous dtype (for example, converting a
2436: (8)                                         regular array to a structured array), then the behavior of the view
2437: (8)                                         cannot be predicted just from the superficial appearance of ``a``
(shown
2438: (8)                                         by ``print(a)``). It also depends on exactly how ``a`` is stored in
memory. Therefore if ``a`` is C-ordered versus fortran-ordered, versus
2439: (8)
2440: (8)
2441: (8)
2442: (8)
2443: (8)                                         defined as a slice or transpose, etc., the view may give different
results.
2444: (12)
2445: (16)
2446: (12)
2447: (16)
2448: (8)
2449: (12)
2450: (16)
2451: (20)
2452: (20)
2453: (16)
2454: (20)
2455: (12)
2456: (16)
2457: (8)
2458: (12)
2459: (8)
2460: (12)
2461: (8)
2462: (12)
2463: (16)
2464: (20)
2465: (16)
2466: (20)
2467: (12)
2468: (16)
2469: (8)
2470: (4)
2471: (8)

        if dtype is None:
            if type is None:
                output = ndarray.view(self)
            else:
                output = ndarray.view(self, type)
        elif type is None:
            try:
                if issubclass(dtype, ndarray):
                    output = ndarray.view(self, dtype)
                    dtype = None
                else:
                    output = ndarray.view(self, dtype)
            except TypeError:
                output = ndarray.view(self, dtype)
        else:
            output = ndarray.view(self, dtype, type)
    if getmask(output) is not nomask:
        output._mask = output._mask.view()
    if getattr(output, '_fill_value', None) is not None:
        if fill_value is None:
            if dtype is None:
                pass # leave _fill_value as is
            else:
                output._fill_value = None
        else:
            output.fill_value = fill_value
    return output
def __getitem__(self, indx):
    """

```

```

2472: (8)             x.__getitem__(y) <==> x[y]
2473: (8)             Return the item described by i, as a masked array.
2474: (8)
2475: (8)             """
2476: (8)             dout = self.data[indx]
2477: (8)             _mask = self._mask
2478: (12)            def _is_scalar(m):
2479: (8)                return not isinstance(m, np.ndarray)
2480: (12)            def _scalar_heuristic(arr, elem):
2481: (12)                """
2482: (12)                Return whether `elem` is a scalar result of indexing `arr`, or
2483: (12)                None
2484: (12)                if undecidable without promoting nomask to a full mask
2485: (16)                """
2486: (12)                if not isinstance(elem, np.ndarray):
2487: (16)                    return True
2488: (20)                elif arr.dtype.type is np.object_:
2489: (12)                    if arr.dtype is not elem.dtype:
2490: (16)                        return True
2491: (12)                    elif type(arr).__getitem__ == ndarray.__getitem__:
2492: (8)                        return False
2493: (12)                    return None
2494: (12)                if _mask is not nomask:
2495: (8)                    mout = _mask[indx]
2496: (12)                    scalar_expected = _is_scalar(mout)
2497: (12)                else:
2498: (12)                    mout = nomask
2499: (16)                    scalar_expected = _scalar_heuristic(self.data, dout)
2500: (8)                if scalar_expected:
2501: (12)                    if isinstance(dout, np.void):
2502: (16)                        return mvoid(dout, mask=mout, hardmask=self._hardmask)
2503: (12)                    elif (self.dtype.type is np.object_ and
2504: (18)                        isinstance(dout, np.ndarray) and
2505: (18)                        dout is not masked):
2506: (16)                        if mout:
2507: (20)                            return MaskedArray(dout, mask=True)
2508: (16)                        else:
2509: (20)                            return dout
2510: (12)                    else:
2511: (16)                        if mout:
2512: (20)                            return masked
2513: (16)                        else:
2514: (20)                            return dout
2515: (8)
2516: (12)                else:
2517: (12)                    dout = dout.view(type(self))
2518: (12)                    dout._update_from(self)
2519: (16)                    if is_string_or_list_of_strings(indx):
2520: (20)                        if self._fill_value is not None:
2521: (20)                            dout._fill_value = self._fill_value[indx]
2522: (24)                            if not isinstance(dout._fill_value, np.ndarray):
2523: (20)                                raise RuntimeError('Internal NumPy error.')
2524: (24)                            if dout._fill_value.ndim > 0:
2525: (32)                                if not (dout._fill_value ==
2526: (28)                                    dout._fill_value.flat[0]).all():
2527: (32)                                    warnings.warn(
2528: (32)                                        "Upon accessing multidimensional field "
2529: (32)                                        f"{{indx!s}}, need to keep dimensionality "
2530: (32)                                        "of fill_value at 0. Discarding "
2531: (32)                                        "heterogeneous fill_value and setting "
2532: (32)                                        f"all to {{dout._fill_value[0]!s}}.",
2533: (24)                                        stacklevel=2)
2534: (16)                            dout._fill_value =
2535: (12)                            dout._fill_value.flat[0:1].squeeze(axis=0)
2536: (16)                            dout._isfield = True
2537: (16)                            if mout is not nomask:
2538: (8)                                dout._mask = reshape(mout, dout.shape)
2539: (16)                                dout._sharedmask = True
2540: (8)                            return dout

```

```

2539: (4) @np.errstate(over='ignore', invalid='ignore')
2540: (4) def __setitem__(self, indx, value):
2541: (8) """
2542: (8)     x.__setitem__(i, y) <==> x[i]=y
2543: (8)     Set item described by index. If value is masked, masks those
2544: (8)     locations.
2545: (8) """
2546: (8)     if self is masked:
2547: (12)         raise MaskError('Cannot alter the masked element.')
2548: (8)     _data = self._data
2549: (8)     _mask = self._mask
2550: (8)     if isinstance(indx, str):
2551: (12)         _data[indx] = value
2552: (12)         if _mask is nomask:
2553: (16)             self._mask = _mask = make_mask_none(self.shape, self.dtype)
2554: (12)             _mask[indx] = getmask(value)
2555: (12)             return
2556: (8)     _dtype = _data.dtype
2557: (8)     if value is masked:
2558: (12)         if _mask is nomask:
2559: (16)             _mask = self._mask = make_mask_none(self.shape, _dtype)
2560: (12)             if _dtype.names is not None:
2561: (16)                 _mask[indx] = tuple([True] * len(_dtype.names))
2562: (12)             else:
2563: (16)                 _mask[indx] = True
2564: (12)             return
2565: (8)     dval = getattr(value, '_data', value)
2566: (8)     mval = getmask(value)
2567: (8)     if _dtype.names is not None and mval is nomask:
2568: (12)         mval = tuple([False] * len(_dtype.names))
2569: (8)     if _mask is nomask:
2570: (12)         _data[indx] = dval
2571: (12)         if mval is not nomask:
2572: (16)             _mask = self._mask = make_mask_none(self.shape, _dtype)
2573: (16)             _mask[indx] = mval
2574: (8)     elif not self._hardmask:
2575: (12)         if (isinstance(indx, masked_array) and
2576: (20)             not isinstance(value, masked_array)):
2577: (16)             _data[indx.data] = dval
2578: (12)         else:
2579: (16)             _data[indx] = dval
2580: (16)             _mask[indx] = mval
2581: (8)     elif hasattr(indx, 'dtype') and (indx.dtype == MaskType):
2582: (12)         indx = indx * umath.logical_not(_mask)
2583: (12)         _data[indx] = dval
2584: (8)     else:
2585: (12)         if _dtype.names is not None:
2586: (16)             err_msg = "Flexible 'hard' masks are not yet supported."
2587: (16)             raise NotImplementedError(err_msg)
2588: (12)         mindx = mask_or(_mask[indx], mval, copy=True)
2589: (12)         dindx = self._data[indx]
2590: (12)         if dindx.size > 1:
2591: (16)             np.copyto(dindx, dval, where=~mindx)
2592: (12)         elif mindx is nomask:
2593: (16)             dindx = dval
2594: (12)             _data[indx] = dindx
2595: (12)             _mask[indx] = mindx
2596: (8)         return
2597: (4)     @property
2598: (4)     def dtype(self):
2599: (8)         return super().dtype
2600: (4)     @dtype.setter
2601: (4)     def dtype(self, dtype):
2602: (8)         super(MaskedArray, type(self)).dtype.__set__(self, dtype)
2603: (8)         if self._mask is not nomask:
2604: (12)             self._mask = self._mask.view(make_mask_descr(dtype), ndarray)
2605: (12)             try:
2606: (16)                 self._mask.shape = self.shape
2607: (12)             except (AttributeError, TypeError):

```

```

2608: (16)                               pass
2609: (4)                                @property
2610: (4)                                def shape(self):
2611: (8)                                  return super().shape
2612: (4)                                @shape.setter
2613: (4)                                def shape(self, shape):
2614: (8)                                  super(MaskedArray, type(self)).shape.__set__(self, shape)
2615: (8)                                  if getmask(self) is not nomask:
2616: (12)                                      self._mask.shape = self.shape
2617: (4)                                def __setmask__(self, mask, copy=False):
2618: (8)                                  """
2619: (8)                                      Set the mask.
2620: (8)                                  """
2621: (8)                                  idtype = self.dtype
2622: (8)                                  current_mask = self._mask
2623: (8)                                  if mask is masked:
2624: (12)                                      mask = True
2625: (8)                                  if current_mask is nomask:
2626: (12)                                      if mask is nomask:
2627: (16)                                          return
2628: (12)                                      current_mask = self._mask = make_mask_none(self.shape, idtype)
2629: (8)                                  if idtype.names is None:
2630: (12)                                      if self._hardmask:
2631: (16)                                          current_mask |= mask
2632: (12)                                      elif isinstance(mask, (int, float, np.bool_, np.number)):
2633: (16)                                          current_mask[...] = mask
2634: (12)                                      else:
2635: (16)                                          current_mask.flat = mask
2636: (8)                                  else:
2637: (12)                                      mdtype = current_mask.dtype
2638: (12)                                      mask = np.array(mask, copy=False)
2639: (12)                                      if not mask.ndim:
2640: (16)                                          if mask.dtype.kind == 'b':
2641: (20)                                              mask = np.array(tuple([mask.item()] * len(mdtype)),
2642: (36)                                                dtype=mdtype)
2643: (16)                                      else:
2644: (20)                                          mask = mask.astype(mdtype)
2645: (12)                                      else:
2646: (16)                                          try:
2647: (20)                                              mask = np.array(mask, copy=copy, dtype=mdtype)
2648: (16)                                          except TypeError:
2649: (20)                                              mask = np.array([tuple([m] * len(mdtype)) for m in mask],
2650: (36)                                                dtype=mdtype)
2651: (12)                                      if self._hardmask:
2652: (16)                                          for n in idtype.names:
2653: (20)                                              current_mask[n] |= mask[n]
2654: (12)                                          elif isinstance(mask, (int, float, np.bool_, np.number)):
2655: (16)                                              current_mask[...] = mask
2656: (12)                                          else:
2657: (16)                                              current_mask.flat = mask
2658: (8)                                      if current_mask.shape:
2659: (12)                                          current_mask.shape = self.shape
2660: (8)                                      return
2661: (4)                                _set_mask = __setmask__
2662: (4)                                @property
2663: (4)                                def mask(self):
2664: (8)                                  """ Current mask. """
2665: (8)                                  return self._mask.view()
2666: (4)                                @mask.setter
2667: (4)                                def mask(self, value):
2668: (8)                                  self.__setmask__(value)
2669: (4)                                @property
2670: (4)                                def recordmask(self):
2671: (8)                                  """
2672: (8)                                      Get or set the mask of the array if it has no named fields. For
2673: (8)                                      structured arrays, returns a ndarray of booleans where entries are
2674: (8)                                      ``True`` if **all** the fields are masked, ``False`` otherwise:
2675: (8)                                      >>> x = np.ma.array([(1, 1), (2, 2), (3, 3), (4, 4), (5, 5)],
2676: (8)                                            mask=[(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)],
```

```

2677: (8) ...
2678: (8)     dtype=[('a', int), ('b', int)])
2679: (8)     >>> x.recordmask
2680: (8)     array([False, False, True, False, False])
2681: (8)     """
2682: (8)     _mask = self._mask.view(ndarray)
2683: (12)     if _mask.dtype.names is None:
2684: (8)         return _mask
2685: (4)     return np.all(flatten_structured_array(_mask), axis=-1)
@recordmask.setter
2686: (4)     def recordmask(self, mask):
2687: (8)         raise NotImplementedError("Coming soon: setting the mask per
records!")
2688: (4)     def harden_mask(self):
2689: (8)         """
2690: (8)             Force the mask to hard, preventing unmasking by assignment.
2691: (8)             Whether the mask of a masked array is hard or soft is determined by
2692: (8)             its `~ma.MaskedArray.hardmask` property. `harden_mask` sets
2693: (8)             `~ma.MaskedArray.hardmask` to ``True`` (and returns the modified
2694: (8)             self).
2695: (8)             See Also
2696: (8)             -----
2697: (8)             ma.MaskedArray.hardmask
2698: (8)             ma.MaskedArray.soften_mask
2699: (8)             """
2700: (8)             self._hardmask = True
2701: (8)             return self
def soften_mask(self):
2702: (4)         """
2703: (8)             Force the mask to soft (default), allowing unmasking by assignment.
2704: (8)             Whether the mask of a masked array is hard or soft is determined by
2705: (8)             its `~ma.MaskedArray.hardmask` property. `soften_mask` sets
2706: (8)             `~ma.MaskedArray.hardmask` to ``False`` (and returns the modified
2707: (8)             self).
2708: (8)             See Also
2709: (8)             -----
2710: (8)             ma.MaskedArray.hardmask
2711: (8)             ma.MaskedArray.harden_mask
2712: (8)             """
2713: (8)             self._hardmask = False
2714: (8)             return self
@property
2715: (8)             def hardmask(self):
2716: (4)                 """
2717: (4)                     Specifies whether values can be unmasked through assignments.
2718: (8)                     By default, assigning definite values to masked array entries will
2719: (8)                     unmask them. When `hardmask` is ``True``, the mask will not change
2720: (8)                     through assignments.
2721: (8)                     See Also
2722: (8)                     -----
2723: (8)                     ma.MaskedArray.harden_mask
2724: (8)                     ma.MaskedArray.soften_mask
2725: (8)                     Examples
2726: (8)                     -----
2727: (8)                     >>> x = np.arange(10)
2728: (8)                     >>> m = np.ma.masked_array(x, x>5)
2729: (8)                     >>> assert not m.hardmask
2730: (8)                     Since `m` has a soft mask, assigning an element value unmasks that
2731: (8)                     element:
2732: (8)                     >>> m[8] = 42
2733: (8)                     >>> m
2734: (8)                     masked_array(data=[0, 1, 2, 3, 4, 5, --, --, 42, --],
2735: (8)                         mask=[False, False, False, False, False, False,
2736: (8)                             True, True, False, True],
2737: (21)                             fill_value=999999)
2738: (27)                     After hardening, the mask is not affected by assignments:
2739: (15)                     >>> hardened = np.ma.harden_mask(m)
2740: (8)                     >>> assert m.hardmask and hardened is m
2741: (8)                     >>> m[:] = 23
2742: (8)                     >>> m
2743: (8)
2744: (8)

```

```

2745: (8)                         masked_array(data=[23, 23, 23, 23, 23, 23, --, --, 23, --],
2746: (21)                           mask=[False, False, False, False, False, False,
2747: (27)                             True, True, False, True],
2748: (15)                           fill_value=999999)
2749: (8)                           """
2750: (8)                         return self._hardmask
2751: (4) def unshare_mask(self):
2752: (8)                           """
2753: (8)                         Copy the mask and set the `sharedmask` flag to ``False``.
2754: (8)                         Whether the mask is shared between masked arrays can be seen from
2755: (8)                         the `sharedmask` property. `unshare_mask` ensures the mask is not
2756: (8)                         shared. A copy of the mask is only made if it was shared.
2757: (8)                         See Also
2758: (8)                           -----
2759: (8)                         sharedmask
2760: (8)                           """
2761: (8)                         if self._sharedmask:
2762: (12)                           self._mask = self._mask.copy()
2763: (12)                           self._sharedmask = False
2764: (8)                         return self
2765: (4) @property
2766: (4) def sharedmask(self):
2767: (8)                           """ Share status of the mask (read-only). """
2768: (8)                           return self._sharedmask
2769: (4) def shrink_mask(self):
2770: (8)                           """
2771: (8)                         Reduce a mask to nomask when possible.
2772: (8)                         Parameters
2773: (8)                           -----
2774: (8)                           None
2775: (8)                         Returns
2776: (8)                           -----
2777: (8)                           None
2778: (8)                         Examples
2779: (8)                           -----
2780: (8)                         >>> x = np.ma.array([[1,2 ], [3, 4]], mask=[0]*4)
2781: (8)                         >>> x.mask
2782: (8)                         array([[False, False],
2783: (15)                           [False, False]])
2784: (8)                         >>> x.shrink_mask()
2785: (8)                         masked_array(
2786: (10)                           data=[[1, 2],
2787: (16)                             [3, 4]],
2788: (10)                           mask=False,
2789: (10)                           fill_value=999999)
2790: (8)                         >>> x.mask
2791: (8)                         False
2792: (8)                           """
2793: (8)                         self._mask = _shrink_mask(self._mask)
2794: (8)                         return self
2795: (4) @property
2796: (4) def baseclass(self):
2797: (8)                           """ Class of the underlying data (read-only). """
2798: (8)                           return self._baseclass
2799: (4) def _get_data(self):
2800: (8)                           """
2801: (8)                         Returns the underlying data, as a view of the masked array.
2802: (8)                         If the underlying data is a subclass of :class:`numpy.ndarray`, it is
2803: (8)                         returned as such.
2804: (8)                         >>> x = np.ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1,
2805: (8)                           0]])
2806: (8)                         >>> x.data
2807: (8)                           matrix([[1, 2],
2808: (16)                             [3, 4]])
2809: (8)                         The type of the data can be accessed through the :attr:`baseclass`
2810: (8)                         attribute.
2811: (8)                           """
2812: (4)                         return ndarray.view(self, self._baseclass)
2812: (4)                         _data = property(fget=_get_data)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2813: (4)             data = property(fget=_get_data)
2814: (4)             @property
2815: (4)             def flat(self):
2816: (8)                 """ Return a flat iterator, or set a flattened version of self to
value. """
2817: (8)                 return MaskedIterator(self)
2818: (4)             @flat.setter
2819: (4)             def flat(self, value):
2820: (8)                 y = self.ravel()
2821: (8)                 y[:] = value
2822: (4)             @property
2823: (4)             def fill_value(self):
2824: (8)                 """
2825: (8)                 The filling value of the masked array is a scalar. When setting, None
2826: (8)                 will set to a default based on the data type.
2827: (8)                 Examples
2828: (8)                 -----
2829: (8)                 >>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
2830: (8)                     ...     np.ma.array([0, 1], dtype=dt).get_fill_value()
2831: (8)                     ...
2832: (8)                     999999
2833: (8)                     999999
2834: (8)                     1e+20
2835: (8)                     (1e+20+0j)
2836: (8)                     >>> x = np.ma.array([0, 1.], fill_value=-np.inf)
2837: (8)                     >>> x.fill_value
2838: (8)                     -inf
2839: (8)                     >>> x.fill_value = np.pi
2840: (8)                     >>> x.fill_value
2841: (8)                     3.1415926535897931 # may vary
2842: (8)                     Reset to default:
2843: (8)                     >>> x.fill_value = None
2844: (8)                     >>> x.fill_value
2845: (8)                     1e+20
2846: (8)                     """
2847: (8)                     if self._fill_value is None:
2848: (12)                         self._fill_value = _check_fill_value(None, self.dtype)
2849: (8)                     if isinstance(self._fill_value, ndarray):
2850: (12)                         return self._fill_value[()]
2851: (8)                     return self._fill_value
2852: (4)             @fill_value.setter
2853: (4)             def fill_value(self, value=None):
2854: (8)                 target = _check_fill_value(value, self.dtype)
2855: (8)                 if not target.ndim == 0:
2856: (12)                     warnings.warn(
2857: (16)                         "Non-scalar arrays for the fill value are deprecated. Use "
2858: (16)                         "arrays with scalar values instead. The filled function "
2859: (16)                         "still supports any array as `fill_value`.",
2860: (16)                         DeprecationWarning, stacklevel=2)
2861: (8)                     _fill_value = self._fill_value
2862: (8)                     if _fill_value is None:
2863: (12)                         self._fill_value = target
2864: (8)                     else:
2865: (12)                         _fill_value[()] = target
2866: (4)                     get_fill_value = fill_value.fget
2867: (4)                     set_fill_value = fill_value.fset
2868: (4)                     def filled(self, fill_value=None):
2869: (8)                         """
2870: (8)                         Return a copy of self, with masked values filled with a given value.
2871: (8)                         **However**, if there are no masked values to fill, self will be
2872: (8)                         returned instead as an ndarray.
2873: (8)                         Parameters
2874: (8)                         -----
2875: (8)                         fill_value : array_like, optional
2876: (12)                             The value to use for invalid entries. Can be scalar or non-scalar.
2877: (12)                             If non-scalar, the resulting ndarray must be broadcastable over
2878: (12)                             input array. Default is None, in which case, the `fill_value`
2879: (12)                             attribute of the array is used instead.
2880: (8)                         Returns

```

```

2881: (8)          -----
2882: (8)          filled_array : ndarray
2883: (12)          A copy of ``self`` with invalid entries replaced by *fill_value*
2884: (12)          (be it the function argument or the attribute of ``self``), or
2885: (12)          ``self`` itself as an ndarray if there are no invalid entries to
2886: (12)          be replaced.
2887: (8)          Notes
2888: (8)          -----
2889: (8)          The result is **not** a MaskedArray!
2890: (8)          Examples
2891: (8)          -----
2892: (8)          >>> x = np.ma.array([1,2,3,4,5], mask=[0,0,1,0,1], fill_value=-999)
2893: (8)          >>> x.filled()
2894: (8)          array([ 1,  2, -999,   4, -999])
2895: (8)          >>> x.filled(fill_value=1000)
2896: (8)          array([ 1,  2, 1000,   4, 1000])
2897: (8)          >>> type(x.filled())
2898: (8)          <class 'numpy.ndarray'>
2899: (8)          Subclassing is preserved. This means that if, e.g., the data part of
2900: (8)          the masked array is a recarray, `filled` returns a recarray:
2901: (8)          >>> x = np.array([(-1, 2), (-3, 4)], dtype='i8,i8').view(np.recarray)
2902: (8)          >>> m = np.ma.array(x, mask=[(True, False), (False, True)])
2903: (8)          >>> m.filled()
2904: (8)          rec.array([(999999,           2), (      -3, 999999)],
2905: (18)          dtype=[('f0', '<i8'), ('f1', '<i8')])"
2906: (8)
2907: (8)          m = self._mask
2908: (8)          if m is nomask:
2909: (12)          return self._data
2910: (8)
2911: (12)          if fill_value is None:
2912: (8)          fill_value = self.fill_value
2913: (12)          else:
2914: (8)          fill_value = _check_fill_value(fill_value, self.dtype)
2915: (12)          if self is masked_singleton:
2916: (8)          return np.asarray(fill_value)
2917: (12)          if m.dtype.names is not None:
2918: (12)          result = self._data.copy('K')
2919: (8)          _recursive_filled(result, self._mask, fill_value)
2920: (12)          elif not m.any():
2921: (8)          return self._data
2922: (12)          else:
2923: (12)          result = self._data.copy('K')
2924: (16)          try:
2925: (12)          np.copyto(result, fill_value, where=m)
2926: (16)          except (TypeError, AttributeError):
2927: (16)          fill_value = narray(fill_value, dtype=object)
2928: (16)          d = result.astype(object)
2929: (12)          result = np.choose(m, (d, fill_value))
2930: (16)          except IndexError:
2931: (20)          if self._data.shape:
2932: (16)          raise
2933: (20)          elif m:
2934: (16)          result = np.array(fill_value, dtype=self.dtype)
2935: (20)          else:
2936: (8)          result = self._data
2937: (4)          return result
2938: (8)          def compressed(self):
2939: (8)          """
2940: (8)          Return all the non-masked data as a 1-D array.
2941: (8)          Returns
2942: (8)          -----
2943: (12)          data : ndarray
2944: (8)          A new `ndarray` holding the non-masked data is returned.
2945: (8)
2946: (8)          Notes
2947: (8)          -----
2948: (8)          The result is **not** a MaskedArray!
2949: (8)          Examples
2950: (8)          -----
2951: (8)          >>> x = np.ma.array(np.arange(5), mask=[0]*2 + [1]*3)

```

```

2950: (8)             >>> x.compressed()
2951: (8)             array([0, 1])
2952: (8)             >>> type(x.compressed())
2953: (8)             <class 'numpy.ndarray'>
2954: (8)             """
2955: (8)             data = ndarray.ravel(self._data)
2956: (8)             if self._mask is not nomask:
2957: (12)                 data = data.compress(np.logical_not(ndarray.ravel(self._mask)))
2958: (8)             return data
2959: (4)             def compress(self, condition, axis=None, out=None):
2960: (8)             """
2961: (8)                 Return `a` where condition is ``True``.
2962: (8)                 If condition is a `~ma.MaskedArray`, missing values are considered
2963: (8)                     as ``False``.
2964: (8)             Parameters
2965: (8)             -----
2966: (8)             condition : var
2967: (12)                 Boolean 1-d array selecting which entries to return. If
2968: (8)             len(condition)
2969: (12)             truncated
2970: (8)             to length of condition array.
2971: (12)             axis : {None, int}, optional
2972: (8)                 Axis along which the operation must be performed.
2973: (12)             out : {None, ndarray}, optional
2974: (8)                 Alternative output array in which to place the result. It must
2975: (12)             have
2976: (8)             the same shape as the expected output but the type will be cast if
2977: (8)             necessary.
2978: (8)             Returns
2979: (12)             -----
2980: (8)             result : MaskedArray
2981: (8)                 A :class:`~ma.MaskedArray` object.
2982: (8)             Notes
2983: (8)             -----
2984: (8)                 Please note the difference with :meth:`compressed` !
2985: (8)                 The output of :meth:`compress` has a mask, the output of
2986: (8)                 :meth:`compressed` does not.
2987: (8)             Examples
2988: (8)             -----
2989: (8)             >>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
2990: (10)            >>> x
2991: (16)            masked_array(
2992: (16)                data=[[1, --, 3],
2993: (10)                  [--, 5, --],
2994: (16)                  [7, --, 9]],
2995: (16)                  mask=[[False, True, False],
2996: (10)                    [True, False, True],
2997: (8)                      [False, True, False]],
2998: (8)                      fill_value=999999)
2999: (21)            >>> x.compress([1, 0, 1])
3000: (15)            masked_array(data=[1, 3],
3001: (8)                          mask=[False, False],
3002: (8)                          fill_value=999999)
3003: (10)            >>> x.compress([1, 0, 1], axis=1)
3004: (16)            masked_array(
3005: (16)                data=[[1, 3],
3006: (10)                  [--, --],
3007: (16)                  [7, 9]],
3008: (16)                  mask=[[False, False],
3009: (10)                    [True, True],
3010: (8)                      [False, False]],
3011: (8)                      fill_value=999999)
3012: (8)            """
3013: (8)            (_data, _mask) = (self._data, self._mask)
3014: (8)            condition = np.asarray(condition)
3015: (8)            _new = _data.compress(condition, axis=axis, out=out).view(type(self))
3016: (8)            _new._update_from(self)
3017: (8)            if _mask is not nomask:

```

```

3016: (12)             _new._mask = _mask.compress(condition, axis=axis)
3017: (8)              return _new
3018: (4)              def _insert_masked_print(self):
3019: (8)                  """
3020: (8)                      Replace masked values with masked_print_option, casting all innermost
3021: (8)                      dtypes to object.
3022: (8)                  """
3023: (8)              if masked_print_option.enabled():
3024: (12)                  mask = self._mask
3025: (12)                  if mask is nomask:
3026: (16)                      res = self._data
3027: (12)                  else:
3028: (16)                      data = self._data
3029: (16)                      print_width = (self._print_width if self.ndim > 1
3030: (31)                          else self._print_width_1d)
3031: (16)                      for axis in range(self.ndim):
3032: (20)                          if data.shape[axis] > print_width:
3033: (24)                              ind = print_width // 2
3034: (24)                              arr = np.split(data, (ind, -ind), axis=axis)
3035: (24)                              data = np.concatenate((arr[0], arr[2]), axis=axis)
3036: (24)                              arr = np.split(mask, (ind, -ind), axis=axis)
3037: (24)                              mask = np.concatenate((arr[0], arr[2]), axis=axis)
3038: (16)                      rdtype = _replace_dtype_fields(self.dtype, "O")
3039: (16)                      res = data.astype(rdtype)
3040: (16)                      _recursive_printoption(res, mask, masked_print_option)
3041: (8)                  else:
3042: (12)                      res = self.filled(self.fill_value)
3043: (8)              return res
3044: (4)              def __str__(self):
3045: (8)                  return str(self._insert_masked_print())
3046: (4)              def __repr__(self):
3047: (8)                  """
3048: (8)                      Literal string representation.
3049: (8)                  """
3050: (8)              if self._baseclass is np.ndarray:
3051: (12)                  name = 'array'
3052: (8)
3053: (12)              else:
3054: (8)                  name = self._baseclass.__name__
3055: (12)
3056: (12)
3057: (16)
3058: (16)
3059: (16)
3060: (16)
3061: (16)
3062: (16)
3063: (12)
3064: (12)
3065: (12)
3066: (16)
3067: (16)
3068: (12)
3069: (12)
3070: (8)
3071: (8)
3072: (12)
3073: (12)
3074: (12)
3075: (8)
3076: (8)
3077: (8)
3078: (12)
3079: (8)
3080: (8)
3081: (8)
3082: (12)
3083: (12)
3084: (12)
                    )
                    keys = ['data', 'mask', 'fill_value']
                    if dtype_needed:
                        keys.append('dtype')
                    is_one_row = builtins.all(dim == 1 for dim in self.shape[:-1])
                    min_indent = 2
                    if is_one_row:
                        indents = {}
                        indents[keys[0]] = prefix
                        for k in keys[1:]:
                            indents[k] = indents[keys[0]] + ' '
                    else:
                        indents = {}
                        for k in keys:
                            indents[k] = ' ' * min_indent
                    for k in keys:
                        if k == 'data' or k == 'mask':
                            v = getattr(self, k)
                            if v is None:
                                v = ''
                            if v == '':
                                v = 'None'
                            if k == 'data':
                                v = f'{v}:\n{v}'
```

```

3085: (16)             n = builtins.max(min_indent, len(prefix + keys[0]) - len(k))
3086: (16)             indents[k] = ' ' * n
3087: (12)             prefix = '' # absorbed into the first indent
3088: (8)
3089: (12)             else:
3090: (12)                 indents = {k: ' ' * min_indent for k in keys}
3091: (8)                 prefix = prefix + '\n' # first key on the next line
3092: (8)             reprs = {}
3093: (12)             reprs['data'] = np.array2string(
3094: (12)                 self._insert_masked_print(),
3095: (12)                 separator=", ",
3096: (12)                 prefix=indents['data'] + 'data=',
3097: (8)                 suffix=',')
3098: (12)             reprs['mask'] = np.array2string(
3099: (12)                 self._mask,
3100: (12)                 separator=", ",
3101: (12)                 prefix=indents['mask'] + 'mask=',
3102: (8)                 suffix=',')
3103: (8)             reprs['fill_value'] = repr(self.fill_value)
3104: (12)             if dtype_needed:
3105: (8)                 reprs['dtype'] = np.core.arrayprint.dtype_short_repr(self.dtype)
3106: (12)             result = ',\n'.join(
3107: (12)                 '{}{}={}'.format(indents[k], k, reprs[k])
3108: (8)                 for k in keys
3109: (8)
3110: (4)             )
3111: (8)             return prefix + result + ')'
3112: (12)         def _delegate_binop(self, other):
3113: (8)             if isinstance(other, type(self)):
3114: (8)                 return False
3115: (12)             array_ufunc = getattr(other, "__array_ufunc__", False)
3116: (12)             if array_ufunc is False:
3117: (8)                 other_priority = getattr(other, "__array_priority__", -1000000)
3118: (12)                 return self.__array_priority__ < other_priority
3119: (4)             else:
3120: (8)                 return array_ufunc is None
3121: (8)         def _comparison(self, other, compare):
3122: (8)             """Compare self with other using operator.eq or operator.ne.
3123: (8)             When either of the elements is masked, the result is masked as well,
3124: (8)             but the underlying boolean data are still set, with self and other
3125: (8)             considered equal if both are masked, and unequal otherwise.
3126: (8)             For structured arrays, all fields are combined, with masked values
3127: (8)             ignored. The result is masked if all fields were masked, with self
3128: (8)             and other considered equal only if both were fully masked.
3129: (8)
3130: (8)             omask = getmask(other)
3131: (8)             smask = self.mask
3132: (8)             mask = mask_or(smask, omask, copy=True)
3133: (12)             odata = getdata(other)
3134: (16)             if mask.dtype.names is not None:
3135: (12)                 if compare not in (operator.eq, operator.ne):
3136: (12)                     return NotImplemented
3137: (12)                 broadcast_shape = np.broadcast(self, odata).shape
3138: (12)                 sbroadcast = np.broadcast_to(self, broadcast_shape, subok=True)
3139: (12)                 sbroadcast._mask = mask
3140: (12)                 sdata = sbroadcast.filled(odata)
3141: (16)                 mask = (mask == np.ones(()), mask.dtype))
3142: (8)                 if omask is np.False_:
3143: (12)                     omask = np.zeros(()), smask.dtype)
3144: (8)
3145: (8)             else:
3146: (12)                 sdata = self.data
3147: (8)             check = compare(sdata, odata)
3148: (12)             if isinstance(check, (np.bool_, bool)):
3149: (16)                 return masked if mask else check
3150: (12)             if mask is not nomask:
3151: (16)                 if compare in (operator.eq, operator.ne):
3152: (8)                     check = np.where(mask, compare(smask, omask), check)
3153: (8)                     if mask.shape != check.shape:
3154: (12)                         mask = np.broadcast_to(mask, check.shape).copy()
3155: (16)                     check = check.view(type(self))
3156: (8)                     check._update_from(self)

```

```

3154: (8)             check._mask = mask
3155: (8)             if check._fill_value is not None:
3156: (12)             try:
3157: (16)                 fill = _check_fill_value(check._fill_value, np.bool_)
3158: (12)             except (TypeError, ValueError):
3159: (16)                 fill = _check_fill_value(None, np.bool_)
3160: (12)                 check._fill_value = fill
3161: (8)             return check
3162: (4)             def __eq__(self, other):
3163: (8)                 """Check whether other equals self elementwise.
3164: (8)                 When either of the elements is masked, the result is masked as well,
3165: (8)                 but the underlying boolean data are still set, with self and other
3166: (8)                 considered equal if both are masked, and unequal otherwise.
3167: (8)                 For structured arrays, all fields are combined, with masked values
3168: (8)                 ignored. The result is masked if all fields were masked, with self
3169: (8)                 and other considered equal only if both were fully masked.
3170: (8)                 """
3171: (8)                 return self._comparison(other, operator.eq)
3172: (4)             def __ne__(self, other):
3173: (8)                 """Check whether other does not equal self elementwise.
3174: (8)                 When either of the elements is masked, the result is masked as well,
3175: (8)                 but the underlying boolean data are still set, with self and other
3176: (8)                 considered equal if both are masked, and unequal otherwise.
3177: (8)                 For structured arrays, all fields are combined, with masked values
3178: (8)                 ignored. The result is masked if all fields were masked, with self
3179: (8)                 and other considered equal only if both were fully masked.
3180: (8)                 """
3181: (8)                 return self._comparison(other, operator.ne)
3182: (4)             def __le__(self, other):
3183: (8)                 return self._comparison(other, operator.le)
3184: (4)             def __lt__(self, other):
3185: (8)                 return self._comparison(other, operator.lt)
3186: (4)             def __ge__(self, other):
3187: (8)                 return self._comparison(other, operator.ge)
3188: (4)             def __gt__(self, other):
3189: (8)                 return self._comparison(other, operator.gt)
3190: (4)             def __add__(self, other):
3191: (8)                 """
3192: (8)                 Add self to other, and return a new masked array.
3193: (8)                 """
3194: (8)                 if self._delegate_binop(other):
3195: (12)                     return NotImplemented
3196: (8)                     return add(self, other)
3197: (4)             def __radd__(self, other):
3198: (8)                 """
3199: (8)                 Add other to self, and return a new masked array.
3200: (8)                 """
3201: (8)                 return add(other, self)
3202: (4)             def __sub__(self, other):
3203: (8)                 """
3204: (8)                 Subtract other from self, and return a new masked array.
3205: (8)                 """
3206: (8)                 if self._delegate_binop(other):
3207: (12)                     return NotImplemented
3208: (8)                     return subtract(self, other)
3209: (4)             def __rsub__(self, other):
3210: (8)                 """
3211: (8)                 Subtract self from other, and return a new masked array.
3212: (8)                 """
3213: (8)                 return subtract(other, self)
3214: (4)             def __mul__(self, other):
3215: (8)                 "Multiply self by other, and return a new masked array."
3216: (8)                 if self._delegate_binop(other):
3217: (12)                     return NotImplemented
3218: (8)                     return multiply(self, other)
3219: (4)             def __rmul__(self, other):
3220: (8)                 """
3221: (8)                 Multiply other by self, and return a new masked array.
3222: (8)                 """

```

```

3223: (8)             return multiply(other, self)
3224: (4)             def __div__(self, other):
3225: (8)                 """
3226: (8)                     Divide other into self, and return a new masked array.
3227: (8)                     """
3228: (8)                     if self._delegate_binop(other):
3229: (12)                         return NotImplemented
3230: (8)                     return divide(self, other)
3231: (4)             def __truediv__(self, other):
3232: (8)                 """
3233: (8)                     Divide other into self, and return a new masked array.
3234: (8)                     """
3235: (8)                     if self._delegate_binop(other):
3236: (12)                         return NotImplemented
3237: (8)                     return true_divide(self, other)
3238: (4)             def __rtruediv__(self, other):
3239: (8)                 """
3240: (8)                     Divide self into other, and return a new masked array.
3241: (8)                     """
3242: (8)                     return true_divide(other, self)
3243: (4)             def __floordiv__(self, other):
3244: (8)                 """
3245: (8)                     Divide other into self, and return a new masked array.
3246: (8)                     """
3247: (8)                     if self._delegate_binop(other):
3248: (12)                         return NotImplemented
3249: (8)                     return floor_divide(self, other)
3250: (4)             def __rfloordiv__(self, other):
3251: (8)                 """
3252: (8)                     Divide self into other, and return a new masked array.
3253: (8)                     """
3254: (8)                     return floor_divide(other, self)
3255: (4)             def __pow__(self, other):
3256: (8)                 """
3257: (8)                     Raise self to the power other, masking the potential NaNs/Infs
3258: (8)                     """
3259: (8)                     if self._delegate_binop(other):
3260: (12)                         return NotImplemented
3261: (8)                     return power(self, other)
3262: (4)             def __rpow__(self, other):
3263: (8)                 """
3264: (8)                     Raise other to the power self, masking the potential NaNs/Infs
3265: (8)                     """
3266: (8)                     return power(other, self)
3267: (4)             def __iadd__(self, other):
3268: (8)                 """
3269: (8)                     Add other to self in-place.
3270: (8)                     """
3271: (8)                     m = getmask(other)
3272: (8)                     if self._mask is nomask:
3273: (12)                         if m is not nomask and m.any():
3274: (16)                             self._mask = make_mask_none(self.shape, self.dtype)
3275: (16)                             self._mask += m
3276: (8)                     else:
3277: (12)                         if m is not nomask:
3278: (16)                             self._mask += m
3279: (8)                     other_data = getdata(other)
3280: (8)                     other_data = np.where(self._mask, other_data.dtype.type(0),
other_data)
3281: (8)                     self._data.__iadd__(other_data)
3282: (8)                     return self
3283: (4)             def __isub__(self, other):
3284: (8)                 """
3285: (8)                     Subtract other from self in-place.
3286: (8)                     """
3287: (8)                     m = getmask(other)
3288: (8)                     if self._mask is nomask:
3289: (12)                         if m is not nomask and m.any():
3290: (16)                             self._mask = make_mask_none(self.shape, self.dtype)

```

```

3291: (16)                     self._mask += m
3292: (8)                      elif m is not nomask:
3293: (12)                        self._mask += m
3294: (8)                      other_data = getdata(other)
3295: (8)                      other_data = np.where(self._mask, other_data.dtype.type(0),
other_data)
3296: (8)                          self._data.__isub__(other_data)
3297: (8)                      return self
3298: (4)                      def __imul__(self, other):
3299: (8)                        """
3300: (8)                            Multiply self by other in-place.
3301: (8)                        """
3302: (8)                        m = getmask(other)
3303: (8)                        if self._mask is nomask:
3304: (12)                          if m is not nomask and m.any():
3305: (16)                            self._mask = make_mask_none(self.shape, self.dtype)
3306: (16)                            self._mask += m
3307: (8)                        elif m is not nomask:
3308: (12)                          self._mask += m
3309: (8)                        other_data = getdata(other)
3310: (8)                        other_data = np.where(self._mask, other_data.dtype.type(1),
other_data)
3311: (8)                          self._data.__imul__(other_data)
3312: (8)                      return self
3313: (4)                      def __idiv__(self, other):
3314: (8)                        """
3315: (8)                            Divide self by other in-place.
3316: (8)                        """
3317: (8)                        other_data = getdata(other)
3318: (8)                        dom_mask = _DomainSafeDivide().__call__(self._data, other_data)
3319: (8)                        other_mask = getmask(other)
3320: (8)                        new_mask = mask_or(other_mask, dom_mask)
3321: (8)                        if dom_mask.any():
3322: (12)                          (_, fval) = ufunc_fills[np.divide]
3323: (12)                          other_data = np.where(
3324: (20)                            dom_mask, other_data.dtype.type(fval), other_data)
3325: (8)                        self._mask |= new_mask
3326: (8)                        other_data = np.where(self._mask, other_data.dtype.type(1),
other_data)
3327: (8)                          self._data.__idiv__(other_data)
3328: (8)                      return self
3329: (4)                      def __ifloordiv__(self, other):
3330: (8)                        """
3331: (8)                            Floor divide self by other in-place.
3332: (8)                        """
3333: (8)                        other_data = getdata(other)
3334: (8)                        dom_mask = _DomainSafeDivide().__call__(self._data, other_data)
3335: (8)                        other_mask = getmask(other)
3336: (8)                        new_mask = mask_or(other_mask, dom_mask)
3337: (8)                        if dom_mask.any():
3338: (12)                          (_, fval) = ufunc_fills[np.floor_divide]
3339: (12)                          other_data = np.where(
3340: (20)                            dom_mask, other_data.dtype.type(fval), other_data)
3341: (8)                        self._mask |= new_mask
3342: (8)                        other_data = np.where(self._mask, other_data.dtype.type(1),
other_data)
3343: (8)                          self._data.__ifloordiv__(other_data)
3344: (8)                      return self
3345: (4)                      def __itruediv__(self, other):
3346: (8)                        """
3347: (8)                            True divide self by other in-place.
3348: (8)                        """
3349: (8)                        other_data = getdata(other)
3350: (8)                        dom_mask = _DomainSafeDivide().__call__(self._data, other_data)
3351: (8)                        other_mask = getmask(other)
3352: (8)                        new_mask = mask_or(other_mask, dom_mask)
3353: (8)                        if dom_mask.any():
3354: (12)                          (_, fval) = ufunc_fills[np.true_divide]
3355: (12)                          other_data = np.where(

```

```

3356: (20)                                dom_mask, other_data.dtype.type(fval), other_data)
3357: (8)                                 self._mask |= new_mask
3358: (8)                                 other_data = np.where(self._mask, other_data.dtype.type(1),
3359: (8)                                         self._data.__itruediv__(other_data)
3360: (8)                                         return self
3361: (4)                                 def __ipow__(self, other):
3362: (8)                                     """
3363: (8)                                         Raise self to the power other, in place.
3364: (8)                                     """
3365: (8)                                 other_data = getdata(other)
3366: (8)                                 other_data = np.where(self._mask, other_data.dtype.type(1),
3367: (8)                                         other_data = getmask(other)
3368: (8)                                         with np.errstate(divide='ignore', invalid='ignore'):
3369: (12)                                             self._data.__ipow__(other_data)
3370: (8)                                         invalid = np.logical_not(np.isfinite(self._data))
3371: (8)                                         if invalid.any():
3372: (12)                                             if self._mask is not nomask:
3373: (16)                                                 self._mask |= invalid
3374: (12)                                             else:
3375: (16)                                                 self._mask = invalid
3376: (12)                                                 np.copyto(self._data, self.fill_value, where=invalid)
3377: (8)                                         new_mask = mask_or(other_mask, invalid)
3378: (8)                                         self._mask = mask_or(self._mask, new_mask)
3379: (8)                                         return self
3380: (4)                                 def __float__(self):
3381: (8)                                     """
3382: (8)                                         Convert to float.
3383: (8)                                     """
3384: (8)                                         if self.size > 1:
3385: (12)                                             raise TypeError("Only length-1 arrays can be converted "
3386: (28)                                                 "to Python scalars")
3387: (8)                                         elif self._mask:
3388: (12)                                             warnings.warn("Warning: converting a masked element to nan.",
stacklevel=2)
3389: (12)                                         return np.nan
3390: (8)                                         return float(self.item())
3391: (4)                                 def __int__(self):
3392: (8)                                     """
3393: (8)                                         Convert to int.
3394: (8)                                     """
3395: (8)                                         if self.size > 1:
3396: (12)                                             raise TypeError("Only length-1 arrays can be converted "
3397: (28)                                                 "to Python scalars")
3398: (8)                                         elif self._mask:
3399: (12)                                             raise MaskError('Cannot convert masked element to a Python int.')
3400: (8)                                         return int(self.item())
3401: (4) @property
3402: (4)                                 def imag(self):
3403: (8)                                     """
3404: (8)                                         The imaginary part of the masked array.
3405: (8)                                         This property is a view on the imaginary part of this `MaskedArray`.
3406: (8)                                         See Also
3407: (8)                                         -----
3408: (8)                                         real
3409: (8)                                         Examples
3410: (8)                                         -----
3411: (8)                                         >>> x = np.ma.array([1+1.j, -2j, 3.45+1.6j], mask=[False, True,
False])
3412: (8)                                         >>> x.imag
3413: (8)                                         masked_array(data=[1.0, --, 1.6],
3414: (21)                                             mask=[False, True, False],
3415: (15)                                             fill_value=1e+20)
3416: (8)                                         """
3417: (8)                                         result = self._data.imag.view(type(self))
3418: (8)                                         result.__setmask__(self._mask)
3419: (8)                                         return result
3420: (4)                                         get_imag = imag.fget

```

```

3421: (4) @property
3422: (4) def real(self):
3423: (8)     """
3424: (8)         The real part of the masked array.
3425: (8)         This property is a view on the real part of this `MaskedArray`.
3426: (8)         See Also
3427: (8)             -----
3428: (8)             imag
3429: (8)             Examples
3430: (8)             -----
3431: (8)             >>> x = np.ma.array([1+1.j, -2j, 3.45+1.6j], mask=[False, True,
3432: (8)             False])
3433: (8)             >>> x.real
3434: (21)             masked_array(data=[1.0, --, 3.45],
3435: (15)                 mask=[False, True, False],
3436: (8)                 fill_value=1e+20)
3437: (8)             """
3438: (8)             result = self._data.real.view(type(self))
3439: (8)             result.__setmask__(self._mask)
3440: (8)             return result
3441: (4)             get_real = real.fget
3442: (8)             def count(self, axis=None, keepdims=np._NoValue):
3443: (8)                 """
3444: (8)                     Count the non-masked elements of the array along the given axis.
3445: (8)                     Parameters
3446: (8)                     -----
3447: (12)                     axis : None or int or tuple of ints, optional
3448: (12)                         Axis or axes along which the count is performed.
3449: (12)                         The default, None, performs the count over all
3450: (12)                         the dimensions of the input array. `axis` may be negative, in
3451: (12)                         which case it counts from the last to the first axis.
3452: (12)                         .. versionadded:: 1.10.0
3453: (12)                         If this is a tuple of ints, the count is performed on multiple
3454: (8)                         axes, instead of a single axis or all the axes as before.
3455: (12)             keepdims : bool, optional
3456: (12)                 If this is set to True, the axes which are reduced are left
3457: (12)                 in the result as dimensions with size one. With this option,
3458: (8)                 the result will broadcast correctly against the array.
3459: (8)             Returns
3460: (8)             -----
3461: (12)             result : ndarray or scalar
3462: (8)                 An array with the same shape as the input array, with the
3463: (12)                 axis removed. If the array is a 0-d array, or if `axis` is None, a
3464: (8)                 scalar is returned.
3465: (8)             See Also
3466: (8)             -----
3467: (12)             ma.count_masked : Count masked elements in array or along a given
3468: (8)             Examples
3469: (8)             -----
3470: (8)             >>> import numpy.ma as ma
3471: (8)             >>> a = ma.arange(6).reshape((2, 3))
3472: (8)             >>> a[1, :] = ma.masked
3473: (8)             >>> a
3474: (10)             masked_array(
3475: (16)                 data=[[0, 1, 2],
3476: (10)                     [--, --, --]],
3477: (16)                     mask=[[False, False, False],
3478: (10)                         [True, True, True]],
3479: (8)                         fill_value=999999)
3480: (8)             >>> a.count()
3481: (8)             3
3482: (8)             When the `axis` keyword is specified an array of appropriate size is
3483: (8)             returned.
3484: (8)             >>> a.count(axis=0)
3485: (8)             array([1, 1, 1])
3486: (8)             >>> a.count(axis=1)
3487: (8)             array([3, 0])

```

```

3487: (8) """
3488: (8)     kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
3489: (8)     m = self._mask
3490: (8)     if isinstance(self.data, np.matrix):
3491: (12)         if m is nomask:
3492: (16)             m = np.zeros(self.shape, dtype=np.bool_)
3493: (12)             m = m.view(type(self.data))
3494: (8)
3495: (12)     if m is nomask:
3496: (16)         if self.shape == ():
3497: (20)             if axis not in (None, 0):
3498: (16)                 raise np.AxisError(axis=axis, ndim=self.ndim)
3499: (12)             return 1
3500: (16)         elif axis is None:
3501: (20)             if kwargs.get('keepdims', False):
3502: (16)                 return np.array(self.size, dtype=np.intp, ndmin=self.ndim)
3503: (12)             return self.size
3504: (12)         axes = normalize_axis_tuple(axis, self.ndim)
3505: (12)         items = 1
3506: (16)         for ax in axes:
3507: (12)             items *= self.shape[ax]
3508: (16)         if kwargs.get('keepdims', False):
3509: (16)             out_dims = list(self.shape)
3510: (20)             for a in axes:
3511: (12)                 out_dims[a] = 1
3512: (16)         else:
3513: (28)             out_dims = [d for n, d in enumerate(self.shape)
3514: (12)                         if n not in axes]
3515: (8)
3516: (12)         return np.full(out_dims, items, dtype=np.intp)
3517: (8)
3518: (4)     def ravel(self, order='C'):
3519: (8)         """
3520: (8)         Returns a 1D version of self, as a view.
3521: (8)         Parameters
3522: (8)         -----
3523: (8)         order : {'C', 'F', 'A', 'K'}, optional
3524: (12)             The elements of `a` are read using this index order. 'C' means to
3525: (12)             index the elements in C-like order, with the last axis index
3526: (12)             changing fastest, back to the first axis index changing slowest.
3527: (12)             'F' means to index the elements in Fortran-like index order, with
3528: (12)             the first index changing fastest, and the last index changing
3529: (12)             slowest. Note that the 'C' and 'F' options take no account of the
3530: (12)             memory layout of the underlying array, and only refer to the order
3531: (12)             of axis indexing. 'A' means to read the elements in Fortran-like
3532: (12)             index order if `m` is Fortran *contiguous* in memory, C-like order
3533: (12)             otherwise. 'K' means to read the elements in the order they occur
3534: (12)             in memory, except for reversing the data when strides are
negative.
3535: (12)             By default, 'C' index order is used.
3536: (12)             (Masked arrays currently use 'A' on the data when 'K' is passed.)
3537: (8)         Returns
3538: (8)         -----
3539: (8)         MaskedArray
3540: (12)             Output view is of shape ``self.size,)`` (or
3541: (12)             ``np.ma.product(self.shape),``).
3542: (8)
3543: (8)         Examples
3544: (8)         >>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
3545: (8)
3546: (8)         masked_array(
3547: (10)             data=[[1, --, 3],
3548: (16)                 [--, 5, --],
3549: (16)                 [7, --, 9]],
3550: (10)             mask=[[False, True, False],
3551: (16)                 [True, False, True],
3552: (16)                 [False, True, False]],
3553: (10)             fill_value=999999)
3554: (8)         >>> x.ravel()

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

3555: (8)             masked_array(data=[1, --, 3, --, 5, --, 7, --, 9],
3556: (21)             mask=[False, True, False, True, False, True, False,
3557: (27)                 False],
3558: (15)                 fill_value=999999)
3559: (8)             """
3560: (8)             if order in "kKaA":
3561: (12)                 order = "F" if self._data.flags.fnc else "C"
3562: (8)             r = ndarray.ravel(self._data, order=order).view(type(self))
3563: (8)             r._update_from(self)
3564: (8)             if self._mask is not nomask:
3565: (12)                 r._mask = ndarray.ravel(self._mask, order=order).reshape(r.shape)
3566: (8)             else:
3567: (12)                 r._mask = nomask
3568: (8)             return r
3569: (4)             def reshape(self, *s, **kwargs):
3570: (8)             """
3571: (8)             Give a new shape to the array without changing its data.
3572: (8)             Returns a masked array containing the same data, but with a new shape.
3573: (8)             The result is a view on the original array; if this is not possible, a
3574: (8)             ValueError is raised.
3575: (8)             Parameters
3576: (8)             -----
3577: (8)             shape : int or tuple of ints
3578: (12)                 The new shape should be compatible with the original shape. If an
3579: (12)                 integer is supplied, then the result will be a 1-D array of that
3580: (12)                 length.
3581: (8)             order : {'C', 'F'}, optional
3582: (12)                 Determines whether the array data should be viewed as in C
3583: (12)                 (row-major) or FORTRAN (column-major) order.
3584: (8)             Returns
3585: (8)             -----
3586: (8)             reshaped_array : array
3587: (12)                 A new view on the array.
3588: (8)             See Also
3589: (8)             -----
3590: (8)             reshape : Equivalent function in the masked array module.
3591: (8)             numpy.ndarray.reshape : Equivalent method on ndarray object.
3592: (8)             numpy.reshape : Equivalent function in the NumPy module.
3593: (8)             Notes
3594: (8)             -----
3595: (8)             The reshaping operation cannot guarantee that a copy will not be made,
3596: (8)             to modify the shape in place, use ``a.shape = s``
3597: (8)             Examples
3598: (8)             -----
3599: (8)             >>> x = np.ma.array([[1,2],[3,4]], mask=[1,0,0,1])
3600: (8)             >>> x
3601: (8)             masked_array(
3602: (10)                 data=[[--, 2],
3603: (16)                     [3, --]],
3604: (10)                 mask=[[ True, False],
3605: (16)                     [False,  True]],
3606: (10)                 fill_value=999999)
3607: (8)             >>> x = x.reshape((4,1))
3608: (8)             >>> x
3609: (8)             masked_array(
3610: (10)                 data=[[--],
3611: (16)                     [2],
3612: (16)                     [3],
3613: (16)                     [--]],
3614: (10)                 mask=[[ True],
3615: (16)                     [False],
3616: (16)                     [False],
3617: (16)                     [ True]],
3618: (10)                 fill_value=999999)
3619: (8)             """
3620: (8)             kwargs.update(order=kwargs.get('order', 'C'))
3621: (8)             result = self._data.reshape(*s, **kwargs).view(type(self))
3622: (8)             result._update_from(self)

```

```

3623: (8)             mask = self._mask
3624: (8)             if mask is not nomask:
3625: (12)             result._mask = mask.reshape(*s, **kwargs)
3626: (8)             return result
3627: (4)             def resize(self, newshape, refcheck=True, order=False):
3628: (8)                 """
3629: (8)                 .. warning::
3630: (12)                     This method does nothing, except raise a ValueError exception. A
3631: (12)                     masked array does not own its data and therefore cannot safely be
3632: (12)                     resized in place. Use the `numpy.ma.resize` function instead.
3633: (8)             This method is difficult to implement safely and may be deprecated in
3634: (8)             future releases of NumPy.
3635: (8)                 """
3636: (8)             errmsg = "A masked array does not own its data \"\
3637: (17)                 \"and therefore cannot be resized.\n\" \
3638: (17)                 \"Use the numpy.ma.resize function instead.\""
3639: (8)             raise ValueError(errmsg)
3640: (4)             def put(self, indices, values, mode='raise'):
3641: (8)                 """
3642: (8)                 Set storage-indexed locations to corresponding values.
3643: (8)                 Sets self._data.flat[n] = values[n] for each n in indices.
3644: (8)                 If `values` is shorter than `indices` then it will repeat.
3645: (8)                 If `values` has some masked values, the initial mask is updated
3646: (8)                 in consequence, else the corresponding values are unmasked.
3647: (8)                 Parameters
3648: (8)                 -----
3649: (8)                 indices : 1-D array_like
3650: (12)                   Target indices, interpreted as integers.
3651: (8)                 values : array_like
3652: (12)                   Values to place in self._data copy at target indices.
3653: (8)                 mode : {'raise', 'wrap', 'clip'}, optional
3654: (12)                   Specifies how out-of-bounds indices will behave.
3655: (12)                   'raise' : raise an error.
3656: (12)                   'wrap' : wrap around.
3657: (12)                   'clip' : clip to the range.
3658: (8)                 Notes
3659: (8)                 -----
3660: (8)                 `values` can be a scalar or length 1 array.
3661: (8)                 Examples
3662: (8)                 -----
3663: (8)                 >>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
3664: (8)
3665: (8)                 masked_array(
3666: (10)                   data=[[1, --, 3],
3667: (16)                       [--, 5, --],
3668: (16)                       [7, --, 9]],
3669: (10)                   mask=[[False, True, False],
3670: (16)                       [True, False, True],
3671: (16)                       [False, True, False]],
3672: (10)                   fill_value=999999)
3673: (8)
3674: (8)
3675: (8)                 masked_array(
3676: (10)                   data=[[10, --, 3],
3677: (16)                       [--, 20, --],
3678: (16)                       [7, --, 30]],
3679: (10)                   mask=[[False, True, False],
3680: (16)                       [True, False, True],
3681: (16)                       [False, True, False]],
3682: (10)                   fill_value=999999)
3683: (8)
3684: (8)
3685: (8)                 masked_array(
3686: (10)                   data=[[10, --, 3],
3687: (16)                       [--, 999, --],
3688: (16)                       [7, --, 30]],
3689: (10)                   mask=[[False, True, False],
3690: (16)                       [True, False, True],
3691: (16)                       [False, True, False]],

```

```

3692: (10)           fill_value=999999)
3693: (8)           """
3694: (8)           if self._hardmask and self._mask is not nomask:
3695: (12)             mask = self._mask[indices]
3696: (12)             indices = narray(indices, copy=False)
3697: (12)             values = narray(values, copy=False, subok=True)
3698: (12)             values.resize(indices.shape)
3699: (12)             indices = indices[~mask]
3700: (12)             values = values[~mask]
3701: (8)             self._data.put(indices, values, mode=mode)
3702: (8)           if self._mask is nomask and getmask(values) is nomask:
3703: (12)             return
3704: (8)             m = getmaskarray(self)
3705: (8)           if getmask(values) is nomask:
3706: (12)             m.put(indices, False, mode=mode)
3707: (8)           else:
3708: (12)             m.put(indices, values._mask, mode=mode)
3709: (8)           m = make_mask(m, copy=False, shrink=True)
3710: (8)           self._mask = m
3711: (8)           return
3712: (4)           def ids(self):
3713: (8)             """
3714: (8)             Return the addresses of the data and mask areas.
3715: (8)             Parameters
3716: (8)             -----
3717: (8)             None
3718: (8)             Examples
3719: (8)             -----
3720: (8)             >>> x = np.ma.array([1, 2, 3], mask=[0, 1, 1])
3721: (8)             >>> x.ids()
3722: (8)             (166670640, 166659832) # may vary
3723: (8)             If the array has no mask, the address of `nomask` is returned. This
address
3724: (8)             is typically not close to the data in memory:
3725: (8)             >>> x = np.ma.array([1, 2, 3])
3726: (8)             >>> x.ids()
3727: (8)             (166691080, 3083169284) # may vary
3728: (8)             """
3729: (8)           if self._mask is nomask:
3730: (12)             return (self.ctypes.data, id(nomask))
3731: (8)           return (self.ctypes.data, self._mask.ctypes.data)
3732: (4)           def iscontiguous(self):
3733: (8)             """
3734: (8)             Return a boolean indicating whether the data is contiguous.
3735: (8)             Parameters
3736: (8)             -----
3737: (8)             None
3738: (8)             Examples
3739: (8)             -----
3740: (8)             >>> x = np.ma.array([1, 2, 3])
3741: (8)             >>> x.iscontiguous()
3742: (8)             True
3743: (8)             `iscontiguous` returns one of the flags of the masked array:
3744: (8)             >>> x.flags
3745: (10)               C_CONTIGUOUS : True
3746: (10)               F_CONTIGUOUS : True
3747: (10)               OWNDATA : False
3748: (10)               WRITEABLE : True
3749: (10)               ALIGNED : True
3750: (10)               WRITEBACKIFCOPY : False
3751: (8)             """
3752: (8)             return self.flags['CONTIGUOUS']
3753: (4)           def all(self, axis=None, out=None, keepdims=np._NoValue):
3754: (8)             """
3755: (8)             Returns True if all elements evaluate to True.
3756: (8)             The output array is masked where all the values along the given axis
3757: (8)             are masked: if the output would have been a scalar and that all the
3758: (8)             values are masked, then the output is `masked`.
3759: (8)             Refer to `numpy.all` for full documentation.

```

```

3760: (8)             See Also
3761: (8)
3762: (8)             -----
3763: (8)             numpy.ndarray.all : corresponding function for ndarrays
3764: (8)             numpy.all : equivalent function
3765: (8)
3766: (8)             Examples
3767: (8)             -----
3768: (8)             >>> np.ma.array([1,2,3]).all()
3769: (8)             True
3770: (8)             >>> a = np.ma.array([1,2,3], mask=True)
3771: (8)             >>> (a.all() is np.ma.masked)
3772: (8)             True
3773: (8)             """
3774: (8)             kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
3775: (12)            mask = _check_mask_axis(self._mask, axis, **kwargs)
3776: (12)            if out is None:
3777: (16)              d = self.filled(True).all(axis=axis, **kwargs).view(type(self))
3778: (12)              if d.ndim:
3779: (16)                d.__setmask__(mask)
3780: (12)              elif mask:
3781: (16)                return masked
3782: (12)              return d
3783: (8)              self.filled(True).all(axis=axis, out=out, **kwargs)
3784: (16)            if isinstance(out, MaskedArray):
3785: (8)              if out.ndim or mask:
3786: (12)                out.__setmask__(mask)
3787: (8)            return out
3788: (4)             def any(self, axis=None, out=None, keepdims=np._NoValue):
3789: (8)             """
3790: (8)             Returns True if any of the elements of `a` evaluate to True.
3791: (8)             Masked values are considered as False during computation.
3792: (8)             Refer to `numpy.any` for full documentation.
3793: (8)             See Also
3794: (8)
3795: (8)             -----
3796: (8)             numpy.ndarray.any : corresponding function for ndarrays
3797: (8)             numpy.any : equivalent function
3798: (8)             """
3799: (12)            kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
3800: (12)            mask = _check_mask_axis(self._mask, axis, **kwargs)
3801: (16)            if out is None:
3802: (12)              d = self.filled(False).any(axis=axis, **kwargs).view(type(self))
3803: (12)              if d.ndim:
3804: (16)                d.__setmask__(mask)
3805: (8)              elif mask:
3806: (12)                d = masked
3807: (12)              return d
3808: (16)            self.filled(False).any(axis=axis, out=out, **kwargs)
3809: (8)            if isinstance(out, MaskedArray):
3810: (12)              if out.ndim or mask:
3811: (8)                out.__setmask__(mask)
3812: (8)            return out
3813: (4)             def nonzero(self):
3814: (8)             """
3815: (8)             Returns the indices of unmasked elements that are not zero.
3816: (8)             Returns a tuple of arrays, one for each dimension, containing the
3817: (8)             indices of the non-zero elements in that dimension. The corresponding
3818: (8)             non-zero values can be obtained with::
3819: (12)               a[a.nonzero()]
3820: (8)             To group the indices by element, rather than dimension, use
3821: (8)             instead::
3822: (12)               np.transpose(a.nonzero())
3823: (8)             The result of this is always a 2d array, with a row for each non-zero
3824: (8)             element.
3825: (8)             Parameters
3826: (8)             -----
3827: (8)             None
3828: (8)             Returns
3829: (8)             -----
3830: (8)             tuple_of_arrays : tuple
3831: (12)               Indices of elements that are non-zero.

```

```

3829: (8)          See Also
3830: (8)
3831: (8)
3832: (12)         numpy.nonzero :
3833: (8)           Function operating on ndarrays.
3834: (12)         flatnonzero :
3835: (12)           Return indices that are non-zero in the flattened version of the
3836: (8)             array.
3837: (12)         numpy.ndarray.nonzero :
3838: (8)           Equivalent ndarray method.
3839: (12)         count_nonzero :
3840: (8)           Counts the number of non-zero elements in the input array.
3841: (8)          Examples
3842: (8)
3843: (8)
3844: (8)
3845: (8)
3846: (10)
3847: (16)
3848: (16)
3849: (10)
3850: (10)
3851: (8)
3852: (8)
3853: (8)
3854: (8)
3855: (8)
3856: (8)
3857: (10)
3858: (16)
3859: (16)
3860: (10)
3861: (16)
3862: (16)
3863: (10)
3864: (8)
3865: (8)
3866: (8)
3867: (8)
3868: (8)
3869: (15)
3870: (8)
3871: (8)
3872: (8)
3873: (8)
3874: (8)
3875: (8)
3876: (8)
3877: (10)
3878: (16)
3879: (16)
3880: (10)
3881: (10)
3882: (8)
3883: (8)
3884: (8)
3885: (8)
3886: (8)
3887: (8)
3888: (8)
3889: (4)
3890: (8)
3891: (8)
3892: (8)
3893: (8)
3894: (8)
3895: (12)
3896: (35)

-----
```

`numpy.nonzero` :  
 Function operating on ndarrays.

`flatnonzero` :  
 Return indices that are non-zero in the flattened version of the array.

`numpy.ndarray.nonzero` :  
 Equivalent ndarray method.

`count_nonzero` :  
 Counts the number of non-zero elements in the input array.

**Examples**

```

>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(
  data=[[1.,  0.,  0.],
        [0.,  1.,  0.],
        [0.,  0.,  1.]],
  mask=False,
  fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))
Masked elements are ignored.
>>> x[1, 1] = ma.masked
>>> x
masked_array(
  data=[[1.0,  0.0,  0.0],
        [0.0,  --,  0.0],
        [0.0,  0.0,  1.0]],
  mask=[[False, False, False],
        [False, True, False],
        [False, False, False]],
  fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))
Indices can also be grouped by element.
>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
A common use for ``nonzero`` is to find the indices of an array, where
a condition is True. Given an array `a`, the condition `a > 3` is a
boolean array and since False is interpreted as 0, ma.nonzero(a > 3)
yields the indices of the `a` where the condition is true.
>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
>>> masked_array(
    data=[[False, False, False],
          [ True,  True,  True],
          [ True,  True,  True]],
    mask=False,
    fill_value=True)
>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
The ``nonzero`` method of the condition array can also be called.
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
"""
    return narray(self.filled(0), copy=False).nonzero()
def trace(self, offset=0, axis1=0, axis2=1, dtype=None, out=None):
    """
    (this docstring should be overwritten)
    """
    m = self._mask
    if m is nomask:
        result = super().trace(offset=offset, axis1=axis1, axis2=axis2,
                              out=out)

```

```

3897: (12)
3898: (8)
3899: (12)
3900: (12)
3901: (4)
3902: (4)
3903: (8)
3904: (8)
3905: (8)
3906: (8)
3907: (8)
3908: (8)
3909: (8)
3910: (8)
3911: (8)
3912: (8)
3913: (8)
3914: (12)
3915: (8)
3916: (12)
3917: (12)
3918: (12)
3919: (12)
3920: (12)
3921: (12)
3922: (12)
3923: (8)
3924: (12)
3925: (12)
3926: (12)
3927: (12)
3928: (12)
3929: (8)
3930: (8)
3931: (8)
3932: (8)
3933: (8)
3934: (4)
3935: (8)
3936: (8)
3937: (8)
3938: (8)
3939: (8)
3940: (8)
3941: (8)
3942: (8)
3943: (8)
3944: (8)
3945: (8)
3946: (8)
3947: (8)
3948: (10)
3949: (16)
3950: (16)
3951: (10)
3952: (16)
3953: (16)
3954: (10)
3955: (8)
3956: (8)
3957: (8)
3958: (8)
3959: (21)
3960: (15)
3961: (8)
3962: (8)
3963: (21)
3964: (15)
3965: (8)

            return result.astype(dtype)
        else:
            D = self.diagonal(offset=offset, axis1=axis1, axis2=axis2)
            return D.astype(dtype).filled(0).sum(axis=-1, out=out)
    trace.__doc__ = ndarray.trace.__doc__
def dot(self, b, out=None, strict=False):
    """
    a.dot(b, out=None)
    Masked dot product of two arrays. Note that `out` and `strict` are
    located in different positions than in `ma.dot`. In order to
    maintain compatibility with the functional version, it is
    recommended that the optional arguments be treated as keyword only.
    At some point that may be mandatory.
    .. versionadded:: 1.10.0
    Parameters
    -----
    b : masked_array_like
        Inputs array.
    out : masked_array, optional
        Output argument. This must have the exact kind that would be
        returned if it was not used. In particular, it must have the
        right type, must be C-contiguous, and its dtype must be the
        dtype that would be returned for `ma.dot(a,b)`. This is a
        performance feature. Therefore, if these conditions are not
        met, an exception is raised, instead of attempting to be
        flexible.
    strict : bool, optional
        Whether masked data are propagated (True) or set to 0 (False)
        for the computation. Default is False. Propagating the mask
        means that if a masked value appears in a row or column, the
        whole row or column is considered masked.
        .. versionadded:: 1.10.2
    See Also
    -----
    numpy.ma.dot : equivalent function
    """
    return dot(self, b, out=out, strict=strict)
def sum(self, axis=None, dtype=None, out=None, keepdims=np._NoValue):
    """
    Return the sum of the array elements over the given axis.
    Masked elements are set to 0 internally.
    Refer to `numpy.sum` for full documentation.
    See Also
    -----
    numpy.ndarray.sum : corresponding function for ndarrays
    numpy.sum : equivalent function
    Examples
    -----
    >>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
    >>> x
    masked_array(
        data=[[1, --, 3],
              [--, 5, --],
              [7, --, 9]],
        mask=[[False, True, False],
              [True, False, True],
              [False, True, False]],
        fill_value=999999)
    >>> x.sum()
    25
    >>> x.sum(axis=1)
    masked_array(data=[4, 5, 16],
                 mask=[False, False, False],
                 fill_value=999999)
    >>> x.sum(axis=0)
    masked_array(data=[8, 5, 12],
                 mask=[False, False, False],
                 fill_value=999999)
    >>> print(type(x.sum(axis=0, dtype=np.int64)[0]))

```

```

3966: (8) <class 'numpy.int64'>
3967: (8)
3968: (8)
3969: (8)
3970: (8)
3971: (8)
3972: (12)
3973: (12)
3974: (12)
3975: (16)
3976: (16)
3977: (12)
3978: (16)
3979: (12)
3980: (8)
3981: (8)
3982: (12)
3983: (12)
3984: (16)
3985: (12)
3986: (8)
3987: (4)
3988: (8)
3989: (8) Return the cumulative sum of the array elements over the given axis.
3990: (8) Masked values are set to 0 internally during the computation.
3991: (8) However, their position is saved, and the result will be masked at
3992: (8) the same locations.
3993: (8) Refer to `numpy.cumsum` for full documentation.
3994: (8) Notes
3995: (8) -----
3996: (8) The mask is lost if `out` is not a valid :class:`ma.MaskedArray` !
3997: (8) Arithmetic is modular when using integer types, and no error is
3998: (8) raised on overflow.
3999: (8) See Also
4000: (8) -----
4001: (8) numpy.ndarray.cumsum : corresponding function for ndarrays
4002: (8) numpy.cumsum : equivalent function
4003: (8) Examples
4004: (8) -----
4005: (8) >>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
4006: (8) >>> marr.cumsum()
4007: (8) masked_array(data=[0, 1, 3, --, --, --, 9, 16, 24, 33],
4008: (21) mask=[False, False, False, True, True, True, False,
4009: (27) False, False],
4010: (15) fill_value=999999)
4011: (8) """
4012: (8) result = self.filled(0).cumsum(axis=axis, dtype=dtype, out=out)
4013: (8) if out is not None:
4014: (12)     if isinstance(out, MaskedArray):
4015: (16)         out.__setmask__(self.mask)
4016: (12)     return out
4017: (8) result = result.view(type(self))
4018: (8) result.__setmask__(self._mask)
4019: (8) return result
4020: (4) def prod(self, axis=None, dtype=None, out=None, keepdims=np._NoValue):
4021: (8) """
4022: (8) Return the product of the array elements over the given axis.
4023: (8) Masked elements are set to 1 internally for computation.
4024: (8) Refer to `numpy.prod` for full documentation.
4025: (8) Notes
4026: (8) -----
4027: (8) Arithmetic is modular when using integer types, and no error is raised
4028: (8) on overflow.
4029: (8) See Also
4030: (8) -----
4031: (8) numpy.ndarray.prod : corresponding function for ndarrays
4032: (8) numpy.prod : equivalent function
4033: (8) """

```

```

4034: (8)         kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
4035: (8)         _mask = self._mask
4036: (8)         newmask = _check_mask_axis(_mask, axis, **kwargs)
4037: (8)         if out is None:
4038: (12)             result = self.filled(1).prod(axis, dtype=dtype, **kwargs)
4039: (12)             rndim = getattr(result, 'ndim', 0)
4040: (12)             if rndim:
4041: (16)                 result = result.view(type(self))
4042: (16)                 result.__setmask__(newmask)
4043: (12)             elif newmask:
4044: (16)                 result = masked
4045: (12)             return result
4046: (8)         result = self.filled(1).prod(axis, dtype=dtype, out=out, **kwargs)
4047: (8)         if isinstance(out, MaskedArray):
4048: (12)             outmask = getmask(out)
4049: (12)             if outmask is nomask:
4050: (16)                 outmask = out._mask = make_mask_none(out.shape)
4051: (12)                 outmask.flat = newmask
4052: (8)             return out
4053: (4)         product = prod
4054: (4)         def cumprod(self, axis=None, dtype=None, out=None):
4055: (8)             """
4056: (8)             Return the cumulative product of the array elements over the given
axis.
4057: (8)             Masked values are set to 1 internally during the computation.
4058: (8)             However, their position is saved, and the result will be masked at
the same locations.
4059: (8)             Refer to `numpy.cumprod` for full documentation.
4060: (8)             Notes
4061: (8)             -----
4062: (8)             The mask is lost if `out` is not a valid MaskedArray !
4063: (8)             Arithmetic is modular when using integer types, and no error is
raised on overflow.
4064: (8)             See Also
4065: (8)             -----
4066: (8)             numpy.ndarray.cumprod : corresponding function for ndarrays
4067: (8)             numpy.cumprod : equivalent function
4068: (8)             """
4069: (8)             result = self.filled(1).cumprod(axis=axis, dtype=dtype, out=out)
4070: (8)             if out is not None:
4071: (12)                 if isinstance(out, MaskedArray):
4072: (16)                     out.__setmask__(self._mask)
4073: (12)                 return out
4074: (16)             result = result.view(type(self))
4075: (12)             result.__setmask__(self._mask)
4076: (8)             return result
4077: (8)         def mean(self, axis=None, dtype=None, out=None, keepdims=np._NoValue):
4078: (8)             """
4079: (8)             Returns the average of the array elements along given axis.
4080: (8)             Masked entries are ignored, and result elements which are not
finite will be masked.
4081: (8)             Refer to `numpy.mean` for full documentation.
4082: (8)             See Also
4083: (8)             -----
4084: (8)             numpy.ndarray.mean : corresponding function for ndarrays
4085: (8)             numpy.mean : Equivalent function
4086: (8)             numpy.ma.average : Weighted average.
4087: (8)             Examples
4088: (8)             -----
4089: (8)             >>> a = np.ma.array([1,2,3], mask=[False, False, True])
4090: (8)             >>> a
4091: (8)             masked_array(data=[1, 2, --],
4092: (8)                         mask=[False, False, True],
4093: (8)                         fill_value=999999)
4094: (8)             >>> a.mean()
4095: (21)             1.5
4096: (15)             """
4097: (8)             kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
4098: (8)             if self._mask is nomask:
4099: (8)
4100: (8)
4101: (8)

```

```

4102: (12)             result = super().mean(axis=axis, dtype=dtype, **kwargs)[()]
4103: (8)              else:
4104: (12)                  is_float16_result = False
4105: (12)                  if dtype is None:
4106: (16)                      if issubclass(self.dtype.type, (ntypes.integer,
4107: (20)                          ntypes.bool_)):
4108: (16)                          dtype = mu.dtype('f8')
4109: (20)                          elif issubclass(self.dtype.type, ntypes.float16):
4110: (20)                              dtype = mu.dtype('f4')
4111: (12)                              is_float16_result = True
4112: (12)                          dsum = self.sum(axis=axis, dtype=dtype, **kwargs)
4113: (12)                          cnt = self.count(axis=axis, **kwargs)
4114: (16)                          if cnt.shape == () and (cnt == 0):
4115: (12)                              result = masked
4116: (16)                          elif is_float16_result:
4117: (12)                              result = self.dtype.type(dsum * 1. / cnt)
4118: (16)                          else:
4119: (8)                              result = dsum * 1. / cnt
4120: (12)                  if out is not None:
4121: (12)                      out.flat = result
4122: (16)                      if isinstance(out, MaskedArray):
4123: (16)                          outmask = getmask(out)
4124: (20)                          if outmask is nomask:
4125: (16)                              outmask = out._mask = make_mask_none(out.shape)
4126: (12)                          outmask.flat = getmask(result)
4127: (8)                      return out
4128: (4)                  return result
4129: (8)              def anom(self, axis=None, dtype=None):
4130: (8)                  """
4131: (8)                  Compute the anomalies (deviations from the arithmetic mean)
4132: (8)                  along the given axis.
4133: (8)                  Returns an array of anomalies, with the same shape as the input and
4134: (8)                  where the arithmetic mean is computed along the given axis.
4135: (8)                  Parameters
4136: (8)                  -----
4137: (12)                      axis : int, optional
4138: (12)                          Axis over which the anomalies are taken.
4139: (8)                          The default is to use the mean of the flattened array as
4140: (12)                          reference.
4141: (13)                  dtype : dtype, optional
4142: (13)                      Type to use in computing the variance. For arrays of integer type
4143: (8)                          the default is float32; for arrays of float types it is the same
4144: (8)                          as
4145: (8)                          the array type.
4146: (8)              See Also
4147: (8)                  -----
4148: (8)                  mean : Compute the mean of the array.
4149: (8)              Examples
4150: (8)                  -----
4151: (21)                  >>> a = np.ma.array([1,2,3])
4152: (15)                  >>> a.anom()
4153: (8)                  masked_array(data=[-1.,  0.,  1.],
4154: (15)                                  mask=False,
4155: (8)                                  fill_value=1e+20)
4156: (12)                  """
4157: (8)                  m = self.mean(axis, dtype)
4158: (12)                  if not axis:
4159: (4)                      return self - m
4160: (12)                  else:
4161: (8)                      return self - expand_dims(m, axis)
4162: (8)              def var(self, axis=None, dtype=None, out=None, ddof=0,
4163: (8)                  keepdims=np._NoValue):
4164: (8)                  """
4165: (8)                  Returns the variance of the array elements along given axis.
4166: (8)                  Masked entries are ignored, and result elements which are not
4167: (8)                  finite will be masked.

```

```

4168: (8)             numpy.ndarray.var : corresponding function for ndarrays
4169: (8)             numpy.var : Equivalent function
4170: (8)
4171: (8)             """
4172: (8)             kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
4173: (12)            if self._mask is nomask:
4174: (30)              ret = super().var(axis=axis, dtype=dtype, out=out, ddof=ddof,
4175: (12)                  **kwargs)[()]
4176: (16)              if out is not None:
4177: (20)                if isinstance(out, MaskedArray):
4178: (16)                  out.__setmask__(nomask)
4179: (12)                  return out
4180: (8)                  return ret
4181: (8)              cnt = self.count(axis=axis, **kwargs) - ddof
4182: (8)              danom = self - self.mean(axis, dtype, keepdims=True)
4183: (12)              if iscomplexobj(self):
4184: (8)                danom = umath.absolute(danom) ** 2
4185: (12)              else:
4186: (8)                danom *= danom
4187: (8)              dvar = divide(danom.sum(axis, **kwargs), cnt).view(type(self))
4188: (12)              if dvar.ndim:
4189: (12)                dvar._mask = mask_or(self._mask.all(axis, **kwargs), (cnt <= 0))
4190: (8)                dvar._update_from(self)
4191: (12)              elif getmask(dvar):
4192: (12)                dvar = masked
4193: (16)                if out is not None:
4194: (20)                  if isinstance(out, MaskedArray):
4195: (20)                      out.flat = 0
4196: (16)                      out.__setmask__(True)
4197: (20)                      elif out.dtype.kind in 'biu':
4198: (29)                        errmsg = "Masked data information would be lost in one or
```
4199: (20)                          "more location."
4200: (16)                          raise MaskError(errmsg)
4201: (20)                      else:
4202: (16)                        out.flat = np.nan
4203: (8)                        return out
4204: (12)                      if out is not None:
4205: (12)                        out.flat = dvar
4206: (16)                        if isinstance(out, MaskedArray):
4207: (12)                          out.__setmask__(dvar.mask)
4208: (8)                          return out
4209: (4)                          return dvar
4210: (4)                          var.__doc__ = np.var.__doc__
4211: (12)                      def std(self, axis=None, dtype=None, out=None, ddof=0,
4212: (8)                          keepdims=np._NoValue):
4213: (8)                          """
4214: (8)                          Returns the standard deviation of the array elements along given axis.
4215: (8)                          Masked entries are ignored.
4216: (8)                          Refer to `numpy.std` for full documentation.
4217: (8)                          See Also
4218: (8)                          -----
4219: (8)                          numpy.ndarray.std : corresponding function for ndarrays
4220: (8)                          numpy.std : Equivalent function
4221: (8)                          """
4222: (8)                          kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
4223: (8)                          dvar = self.var(axis, dtype, out, ddof, **kwargs)
4224: (12)                          if dvar is not masked:
4225: (16)                            if out is not None:
4226: (16)                              np.power(out, 0.5, out=out, casting='unsafe')
4227: (12)                              return out
4228: (8)                              dvar = sqrt(dvar)
4229: (4)                              return dvar
4230: (8)
4231: (8)                          def round(self, decimals=0, out=None):
4232: (8)                          """
4233: (8)                          Return each element rounded to the given number of decimals.
4234: (8)                          Refer to `numpy.around` for full documentation.
4235: (8)                          See Also
```
4236: (8)                          -----
4237: (8)                          numpy.ndarray.round : corresponding function for ndarrays

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4236: (8)           numpy.around : equivalent function
4237: (8)
4238: (8)
4239: (8)
4240: (12)
4241: (12)
4242: (8)
4243: (12)
4244: (8)
4245: (12)
4246: (8)
4247: (12)
4248: (8)
4249: (4)           def argsort(self, axis=np._NoValue, kind=None, order=None,
4250: (16)             endwith=True, fill_value=None):
4251: (8)
4252: (8)             """
4253: (8)             Return an ndarray of indices that sort the array along the
4254: (8)             specified axis. Masked values are filled beforehand to
4255: (8)             `fill_value`.
4256: (8)             Parameters
4257: (8)             -----
4258: (12)             axis : int, optional
4259: (12)               Axis along which to sort. If None, the default, the flattened
4260: (12)               array
4261: (16)               is used.
4262: (16)               .. versionchanged:: 1.13.0
4263: (16)                 Previously, the default was documented to be -1, but that was
4264: (16)                 in error. At some future date, the default will change to -1,
4265: (16)                 as
4266: (8)                 originally intended.
4267: (12)                 Until then, the axis should be given explicitly when
4268: (8)                   ``arr.ndim > 1``, to avoid a FutureWarning.
4269: (12)               kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
4270: (12)               The sorting algorithm used.
4271: (12)               order : list, optional
4272: (8)                 When `a` is an array with fields defined, this argument specifies
4273: (12)                 which fields to compare first, second, etc. Not all fields need
4274: (12)                 be
4275: (12)                 specified.
4276: (12)               endwith : {True, False}, optional
4277: (12)                 Whether missing values (if any) should be treated as the largest
4278: (8)                   values
4279: (12)                   (True) or the smallest values (False)
4280: (12)                   When the array contains unmasked values at the same extremes of
4281: (8)                     datatype, the ordering of these values and the masked values is
4282: (8)                     undefined.
4283: (8)               fill_value : scalar or None, optional
4284: (12)                 Value used internally for the masked values.
4285: (12)                 If ``fill_value`` is not None, it supersedes ``endwith``.
4286: (8)               Returns
4287: (8)               -----
4288: (8)               index_array : ndarray, int
4289: (12)                 Array of indices that sort `a` along the specified axis.
4290: (12)                 In other words, ``a[index_array]`` yields a sorted `a`.
4291: (8)               See Also
4292: (8)               -----
4293: (8)               ma.MaskedArray.sort : Describes sorting algorithms used.
4294: (8)               lexsort : Indirect stable sort with multiple keys.
4295: (8)               numpy.ndarray.sort : Inplace sort.
4296: (8)               Notes
4297: (8)               -----
4298: (8)               See `sort` for notes on the different sorting algorithms.
4299: (21)               Examples
4296: (8)               -----
4297: (8)               >>> a = np.ma.array([3,2,1], mask=[False, False, True])
4298: (8)               >>> a
4299: (21)               masked_array(data=[3, 2, --],
4299: (21)                           mask=[False, False, True]),

```

```

4300: (15)                                fill_value=999999)
4301: (8)                                 >>> a.argsort()
4302: (8)                                 array([1, 0, 2])
4303: (8)                                 """
4304: (8)                                 if axis is np._NoValue:
4305: (12)                               axis = _deprecate_argsort_axis(self)
4306: (8)                                 if fill_value is None:
4307: (12)                               if endwith:
4308: (16)                                 if np.issubdtype(self.dtype, np.floating):
4309: (20)                                   fill_value = np.nan
4310: (16)                                 else:
4311: (20)                                   fill_value = minimum_fill_value(self)
4312: (12)                                 else:
4313: (16)                                   fill_value = maximum_fill_value(self)
4314: (8)                                 filled = self.filled(fill_value)
4315: (8)                                 return filled.argsort(axis=axis, kind=kind, order=order)
4316: (4)                                 def argmin(self, axis=None, fill_value=None, out=None, *,
4317: (16)                                       keepdims=np._NoValue):
4318: (8)                                 """
4319: (8)                                 Return array of indices to the minimum values along the given axis.
4320: (8)                                 Parameters
4321: (8)                                 -----
4322: (8)                                 axis : {None, integer}
4323: (12)                               If None, the index is into the flattened array, otherwise along
4324: (12)                               the specified axis
4325: (8)                                 fill_value : scalar or None, optional
4326: (12)                               Value used to fill in the masked values. If None, the output of
4327: (12)                               minimum_fill_value(self._data) is used instead.
4328: (8)                                 out : {None, array}, optional
4329: (12)                               Array into which the result can be placed. Its type is preserved
4330: (12)                               and it must be of the right shape to hold the output.
4331: (8)                                 Returns
4332: (8)                                 -----
4333: (8)                                 ndarray or scalar
4334: (12)                               If multi-dimension input, returns a new ndarray of indices to the
4335: (12)                               minimum values along the given axis. Otherwise, returns a scalar
4336: (12)                               of index to the minimum values along the given axis.
4337: (8)                                 Examples
4338: (8)                                 -----
4339: (8)                                 >>> x = np.ma.array(np.arange(4), mask=[1,1,0,0])
4340: (8)                                 >>> x.shape = (2,2)
4341: (8)                                 >>> x
4342: (8)                                 masked_array(
4343: (10)                                   data=[[--, --],
4344: (16)                                     [2, 3]],
4345: (10)                                   mask=[[ True,  True],
4346: (16)                                     [False, False]],
4347: (10)                                   fill_value=999999)
4348: (8)                                 >>> x.argmin(axis=0, fill_value=-1)
4349: (8)                                 array([0, 0])
4350: (8)                                 >>> x.argmin(axis=0, fill_value=9)
4351: (8)                                 array([1, 1])
4352: (8)                                 """
4353: (8)                                 if fill_value is None:
4354: (12)                                   fill_value = minimum_fill_value(self)
4355: (8)                                 d = self.filled(fill_value).view(ndarray)
4356: (8)                                 keepdims = False if keepdims is np._NoValue else bool(keepdims)
4357: (8)                                 return d.argmax(axis, out=out, keepdims=keepdims)
4358: (4)                                 def argmax(self, axis=None, fill_value=None, out=None, *,
4359: (16)                                       keepdims=np._NoValue):
4360: (8)                                 """
4361: (8)                                 Returns array of indices of the maximum values along the given axis.
4362: (8)                                 Masked values are treated as if they had the value fill_value.
4363: (8)                                 Parameters
4364: (8)                                 -----
4365: (8)                                 axis : {None, integer}
4366: (12)                               If None, the index is into the flattened array, otherwise along
4367: (12)                               the specified axis
4368: (8)                                 fill_value : scalar or None, optional

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

4369: (12)           Value used to fill in the masked values. If None, the output of
4370: (12)           maximum_fill_value(self._data) is used instead.
4371: (8)            out : {None, array}, optional
4372: (12)           Array into which the result can be placed. Its type is preserved
4373: (12)           and it must be of the right shape to hold the output.
4374: (8)            Returns
4375: (8)           -----
4376: (8)           index_array : {integer_array}
4377: (8)            Examples
4378: (8)           -----
4379: (8)           >>> a = np.arange(6).reshape(2,3)
4380: (8)           >>> a.argmax()
4381: (8)           5
4382: (8)           >>> a.argmax(0)
4383: (8)           array([1, 1, 1])
4384: (8)           >>> a.argmax(1)
4385: (8)           array([2, 2])
4386: (8)           """
4387: (8)           if fill_value is None:
4388: (12)             fill_value = maximum_fill_value(self._data)
4389: (8)           d = self.filled(fill_value).view(ndarray)
4390: (8)           keepdims = False if keepdims is np._NoValue else bool(keepdims)
4391: (8)           return d.argmax(axis, out=out, keepdims=keepdims)
4392: (4)            def sort(self, axis=-1, kind=None, order=None,
4393: (13)              endwith=True, fill_value=None):
4394: (8)           """
4395: (8)           Sort the array, in-place
4396: (8)            Parameters
4397: (8)           -----
4398: (8)           a : array_like
4399: (12)             Array to be sorted.
4400: (8)           axis : int, optional
4401: (12)             Axis along which to sort. If None, the array is flattened before
4402: (12)             sorting. The default is -1, which sorts along the last axis.
4403: (8)           kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
4404: (12)             The sorting algorithm used.
4405: (8)           order : list, optional
4406: (12)             When `a` is a structured array, this argument specifies which
fields
4407: (12)             to compare first, second, and so on. This list does not need to
4408: (12)             include all of the fields.
4409: (8)            endwith : {True, False}, optional
4410: (12)             Whether missing values (if any) should be treated as the largest
values
4411: (12)             (True) or the smallest values (False)
4412: (12)             When the array contains unmasked values sorting at the same
extremes of the
4413: (12)             datatype, the ordering of these values and the masked values is
4414: (12)             undefined.
4415: (8)            fill_value : scalar or None, optional
4416: (12)             Value used internally for the masked values.
4417: (12)             If ``fill_value`` is not None, it supersedes ``endwith``.
4418: (8)            Returns
4419: (8)           -----
4420: (8)           sorted_array : ndarray
4421: (12)             Array of the same type and shape as `a`.
4422: (8)            See Also
4423: (8)           -----
4424: (8)           numpy.ndarray.sort : Method to sort an array in-place.
4425: (8)           argsort : Indirect sort.
4426: (8)           lexsort : Indirect stable sort on multiple keys.
4427: (8)           searchsorted : Find elements in a sorted array.
4428: (8)            Notes
4429: (8)           -----
4430: (8)           See ``sort`` for notes on the different sorting algorithms.
4431: (8)            Examples
4432: (8)           -----
4433: (8)           >>> a = np.ma.array([1, 2, 5, 4, 3], mask=[0, 1, 0, 1, 0])
4434: (8)           >>> # Default

```

```

4435: (8)          >>> a.sort()
4436: (8)          >>> a
4437: (8)          masked_array(data=[1, 3, 5, --, --],
4438: (21)            mask=[False, False, False, True, True],
4439: (15)              fill_value=999999)
4440: (8)          >>> a = np.ma.array([1, 2, 5, 4, 3],mask=[0, 1, 0, 1, 0])
4441: (8)          >>> # Put missing values in the front
4442: (8)          >>> a.sort(endwith=False)
4443: (8)          >>> a
4444: (8)          masked_array(data=[--, --, 1, 3, 5],
4445: (21)            mask=[ True, True, False, False, False],
4446: (15)              fill_value=999999)
4447: (8)          >>> a = np.ma.array([1, 2, 5, 4, 3],mask=[0, 1, 0, 1, 0])
4448: (8)          >>> # fill_value takes over endwith
4449: (8)          >>> a.sort(endwith=False, fill_value=3)
4450: (8)          >>> a
4451: (8)          masked_array(data=[1, --, --, 3, 5],
4452: (21)            mask=[False, True, True, False, False],
4453: (15)              fill_value=999999)
4454: (8)          """
4455: (8)          if self._mask is nomask:
4456: (12)              ndarray.sort(self, axis=axis, kind=kind, order=order)
4457: (12)              return
4458: (8)          if self is masked:
4459: (12)              return
4460: (8)          sidx = self.argsort(axis=axis, kind=kind, order=order,
4461: (28)                  fill_value=fill_value, endwith=endwith)
4462: (8)          self[...] = np.take_along_axis(self, sidx, axis=axis)
def min(self, axis=None, out=None, fill_value=None, keepdims=np._NoValue):
    """
        Return the minimum along a given axis.

    Parameters
    -----
    axis : None or int or tuple of ints, optional
        Axis along which to operate. By default, ``axis`` is None and the
        flattened input is used.
        .. versionadded:: 1.7.0
        If this is a tuple of ints, the minimum is selected over multiple
        axes, instead of a single axis or all the axes as before.
    out : array_like, optional
        Alternative output array in which to place the result. Must be of
        the same shape and buffer length as the expected output.
    fill_value : scalar or None, optional
        Value used to fill in the masked values.
        If None, use the output of `minimum_fill_value` .
    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the array.

    Returns
    -----
    amin : array_like
        New array holding the result.
        If ``out`` was specified, ``out`` is returned.

    See Also
    -----
    ma.minimum_fill_value
        Returns the minimum filling value for a given datatype.

    Examples
    -----
    >>> import numpy.ma as ma
    >>> x = [[1., -2., 3.], [0.2, -0.7, 0.1]]
    >>> mask = [[1, 1, 0], [0, 0, 1]]
    >>> masked_x = ma.masked_array(x, mask)
    >>> masked_x
    masked_array(
        data=[[--, --, 3.0],
              [0.2, -0.7, --]],
        mask=[[ True,  True, False],
               [False, False,  True]],
```

```

4504: (16)           [False, False,  True]],  

4505: (10)          fill_value=1e+20)  

4506: (8)           >>> ma.min(masked_x)  

4507: (8)           -0.7  

4508: (8)           >>> ma.min(masked_x, axis=-1)  

4509: (8)           masked_array(data=[3.0, -0.7],  

4510: (21)          mask=[False, False],  

4511: (16)          fill_value=1e+20)  

4512: (8)           >>> ma.min(masked_x, axis=0, keepdims=True)  

4513: (8)           masked_array(data=[[0.2, -0.7, 3.0]],  

4514: (21)          mask=[[False, False, False]],  

4515: (16)          fill_value=1e+20)  

4516: (8)           >>> mask = [[1, 1, 1,], [1, 1, 1]]  

4517: (8)           >>> masked_x = ma.masked_array(x, mask)  

4518: (8)           >>> ma.min(masked_x, axis=0)  

4519: (8)           masked_array(data=[--, --, --],  

4520: (21)          mask=[ True,  True,  True],  

4521: (16)          fill_value=1e+20,  

4522: (20)          dtype=float64)  

4523: (8)          """  

4524: (8)          kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}  

4525: (8)          _mask = self._mask  

4526: (8)          newmask = _check_mask_axis(_mask, axis, **kwargs)  

4527: (8)          if fill_value is None:  

4528: (12)          fill_value = minimum_fill_value(self)  

4529: (8)          if out is None:  

4530: (12)          result = self.filled(fill_value).min(  

4531: (16)          axis=axis, out=out, **kwargs).view(type(self))  

4532: (12)          if result.ndim:  

4533: (16)          result.__setmask__(newmask)  

4534: (16)          if newmask.ndim:  

4535: (20)          np.copyto(result, result.fill_value, where=newmask)  

4536: (12)          elif newmask:  

4537: (16)          result = masked  

4538: (12)          return result  

4539: (8)          result = self.filled(fill_value).min(axis=axis, out=out, **kwargs)  

4540: (8)          if isinstance(out, MaskedArray):  

4541: (12)          outmask = getmask(out)  

4542: (12)          if outmask is nomask:  

4543: (16)          outmask = out._mask = make_mask_none(out.shape)  

4544: (12)          outmask.flat = newmask  

4545: (8)          else:  

4546: (12)          if out.dtype.kind in 'biu':  

4547: (16)          errmsg = "Masked data information would be lost in one or  

more"\  

4548: (25)          " location."  

4549: (16)          raise MaskError(errmsg)  

4550: (12)          np.copyto(out, np.nan, where=newmask)  

4551: (8)          return out  

4552: (4)          def max(self, axis=None, out=None, fill_value=None, keepdims=np._NoValue):  

4553: (8)          """  

4554: (8)          Return the maximum along a given axis.  

4555: (8)          Parameters  

4556: (8)          -----  

4557: (8)          axis : None or int or tuple of ints, optional  

4558: (12)          Axis along which to operate. By default, ``axis`` is None and the  

4559: (12)          flattened input is used.  

4560: (12)          .. versionadded:: 1.7.0  

4561: (12)          If this is a tuple of ints, the maximum is selected over multiple  

4562: (12)          axes, instead of a single axis or all the axes as before.  

4563: (8)          out : array_like, optional  

4564: (12)          Alternative output array in which to place the result. Must  

4565: (12)          be of the same shape and buffer length as the expected output.  

4566: (8)          fill_value : scalar or None, optional  

4567: (12)          Value used to fill in the masked values.  

4568: (12)          If None, use the output of maximum_fill_value().  

4569: (8)          keepdims : bool, optional  

4570: (12)          If this is set to True, the axes which are reduced are left  

4571: (12)          in the result as dimensions with size one. With this option,

```

```

4572: (12)           the result will broadcast correctly against the array.
4573: (8)           Returns
4574: (8)
4575: (8)
4576: (12)           -----
4577: (12)           amax : array_like
4578: (8)           New array holding the result.
4579: (8)           If ``out`` was specified, ``out`` is returned.
4580: (8)           See Also
4581: (12)           -----
4582: (8)           ma.maximum_fill_value
4583: (8)           Returns the maximum filling value for a given datatype.
4584: (8)           Examples
4585: (8)
4586: (8)
4587: (8)
4588: (8)
4589: (8)
4590: (10)
4591: (16)
4592: (16)
4593: (10)
4594: (16)
4595: (16)
4596: (10)
4597: (8)
4598: (8)
4599: (8)
4600: (8)
4601: (21)
4602: (15)
4603: (8)
4604: (8)
4605: (10)
4606: (16)
4607: (16)
4608: (10)
4609: (16)
4610: (16)
4611: (10)
4612: (8)
4613: (8)
4614: (8)
4615: (8)
4616: (21)
4617: (15)
4618: (20)
4619: (8)
4620: (8)
4621: (8)
4622: (8)
4623: (8)
4624: (12)
4625: (8)
4626: (12)
4627: (16)
4628: (12)
4629: (16)
4630: (16)
4631: (20)
4632: (12)
4633: (16)
4634: (12)
4635: (8)
4636: (8)
4637: (12)
4638: (12)
4639: (16)
4640: (12)

           the result will broadcast correctly against the array.

           Returns
           -----
           amax : array_like
           New array holding the result.
           If ``out`` was specified, ``out`` is returned.

See Also
-----
ma.maximum_fill_value
           Returns the maximum filling value for a given datatype.

Examples
-----
>>> import numpy.ma as ma
>>> x = [[-1., 2.5], [4., -2.], [3., 0.]]
>>> mask = [[0, 0], [1, 0], [1, 0]]
>>> masked_x = ma.masked_array(x, mask)
>>> masked_x
masked_array(
    data=[[-1.0, 2.5],
          [--, -2.0],
          [--, 0.0]],
    mask=[[False, False],
          [True, False],
          [True, False]],
    fill_value=1e+20)
>>> ma.max(masked_x)
2.5
>>> ma.max(masked_x, axis=0)
masked_array(data=[-1.0, 2.5],
             mask=[False, False],
             fill_value=1e+20)
>>> ma.max(masked_x, axis=1, keepdims=True)
masked_array(
    data=[[2.5],
          [-2.0],
          [0.0]],
    mask=[[False],
          [False],
          [False]],
    fill_value=1e+20)
>>> mask = [[1, 1], [1, 1], [1, 1]]
>>> masked_x = ma.masked_array(x, mask)
>>> ma.max(masked_x, axis=1)
masked_array(data=[--, --, --],
             mask=[ True,  True,  True],
             fill_value=1e+20,
             dtype=float64)
"""

kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
_mask = self._mask
newmask = _check_mask_axis(_mask, axis, **kwargs)
if fill_value is None:
    fill_value = maximum_fill_value(self)
if out is None:
    result = self.filled(fill_value).max(
        axis=axis, out=out, **kwargs).view(type(self))
    if result.ndim:
        result.__setmask__(newmask)
        if newmask.ndim:
            np.copyto(result, result.fill_value, where=newmask)
elif newmask:
    result = masked
    return result
result = self.filled(fill_value).max(axis=axis, out=out, **kwargs)
if isinstance(out, MaskedArray):
    outmask = getmask(out)
    if outmask is nomask:
        outmask = out._mask = make_mask_none(out.shape)
    outmask.flat = newmask

```

```

4641: (8)
4642: (12)
4643: (16)
more"\_
4644: (25)
4645: (16)
4646: (12)
4647: (8)
4648: (4)
4649: (8)
4650: (8)
4651: (8)
4652: (8)
4653: (12)
4654: (12)
4655: (12)
4656: (12)
4657: (12)
4658: (12)
4659: (8)
4660: (8)
4661: (8)
4662: (12)
4663: (12)
4664: (8)
4665: (12)
4666: (12)
4667: (12)
4668: (8)
4669: (12)
4670: (8)
4671: (12)
4672: (12)
4673: (12)
4674: (8)
4675: (8)
4676: (8)
4677: (12)
4678: (12)
4679: (8)
4680: (8)
4681: (8)
4682: (8)
4683: (8)
4684: (8)
4685: (21)
4686: (15)
4687: (8)
4688: (8)
4689: (21)
4690: (15)
4691: (8)
4692: (8)
4693: (8)
4694: (8)
4695: (8)
4696: (8)
4697: (8)
4698: (8)
4699: (8)
4700: (8)
4701: (21)
4702: (15)
4703: (20)
4704: (8)
4705: (8)
4706: (8)
4707: (8)
4708: (21)

        else:
            if out.dtype.kind in 'biu':
                errmsg = "Masked data information would be lost in one or
                           " location."
                raise MaskError(errmsg)
            np.copyto(out, np.nan, where=newmask)
        return out
    def ptp(self, axis=None, out=None, fill_value=None, keepdims=False):
        """
        Return (maximum - minimum) along the given dimension
        (i.e. peak-to-peak value).
        .. warning::
            `ptp` preserves the data type of the array. This means the
            return value for an input of signed integers with n bits
            (e.g. `np.int8`, `np.int16`, etc) is also a signed integer
            with n bits. In that case, peak-to-peak values greater than
            ``2**({n}-1`` will be returned as negative values. An example
            with a work-around is shown below.

        Parameters
        -----
        axis : {None, int}, optional
            Axis along which to find the peaks. If None (default) the
            flattened array is used.
        out : {None, array_like}, optional
            Alternative output array in which to place the result. It must
            have the same shape and buffer length as the expected output
            but the type will be cast if necessary.
        fill_value : scalar or None, optional
            Value used to fill in the masked values.
        keepdims : bool, optional
            If this is set to True, the axes which are reduced are left
            in the result as dimensions with size one. With this option,
            the result will broadcast correctly against the array.

        Returns
        -----
        ptp : ndarray.
            A new array holding the result, unless ``out`` was
            specified, in which case a reference to ``out`` is returned.

        Examples
        -----
        >>> x = np.ma.MaskedArray([[4, 9, 2, 10],
           ...                      [6, 9, 7, 12]])
        >>> x.ptp(axis=1)
        masked_array(data=[8, 6],
                     mask=False,
                     fill_value=999999)
        >>> x.ptp(axis=0)
        masked_array(data=[2, 0, 5, 2],
                     mask=False,
                     fill_value=999999)
        >>> x.ptp()
        10
        This example shows that a negative value can be returned when
        the input is an array of signed integers.
        >>> y = np.ma.MaskedArray([[1, 127],
           ...                      [0, 127],
           ...                      [-1, 127],
           ...                      [-2, 127]], dtype=np.int8)
        >>> y.ptp(axis=1)
        masked_array(data=[ 126,  127, -128, -127],
                     mask=False,
                     fill_value=999999,
                     dtype=int8)
        A work-around is to use the `view()` method to view the result as
        unsigned integers with the same bit width:
        >>> y.ptp(axis=1).view(np.uint8)
        masked_array(data=[126, 127, 128, 129],
                     mask=False,

```

```

4709: (15)
4710: (20)
4711: (8)
4712: (8)
4713: (12)
4714: (30)
4715: (12)
4716: (31)
4717: (12)
4718: (8)
4719: (28)
4720: (8)
4721: (29)
4722: (8)
4723: (8)
4724: (4)
4725: (8)
4726: (22)
4727: (22)
4728: (8)
4729: (4)
4730: (8)
4731: (22)
4732: (22)
4733: (8)
4734: (4)
4735: (8)
4736: (8)
4737: (8)
4738: (8)
4739: (8)
4740: (8)
4741: (12)
4742: (8)
4743: (12)
4744: (8)
4745: (12)
4746: (8)
4747: (12)
4748: (16)
4749: (12)
4750: (16)
4751: (16)
4752: (12)
4753: (8)
4754: (4)
4755: (4)
4756: (4)
4757: (4)
4758: (4)
4759: (4)
4760: (4)
4761: (4)
4762: (4)
4763: (8)
4764: (8)
list.
4765: (8)
4766: (8)
4767: (8)
4768: (8)
4769: (8)
4770: (8)
4771: (12)
4772: (8)
4773: (8)
4774: (8)
4775: (12)
4776: (8)

fill_value=999999,
dtype=uint8)
"""
if out is None:
    result = self.max(axis=axis, fill_value=fill_value,
                      keepdims=keepdims)
    result -= self.min(axis=axis, fill_value=fill_value,
                      keepdims=keepdims)
return result
out.flat = self.max(axis=axis, out=out, fill_value=fill_value,
                     keepdims=keepdims)
min_value = self.min(axis=axis, fill_value=fill_value,
                     keepdims=keepdims)
np.subtract(out, min_value, out=out, casting='unsafe')
return out
def partition(self, *args, **kwargs):
    warnings.warn("Warning: 'partition' will ignore the 'mask' "
                  "of the {self.__class__.__name__}.",
                  stacklevel=2)
    return super().partition(*args, **kwargs)
def argpartition(self, *args, **kwargs):
    warnings.warn("Warning: 'argpartition' will ignore the 'mask' "
                  "of the {self.__class__.__name__}.",
                  stacklevel=2)
    return super().argpartition(*args, **kwargs)
def take(self, indices, axis=None, out=None, mode='raise'):
    """
    """
    (_data, _mask) = (self._data, self._mask)
    cls = type(self)
    maskindices = getmask(indices)
    if maskindices is not nomask:
        indices = indices.filled(0)
    if out is None:
        out = _data.take(indices, axis=axis, mode=mode)[...].view(cls)
    else:
        np.take(_data, indices, axis=axis, mode=mode, out=out)
    if isinstance(out, MaskedArray):
        if _mask is nomask:
            outmask = maskindices
        else:
            outmask = _mask.take(indices, axis=axis, mode=mode)
            outmask |= maskindices
        out.__setmask__(outmask)
    return out[()]
copy = _arraymethod('copy')
diagonal = _arraymethod('diagonal')
flatten = _arraymethod('flatten')
repeat = _arraymethod('repeat')
squeeze = _arraymethod('squeeze')
swapaxes = _arraymethod('swapaxes')
T = property(fget=lambda self: self.transpose())
transpose = _arraymethod('transpose')
def tolist(self, fill_value=None):
    """
    Return the data portion of the masked array as a hierarchical Python
    list.
    Data items are converted to the nearest compatible Python type.
    Masked values are converted to `fill_value`. If `fill_value` is None,
    the corresponding entries in the output list will be ``None``.
    Parameters
    -----
    fill_value : scalar, optional
        The value to use for invalid entries. Default is None.
    Returns
    -----
    result : list
        The Python list representation of the masked array.
    Examples

```



```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

4843: (4)
4844: (8)
4845: (8)
4846: (8)
4847: (10)
4848: (8)
4849: (8)
4850: (8)
4851: (12)
4852: (8)
4853: (8)
4854: (4)
4855: (8)
4856: (8)
4857: (8)
4858: (8)
4859: (8)
4860: (8)
4861: (8)
4862: (8)
4863: (8)
4864: (8)
4865: (8)
4866: (12)
4867: (12)
corresponding
4868: (12)
4869: (8)
4870: (8)
4871: (8)
is
4872: (8)
4873: (8)
4874: (8)
4875: (8)
4876: (8)
4877: (8)
4878: (10)
4879: (16)
4880: (16)
4881: (10)
4882: (16)
4883: (16)
4884: (10)
4885: (8)
4886: (8)
4887: (15)
4888: (15)
4889: (14)
4890: (8)
4891: (8)
4892: (8)
4893: (8)
4894: (12)
4895: (8)
4896: (8)
4897: (28)
4898: (8)
4899: (8)
4900: (8)
4901: (4)
4902: (4)
4903: (8)
4904: (8)
4905: (8)
4906: (8)
4907: (8)
4908: (8)
4909: (4)

        def tofile(self, fid, sep="", format="%s"):
            """
            Save a masked array to a file in binary format.
            .. warning::
                This function is not implemented yet.
            Raises
            -----
            NotImplementedError
                When `tofile` is called.
            """
            raise NotImplementedError("MaskedArray.tofile() not implemented yet.")

    def toflex(self):
        """
        Transforms a masked array into a flexible-type array.
        The flexible type array that is returned will have two fields:
        * the ``_data`` field stores the ``_data`` part of the array.
        * the ``_mask`` field stores the ``_mask`` part of the array.
        Parameters
        -----
        None
        Returns
        -----
        record : ndarray
            A new flexible-type `ndarray` with two fields: the first element
            containing a value, the second element containing the
            mask boolean. The returned record shape matches self.shape.

    Notes
    -----
    A side-effect of transforming a masked array into a flexible `ndarray`
    is
    that meta information (``fill_value``, ...) will be lost.

    Examples
    -----
    >>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
    >>> x
    masked_array(
        data=[[1, --, 3],
              [--, 5, --],
              [7, --, 9]],
        mask=[[False, True, False],
              [True, False, True],
              [False, True, False]],
        fill_value=999999)
    >>> x.toflex()
    array([(1, False), (2, True), (3, False)],
          [(4, True), (5, False), (6, True)],
          [(7, False), (8, True), (9, False)]],
          dtype=[('_data', '<i8'), ('_mask', '?')])
    """
    dtype = self.dtype
    _mask = self._mask
    if _mask is None:
        _mask = make_mask_none(self.shape, dtype)
    mdtype = self._mask.dtype
    record = np.ndarray(shape=self.shape,
                        dtype=[('_data', dtype), ('_mask', mdtype)])
    record['_data'] = self._data
    record['_mask'] = self._mask
    return record

    torecords = toflex

    def __getstate__(self):
        """
        Return the internal state of the masked array, for pickling
        purposes.
        """
        cf = 'CF'[self.flags.fnc]
        data_state = super().__reduce__()[2]
        return data_state + (getmaskarray(self).tobytes(cf), self._fill_value)

    def __setstate__(self, state):

```

```

4910: (8)         """Restore the internal state of the masked array, for
4911: (8)         pickling purposes. ``state`` is typically the output of the
4912: (8)         ``__getstate__`` output, and is a 5-tuple:
4913: (8)         - class name
4914: (8)         - a tuple giving the shape of the data
4915: (8)         - a typecode for the data
4916: (8)         - a binary string for the data
4917: (8)         - a binary string for the mask.
4918: (8)         """
4919: (8)         (_, shp, typ, isf, msk, flv) = state
4920: (8)         super().__setstate__((shp, typ, isf, raw))
4921: (8)         self._mask.__setstate__((shp, make_mask_descr(typ), isf, msk))
4922: (8)         self.fill_value = flv
4923: (4)         def __reduce__(self):
4924: (8)             """Return a 3-tuple for pickling a MaskedArray.
4925: (8)             """
4926: (8)             return (_mareconstruct,
4927: (16)                 (self.__class__, self._baseclass, (0,), 'b',),
4928: (16)                 self.__getstate__())
4929: (4)         def __deepcopy__(self, memo=None):
4930: (8)             from copy import deepcopy
4931: (8)             copied = MaskedArray.__new__(type(self), self, copy=True)
4932: (8)             if memo is None:
4933: (12)                 memo = {}
4934: (8)                 memo[id(self)] = copied
4935: (8)                 for (k, v) in self.__dict__.items():
4936: (12)                     copied.__dict__[k] = deepcopy(v, memo)
4937: (8)                     if self.dtype.hasobject:
4938: (12)                         copied._data[...] = deepcopy(copied._data)
4939: (8)             return copied
4940: (0)         def _mareconstruct(subtype, baseclass, baseshape, basetype,):
4941: (4)             """Internal function that builds a new MaskedArray from the
4942: (4)             information stored in a pickle.
4943: (4)             """
4944: (4)             _data = ndarray.__new__(baseclass, baseshape, basetype)
4945: (4)             _mask = ndarray.__new__(ndarray, baseshape, make_mask_descr(basetype))
4946: (4)             return subtype.__new__(subtype, _data, mask=_mask, dtype=basetype,)
4947: (0)         class mvoid(MaskedArray):
4948: (4)             """
4949: (4)                 Fake a 'void' object to use for masked array with structured dtypes.
4950: (4)             """
4951: (4)             def __new__(self, data, mask=nomask, dtype=None, fill_value=None,
4952: (16)                 hardmask=False, copy=False, subok=True):
4953: (8)                 _data = np.array(data, copy=copy, subok=subok, dtype=dtype)
4954: (8)                 _data = _data.view(self)
4955: (8)                 _data._hardmask = hardmask
4956: (8)                 if mask is not nomask:
4957: (12)                     if isinstance(mask, np.void):
4958: (16)                         _data._mask = mask
4959: (12)                     else:
4960: (16)                         try:
4961: (20)                             _data._mask = np.void(mask)
4962: (16)                         except TypeError:
4963: (20)                             mdtype = make_mask_descr(dtype)
4964: (20)                             _data._mask = np.array(mask, dtype=mdtype)[()]
4965: (8)                         if fill_value is not None:
4966: (12)                             _data.fill_value = fill_value
4967: (8)                         return _data
4968: (4)             @property
4969: (4)             def _data(self):
4970: (8)                 return super().__data__()
4971: (4)             def __getitem__(self, indx):
4972: (8)                 """
4973: (8)                     Get the index.
4974: (8)                 """
4975: (8)                 m = self._mask
4976: (8)                 if isinstance(m[indx], ndarray):
4977: (12)                     return masked_array(
4978: (16)                         data=self._data[indx], mask=m[indx],

```

```

4979: (16)           fill_value=self._fill_value[indx],
4980: (16)           hard_mask=self._hardmask)
4981: (8)           if m is not nomask and m[indx]:
4982: (12)             return masked
4983: (8)           return self._data[indx]
4984: (4)           def __setitem__(self, indx, value):
4985: (8)             self._data[indx] = value
4986: (8)             if self._hardmask:
4987: (12)               self._mask[indx] |= getattr(value, "_mask", False)
4988: (8)             else:
4989: (12)               self._mask[indx] = getattr(value, "_mask", False)
4990: (4)           def __str__(self):
4991: (8)             m = self._mask
4992: (8)             if m is nomask:
4993: (12)               return str(self._data)
4994: (8)             rdtype = _replace_dtype_fields(self._data.dtype, "O")
4995: (8)             data_arr = super().__data
4996: (8)             res = data_arr.astype(rdtype)
4997: (8)             _recursive_printoption(res, self._mask, masked_print_option)
4998: (8)             return str(res)
4999: (4)           __repr__ = __str__
5000: (4)           def __iter__(self):
5001: (8)             "Defines an iterator for mvoid"
5002: (8)             (_data, _mask) = (self._data, self._mask)
5003: (8)             if _mask is nomask:
5004: (12)               yield from _data
5005: (8)             else:
5006: (12)               for (d, m) in zip(_data, _mask):
5007: (16)                 if m:
5008: (20)                   yield masked
5009: (16)                 else:
5010: (20)                   yield d
5011: (4)           def __len__(self):
5012: (8)             return self._data.__len__()
5013: (4)           def filled(self, fill_value=None):
5014: (8)             """
5015: (8)             Return a copy with masked fields filled with a given value.
5016: (8)             Parameters
5017: (8)             -----
5018: (8)             fill_value : array_like, optional
5019: (12)               The value to use for invalid entries. Can be scalar or
5020: (12)               non-scalar. If latter is the case, the filled array should
5021: (12)               be broadcastable over input array. Default is None, in
5022: (12)               which case the `fill_value` attribute is used instead.
5023: (8)             Returns
5024: (8)             -----
5025: (8)             filled_void
5026: (12)               A `np.void` object
5027: (8)             See Also
5028: (8)             -----
5029: (8)             MaskedArray.filled
5030: (8)             """
5031: (8)             return asarray(self).filled(fill_value)[()]
5032: (4)           def tolist(self):
5033: (8)             """
5034: (4)             Transforms the mvoid object into a tuple.
5035: (4)             Masked fields are replaced by None.
5036: (4)             Returns
5037: (4)             -----
5038: (4)             returned_tuple
5039: (8)               Tuple of fields
5040: (8)             """
5041: (8)             _mask = self._mask
5042: (8)             if _mask is nomask:
5043: (12)               return self._data.tolist()
5044: (8)             result = []
5045: (8)             for (d, m) in zip(self._data, self._mask):
5046: (12)               if m:
5047: (16)                 result.append(None)

```

```

5048: (12)
5049: (16)
5050: (8)
5051: (0)
5052: (4)
5053: (4)
5054: (4)
5055: (4)
5056: (4)
5057: (4)
5058: (4)
5059: (8)
5060: (4)
5061: (4)
5062: (4)
5063: (8)
5064: (4)
5065: (4)
5066: (4)
5067: (4)
5068: (4)
5069: (4)
5070: (4)
5071: (4)
5072: (4)
5073: (4)
5074: (11)
5075: (11)
5076: (4)
5077: (4)
5078: (4)
5079: (6)
5080: (12)
5081: (12)
5082: (6)
5083: (12)
5084: (12)
5085: (6)
5086: (4)
5087: (4)
5088: (4)
5089: (4)
5090: (4)
5091: (4)
5092: (4)
5093: (4)
5094: (0)
5095: (0)
5096: (0)
5097: (4)
5098: (4)
5099: (4)
5100: (8)
5101: (4)
5102: (8)
5103: (12)
5104: (12)
5105: (12)
5106: (12)
5107: (12)
5108: (8)
5109: (4)
5110: (8)
5111: (12)
5112: (8)
5113: (12)
5114: (8)
5115: (12)
5116: (12)

        else:
            result.append(d.item())
        return tuple(result)
def isMaskedArray(x):
    """
    Test whether input is an instance of MaskedArray.
    This function returns True if `x` is an instance of MaskedArray
    and returns False otherwise. Any object is accepted as input.
    Parameters
    -----
    x : object
        Object to test.
    Returns
    -----
    result : bool
        True if `x` is a MaskedArray.
    See Also
    -----
    isMA : Alias to isMaskedArray.
    isarray : Alias to isMaskedArray.
    Examples
    -----
    >>> import numpy.ma as ma
    >>> a = np.eye(3, 3)
    >>> a
    array([[ 1.,  0.,  0.],
           [ 0.,  1.,  0.],
           [ 0.,  0.,  1.]])
    >>> m = ma.masked_values(a, 0)
    >>> m
    masked_array(
        data=[[1.0, --, --],
              [--, 1.0, --],
              [--, --, 1.0]],
        mask=[[False, True, True],
              [True, False, True],
              [True, True, False]],
        fill_value=0.0)
    >>> ma.isMaskedArray(a)
    False
    >>> ma.isMaskedArray(m)
    True
    >>> ma.isMaskedArray([0, 1, 2])
    False
    """
    return isinstance(x, MaskedArray)
isarray = isMaskedArray
isMA = isMaskedArray # backward compatibility
class MaskedConstant(MaskedArray):
    __singleton = None
    @classmethod
    def __has_singleton(cls):
        return cls.__singleton is not None and type(cls.__singleton) is cls
    def __new__(cls):
        if not cls.__has_singleton():
            data = np.array(0.)
            mask = np.array(True)
            data.flags.writeable = False
            mask.flags.writeable = False
            cls.__singleton = MaskedArray(data, mask=mask).view(cls)
        return cls.__singleton
    def __array_finalize__(self, obj):
        if not self.__has_singleton():
            return super().__array_finalize__(obj)
        elif self is self.__singleton:
            pass
        else:
            self.__class__ = MaskedArray
            MaskedArray.__array_finalize__(self, obj)

```

```

5117: (4) def __array_prepare__(self, obj, context=None):
5118: (8)     return self.view(MaskedArray).__array_prepare__(obj, context)
5119: (4) def __array_wrap__(self, obj, context=None):
5120: (8)     return self.view(MaskedArray).__array_wrap__(obj, context)
5121: (4) def __str__(self):
5122: (8)     return str(masked_print_option._display)
5123: (4) def __repr__(self):
5124: (8)     if self is MaskedConstant.__singleton:
5125: (12)         return 'masked'
5126: (8)     else:
5127: (12)         return object.__repr__(self)
5128: (4) def __format__(self, format_spec):
5129: (8)     try:
5130: (12)         return object.__format__(self, format_spec)
5131: (8)     except TypeError:
5132: (12)         warnings.warn(
5133: (16)             "Format strings passed to MaskedConstant are ignored, but in
future may "
5134: (16)             "error or produce different behavior",
5135: (16)             FutureWarning, stacklevel=2
5136: (12)         )
5137: (12)         return object.__format__(self, "")
5138: (4) def __reduce__(self):
5139: (8)     """Override of MaskedArray's __reduce__.
5140: (8)     """
5141: (8)     return (self.__class__, ())
5142: (4) def __iop__(self, other):
5143: (8)     return self
5144: (4)     __iadd__ = \
5145: (4)     __isub__ = \
5146: (4)     __imul__ = \
5147: (4)     __ifloordiv__ = \
5148: (4)     __itruediv__ = \
5149: (4)     __ipow__ = \
5150: (8)     __iop__
5151: (4)     del __iop__ # don't leave this around
5152: (4) def copy(self, *args, **kwargs):
5153: (8)     """ Copy is a no-op on the maskedconstant, as it is a scalar """
5154: (8)     return self
5155: (4) def __copy__(self):
5156: (8)     return self
5157: (4) def __deepcopy__(self, memo):
5158: (8)     return self
5159: (4) def __setattr__(self, attr, value):
5160: (8)     if not self.__has_singleton():
5161: (12)         return super().__setattr__(attr, value)
5162: (8)     elif self is self.__singleton:
5163: (12)         raise AttributeError(
5164: (16)             f"attributes of {self!r} are not writeable")
5165: (8)     else:
5166: (12)         return super().__setattr__(attr, value)
5167: (0) masked = masked_singleton = MaskedConstant()
5168: (0) masked_array = MaskedArray
5169: (0) def array(data, dtype=None, copy=False, order=None,
5170: (10)             mask=nomask, fill_value=None, keep_mask=True,
5171: (10)             hard_mask=False, shrink=True, subok=True, ndmin=0):
5172: (4)     """
5173: (4)     Shortcut to MaskedArray.
5174: (4)     The options are in a different order for convenience and backwards
5175: (4)     compatibility.
5176: (4)     """
5177: (4)     return MaskedArray(data, mask=mask, dtype=dtype, copy=copy,
5178: (23)                 subok=subok, keep_mask=keep_mask,
5179: (23)                 hard_mask=hard_mask, fill_value=fill_value,
5180: (23)                 ndmin=ndmin, shrink=shrink, order=order)
5181: (0) array.__doc__ = masked_array.__doc__
5182: (0) def is_masked(x):
5183: (4)     """
5184: (4)     Determine whether input has masked values.

```

```

5185: (4)           Accepts any object as input, but always returns False unless the
5186: (4)           input is a MaskedArray containing masked values.
5187: (4)           Parameters
5188: (4)           -----
5189: (4)           x : array_like
5190: (8)             Array to check for masked values.
5191: (4)           Returns
5192: (4)           -----
5193: (4)           result : bool
5194: (8)             True if `x` is a MaskedArray with masked values, False otherwise.
5195: (4)           Examples
5196: (4)           -----
5197: (4)             >>> import numpy.ma as ma
5198: (4)             >>> x = ma.masked_equal([0, 1, 0, 2, 3], 0)
5199: (4)             >>> x
5200: (4)               masked_array(data=[--, 1, --, 2, 3],
5201: (17)                 mask=[ True, False, True, False, False],
5202: (11)                   fill_value=0)
5203: (4)             >>> ma.is_masked(x)
5204: (4)               True
5205: (4)             >>> x = ma.masked_equal([0, 1, 0, 2, 3], 42)
5206: (4)             >>> x
5207: (4)               masked_array(data=[0, 1, 0, 2, 3],
5208: (17)                 mask=False,
5209: (11)                   fill_value=42)
5210: (4)             >>> ma.is_masked(x)
5211: (4)               False
5212: (4)             Always returns False if `x` isn't a MaskedArray.
5213: (4)             >>> x = [False, True, False]
5214: (4)             >>> ma.is_masked(x)
5215: (4)               False
5216: (4)             >>> x = 'a string'
5217: (4)             >>> ma.is_masked(x)
5218: (4)               False
5219: (4)             """
5220: (4)             m = getmask(x)
5221: (4)             if m is nomask:
5222: (8)               return False
5223: (4)             elif m.any():
5224: (8)               return True
5225: (4)             return False
5226: (0)             class _extrema_operation(_MaskedUFunc):
5227: (4)               """
5228: (4)                 Generic class for maximum/minimum functions.
5229: (4)                 .. note::
5230: (6)                   This is the base class for `_maximum_operation` and
5231: (6)                   `_minimum_operation`.
5232: (4)               """
5233: (4)               def __init__(self, ufunc, compare, fill_value):
5234: (8)                 super().__init__(ufunc)
5235: (8)                 self.compare = compare
5236: (8)                 self.fill_value_func = fill_value
5237: (4)               def __call__(self, a, b):
5238: (8)                 "Executes the call behavior."
5239: (8)                 return where(self.compare(a, b), a, b)
5240: (4)               def reduce(self, target, axis=np._NoValue):
5241: (8)                 "Reduce target along the given axis."
5242: (8)                 target = narray(target, copy=False, subok=True)
5243: (8)                 m = getmask(target)
5244: (8)                 if axis is np._NoValue and target.ndim > 1:
5245: (12)                   warnings.warn(
5246: (16)                     f"In the future the default for ma.{self.__name__}.reduce will
be axis=0, "
5247: (16)                     f"not the current None, to match np.{self.__name__}.reduce. "
5248: (16)                     "Explicitly pass 0 or None to silence this warning.", "
5249: (16)                         MaskedArrayFutureWarning, stacklevel=2)
5250: (12)                   axis = None
5251: (8)                   if axis is not np._NoValue:
5252: (12)                     kwargs = dict(axis=axis)

```

```

5253: (8)           else:
5254: (12)          kwargs = dict()
5255: (8)          if m is nomask:
5256: (12)            t = self.f.reduce(target, **kwargs)
5257: (8)          else:
5258: (12)            target = target.filled(
5259: (16)              self.fill_value_func(target)).view(type(target))
5260: (12)            t = self.f.reduce(target, **kwargs)
5261: (12)            m = umath.logical_and.reduce(m, **kwargs)
5262: (12)            if hasattr(t, '_mask'):
5263: (16)              t._mask = m
5264: (12)            elif m:
5265: (16)              t = masked
5266: (8)          return t
5267: (4)          def outer(self, a, b):
5268: (8)            "Return the function applied to the outer product of a and b."
5269: (8)            ma = getmask(a)
5270: (8)            mb = getmask(b)
5271: (8)            if ma is nomask and mb is nomask:
5272: (12)              m = nomask
5273: (8)            else:
5274: (12)              ma = getmaskarray(a)
5275: (12)              mb = getmaskarray(b)
5276: (12)              m = logical_or.outer(ma, mb)
5277: (8)            result = self.f.outer(filled(a), filled(b))
5278: (8)            if not isinstance(result, MaskedArray):
5279: (12)              result = result.view(MaskedArray)
5280: (8)            result._mask = m
5281: (8)            return result
5282: (0)          def min(obj, axis=None, out=None, fill_value=None, keepdims=np._NoValue):
5283: (4)            kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
5284: (4)            try:
5285: (8)              return obj.min(axis=axis, fill_value=fill_value, out=out, **kwargs)
5286: (4)            except (AttributeError, TypeError):
5287: (8)              return asanyarray(obj).min(axis=axis, fill_value=fill_value,
5288: (35)                  out=out, **kwargs)
5289: (0)          min.__doc__ = MaskedArray.min.__doc__
5290: (0)          def max(obj, axis=None, out=None, fill_value=None, keepdims=np._NoValue):
5291: (4)            kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
5292: (4)            try:
5293: (8)              return obj.max(axis=axis, fill_value=fill_value, out=out, **kwargs)
5294: (4)            except (AttributeError, TypeError):
5295: (8)              return asanyarray(obj).max(axis=axis, fill_value=fill_value,
5296: (35)                  out=out, **kwargs)
5297: (0)          max.__doc__ = MaskedArray.max.__doc__
5298: (0)          def ptp(obj, axis=None, out=None, fill_value=None, keepdims=np._NoValue):
5299: (4)            kwargs = {} if keepdims is np._NoValue else {'keepdims': keepdims}
5300: (4)            try:
5301: (8)              return obj.ptp(axis, out=out, fill_value=fill_value, **kwargs)
5302: (4)            except (AttributeError, TypeError):
5303: (8)              return asanyarray(obj).ptp(axis=axis, fill_value=fill_value,
5304: (35)                  out=out, **kwargs)
5305: (0)          ptp.__doc__ = MaskedArray.ptp.__doc__
5306: (0)          class _frommethod:
5307: (4)            """
5308: (4)              Define functions from existing MaskedArray methods.
5309: (4)              Parameters
5310: (4)              -----
5311: (4)              methodname : str
5312: (8)                Name of the method to transform.
5313: (4)              """
5314: (4)              def __init__(self, methodname, reversed=False):
5315: (8)                self.__name__ = methodname
5316: (8)                self.__doc__ = self.getdoc()
5317: (8)                self.reversed = reversed
5318: (4)              def getdoc(self):
5319: (8)                "Return the doc of the function (from the doc of the method)."
5320: (8)                meth = getattr(MaskedArray, self.__name__, None) or \
5321: (12)                  getattr(np, self.__name__, None)

```

```

5322: (8)             signature = self.__name__ + get_object_signature(meth)
5323: (8)             if meth is not None:
5324: (12)                 doc = """%s\n%s""" % (
5325: (16)                     signature, getattr(meth, '__doc__', None))
5326: (12)                 return doc
5327: (4)             def __call__(self, a, *args, **params):
5328: (8)                 if self.reversed:
5329: (12)                     args = list(args)
5330: (12)                     a, args[0] = args[0], a
5331: (8)                     marr = asanyarray(a)
5332: (8)                     method_name = self.__name__
5333: (8)                     method = getattr(type(marr), method_name, None)
5334: (8)                     if method is None:
5335: (12)                         method = getattr(np, method_name)
5336: (8)                     return method(marr, *args, **params)
5337: (0)             all = _frommethod('all')
5338: (0)             anomalies = anom = _frommethod('anom')
5339: (0)             any = _frommethod('any')
5340: (0)             compress = _frommethod('compress', reversed=True)
5341: (0)             cumprod = _frommethod('cumprod')
5342: (0)             cumsum = _frommethod('cumsum')
5343: (0)             copy = _frommethod('copy')
5344: (0)             diagonal = _frommethod('diagonal')
5345: (0)             harden_mask = _frommethod('harden_mask')
5346: (0)             ids = _frommethod('ids')
5347: (0)             maximum = _extrema_operation(umath.maximum, greater, maximum_fill_value)
5348: (0)             mean = _frommethod('mean')
5349: (0)             minimum = _extrema_operation(umath.minimum, less, minimum_fill_value)
5350: (0)             nonzero = _frommethod('nonzero')
5351: (0)             prod = _frommethod('prod')
5352: (0)             product = _frommethod('prod')
5353: (0)             ravel = _frommethod('ravel')
5354: (0)             repeat = _frommethod('repeat')
5355: (0)             shrink_mask = _frommethod('shrink_mask')
5356: (0)             soften_mask = _frommethod('soften_mask')
5357: (0)             std = _frommethod('std')
5358: (0)             sum = _frommethod('sum')
5359: (0)             swapaxes = _frommethod('swapaxes')
5360: (0)             trace = _frommethod('trace')
5361: (0)             var = _frommethod('var')
5362: (0)             count = _frommethod('count')
5363: (0)             def take(a, indices, axis=None, out=None, mode='raise'):
5364: (4)                 """
5365: (4)                 """
5366: (4)                 a = masked_array(a)
5367: (4)                 return a.take(indices, axis=axis, out=out, mode=mode)
5368: (0)             def power(a, b, third=None):
5369: (4)                 """
5370: (4)                     Returns element-wise base array raised to power from second array.
5371: (4)                     This is the masked array version of `numpy.power`. For details see
5372: (4)                     `numpy.power`.
5373: (4)                     See Also
5374: (4)                     -----
5375: (4)                     numpy.power
5376: (4)                     Notes
5377: (4)                     -----
5378: (4)                     The *out* argument to `numpy.power` is not supported, `third` has to be
5379: (4)                     None.
5380: (4)                     Examples
5381: (4)                     -----
5382: (4)                     >>> import numpy.ma as ma
5383: (4)                     >>> x = [11.2, -3.973, 0.801, -1.41]
5384: (4)                     >>> mask = [0, 0, 0, 1]
5385: (4)                     >>> masked_x = ma.masked_array(x, mask)
5386: (4)                     >>> masked_
5387: (4)                     masked_array(data=[11.2, -3.973, 0.801, --],
5388: (13)                         mask=[False, False, False, True],
5389: (7)                         fill_value=1e+20)
5390: (4)                     >>> ma.power(masked_x, 2)

```

```

5391: (4)
5392: (19)
5393: (13)
5394: (7)
5395: (4)
5396: (4)
5397: (4)
5398: (4)
5399: (13)
5400: (7)
5401: (4)
5402: (4)
5403: (13)
5404: (7)
5405: (4)
5406: (4)
5407: (8)
5408: (4)
5409: (4)
5410: (4)
5411: (4)
5412: (4)
5413: (4)
5414: (8)
5415: (4)
5416: (8)
5417: (4)
5418: (8)
5419: (4)
5420: (4)
5421: (4)
5422: (8)
5423: (12)
5424: (8)
5425: (4)
5426: (8)
5427: (12)
5428: (8)
5429: (12)
5430: (8)
5431: (4)
5432: (0)
5433: (0)
5434: (0)
      fill_value=None):
5435: (4)
      "Function version of the eponymous method."
5436: (4)
5437: (4)
5438: (8)
5439: (4)
5440: (8)
5441: (25)
5442: (4)
5443: (8)
5444: (0)
5445: (0)
      def sort(a, axis=-1, kind=None, order=None, endwith=True,
      fill_value=None):
5446: (4)
5447: (4)
      Return a sorted copy of the masked array.
5448: (4)
      Equivalent to creating a copy of the array
5449: (4)
      and applying the MaskedArray ``sort()`` method.
5450: (4)
      Refer to ``MaskedArray.sort`` for the full documentation
5451: (4)
      See Also
      -----
5452: (4)
5453: (4)
      MaskedArray.sort : equivalent method
5454: (4)
      Examples
      -----
5455: (4)
5456: (4)
5457: (4)
5458: (4)
      >>> import numpy.ma as ma
      >>> x = [11.2, -3.973, 0.801, -1.41]
      >>> mask = [0, 0, 0, 1]

```

```

5459: (4)             >>> masked_x = ma.masked_array(x, mask)
5460: (4)             >>> masked_x
5461: (4)             masked_array(data=[11.2, -3.973, 0.801, --],
5462: (17)                 mask=[False, False, False, True],
5463: (11)                 fill_value=1e+20)
5464: (4)             >>> ma.sort(masked_x)
5465: (4)             masked_array(data=[-3.973, 0.801, 11.2, --],
5466: (17)                 mask=[False, False, False, True],
5467: (11)                 fill_value=1e+20)
5468: (4)             """
5469: (4)             a = np.array(a, copy=True, subok=True)
5470: (4)             if axis is None:
5471: (8)                 a = a.flatten()
5472: (8)                 axis = 0
5473: (4)             if isinstance(a, MaskedArray):
5474: (8)                 a.sort(axis=axis, kind=kind, order=order,
5475: (15)                     endwith=endwith, fill_value=fill_value)
5476: (4)             else:
5477: (8)                 a.sort(axis=axis, kind=kind, order=order)
5478: (4)             return a
5479: (0)             def compressed(x):
5480: (4)             """
5481: (4)             Return all the non-masked data as a 1-D array.
5482: (4)             This function is equivalent to calling the "compressed" method of a
5483: (4)             `ma.MaskedArray`, see `ma.MaskedArray.compressed` for details.
5484: (4)             See Also
5485: (4)             -----
5486: (4)             ma.MaskedArray.compressed : Equivalent method.
5487: (4)             Examples
5488: (4)             -----
5489: (4)             Create an array with negative values masked:
5490: (4)             >>> import numpy as np
5491: (4)             >>> x = np.array([[1, -1, 0], [2, -1, 3], [7, 4, -1]])
5492: (4)             >>> masked_x = np.ma.masked_array(x, mask=x < 0)
5493: (4)             >>> masked_x
5494: (4)             masked_array(
5495: (6)                 data=[[1, --, 0],
5496: (12)                   [2, --, 3],
5497: (12)                   [7, 4, --]],
5498: (6)                 mask=[[False, True, False],
5499: (12)                   [False, True, False],
5500: (12)                   [False, False, True]],
5501: (6)                 fill_value=999999)
5502: (4)             Compress the masked array into a 1-D array of non-masked values:
5503: (4)             >>> np.ma.compressed(masked_x)
5504: (4)             array([1, 0, 2, 3, 7, 4])
5505: (4)             """
5506: (4)             return asanyarray(x).compressed()
5507: (0)             def concatenate(arrays, axis=0):
5508: (4)             """
5509: (4)             Concatenate a sequence of arrays along the given axis.
5510: (4)             Parameters
5511: (4)             -----
5512: (4)             arrays : sequence of array_like
5513: (8)                 The arrays must have the same shape, except in the dimension
5514: (8)                 corresponding to `axis` (the first, by default).
5515: (4)             axis : int, optional
5516: (8)                 The axis along which the arrays will be joined. Default is 0.
5517: (4)             Returns
5518: (4)             -----
5519: (4)             result : MaskedArray
5520: (8)                 The concatenated array with any masked entries preserved.
5521: (4)             See Also
5522: (4)             -----
5523: (4)             numpy.concatenate : Equivalent function in the top-level NumPy module.
5524: (4)             Examples
5525: (4)             -----
5526: (4)             >>> import numpy.ma as ma
5527: (4)             >>> a = ma.arange(3)

```

```

5528: (4)          >>> a[1] = ma.masked
5529: (4)          >>> b = ma.arange(2, 5)
5530: (4)          >>> a
5531: (4)          masked_array(data=[0, --, 2],
5532: (17)             mask=[False, True, False],
5533: (11)             fill_value=999999)
5534: (4)          >>> b
5535: (4)          masked_array(data=[2, 3, 4],
5536: (17)             mask=False,
5537: (11)             fill_value=999999)
5538: (4)          >>> ma.concatenate([a, b])
5539: (4)          masked_array(data=[0, --, 2, 2, 3, 4],
5540: (17)             mask=[False, True, False, False, False, False],
5541: (11)             fill_value=999999)
5542: (4)          """
5543: (4)          d = np.concatenate([getdata(a) for a in arrays], axis)
5544: (4)          rcls = get_masked_subclass(*arrays)
5545: (4)          data = d.view(rcls)
5546: (4)          for x in arrays:
5547: (8)              if getmask(x) is not nomask:
5548: (12)                  break
5549: (4)          else:
5550: (8)              return data
5551: (4)          dm = np.concatenate([getmaskarray(a) for a in arrays], axis)
5552: (4)          dm = dm.reshape(d.shape)
5553: (4)          data._mask = _shrink_mask(dm)
5554: (4)          return data
5555: (0)          def diag(v, k=0):
5556: (4)          """
5557: (4)          Extract a diagonal or construct a diagonal array.
5558: (4)          This function is the equivalent of `numpy.diag` that takes masked
5559: (4)          values into account, see `numpy.diag` for details.
5560: (4)          See Also
5561: (4)          -----
5562: (4)          numpy.diag : Equivalent function for ndarrays.
5563: (4)          Examples
5564: (4)          -----
5565: (4)          Create an array with negative values masked:
5566: (4)          >>> import numpy as np
5567: (4)          >>> x = np.array([[11.2, -3.973, 18], [0.801, -1.41, 12], [7, 33, -12]])
5568: (4)          >>> masked_x = np.ma.masked_array(x, mask=x < 0)
5569: (4)          >>> masked_x
5570: (4)          masked_array(
5571: (6)              data=[[11.2, --, 18.0],
5572: (12)                  [0.801, --, 12.0],
5573: (12)                  [7.0, 33.0, --]],
5574: (6)              mask=[[False, True, False],
5575: (12)                  [False, True, False],
5576: (12)                  [False, False, True]],
5577: (6)              fill_value=1e+20)
5578: (4)          Isolate the main diagonal from the masked array:
5579: (4)          >>> np.ma.diag(masked_x)
5580: (4)          masked_array(data=[11.2, --, --],
5581: (17)              mask=[False, True, True],
5582: (11)              fill_value=1e+20)
5583: (4)          Isolate the first diagonal below the main diagonal:
5584: (4)          >>> np.ma.diag(masked_x, -1)
5585: (4)          masked_array(data=[0.801, 33.0],
5586: (17)              mask=[False, False],
5587: (11)              fill_value=1e+20)
5588: (4)          """
5589: (4)          output = np.diag(v, k).view(MaskedArray)
5590: (4)          if getmask(v) is not nomask:
5591: (8)              output._mask = np.diag(v._mask, k)
5592: (4)          return output
5593: (0)          def left_shift(a, n):
5594: (4)          """
5595: (4)          Shift the bits of an integer to the left.
5596: (4)          This is the masked array version of `numpy.left_shift`, for details

```

```

5597: (4)          see that function.
5598: (4)          See Also
5599: (4)
5600: (4)          -----
5601: (4)          numpy.left_shift
5602: (4)          """
5603: (4)          m = getmask(a)
5604: (8)          if m is nomask:
5605: (8)              d = umath.left_shift(filled(a), n)
5606: (8)              return masked_array(d)
5607: (8)          else:
5608: (8)              d = umath.left_shift(filled(a, 0), n)
5609: (8)              return masked_array(d, mask=m)
5610: (0)          def right_shift(a, n):
5611: (4)          """
5612: (4)          Shift the bits of an integer to the right.
5613: (4)          This is the masked array version of `numpy.right_shift`, for details
5614: (4)          see that function.
5615: (4)          See Also
5616: (4)
5617: (4)          -----
5618: (4)          numpy.right_shift
5619: (4)          Examples
5620: (4)
5621: (4)          -----
5622: (4)          >>> import numpy.ma as ma
5623: (4)          >>> x = [11, 3, 8, 1]
5624: (4)          >>> mask = [0, 0, 0, 1]
5625: (4)          >>> masked_x = ma.masked_array(x, mask)
5626: (4)          >>> masked_x
5627: (4)          masked_array(data=[11, 3, 8, --],
5628: (4)                  mask=[False, False, False, True],
5629: (4)                  fill_value=999999)
5630: (4)          >>> ma.right_shift(masked_x,1)
5631: (4)          masked_array(data=[5, 1, 4, --],
5632: (4)                  mask=[False, False, False, True],
5633: (4)                  fill_value=999999)
5634: (4)          """
5635: (4)          m = getmask(a)
5636: (4)          if m is nomask:
5637: (8)              d = umath.right_shift(filled(a), n)
5638: (8)              return masked_array(d)
5639: (0)          def put(a, indices, values, mode='raise'):
5640: (4)          """
5641: (4)          Set storage-indexed locations to corresponding values.
5642: (4)          This function is equivalent to `MaskedArray.put`, see that method
5643: (4)          for details.
5644: (4)          See Also
5645: (4)
5646: (4)          -----
5647: (4)          MaskedArray.put
5648: (4)          """
5649: (4)          try:
5650: (4)              return a.put(indices, values, mode=mode)
5651: (4)          except AttributeError:
5652: (4)              return narray(a, copy=False).put(indices, values, mode=mode)
5653: (0)          def putmask(a, mask, values): # , mode='raise'):
5654: (4)          """
5655: (4)          Changes elements of an array based on conditional and input values.
5656: (4)          This is the masked array version of `numpy.putmask`, for details see
5657: (4)          `numpy.putmask`.
5658: (4)          See Also
5659: (4)
5660: (4)          -----
5661: (4)          numpy.putmask
5662: (4)          Notes
5663: (4)
5664: (4)          -----
5665: (4)          Using a masked array as `values` will **not** transform a `ndarray` into
5666: (4)          a `MaskedArray`.
5667: (4)          """
5668: (4)          if not isinstance(a, MaskedArray):

```

```

5666: (8)           a = a.view(MaskedArray)
5667: (4)           (valdata, valmask) = (getdata(values), getmask(values))
5668: (4)           if getmask(a) is nomask:
5669: (8)             if valmask is not nomask:
5670: (12)               a._sharedmask = True
5671: (12)               a._mask = make_mask_none(a.shape, a.dtype)
5672: (12)               np.copyto(a._mask, valmask, where=mask)
5673: (4)           elif a._hardmask:
5674: (8)             if valmask is not nomask:
5675: (12)               m = a._mask.copy()
5676: (12)               np.copyto(m, valmask, where=mask)
5677: (12)               a.mask |= m
5678: (4)           else:
5679: (8)             if valmask is nomask:
5680: (12)               valmask = getmaskarray(values)
5681: (8)               np.copyto(a._mask, valmask, where=mask)
5682: (4)               np.copyto(a._data, valdata, where=mask)
5683: (4)           return
5684: (0)           def transpose(a, axes=None):
5685: (4)             """
5686: (4)             Permute the dimensions of an array.
5687: (4)             This function is exactly equivalent to `numpy.transpose`.
5688: (4)             See Also
5689: (4)             -----
5690: (4)             numpy.transpose : Equivalent function in top-level NumPy module.
5691: (4)             Examples
5692: (4)             -----
5693: (4)             >>> import numpy.ma as ma
5694: (4)             >>> x = ma.arange(4).reshape((2,2))
5695: (4)             >>> x[1, 1] = ma.masked
5696: (4)             >>> x
5697: (4)             masked_array(
5698: (6)               data=[[0, 1],
5699: (12)                 [2, --]],
5700: (6)               mask=[[False, False],
5701: (12)                 [False, True]],
5702: (6)               fill_value=999999)
5703: (4)             >>> ma.transpose(x)
5704: (4)             masked_array(
5705: (6)               data=[[0, 2],
5706: (12)                 [1, --]],
5707: (6)               mask=[[False, False],
5708: (12)                 [False, True]],
5709: (6)               fill_value=999999)
5710: (4)             """
5711: (4)             try:
5712: (8)               return a.transpose(axes)
5713: (4)             except AttributeError:
5714: (8)               return narray(a, copy=False).transpose(axes).view(MaskedArray)
5715: (0)           def reshape(a, new_shape, order='C'):
5716: (4)             """
5717: (4)             Returns an array containing the same data with a new shape.
5718: (4)             Refer to `MaskedArray.reshape` for full documentation.
5719: (4)             See Also
5720: (4)             -----
5721: (4)             MaskedArray.reshape : equivalent function
5722: (4)             """
5723: (4)             try:
5724: (8)               return a.reshape(new_shape, order=order)
5725: (4)             except AttributeError:
5726: (8)               _tmp = narray(a, copy=False).reshape(new_shape, order=order)
5727: (8)               return _tmp.view(MaskedArray)
5728: (0)           def resize(x, new_shape):
5729: (4)             """
5730: (4)             Return a new masked array with the specified size and shape.
5731: (4)             This is the masked equivalent of the `numpy.resize` function. The new
5732: (4)             array is filled with repeated copies of `x` (in the order that the
5733: (4)             data are stored in memory). If `x` is masked, the new array will be
5734: (4)             masked, and the new mask will be a repetition of the old one.

```

```

5735: (4)          See Also
5736: (4)          -----
5737: (4)          numpy.resize : Equivalent function in the top level NumPy module.
5738: (4)          Examples
5739: (4)          -----
5740: (4)          >>> import numpy.ma as ma
5741: (4)          >>> a = ma.array([[1, 2], [3, 4]])
5742: (4)          >>> a[0, 1] = ma.masked
5743: (4)          >>> a
5744: (4)          masked_array(
5745: (6)            data=[[1, --],
5746: (12)           [3, 4]],
5747: (6)           mask=[[False, True],
5748: (12)             [False, False]],
5749: (6)             fill_value=999999)
5750: (4)           >>> np.resize(a, (3, 3))
5751: (4)           masked_array(
5752: (6)             data=[[1, 2, 3],
5753: (12)               [4, 1, 2],
5754: (12)                 [3, 4, 1]],
5755: (6)               mask=False,
5756: (6)               fill_value=999999)
5757: (4)             >>> ma.resize(a, (3, 3))
5758: (4)             masked_array(
5759: (6)               data=[[1, --, 3],
5760: (12)                 [4, 1, --],
5761: (12)                   [3, 4, 1]],
5762: (6)                   mask=[[False, True, False],
5763: (12)                     [False, False, True],
5764: (12)                       [False, False, False]],
5765: (6)                       fill_value=999999)
5766: (4)           A MaskedArray is always returned, regardless of the input type.
5767: (4)           >>> a = np.array([[1, 2], [3, 4]])
5768: (4)           >>> ma.resize(a, (3, 3))
5769: (4)           masked_array(
5770: (6)             data=[[1, 2, 3],
5771: (12)               [4, 1, 2],
5772: (12)                 [3, 4, 1]],
5773: (6)                 mask=False,
5774: (6)                 fill_value=999999)
5775: (4)               """
5776: (4)               m = getmask(x)
5777: (4)               if m is not nomask:
5778: (8)                 m = np.resize(m, new_shape)
5779: (4)               result = np.resize(x, new_shape).view(get_masked_subclass(x))
5780: (4)               if result.ndim:
5781: (8)                 result._mask = m
5782: (4)               return result
5783: (0)             def ndim(obj):
5784: (4)               """
5785: (4)               maskedarray version of the numpy function.
5786: (4)               """
5787: (4)               return np.ndim(getdata(obj))
5788: (0)             ndim.__doc__ = np.ndim.__doc__
5789: (0)             def shape(obj):
5790: (4)               "maskedarray version of the numpy function."
5791: (4)               return np.shape(getdata(obj))
5792: (0)             shape.__doc__ = np.shape.__doc__
5793: (0)             def size(obj, axis=None):
5794: (4)               "maskedarray version of the numpy function."
5795: (4)               return np.size(getdata(obj), axis)
5796: (0)             size.__doc__ = np.size.__doc__
5797: (0)             def diff(a, /, n=1, axis=-1, prepend=np._NoValue, append=np._NoValue):
5798: (4)               """
5799: (4)               Calculate the n-th discrete difference along the given axis.
5800: (4)               The first difference is given by ``out[i] = a[i+1] - a[i]`` along
5801: (4)               the given axis, higher differences are calculated by using `diff`
5802: (4)               recursively.
5803: (4)               Preserves the input mask.

```

```

5804: (4)          Parameters
5805: (4)          -----
5806: (4)          a : array_like
5807: (8)          Input array
5808: (4)          n : int, optional
5809: (8)          The number of times values are differenced. If zero, the input
5810: (8)          is returned as-is.
5811: (4)          axis : int, optional
5812: (8)          The axis along which the difference is taken, default is the
5813: (8)          last axis.
5814: (4)          prepend, append : array_like, optional
5815: (8)          Values to prepend or append to `a` along axis prior to
5816: (8)          performing the difference. Scalar values are expanded to
5817: (8)          arrays with length 1 in the direction of axis and the shape
5818: (8)          of the input array in along all other axes. Otherwise the
5819: (8)          dimension and shape must match `a` except along axis.
5820: (4)          Returns
5821: (4)          -----
5822: (4)          diff : MaskedArray
5823: (8)          The n-th differences. The shape of the output is the same as `a`
5824: (8)          except along `axis` where the dimension is smaller by `n`. The
5825: (8)          type of the output is the same as the type of the difference
5826: (8)          between any two elements of `a`. This is the same as the type of
5827: (8)          `a` in most cases. A notable exception is `datetime64`, which
5828: (8)          results in a `timedelta64` output array.
5829: (4)          See Also
5830: (4)          -----
5831: (4)          numpy.diff : Equivalent function in the top-level NumPy module.
5832: (4)          Notes
5833: (4)          -----
5834: (4)          Type is preserved for boolean arrays, so the result will contain
5835: (4)          `False` when consecutive elements are the same and `True` when they
5836: (4)          differ.
5837: (4)          For unsigned integer arrays, the results will also be unsigned. This
5838: (4)          should not be surprising, as the result is consistent with
5839: (4)          calculating the difference directly:
5840: (4)          >>> u8_arr = np.array([1, 0], dtype=np.uint8)
5841: (4)          >>> np.ma.diff(u8_arr)
5842: (4)          masked_array(data=[255],
5843: (17)              mask=False,
5844: (11)              fill_value=999999,
5845: (16)              dtype=uint8)
5846: (4)          >>> u8_arr[1,...] - u8_arr[0,...]
5847: (4)          255
5848: (4)          If this is not desirable, then the array should be cast to a larger
5849: (4)          integer type first:
5850: (4)          >>> i16_arr = u8_arr.astype(np.int16)
5851: (4)          >>> np.ma.diff(i16_arr)
5852: (4)          masked_array(data=[-1],
5853: (17)              mask=False,
5854: (11)              fill_value=999999,
5855: (16)              dtype=int16)
5856: (4)          Examples
5857: (4)          -----
5858: (4)          >>> a = np.array([1, 2, 3, 4, 7, 0, 2, 3])
5859: (4)          >>> x = np.ma.masked_where(a < 2, a)
5860: (4)          >>> np.ma.diff(x)
5861: (4)          masked_array(data=[--, 1, 1, 3, --, --, 1],
5862: (12)              mask=[ True, False, False, False,  True,  True, False],
5863: (8)              fill_value=999999)
5864: (4)          >>> np.ma.diff(x, n=2)
5865: (4)          masked_array(data=[--, 0, 2, --, --, --],
5866: (16)              mask=[ True, False, False,  True,  True,  True],
5867: (8)              fill_value=999999)
5868: (4)          >>> a = np.array([[1, 3, 1, 5, 10], [0, 1, 5, 6, 8]])
5869: (4)          >>> x = np.ma.masked_equal(a, value=1)
5870: (4)          >>> np.ma.diff(x)
5871: (4)          masked_array(
5872: (8)              data=[--, --, --, 5],

```

```

5873: (16)           [--, --, 1, 2]],  

5874: (8)            mask=[[ True,  True,  True, False],  

5875: (16)              [ True,  True, False, False]],  

5876: (8)            fill_value=1)  

5877: (4)           >>> np.ma.diff(x, axis=0)  

5878: (4)            masked_array(data=[[--, --, --, 1, -2]],  

5879: (12)              mask=[[ True,  True,  True, False, False]],  

5880: (8)              fill_value=1)  

5881: (4)            """  

5882: (4)            if n == 0:  

5883: (8)              return a  

5884: (4)            if n < 0:  

5885: (8)              raise ValueError("order must be non-negative but got " + repr(n))  

5886: (4)            a = np.ma.asanyarray(a)  

5887: (4)            if a.ndim == 0:  

5888: (8)              raise ValueError(  

5889: (12)                  "diff requires input that is at least one dimensional"  

5890: (12)              )  

5891: (4)            combined = []  

5892: (4)            if prepend is not np._NoValue:  

5893: (8)              prepend = np.ma.asanyarray(prepend)  

5894: (8)              if prepend.ndim == 0:  

5895: (12)                  shape = list(a.shape)  

5896: (12)                  shape[axis] = 1  

5897: (12)                  prepend = np.broadcast_to(prepend, tuple(shape))  

5898: (8)                  combined.append(prepend)  

5899: (4)            combined.append(a)  

5900: (4)            if append is not np._NoValue:  

5901: (8)              append = np.ma.asanyarray(append)  

5902: (8)              if append.ndim == 0:  

5903: (12)                  shape = list(a.shape)  

5904: (12)                  shape[axis] = 1  

5905: (12)                  append = np.broadcast_to(append, tuple(shape))  

5906: (8)                  combined.append(append)  

5907: (4)            if len(combined) > 1:  

5908: (8)              a = np.ma.concatenate(combined, axis)  

5909: (4)            return np.diff(a, n, axis)  

5910: (0)          def where(condition, x=_NoValue, y=_NoValue):  

5911: (4)            """  

5912: (4)            Return a masked array with elements from `x` or `y`, depending on  

5913: (4)            condition.  

5914: (8)            .. note::  

5915: (8)              When only `condition` is provided, this function is identical to  

5916: (8)              `nonzero`. The rest of this documentation covers only the case where  

5917: (8)              all three arguments are provided.  

5918: (4)          Parameters  

5919: (4)            condition : array_like, bool  

5920: (8)              Where True, yield `x`, otherwise yield `y`.  

5921: (4)            x, y : array_like, optional  

5922: (8)              Values from which to choose. `x`, `y` and `condition` need to be  

5923: (8)              broadcastable to some shape.  

5924: (4)          Returns  

5925: (4)            out : MaskedArray  

5926: (4)            An masked array with `masked` elements where the condition is masked,  

5927: (8)            elements from `x` where `condition` is True, and elements from `y`  

5928: (8)            elsewhere.  

5929: (8)          See Also  

5930: (4)            numpy.where : Equivalent function in the top-level NumPy module.  

5931: (4)          nonzero : The function that is called when x and y are omitted  

5932: (4)          Examples  

5933: (4)            >>> x = np.ma.array(np.arange(9.).reshape(3, 3), mask=[[0, 1, 0],  

5934: (4)                          ...                                [1, 0, 1],  

5935: (4)                          ...                                [0, 1, 0]])  

5936: (4)            >>> x  

5937: (4)            masked_array(  

5938: (4)            masked_array(  

5939: (4)            masked_array(  

5940: (4)

```

```

5941: (6)             data=[[0.0, --, 2.0],
5942: (12)            [--, 4.0, --],
5943: (12)            [6.0, --, 8.0]],
5944: (6)            mask=[[False, True, False],
5945: (12)            [ True, False,  True],
5946: (12)            [False,  True, False]],
5947: (6)            fill_value=1e+20)
5948: (4) >>> np.ma.where(x > 5, x, -3.1416)
5949: (4) masked_array(
5950: (6)             data=[[-3.1416, --, -3.1416],
5951: (12)            [--, -3.1416, --],
5952: (12)            [6.0, --, 8.0]],
5953: (6)            mask=[[False, True, False],
5954: (12)            [ True, False,  True],
5955: (12)            [False,  True, False]],
5956: (6)            fill_value=1e+20)
5957: (4) """
5958: (4) missing = (x is _NoValue, y is _NoValue).count(True)
5959: (4) if missing == 1:
5960: (8)     raise ValueError("Must provide both 'x' and 'y' or neither.")
5961: (4) if missing == 2:
5962: (8)     return nonzero(condition)
5963: (4) cf = filled(condition, False)
5964: (4) xd = getdata(x)
5965: (4) yd = getdata(y)
5966: (4) cm = getmaskarray(condition)
5967: (4) xm = getmaskarray(x)
5968: (4) ym = getmaskarray(y)
5969: (4) if x is masked and y is not masked:
5970: (8)     xd = np.zeros(), dtype=yd.dtype)
5971: (8)     xm = np.ones(), dtype=ym.dtype)
5972: (4) elif y is masked and x is not masked:
5973: (8)     yd = np.zeros(), dtype=xd.dtype)
5974: (8)     ym = np.ones(), dtype=xm.dtype)
5975: (4) data = np.where(cf, xd, yd)
5976: (4) mask = np.where(cf, xm, ym)
5977: (4) mask = np.where(cm, np.ones(), dtype=mask.dtype), mask)
5978: (4) mask = _shrink_mask(mask)
5979: (4) return masked_array(data, mask=mask)
5980: (0) def choose(indices, choices, out=None, mode='raise'):
5981: (4) """
5982: (4)     Use an index array to construct a new array from a list of choices.
5983: (4)     Given an array of integers and a list of n choice arrays, this method
5984: (4)     will create a new array that merges each of the choice arrays. Where a
5985: (4)     value in `index` is i, the new array will have the value that choices[i]
5986: (4)     contains in the same place.
5987: (4)     Parameters
5988: (4)     -----
5989: (4)     indices : ndarray of ints
5990: (8)         This array must contain integers in ``[0, n-1]``, where n is the
5991: (8)         number of choices.
5992: (4)     choices : sequence of arrays
5993: (8)         Choice arrays. The index array and all of the choices should be
5994: (8)         broadcastable to the same shape.
5995: (4)     out : array, optional
5996: (8)         If provided, the result will be inserted into this array. It should
5997: (8)         be of the appropriate shape and `dtype`.
5998: (4)     mode : {'raise', 'wrap', 'clip'}, optional
5999: (8)         Specifies how out-of-bounds indices will behave.
6000: (8)         * 'raise' : raise an error
6001: (8)         * 'wrap' : wrap around
6002: (8)         * 'clip' : clip to the range
6003: (4)     Returns
6004: (4)     -----
6005: (4)     merged_array : array
6006: (4)     See Also
6007: (4)     -----
6008: (4)     choose : equivalent function
6009: (4)     Examples

```

```

6010: (4)
6011: (4)
6012: (4)
6013: (4)
6014: (4)
6015: (17)
6016: (11)
6017: (4)
6018: (4)
6019: (8)
6020: (8)
6021: (12)
6022: (8)
6023: (4)
6024: (8)
6025: (8)
6026: (12)
6027: (8)
6028: (4)
6029: (4)
6030: (4)
6031: (4)
6032: (4)
6033: (27)
6034: (4)
6035: (4)
6036: (8)
6037: (12)
6038: (8)
6039: (4)
6040: (4)
6041: (0)
6042: (4)
6043: (4)
6044: (4)
6045: (4)
6046: (4)
6047: (4)
6048: (4)
6049: (4)
6050: (4)
6051: (4)
6052: (8)
6053: (4)
6054: (8)
6055: (8)
6056: (4)
6057: (4)
6058: (4)
6059: (4)
6060: (4)
6061: (4)
6062: (4)
6063: (4)
6064: (4)
6065: (4)
6066: (4)
6067: (4)
6068: (17)
6069: (8)
6070: (4)
6071: (4)
6072: (17)
6073: (8)
6074: (4)
6075: (4)
6076: (17)
6077: (8)
6078: (4)

       -----
>>> choice = np.array([[1,1,1], [2,2,2], [3,3,3]])
>>> a = np.array([2, 1, 0])
>>> np.ma.choose(a, choice)
masked_array(data=[3, 2, 1],
             mask=False,
             fill_value=999999)
"""

def fmask(x):
    "Returns the filled array, or True if masked."
    if x is masked:
        return True
    return filled(x)

def nmask(x):
    "Returns the mask, True if ``masked``, False if ``nomask``."
    if x is masked:
        return True
    return getmask(x)

c = filled(indices, 0)
masks = [nmask(x) for x in choices]
data = [fmask(x) for x in choices]
outputmask = np.choose(c, masks, mode=mode)
outputmask = make_mask(mask_or(outputmask, getmask(indices)),
                       copy=False, shrink=True)
d = np.choose(c, data, mode=mode, out=out).view(MaskedArray)
if out is not None:
    if isinstance(out, MaskedArray):
        out._setmask_(outputmask)
    return out
d._setmask_(outputmask)
return d

def round_(a, decimals=0, out=None):
"""
Return a copy of a, rounded to 'decimals' places.
When 'decimals' is negative, it specifies the number of positions
to the left of the decimal point. The real and imaginary parts of
complex numbers are rounded separately. Nothing is done if the
array is not of float type and 'decimals' is greater than or equal
to 0.

Parameters
-----
decimals : int
    Number of decimals to round to. May be negative.
out : array_like
    Existing array to use for output.
    If not given, returns a default copy of a.

Notes
-----
If out is given and does not have a mask attribute, the mask of a
is lost!

Examples
-----
>>> import numpy.ma as ma
>>> x = [11.2, -3.973, 0.801, -1.41]
>>> mask = [0, 0, 0, 1]
>>> masked_x = ma.masked_array(x, mask)
>>> masked_x
masked_array(data=[11.2, -3.973, 0.801, --],
              mask=[False, False, False, True],
              fill_value=1e+20)
>>> ma.round_(masked_x)
masked_array(data=[11.0, -4.0, 1.0, --],
              mask=[False, False, False, True],
              fill_value=1e+20)
>>> ma.round(masked_x, decimals=1)
masked_array(data=[11.2, -4.0, 0.8, --],
              mask=[False, False, False, True],
              fill_value=1e+20)
>>> ma.round_(masked_x, decimals=-1)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

6079: (4) masked_array(data=[10.0, -0.0, 0.0, --],
6080: (17)         mask=[False, False, False, True],
6081: (8)         fill_value=1e+20)
6082: (4) """
6083: (4)     if out is None:
6084: (8)         return np.round_(a, decimals, out)
6085: (4)     else:
6086: (8)         np.round_(getdata(a), decimals, out)
6087: (8)         if hasattr(out, '_mask'):
6088: (12)             out._mask = getmask(a)
6089: (8)         return out
6090: (0) round = round_
6091: (0) def _mask_propagate(a, axis):
6092: (4) """
6093: (4)     Mask whole 1-d vectors of an array that contain masked values.
6094: (4) """
6095: (4)     a = array(a, subok=False)
6096: (4)     m = getmask(a)
6097: (4)     if m is nomask or not m.any() or axis is None:
6098: (8)         return a
6099: (4)     a._mask = a._mask.copy()
6100: (4)     axes = normalize_axis_tuple(axis, a.ndim)
6101: (4)     for ax in axes:
6102: (8)         a._mask |= m.any(axis=ax, keepdims=True)
6103: (4)     return a
6104: (0) def dot(a, b, strict=False, out=None):
6105: (4) """
6106: (4)     Return the dot product of two arrays.
6107: (4)     This function is the equivalent of `numpy.dot` that takes masked values
6108: (4)     into account. Note that `strict` and `out` are in different position
6109: (4)     than in the method version. In order to maintain compatibility with the
6110: (4)     corresponding method, it is recommended that the optional arguments be
6111: (4)     treated as keyword only. At some point that may be mandatory.
6112: (4)     Parameters
6113: (4)     -----
6114: (4)     a, b : masked_array_like
6115: (8)         Inputs arrays.
6116: (4)     strict : bool, optional
6117: (8)         Whether masked data are propagated (True) or set to 0 (False) for
6118: (8)         the computation. Default is False. Propagating the mask means that
6119: (8)         if a masked value appears in a row or column, the whole row or
6120: (8)         column is considered masked.
6121: (4)     out : masked_array, optional
6122: (8)         Output argument. This must have the exact kind that would be returned
6123: (8)         if it was not used. In particular, it must have the right type, must
be
6124: (8)         C-contiguous, and its dtype must be the dtype that would be returned
6125: (8)         for `dot(a,b)`. This is a performance feature. Therefore, if these
6126: (8)         conditions are not met, an exception is raised, instead of attempting
6127: (8)         to be flexible.
6128: (8)         .. versionadded:: 1.10.2
6129: (4)     See Also
6130: (4)     -----
6131: (4)     numpy.dot : Equivalent function for ndarrays.
6132: (4)     Examples
6133: (4)     -----
6134: (4)     >>> a = np.ma.array([[1, 2, 3], [4, 5, 6]], mask=[[1, 0, 0], [0, 0, 0]])
6135: (4)     >>> b = np.ma.array([[1, 2], [3, 4], [5, 6]], mask=[[1, 0], [0, 0], [0,
0]])
6136: (4)     >>> np.ma.dot(a, b)
6137: (4)     masked_array(
6138: (6)         data=[[21, 26],
6139: (12)             [45, 64]],
6140: (6)         mask=[[False, False],
6141: (12)             [False, False]],
6142: (6)             fill_value=999999)
6143: (4)     >>> np.ma.dot(a, b, strict=True)
6144: (4)     masked_array(
6145: (6)         data=[[--, --],

```

```

6146: (12)           [--, 64]],  

6147: (6)            mask=[[ True,  True],  

6148: (12)           [ True, False]],  

6149: (6)            fill_value=999999)  

6150: (4)           """  

6151: (4)           if strict is True:  

6152: (8)             if np.ndim(a) == 0 or np.ndim(b) == 0:  

6153: (12)               pass  

6154: (8)             elif b.ndim == 1:  

6155: (12)               a = _mask_propagate(a, a.ndim - 1)  

6156: (12)               b = _mask_propagate(b, b.ndim - 1)  

6157: (8)             else:  

6158: (12)               a = _mask_propagate(a, a.ndim - 1)  

6159: (12)               b = _mask_propagate(b, b.ndim - 2)  

6160: (4)             am = ~getmaskarray(a)  

6161: (4)             bm = ~getmaskarray(b)  

6162: (4)           if out is None:  

6163: (8)             d = np.dot(filled(a, 0), filled(b, 0))  

6164: (8)             m = ~np.dot(am, bm)  

6165: (8)             if np.ndim(d) == 0:  

6166: (12)               d = np.asarray(d)  

6167: (8)             r = d.view(get_masked_subclass(a, b))  

6168: (8)             r._setmask_(m)  

6169: (8)             return r  

6170: (4)           else:  

6171: (8)             d = np.dot(filled(a, 0), filled(b, 0), out._data)  

6172: (8)             if out.mask.shape != d.shape:  

6173: (12)               out._mask = np.empty(d.shape, MaskType)  

6174: (8)             np.dot(am, bm, out._mask)  

6175: (8)             np.logical_not(out._mask, out._mask)  

6176: (8)             return out  

6177: (0)           def inner(a, b):  

6178: (4)           """  

6179: (4)             Returns the inner product of a and b for arrays of floating point types.  

6180: (4)             Like the generic NumPy equivalent the product sum is over the last  

dimension  

6181: (4)             of a and b. The first argument is not conjugated.  

6182: (4)           """  

6183: (4)             fa = filled(a, 0)  

6184: (4)             fb = filled(b, 0)  

6185: (4)             if fa.ndim == 0:  

6186: (8)               fa.shape = (1,)  

6187: (4)             if fb.ndim == 0:  

6188: (8)               fb.shape = (1,)  

6189: (4)             return np.inner(fa, fb).view(MaskedArray)  

6190: (0)           inner.__doc__ = doc_note(np.inner.__doc__,  

6191: (25)                         "Masked values are replaced by 0.")  

6192: (0)           innerproduct = inner  

6193: (0)           def outer(a, b):  

6194: (4)             "maskedarray version of the numpy function."  

6195: (4)             fa = filled(a, 0).ravel()  

6196: (4)             fb = filled(b, 0).ravel()  

6197: (4)             d = np.outer(fa, fb)  

6198: (4)             ma = getmask(a)  

6199: (4)             mb = getmask(b)  

6200: (4)             if ma is nomask and mb is nomask:  

6201: (8)               return masked_array(d)  

6202: (4)             ma = getmaskarray(a)  

6203: (4)             mb = getmaskarray(b)  

6204: (4)             m = make_mask(1 - np.outer(1 - ma, 1 - mb), copy=False)  

6205: (4)             return masked_array(d, mask=m)  

6206: (0)           outer.__doc__ = doc_note(np.outer.__doc__,  

6207: (25)                         "Masked values are replaced by 0.")  

6208: (0)           outerproduct = outer  

6209: (0)           def __convolve_or_correlate(f, a, v, mode, propagate_mask):  

6210: (4)           """  

6211: (4)             Helper function for ma.correlate and ma.convolve  

6212: (4)           """  

6213: (4)             if propagate_mask:

```

```

6214: (8)             mask = (
6215: (12)             f(getmaskarray(a), np.ones(np.shape(v), dtype=bool), mode=mode)
6216: (10)             | f(np.ones(np.shape(a), dtype=bool), getmaskarray(v), mode=mode)
6217: (8)             )
6218: (8)             data = f(getdata(a), getdata(v), mode=mode)
6219: (4)         else:
6220: (8)             mask = ~f(~getmaskarray(a), ~getmaskarray(v))
6221: (8)             data = f(filled(a, 0), filled(v, 0), mode=mode)
6222: (4)         return masked_array(data, mask=mask)
6223: (0)     def correlate(a, v, mode='valid', propagate_mask=True):
6224: (4)         """
6225: (4)             Cross-correlation of two 1-dimensional sequences.
6226: (4)         Parameters
6227: (4)             -----
6228: (4)             a, v : array_like
6229: (8)                 Input sequences.
6230: (4)             mode : {'valid', 'same', 'full'}, optional
6231: (8)                 Refer to the `np.convolve` docstring. Note that the default
6232: (8)                 is 'valid', unlike `convolve`, which uses 'full'.
6233: (4)             propagate_mask : bool
6234: (8)                 If True, then a result element is masked if any masked element
contributes towards it.
6235: (8)                 If False, then a result element is only masked if no non-masked
element
6236: (8)                     contribute towards it
6237: (4)         Returns
6238: (4)             -----
6239: (4)             out : MaskedArray
6240: (8)                 Discrete cross-correlation of `a` and `v`.
6241: (4)         See Also
6242: (4)             -----
6243: (4)             numpy.correlate : Equivalent function in the top-level NumPy module.
6244: (4)             """
6245: (4)         return _convolve_or_correlate(np.correlate, a, v, mode, propagate_mask)
6246: (0)     def convolve(a, v, mode='full', propagate_mask=True):
6247: (4)         """
6248: (4)             Returns the discrete, linear convolution of two one-dimensional sequences.
6249: (4)         Parameters
6250: (4)             -----
6251: (4)             a, v : array_like
6252: (8)                 Input sequences.
6253: (4)             mode : {'valid', 'same', 'full'}, optional
6254: (8)                 Refer to the `np.convolve` docstring.
6255: (4)             propagate_mask : bool
6256: (8)                 If True, then if any masked element is included in the sum for a
result
6257: (8)                     element, then the result is masked.
6258: (8)                     If False, then the result element is only masked if no non-masked
cells
6259: (8)                         contribute towards it
6260: (4)         Returns
6261: (4)             -----
6262: (4)             out : MaskedArray
6263: (8)                 Discrete, linear convolution of `a` and `v`.
6264: (4)         See Also
6265: (4)             -----
6266: (4)             numpy.convolve : Equivalent function in the top-level NumPy module.
6267: (4)             """
6268: (4)         return _convolve_or_correlate(np.convolve, a, v, mode, propagate_mask)
6269: (0)     def allequal(a, b, fill_value=True):
6270: (4)         """
6271: (4)             Return True if all entries of a and b are equal, using
fill_value as a truth value where either or both are masked.
6272: (4)         Parameters
6273: (4)             -----
6274: (4)             a, b : array_like
6275: (8)                 Input arrays to compare.
6276: (8)             fill_value : bool, optional
6277: (4)                 Whether masked values in a or b are considered equal (True) or not
6278: (8)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

6279: (8)                               (False).
6280: (4)                               Returns
6281: (4)                               -----
6282: (4)                               y : bool
6283: (8)                               Returns True if the two arrays are equal within the given
6284: (8)                               tolerance, False otherwise. If either array contains NaN,
6285: (8)                               then False is returned.
6286: (4)                               See Also
6287: (4)                               -----
6288: (4)                               all, any
6289: (4)                               numpy.ma.allclose
6290: (4)                               Examples
6291: (4)                               -----
6292: (4)                               >>> a = np.ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
6293: (4)                               >>> a
6294: (4)                               masked_array(data=[1000000000.0, 1e-07, --],
6295: (17)                               mask=[False, False, True],
6296: (11)                               fill_value=1e+20)
6297: (4)                               >>> b = np.array([1e10, 1e-7, -42.0])
6298: (4)                               >>> b
6299: (4)                               array([ 1.00000000e+10,  1.00000000e-07, -4.20000000e+01])
6300: (4)                               >>> np.ma.allequal(a, b, fill_value=False)
6301: (4)                               False
6302: (4)                               >>> np.ma.allequal(a, b)
6303: (4)                               True
6304: (4)                               """
6305: (4)                               m = mask_or(getmask(a), getmask(b))
6306: (4)                               if m is nomask:
6307: (8)                                   x = getdata(a)
6308: (8)                                   y = getdata(b)
6309: (8)                                   d = umath.equal(x, y)
6310: (8)                                   return d.all()
6311: (4)                               elif fill_value:
6312: (8)                                   x = getdata(a)
6313: (8)                                   y = getdata(b)
6314: (8)                                   d = umath.equal(x, y)
6315: (8)                                   dm = array(d, mask=m, copy=False)
6316: (8)                                   return dm.filled(True).all(None)
6317: (4)                               else:
6318: (8)                                   return False
6319: (0)                               def allclose(a, b, masked_equal=True, rtol=1e-5, atol=1e-8):
6320: (4)                               """
6321: (4)                               Returns True if two arrays are element-wise equal within a tolerance.
6322: (4)                               This function is equivalent to `allclose` except that masked values
6323: (4)                               are treated as equal (default) or unequal, depending on the `masked_equal`
6324: (4)                               argument.
6325: (4)                               Parameters
6326: (4)                               -----
6327: (4)                               a, b : array_like
6328: (8)                               Input arrays to compare.
6329: (4)                               masked_equal : bool, optional
6330: (8)                               Whether masked values in `a` and `b` are considered equal (True) or
not
6331: (8)                               (False). They are considered equal by default.
6332: (4)                               rtol : float, optional
6333: (8)                               Relative tolerance. The relative difference is equal to ``rtol * b``.
6334: (8)                               Default is 1e-5.
6335: (4)                               atol : float, optional
6336: (8)                               Absolute tolerance. The absolute difference is equal to `atol`.
6337: (8)                               Default is 1e-8.
6338: (4)                               Returns
6339: (4)                               -----
6340: (4)                               y : bool
6341: (8)                               Returns True if the two arrays are equal within the given
6342: (8)                               tolerance, False otherwise. If either array contains NaN, then
6343: (8)                               False is returned.
6344: (4)                               See Also
6345: (4)                               -----
6346: (4)                               all, any

```

```

6347: (4) numpy.allclose : the non-masked `allclose` .
6348: (4) Notes -----
6349: (4) If the following equation is element-wise True, then `allclose` returns
6350: (4) True::: absolute(`a` - `b`) <= (`atol` + `rtol` * absolute(`b`))
6351: (4) Return True if all elements of `a` and `b` are equal subject to
6352: (6) given tolerances.
6353: (4)
6354: (4)
6355: (4) Examples -----
6356: (4)
6357: (4) >>> a = np.ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
6358: (4)
6359: (4) >>> a
6360: (17) masked_array(data=[10000000000.0, 1e-07, --],
6361: (11) mask=[False, False, True],
6362: (4) fill_value=1e+20)
6363: (4) >>> b = np.ma.array([1e10, 1e-8, -42.0], mask=[0, 0, 1])
6364: (4) >>> np.ma.allclose(a, b)
6365: (4) False
6366: (4) >>> a = np.ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
6367: (4) >>> b = np.ma.array([1.00001e10, 1e-9, -42.0], mask=[0, 0, 1])
6368: (4) >>> np.ma.allclose(a, b)
6369: (4) True
6370: (4) >>> np.ma.allclose(a, b, masked_equal=False)
6371: (4) False
6372: (4) Masked values are not compared directly.
6373: (4) >>> a = np.ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
6374: (4) >>> b = np.ma.array([1.00001e10, 1e-9, 42.0], mask=[0, 0, 1])
6375: (4) >>> np.ma.allclose(a, b)
6376: (4) True
6377: (4) >>> np.ma.allclose(a, b, masked_equal=False)
6378: (4) False
6379: (4) """
6380: (4) x = masked_array(a, copy=False)
6381: (4) y = masked_array(b, copy=False)
6382: (8) if y.dtype.kind != "m":
6383: (8)     dtype = np.result_type(y, 1.)
6384: (12)     if y.dtype != dtype:
6385: (4)         y = masked_array(y, dtype=dtype, copy=False)
6386: (4) m = mask_or(getmask(x), getmask(y))
6387: (4) xinf = np.isinf(masked_array(x, copy=False, mask=m)).filled(False)
6388: (8) if not np.all(xinf == filled(np.isinf(y), False)):
6389: (4)     return False
6390: (8) if not np.any(xinf):
6391: (19)     d = filled(less_equal(absOLUTE(x - y), atol + rtol * absolute(y)),
6392: (8)                         masked_equal)
6393: (4)     return np.all(d)
6394: (8) if not np.all(filled(x[xinf] == y[xinf], masked_equal)):
6395: (4)     return False
6396: (4) x = x[~xinf]
6397: (4) y = y[~xinf]
6398: (15) d = filled(less_equal(absolute(x - y), atol + rtol * absolute(y)),
6399: (4)                         masked_equal)
6400: (0) return np.all(d)
6401: (4) def asarray(a, dtype=None, order=None):
6402: (4) """
6403: (4) Convert the input to a masked array of the given data-type.
6404: (4) No copy is performed if the input is already an `ndarray`. If `a` is
6405: (4) a subclass of `MaskedArray`, a base class `MaskedArray` is returned.
6406: (4) Parameters -----
6407: (4) a : array_like
6408: (8) Input data, in any form that can be converted to a masked array. This
6409: (8) includes lists, lists of tuples, tuples, tuples of tuples, tuples
6410: (8) of lists, ndarrays and masked arrays.
6411: (4) dtype : dtype, optional
6412: (8) By default, the data-type is inferred from the input data.
6413: (4) order : {'C', 'F'}, optional
6414: (8) Whether to use row-major ('C') or column-major ('FORTRAN') memory
6415: (8) representation. Default is 'C'.

```

```

6416: (4)          Returns
6417: (4)          -----
6418: (4)          out : MaskedArray
6419: (8)          Masked array interpretation of `a`.
6420: (4)          See Also
6421: (4)          -----
6422: (4)          asanyarray : Similar to `asarray`, but conserves subclasses.
6423: (4)          Examples
6424: (4)          -----
6425: (4)          >>> x = np.arange(10.).reshape(2, 5)
6426: (4)          >>> x
6427: (4)          array([[0., 1., 2., 3., 4.],
6428: (11)           [5., 6., 7., 8., 9.]])
6429: (4)          >>> np.ma.asarray(x)
6430: (4)          masked_array(
6431: (6)          data=[[0., 1., 2., 3., 4.],
6432: (12)           [5., 6., 7., 8., 9.]],
6433: (6)           mask=False,
6434: (6)           fill_value=1e+20)
6435: (4)          >>> type(np.ma.asarray(x))
6436: (4)          <class 'numpy.ma.core.MaskedArray'>
6437: (4)          """
6438: (4)          order = order or 'C'
6439: (4)          return masked_array(a, dtype=dtype, copy=False, keep_mask=True,
6440: (24)             subok=False, order=order)
6441: (0)          def asanyarray(a, dtype=None):
6442: (4)          """
6443: (4)          Convert the input to a masked array, conserving subclasses.
6444: (4)          If `a` is a subclass of `MaskedArray`, its class is conserved.
6445: (4)          No copy is performed if the input is already an `ndarray`.
6446: (4)          Parameters
6447: (4)          -----
6448: (4)          a : array_like
6449: (8)          Input data, in any form that can be converted to an array.
6450: (4)          dtype : dtype, optional
6451: (8)          By default, the data-type is inferred from the input data.
6452: (4)          order : {'C', 'F'}, optional
6453: (8)          Whether to use row-major ('C') or column-major ('FORTRAN') memory
6454: (8)          representation. Default is 'C'.
6455: (4)          Returns
6456: (4)          -----
6457: (4)          out : MaskedArray
6458: (8)          MaskedArray interpretation of `a`.
6459: (4)          See Also
6460: (4)          -----
6461: (4)          asarray : Similar to `asanyarray`, but does not conserve subclass.
6462: (4)          Examples
6463: (4)          -----
6464: (4)          >>> x = np.arange(10.).reshape(2, 5)
6465: (4)          >>> x
6466: (4)          array([[0., 1., 2., 3., 4.],
6467: (11)           [5., 6., 7., 8., 9.]])
6468: (4)          >>> np.ma.asanyarray(x)
6469: (4)          masked_array(
6470: (6)          data=[[0., 1., 2., 3., 4.],
6471: (12)           [5., 6., 7., 8., 9.]],
6472: (6)           mask=False,
6473: (6)           fill_value=1e+20)
6474: (4)          >>> type(np.ma.asanyarray(x))
6475: (4)          <class 'numpy.ma.core.MaskedArray'>
6476: (4)          """
6477: (4)          if isinstance(a, MaskedArray) and (dtype is None or dtype == a.dtype):
6478: (8)              return a
6479: (4)          return masked_array(a, dtype=dtype, copy=False, keep_mask=True,
6480: (0)             subok=True)
6481: (4)          def fromfile(file, dtype=float, count=-1, sep=''):
6482: (8)              raise NotImplementedError(
6483: (0)                  "fromfile() not yet implemented for a MaskedArray.")
6483: (0)          def fromflex(fxarray):

```

```

6484: (4)
6485: (4)
6486: (4)
6487: (4)
6488: (4)
6489: (4)
6490: (4)
6491: (8)
6492: (8)
6493: (4)
6494: (4)
6495: (4)
6496: (8)
6497: (4)
6498: (4)
6499: (4)
6500: (4)
6501: (4)
6502: (4)
6503: (4)
6504: (4)
6505: (4)
6506: (11)
6507: (11)
6508: (10)
6509: (4)
6510: (4)
6511: (4)
6512: (6)
6513: (12)
6514: (12)
6515: (6)
6516: (12)
6517: (12)
6518: (6)
6519: (4)
6520: (4)
6521: (4)
6522: (4)
6523: (4)
6524: (11)
6525: (10)
6526: (4)
6527: (4)
6528: (4)
6529: (6)
6530: (12)
6531: (6)
6532: (12)
6533: (6)
6534: (6)
6535: (4)
6536: (4)
6537: (0)
6538: (4)
6539: (4)
6540: (4)
6541: (4)
6542: (8)
6543: (12)
6544: (4)
6545: (4)
6546: (4)
6547: (8)
6548: (8)
6549: (8)
6550: (4)
6551: (8)
6552: (8)

        """
        Build a masked array from a suitable flexible-type array.
        The input array has to have a data-type with ``_data`` and ``_mask``
        fields. This type of array is output by `MaskedArray.toflex`.
        Parameters
        -----
        fxarray : ndarray
            The structured input array, containing ``_data`` and ``_mask`` fields.
            If present, other fields are discarded.
        Returns
        -----
        result : MaskedArray
            The constructed masked array.
        See Also
        -----
        MaskedArray.toflex : Build a flexible-type array from a masked array.
        Examples
        -----
        >>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[0] + [1, 0] * 4)
        >>> rec = x.toflex()
        >>> rec
        array([[(0, False), (1, True), (2, False)],
               [(3, True), (4, False), (5, True)],
               [(6, False), (7, True), (8, False)]],
              dtype=[('_data', '<i8'), ('_mask', '?')])
        >>> x2 = np.ma.fromflex(rec)
        >>> x2
        masked_array(
            data=[[0, --, 2],
                  [--, 4, --],
                  [6, --, 8]],
            mask=[[False, True, False],
                  [True, False, True],
                  [False, True, False]],
            fill_value=999999)
        Extra fields can be present in the structured array but are discarded:
        >>> dt = [('data', '<i4'), ('_mask', '|b1'), ('field3', '<f4')]
        >>> rec2 = np.zeros((2, 2), dtype=dt)
        >>> rec2
        array([[ (0, False, 0.), (0, False, 0.)],
               [(0, False, 0.), (0, False, 0.)]],
              dtype=[('_data', '<i4'), ('_mask', '?'), ('field3', '<f4')])
        >>> y = np.ma.fromflex(rec2)
        >>> y
        masked_array(
            data=[[0, 0],
                  [0, 0]],
            mask=[[False, False],
                  [False, False]],
            fill_value=999999,
            dtype=int32)
        """
        return masked_array(fxarray['_data'], mask=fxarray['_mask'])

class _convert2ma:
    """
        Convert functions from numpy to numpy.ma.
        Parameters
        -----
        _methodname : string
            Name of the method to transform.
    """
    __doc__ = None
    def __init__(self, funcname, np_ret, np_ma_ret, params=None):
        self._func = getattr(np, funcname)
        self.__doc__ = self.getdoc(np_ret, np_ma_ret)
        self._extras = params or {}
    def getdoc(self, np_ret, np_ma_ret):
        "Return the doc of the function (from the doc of the method)."
        doc = getattr(self._func, '__doc__', None)

```

```

6553: (8)             sig = get_object_signature(self._func)
6554: (8)             if doc:
6555: (12)             doc = self._replace_return_type(doc, np_ret, np_ma_ret)
6556: (12)             if sig:
6557: (16)                 sig = "%s%s\n" % (self._func.__name__, sig)
6558: (12)                 doc = sig + doc
6559: (8)             return doc
6560: (4)         def _replace_return_type(self, doc, np_ret, np_ma_ret):
6561: (8)             """
6562: (8)                 Replace documentation of ``np`` function's return type.
6563: (8)                 Replaces it with the proper type for the ``np.ma`` function.
6564: (8)             Parameters
6565: (8)             -----
6566: (8)             doc : str
6567: (12)                 The documentation of the ``np`` method.
6568: (8)             np_ret : str
6569: (12)                 The return type string of the ``np`` method that we want to
6570: (12)                 replace. (e.g. "out : ndarray")
6571: (8)             np_ma_ret : str
6572: (12)                 The return type string of the ``np.ma`` method.
6573: (12)                 (e.g. "out : MaskedArray")
6574: (8)             """
6575: (8)             if np_ret not in doc:
6576: (12)                 raise RuntimeError(
6577: (16)                     f"Failed to replace `'{np_ret}` with `'{np_ma_ret}`. "
6578: (16)                     f"The documentation string for return type, {np_ret}, is not "
6579: (16)                     f"found in the docstring for `np.{self._func.__name__}`. "
6580: (16)                     f"Fix the docstring for `np.{self._func.__name__}` or "
6581: (16)                     f"update the expected string for return type."
6582: (12)                 )
6583: (8)             return doc.replace(np_ret, np_ma_ret)
6584: (4)         def __call__(self, *args, **params):
6585: (8)             _extras = self._extras
6586: (8)             common_params = set(params).intersection(_extras)
6587: (8)             for p in common_params:
6588: (12)                 _extras[p] = params.pop(p)
6589: (8)             result = self._func.__call__(*args, **params).view(MaskedArray)
6590: (8)             if "fill_value" in common_params:
6591: (12)                 result.fill_value = _extras.get("fill_value", None)
6592: (8)             if "hardmask" in common_params:
6593: (12)                 result._hardmask = bool(_extras.get("hard_mask", False))
6594: (8)             return result
6595: (0)         arange = _convert2ma(
6596: (4)             'arange',
6597: (4)             params=dict(fill_value=None, hardmask=False),
6598: (4)             np_ret='arange : ndarray',
6599: (4)             np_ma_ret='arange : MaskedArray',
6600: (0)
6601: (0)
6602: (4)
6603: (4)
6604: (4)
6605: (4)
6606: (0)
6607: (0)
6608: (4)
6609: (4)
6610: (4)
6611: (4)
6612: (0)
6613: (0)
6614: (4)
6615: (4)
6616: (4)
6617: (0)
6618: (0)
6619: (4)
6620: (4)
6621: (4)
)
empty = _convert2ma(
    'empty',
    params=dict(fill_value=None, hardmask=False),
    np_ret='out : ndarray',
    np_ma_ret='out : MaskedArray',
)
empty_like = _convert2ma(
    'empty_like',
    np_ret='out : ndarray',
    np_ma_ret='out : MaskedArray',
)
frombuffer = _convert2ma(
    'frombuffer',
    np_ret='out : ndarray',
    np_ma_ret='out: MaskedArray',
)

```

```

6622: (0)
6623: (0)
6624: (3)
6625: (3)
6626: (3)
6627: (0)
6628: (0)
6629: (4)
6630: (4)
6631: (4)
6632: (4)
6633: (0)
6634: (0)
6635: (4)
6636: (4)
6637: (4)
6638: (4)
6639: (0)
6640: (0)
6641: (4)
6642: (4)
6643: (4)
6644: (4)
6645: (0)
6646: (0)
6647: (4)
6648: (4)
6649: (4)
6650: (0)
6651: (0)
6652: (4)
6653: (4)
6654: (4)
6655: (4)
6656: (0)
6657: (0)
6658: (4)
6659: (4)
6660: (4)
6661: (4)
6662: (0)
6663: (0)
6664: (4)
6665: (4)
6666: (4)
6667: (0)
6668: (0)
def append(a, b, axis=None):
    """Append values to the end of an array.
    .. versionadded:: 1.9.0
    Parameters
    -----
    a : array_like
        Values are appended to a copy of this array.
    b : array_like
        These values are appended to a copy of `a`. It must be of the
        correct shape (the same shape as `a`, excluding `axis`). If `axis`
        is not specified, `b` can be any shape and will be flattened
        before use.
    axis : int, optional
        The axis along which `v` are appended. If `axis` is not given,
        both `a` and `b` are flattened before use.
    Returns
    -----
    append : MaskedArray
        A copy of `a` with `b` appended to `axis`. Note that `append`
        does not occur in-place: a new array is allocated and filled. If
        `axis` is None, the result is a flattened array.
    See Also
    -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

6691: (4)         numpy.append : Equivalent function in the top-level NumPy module.
6692: (4)         Examples
6693: (4)
6694: (4)
6695: (4)
6696: (4)
6697: (4)
6698: (4)
6699: (17)
6700: (23)
6701: (11)
6702: (4)
6703: (4)         return concatenate([a, b], axis)

-----

```

## File 262 - extras.py:

```

1: (0)         """
2: (0)         Masked arrays add-ons.
3: (0)         A collection of utilities for `numpy.ma`.
4: (0)         :author: Pierre Gerard-Marchant
5: (0)         :contact: pierregm_at_uga_dot_edu
6: (0)         :version: $Id: extras.py 3473 2007-10-29 15:18:13Z jarrod.millman $
7: (0)         """
8: (0)         __all__ = [
9: (4)             'apply_along_axis', 'apply_over_axes', 'atleast_1d', 'atleast_2d',
10: (4)            'atleast_3d', 'average', 'clump_masked', 'clump_unmasked', 'column_stack',
11: (4)            'compress_cols', 'compress_nd', 'compress_rowcols', 'compress_rows',
12: (4)            'count_masked', 'corrcoef', 'cov', 'diagflat', 'dot', 'dstack', 'ediff1d',
13: (4)            'flatnotmasked_contiguous', 'flatnotmasked_edges', 'hsplit', 'hstack',
14: (4)            'isin', 'in1d', 'intersect1d', 'mask_cols', 'mask_rowcols', 'mask_rows',
15: (4)            'masked_all', 'masked_all_like', 'median', 'mr_', 'ndenumerate',
16: (4)            'notmasked_contiguous', 'notmasked_edges', 'polyfit', 'row_stack',
17: (4)            'setdiff1d', 'setxor1d', 'stack', 'unique', 'union1d', 'vander', 'vstack',
18: (4)        ]
19: (0)         import itertools
20: (0)         import warnings
21: (0)         from . import core as ma
22: (0)         from .core import (
23: (4)             MaskedArray, MAError, add, array, asarray, concatenate, filled, count,
24: (4)             getmask, getmaskarray, make_mask_descr, masked, masked_array, mask_or,
25: (4)             nomask, ones, sort, zeros, getdata, get_masked_subclass, dot
26: (4)         )
27: (0)         import numpy as np
28: (0)         from numpy import ndarray, array as nxarray
29: (0)         from numpy.core.multiarray import normalize_axis_index
30: (0)         from numpy.core.numeric import normalize_axis_tuple
31: (0)         from numpy.lib.function_base import _ureduce
32: (0)         from numpy.lib.index_tricks import AxisConcatenator
33: (0)         def issequence(seq):
34: (4)             """
35: (4)             Is seq a sequence (ndarray, list or tuple)?
36: (4)             """
37: (4)             return isinstance(seq, (ndarray, tuple, list))
38: (0)         def count_masked(arr, axis=None):
39: (4)             """
40: (4)                 Count the number of masked elements along the given axis.
41: (4)                 Parameters
42: (4)
43: (4)                 arr : array_like
44: (8)                     An array with (possibly) masked elements.
45: (4)                 axis : int, optional
46: (8)                     Axis along which to count. If None (default), a flattened
47: (8)                     version of the array is used.
48: (4)                 Returns
49: (4)
50: (4)                 count : int, ndarray
51: (8)                     The total number of masked elements (axis=None) or the number

```

```

52: (8)          of masked elements along each slice of the given axis.
53: (4)          See Also
54: (4)
55: (4)          MaskedArray.count : Count non-masked elements.
56: (4)          Examples
57: (4)
58: (4)          >>> import numpy.ma as ma
59: (4)          >>> a = np.arange(9).reshape((3,3))
60: (4)          >>> a = ma.array(a)
61: (4)          >>> a[1, 0] = ma.masked
62: (4)          >>> a[1, 2] = ma.masked
63: (4)          >>> a[2, 1] = ma.masked
64: (4)          >>> a
65: (4)          masked_array(
66: (6)              data=[[0, 1, 2],
67: (12)                  [--, 4, --],
68: (12)                  [6, --, 8]],
69: (6)              mask=[[False, False, False],
70: (12)                  [True, False, True],
71: (12)                  [False, True, False]],
72: (6)              fill_value=999999)
73: (4)          >>> ma.count_masked(a)
74: (4)          3
75: (4)          When the `axis` keyword is used an array is returned.
76: (4)          >>> ma.count_masked(a, axis=0)
77: (4)          array([1, 1, 1])
78: (4)          >>> ma.count_masked(a, axis=1)
79: (4)          array([0, 2, 1])
80: (4)          """
81: (4)          m = getmaskarray(arr)
82: (4)          return m.sum(axis)
83: (0)          def masked_all(shape, dtype=float):
84: (4)          """
85: (4)          Empty masked array with all elements masked.
86: (4)          Return an empty masked array of the given shape and dtype, where all the
87: (4)          data are masked.
88: (4)          Parameters
89: (4)
90: (4)          shape : int or tuple of ints
91: (8)              Shape of the required MaskedArray, e.g., ``(2, 3)`` or ``2``.
92: (4)          dtype : dtype, optional
93: (8)              Data type of the output.
94: (4)          Returns
95: (4)
96: (4)          a : MaskedArray
97: (8)              A masked array with all data masked.
98: (4)          See Also
99: (4)
100: (4)          masked_all_like : Empty masked array modelled on an existing array.
101: (4)          Examples
102: (4)
103: (4)          >>> import numpy.ma as ma
104: (4)          >>> ma.masked_all((3, 3))
105: (4)          masked_array(
106: (6)              data=[[--, --, --],
107: (12)                  [--, --, --],
108: (12)                  [--, --, --]],
109: (6)              mask=[[ True,  True,  True],
110: (12)                  [ True,  True,  True],
111: (12)                  [ True,  True,  True]],
112: (6)              fill_value=1e+20,
113: (6)              dtype=float64)
114: (4)          The `dtype` parameter defines the underlying data type.
115: (4)          >>> a = ma.masked_all((3, 3))
116: (4)          >>> a.dtype
117: (4)          dtype('float64')
118: (4)          >>> a = ma.masked_all((3, 3), dtype=np.int32)
119: (4)          >>> a.dtype
120: (4)          dtype('int32')

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

121: (4)
122: (4)
123: (21)
124: (4)
125: (0)
126: (4)
127: (4)
128: (4)
129: (4)
130: (4)
131: (4)
132: (4)
133: (8)
134: (4)
135: (4)
136: (4)
137: (8)
138: (4)
139: (4)
140: (4)
141: (8)
142: (4)
143: (4)
144: (4)
145: (4)
146: (4)
147: (4)
148: (4)
149: (4)
150: (4)
151: (11)
152: (4)
153: (4)
154: (6)
155: (12)
156: (6)
157: (12)
158: (6)
159: (6)
160: (4)
161: (4)
162: (4)
163: (4)
164: (4)
165: (4)
166: (4)
167: (4)
168: (4)
169: (0)
170: (4)
171: (4)
172: (4)
173: (4)
174: (4)
175: (4)
that
176: (4)
177: (4)
178: (4)
179: (4)
funcname : str
180: (8)
181: (8)
182: (4)
183: (4)
184: (8)
185: (8)
186: (4)
187: (8)
188: (8)

        """
        a = masked_array(np.empty(shape, dtype),
                        mask=np.ones(shape, make_mask_descr(dtype)))
        return a
    def masked_all_like(arr):
        """
        Empty masked array with the properties of an existing array.
        Return an empty masked array of the same shape and dtype as
        the array `arr`, where all the data are masked.
        Parameters
        -----
        arr : ndarray
            An array describing the shape and dtype of the required MaskedArray.
        Returns
        -----
        a : MaskedArray
            A masked array with all data masked.
        Raises
        -----
        AttributeError
            If `arr` doesn't have a shape attribute (i.e. not an ndarray)
        See Also
        -----
        masked_all : Empty masked array with all elements masked.
        Examples
        -----
        >>> import numpy.ma as ma
        >>> arr = np.zeros((2, 3), dtype=np.float32)
        >>> arr
        array([[0., 0., 0.],
               [0., 0., 0.]], dtype=float32)
        >>> ma.masked_all_like(arr)
        masked_array(
            data=[[--, --, --],
                  [--, --, --]],
            mask=[[ True,  True,  True],
                  [ True,  True,  True]],
            fill_value=1e+20,
            dtype=float32)
        The dtype of the masked array matches the dtype of `arr`.
        >>> arr.dtype
        dtype('float32')
        >>> ma.masked_all_like(arr).dtype
        dtype('float32')
        """
        a = np.empty_like(arr).view(MaskedArray)
        a._mask = np.ones(a.shape, dtype=make_mask_descr(a.dtype))
        return a
    class _fromnxfunction:
        """
        Defines a wrapper to adapt NumPy functions to masked arrays.
        An instance of `_fromnxfunction` can be called with the same parameters
        as the wrapped NumPy function. The docstring of `newfunc` is adapted from
        the wrapped function as well, see `getdoc`.
        This class should not be used directly. Instead, one of its extensions
        provides support for a specific type of input should be used.
        Parameters
        -----
        funcname : str
            The name of the function to be adapted. The function should be
            in the NumPy namespace (i.e. ``np.funcname``).
        """
        def __init__(self, funcname):
            self.__name__ = funcname
            self.__doc__ = self.getdoc()
        def getdoc(self):
            """
            Retrieve the docstring and signature from the function.
        
```

```

189: (8)             The ``__doc__`` attribute of the function is used as the docstring for
190: (8)             the new masked array version of the function. A note on application
191: (8)             of the function to the mask is appended.
192: (8)             Parameters
193: (8)             -----
194: (8)             None
195: (8)             """
196: (8)             npfunc = getattr(np, self.__name__, None)
197: (8)             doc = getattr(npfunc, '__doc__', None)
198: (8)             if doc:
199: (12)                 sig = self.__name__ + ma.get_object_signature(npfunc)
200: (12)                 doc = ma.doc_note(doc, "The function is applied to both the _data
"
201: (35)                     "and the _mask, if any.")
202: (12)                 return '\n\n'.join((sig, doc))
203: (8)             return
204: (4)             def __call__(self, *args, **params):
205: (8)                 pass
206: (0)             class _fromnxfunction_single(_fromnxfunction):
207: (4)                 """
208: (4)                 A version of `_fromnxfunction` that is called with a single array
209: (4)                 argument followed by auxiliary args that are passed verbatim for
210: (4)                 both the data and mask calls.
211: (4)                 """
212: (4)                 def __call__(self, x, *args, **params):
213: (8)                     func = getattr(np, self.__name__)
214: (8)                     if isinstance(x, ndarray):
215: (12)                         _d = func(x.__array__(), *args, **params)
216: (12)                         _m = func(getmaskarray(x), *args, **params)
217: (12)                         return masked_array(_d, mask=_m)
218: (8)                     else:
219: (12)                         _d = func(np.asarray(x), *args, **params)
220: (12)                         _m = func(getmaskarray(x), *args, **params)
221: (12)                         return masked_array(_d, mask=_m)
222: (0)             class _fromnxfunction_seq(_fromnxfunction):
223: (4)                 """
224: (4)                 A version of `_fromnxfunction` that is called with a single sequence
225: (4)                 of arrays followed by auxiliary args that are passed verbatim for
226: (4)                 both the data and mask calls.
227: (4)                 """
228: (4)                 def __call__(self, x, *args, **params):
229: (8)                     func = getattr(np, self.__name__)
230: (8)                     _d = func(tuple([np.asarray(a) for a in x]), *args, **params)
231: (8)                     _m = func(tuple([getmaskarray(a) for a in x]), *args, **params)
232: (8)                     return masked_array(_d, mask=_m)
233: (0)             class _fromnxfunction_args(_fromnxfunction):
234: (4)                 """
235: (4)                 A version of `_fromnxfunction` that is called with multiple array
236: (4)                 arguments. The first non-array-like input marks the beginning of the
237: (4)                 arguments that are passed verbatim for both the data and mask calls.
238: (4)                 Array arguments are processed independently and the results are
239: (4)                 returned in a list. If only one array is found, the return value is
240: (4)                 just the processed array instead of a list.
241: (4)                 """
242: (4)                 def __call__(self, *args, **params):
243: (8)                     func = getattr(np, self.__name__)
244: (8)                     arrays = []
245: (8)                     args = list(args)
246: (8)                     while len(args) > 0 and issequence(args[0]):
247: (12)                         arrays.append(args.pop(0))
248: (8)                     res = []
249: (8)                     for x in arrays:
250: (12)                         _d = func(np.asarray(x), *args, **params)
251: (12)                         _m = func(getmaskarray(x), *args, **params)
252: (12)                         res.append(masked_array(_d, mask=_m))
253: (8)                     if len(arrays) == 1:
254: (12)                         return res[0]
255: (8)                     return res
256: (0)             class _fromnxfunction_allargs(_fromnxfunction):

```

```

257: (4)
258: (4)      """
259: (4)      A version of `_fromnxfunction` that is called with multiple array
260: (4)      arguments. Similar to `_fromnxfunction_args` except that all args
261: (4)      are converted to arrays even if they are not so already. This makes
262: (4)      it possible to process scalars as 1-D arrays. Only keyword arguments
263: (4)      are passed through verbatim for the data and mask calls. Arrays
264: (4)      arguments are processed independently and the results are returned
265: (4)      in a list. If only one arg is present, the return value is just the
266: (4)      processed array instead of a list.
267: (4)      """
268: (8)      def __call__(self, *args, **params):
269: (8)          func = getattr(np, self.__name__)
270: (8)          res = []
271: (12)         for x in args:
272: (12)             _d = func(np.asarray(x), **params)
273: (12)             _m = func(getmaskarray(x), **params)
274: (8)             res.append(masked_array(_d, mask=_m))
275: (12)         if len(args) == 1:
276: (8)             return res[0]
277: (8)         return res
278: (0)      atleast_1d = _fromnxfunction_allargs('atleast_1d')
279: (0)      atleast_2d = _fromnxfunction_allargs('atleast_2d')
280: (0)      atleast_3d = _fromnxfunction_allargs('atleast_3d')
281: (0)      vstack = row_stack = _fromnxfunction_seq('vstack')
282: (0)      hstack = _fromnxfunction_seq('hstack')
283: (0)      column_stack = _fromnxfunction_seq('column_stack')
284: (0)      dstack = _fromnxfunction_seq('dstack')
285: (0)      stack = _fromnxfunction_seq('stack')
286: (0)      hsplit = _fromnxfunction_single('hsplit')
287: (0)      diagflat = _fromnxfunction_single('diagflat')
288: (0)      def flatten_inplace(seq):
289: (4)          """Flatten a sequence in place."""
290: (4)          k = 0
291: (8)          while (k != len(seq)):
292: (12)              while hasattr(seq[k], '__iter__'):
293: (8)                  seq[k:(k + 1)] = seq[k]
294: (4)                  k += 1
295: (8)          return seq
296: (4)      def apply_along_axis(func1d, axis, arr, *args, **kwargs):
297: (4)          """
298: (4)              (This docstring should be overwritten)
299: (4)          """
300: (4)          arr = array(arr, copy=False, subok=True)
301: (4)          nd = arr.ndim
302: (4)          axis = normalize_axis_index(axis, nd)
303: (4)          ind = [0] * (nd - 1)
304: (4)          i = np.zeros(nd, 'O')
305: (4)          indlist = list(range(nd))
306: (4)          indlist.remove(axis)
307: (4)          i[axis] = slice(None, None)
308: (4)          outshape = np.asarray(arr.shape).take(indlist)
309: (4)          i.put(indlist, ind)
310: (4)          res = func1d(arr[tuple(i.tolist())], *args, **kwargs)
311: (4)          asscalar = np.isscalar(res)
312: (4)          if not asscalar:
313: (8)              try:
314: (8)                  len(res)
315: (8)              except TypeError:
316: (12)                  asscalar = True
317: (4)          dtypes = []
318: (8)          if asscalar:
319: (8)              dtypes.append(np.asarray(res).dtype)
320: (8)              outarr = zeros(outshape, object)
321: (8)              outarr[tuple(ind)] = res
322: (8)              Ntot = np.prod(outshape)
323: (8)              k = 1
324: (12)              while k < Ntot:
325: (12)                  ind[-1] += 1

```

```

326: (12)           while (ind[n] >= outshape[n]) and (n > (1 - nd)):
327: (16)             ind[n - 1] += 1
328: (16)             ind[n] = 0
329: (16)             n -= 1
330: (12)             i.put(indlist, ind)
331: (12)             res = func1d(arr[tuple(i.tolist())], *args, **kwargs)
332: (12)             outarr[tuple(ind)] = res
333: (12)             dtypes.append(asarray(res).dtype)
334: (12)             k += 1
335: (4)         else:
336: (8)             res = array(res, copy=False, subok=True)
337: (8)             j = i.copy()
338: (8)             j[axis] = ([slice(None, None)] * res.ndim)
339: (8)             j.put(indlist, ind)
340: (8)             Ntot = np.prod(outshape)
341: (8)             holdshape = outshape
342: (8)             outshape = list(arr.shape)
343: (8)             outshape[axis] = res.shape
344: (8)             dtypes.append(asarray(res).dtype)
345: (8)             outshape = flatten_inplace(outshape)
346: (8)             outarr = zeros(outshape, object)
347: (8)             outarr[tuple(flatten_inplace(j.tolist()))] = res
348: (8)             k = 1
349: (8)             while k < Ntot:
350: (12)               ind[-1] += 1
351: (12)               n = -1
352: (12)               while (ind[n] >= holdshape[n]) and (n > (1 - nd)):
353: (16)                 ind[n - 1] += 1
354: (16)                 ind[n] = 0
355: (16)                 n -= 1
356: (12)                 i.put(indlist, ind)
357: (12)                 j.put(indlist, ind)
358: (12)                 res = func1d(arr[tuple(i.tolist())], *args, **kwargs)
359: (12)                 outarr[tuple(flatten_inplace(j.tolist()))] = res
360: (12)                 dtypes.append(asarray(res).dtype)
361: (12)                 k += 1
362: (4)             max_dtypes = np.dtype(np.asarray(dtypes).max())
363: (4)             if not hasattr(arr, '_mask'):
364: (8)               result = np.asarray(outarr, dtype=max_dtypes)
365: (4)             else:
366: (8)               result = asarray(outarr, dtype=max_dtypes)
367: (8)               result.fill_value = ma.default_fill_value(result)
368: (4)             return result
369: (0)             apply_along_axis.__doc__ = np.apply_along_axis.__doc__
370: (0)             def apply_over_axes(func, a, axes):
371: (4)               """
372: (4)               (This docstring will be overwritten)
373: (4)               """
374: (4)               val = asarray(a)
375: (4)               N = a.ndim
376: (4)               if array(axes).ndim == 0:
377: (8)                 axes = (axes,)
378: (4)               for axis in axes:
379: (8)                 if axis < 0:
380: (12)                   axis = N + axis
381: (8)                   args = (val, axis)
382: (8)                   res = func(*args)
383: (8)                   if res.ndim == val.ndim:
384: (12)                     val = res
385: (8)                   else:
386: (12)                     res = ma.expand_dims(res, axis)
387: (12)                     if res.ndim == val.ndim:
388: (16)                       val = res
389: (12)                     else:
390: (16)                       raise ValueError("function is not returning "
391: (24)                           "an array of the correct shape")
392: (4)               return val
393: (0)             if apply_over_axes.__doc__ is not None:
394: (4)               apply_over_axes.__doc__ = np.apply_over_axes.__doc__[
```

```

395: (8)          :np.apply_over_axes.__doc__.find('Notes')].rstrip() + \
396: (4)          """
397: (4)          Examples
398: (4)          -----
399: (4)          >>> a = np.ma.arange(24).reshape(2,3,4)
400: (4)          >>> a[:,0,1] = np.ma.masked
401: (4)          >>> a[:,1,:] = np.ma.masked
402: (4)          >>> a
403: (4)          masked_array(
404: (6)          data=[[0, --, 2, 3],
405: (13)          [--, --, --, --],
406: (13)          [8, 9, 10, 11]],
407: (12)          [[12, --, 14, 15],
408: (13)          [--, --, --, --],
409: (13)          [20, 21, 22, 23]]],
410: (6)          mask=[[False, True, False, False],
411: (13)          [ True, True, True, True],
412: (13)          [False, False, False, False]],
413: (12)          [[False, True, False, False],
414: (13)          [ True, True, True, True],
415: (13)          [False, False, False, False]],
416: (6)          fill_value=999999)
417: (4)          >>> np.ma.apply_over_axes(np.ma.sum, a, [0,2])
418: (4)          masked_array(
419: (6)          data=[[46],
420: (13)          [--],
421: (13)          [124]]],
422: (6)          mask=[[False],
423: (13)          [ True],
424: (13)          [False]]],
425: (6)          fill_value=999999)
426: (4)          Tuple axis arguments to ufuncs are equivalent:
427: (4)          >>> np.ma.sum(a, axis=(0,2)).reshape((1,-1,1))
428: (4)          masked_array(
429: (6)          data=[[46],
430: (13)          [--],
431: (13)          [124]]],
432: (6)          mask=[[False],
433: (13)          [ True],
434: (13)          [False]]],
435: (6)          fill_value=999999)
436: (4)
437: (0)          """
438: (12)          def average(a, axis=None, weights=None, returned=False, *,
439: (4)              keepdims=np._NoValue):
440: (4)              Return the weighted average of array over the given axis.
441: (4)              Parameters
442: (4)              -----
443: (4)              a : array_like
444: (8)                  Data to be averaged.
445: (8)                  Masked entries are not taken into account in the computation.
446: (4)              axis : int, optional
447: (8)                  Axis along which to average `a`. If None, averaging is done over
448: (8)                  the flattened array.
449: (4)              weights : array_like, optional
450: (8)                  The importance that each element has in the computation of the
average.
451: (8)                  The weights array can either be 1-D (in which case its length must be
452: (8)                  the size of `a` along the given axis) or of the same shape as `a`.
453: (8)                  If ``weights=None``, then all data in `a` are assumed to have a
454: (8)                  weight equal to one. The 1-D calculation is::
455: (12)                  avg = sum(a * weights) / sum(weights)
456: (8)                  The only constraint on `weights` is that `sum(weights)` must not be 0.
457: (4)              returned : bool, optional
458: (8)                  Flag indicating whether a tuple ``(`result, sum of weights)``
459: (8)                  should be returned as output (True), or just the result (False).
460: (8)                  Default is False.
461: (4)              keepdims : bool, optional
462: (8)                  If this is set to True, the axes which are reduced are left

```

```

463: (8)           in the result as dimensions with size one. With this option,
464: (8)           the result will broadcast correctly against the original `a`.
465: (8)           *Note:* `keepdims` will not work with instances of `numpy.matrix`
466: (8)           or other classes whose methods do not support `keepdims`.
467: (8)           .. versionadded:: 1.23.0
468: (4)           Returns
469: (4)           -----
470: (4)           average, [sum_of_weights] : (tuple of) scalar or MaskedArray
471: (8)           The average along the specified axis. When returned is `True`,
472: (8)           return a tuple with the average as the first element and the sum
473: (8)           of the weights as the second element. The return type is `np.float64`
474: (8)           if `a` is of integer type and floats smaller than `float64`, or the
475: (8)           input data-type, otherwise. If returned, `sum_of_weights` is always
476: (8)           `float64`.
477: (4)           Examples
478: (4)           -----
479: (4)           >>> a = np.ma.array([1., 2., 3., 4.], mask=[False, False, True, True])
480: (4)           >>> np.ma.average(a, weights=[3, 1, 0, 0])
481: (4)           1.25
482: (4)           >>> x = np.ma.arange(6.).reshape(3, 2)
483: (4)           >>> x
484: (4)           masked_array(
485: (6)             data=[[0., 1.],
486: (12)               [2., 3.],
487: (12)               [4., 5.]],
488: (6)             mask=False,
489: (6)             fill_value=1e+20)
490: (4)           >>> avg, sumweights = np.ma.average(x, axis=0, weights=[1, 2, 3],
491: (4)                           ...                               returned=True)
492: (4)           >>> avg
493: (4)           masked_array(data=[2.6666666666666665, 3.6666666666666665],
494: (17)             mask=[False, False],
495: (11)             fill_value=1e+20)
496: (4)           With ``keepdims=True``, the following result has shape (3, 1).
497: (4)           >>> np.ma.average(x, axis=1, keepdims=True)
498: (4)           masked_array(
499: (6)             data=[[0.5],
500: (12)               [2.5],
501: (12)               [4.5]],
502: (6)             mask=False,
503: (6)             fill_value=1e+20)
504: (4)           """
505: (4)           a = asarray(a)
506: (4)           m = getmask(a)
507: (4)           if keepdims is np._NoValue:
508: (8)             keepdims_kw = {}
509: (4)           else:
510: (8)             keepdims_kw = {'keepdims': keepdims}
511: (4)           if weights is None:
512: (8)             avg = a.mean(axis, **keepdims_kw)
513: (8)             scl = avg.dtype.type(a.count(axis))
514: (4)           else:
515: (8)             wgt = asarray(weights)
516: (8)             if issubclass(a.dtype.type, (np.integer, np.bool_)):
517: (12)               result_dtype = np.result_type(a.dtype, wgt.dtype, 'f8')
518: (8)             else:
519: (12)               result_dtype = np.result_type(a.dtype, wgt.dtype)
520: (8)             if a.shape != wgt.shape:
521: (12)               if axis is None:
522: (16)                 raise TypeError(
523: (20)                   "Axis must be specified when shapes of a and weights "
524: (20)                   "differ.")
525: (12)               if wgt.ndim != 1:
526: (16)                 raise TypeError(
527: (20)                   "1D weights expected when shapes of a and weights "
528: (12)                   "differ.")
529: (16)               if wgt.shape[0] != a.shape[axis]:
530: (20)                 raise ValueError(
530: (20)                   "Length of weights not compatible with specified axis.")

```

```

531: (12)
531: subok=True)
532: (12)
532: wgt = np.broadcast_to(wgt, (a.ndim-1)*(1,) + wgt.shape,
533: (8)
533:         wgt = wgt.swapaxes(-1, axis)
534: (12)
534:     if m is not nomask:
535: (12)
535:         wgt = wgt*(~a.mask)
536: (8)
536:         wgt.mask |= a.mask
537: (8)
537:         scl = wgt.sum(axis=axis, dtype=result_dtype, **keepdims_kw)
538: (26)
538:         avg = np.multiply(a, wgt,
539: (4)                         dtype=result_dtype).sum(axis, **keepdims_kw) / scl
539: if returned:
540: (8)
540:     if scl.shape != avg.shape:
541: (12)
541:         scl = np.broadcast_to(scl, avg.shape).copy()
542: (8)
542:         return avg, scl
543: (4)
543: else:
544: (8)
544:     return avg
545: (0)
545: def median(a, axis=None, out=None, overwrite_input=False, keepdims=False):
546: (4)
546: """
547: (4)
547: Compute the median along the specified axis.
548: (4)
548: Returns the median of the array elements.
549: (4)
549: Parameters
550: (4)
551: (4)
551: a : array_like
552: (8)
552:     Input array or object that can be converted to an array.
553: (4)
554: (8)
554:     Axis along which the medians are computed. The default (None) is
555: (8)
555:     to compute the median along a flattened version of the array.
556: (4)
556: out : ndarray, optional
557: (8)
557:     Alternative output array in which to place the result. It must
558: (8)
558:     have the same shape and buffer length as the expected output
559: (8)
559:     but the type will be cast if necessary.
560: (4)
560: overwrite_input : bool, optional
561: (8)
561:     If True, then allow use of memory of input array (a) for
562: (8)
562:     calculations. The input array will be modified by the call to
563: (8)
563:     median. This will save memory when you do not need to preserve
564: (8)
564:     the contents of the input array. Treat the input as undefined,
565: (8)
565:     but it will probably be fully or partially sorted. Default is
566: (8)
566:     False. Note that, if `overwrite_input` is True, and the input
567: (8)
567:     is not already an `ndarray`, an error will be raised.
568: (4)
568: keepdims : bool, optional
569: (8)
569:     If this is set to True, the axes which are reduced are left
570: (8)
570:     in the result as dimensions with size one. With this option,
571: (8)
571:     the result will broadcast correctly against the input array.
572: (8)
572: .. versionadded:: 1.10.0
573: (4)
573: Returns
574: (4)
575: (4)
575: median : ndarray
576: (8)
576:     A new array holding the result is returned unless out is
577: (8)
577:     specified, in which case a reference to out is returned.
578: (8)
578:     Return data-type is `float64` for integers and floats smaller than
579: (8)
579:     `float64`, or the input data-type, otherwise.
580: (4)
580: See Also
581: (4)
581: -----
582: (4)
582: mean
583: (4)
583: Notes
584: (4)
584: -----
585: (4)
585: Given a vector ``V`` with ``N`` non masked values, the median of ``V``
586: (4)
586: is the middle value of a sorted copy of ``V`` (``Vs``) - i.e.
587: (4)
587: ``Vs[(N-1)/2]``, when ``N`` is odd, or ``{Vs[N/2 - 1] + Vs[N/2]}/2`` when ``N`` is even.
588: (4)
588: Examples
589: (4)
589: -----
590: (4)
590: >>> x = np.ma.array(np.arange(8), mask=[0]*4 + [1]*4)
591: (4)
591: >>> np.ma.median(x)
592: (4)
592: 1.5
593: (4)
593: >>> x = np.ma.array(np.arange(10).reshape(2, 5), mask=[0]*6 + [1]*4)
594: (4)
594: >>> np.ma.median(x)
595: (4)
595: 2.5
596: (4)
596: >>> np.ma.median(x, axis=-1, overwrite_input=True)
597: (4)
597: masked_array(data=[2.0, 5.0],
598: (4)

```

```

599: (17)                         mask=[False, False],
600: (11)                         fill_value=1e+20)
601: (4)
602: (4)
603: (8)                         if not hasattr(a, 'mask'):
604: (22)                           m = np.median(getdata(a, subok=True), axis=axis,
605: (22)                                         out=out, overwrite_input=overwrite_input,
606: (8)                                           keepdims=keepdims)
607: (12)                           if isinstance(m, np.ndarray) and 1 <= m.ndim:
608: (8)                             return masked_array(m, copy=False)
609: (12)                           else:
610: (4)                             return m
611: (20)                         return _ureduce(a, func=_median, keepdims=keepdims, axis=axis, out=out,
612: (0)                                         overwrite_input=overwrite_input)
def _median(a, axis=None, out=None, overwrite_input=False):
613: (4)                         if np.issubdtype(a.dtype, np.inexact):
614: (8)                           fill_value = np.inf
615: (4)                         else:
616: (8)                           fill_value = None
617: (4)                         if overwrite_input:
618: (8)                           if axis is None:
619: (12)                             asorted = a.ravel()
620: (12)                             asorted.sort(fill_value=fill_value)
621: (8)                           else:
622: (12)                             a.sort(axis=axis, fill_value=fill_value)
623: (12)                             asorted = a
624: (4)
625: (8)                         asorted = sort(a, axis=axis, fill_value=fill_value)
626: (4)                         if axis is None:
627: (8)                           axis = 0
628: (4)
629: (8)                           axis = normalize_axis_index(axis, asorted.ndim)
630: (4)                         if asorted.shape[axis] == 0:
631: (8)                           indexer = [slice(None)] * asorted.ndim
632: (8)                           indexer[axis] = slice(0, 0)
633: (8)                           indexer = tuple(indexer)
634: (8)                           return np.ma.mean(asorted[indexer], axis=axis, out=out)
635: (4)                         if asorted.ndim == 1:
636: (8)                           idx, odd = divmod(count(asorted), 2)
637: (8)                           mid = asorted[idx + odd - 1:idx + 1]
638: (8)                           if np.issubdtype(asorted.dtype, np.inexact) and asorted.size > 0:
639: (12)                             s = mid.sum(out=out)
640: (12)                             if not odd:
641: (16)                               s = np.true_divide(s, 2., casting='safe', out=out)
642: (12)                               s = np.lib.utils._median_nancheck(asorted, s, axis)
643: (8)
644: (12)                             s = mid.mean(out=out)
645: (8)                             if np.ma.is_masked(s) and not np.all(asorted.mask):
646: (12)                               return np.ma.minimum_fill_value(asorted)
647: (8)
648: (4)                         counts = count(asorted, axis=axis, keepdims=True)
649: (4)                         h = counts // 2
650: (4)                         odd = counts % 2 == 1
651: (4)                         l = np.where(odd, h, h-1)
652: (4)                         lh = np.concatenate([l,h], axis=axis)
653: (4)                         low_high = np.take_along_axis(asorted, lh, axis=axis)
654: (4)
655: (8)                         def replace_masked(s):
656: (12)                           if np.ma.is_masked(s):
657: (12)                             rep = (~np.all(asorted.mask, axis=axis, keepdims=True)) & s.mask
658: (12)                             s.data[rep] = np.ma.minimum_fill_value(asorted)
659: (4)                             s.mask[rep] = False
660: (4)                         replace_masked(low_high)
661: (4)                         if np.issubdtype(asorted.dtype, np.inexact):
662: (8)                           s = np.ma.sum(low_high, axis=axis, out=out)
663: (8)                           np.true_divide(s.data, 2., casting='unsafe', out=s.data)
664: (4)                           s = np.lib.utils._median_nancheck(asorted, s, axis)
665: (8)                           else:
666: (4)                             s = np.ma.mean(low_high, axis=axis, out=out)
667: (0)                         return s
def compress_nd(x, axis=None):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

668: (4)         """Suppress slices from multiple dimensions which contain masked values.
669: (4)         Parameters
670: (4)         -----
671: (4)         x : array_like, MaskedArray
672: (8)           The array to operate on. If not a MaskedArray instance (or if no array
673: (8)           elements are masked), `x` is interpreted as a MaskedArray with `mask`
674: (8)           set to `nomask`.
675: (4)         axis : tuple of ints or int, optional
676: (8)           Which dimensions to suppress slices from can be configured with this
677: (8)           parameter.
678: (8)             - If axis is a tuple of ints, those are the axes to suppress slices
from.
679: (8)             - If axis is an int, then that is the only axis to suppress slices
from.
680: (8)             - If axis is None, all axis are selected.
681: (4)         Returns
682: (4)         -----
683: (4)         compress_array : ndarray
684: (8)           The compressed array.
685: (4)         """
686: (4)         x = asarray(x)
687: (4)         m = getmask(x)
688: (4)         if axis is None:
689: (8)           axis = tuple(range(x.ndim))
690: (4)         else:
691: (8)           axis = normalize_axis_tuple(axis, x.ndim)
692: (4)         if m is nomask or not m.any():
693: (8)           return x._data
694: (4)         if m.all():
695: (8)           return nxarray([])
696: (4)         data = x._data
697: (4)         for ax in axis:
698: (8)           axes = tuple(list(range(ax)) + list(range(ax + 1, x.ndim)))
699: (8)           data = data[(slice(None),)*ax + (~m.any(axis=axes),)]
700: (4)         return data
701: (0)     def compress_rowcols(x, axis=None):
702: (4)         """
703: (4)         Suppress the rows and/or columns of a 2-D array that contain
704: (4)         masked values.
705: (4)         The suppression behavior is selected with the `axis` parameter.
706: (4)         - If axis is None, both rows and columns are suppressed.
707: (4)         - If axis is 0, only rows are suppressed.
708: (4)         - If axis is 1 or -1, only columns are suppressed.
709: (4)         Parameters
710: (4)         -----
711: (4)         x : array_like, MaskedArray
712: (8)           The array to operate on. If not a MaskedArray instance (or if no
array
713: (8)           elements are masked), `x` is interpreted as a MaskedArray with
714: (8)           `mask` set to `nomask`. Must be a 2D array.
715: (4)         axis : int, optional
716: (8)           Axis along which to perform the operation. Default is None.
717: (4)         Returns
718: (4)         -----
719: (4)         compressed_array : ndarray
720: (8)           The compressed array.
721: (4)         Examples
722: (4)         -----
723: (4)         >>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
724: (4)                         ...                               [1, 0, 0],
725: (4)                         ...                               [0, 0, 0]])
726: (4)         >>> x
727: (4)         masked_array(
728: (6)             data=[[--, 1, 2],
729: (12)                 [--, 4, 5],
730: (12)                 [6, 7, 8]],
731: (6)             mask=[[ True, False, False],
732: (12)                 [ True, False, False],
733: (12)                 [False, False, False]],

```

```

734: (6)           fill_value=999999)
735: (4)           >>> np.ma.compress_rowcols(x)
736: (4)           array([[7, 8]])
737: (4)           >>> np.ma.compress_rowcols(x, 0)
738: (4)           array([[6, 7, 8]])
739: (4)           >>> np.ma.compress_rowcols(x, 1)
740: (4)           array([[1, 2],
741: (11)             [4, 5],
742: (11)             [7, 8]])
743: (4)           """
744: (4)           if asarray(x).ndim != 2:
745: (8)             raise NotImplementedError("compress_rowcols works for 2D arrays
only.")
746: (4)           return compress_nd(x, axis=axis)
747: (0) def compress_rows(a):
748: (4)           """
749: (4)           Suppress whole rows of a 2-D array that contain masked values.
750: (4)           This is equivalent to ``np.ma.compress_rowcols(a, 0)``, see
751: (4)           `compress_rowcols` for details.
752: (4)           See Also
753: (4)           -----
754: (4)           compress_rowcols
755: (4)           """
756: (4)           a = asarray(a)
757: (4)           if a.ndim != 2:
758: (8)             raise NotImplementedError("compress_rows works for 2D arrays only.")
759: (4)           return compress_rowcols(a, 0)
760: (0) def compress_cols(a):
761: (4)           """
762: (4)           Suppress whole columns of a 2-D array that contain masked values.
763: (4)           This is equivalent to ``np.ma.compress_rowcols(a, 1)``, see
764: (4)           `compress_rowcols` for details.
765: (4)           See Also
766: (4)           -----
767: (4)           compress_rowcols
768: (4)           """
769: (4)           a = asarray(a)
770: (4)           if a.ndim != 2:
771: (8)             raise NotImplementedError("compress_cols works for 2D arrays only.")
772: (4)           return compress_rowcols(a, 1)
773: (0) def mask_rowcols(a, axis=None):
774: (4)           """
775: (4)           Mask rows and/or columns of a 2D array that contain masked values.
776: (4)           Mask whole rows and/or columns of a 2D array that contain
777: (4)           masked values. The masking behavior is selected using the
778: (4)           `axis` parameter.
779: (6)           - If `axis` is None, rows *and* columns are masked.
780: (6)           - If `axis` is 0, only rows are masked.
781: (6)           - If `axis` is 1 or -1, only columns are masked.
782: (4)           Parameters
783: (4)           -----
784: (4)           a : array_like, MaskedArray
785: (8)             The array to mask. If not a MaskedArray instance (or if no array
786: (8)             elements are masked), the result is a MaskedArray with `mask` set
787: (8)             to `nomask` (False). Must be a 2D array.
788: (4)           axis : int, optional
789: (8)             Axis along which to perform the operation. If None, applies to a
790: (8)             flattened version of the array.
791: (4)           Returns
792: (4)           -----
793: (4)           a : MaskedArray
794: (8)             A modified version of the input array, masked depending on the value
795: (8)             of the `axis` parameter.
796: (4)           Raises
797: (4)           -----
798: (4)           NotImplementedError
799: (8)             If input array `a` is not 2D.
800: (4)           See Also
801: (4)           -----

```

```

802: (4)                         mask_rows : Mask rows of a 2D array that contain masked values.
803: (4)                         mask_cols : Mask cols of a 2D array that contain masked values.
804: (4)                         masked_where : Mask where a condition is met.
805: (4)                         Notes
806: (4)                         -----
807: (4)                         The input array's mask is modified by this function.
808: (4)                         Examples
809: (4)                         -----
810: (4)                         >>> import numpy.ma as ma
811: (4)                         >>> a = np.zeros((3, 3), dtype=int)
812: (4)                         >>> a[1, 1] = 1
813: (4)                         >>> a
814: (4)                         array([[0, 0, 0],
815: (11)                           [0, 1, 0],
816: (11)                           [0, 0, 0]])
817: (4)                         >>> a = ma.masked_equal(a, 1)
818: (4)                         >>> a
819: (4)                         masked_array(
820: (6)                           data=[[0, 0, 0],
821: (12)                             [0, --, 0],
822: (12)                             [0, 0, 0]],
823: (6)                           mask=[[False, False, False],
824: (12)                             [False, True, False],
825: (12)                             [False, False, False]],
826: (6)                           fill_value=1)
827: (4)                         >>> ma.mask_rowcols(a)
828: (4)                         masked_array(
829: (6)                           data=[[0, --, 0],
830: (12)                             [--, --, --],
831: (12)                             [0, --, 0]],
832: (6)                           mask=[[False, True, False],
833: (12)                             [True, True, True],
834: (12)                             [False, True, False]],
835: (6)                           fill_value=1)
836: (4)                         """
837: (4)                         a = array(a, subok=False)
838: (4)                         if a.ndim != 2:
839: (8)                           raise NotImplementedError("mask_rowcols works for 2D arrays only.")
840: (4)                         m = getmask(a)
841: (4)                         if m is nomask or not m.any():
842: (8)                           return a
843: (4)                         maskedval = m.nonzero()
844: (4)                         a._mask = a._mask.copy()
845: (4)                         if not axis:
846: (8)                           a[np.unique(maskedval[0])] = masked
847: (4)                         if axis in [None, 1, -1]:
848: (8)                           a[:, np.unique(maskedval[1])] = masked
849: (4)                         return a
850: (0)                         def mask_rows(a, axis=np._NoValue):
851: (4)                         """
852: (4)                         Mask rows of a 2D array that contain masked values.
853: (4)                         This function is a shortcut to ``mask_rowcols`` with `axis` equal to 0.
854: (4)                         See Also
855: (4)                         -----
856: (4)                         mask_rowcols : Mask rows and/or columns of a 2D array.
857: (4)                         masked_where : Mask where a condition is met.
858: (4)                         Examples
859: (4)                         -----
860: (4)                         >>> import numpy.ma as ma
861: (4)                         >>> a = np.zeros((3, 3), dtype=int)
862: (4)                         >>> a[1, 1] = 1
863: (4)                         >>> a
864: (4)                         array([[0, 0, 0],
865: (11)                           [0, 1, 0],
866: (11)                           [0, 0, 0]])
867: (4)                         >>> a = ma.masked_equal(a, 1)
868: (4)                         >>> a
869: (4)                         masked_array(
870: (6)                           data=[[0, 0, 0],

```

```

871: (12)                 [0, --, 0],
872: (12)                 [0, 0, 0]],
873: (6)                  mask=[[False, False, False],
874: (12)                   [False, True, False],
875: (12)                   [False, False, False]],
876: (6)                  fill_value=1)
877: (4)                  >>> ma.mask_rows(a)
878: (4)                  masked_array(
879: (6)                    data=[[0, 0, 0],
880: (12)                      [--, --, --],
881: (12)                      [0, 0, 0]],
882: (6)                    mask=[[False, False, False],
883: (12)                      [True, True, True],
884: (12)                      [False, False, False]],
885: (6)                    fill_value=1)
886: (4)                  """
887: (4)                  if axis is not np._NoValue:
888: (8)                      warnings.warn(
889: (12)                        "The axis argument has always been ignored, in future passing it "
890: (12)                        "will raise TypeError", DeprecationWarning, stacklevel=2)
891: (4)                  return mask_rowcols(a, 0)
892: (0)                  def mask_cols(a, axis=np._NoValue):
893: (4)                  """
894: (4)                  Mask columns of a 2D array that contain masked values.
895: (4)                  This function is a shortcut to ``mask_rowcols`` with `axis` equal to 1.
896: (4)                  See Also
897: (4)                  -----
898: (4)                  mask_rowcols : Mask rows and/or columns of a 2D array.
899: (4)                  masked_where : Mask where a condition is met.
900: (4)                  Examples
901: (4)                  -----
902: (4)                  >>> import numpy.ma as ma
903: (4)                  >>> a = np.zeros((3, 3), dtype=int)
904: (4)                  >>> a[1, 1] = 1
905: (4)                  >>> a
906: (4)                  array([[0, 0, 0],
907: (11)                    [0, 1, 0],
908: (11)                    [0, 0, 0]])
909: (4)                  >>> a = ma.masked_equal(a, 1)
910: (4)                  >>> a
911: (4)                  masked_array(
912: (6)                    data=[[0, 0, 0],
913: (12)                      [0, --, 0],
914: (12)                      [0, 0, 0]],
915: (6)                    mask=[[False, False, False],
916: (12)                      [False, True, False],
917: (12)                      [False, False, False]],
918: (6)                    fill_value=1)
919: (4)                  >>> ma.mask_cols(a)
920: (4)                  masked_array(
921: (6)                    data=[[0, --, 0],
922: (12)                      [0, --, 0],
923: (12)                      [0, --, 0]],
924: (6)                    mask=[[False, True, False],
925: (12)                      [False, True, False],
926: (12)                      [False, True, False]],
927: (6)                    fill_value=1)
928: (4)                  """
929: (4)                  if axis is not np._NoValue:
930: (8)                      warnings.warn(
931: (12)                        "The axis argument has always been ignored, in future passing it "
932: (12)                        "will raise TypeError", DeprecationWarning, stacklevel=2)
933: (4)                  return mask_rowcols(a, 1)
934: (0)                  def ediff1d(arr, to_end=None, to_begin=None):
935: (4)                  """
936: (4)                  Compute the differences between consecutive elements of an array.
937: (4)                  This function is the equivalent of `numpy.ediff1d` that takes masked
938: (4)                  values into account, see `numpy.ediff1d` for details.
939: (4)                  See Also

```

```

940: (4)      -----
941: (4)      numpy.ediff1d : Equivalent function for ndarrays.
942: (4)
943: (4)      """
944: (4)      arr = ma.asanyarray(arr).flat
945: (4)      ed = arr[1:] - arr[:-1]
946: (4)      arrays = [ed]
947: (8)      if to_begin is not None:
948: (4)          arrays.insert(0, to_begin)
949: (8)      if to_end is not None:
950: (4)          arrays.append(to_end)
951: (8)      if len(arrays) != 1:
952: (4)          ed = hstack(arrays)
953: (0)      return ed
954: (4)
955: (4)      def unique(ar1, return_index=False, return_inverse=False):
956: (4)          """
957: (4)          Finds the unique elements of an array.
958: (4)          Masked values are considered the same element (masked). The output array
959: (4)          is always a masked array. See `numpy.unique` for more details.
960: (4)          See Also
961: (4)          -----
962: (4)          numpy.unique : Equivalent function for ndarrays.
963: (4)          Examples
964: (4)          -----
965: (4)          >>> import numpy.ma as ma
966: (4)          >>> a = [1, 2, 1000, 2, 3]
967: (4)          >>> mask = [0, 0, 1, 0, 0]
968: (4)          >>> masked_a = ma.masked_array(a, mask)
969: (16)         masked_array(data=[1, 2, --, 2, 3],
970: (8)             mask=[False, False, True, False, False],
971: (4)             fill_value=999999)
972: (4)         >>> ma.unique(masked_a)
973: (16)         masked_array(data=[1, 2, 3, --],
974: (8)             mask=[False, False, False, True],
975: (4)             fill_value=999999)
976: (4)         >>> ma.unique(masked_a, return_index=True)
977: (16)         (masked_array(data=[1, 2, 3, --],
978: (8)             mask=[False, False, False, True],
979: (4)             fill_value=999999), array([0, 1, 4, 2]))
980: (4)         >>> ma.unique(masked_a, return_inverse=True)
981: (16)         (masked_array(data=[1, 2, 3, --],
982: (8)             mask=[False, False, False, True],
983: (4)             fill_value=999999), array([0, 1, 3, 1, 2]))
984: (4)         >>> ma.unique(masked_a, return_index=True, return_inverse=True)
985: (16)         (masked_array(data=[1, 2, 3, --],
986: (8)             mask=[False, False, False, True],
987: (4)             fill_value=999999), array([0, 1, 4, 2]), array([0, 1, 3, 1, 2]))
988: (4)
989: (23)        """
990: (23)        output = np.unique(ar1,
991: (4)            return_index=return_index,
992: (4)            return_inverse=return_inverse)
993: (8)        if isinstance(output, tuple):
994: (8)            output = list(output)
995: (8)            output[0] = output[0].view(MaskedArray)
996: (8)            output = tuple(output)
997: (4)
998: (0)        def intersect1d(ar1, ar2, assume_unique=False):
999: (4)
1000: (4)            """
1001: (4)            Returns the unique elements common to both arrays.
1002: (4)            Masked values are considered equal one to the other.
1003: (4)            The output is always a masked array.
1004: (4)            See `numpy.intersect1d` for more details.
1005: (4)            See Also
1006: (4)            -----
1007: (4)            numpy.intersect1d : Equivalent function for ndarrays.
1008: (4)            Examples
-----
```

```

1009: (4)          >>> x = np.ma.array([1, 3, 3, 3], mask=[0, 0, 0, 1])
1010: (4)          >>> y = np.ma.array([3, 1, 1, 1], mask=[0, 0, 0, 1])
1011: (4)          >>> np.ma.intersect1d(x, y)
1012: (4)          masked_array(data=[1, 3, --],
1013: (17)             mask=[False, False, True],
1014: (11)             fill_value=999999)
1015: (4)          """
1016: (4)          if assume_unique:
1017: (8)              aux = ma.concatenate((ar1, ar2))
1018: (4)          else:
1019: (8)              aux = ma.concatenate((unique(ar1), unique(ar2)))
1020: (4)          aux.sort()
1021: (4)          return aux[:-1][aux[1:] == aux[:-1]]
1022: (0)          def setxor1d(ar1, ar2, assume_unique=False):
1023: (4)          """
1024: (4)          Set exclusive-or of 1-D arrays with unique elements.
1025: (4)          The output is always a masked array. See `numpy.setxor1d` for more
details.
1026: (4)          See Also
1027: (4)          -----
1028: (4)          numpy.setxor1d : Equivalent function for ndarrays.
1029: (4)          """
1030: (4)          if not assume_unique:
1031: (8)              ar1 = unique(ar1)
1032: (8)              ar2 = unique(ar2)
1033: (4)              aux = ma.concatenate((ar1, ar2))
1034: (4)              if aux.size == 0:
1035: (8)                  return aux
1036: (4)              aux.sort()
1037: (4)              auxf = aux.filled()
1038: (4)              flag = ma.concatenate(([True], (auxf[1:] != auxf[:-1]), [True]))
1039: (4)              flag2 = (flag[1:] == flag[:-1])
1040: (4)              return aux[flag2]
1041: (0)          def in1d(ar1, ar2, assume_unique=False, invert=False):
1042: (4)          """
1043: (4)          Test whether each element of an array is also present in a second
array.
1044: (4)          The output is always a masked array. See `numpy.in1d` for more details.
1045: (4)          We recommend using :func:`isin` instead of `in1d` for new code.
1046: (4)          See Also
1047: (4)          -----
1048: (4)          isin      : Version of this function that preserves the shape of ar1.
1049: (4)          numpy.in1d : Equivalent function for ndarrays.
1050: (4)          Notes
1051: (4)          -----
1052: (4)          .. versionadded:: 1.4.0
1053: (4)          """
1054: (4)          if not assume_unique:
1055: (4)              ar1, rev_idx = unique(ar1, return_inverse=True)
1056: (8)              ar2 = unique(ar2)
1057: (8)              ar = ma.concatenate((ar1, ar2))
1058: (4)              order = ar.argsort(kind='mergesort')
1059: (4)              sar = ar[order]
1060: (4)              if invert:
1061: (4)                  bool_ar = (sar[1:] != sar[:-1])
1062: (8)              else:
1063: (4)                  bool_ar = (sar[1:] == sar[:-1])
1064: (8)              flag = ma.concatenate((bool_ar, [invert]))
1065: (4)              indx = order.argsort(kind='mergesort')[:len(ar1)]
1066: (4)              if assume_unique:
1067: (4)                  return flag[indx]
1068: (8)              else:
1069: (4)                  return flag[indx][rev_idx]
1070: (8)          def isin(element, test_elements, assume_unique=False, invert=False):
1071: (0)          """
1072: (4)          Calculates `element in test_elements`, broadcasting over
`element` only.
1073: (4)          The output is always a masked array of the same shape as `element`.
1074: (4)          See `numpy.isin` for more details.
1075: (4)
1076: (4)

```

```

1077: (4)           See Also
1078: (4)
1079: (4)           -----
1080: (4)           in1d      : Flattened version of this function.
1081: (4)           numpy.isin : Equivalent function for ndarrays.
1082: (4)
1083: (4)           Notes
1084: (4)           -----
1085: (4)           .. versionadded:: 1.13.0
1086: (4)           """
1087: (16)          element = ma.asarray(element)
1088: (0)           return in1d(element, test_elements, assume_unique=assume_unique,
1089: (4)             invert=invert).reshape(element.shape)
1090: (4)           def union1d(ar1, ar2):
1091: (4)             """
1092: (4)             Union of two arrays.
1093: (4)             The output is always a masked array. See `numpy.union1d` for more details.
1094: (4)             See Also
1095: (4)             -----
1096: (4)             numpy.union1d : Equivalent function for ndarrays.
1097: (0)           """
1098: (4)           return unique(ma.concatenate((ar1, ar2), axis=None))
1099: (4)           def setdiff1d(ar1, ar2, assume_unique=False):
1100: (4)             """
1101: (4)             Set difference of 1D arrays with unique elements.
1102: (4)             The output is always a masked array. See `numpy.setdiff1d` for more
1103: (4)             details.
1104: (4)             See Also
1105: (4)             -----
1106: (4)             numpy.setdiff1d : Equivalent function for ndarrays.
1107: (4)             Examples
1108: (4)             -----
1109: (4)             >>> x = np.ma.array([1, 2, 3, 4], mask=[0, 1, 0, 1])
1110: (17)            >>> np.ma.setdiff1d(x, [1, 2])
1111: (11)            masked_array(data=[3, --],
1112: (4)              mask=[False, True],
1113: (4)              fill_value=999999)
1114: (8)             """
1115: (4)             if assume_unique:
1116: (8)               ar1 = ma.asarray(ar1).ravel()
1117: (8)             else:
1118: (4)               ar1 = unique(ar1)
1119: (0)             ar2 = unique(ar2)
1120: (4)             return ar1[in1d(ar1, ar2, assume_unique=True, invert=True)]
1121: (4)             def _covhelper(x, y=None, rowvar=True, allow_masked=True):
1122: (4)               """
1123: (4)               Private function for the computation of covariance and correlation
1124: (4)               coefficients.
1125: (4)               """
1126: (4)               x = ma.array(x, ndmin=2, copy=True, dtype=float)
1127: (8)               xmask = ma.getmaskarray(x)
1128: (4)               if not allow_masked and xmask.any():
1129: (8)                 raise ValueError("Cannot process masked data.")
1130: (4)               if x.shape[0] == 1:
1131: (8)                 rowvar = True
1132: (4)               rowvar = int(bool(rowvar))
1133: (4)               axis = 1 - rowvar
1134: (4)               if rowvar:
1135: (8)                 tup = (slice(None), None)
1136: (4)               else:
1137: (8)                 tup = (None, slice(None))
1138: (4)               if y is None:
1139: (8)                 xnotmask = np.logical_not(xmask).astype(int)
1140: (8)               else:
1141: (8)                 y = array(y, copy=False, ndmin=2, dtype=float)
1142: (12)                ymask = ma.getmaskarray(y)
1143: (8)                if not allow_masked and ymask.any():
1144: (12)                  raise ValueError("Cannot process masked data.")
1145: (16)                if y.shape == x.shape:
1146: (8)                  common_mask = np.logical_or(xmask, ymask)

```

```

1146: (16)
1147: (20)
1148: (20)
1149: (20)
1150: (8)
1151: (8)
axis)).astype(int)
1152: (4)
1153: (4)
1154: (0)
1155: (4)
1156: (4)
1157: (4)
1158: (4)
1159: (4)
1160: (4)
1161: (4)
1162: (4)
1163: (4)
1164: (4)
1165: (4)
1166: (4)
1167: (8)
1168: (8)
1169: (8)
1170: (4)
1171: (8)
1172: (8)
1173: (4)
1174: (8)
1175: (8)
relationship
1176: (8)
1177: (8)
1178: (4)
1179: (8)
1180: (8)
1181: (8)
1182: (8)
1183: (4)
1184: (8)
1185: (8)
1186: (8)
missing.
1187: (4)
1188: (8)
1189: (8)
1190: (8)
1191: (8)
1192: (4)
1193: (4)
1194: (4)
1195: (8)
1196: (4)
1197: (4)
1198: (4)
1199: (4)
1200: (4)
1201: (8)
1202: (4)
1203: (8)
1204: (12)
1205: (8)
1206: (12)
1207: (4)
1208: (4)
1209: (8)
1210: (8)
1211: (4)

        if common_mask is not nomask:
            xmask = x._mask = y._mask = ymask = common_mask
            x._sharedmask = False
            y._sharedmask = False
            x = ma.concatenate((x, y), axis)
            xnotmask = np.logical_not(np.concatenate((xmask, ymask),
axis)).astype(int)
        x -= x.mean(axis=rowvar)[tup]
    return (x, xnotmask, rowvar)
def cov(x, y=None, rowvar=True, bias=False, allow_masked=True, ddof=None):
"""
Estimate the covariance matrix.
Except for the handling of missing data this function does the same as
`numpy.cov`. For more details and examples, see `numpy.cov`.
By default, masked values are recognized as such. If `x` and `y` have the
same shape, a common mask is allocated: if ``x[i,j]`` is masked, then
``y[i,j]`` will also be masked.
Setting `allow_masked` to False will raise an exception if values are
missing in either of the input arrays.
Parameters
-----
x : array_like
    A 1-D or 2-D array containing multiple variables and observations.
    Each row of `x` represents a variable, and each column a single
    observation of all those variables. Also see `rowvar` below.
y : array_like, optional
    An additional set of variables and observations. `y` has the same
    shape as `x`.
rowvar : bool, optional
    If `rowvar` is True (default), then each row represents a
    variable, with observations in the columns. Otherwise, the
    is transposed: each column represents a variable, while the rows
    contain observations.
bias : bool, optional
    Default normalization (False) is by ``N-1``, where ``N`` is the
    number of observations given (unbiased estimate). If `bias` is True,
    then normalization is by ``N``. This keyword can be overridden by
    the keyword ``ddof`` in numpy versions >= 1.5.
allow_masked : bool, optional
    If True, masked values are propagated pair-wise: if a value is masked
    in `x`, the corresponding value is masked in `y`.
    If False, raises a `ValueError` exception when some values are
missing.
ddof : {None, int}, optional
    If not ``None`` normalization is by ``N - ddof``, where ``N`` is
    the number of observations; this overrides the value implied by
    ``bias``. The default value is ``None``.
.. versionadded:: 1.5
Raises
-----
ValueError
    Raised if some values are missing and `allow_masked` is False.
See Also
-----
numpy.cov
"""
if ddof is not None and ddof != int(ddof):
    raise ValueError("ddof must be an integer")
if ddof is None:
    if bias:
        ddof = 0
    else:
        ddof = 1
(x, xnotmask, rowvar) = _covhelper(x, y, rowvar, allow_masked)
if not rowvar:
    fact = np.dot(xnotmask.T, xnotmask) * 1. - ddof
    result = (dot(x.T, x.conj(), strict=False) / fact).squeeze()
else:

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1212: (8)           fact = np.dot(xnotmask, xnotmask.T) * 1. - ddof
1213: (8)           result = (dot(x, x.T.conj(), strict=False) / fact).squeeze()
1214: (4)           return result
1215: (0) def corrcoef(x, y=None, rowvar=True, bias=np._NoValue, allow_masked=True,
1216: (13)             ddof=np._NoValue):
1217: (4)             """
1218: (4)             Return Pearson product-moment correlation coefficients.
1219: (4)             Except for the handling of missing data this function does the same as
1220: (4)             `numpy.corrcoef`. For more details and examples, see `numpy.corrcoef`.
1221: (4)             Parameters
1222: (4)             -----
1223: (4)             x : array_like
1224: (8)               A 1-D or 2-D array containing multiple variables and observations.
1225: (8)               Each row of `x` represents a variable, and each column a single
1226: (8)               observation of all those variables. Also see `rowvar` below.
1227: (4)             y : array_like, optional
1228: (8)               An additional set of variables and observations. `y` has the same
1229: (8)               shape as `x`.
1230: (4)             rowvar : bool, optional
1231: (8)               If `rowvar` is True (default), then each row represents a
1232: (8)               variable, with observations in the columns. Otherwise, the
relationship
1233: (8)               is transposed: each column represents a variable, while the rows
1234: (8)               contain observations.
1235: (4)             bias : _NoValue, optional
1236: (8)               Has no effect, do not use.
1237: (8)               .. deprecated:: 1.10.0
1238: (4)             allow_masked : bool, optional
1239: (8)               If True, masked values are propagated pair-wise: if a value is masked
in `x`, the corresponding value is masked in `y`.
1240: (8)               If False, raises an exception. Because `bias` is deprecated, this
1241: (8)               argument needs to be treated as keyword only to avoid a warning.
1242: (8)             ddof : _NoValue, optional
1243: (4)               Has no effect, do not use.
1244: (8)               .. deprecated:: 1.10.0
1245: (8)             See Also
1246: (4)             -----
1247: (4)             numpy.corrcoef : Equivalent function in top-level NumPy module.
1248: (4)             cov : Estimate the covariance matrix.
1249: (4)             Notes
1250: (4)             -----
1251: (4)             This function accepts but discards arguments `bias` and `ddof`. This is
1252: (4)             for backwards compatibility with previous versions of this function.
These
1253: (4)
1254: (4)             arguments had no effect on the return values of the function and can be
1255: (4)             safely ignored in this and previous versions of numpy.
1256: (4)             """
1257: (4)             msg = 'bias and ddof have no effect and are deprecated'
1258: (4)             if bias is not np._NoValue or ddof is not np._NoValue:
1259: (8)               warnings.warn(msg, DeprecationWarning, stacklevel=2)
1260: (4)             (x, xnotmask, rowvar) = _covhelper(x, y, rowvar, allow_masked)
1261: (4)             if not rowvar:
1262: (8)               fact = np.dot(xnotmask.T, xnotmask) * 1.
1263: (8)               c = (dot(x.T, x.conj(), strict=False) / fact).squeeze()
1264: (4)             else:
1265: (8)               fact = np.dot(xnotmask, xnotmask.T) * 1.
1266: (8)               c = (dot(x, x.T.conj(), strict=False) / fact).squeeze()
try:
1267: (4)               diag = ma.diagonal(c)
except ValueError:
1268: (8)               return 1
1269: (4)             if xnotmask.all():
1270: (8)               _denom = ma.sqrt(ma.multiply.outer(diag, diag))
else:
1271: (8)               _denom = diagflat(diag)
1272: (8)               _denom._sharedmask = False # We know return is always a copy
1273: (4)             n = x.shape[1 - rowvar]
1274: (8)             if rowvar:
1275: (8)               for i in range(n - 1):

```

```

1279: (16)                     for j in range(i + 1, n):
1280: (20)                         _x = mask_cols(vstack((x[i], x[j]))).var(axis=1)
1281: (20)                         _denom[i, j] = _denom[j, i] =
ma.sqrt(ma.multiply.reduce(_x))
1282: (8)                     else:
1283: (12)                         for i in range(n - 1):
1284: (16)                             for j in range(i + 1, n):
1285: (20)                                 _x = mask_cols(
1286: (28)                                     vstack((x[:, i], x[:, j]))).var(axis=1)
1287: (20)                                 _denom[i, j] = _denom[j, i] =
ma.sqrt(ma.multiply.reduce(_x))
1288: (4)                     return c / _denom
1289: (0)             class MAxisConcatenator(AxisConcatenator):
1290: (4)                 """
1291: (4)                     Translate slice objects to concatenation along an axis.
1292: (4)                     For documentation on usage, see `mr_class`.
1293: (4)                     See Also
1294: (4)                     -----
1295: (4)                     mr_class
1296: (4)                     """
1297: (4)                     concatenate = staticmethod(concatenate)
1298: (4)                     @classmethod
1299: (4)                     def makemat(cls, arr):
1300: (8)                         data = super().makemat(arr.data, copy=False)
1301: (8)                         return array(data, mask=arr.mask)
1302: (4)                     def __getitem__(self, key):
1303: (8)                         if isinstance(key, str):
1304: (12)                             raise MAError("Unavailable for masked array.")
1305: (8)                         return super().__getitem__(key)
1306: (0)             class mr_class(MAxisConcatenator):
1307: (4)                 """
1308: (4)                     Translate slice objects to concatenation along the first axis.
1309: (4)                     This is the masked array version of `lib.index_tricks.RClass`.
1310: (4)                     See Also
1311: (4)                     -----
1312: (4)                     lib.index_tricks.RClass
1313: (4)                     Examples
1314: (4)                     -----
1315: (4)                     >>> np.ma.mr_[np.ma.array([1,2,3]), 0, 0, np.ma.array([4,5,6])]
1316: (4)                     masked_array(data=[1, 2, 3, ..., 4, 5, 6],
1317: (17)                         mask=False,
1318: (11)                         fill_value=999999)
1319: (4)                     """
1320: (4)                     def __init__(self):
1321: (8)                         MAxisConcatenator.__init__(self, 0)
1322: (0)             mr_ = mr_class()
1323: (0)             def ndenumerate(a, compressed=True):
1324: (4)                 """
1325: (4)                     Multidimensional index iterator.
1326: (4)                     Return an iterator yielding pairs of array coordinates and values,
1327: (4)                     skipping elements that are masked. With `compressed=False`,
1328: (4)                     `ma.masked` is yielded as the value of masked elements. This
1329: (4)                     behavior differs from that of `numpy.ndenumerate`, which yields the
1330: (4)                     value of the underlying data array.
1331: (4)                     Notes
1332: (4)                     -----
1333: (4)                     .. versionadded:: 1.23.0
1334: (4)                     Parameters
1335: (4)                     -----
1336: (4)                     a : array_like
1337: (8)                         An array with (possibly) masked elements.
1338: (4)                     compressed : bool, optional
1339: (8)                         If True (default), masked elements are skipped.
1340: (4)                     See Also
1341: (4)                     -----
1342: (4)                     numpy.ndenumerate : Equivalent function ignoring any mask.
1343: (4)                     Examples
1344: (4)                     -----
1345: (4)                     >>> a = np.ma.arange(9).reshape((3, 3))

```

```

1346: (4)          >>> a[1, 0] = np.ma.masked
1347: (4)          >>> a[1, 2] = np.ma.masked
1348: (4)          >>> a[2, 1] = np.ma.masked
1349: (4)          >>> a
1350: (4)          masked_array(
1351: (6)            data=[[0, 1, 2],
1352: (12)              [--, 4, --],
1353: (12)              [6, --, 8]],
1354: (6)            mask=[[False, False, False],
1355: (12)              [True, False, True],
1356: (12)              [False, True, False]],
1357: (6)            fill_value=999999)
1358: (4)          >>> for index, x in np.ma.ndenumerate(a):
1359: (4)            ...      print(index, x)
1360: (4)            (0, 0) 0
1361: (4)            (0, 1) 1
1362: (4)            (0, 2) 2
1363: (4)            (1, 1) 4
1364: (4)            (2, 0) 6
1365: (4)            (2, 2) 8
1366: (4)          >>> for index, x in np.ma.ndenumerate(a, compressed=False):
1367: (4)            ...      print(index, x)
1368: (4)            (0, 0) 0
1369: (4)            (0, 1) 1
1370: (4)            (0, 2) 2
1371: (4)            (1, 0) --
1372: (4)            (1, 1) 4
1373: (4)            (1, 2) --
1374: (4)            (2, 0) 6
1375: (4)            (2, 1) --
1376: (4)            (2, 2) 8
1377: (4)          """
1378: (4)          for it, mask in zip(np.ndenumerate(a), getmaskarray(a).flat):
1379: (8)            if not mask:
1380: (12)              yield it
1381: (8)            elif not compressed:
1382: (12)              yield it[0], masked
1383: (0)          def flatnotmasked_edges(a):
1384: (4)            """
1385: (4)              Find the indices of the first and last unmasked values.
1386: (4)              Expects a 1-D `MaskedArray`, returns None if all values are masked.
1387: (4)              Parameters
1388: (4)                -----
1389: (4)                a : array_like
1390: (8)                  Input 1-D `MaskedArray`
1391: (4)              Returns
1392: (4)                -----
1393: (4)                edges : ndarray or None
1394: (8)                  The indices of first and last non-masked value in the array.
1395: (8)                  Returns None if all values are masked.
1396: (4)              See Also
1397: (4)                -----
1398: (4)                flatnotmasked_contiguous, notmasked_contiguous, notmasked_edges
1399: (4)                clump_masked, clump_unmasked
1400: (4)              Notes
1401: (4)                -----
1402: (4)                Only accepts 1-D arrays.
1403: (4)              Examples
1404: (4)                -----
1405: (4)                >>> a = np.ma.arange(10)
1406: (4)                >>> np.ma.flatnotmasked_edges(a)
1407: (4)                array([0, 9])
1408: (4)                >>> mask = (a < 3) | (a > 8) | (a == 5)
1409: (4)                >>> a[mask] = np.ma.masked
1410: (4)                >>> np.array(a[~a.mask])
1411: (4)                array([3, 4, 6, 7, 8])
1412: (4)                >>> np.ma.flatnotmasked_edges(a)
1413: (4)                array([3, 8])
1414: (4)                >>> a[:] = np.ma.masked

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1415: (4)          >>> print(np.ma.flatnotmasked_edges(a))
1416: (4)          None
1417: (4)          """
1418: (4)          m = getmask(a)
1419: (4)          if m is nomask or not np.any(m):
1420: (8)              return np.array([0, a.size - 1])
1421: (4)          unmasked = np.flatnonzero(~m)
1422: (4)          if len(unmasked) > 0:
1423: (8)              return unmasked[[0, -1]]
1424: (4)          else:
1425: (8)              return None
1426: (0)          def notmasked_edges(a, axis=None):
1427: (4)          """
1428: (4)          Find the indices of the first and last unmasked values along an axis.
1429: (4)          If all values are masked, return None. Otherwise, return a list
1430: (4)          of two tuples, corresponding to the indices of the first and last
1431: (4)          unmasked values respectively.
1432: (4)          Parameters
1433: (4)          -----
1434: (4)          a : array_like
1435: (8)              The input array.
1436: (4)          axis : int, optional
1437: (8)              Axis along which to perform the operation.
1438: (8)              If None (default), applies to a flattened version of the array.
1439: (4)          Returns
1440: (4)          -----
1441: (4)          edges : ndarray or list
1442: (8)              An array of start and end indexes if there are any masked data in
1443: (8)              the array. If there are no masked data in the array, `edges` is a
1444: (8)              list of the first and last index.
1445: (4)          See Also
1446: (4)          -----
1447: (4)          flatnotmasked_contiguous, flatnotmasked_edges, notmasked_contiguous
1448: (4)          clump_masked, clump_unmasked
1449: (4)          Examples
1450: (4)          -----
1451: (4)          >>> a = np.arange(9).reshape((3, 3))
1452: (4)          >>> m = np.zeros_like(a)
1453: (4)          >>> m[1:, 1:] = 1
1454: (4)          >>> am = np.ma.array(a, mask=m)
1455: (4)          >>> np.array(am[~am.mask])
1456: (4)          array([0, 1, 2, 3, 6])
1457: (4)          >>> np.ma.notmasked_edges(am)
1458: (4)          array([0, 6])
1459: (4)          """
1460: (4)          a = asarray(a)
1461: (4)          if axis is None or a.ndim == 1:
1462: (8)              return flatnotmasked_edges(a)
1463: (4)          m = getmaskarray(a)
1464: (4)          idx = array(np.indices(a.shape), mask=np.asarray([m] * a.ndim))
1465: (4)          return [tuple([idx[i].min(axis).compressed() for i in range(a.ndim)]),
1466: (12)              tuple([idx[i].max(axis).compressed() for i in range(a.ndim)]), ]
1467: (0)          def flatnotmasked_contiguous(a):
1468: (4)          """
1469: (4)          Find contiguous unmasked data in a masked array.
1470: (4)          Parameters
1471: (4)          -----
1472: (4)          a : array_like
1473: (8)              The input array.
1474: (4)          Returns
1475: (4)          -----
1476: (4)          slice_list : list
1477: (8)              A sorted sequence of `slice` objects (start index, end index).
1478: (8)              .. versionchanged:: 1.15.0
1479: (12)                  Now returns an empty list instead of None for a fully masked array
1480: (4)          See Also
1481: (4)          -----
1482: (4)          flatnotmasked_edges, notmasked_contiguous, notmasked_edges
1483: (4)          clump_masked, clump_unmasked

```

```

1484: (4)          Notes
1485: (4)          -----
1486: (4)          Only accepts 2-D arrays at most.
1487: (4)          Examples
1488: (4)          -----
1489: (4)          >>> a = np.ma.arange(10)
1490: (4)          >>> np.ma.flatnotmasked_contiguous(a)
1491: (4)          [slice(0, 10, None)]
1492: (4)          >>> mask = (a < 3) | (a > 8) | (a == 5)
1493: (4)          >>> a[mask] = np.ma.masked
1494: (4)          >>> np.array(a[~a.mask])
1495: (4)          array([3, 4, 6, 7, 8])
1496: (4)          >>> np.ma.flatnotmasked_contiguous(a)
1497: (4)          [slice(3, 5, None), slice(6, 9, None)]
1498: (4)          >>> a[:] = np.ma.masked
1499: (4)          >>> np.ma.flatnotmasked_contiguous(a)
1500: (4)          []
1501: (4)          """
1502: (4)          m = getmask(a)
1503: (4)          if m is nomask:
1504: (8)              return [slice(0, a.size)]
1505: (4)          i = 0
1506: (4)          result = []
1507: (4)          for (k, g) in itertools.groupby(m.ravel()):
1508: (8)              n = len(list(g))
1509: (8)              if not k:
1510: (12)                  result.append(slice(i, i + n))
1511: (8)              i += n
1512: (4)          return result
1513: (0)          def notmasked_contiguous(a, axis=None):
1514: (4)          """
1515: (4)          Find contiguous unmasked data in a masked array along the given axis.
1516: (4)          Parameters
1517: (4)          -----
1518: (4)          a : array_like
1519: (8)              The input array.
1520: (4)          axis : int, optional
1521: (8)              Axis along which to perform the operation.
1522: (8)              If None (default), applies to a flattened version of the array, and
this
1523: (8)              is the same as `flatnotmasked_contiguous` .
1524: (4)          Returns
1525: (4)          -----
1526: (4)          endpoints : list
1527: (8)              A list of slices (start and end indexes) of unmasked indexes
1528: (8)              in the array.
1529: (8)              If the input is 2d and axis is specified, the result is a list of
lists.
1530: (4)          See Also
1531: (4)          -----
1532: (4)          flatnotmasked_edges, flatnotmasked_contiguous, notmasked_edges
1533: (4)          clump_masked, clump_unmasked
1534: (4)          Notes
1535: (4)          -----
1536: (4)          Only accepts 2-D arrays at most.
1537: (4)          Examples
1538: (4)          -----
1539: (4)          >>> a = np.arange(12).reshape((3, 4))
1540: (4)          >>> mask = np.zeros_like(a)
1541: (4)          >>> mask[1:, :-1] = 1; mask[0, 1] = 1; mask[-1, 0] = 0
1542: (4)          >>> ma = np.ma.array(a, mask=mask)
1543: (4)          >>> ma
1544: (4)          masked_array(
1545: (6)              data=[[0, --, 2, 3],
1546: (12)                  [--, --, --, 7],
1547: (12)                  [8, --, --, 11]],
1548: (6)              mask=[[False, True, False, False],
1549: (12)                  [True, True, True, False],
1550: (12)                  [False, True, True, False]]),

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1551: (6)          fill_value=999999)
1552: (4)          >>> np.array(ma[~ma.mask])
1553: (4)          array([ 0,  2,  3,  7,  8, 11])
1554: (4)          >>> np.ma.notmasked_contiguous(ma)
1555: (4)          [slice(0, 1, None), slice(2, 4, None), slice(7, 9, None), slice(11, 12,
None)]
1556: (4)          >>> np.ma.notmasked_contiguous(ma, axis=0)
1557: (4)          [[slice(0, 1, None), slice(2, 3, None)], [], [slice(0, 1, None)],
[slice(0, 3, None)]]
1558: (4)          >>> np.ma.notmasked_contiguous(ma, axis=1)
1559: (4)          [[slice(0, 1, None), slice(2, 4, None)], [slice(3, 4, None)], [slice(0, 1,
None), slice(3, 4, None)]]
1560: (4)          """
1561: (4)          a = asarray(a)
1562: (4)          nd = a.ndim
1563: (4)          if nd > 2:
1564: (8)              raise NotImplementedError("Currently limited to at most 2D array.")
1565: (4)          if axis is None or nd == 1:
1566: (8)              return flatnotmasked_contiguous(a)
1567: (4)          result = []
1568: (4)          other = (axis + 1) % 2
1569: (4)          idx = [0, 0]
1570: (4)          idx[axis] = slice(None, None)
1571: (4)          for i in range(a.shape[other]):
1572: (8)              idx[other] = i
1573: (8)              result.append(flatnotmasked_contiguous(a[tuple(idx)])))
1574: (4)          return result
1575: (0)          def _ezclump(mask):
1576: (4)          """
1577: (4)          Finds the clumps (groups of data with the same values) for a 1D bool
array.
1578: (4)          Returns a series of slices.
1579: (4)          """
1580: (4)          if mask.ndim > 1:
1581: (8)              mask = mask.ravel()
1582: (4)          idx = (mask[1:] ^ mask[:-1]).nonzero()
1583: (4)          idx = idx[0] + 1
1584: (4)          if mask[0]:
1585: (8)              if len(idx) == 0:
1586: (12)                  return [slice(0, mask.size)]
1587: (8)              r = [slice(0, idx[0])]
1588: (8)              r.extend((slice(left, right)
1589: (18)                  for left, right in zip(idx[1:-1:2], idx[2::2])))
1590: (4)          else:
1591: (8)              if len(idx) == 0:
1592: (12)                  return []
1593: (8)              r = [slice(left, right) for left, right in zip(idx[:-1:2], idx[1::2])]
1594: (4)          if mask[-1]:
1595: (8)              r.append(slice(idx[-1], mask.size))
1596: (4)          return r
1597: (0)          def clump_unmasked(a):
1598: (4)          """
1599: (4)          Return list of slices corresponding to the unmasked clumps of a 1-D array.
1600: (4)          (A "clump" is defined as a contiguous region of the array).
1601: (4)          Parameters
1602: (4)          -----
1603: (4)          a : ndarray
1604: (8)              A one-dimensional masked array.
1605: (4)          Returns
1606: (4)          -----
1607: (4)          slices : list of slice
1608: (8)              The list of slices, one for each continuous region of unmasked
elements in `a`.
1609: (8)
1610: (4)          Notes
1611: (4)          -----
1612: (4)          .. versionadded:: 1.4.0
1613: (4)          See Also
1614: (4)          -----
1615: (4)          flatnotmasked_edges, flatnotmasked_contiguous, notmasked_edges

```

```

1616: (4)          notmasked_contiguous, clump_masked
1617: (4)          Examples
1618: (4)
1619: (4)          -----
1620: (4)          >>> a = np.ma.masked_array(np.arange(10))
1621: (4)          >>> a[[0, 1, 2, 6, 8, 9]] = np.ma.masked
1622: (4)          >>> np.ma.clump_unmasked(a)
1623: (4)          [slice(3, 6, None), slice(7, 8, None)]
1624: (4)          """
1625: (4)          mask = getattr(a, '_mask', nomask)
1626: (8)          if mask is nomask:
1627: (4)              return [slice(0, a.size)]
1628: (0)          return _ezclump(~mask)
def clump_masked(a):
    """
1630: (4)          Returns a list of slices corresponding to the masked clumps of a 1-D
array.
1631: (4)          (A "clump" is defined as a contiguous region of the array).
1632: (4)          Parameters
1633: (4)          -----
1634: (4)          a : ndarray
1635: (8)          A one-dimensional masked array.
1636: (4)          Returns
1637: (4)          -----
1638: (4)          slices : list of slice
1639: (8)          The list of slices, one for each continuous region of masked elements
1640: (8)          in `a`.
1641: (4)          Notes
1642: (4)          -----
1643: (4)          .. versionadded:: 1.4.0
1644: (4)          See Also
1645: (4)          -----
1646: (4)          flatnotmasked_edges, flatnotmasked_contiguous, notmasked_edges
1647: (4)          notmasked_contiguous, clump_unmasked
1648: (4)          Examples
1649: (4)          -----
1650: (4)          >>> a = np.ma.masked_array(np.arange(10))
1651: (4)          >>> a[[0, 1, 2, 6, 8, 9]] = np.ma.masked
1652: (4)          >>> np.ma.clump_masked(a)
1653: (4)          [slice(0, 3, None), slice(6, 7, None), slice(8, 10, None)]
1654: (4)          """
1655: (4)          mask = ma.getmask(a)
1656: (4)          if mask is nomask:
1657: (8)              return []
1658: (4)          return _ezclump(mask)
def vander(x, n=None):
    """
1661: (4)          Masked values in the input array result in rows of zeros.
1662: (4)          """
1663: (4)          _vander = np.vander(x, n)
1664: (4)          m = getmask(x)
1665: (4)          if m is not nomask:
1666: (8)              _vander[m] = 0
1667: (4)          return _vander
1668: (0)          vander.__doc__ = ma.doc_note(np.vander.__doc__, vander.__doc__)
1669: (0)          def polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False):
1670: (4)          """
1671: (4)          Any masked values in x is propagated in y, and vice-versa.
1672: (4)          """
1673: (4)          x = asarray(x)
1674: (4)          y = asarray(y)
1675: (4)          m = getmask(x)
1676: (4)          if y.ndim == 1:
1677: (8)              m = mask_or(m, getmask(y))
1678: (4)          elif y.ndim == 2:
1679: (8)              my = getmask(mask_rows(y))
1680: (8)              if my is not nomask:
1681: (12)                  m = mask_or(m, my[:, 0])
1682: (4)          else:
1683: (8)              raise TypeError("Expected a 1D or 2D array for y!")

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1684: (4)         if w is not None:
1685: (8)             w = asarray(w)
1686: (8)             if w.ndim != 1:
1687: (12)                 raise TypeError("expected a 1-d array for weights")
1688: (8)             if w.shape[0] != y.shape[0]:
1689: (12)                 raise TypeError("expected w and y to have the same length")
1690: (8)             m = mask_or(m, getmask(w))
1691: (4)             if m is not nomask:
1692: (8)                 not_m = ~m
1693: (8)                 if w is not None:
1694: (12)                     w = w[not_m]
1695: (8)                     return np.polyfit(x[not_m], y[not_m], deg, rcond, full, w, cov)
1696: (4)             else:
1697: (8)                 return np.polyfit(x, y, deg, rcond, full, w, cov)
1698: (0)         polyfit.__doc__ = ma.doc_note(np.polyfit.__doc__, polyfit.__doc__)
-----
```

File 263 - mrecords.py:

```

1: (0)         """:mod:`numpy.ma.mrecords`  

2: (0)         Defines the equivalent of :class:`numpy.recarrays` for masked arrays,  

3: (0)         where fields can be accessed as attributes.  

4: (0)         Note that :class:`numpy.ma.MaskedArray` already supports structured datatypes  

5: (0)         and the masking of individual fields.  

6: (0)         .. moduleauthor:: Pierre Gerard-Marchant  

7: (0)  

8: (0)         from numpy.ma import (  

9: (4)             MAError, MaskedArray, masked, nomask, masked_array, getdata,  

10: (4)                 getmaskarray, filled  

11: (0)         )  

12: (0)         import numpy.ma as ma  

13: (0)         import warnings  

14: (0)         import numpy as np  

15: (0)         from numpy import (  

16: (4)             bool_, dtype, ndarray, recarray, array as narray  

17: (0)         )  

18: (0)         from numpy.core.records import (  

19: (4)             fromarrays as recfromarrays, fromrecords as recfromrecords  

20: (0)         )  

21: (0)         _byteorderconv = np.core.records._byteorderconv  

22: (0)         _check_fill_value = ma.core._check_fill_value  

23: (0)         __all__ = [  

24: (4)             'MaskedRecords', 'mrecarray', 'fromarrays', 'fromrecords',  

25: (4)                 'fromtextfile', 'addfield',  

26: (0)         ]  

27: (0)         reserved_fields = ['_data', '_mask', '_fieldmask', 'dtype']  

28: (0)         def __checknames(descr, names=None):  

29: (4)             """  

30: (4)             Checks that field names ``descr`` are not reserved keywords.  

31: (4)             If this is the case, a default 'f%i' is substituted. If the argument  

32: (4)             `names` is not None, updates the field names to valid names.  

33: (4)  

34: (4)             ndescr = len(descr)  

35: (4)             default_names = ['f%i' % i for i in range(ndescr)]  

36: (4)             if names is None:  

37: (8)                 new_names = default_names  

38: (4)             else:  

39: (8)                 if isinstance(names, (tuple, list)):  

40: (12)                     new_names = names  

41: (8)                 elif isinstance(names, str):  

42: (12)                     new_names = names.split(',')  

43: (8)  

44: (12)                     raise NameError(f'illegal input names {names!r}')  

45: (8)                     nnames = len(new_names)  

46: (8)                     if nnames < ndescr:  

47: (12)                         new_names += default_names[nnames:]  

48: (4)                     ndescr = []  

49: (4)                     for (n, d, t) in zip(new_names, default_names, descr.descr):
```

```

50: (8)             if n in reserved_fields:
51: (12)             if t[0] in reserved_fields:
52: (16)                 ndescr.append((d, t[1]))
53: (12)             else:
54: (16)                 ndescr.append(t)
55: (8)             else:
56: (12)                 ndescr.append((n, t[1]))
57: (4)         return np.dtype(ndescr)
58: (0)     def __get_fieldmask(self):
59: (4)         mdescr = [(n, '|b1') for n in self.dtype.names]
60: (4)         fdmask = np.empty(self.shape, dtype=mdescr)
61: (4)         fdmask.flat = tuple([False] * len(mdescr))
62: (4)         return fdmask
63: (0)     class MaskedRecords(MaskedArray):
64: (4)         """
65: (4)             Attributes
66: (4)             -----
67: (4)             __data : recarray
68: (8)                 Underlying data, as a record array.
69: (4)             __mask : boolean array
70: (8)                 Mask of the records. A record is masked when all its fields are
71: (8)                 masked.
72: (4)             __fieldmask : boolean recarray
73: (8)                 Record array of booleans, setting the mask of each individual field
74: (8)                 of each record.
75: (4)             __fill_value : record
76: (8)                 Filling values for each field.
77: (4)         """
78: (4)     def __new__(cls, shape, dtype=None, buf=None, offset=0, strides=None,
79: (16)                 formats=None, names=None, titles=None,
80: (16)                 byteorder=None, aligned=False,
81: (16)                 mask=nomask, hard_mask=False, fill_value=None, keep_mask=True,
82: (16)                 copy=False,
83: (16)                 **options):
84: (8)         self = recarray.__new__(cls, shape, dtype=dtype, buf=buf,
offset=offset,
85: (32)                     strides=strides, formats=formats, names=names,
86: (32)                     titles=titles, byteorder=byteorder,
87: (32)                     aligned=aligned,)
88: (8)         mdtype = ma.make_mask_descr(self.dtype)
89: (8)         if mask is nomask or not np.size(mask):
90: (12)             if not keep_mask:
91: (16)                 self.__mask = tuple([False] * len(mdtype))
92: (8)             else:
93: (12)                 mask = np.array(mask, copy=copy)
94: (12)                 if mask.shape != self.shape:
95: (16)                     (nd, nm) = (self.size, mask.size)
96: (16)                     if nm == 1:
97: (20)                         mask = np.resize(mask, self.shape)
98: (16)                     elif nm == nd:
99: (20)                         mask = np.reshape(mask, self.shape)
100: (16)                     else:
101: (20)                         msg = "Mask and data not compatible: data size is %i, " +
\\
102: (26)                             "mask size is %i."
103: (20)                         raise MAError(msg % (nd, nm))
104: (12)                         if not keep_mask:
105: (16)                             self.__setmask__(mask)
106: (16)                             self.__sharedmask = True
107: (12)                         else:
108: (16)                             if mask.dtype == mdtype:
109: (20)                                 __mask = mask
110: (16)                             else:
111: (20)                                 __mask = np.array([tuple([m] * len(mdtype)) for m in mask],
112: (37)                                     dtype=mdtype)
113: (16)                                 self.__mask = __mask
114: (8)             return self
115: (4)     def __array_finalize__(self, obj):
116: (8)         __mask = getattr(obj, '__mask', None)

```

```

117: (8)           if _mask is None:
118: (12)             objmask = getattr(obj, '_mask', nomask)
119: (12)             _dtype = ndarray.__getattribute__(self, 'dtype')
120: (12)             if objmask is nomask:
121: (16)               _mask = ma.make_mask_none(self.shape, dtype=_dtype)
122: (12)             else:
123: (16)               mdescr = ma.make_mask_descr(_dtype)
124: (16)               _mask = narray([tuple([m] * len(mdescr)) for m in objmask],
125: (31)                           dtype=mdescr).view(recarray)
126: (8)             _dict = self.__dict__
127: (8)             _dict.update(_mask=_mask)
128: (8)             self._update_from(obj)
129: (8)             if _dict['_baseclass'] == ndarray:
130: (12)               _dict['_baseclass'] = recarray
131: (8)             return
132: (4)           @property
133: (4)           def __data__(self):
134: (8)             """
135: (8)               Returns the data as a recarray.
136: (8)             """
137: (8)             return ndarray.view(self, recarray)
138: (4)           @property
139: (4)           def __fieldmask__(self):
140: (8)             """
141: (8)               Alias to mask.
142: (8)             """
143: (8)             return self._mask
144: (4)           def __len__(self):
145: (8)             """
146: (8)               Returns the length
147: (8)             """
148: (8)             if self.ndim:
149: (12)               return len(self._data)
150: (8)             return len(self.dtype)
151: (4)           def __getattribute__(self, attr):
152: (8)             try:
153: (12)               return object.__getattribute__(self, attr)
154: (8)             except AttributeError:
155: (12)               pass
156: (8)             fielddict = ndarray.__getattribute__(self, 'dtype').fields
157: (8)             try:
158: (12)               res = fielddict[attr][:2]
159: (8)             except (TypeError, KeyError) as e:
160: (12)               raise AttributeError(
161: (16)                 f'record array has no attribute {attr}') from e
162: (8)             _localdict = ndarray.__getattribute__(self, '__dict__')
163: (8)             _data = ndarray.view(self, _localdict['_baseclass'])
164: (8)             obj = _data.getfield(*res)
165: (8)             if obj.dtype.names is not None:
166: (12)               raise NotImplementedError("MaskedRecords is currently limited to"
167: (38)                               "simple records.")
168: (8)             hasmasked = False
169: (8)             _mask = _localdict.get('_mask', None)
170: (8)             if _mask is not None:
171: (12)               try:
172: (16)                 _mask = _mask[attr]
173: (12)               except IndexError:
174: (16)                 pass
175: (12)               tp_len = len(_mask.dtype)
176: (12)               hasmasked = _mask.view((bool, ((tp_len,) if tp_len else
177: (12)                 ()))).any()
178: (8)             if (obj.shape or hasmasked):
179: (12)               obj = obj.view(MaskedArray)
180: (12)               obj._baseclass = ndarray
181: (12)               obj._isfield = True
182: (12)               obj._mask = _mask
183: (12)               _fill_value = _localdict.get('_fill_value', None)
184: (16)               if _fill_value is not None:

```

```

185: (20)           obj._fill_value = _fill_value[attr]
186: (16)           except ValueError:
187: (20)             obj._fill_value = None
188: (8)
189: (12)           else:
190: (8)             obj = obj.item()
191: (4)             return obj
192: (8)           def __setattr__(self, attr, val):
193: (8)             """
194: (8)               Sets the attribute attr to the value val.
195: (8)             """
196: (12)             if attr in ['mask', 'fieldmask']:
197: (12)               self.__setmask__(val)
198: (8)               return
199: (8)             _localdict = object.__getattribute__(self, '__dict__')
200: (8)             newattr = attr not in _localdict
201: (12)             try:
202: (8)               ret = object.__setattr__(self, attr, val)
203: (12)             except Exception:
204: (12)               fielddict = ndarray.__getattribute__(self, 'dtype').fields or {}
205: (12)               optinfo = ndarray.__getattribute__(self, '_optinfo') or {}
206: (16)               if not (attr in fielddict or attr in optinfo):
207: (8)                 raise
208: (12)             else:
209: (12)               fielddict = ndarray.__getattribute__(self, 'dtype').fields or {}
210: (16)               if attr not in fielddict:
211: (12)                 return ret
212: (16)               if newattr:
213: (20)                 try:
214: (16)                   object.__delattr__(self, attr)
215: (20)                 except Exception:
216: (8)                   return ret
217: (12)               try:
218: (8)                 res = fielddict[attr][:2]
219: (12)               except (TypeError, KeyError) as e:
220: (16)                 raise AttributeError(
221: (8)                   f'record array has no attribute {attr}') from e
222: (12)               if val is masked:
223: (12)                 _fill_value = _localdict['_fill_value']
224: (16)                 if _fill_value is not None:
225: (12)                   dval = _localdict['_fill_value'][attr]
226: (16)                 else:
227: (12)                   dval = val
228: (8)                   mval = True
229: (12)                 else:
230: (12)                   dval = filled(val)
231: (8)                   mval = getmaskarray(val)
232: (8)                 obj = ndarray.__getattribute__(self, '_data').setfield(dval, *res)
233: (8)                 _localdict['_mask'].__setitem__(attr, mval)
234: (4)               return obj
235: (8)             def __getitem__(self, indx):
236: (8)               """
237: (8)                 Returns all the fields sharing the same fieldname base.
238: (8)                 The fieldname base is either `'_data` or `'_mask`.
239: (8)               """
240: (8)               _localdict = self.__dict__
241: (8)               _mask = ndarray.__getattribute__(self, '_mask')
242: (8)               _data = ndarray.view(self, _localdict['_baseclass'])
243: (12)               if isinstance(indx, str):
244: (12)                 obj = _data[indx].view(MaskedArray)
245: (12)                 obj._mask = _mask[indx]
246: (12)                 obj._sharedmask = True
247: (12)                 fval = _localdict['_fill_value']
248: (16)                 if fval is not None:
249: (12)                   obj._fill_value = fval[indx]
250: (16)                 if not obj.ndim and obj._mask:
251: (12)                   return masked
252: (8)                   return obj
253: (8)               obj = np.array(_data[indx], copy=False).view(mrecarray)
254: (8)               obj._mask = np.array(_mask[indx], copy=False).view(recarray)

```

```

254: (8)             return obj
255: (4)             def __setitem__(self, indx, value):
256: (8)                 """
257: (8)                     Sets the given record to value.
258: (8)                     """
259: (8)                     MaskedArray.__setitem__(self, indx, value)
260: (8)                     if isinstance(indx, str):
261: (12)                         self._mask[indx] = ma.getmaskarray(value)
262: (4)             def __str__(self):
263: (8)                 """
264: (8)                     Calculates the string representation.
265: (8)                     """
266: (8)                     if self.size > 1:
267: (12)                         mstr = [f"{{', '.join([str(i) for i in s])}}"
268: (20)                             for s in zip(*[getattr(self, f) for f in
269: (12)                               self.dtype.names])]
270: (8)
271: (12)
272: (20)
273: (8)             self.dtype.names)])
274: (12)             return f"{{', '.join(mstr)}}"
275: (4)             else:
276: (8)                 mstr = [f"{{', '.join([str(i) for i in s])}}"
277: (20)                     for s in zip([getattr(self, f) for f in
278: (12)                       self.dtype.names])]
279: (8)             return f"{{', '.join(mstr)}}"
280: (4)             def __repr__(self):
281: (8)                 """
282: (8)                     Calculates the repr representation.
283: (8)                     """
284: (8)                     _names = self.dtype.names
285: (16)                     fmt = "%%%is : %%s" % (max([len(n) for n in _names]) + 4,)
286: (12)                     reprstr = [fmt % (f, getattr(self, f)) for f in self.dtype.names]
287: (8)                     reprstr.insert(0, 'masked_records(')
288: (8)                     reprstr.extend([fmt % ('    fill_value', self.fill_value),
289: (24)                           ',')])
290: (8)                     return str("\n".join(reprstr))
291: (4)             def view(self, dtype=None, type=None):
292: (8)                 """
293: (8)                     Returns a view of the mrecarray.
294: (8)                     """
295: (12)                     if dtype is None:
296: (16)                         if type is None:
297: (20)                             output = ndarray.view(self)
298: (16)                         else:
299: (20)                             output = ndarray.view(self, type)
300: (12)                     elif type is None:
301: (16)                         try:
302: (20)                             if issubclass(dtype, ndarray):
303: (16)                                 output = ndarray.view(self, dtype)
304: (20)                             else:
305: (20)                                 output = ndarray.view(self, dtype)
306: (16)                     except TypeError:
307: (20)                         dtype = np.dtype(dtype)
308: (16)                         if dtype.fields is None:
309: (20)                             basetype = self.__class__.__bases__[0]
310: (20)                             output = self.__array__().view(dtype, basetype)
311: (16)                             output._update_from(self)
312: (16)                         else:
313: (20)                             output = ndarray.view(self, dtype)
314: (16)                             output._fill_value = None
315: (8)                     else:
316: (12)                         output = ndarray.view(self, dtype, type)
317: (8)                     if (getattr(output, '_mask', nomask) is not nomask):
318: (12)                         mdtype = ma.make_mask_descr(output.dtype)
319: (12)                         output._mask = self._mask.view(mdtype, ndarray)
320: (8)                         output._mask.shape = output.shape
321: (8)                     return output
322: (4)             def harden_mask(self):
323: (8)                 """
324: (8)                     Forces the mask to hard.
325: (8)                     """
326: (8)                     self._hardmask = True

```

```

321: (4)             def soften_mask(self):
322: (8)                 """
323: (8)                     Forces the mask to soft
324: (8)                     """
325: (8)                     self._hardmask = False
326: (4)             def copy(self):
327: (8)                 """
328: (8)                     Returns a copy of the masked record.
329: (8)                     """
330: (8)                     copied = self._data.copy().view(type(self))
331: (8)                     copied._mask = self._mask.copy()
332: (8)                     return copied
333: (4)             def tolist(self, fill_value=None):
334: (8)                 """
335: (8)                     Return the data portion of the array as a list.
336: (8)                     Data items are converted to the nearest compatible Python type.
337: (8)                     Masked values are converted to fill_value. If fill_value is None,
338: (8)                     the corresponding entries in the output list will be ``None``.
339: (8)                     """
340: (8)                     if fill_value is not None:
341: (12)                         return self.filled(fill_value).tolist()
342: (8)                     result = narray(self.filled().tolist(), dtype=object)
343: (8)                     mask = narray(self._mask.tolist())
344: (8)                     result[mask] = None
345: (8)                     return result.tolist()
346: (4)             def __getstate__(self):
347: (8)                 """Return the internal state of the masked array.
348: (8)                 This is for pickling.
349: (8)                 """
350: (8)                 state = (1,
351: (17)                     self.shape,
352: (17)                     self.dtype,
353: (17)                     self.flags.fnc,
354: (17)                     self._data.tobytes(),
355: (17)                     self._mask.tobytes(),
356: (17)                     self._fill_value,
357: (17)                     )
358: (8)                 return state
359: (4)             def __setstate__(self, state):
360: (8)                 """
361: (8)                     Restore the internal state of the masked array.
362: (8)                     This is for pickling. ``state`` is typically the output of the
363: (8)                     ``__getstate__`` output, and is a 5-tuple:
364: (8)                     - class name
365: (8)                     - a tuple giving the shape of the data
366: (8)                     - a typecode for the data
367: (8)                     - a binary string for the data
368: (8)                     - a binary string for the mask.
369: (8)                     """
370: (8)                     (ver, shp, typ, isf, raw, msk, flv) = state
371: (8)                     ndarray.__setstate__(self, (shp, typ, isf, raw))
372: (8)                     mdtype = dtype([(k, bool_) for (k, _) in self.dtype.descr])
373: (8)                     self.__dict__['_mask'].__setstate__((shp, mdtype, isf, msk))
374: (8)                     self.fill_value = flv
375: (4)             def __reduce__(self):
376: (8)                 """
377: (8)                     Return a 3-tuple for pickling a MaskedArray.
378: (8)                     """
379: (8)                     return (_mrreconstruct,
380: (16)                         (self.__class__, self._baseclass, (0,), 'b',),
381: (16)                         self.__getstate__())
382: (0)             def _mrreconstruct(subtype, baseclass, baseshape, basetype,):
383: (4)                 """
384: (4)                     Build a new MaskedArray from the information stored in a pickle.
385: (4)                     """
386: (4)                     _data = ndarray.__new__(baseclass, baseshape, basetype).view(subtype)
387: (4)                     _mask = ndarray.__new__(ndarray, baseshape, 'b1')
388: (4)                     return subtype.__new__(subtype, _data, mask=_mask, dtype=basetype,)
389: (0)             mrecarray = MaskedRecords

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

390: (0)
391: (15)
392: (15)
393: (4)
394: (4)
395: (4)
396: (4)
397: (4)
398: (8)
converted
399: (8)
is
400: (8)
401: (4)
402: (8)
403: (4)
404: (8)
405: (8)
406: (4)
407: (8)
will
408: (8)
dtype
409: (8)
410: (4)
411: (8)
412: (4)
413: (8)
414: (4)
415: (4)
416: (4)
processing.
417: (4)
418: (4)
419: (4)
420: (4)
421: (27)
422: (27)
423: (27)
424: (4)
425: (4)
426: (8)
427: (4)
428: (0)
429: (16)
430: (16)
431: (4)
432: (4)
433: (4)
434: (4)
435: (4)
436: (8)
437: (8)
is
438: (8)
439: (4)
440: (8)
441: (4)
442: (8)
443: (8)
444: (4)
445: (8)
will
446: (8)
dtype
447: (8)
448: (4)
449: (8)
450: (4)

def fromarrays(arraylist, dtype=None, shape=None, formats=None,
               names=None, titles=None, aligned=False, byteorder=None,
               fill_value=None):
    """
    Creates a mrecarray from a (flat) list of masked arrays.

    Parameters
    -----
    arraylist : sequence
        A list of (masked) arrays. Each element of the sequence is first
        converted to a masked array if needed. If a 2D array is passed as argument, it
        will be processed line by line.
    dtype : {None, dtype}, optional
        Data type descriptor.
    shape : {None, integer}, optional
        Number of records. If None, shape is defined from the shape of the
        first array in the list.
    formats : {None, sequence}, optional
        Sequence of formats for each individual field. If None, the formats
        will be autodetected by inspecting the fields and selecting the highest
        possible.
    names : {None, sequence}, optional
        Sequence of the names of each field.
    fill_value : {None, sequence}, optional
        Sequence of data to be used as filling values.

    Notes
    -----
    Lists of tuples should be preferred over lists of lists for faster
    processing.

    """
    datalist = [getdata(x) for x in arraylist]
    masklist = [np.atleast_1d(getmaskarray(x)) for x in arraylist]
    _array = recfromarrays(datalist,
                           dtype=dtype, shape=shape, formats=formats,
                           names=names, titles=titles, aligned=aligned,
                           byteorder=byteorder).view(mrecarray)
    _array._mask.flat = list(zip(*masklist))
    if fill_value is not None:
        _array.fill_value = fill_value
    return _array

def fromrecords(reclist, dtype=None, shape=None, formats=None, names=None,
                titles=None, aligned=False, byteorder=None,
                fill_value=None, mask=nomask):
    """
    Creates a MaskedRecords from a list of records.

    Parameters
    -----
    reclist : sequence
        A list of records. Each element of the sequence is first converted
        to a masked array if needed. If a 2D array is passed as argument, it
        will be processed line by line.
    dtype : {None, dtype}, optional
        Data type descriptor.
    shape : {None,int}, optional
        Number of records. If None, ``shape`` is defined from the shape of the
        first array in the list.
    formats : {None, sequence}, optional
        Sequence of formats for each individual field. If None, the formats
        will be autodetected by inspecting the fields and selecting the highest
        possible.
    names : {None, sequence}, optional
        Sequence of the names of each field.
    fill_value : {None, sequence}, optional

```

```

451: (8)                               Sequence of data to be used as filling values.
452: (4)                               mask : {nomask, sequence}, optional.
453: (8)                               External mask to apply on the data.
454: (4)                               Notes
455: (4)
456: (4)                               Lists of tuples should be preferred over lists of lists for faster
processing.
457: (4)
458: (4)                               """
459: (4)                               _mask = getattr(reclist, '_mask', None)
460: (8)                               if isinstance(reclist, ndarray):
461: (12)                                 if isinstance(reclist, MaskedArray):
462: (8)                                   reclist = reclist.filled().view(ndarray)
463: (12)                                 if dtype is None:
464: (8)                                   dtype = reclist.dtype
465: (4)                                   reclist = reclist.tolist()
466: (26)                                 mrec = recfromrecords(reclist, dtype=dtype, shape=shape, formats=formats,
467: (26)                                         names=names, titles=titles,
468:                                         aligned=aligned,
469:                                         byteorder=byteorder).view(mrecarray)
470: (4)                               if fill_value is not None:
471: (8)                                 mrec.fill_value = fill_value
472: (4)                               if mask is not nomask:
473: (8)                                 mask = np.array(mask, copy=False)
474: (8)                                 maskrecordlength = len(mask.dtype)
475: (8)                                 if maskrecordlength:
476: (12)                                   mrec._mask.flat = mask
477: (8)                                 elif mask.ndim == 2:
478: (12)                                   mrec._mask.flat = [tuple(m) for m in mask]
479: (8)                                 else:
480: (12)                                   mrec.__setmask__(mask)
481: (4)                               if _mask is not None:
482: (8)                                 mrec._mask[:] = _mask
483: (4)                               return mrec
484: (0)                               def _guessvartypes(arr):
485: (4)                               """
486: (4)                               Tries to guess the dtypes of the str_ndarray `arr`.
487: (4)                               Guesses by testing element-wise conversion. Returns a list of dtypes.
488: (4)                               The array is first converted to ndarray. If the array is 2D, the test
489: (4)                               is performed on the first line. An exception is raised if the file is
490: (4)                               3D or more.
491: (4)                               """
492: (4)                               vartypes = []
493: (8)                               arr = np.asarray(arr)
494: (4)                               if arr.ndim == 2:
495: (8)                                 arr = arr[0]
496: (4)                               elif arr.ndim > 2:
497: (8)                                 raise ValueError("The array should be 2D at most!")
498: (12)                               for f in arr:
499: (8)                                 try:
500: (12)                                   int(f)
501: (16)                                 except (ValueError, TypeError):
502: (12)                                   try:
503: (16)                                     float(f)
504: (20)                                 except (ValueError, TypeError):
505: (16)                                     vartypes.append(arr.dtype)
506: (20)                                 else:
507: (16)                                     vartypes.append(np.dtype(complex))
508: (20)                                 else:
509: (12)                                     vartypes.append(np.dtype(float))
510: (16)                                 else:
511: (8)                                     vartypes.append(np.dtype(int))
512: (12)                                 return vartypes
513: (4)                               def openfile(fname):
514: (0)                               """
515: (4)                               Opens the file handle of file `fname`.
516: (4)                               """
517: (4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

518: (4)
519: (8)
520: (4)
521: (8)
522: (4)
523: (8)
524: (4)
525: (8)
526: (8)
527: (4)
528: (4)
529: (0)
530: (17)
531: (17)
532: (4)
533: (4)
534: (4)
535: (4)
536: (4)
537: (8)
538: (4)
539: (8)
540: (8)
541: (4)
542: (8)
543: (4)
544: (8)
545: (4)
546: (8)
547: (8)
548: (4)
549: (8)
550: (8)
551: (4)
552: (4)
553: (8)
554: (12)
555: (28)
556: (8)
557: (22)
558: (22)
559: (22)
560: (8)
561: (4)
562: (4)
563: (8)
564: (8)
565: (8)
566: (8)
567: (12)
568: (4)
569: (8)
570: (4)
571: (31)
572: (4)
573: (4)
574: (4)
575: (8)
576: (4)
577: (8)
578: (8)
579: (12)
580: (12)
581: (12)
582: (12)
583: (4)
584: (4)
585: (4)
586: (4)

    if hasattr(fname, 'readline'):
        return fname
    try:
        f = open(fname)
    except FileNotFoundError as e:
        raise FileNotFoundError(f"No such file: '{fname}'") from e
    if f.readline()[:2] != "\x00\x00":
        f.seek(0, 0)
    return f
    f.close()
    raise NotImplementedError("Wow, binary file")
def fromtextfile(fname, delimiter=None, commentchar='#', missingchar=' ', varnames=None, vartypes=None, *, delimiter=np._NoValue): # backwards compatibility
"""
Creates a mrecarray from data stored in the file `filename`.
Parameters
-----
fname : {file name/handle}
    Handle of an opened file.
delimiter : {None, string}, optional
    Alphanumeric character used to separate columns in the file.
    If None, any (group of) white spacestring(s) will be used.
commentchar : {'#', string}, optional
    Alphanumeric character used to mark the start of a comment.
missingchar : {' ', string}, optional
    String indicating missing data, and used to create the masks.
varnames : {None, sequence}, optional
    Sequence of the variable names. If None, a list will be created from
    the first non empty line of the file.
vartypes : {None, sequence}, optional
    Sequence of the variables dtypes. If None, it will be estimated from
    the first non-commented line.
Ultra simple: the varnames are in the header, one line"""
if delimiter is not np._NoValue:
    if delimiter is not None:
        raise TypeError("fromtextfile() got multiple values for argument "
                       "'delimiter'")
    warnings.warn("The 'delimiter' keyword argument of "
                  "numpy.ma.mrecords.fromtextfile() is deprecated "
                  "since NumPy 1.22.0, use 'delimiter' instead.",
                  DeprecationWarning, stacklevel=2)
    delimiter = delimiter
ftext = openfile(fname)
while True:
    line = ftext.readline()
    firstline = line[:line.find(commentchar)].strip()
    _varnames = firstline.split(delimiter)
    if len(_varnames) > 1:
        break
if varnames is None:
    varnames = _varnames
_variables = masked_array([line.strip().split(delimiter) for line in ftext
                           if line[0] != commentchar and len(line) > 1])
(_, nfields) = _variables.shape
ftext.close()
if vartypes is None:
    vartypes = _guessvartypes(_variables[0])
else:
    vartypes = [np.dtype(v) for v in vartypes]
    if len(vartypes) != nfields:
        msg = "Attempting to %i dtypes for %i fields!" % (len(vartypes), nfields)
        msg += " Reverting to default."
        warnings.warn(msg % (len(vartypes), nfields), stacklevel=2)
        vartypes = _guessvartypes(_variables[0])
mdescr = [(n, f) for (n, f) in zip(varnames, vartypes)]
mfillv = [ma.default_fill_value(f) for f in vartypes]
_mask = (_variables.T == missingchar)
_datalist = [masked_array(a, mask=_mask, dtype=t, fill_value=f)
            for (t, a) in zip(vartypes, _variables.T)]

```

```
587: (17)                         for (a, m, t, f) in zip(_variables.T, _mask, vartypes,
mfillv)]
588: (4)                     return fromarrays(_datalist, dtype=mdescr)
589: (0)     def addfield(mrecord, newfield, newfieldname=None):
590: (4)         """Adds a new field to the masked record array
591: (4)         Uses `newfield` as data and `newfieldname` as name. If `newfieldname`
592: (4)         is None, the new field name is set to 'fi', where `i` is the number of
593: (4)         existing fields.
594: (4)         """
595: (4)         _data = mrecord._data
596: (4)         _mask = mrecord._mask
597: (4)         if newfieldname is None or newfieldname in reserved_fields:
598: (8)             newfieldname = 'f%i' % len(_data.dtype)
599: (4)         newfield = ma.array(newfield)
600: (4)         newdtype = np.dtype(_data.dtype.descr + [(newfieldname, newfield.dtype)])
601: (4)         newdata = recarray(_data.shape, newdtype)
602: (4)         [newdata.setfield(_data.getfield(*f), *f)
603: (5)             for f in _data.dtype.fields.values()]
604: (4)         newdata.setfield(newfield._data, *newdata.dtype.fields[newfieldname])
605: (4)         newdata = newdata.view(MaskedRecords)
606: (4)         newmdtype = np.dtype([(n, bool_) for n in newdtype.names])
607: (4)         newmask = recarray(_data.shape, newmdtype)
608: (4)         [newmask.setfield(_mask.getfield(*f), *f)
609: (5)             for f in _mask.dtype.fields.values()]
610: (4)         newmask.setfield(getmaskarray(newfield),
611: (21)                             *newmask.dtype.fields[newfieldname])
612: (4)         newdata._mask = newmask
613: (4)         return newdata
```

## File 264 - setup.py:

```
1: (0)         def configuration(parent_package='',top_path=None):
2: (4)             from numpy.distutils.misc_util import Configuration
3: (4)             config = Configuration('ma', parent_package, top_path)
4: (4)             config.add_subpackage('tests')
5: (4)             config.add_data_files('*.*pyi')
6: (4)             return config
7: (0)         if __name__ == "__main__":
8: (4)             from numpy.distutils.core import setup
9: (4)             config = configuration(top_path='').todict()
10: (4)            setup(**config)
```

## File 265 - testutils.py:

```
1: (0)     """Miscellaneous functions for testing masked arrays and subclasses
2: (0)     :author: Pierre Gerard-Marchant
3: (0)     :contact: pierregm_at_uga_dot_edu
4: (0)     :version: $Id: testutils.py 3529 2007-11-13 08:01:14Z jarrod.millman $
5: (0)
6: (0)     import operator
7: (0)     import numpy as np
8: (0)     from numpy import ndarray, float_
9: (0)     import numpy.core.umath as umath
10: (0)    import numpy.testing
11: (0)    from numpy.testing import (
12: (4)        assert_, assert_allclose, assert_array_almost_equal_nulp,
13: (4)        assert_raises, build_err_msg
14: (4)    )
15: (0)    from .core import mask_or, getmask, masked_array, nomask, masked, filled
16: (0)    __all__masked = [
17: (4)        'almost', 'approx', 'assert_almost_equal', 'assert_array_almost_equal',
18: (4)        'assert_array_approx_equal', 'assert_array_compare',
19: (4)        'assert_array_equal', 'assert_array_less', 'assert_close',
20: (4)        'assert_equal', 'assert_equal_records', 'assert_mask_equal',
21: (4)        'assert_not_equal', 'fail_if_array_equal',
```

```

22: (4) ]
23: (0)     from unittest import TestCase
24: (0)     __some_from_testing = [
25: (4)         'TestCase', 'assert_', 'assert_allclose',
'assert_array_almost_equal_nulp',
26: (4)         'assert_raises'
27: (4)     ]
28: (0)     __all__ = __all__masked + __some_from_testing
29: (0) def approx(a, b, fill_value=True, rtol=1e-5, atol=1e-8):
30: (4) """
31: (4)     Returns true if all components of a and b are equal to given tolerances.
32: (4)     If fill_value is True, masked values considered equal. Otherwise,
33: (4)     masked values are considered unequal. The relative error rtol should
34: (4)     be positive and << 1.0 The absolute error atol comes into play for
35: (4)     those elements of b that are very small or zero; it says how small a
36: (4)     must be also.
37: (4) """
38: (4)     m = mask_or(getmask(a), getmask(b))
39: (4)     d1 = filled(a)
40: (4)     d2 = filled(b)
41: (4)     if d1.dtype.char == "O" or d2.dtype.char == "O":
42: (8)         return np.equal(d1, d2).ravel()
43: (4)     x = filled(masked_array(d1, copy=False, mask=m),
fill_value).astype(float_)
44: (4)     y = filled(masked_array(d2, copy=False, mask=m), 1).astype(float_)
45: (4)     d = np.less_equal(umath.absolute(x - y), atol + rtol * umath.absolute(y))
46: (4)     return d.ravel()
47: (0) def almost(a, b, decimal=6, fill_value=True):
48: (4) """
49: (4)     Returns True if a and b are equal up to decimal places.
50: (4)     If fill_value is True, masked values considered equal. Otherwise,
51: (4)     masked values are considered unequal.
52: (4) """
53: (4)     m = mask_or(getmask(a), getmask(b))
54: (4)     d1 = filled(a)
55: (4)     d2 = filled(b)
56: (4)     if d1.dtype.char == "O" or d2.dtype.char == "O":
57: (8)         return np.equal(d1, d2).ravel()
58: (4)     x = filled(masked_array(d1, copy=False, mask=m),
fill_value).astype(float_)
59: (4)     y = filled(masked_array(d2, copy=False, mask=m), 1).astype(float_)
60: (4)     d = np.around(np.abs(x - y), decimal) <= 10.0 ** (-decimal)
61: (4)     return d.ravel()
62: (0) def __assert_equal_on_sequences(actual, desired, err_msg=''):
63: (4) """
64: (4)     Asserts the equality of two non-array sequences.
65: (4) """
66: (4)     assert_equal(len(actual), len(desired), err_msg)
67: (4)     for k in range(len(desired)):
68: (8)         assert_equal(actual[k], desired[k], f'item={k}!r\n{err_msg}')
69: (4)     return
70: (0) def assert_equal_records(a, b):
71: (4) """
72: (4)     Asserts that two records are equal.
73: (4)     Pretty crude for now.
74: (4) """
75: (4)     assert_equal(a.dtype, b.dtype)
76: (4)     for f in a.dtype.names:
77: (8)         (af, bf) = (operator.getitem(a, f), operator.getitem(b, f))
78: (8)         if not (af is masked) and not (bf is masked):
79: (12)             assert_equal(operator.getitem(a, f), operator.getitem(b, f))
80: (4)     return
81: (0) def assert_equal(actual, desired, err_msg=''):
82: (4) """
83: (4)     Asserts that two items are equal.
84: (4) """
85: (4)     if isinstance(desired, dict):
86: (8)         if not isinstance(actual, dict):
87: (12)             raise AssertionError(repr(type(actual)))

```

```

88: (8)             assert_equal(len(actual), len(desired), err_msg)
89: (8)             for k, i in desired.items():
90: (12)                 if k not in actual:
91: (16)                     raise AssertionError(f"{k} not in {actual}")
92: (12)                     assert_equal(actual[k], desired[k], f'key={k!r}\n{err_msg}')
93: (8)                     return
94: (4)             if isinstance(desired, (list, tuple)) and isinstance(actual, (list,
tuple)):
95: (8)                 return _assert_equal_on_sequences(actual, desired, err_msg='')
96: (4)             if not (isinstance(actual, ndarray) or isinstance(desired, ndarray)):
97: (8)                 msg = build_err_msg([actual, desired], err_msg,)
98: (8)                 if not desired == actual:
99: (12)                     raise AssertionError(msg)
100: (8)                 return
101: (4)             if ((actual is masked) and not (desired is masked)) or \
102: (12)                 ((desired is masked) and not (actual is masked)):
103: (8)                 msg = build_err_msg([actual, desired],
104: (28)                     err_msg, header='', names=('x', 'y'))
105: (8)                 raise ValueError(msg)
106: (4)             actual = np.asanyarray(actual)
107: (4)             desired = np.asanyarray(desired)
108: (4)             (actual_dtype, desired_dtype) = (actual.dtype, desired.dtype)
109: (4)             if actual_dtype.char == "S" and desired_dtype.char == "S":
110: (8)                 return _assert_equal_on_sequences(actual.tolist(),
111: (42)                     desired.tolist(),
112: (42)                     err_msg='')
113: (4)             return assert_array_equal(actual, desired, err_msg)
114: (0)             def fail_if_equal(actual, desired, err_msg=''):
115: (4)                 """
116: (4)                     Raises an assertion error if two items are equal.
117: (4)                 """
118: (4)                 if isinstance(desired, dict):
119: (8)                     if not isinstance(actual, dict):
120: (12)                         raise AssertionError(repr(type(actual)))
121: (8)                         fail_if_equal(len(actual), len(desired), err_msg)
122: (8)                         for k, i in desired.items():
123: (12)                             if k not in actual:
124: (16)                                 raise AssertionError(repr(k))
125: (12)                                 fail_if_equal(actual[k], desired[k], f'key={k!r}\n{err_msg}')
126: (8)                             return
127: (4)             if isinstance(desired, (list, tuple)) and isinstance(actual, (list,
tuple)):
128: (8)                 fail_if_equal(len(actual), len(desired), err_msg)
129: (8)                 for k in range(len(desired)):
130: (12)                     fail_if_equal(actual[k], desired[k], f'item={k!r}\n{err_msg}')
131: (8)                     return
132: (4)             if isinstance(actual, np.ndarray) or isinstance(desired, np.ndarray):
133: (8)                 return fail_if_array_equal(actual, desired, err_msg)
134: (4)             msg = build_err_msg([actual, desired], err_msg)
135: (4)             if not desired != actual:
136: (8)                 raise AssertionError(msg)
137: (0)             assert_not_equal = fail_if_equal
138: (0)             def assert_almost_equal(actual, desired, decimal=7, err_msg='', verbose=True):
139: (4)                 """
140: (4)                     Asserts that two items are almost equal.
141: (4)                     The test is equivalent to abs(desired-actual) < 0.5 * 10**(-decimal).
142: (4)                     """
143: (4)                     if isinstance(actual, np.ndarray) or isinstance(desired, np.ndarray):
144: (8)                         return assert_array_almost_equal(actual, desired, decimal=decimal,
145: (41)                             err_msg=err_msg, verbose=verbose)
146: (4)                         msg = build_err_msg([actual, desired],
147: (24)                             err_msg=err_msg, verbose=verbose)
148: (4)                         if not round(abs(desired - actual), decimal) == 0:
149: (8)                             raise AssertionError(msg)
150: (0)             assert_close = assert_almost_equal
151: (0)             def assert_array_compare(comparison, x, y, err_msg='', verbose=True,
header='',
152: (25)                             fill_value=True):
153: (4)                             """

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

154: (4)           Asserts that comparison between two masked arrays is satisfied.
155: (4)           The comparison is elementwise.
156: (4)
157: (4)           m = mask_or(getmask(x), getmask(y))
158: (4)           x = masked_array(x, copy=False, mask=m, keep_mask=False, subok=False)
159: (4)           y = masked_array(y, copy=False, mask=m, keep_mask=False, subok=False)
160: (4)           if ((x is masked) and not (y is masked)) or \
161: (12)             ((y is masked) and not (x is masked)):
162: (8)               msg = build_err_msg([x, y], err_msg=err_msg, verbose=verbose,
163: (28)                 header=header, names=('x', 'y'))
164: (8)               raise ValueError(msg)
165: (4)           return np.testing.assert_array_compare(comparison,
166: (43)                 x.filled(fill_value),
167: (43)                 y.filled(fill_value),
168: (43)                 err_msg=err_msg,
169: (43)                 verbose=verbose, header=header)
170: (0)           def assert_array_equal(x, y, err_msg='', verbose=True):
171: (4)             """
172: (4)             Checks the elementwise equality of two masked arrays.
173: (4)
174: (4)             assert_array_compare(operator.__eq__, x, y,
175: (25)               err_msg=err_msg, verbose=verbose,
176: (25)               header='Arrays are not equal')
177: (0)           def fail_if_array_equal(x, y, err_msg='', verbose=True):
178: (4)             """
179: (4)             Raises an assertion error if two masked arrays are not equal elementwise.
180: (4)
181: (4)             def compare(x, y):
182: (8)               return (not np.all(approx(x, y)))
183: (4)             assert_array_compare(compare, x, y, err_msg=err_msg, verbose=verbose,
184: (25)               header='Arrays are not equal')
185: (0)           def assert_array_approx_equal(x, y, decimal=6, err_msg='', verbose=True):
186: (4)             """
187: (4)             Checks the equality of two masked arrays, up to given number odecimals.
188: (4)             The equality is checked elementwise.
189: (4)
190: (4)             def compare(x, y):
191: (8)               "Returns the result of the loose comparison between x and y)."
192: (8)               return approx(x, y, rtol=10. ** -decimal)
193: (4)             assert_array_compare(compare, x, y, err_msg=err_msg, verbose=verbose,
194: (25)               header='Arrays are not almost equal')
195: (0)           def assert_array_almost_equal(x, y, decimal=6, err_msg='', verbose=True):
196: (4)             """
197: (4)             Checks the equality of two masked arrays, up to given number odecimals.
198: (4)             The equality is checked elementwise.
199: (4)
200: (4)             def compare(x, y):
201: (8)               "Returns the result of the loose comparison between x and y)."
202: (8)               return almost(x, y, decimal)
203: (4)             assert_array_compare(compare, x, y, err_msg=err_msg, verbose=verbose,
204: (25)               header='Arrays are not almost equal')
205: (0)           def assert_array_less(x, y, err_msg='', verbose=True):
206: (4)             """
207: (4)             Checks that x is smaller than y elementwise.
208: (4)
209: (4)             def compare(x, y):
210: (25)               assert_array_compare(operator.__lt__, x, y,
211: (25)                 err_msg=err_msg, verbose=verbose,
212: (0)                   header='Arrays are not less-ordered')
213: (4)
214: (4)             Asserts the equality of two masks.
215: (4)
216: (4)             if m1 is nomask:
217: (8)               assert_(m2 is nomask)
218: (4)             if m2 is nomask:
219: (8)               assert_(m1 is nomask)
220: (4)             assert_array_equal(m1, m2, err_msg=err_msg)

```

## File 266 - \_\_init\_\_.py:

```

1: (0)      """
2: (0)      =====
3: (0)      Masked Arrays
4: (0)      =====
5: (0)      Arrays sometimes contain invalid or missing data. When doing operations
6: (0)      on such arrays, we wish to suppress invalid values, which is the purpose
masked
7: (0)      arrays fulfill (an example of typical use is given below).
8: (0)      For example, examine the following array:
9: (0)      >>> x = np.array([2, 1, 3, np.nan, 5, 2, 3, np.nan])
10: (0)      When we try to calculate the mean of the data, the result is undetermined:
11: (0)      >>> np.mean(x)
12: (0)      nan
13: (0)      The mean is calculated using roughly ``np.sum(x)/len(x)``, but since
14: (0)      any number added to ``NaN`` [1]_ produces ``NaN``, this doesn't work. Enter
15: (0)      masked arrays:
16: (0)      >>> m = np.ma.masked_array(x, np.isnan(x))
17: (0)      >>> m
18: (0)      masked_array(data = [2.0 1.0 3.0 -- 5.0 2.0 3.0 --],
19: (6)          mask = [False False False  True False False  True],
20: (6)          fill_value=1e+20)
21: (0)      Here, we construct a masked array that suppress all ``NaN`` values. We
22: (0)      may now proceed to calculate the mean of the other values:
23: (0)      >>> np.mean(m)
24: (0)      2.6666666666666665
25: (0)      .. [1] Not-a-Number, a floating point value that is the result of an
26: (7)          invalid operation.
27: (0)      .. moduleauthor:: Pierre Gerard-Marchant
28: (0)      .. moduleauthor:: Jarrod Millman
29: (0)
30: (0)      """
31: (0)      from . import core
32: (0)      from .core import *
33: (0)      from . import extras
34: (0)      from .extras import *
35: (0)      __all__ = ['core', 'extras']
36: (0)      __all__ += core.__all__
37: (0)      __all__ += extras.__all__
38: (0)      from numpy._pytesttester import PytestTester
39: (0)      test = PytestTester(__name__)
40: (0)      del PytestTester
-----
```

## File 267 - timer\_comparison.py:

```

1: (0)      import timeit
2: (0)      from functools import reduce
3: (0)      import numpy as np
4: (0)      from numpy import float_
5: (0)      import numpy.core.fromnumeric as fromnumeric
6: (0)      from numpy.testing import build_err_msg
7: (0)      pi = np.pi
8: (0)      class ModuleTester:
9: (4)          def __init__(self, module):
10: (8)              self.module = module
11: (8)              self.allequal = module.allequal
12: (8)              self.arange = module.arange
13: (8)              self.array = module.array
14: (8)              self.concatenate = module.concatenate
15: (8)              self.count = module.count
16: (8)              self.equal = module.equal
17: (8)              self.filled = module.filled
18: (8)              self.getmask = module.getmask
19: (8)              self.getmaskarray = module.getmaskarray
20: (8)              self.id = id
21: (8)              self.inner = module.inner
```

```

22: (8)                     self.make_mask = module.make_mask
23: (8)                     self.masked = module.masked
24: (8)                     self.masked_array = module.masked_array
25: (8)                     self.masked_values = module.masked_values
26: (8)                     self.mask_or = module.mask_or
27: (8)                     self.nomask = module.nomask
28: (8)                     self.ones = module.ones
29: (8)                     self.outer = module.outer
30: (8)                     self.repeat = module.repeat
31: (8)                     self.resize = module.resize
32: (8)                     self.sort = module.sort
33: (8)                     self.take = module.take
34: (8)                     self.transpose = module.transpose
35: (8)                     self.zeros = module.zeros
36: (8)                     self.MaskType = module.MaskType
37: (8)                     try:
38: (12)                         self.umath = module.umath
39: (8)                     except AttributeError:
40: (12)                         self.umath = module.core.umath
41: (8)                     self.testnames = []
42: (4)                     def assert_array_compare(self, comparison, x, y, err_msg='', header='',
43: (25)                                     fill_value=True):
44: (8)                         """
45: (8)                         Assert that a comparison of two masked arrays is satisfied
elementwise.
46: (8)                         """
47: (8)                         xf = self.filled(x)
48: (8)                         yf = self.filled(y)
49: (8)                         m = self.mask_or(self.getmask(x), self.getmask(y))
50: (8)                         x = self.filled(self.masked_array(xf, mask=m), fill_value)
51: (8)                         y = self.filled(self.masked_array(yf, mask=m), fill_value)
52: (8)                         if (x.dtype.char != "O"):
53: (12)                             x = x.astype(float_)
54: (12)                             if isinstance(x, np.ndarray) and x.size > 1:
55: (16)                                 x[np.isnan(x)] = 0
56: (12)                             elif np.isnan(x):
57: (16)                                 x = 0
58: (8)                         if (y.dtype.char != "O"):
59: (12)                             y = y.astype(float_)
60: (12)                             if isinstance(y, np.ndarray) and y.size > 1:
61: (16)                                 y[np.isnan(y)] = 0
62: (12)                             elif np.isnan(y):
63: (16)                                 y = 0
64: (8)                         try:
65: (12)                             cond = (x.shape == () or y.shape == () or x.shape == y.shape
66: (12)                             if not cond:
67: (16)                                 msg = build_err_msg([x, y],
68: (36)                                     err_msg
69: (36)                                     + f'\n(shapes {x.shape}, {y.shape}
mismatch)',

70: (36)
71: (36)
72: (16)
73: (12)
74: (12)
75: (16)
76: (12)
77: (16)
78: (16)
79: (12)
80: (16)
81: (16)
82: (16)
83: (12)
84: (16)
85: (16)
86: (36)
87: (36)
88: (36)
                                         header=header,
                                         names=('x', 'y'))
72: (16)                             assert cond, msg
73: (12)                             val = comparison(x, y)
74: (12)                             if m is not self.nomask and fill_value:
75: (16)                                 val = self.masked_array(val, mask=m)
76: (12)                             if isinstance(val, bool):
77: (16)                                 cond = val
78: (16)                                 reduced = [0]
79: (12)                             else:
80: (16)                                 reduced = val.ravel()
81: (16)                                 cond = reduced.all()
82: (16)                                 reduced = reduced.tolist()
83: (12)                             if not cond:
84: (16)                                 match = 100-100.0*reduced.count(1)/len(reduced)
85: (16)                                 msg = build_err_msg([x, y],
                                         err_msg
                                         + '\n(mismatch %s%%)' % (match,),

                                         header=header,
```

```

89: (36)                                     names=('x', 'y'))
90: (16)         assert cond, msg
91: (8)     except ValueError as e:
92: (12)         msg = build_err_msg([x, y], err_msg, header=header, names=('x',
'y'))
93: (12)             raise ValueError(msg) from e
94: (4)     def assert_array_equal(self, x, y, err_msg=''):
95: (8)         """
96: (8)             Checks the elementwise equality of two masked arrays.
97: (8)         """
98: (8)             self.assert_array_compare(self.equal, x, y, err_msg=err_msg,
99: (34)                             header='Arrays are not equal')
100: (4) @np.errstate(all='ignore')
101: (4)     def test_0(self):
102: (8)         """
103: (8)             Tests creation
104: (8)         """
105: (8)             x = np.array([1., 1., 1., -2., pi/2.0, 4., 5., -10., 10., 1., 2., 3.])
106: (8)             m = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
107: (8)             xm = self.masked_array(x, mask=m)
108: (8)             xm[0]
109: (4) @np.errstate(all='ignore')
110: (4)     def test_1(self):
111: (8)         """
112: (8)             Tests creation
113: (8)         """
114: (8)             x = np.array([1., 1., 1., -2., pi/2.0, 4., 5., -10., 10., 1., 2., 3.])
115: (8)             y = np.array([5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.])
116: (8)             m1 = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
117: (8)             m2 = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1]
118: (8)             xm = self.masked_array(x, mask=m1)
119: (8)             ym = self.masked_array(y, mask=m2)
120: (8)             xf = np.where(m1, 1.e+20, x)
121: (8)             xm.set_fill_value(1.e+20)
122: (8)             assert((xm-ym).filled(0).any())
123: (8)             s = x.shape
124: (8)             assert(xm.size == reduce(lambda x, y:x*y, s))
125: (8)             assert(self.count(xm) == len(m1) - reduce(lambda x, y:x+y, m1))
126: (8)             for s in [(4, 3), (6, 2)]:
127: (12)                 xm.shape = s
128: (12)                 ym.shape = s
129: (12)                 xm.shape = s
130: (12)                 ym.shape = s
131: (12)                 xf.shape = s
132: (12)                 assert(self.count(xm) == len(m1) - reduce(lambda x, y:x+y, m1))
133: (4) @np.errstate(all='ignore')
134: (4)     def test_2(self):
135: (8)         """
136: (8)             Tests conversions and indexing.
137: (8)         """
138: (8)             x1 = np.array([1, 2, 4, 3])
139: (8)             x2 = self.array(x1, mask=[1, 0, 0, 0])
140: (8)             x3 = self.array(x1, mask=[0, 1, 0, 1])
141: (8)             x4 = self.array(x1)
142: (8)             str(x2)
143: (8)             repr(x2)
144: (8)             assert type(x2[1]) is type(x1[1])
145: (8)             assert x1[1] == x2[1]
146: (8)             x1[2] = 9
147: (8)             x2[2] = 9
148: (8)             self.assert_array_equal(x1, x2)
149: (8)             x1[1:3] = 99
150: (8)             x2[1:3] = 99
151: (8)             x2[1] = self.masked
152: (8)             x2[1:3] = self.masked
153: (8)             x2[:] = x1
154: (8)             x2[1] = self.masked
155: (8)             x3[:] = self.masked_array([1, 2, 3, 4], [0, 1, 1, 0])
156: (8)             x4[:] = self.masked_array([1, 2, 3, 4], [0, 1, 1, 0])

```

```

157: (8)           x1 = np.arange(5)*1.0
158: (8)           x2 = self.masked_values(x1, 3.0)
159: (8)           x1 = self.array([1, 'hello', 2, 3], object)
160: (8)           x2 = np.array([1, 'hello', 2, 3], object)
161: (8)           x1[1]
162: (8)           x2[1]
163: (8)           assert x1[1:1].shape == (0,)
164: (8)           n = [0, 0, 1, 0, 0]
165: (8)           m = self.make_mask(n)
166: (8)           m2 = self.make_mask(m)
167: (8)           assert(m is m2)
168: (8)           m3 = self.make_mask(m, copy=1)
169: (8)           assert(m is not m3)
170: (4)           @np.errstate(all='ignore')
171: (4)           def test_3(self):
172: (8)               """
173: (8)               Tests resize/repeat
174: (8)               """
175: (8)               x4 = self.arange(4)
176: (8)               x4[2] = self.masked
177: (8)               y4 = self.resize(x4, (8,))
178: (8)               assert self.allequal(self.concatenate([x4, x4]), y4)
179: (8)               assert self.allequal(self.getmask(y4), [0, 0, 1, 0, 0, 0, 1, 0])
180: (8)               y5 = self.repeat(x4, (2, 2, 2, 2), axis=0)
181: (8)               self.assert_array_equal(y5, [0, 0, 1, 1, 2, 2, 3, 3])
182: (8)               y6 = self.repeat(x4, 2, axis=0)
183: (8)               assert self.allequal(y5, y6)
184: (8)               y7 = x4.repeat((2, 2, 2, 2), axis=0)
185: (8)               assert self.allequal(y5, y7)
186: (8)               y8 = x4.repeat(2, 0)
187: (8)               assert self.allequal(y5, y8)
188: (4)           @np.errstate(all='ignore')
189: (4)           def test_4(self):
190: (8)               """
191: (8)               Test of take, transpose, inner, outer products.
192: (8)               """
193: (8)               x = self.arange(24)
194: (8)               y = np.arange(24)
195: (8)               x[5:6] = self.masked
196: (8)               x = x.reshape(2, 3, 4)
197: (8)               y = y.reshape(2, 3, 4)
198: (8)               assert self.allequal(np.transpose(y, (2, 0, 1)), self.transpose(x, (2, 0, 1)))
199: (8)               assert self.allequal(np.take(y, (2, 0, 1), 1), self.take(x, (2, 0, 1), 1))
200: (8)               assert self.allequal(np.inner(self.filled(x, 0), self.filled(y, 0)),
201: (28)                           self.inner(x, y))
202: (8)               assert self.allequal(np.outer(self.filled(x, 0), self.filled(y, 0)),
203: (28)                           self.outer(x, y))
204: (8)               y = self.array(['abc', 1, 'def', 2, 3], object)
205: (8)               y[2] = self.masked
206: (8)               t = self.take(y, [0, 3, 4])
207: (8)               assert t[0] == 'abc'
208: (8)               assert t[1] == 2
209: (8)               assert t[2] == 3
210: (4)           @np.errstate(all='ignore')
211: (4)           def test_5(self):
212: (8)               """
213: (8)               Tests inplace w/ scalar
214: (8)               """
215: (8)               x = self.arange(10)
216: (8)               y = self.arange(10)
217: (8)               xm = self.arange(10)
218: (8)               xm[2] = self.masked
219: (8)               x += 1
220: (8)               assert self.allequal(x, y+1)
221: (8)               xm += 1
222: (8)               assert self.allequal(xm, y+1)
223: (8)               x = self.arange(10)

```

```

224: (8)             xm = self.arange(10)
225: (8)             xm[2] = self.masked
226: (8)             x -= 1
227: (8)             assert self.allequal(x, y-1)
228: (8)             xm -= 1
229: (8)             assert self.allequal(xm, y-1)
230: (8)             x = self.arange(10)*1.0
231: (8)             xm = self.arange(10)*1.0
232: (8)             xm[2] = self.masked
233: (8)             x *= 2.0
234: (8)             assert self.allequal(x, y*2)
235: (8)             xm *= 2.0
236: (8)             assert self.allequal(xm, y*2)
237: (8)             x = self.arange(10)*2
238: (8)             xm = self.arange(10)*2
239: (8)             xm[2] = self.masked
240: (8)             x /= 2
241: (8)             assert self.allequal(x, y)
242: (8)             xm /= 2
243: (8)             assert self.allequal(xm, y)
244: (8)             x = self.arange(10)*1.0
245: (8)             xm = self.arange(10)*1.0
246: (8)             xm[2] = self.masked
247: (8)             x /= 2.0
248: (8)             assert self.allequal(x, y/2.0)
249: (8)             xm /= self.arange(10)
250: (8)             self.assert_array_equal(xm, self.ones((10,)))
251: (8)             x = self.arange(10).astype(float_)
252: (8)             xm = self.arange(10)
253: (8)             xm[2] = self.masked
254: (8)             x += 1.
255: (8)             assert self.allequal(x, y + 1.)
256: (4)             @np.errstate(all='ignore')
257: (4)             def test_6(self):
258: (8)                 """
259: (8)                 Tests inplace w/ array
260: (8)                 """
261: (8)                 x = self.arange(10, dtype=float_)
262: (8)                 y = self.arange(10)
263: (8)                 xm = self.arange(10, dtype=float_)
264: (8)                 xm[2] = self.masked
265: (8)                 m = xm.mask
266: (8)                 a = self.arange(10, dtype=float_)
267: (8)                 a[-1] = self.masked
268: (8)                 x += a
269: (8)                 xm += a
270: (8)                 assert self.allequal(x, y+a)
271: (8)                 assert self.allequal(xm, y+a)
272: (8)                 assert self.allequal(xm.mask, self.mask_or(m, a.mask))
273: (8)                 x = self.arange(10, dtype=float_)
274: (8)                 xm = self.arange(10, dtype=float_)
275: (8)                 xm[2] = self.masked
276: (8)                 m = xm.mask
277: (8)                 a = self.arange(10, dtype=float_)
278: (8)                 a[-1] = self.masked
279: (8)                 x -= a
280: (8)                 xm -= a
281: (8)                 assert self.allequal(x, y-a)
282: (8)                 assert self.allequal(xm, y-a)
283: (8)                 assert self.allequal(xm.mask, self.mask_or(m, a.mask))
284: (8)                 x = self.arange(10, dtype=float_)
285: (8)                 xm = self.arange(10, dtype=float_)
286: (8)                 xm[2] = self.masked
287: (8)                 m = xm.mask
288: (8)                 a = self.arange(10, dtype=float_)
289: (8)                 a[-1] = self.masked
290: (8)                 x *= a
291: (8)                 xm *= a
292: (8)                 assert self.allequal(x, y*a)

```

```

293: (8)             assert self.allequal(xm, y*a)
294: (8)             assert self.allequal(xm.mask, self.mask_or(m, a.mask))
295: (8)             x = self.arange(10, dtype=float_)
296: (8)             xm = self.arange(10, dtype=float_)
297: (8)             xm[2] = self.masked
298: (8)             m = xm.mask
299: (8)             a = self.arange(10, dtype=float_)
300: (8)             a[-1] = self.masked
301: (8)             x /= a
302: (8)             xm /= a
303: (4)             @np.errstate(all='ignore')
304: (4)             def test_7(self):
305: (8)                 "Tests ufunc"
306: (8)                 d = (self.array([1.0, 0, -1, pi/2]*2, mask=[0, 1]+[0]*6),
307: (13)                     self.array([1.0, 0, -1, pi/2]*2, mask=[1, 0]+[0]*6),)
308: (8)                 for f in ['sqrt', 'log', 'log10', 'exp', 'conjugate',
309: (18)                     ]:
310: (12)                 try:
311: (16)                     uf = getattr(self.umath, f)
312: (12)                 except AttributeError:
313: (16)                     uf = getattr(fromnumeric, f)
314: (12)                     mf = getattr(self.module, f)
315: (12)                     args = d[:uf.nin]
316: (12)                     ur = uf(*args)
317: (12)                     mr = mf(*args)
318: (12)                     self.assert_array_equal(ur.filled(0), mr.filled(0), f)
319: (12)                     self.assert_array_equal(ur._mask, mr._mask)
320: (4)             @np.errstate(all='ignore')
321: (4)             def test_99(self):
322: (8)                 ott = self.array([0., 1., 2., 3.], mask=[1, 0, 0, 0])
323: (8)                 self.assert_array_equal(2.0, self.average(ott, axis=0))
324: (8)                 self.assert_array_equal(2.0, self.average(ott, weights=[1., 1., 2.,
1.]))
325: (8)                 result, wts = self.average(ott, weights=[1., 1., 2., 1.], returned=1)
326: (8)                 self.assert_array_equal(2.0, result)
327: (8)                 assert(wts == 4.0)
328: (8)                 ott[:] = self.masked
329: (8)                 assert(self.average(ott, axis=0) is self.masked)
330: (8)                 ott = self.array([0., 1., 2., 3.], mask=[1, 0, 0, 0])
331: (8)                 ott = ott.reshape(2, 2)
332: (8)                 ott[:, 1] = self.masked
333: (8)                 self.assert_array_equal(self.average(ott, axis=0), [2.0, 0.0])
334: (8)                 assert(self.average(ott, axis=1)[0] is self.masked)
335: (8)                 self.assert_array_equal([2., 0.], self.average(ott, axis=0))
336: (8)                 result, wts = self.average(ott, axis=0, returned=1)
337: (8)                 self.assert_array_equal(wts, [1., 0.])
338: (8)                 w1 = [0, 1, 1, 1, 1, 0]
339: (8)                 w2 = [[0, 1, 1, 1, 1, 0], [1, 0, 0, 0, 0, 1]]
340: (8)                 x = self.arange(6)
341: (8)                 self.assert_array_equal(self.average(x, axis=0), 2.5)
342: (8)                 self.assert_array_equal(self.average(x, axis=0, weights=w1), 2.5)
343: (8)                 y = self.array([self.arange(6), 2.0*self.arange(6)])
344: (8)                 self.assert_array_equal(self.average(y, None),
np.add.reduce(np.arange(6))*3./12.)
345: (8)                 self.assert_array_equal(self.average(y, axis=0), np.arange(6) * 3./2.)
346: (8)                 self.assert_array_equal(self.average(y, axis=1), [self.average(x,
axis=0), self.average(x, axis=0) * 2.0])
347: (8)                 self.assert_array_equal(self.average(y, None, weights=w2), 20./6.)
348: (8)                 self.assert_array_equal(self.average(y, axis=0, weights=w2), [0., 1.,
2., 3., 4., 10.])
349: (8)                 self.assert_array_equal(self.average(y, axis=1), [self.average(x,
axis=0), self.average(x, axis=0) * 2.0])
350: (8)                 m1 = self.zeros(6)
351: (8)                 m2 = [0, 0, 1, 1, 0, 0]
352: (8)                 m3 = [[0, 0, 1, 1, 0, 0], [0, 1, 1, 1, 1, 0]]
353: (8)                 m4 = self.ones(6)
354: (8)                 m5 = [0, 1, 1, 1, 1, 1]
355: (8)                 self.assert_array_equal(self.average(self.masked_array(x, m1),
axis=0), 2.5)

```

```

356: (8)                     self.assert_array_equal(self.average(self.masked_array(x, m2),
axis=0), 2.5)
357: (8)                     self.assert_array_equal(self.average(self.masked_array(x, m5),
axis=0), 0.0)
358: (8)                     self.assert_array_equal(self.count(self.average(self.masked_array(x,
m4), axis=0)), 0)
359: (8)                     z = self.masked_array(y, m3)
360: (8)                     self.assert_array_equal(self.average(z, None), 20./6.)
361: (8)                     self.assert_array_equal(self.average(z, axis=0), [0., 1., 99., 99.,
4.0, 7.5])
362: (8)                     self.assert_array_equal(self.average(z, axis=1), [2.5, 5.0])
363: (8)                     self.assert_array_equal(self.average(z, axis=0, weights=w2), [0., 1.,
99., 99., 4.0, 10.0])
364: (4)                     @np.errstate(all='ignore')
365: (4)                     def test_A(self):
366: (8)                         x = self.arange(24)
367: (8)                         x[5:6] = self.masked
368: (8)                         x = x.reshape(2, 3, 4)
369: (0)                     if __name__ == '__main__':
370: (4)                         setup_base = ("from __main__ import ModuleTester\n"
371: (18)                           "import numpy\n"
372: (18)                           "tester = ModuleTester(module)\n")
373: (4)                         setup_cur = "import numpy.ma.core as module\n" + setup_base
374: (4)                         (nrepeat, nloop) = (10, 10)
375: (4)                         for i in range(1, 8):
376: (8)                             func = 'tester.test_%i()' % i
377: (8)                             cur = timeit.Timer(func, setup_cur).repeat(nrepeat, nloop*10)
378: (8)                             cur = np.sort(cur)
379: (8)                             print("#%i" % i + 50*'.')
380: (8)                             print(eval("ModuleTester.test_%i.__doc__" % i))
381: (8)                             print(f'core_current : {cur[0]:.3f} - {cur[1]:.3f}')

```

---

## File 268 - test\_core.py:

```

1: (0)                     """Tests suite for MaskedArray & subclassing.
2: (0)                     :author: Pierre Gerard-Marchant
3: (0)                     :contact: pierregm_at_uga_dot_edu
4: (0)                     """
5: (0)                     __author__ = "Pierre GF Gerard-Marchant"
6: (0)                     import sys
7: (0)                     import warnings
8: (0)                     import copy
9: (0)                     import operator
10: (0)                    import itertools
11: (0)                    import textwrap
12: (0)                    import pytest
13: (0)                    from functools import reduce
14: (0)                    import numpy as np
15: (0)                    import numpy.ma.core
16: (0)                    import numpy.core.fromnumeric as fromnumeric
17: (0)                    import numpy.core.umath as umath
18: (0)                    from numpy.testing import (
19: (4)                        assert_raises, assert_warnings, suppress_warnings, IS_WASM
20: (4)                    )
21: (0)                    from numpy.testing._private.utils import requires_memory
22: (0)                    from numpy import ndarray
23: (0)                    from numpy.compat import asbytes
24: (0)                    from numpy.ma.testutils import (
25: (4)                        assert_, assert_array_equal, assert_equal, assert_almost_equal,
26: (4)                        assert_equal_records, fail_if_equal, assert_not_equal,
27: (4)                        assert_mask_equal
28: (4)                    )
29: (0)                    from numpy.ma.core import (
30: (4)                        MAError, MaskError, MaskType, MaskedArray, abs, absolute, add, all,
31: (4)                        allclose, allequal, alltrue, angle, anom, arange, arccos, arccosh,
32: (4)                        arcsin, arctan, argsort, array, asarray, choose, concatenate,

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

33: (4) conjugate, cos, cosh, count, default_fill_value, diag, divide, doc_note,
34: (4) empty, empty_like, equal, exp, flatten_mask, filled, fix_invalid,
35: (4) flatten_structured_array, fromflex, getmask, getmaskarray, greater,
36: (4) greater_equal, identity, inner, isMaskedArray, less, less_equal, log,
37: (4) log10, make_mask, make_mask_descr, mask_or, masked, masked_array,
38: (4) masked_equal, masked_greater, masked_greater_equal, masked_inside,
39: (4) masked_less, masked_less_equal, masked_not_equal, masked_outside,
40: (4) masked_print_option, masked_values, masked_where, max, maximum,
41: (4) maximum_fill_value, min, minimum, minimum_fill_value, mod, multiply,
42: (4) mvoid, nomask, not_equal, ones, ones_like, outer, power, product, put,
43: (4) putmask, ravel, repeat, reshape, resize, shape, sin, sinh, sometrue, sort,
44: (4) sqrt, subtract, sum, take, tan, tanh, transpose, where, zeros, zeros_like,
45: (4)
46: (0) )
47: (0) from numpy.compat import pickle
48: (0) pi = np.pi
49: (0) suppress_copy_mask_on_assignment = suppress_warnings()
50: (4) suppress_copy_mask_on_assignment.filter(
51: (4)     numpy.ma.core.MaskedArrayFutureWarning,
52: (4)     "setting an item on a masked array which has a shared mask will not copy")
53: (0) num_dts = [np.dtype(dt_) for dt_ in '?bhilqBHILQefdgFD']
54: (0) num_ids = [dt_.char for dt_ in num_dts]
55: (0) class TestMaskedArray:
56: (4)     def setup_method(self):
57: (8)         x = np.array([1., 1., 1., -2., pi/2.0, 4., 5., -10., 10., 1., 2., 3.])
58: (8)         y = np.array([5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.])
59: (8)         a10 = 10.
60: (8)         m1 = [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
61: (8)         m2 = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1]
62: (8)         xm = masked_array(x, mask=m1)
63: (8)         ym = masked_array(y, mask=m2)
64: (8)         z = np.array([-0.5, 0., 0.5, 0.8])
65: (8)         zm = masked_array(z, mask=[0, 1, 0, 0])
66: (8)         xf = np.where(m1, 1e+20, x)
67: (8)         xm.set_fill_value(1e+20)
68: (4)         self.d = (x, y, a10, m1, m2, xm, ym, z, zm, xf)
69: (8)     def test_basic_attributes(self):
70: (8)         a = array([1, 3, 2])
71: (8)         b = array([1, 3, 2], mask=[1, 0, 1])
72: (8)         assert_equal(a.ndim, 1)
73: (8)         assert_equal(b.ndim, 1)
74: (8)         assert_equal(a.size, 3)
75: (8)         assert_equal(b.size, 3)
76: (8)         assert_equal(a.shape, (3,))
77: (8)         assert_equal(b.shape, (3,))
78: (4)     def test_basic0d(self):
79: (8)         x = masked_array(0)
80: (8)         assert_equal(str(x), '0')
81: (8)         x = masked_array(0, mask=True)
82: (8)         assert_equal(str(x), str(masked_print_option))
83: (8)         x = masked_array(0, mask=False)
84: (8)         assert_equal(str(x), '0')
85: (8)         x = array(0, mask=1)
86: (8)         assert_(x.filled().dtype is x._data.dtype)
87: (4)     def test_basic1d(self):
88: (8)         (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
89: (8)         assert_(not isMaskedArray(x))
90: (8)         assert_(isMaskedArray(xm))
91: (8)         assert_((xm - ym).filled(0).any())
92: (8)         fail_if_equal(xm.mask.astype(int), ym.mask.astype(int))
93: (8)         s = x.shape
94: (8)         assert_equal(np.shape(xm), s)
95: (8)         assert_equal(xm.shape, s)
96: (8)         assert_equal(xm.dtype, x.dtype)
97: (8)         assert_equal(zm.dtype, z.dtype)
98: (8)         assert_equal(xm.size, reduce(lambda x, y:x * y, s))
99: (8)         assert_equal(count(xm), len(m1) - reduce(lambda x, y:x + y, m1))
100: (8)        assert_array_equal(xm, xf)
101: (8)        assert_array_equal(filled(xm, 1.e20), xf)
102: (8)        assert_array_equal(x, xm)

```

```

102: (4)
103: (8)
104: (8)
105: (12)
106: (12)
107: (12)
108: (12)
109: (12)
110: (12)
111: (12)
112: (12)
113: (12)
114: (12)
115: (12)
116: (12)
117: (12)
118: (12)
119: (4)
120: (8)
121: (8)
122: (8)
123: (8)
124: (8)
125: (4)
126: (8)
127: (8)
128: (8)
129: (8)
130: (8)
131: (8)
132: (8)
133: (8)
134: (8)
135: (8)
136: (8)
137: (8)
138: (8)
139: (8)
140: (8)
141: (8)
142: (4)
143: (8)
144: (37)
145: (28)
146: (8)
147: (8)
148: (4)
149: (8)
150: (8)
151: (8)
152: (8)
153: (4)
154: (8)
155: (8)
156: (8)
157: (8)
158: (8)
159: (4)
160: (8)
161: (8)
162: (8)
163: (8)
164: (8)
165: (8)
166: (8)
167: (8)
168: (8)
169: (8)
170: (8)

    def test_basic2d(self):
        (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
        for s in [(4, 3), (6, 2)]:
            x.shape = s
            y.shape = s
            xm.shape = s
            ym.shape = s
            xf.shape = s
            assert_(not isMaskedArray(x))
            assert_(isMaskedArray(xm))
            assert_equal(shape(xm), s)
            assert_equal(xm.shape, s)
            assert_equal(xm.size, reduce(lambda x, y:x * y, s))
            assert_equal(count(xm), len(m1) - reduce(lambda x, y:x + y, m1))
            assert_equal(xm, xf)
            assert_equal(filled(xm, 1.e20), xf)
            assert_equal(x, xm)

    def test_concatenate_basic(self):
        (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
        assert_equal(np.concatenate((x, y)), concatenate((xm, ym)))
        assert_equal(np.concatenate((x, y)), concatenate((x, y)))
        assert_equal(np.concatenate((x, y)), concatenate((xm, y)))
        assert_equal(np.concatenate((x, y, x)), concatenate((x, ym, x)))

    def test_concatenate_alongaxis(self):
        (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
        s = (3, 4)
        x.shape = y.shape = xm.shape = ym.shape = s
        assert_equal(xm.mask, np.reshape(m1, s))
        assert_equal(ym.mask, np.reshape(m2, s))
        xmym = concatenate((xm, ym), 1)
        assert_equal(np.concatenate((x, y), 1), xmym)
        assert_equal(np.concatenate((xm.mask, ym.mask), 1), xmym._mask)
        x = zeros(2)
        y = array(ones(2), mask=[False, True])
        z = concatenate((x, y))
        assert_array_equal(z, [0, 0, 1, 1])
        assert_array_equal(z.mask, [False, False, False, True])
        z = concatenate((y, x))
        assert_array_equal(z, [1, 1, 0, 0])
        assert_array_equal(z.mask, [False, True, False, False])

    def test_concatenate_flexible(self):
        data = masked_array(list(zip(np.random.rand(10),
                                     np.arange(10))),
                            dtype=[('a', float), ('b', int)])
        test = concatenate([data[:5], data[5:]])
        assert_equal_records(test, data)

    def test_creation_ndmin(self):
        x = array([1, 2, 3], mask=[1, 0, 0], ndmin=2)
        assert_equal(x.shape, (1, 3))
        assert_equal(x._data, [[1, 2, 3]])
        assert_equal(x._mask, [[1, 0, 0]])

    def test_creation_ndmin_from_maskedarray(self):
        x = array([1, 2, 3])
        x[-1] = masked
        xx = array(x, ndmin=2, dtype=float)
        assert_equal(x.shape, x._mask.shape)
        assert_equal(xx.shape, xx._mask.shape)

    def test_creation_maskcreation(self):
        data = arange(24, dtype=float)
        data[[3, 6, 15]] = masked
        dma_1 = MaskedArray(data)
        assert_equal(dma_1.mask, data.mask)
        dma_2 = MaskedArray(dma_1)
        assert_equal(dma_2.mask, dma_1.mask)
        dma_3 = MaskedArray(dma_1, mask=[1, 0, 0, 0] * 6)
        fail_if_equal(dma_3.mask, dma_1.mask)
        x = array([1, 2, 3], mask=True)
        assert_equal(x._mask, [True, True, True])
        x = array([1, 2, 3], mask=False)

```

```

171: (8) assert_equal(x._mask, [False, False, False])
172: (8) y = array([1, 2, 3], mask=x._mask, copy=False)
173: (8) assert_(np.may_share_memory(x.mask, y.mask))
174: (8) y = array([1, 2, 3], mask=x._mask, copy=True)
175: (8) assert_(not np.may_share_memory(x.mask, y.mask))
176: (8) x = array([1, 2, 3], mask=None)
177: (8) assert_equal(x._mask, [False, False, False])
178: (4) def test_masked_singleton_array_creation_warns(self):
179: (8)     np.array(np.ma.masked)
180: (8)     with pytest.warns(UserWarning):
181: (12)         np.array([3., np.ma.masked])
182: (4) def test_creation_with_list_of_maskedarrays(self):
183: (8)     x = array(np.arange(5), mask=[1, 0, 0, 0, 0])
184: (8)     data = array((x, x[::-1]))
185: (8)     assert_equal(data, [[0, 1, 2, 3, 4], [4, 3, 2, 1, 0]])
186: (8)     assert_equal(data._mask, [[1, 0, 0, 0, 0], [0, 0, 0, 0, 1]])
187: (8)     x.mask = nomask
188: (8)     data = array((x, x[::-1]))
189: (8)     assert_equal(data, [[0, 1, 2, 3, 4], [4, 3, 2, 1, 0]])
190: (8)     assert_(data.mask is nomask)
191: (4) def test_creation_with_list_of_maskedarrays_no_bool_cast(self):
192: (8)     masked_str = np.ma.masked_array(['a', 'b'], mask=[True, False])
193: (8)     normal_int = np.arange(2)
194: (8)     res = np.ma.asarray([masked_str, normal_int], dtype="U21")
195: (8)     assert_array_equal(res.mask, [[True, False], [False, False]])
196: (8) class NotBool():
197: (12)     def __bool__(self):
198: (16)         raise ValueError("not a bool!")
199: (8)     masked_obj = np.ma.masked_array([NotBool(), 'b'], mask=[True, False])
200: (8)     with pytest.raises(ValueError, match="not a bool!"):
201: (12)         np.asarray([masked_obj], dtype=bool)
202: (8)     res = np.ma.asarray([masked_obj, normal_int])
203: (8)     assert_array_equal(res.mask, [[True, False], [False, False]])
204: (4) def test_creation_from_ndarray_with_padding(self):
205: (8)     x = np.array([('A', 0)], dtype={'names':['f0','f1'],
206: (40)             'formats':['S4','i8'],
207: (40)             'offsets':[0,8]})
```

208: (8) array(x) # used to fail due to 'V' padding field in x.dtype.descr

```

209: (4) def test_unknown_keyword_parameter(self):
210: (8)     with pytest.raises(TypeError, match="unexpected keyword argument"):
211: (12)         MaskedArray([1, 2, 3], maks=[0, 1, 0]) # `mask` is misspelled.
```

```

212: (4) def test_asarray(self):
213: (8)     (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
214: (8)     xm.fill_value = -9999
215: (8)     xm._hardmask = True
216: (8)     xmm = asarray(xm)
217: (8)     assert_equal(xmm._data, xm._data)
218: (8)     assert_equal(xmm._mask, xm._mask)
219: (8)     assert_equal(xmm.fill_value, xm.fill_value)
220: (8)     assert_equal(xmm._hardmask, xm._hardmask)
```

```

221: (4) def test_asarray_default_order(self):
222: (8)     m = np.eye(3).T
223: (8)     assert_(not m.flags.c_contiguous)
224: (8)     new_m = asarray(m)
225: (8)     assert_(new_m.flags.c_contiguous)
```

```

226: (4) def test_asarray_enforce_order(self):
227: (8)     m = np.eye(3).T
228: (8)     assert_(not m.flags.c_contiguous)
229: (8)     new_m = asarray(m, order='C')
230: (8)     assert_(new_m.flags.c_contiguous)
```

```

231: (4) def test_fix_invalid(self):
232: (8)     with np.errstate(invalid='ignore'):
233: (12)         data = masked_array([np.nan, 0., 1.], mask=[0, 0, 1])
234: (12)         data_fixed = fix_invalid(data)
235: (12)         assert_equal(data_fixed._data, [data.fill_value, 0., 1.])
236: (12)         assert_equal(data_fixed._mask, [1., 0., 1.])
```

```

237: (4) def test_maskedelement(self):
238: (8)     x = arange(6)
239: (8)     x[1] = masked

```

```

240: (8)             assert_(str(masked) == '--')
241: (8)             assert_(x[1] is masked)
242: (8)             assert_equal(filled(x[1], 0), 0)
243: (4)             def test_set_element_as_object(self):
244: (8)                 a = empty(1, dtype=object)
245: (8)                 x = (1, 2, 3, 4, 5)
246: (8)                 a[0] = x
247: (8)                 assert_equal(a[0], x)
248: (8)                 assert_(a[0] is x)
249: (8)                 import datetime
250: (8)                 dt = datetime.datetime.now()
251: (8)                 a[0] = dt
252: (8)                 assert_(a[0] is dt)
253: (4)             def test_indexing(self):
254: (8)                 x1 = np.array([1, 2, 4, 3])
255: (8)                 x2 = array(x1, mask=[1, 0, 0, 0])
256: (8)                 x3 = array(x1, mask=[0, 1, 0, 1])
257: (8)                 x4 = array(x1)
258: (8)                 str(x2) # raises?
259: (8)                 repr(x2) # raises?
260: (8)                 assert_equal(np.sort(x1), sort(x2, endwith=False))
261: (8)                 assert_(type(x2[1]) is type(x1[1]))
262: (8)                 assert_(x1[1] == x2[1])
263: (8)                 assert_(x2[0] is masked)
264: (8)                 assert_equal(x1[2], x2[2])
265: (8)                 assert_equal(x1[2:5], x2[2:5])
266: (8)                 assert_equal(x1[:], x2[:])
267: (8)                 assert_equal(x1[1:], x3[1:])
268: (8)                 x1[2] = 9
269: (8)                 x2[2] = 9
270: (8)                 assert_equal(x1, x2)
271: (8)                 x1[1:3] = 99
272: (8)                 x2[1:3] = 99
273: (8)                 assert_equal(x1, x2)
274: (8)                 x2[1] = masked
275: (8)                 assert_equal(x1, x2)
276: (8)                 x2[1:3] = masked
277: (8)                 assert_equal(x1, x2)
278: (8)                 x2[:] = x1
279: (8)                 x2[1] = masked
280: (8)                 assert_(allequal(getmask(x2), array([0, 1, 0, 0])))
281: (8)                 x3[:] = masked_array([1, 2, 3, 4], [0, 1, 1, 0])
282: (8)                 assert_(allequal(getmask(x3), array([0, 1, 1, 0])))
283: (8)                 x4[:] = masked_array([1, 2, 3, 4], [0, 1, 1, 0])
284: (8)                 assert_(allequal(getmask(x4), array([0, 1, 1, 0])))
285: (8)                 assert_(allequal(x4, array([1, 2, 3, 4])))
286: (8)                 x1 = np.arange(5) * 1.0
287: (8)                 x2 = masked_values(x1, 3.0)
288: (8)                 assert_equal(x1, x2)
289: (8)                 assert_(allequal(array([0, 0, 0, 1, 0], MaskType), x2.mask))
290: (8)                 assert_equal(3.0, x2.fill_value)
291: (8)                 x1 = array([1, 'hello', 2, 3], object)
292: (8)                 x2 = np.array([1, 'hello', 2, 3], object)
293: (8)                 s1 = x1[1]
294: (8)                 s2 = x2[1]
295: (8)                 assert_equal(type(s2), str)
296: (8)                 assert_equal(type(s1), str)
297: (8)                 assert_equal(s1, s2)
298: (8)                 assert_(x1[1:1].shape == (0,))
299: (4)             def test_setitem_no_warning(self):
300: (8)                 x = np.ma.arange(60).reshape((6, 10))
301: (8)                 index = (slice(1, 5, 2), [7, 5])
302: (8)                 value = np.ma.masked_all((2, 2))
303: (8)                 value._data[...] = np.inf # not a valid integer...
304: (8)                 x[index] = value
305: (8)                 x[...] = np.ma.masked
306: (8)                 x = np.ma.arange(3., dtype=np.float32)
307: (8)                 value = np.ma.array([2e234, 1, 1], mask=[True, False, False])
308: (8)                 x[...] = value

```

```

309: (8)           x[[0, 1, 2]] = value
310: (4)           @suppress_copy_mask_on_assignment
311: (4)           def test_copy(self):
312: (8)             n = [0, 0, 1, 0, 0]
313: (8)             m = make_mask(n)
314: (8)             m2 = make_mask(m)
315: (8)             assert_(m is m2)
316: (8)             m3 = make_mask(m, copy=True)
317: (8)             assert_(m is not m3)
318: (8)             x1 = np.arange(5)
319: (8)             y1 = array(x1, mask=m)
320: (8)             assert_equal(y1._data.__array_interface__, x1.__array_interface__)
321: (8)             assert_(allequal(x1, y1.data))
322: (8)             assert_equal(y1._mask.__array_interface__, m.__array_interface__)
323: (8)             y1a = array(y1)
324: (8)             assert_(y1a._data.__array_interface__ ==
325: (24)                   y1._data.__array_interface__)
326: (8)             assert_(y1a._mask.__array_interface__ ==
327: (24)                   y1._mask.__array_interface__)
328: (8)             y2 = array(x1, mask=m3)
329: (8)             assert_(y2._data.__array_interface__ == x1.__array_interface__)
330: (8)             assert_(y2._mask.__array_interface__ == m3.__array_interface__)
331: (8)             assert_(y2[2] is masked)
332: (8)             y2[2] = 9
333: (8)             assert_(y2[2] is not masked)
334: (8)             assert_(y2._mask.__array_interface__ == m3.__array_interface__)
335: (8)             assert_(allequal(y2.mask, 0))
336: (8)             y2a = array(x1, mask=m, copy=1)
337: (8)             assert_(y2a._data.__array_interface__ != x1.__array_interface__)
338: (8)             assert_(y2a._mask.__array_interface__ != m.__array_interface__)
339: (8)             assert_(y2a[2] is masked)
340: (8)             y2a[2] = 9
341: (8)             assert_(y2a[2] is not masked)
342: (8)             assert_(y2a._mask.__array_interface__ != m.__array_interface__)
343: (8)             assert_(allequal(y2a.mask, 0))
344: (8)             y3 = array(x1 * 1.0, mask=m)
345: (8)             assert_(filled(y3).dtype is (x1 * 1.0).dtype)
346: (8)             x4 = arange(4)
347: (8)             x4[2] = masked
348: (8)             y4 = resize(x4, (8,))
349: (8)             assert_equal(concatenate([x4, x4]), y4)
350: (8)             assert_equal(getmask(y4), [0, 0, 1, 0, 0, 0, 1, 0])
351: (8)             y5 = repeat(x4, (2, 2, 2, 2), axis=0)
352: (8)             assert_equal(y5, [0, 0, 1, 1, 2, 2, 3, 3])
353: (8)             y6 = repeat(x4, 2, axis=0)
354: (8)             assert_equal(y5, y6)
355: (8)             y7 = x4.repeat((2, 2, 2, 2), axis=0)
356: (8)             assert_equal(y5, y7)
357: (8)             y8 = x4.repeat(2, 0)
358: (8)             assert_equal(y5, y8)
359: (8)             y9 = x4.copy()
360: (8)             assert_equal(y9._data, x4._data)
361: (8)             assert_equal(y9._mask, x4._mask)
362: (8)             x = masked_array([1, 2, 3], mask=[0, 1, 0])
363: (8)             y = masked_array(x)
364: (8)             assert_equal(y._data.ctypes.data, x._data.ctypes.data)
365: (8)             assert_equal(y._mask.ctypes.data, x._mask.ctypes.data)
366: (8)             y = masked_array(x, copy=True)
367: (8)             assert_not_equal(y._data.ctypes.data, x._data.ctypes.data)
368: (8)             assert_not_equal(y._mask.ctypes.data, x._mask.ctypes.data)
369: (4)             def test_copy_0d(self):
370: (8)               x = np.ma.array(43, mask=True)
371: (8)               xc = x.copy()
372: (8)               assert_equal(xc.mask, True)
373: (4)               def test_copy_on_python_builtin(self):
374: (8)                 assert_(isMaskedArray(np.ma.copy([1, 2, 3])))
375: (8)                 assert_(isMaskedArray(np.ma.copy((1, 2, 3))))
376: (4)               def test_copy_immutable(self):
377: (8)                 a = np.ma.array([1, 2, 3])

```

```

378: (8)             b = np.ma.array([4, 5, 6])
379: (8)             a_copy_method = a.copy
380: (8)             b.copy
381: (8)             assert_equal(a_copy_method(), [1, 2, 3])
382: (4)             def test_deepcopy(self):
383: (8)                 from copy import deepcopy
384: (8)                 a = array([0, 1, 2], mask=[False, True, False])
385: (8)                 copied = deepcopy(a)
386: (8)                 assert_equal(copied.mask, a.mask)
387: (8)                 assert_not_equal(id(a._mask), id(copied._mask))
388: (8)                 copied[1] = 1
389: (8)                 assert_equal(copied.mask, [0, 0, 0])
390: (8)                 assert_equal(a.mask, [0, 1, 0])
391: (8)                 copied = deepcopy(a)
392: (8)                 assert_equal(copied.mask, a.mask)
393: (8)                 copied.mask[1] = False
394: (8)                 assert_equal(copied.mask, [0, 0, 0])
395: (8)                 assert_equal(a.mask, [0, 1, 0])
396: (4)             def test_format(self):
397: (8)                 a = array([0, 1, 2], mask=[False, True, False])
398: (8)                 assert_equal(format(a), "[0 -- 2]")
399: (8)                 assert_equal(format(masked), "--")
400: (8)                 assert_equal(format(masked, ""), "--")
401: (8)                 with assert_warnings(FutureWarning):
402: (12)                     with_format_string = format(masked, " >5")
403: (8)                     assert_equal(with_format_string, "--")
404: (4)             def test_str_repr(self):
405: (8)                 a = array([0, 1, 2], mask=[False, True, False])
406: (8)                 assert_equal(str(a), '[0 -- 2]')
407: (8)                 assert_equal(
408: (12)                     repr(a),
409: (12)                     textwrap.dedent('''\n'
410: (12)                         masked_array(data=[0, --, 2],
411: (25)                             mask=[False, True, False],
412: (19)                             fill_value=999999)'''')
413: (8)                 )
414: (8)                 a = np.ma.arange(2000)
415: (8)                 a[1:50] = np.ma.masked
416: (8)                 assert_equal(
417: (12)                     repr(a),
418: (12)                     textwrap.dedent('''\n'
419: (12)                         masked_array(data=[0, --, --, ..., 1997, 1998, 1999],
420: (25)                             mask=[False, True, True, ..., False, False, False],
421: (19)                             fill_value=999999)'''')
422: (8)                 )
423: (8)                 a = np.ma.arange(20)
424: (8)                 assert_equal(
425: (12)                     repr(a),
426: (12)                     textwrap.dedent('''\n'
427: (12)                         masked_array(data=[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
428: (31)                             12, 13,
429: (25)                             14, 15, 16, 17, 18, 19],
430: (19)                             mask=False,
431: (8)                             fill_value=999999)'''')
432: (8)                 )
433: (8)                 a = array([[1, 2, 3], [4, 5, 6]], dtype=np.int8)
434: (8)                 a[1,1] = np.ma.masked
435: (12)                 assert_equal(
436: (12)                     repr(a),
437: (12)                     textwrap.dedent('''\n'
438: (14)                         masked_array(
439: (20)                             data=[[1, 2, 3],
440: (14)                                 [4, --, 6]],
441: (20)                             mask=[[False, False, False],
442: (14)                                 [False, True, False]],
443: (14)                             fill_value=999999,
444: (8)                             dtype=int8)'''')
445: (8)                 )
446: (8)                 assert_equal(

```

```

446: (12)             repr(a[:1]),
447: (12)             textwrap.dedent('''\n
448: (12)             masked_array(data=[[1, 2, 3]],\n
449: (25)                 mask=[[False, False, False]],\n
450: (19)                 fill_value=999999,\n
451: (24)                 dtype=int8)''')
452: (8)
453: (8)
454: (12)             assert_equal(
455: (12)                 repr(a.astype(int)),\n
456: (12)                 textwrap.dedent('''\n
457: (14)                 masked_array(\n
458: (20)                     data=[[1, 2, 3],\n
459: (14)                         [4, --, 6]],\n
460: (20)                         mask=[[False, False, False],\n
461: (14)                             [False, True, False]],\n
462: (8)                             fill_value=999999)''')
463: (4)             )
464: (8)             def test_str_repr_legacy(self):
465: (8)                 oldopts = np.get_printoptions()
466: (8)                 np.set_printoptions(legacy='1.13')
467: (8)                 try:
468: (12)                     a = array([0, 1, 2], mask=[False, True, False])
469: (12)                     assert_equal(str(a), '[0 -- 2]')
470: (34)                     assert_equal(repr(a), 'masked_array(data = [0 -- 2],\n'
471: (34)                         'mask = [False True False],\n'
472: (12)                             'fill_value = 999999)\n')
473: (12)                     a = np.ma.arange(2000)
474: (12)                     a[1:50] = np.ma.masked
475: (12)                     assert_equal(
476: (16)                         repr(a),
477: (16)                         'masked_array(data = [0 -- -- ..., 1997 1998 1999],\n'
478: (16)                             'mask = [False True True ..., False False\n
479: (12)                                 'fill_value = 999999)\n'
480: (8)
481: (12)             finally:
482: (4)                 np.set_printoptions(**oldopts)
483: (8)             def test_0d_unicode(self):
484: (8)                 u = 'caf\xe9'
485: (8)                 utype = type(u)
486: (8)                 arr_nomask = np.ma.array(u)
487: (8)                 arr_masked = np.ma.array(u, mask=True)
488: (8)                 assert_equal(utype(arr_nomask), u)
489: (8)                 assert_equal(utype(arr_masked), '--')
490: (8)             def test_pickling(self):
491: (12)                 for dtype in (int, float, str, object):
492: (12)                     a = arange(10).astype(dtype)
493: (12)                     a.fill_value = 999
494: (21)                     masks = ([0, 0, 0, 1, 0, 1, 0, 1, 0, 1], # partially masked
495: (21)                                     True, # Fully masked
496: (12)                                     False) # Fully unmasked
497: (16)                     for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
498: (20)                         for mask in masks:
499: (20)                             a.mask = mask
500: (20)                             a_pickled = pickle.loads(pickle.dumps(a, protocol=proto))
501: (20)                             assert_equal(a_pickled._mask, a._mask)
502: (20)                             assert_equal(a_pickled._data, a._data)
503: (24)                             if dtype in (object, int):
504: (20)                                 assert_equal(a_pickled.fill_value, 999)
505: (24)                             else:
506: (20)                                 assert_equal(a_pickled.fill_value, dtype(999))
507: (4)                             assert_array_equal(a_pickled.mask, mask)
508: (8)             def test_pickling_subbaseclass(self):
509: (21)                 x = np.array([(1.0, 2), (3.0, 4)],\n
510: (8)                               dtype=[('x', float), ('y', int)]).view(np.recarray)
511: (8)                 a = masked_array(x, mask=[(True, False), (False, True)])
512: (12)                 for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
513: (12)                     a_pickled = pickle.loads(pickle.dumps(a, protocol=proto))
514: (12)                     assert_equal(a_pickled._mask, a._mask)

```

```

514: (12)             assert_equal(a_pickled, a)
515: (12)             assert_(isinstance(a_pickled._data, np.recarray))
516: (4)              def test_pickling_maskedconstant(self):
517: (8)                mc = np.ma.masked
518: (8)                for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
519: (12)                  mc_pickled = pickle.loads(pickle.dumps(mc, protocol=proto))
520: (12)                  assert_equal(mc_pickled._baseclass, mc._baseclass)
521: (12)                  assert_equal(mc_pickled._mask, mc._mask)
522: (12)                  assert_equal(mc_pickled._data, mc._data)
523: (4)              def test_pickling_wstructured(self):
524: (8)                a = array([(1, 1.), (2, 2.)], mask=[(0, 0), (0, 1)],
525: (18)                  dtype=[('a', int), ('b', float)])
526: (8)                for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
527: (12)                  a_pickled = pickle.loads(pickle.dumps(a, protocol=proto))
528: (12)                  assert_equal(a_pickled._mask, a._mask)
529: (12)                  assert_equal(a_pickled, a)
530: (4)              def test_pickling_keepalignment(self):
531: (8)                a = arange(10)
532: (8)                a.shape = (-1, 2)
533: (8)                b = a.T
534: (8)                for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
535: (12)                  test = pickle.loads(pickle.dumps(b, protocol=proto))
536: (12)                  assert_equal(test, b)
537: (4)              def test_single_element_subscript(self):
538: (8)                a = array([1, 3, 2])
539: (8)                b = array([1, 3, 2], mask=[1, 0, 1])
540: (8)                assert_equal(a[0].shape, ())
541: (8)                assert_equal(b[0].shape, ())
542: (8)                assert_equal(b[1].shape, ())
543: (4)              def test_topython(self):
544: (8)                assert_equal(1, int(array(1)))
545: (8)                assert_equal(1.0, float(array(1)))
546: (8)                assert_equal(1, int(array([[1]])))
547: (8)                assert_equal(1.0, float(array([[1]])))
548: (8)                assert_raises(TypeError, float, array([1, 1]))
549: (8)                with suppress_warnings() as sup:
550: (12)                  sup.filter(UserWarning, 'Warning: converting a masked element')
551: (12)                  assert_(np.isnan(float(array([1], mask=[1]))))
552: (12)                  a = array([1, 2, 3], mask=[1, 0, 0])
553: (12)                  assert_raises(TypeError, lambda: float(a))
554: (12)                  assert_equal(float(a[-1]), 3.)
555: (12)                  assert_(np.isnan(float(a[0])))
556: (8)                  assert_raises(TypeError, int, a)
557: (8)                  assert_equal(int(a[-1]), 3)
558: (8)                  assert_raises(MAError, lambda:int(a[0]))
559: (4)              def test_oddfeatures_1(self):
560: (8)                x = arange(20)
561: (8)                x = x.reshape(4, 5)
562: (8)                x.flat[5] = 12
563: (8)                assert_(x[1, 0] == 12)
564: (8)                z = x + 10j * x
565: (8)                assert_equal(z.real, x)
566: (8)                assert_equal(z.imag, 10 * x)
567: (8)                assert_equal((z * conjugate(z)).real, 101 * x * x)
568: (8)                z.imag[...] = 0.0
569: (8)                x = arange(10)
570: (8)                x[3] = masked
571: (8)                assert_(str(x[3]) == str(masked))
572: (8)                c = x >= 8
573: (8)                assert_(count(where(c, masked, masked)) == 0)
574: (8)                assert_(shape(where(c, masked, masked)) == c.shape)
575: (8)                z = masked_where(c, x)
576: (8)                assert_(z.dtype is x.dtype)
577: (8)                assert_(z[3] is masked)
578: (8)                assert_(z[4] is not masked)
579: (8)                assert_(z[7] is not masked)
580: (8)                assert_(z[8] is masked)
581: (8)                assert_(z[9] is masked)
582: (8)                assert_equal(x, z)

```

```

583: (4) def test_oddfeatures_2(self):
584: (8)     x = array([1., 2., 3., 4., 5.])
585: (8)     c = array([1, 1, 1, 0, 0])
586: (8)     x[2] = masked
587: (8)     z = where(c, x, -x)
588: (8)     assert_equal(z, [1., 2., 0., -4., -5])
589: (8)     c[0] = masked
590: (8)     z = where(c, x, -x)
591: (8)     assert_equal(z, [1., 2., 0., -4., -5])
592: (8)     assert_(z[0] is masked)
593: (8)     assert_(z[1] is not masked)
594: (8)     assert_(z[2] is masked)
595: (4) @suppress_copy_mask_on_assignment
596: (4) def test_oddfeatures_3(self):
597: (8)     atest = array([10], mask=True)
598: (8)     btest = array([20])
599: (8)     idx = atest.mask
600: (8)     atest[idx] = btest[idx]
601: (8)     assert_equal(atest, [20])
602: (4) def test_filled_with_object_dtype(self):
603: (8)     a = np.ma.masked_all(1, dtype='O')
604: (8)     assert_equal(a.filled('x')[0], 'x')
605: (4) def test_filled_with_flexible_dtype(self):
606: (8)     flexi = array([(1, 1, 1)],
607: (22)                 dtype=[('i', int), ('s', '|S8'), ('f', float)])
608: (8)     flexi[0] = masked
609: (8)     assert_equal(flexi.filled(),
610: (21)                 np.array([(default_fill_value(0),
611: (32)                               default_fill_value('0'),
612: (32)                               default_fill_value(0.))], dtype=flexi.dtype))
613: (8)     flexi[0] = masked
614: (8)     assert_equal(flexi.filled(1),
615: (21)                 np.array([(1, '1', 1.)], dtype=flexi.dtype))
616: (4) def test_filled_with_mvoid(self):
617: (8)     ndtype = [('a', int), ('b', float)]
618: (8)     a = mvoid((1, 2.), mask=[(0, 1)], dtype=ndtype)
619: (8)     test = a.filled()
620: (8)     assert_equal(tuple(test), (1, default_fill_value(1.)))
621: (8)     test = a.filled((-1, -1))
622: (8)     assert_equal(tuple(test), (1, -1))
623: (8)     a.fill_value = (-999, -999)
624: (8)     assert_equal(tuple(a.filled()), (1, -999))
625: (4) def test_filled_with_nested_dtype(self):
626: (8)     ndtype = [('A', int), ('B', [(('BA', int), ('BB', int))])]
627: (8)     a = array([(1, (1, 1)), (2, (2, 2))],
628: (18)                 mask=[(0, (1, 0)), (0, (0, 1))], dtype=ndtype)
629: (8)     test = a.filled(0)
630: (8)     control = np.array([(1, (0, 1)), (2, (2, 0))], dtype=ndtype)
631: (8)     assert_equal(test, control)
632: (8)     test = a['B'].filled(0)
633: (8)     control = np.array([(0, 1), (2, 0)], dtype=a['B'].dtype)
634: (8)     assert_equal(test, control)
635: (8)     Z = numpy.ma.zeros(2, numpy.dtype([('A", "(2,2)i1,(2,2)i1", (2,2)])))
636: (8)     assert_equal(Z.data.dtype, numpy.dtype([('A', [('f0', 'i1', (2, 2)),
637: (42)                               ('f1', 'i1', (2, 2))], (2, 2))]))
638: (8)     assert_equal(Z.mask.dtype, numpy.dtype([('A', [('f0', '?', (2, 2)),
639: (42)                               ('f1', '?', (2, 2))], (2, 2))]))
640: (4) def test_filled_with_f_order(self):
641: (8)     a = array(np.array([(0, 1, 2), (4, 5, 6)], order='F'),
642: (18)                 mask=np.array([(0, 0, 1), (1, 0, 0)], order='F'),
643: (18)                 order='F') # this is currently ignored
644: (8)     assert_(a.flags['F_CONTIGUOUS'])
645: (8)     assert_(a.filled(0).flags['F_CONTIGUOUS'])
646: (4) def test_optinfo_propagation(self):
647: (8)     x = array([1, 2, 3, ], dtype=float)
648: (8)     x._optinfo['info'] = '??'
649: (8)     y = x.copy()
650: (8)     assert_equal(y._optinfo['info'], '??')
651: (8)     y._optinfo['info'] = '!!!'

```

```

652: (8)             assert_equal(x._optinfo['info'], '???')
653: (4)             def test_optinfo_forward_propagation(self):
654: (8)                 a = array([1,2,2,4])
655: (8)                 a._optinfo["key"] = "value"
656: (8)                 assert_equal(a._optinfo["key"], (a == 2)._optinfo["key"])
657: (8)                 assert_equal(a._optinfo["key"], (a != 2)._optinfo["key"])
658: (8)                 assert_equal(a._optinfo["key"], (a > 2)._optinfo["key"])
659: (8)                 assert_equal(a._optinfo["key"], (a >= 2)._optinfo["key"])
660: (8)                 assert_equal(a._optinfo["key"], (a <= 2)._optinfo["key"])
661: (8)                 assert_equal(a._optinfo["key"], (a + 2)._optinfo["key"])
662: (8)                 assert_equal(a._optinfo["key"], (a - 2)._optinfo["key"])
663: (8)                 assert_equal(a._optinfo["key"], (a * 2)._optinfo["key"])
664: (8)                 assert_equal(a._optinfo["key"], (a / 2)._optinfo["key"])
665: (8)                 assert_equal(a._optinfo["key"], a[:2]._optinfo["key"])
666: (8)                 assert_equal(a._optinfo["key"], a[[0,0,2]]._optinfo["key"])
667: (8)                 assert_equal(a._optinfo["key"], np.exp(a)._optinfo["key"])
668: (8)                 assert_equal(a._optinfo["key"], np.abs(a)._optinfo["key"])
669: (8)                 assert_equal(a._optinfo["key"], array(a, copy=True)._optinfo["key"])
670: (8)                 assert_equal(a._optinfo["key"], np.zeros_like(a)._optinfo["key"])
671: (4)             def test_fancy_printoptions(self):
672: (8)                 fancydtype = np.dtype([('x', int), ('y', [(('t', int), ('s', float))])])
673: (8)                 test = array([(1, (2, 3.0)), (4, (5, 6.0))]),
674: (21)                   mask=[(1, (0, 1)), (0, (1, 0))],
675: (21)                   dtype=fancydtype)
676: (8)                 control = "[(--, (2, --)) (4, (--, 6.0))]"
677: (8)                 assert_equal(str(test), control)
678: (8)                 t_2d0 = masked_array(data = (0, [[0.0, 0.0, 0.0],
679: (40)                       [0.0, 0.0, 0.0]],
680: (36)                         0.0),
681: (29)                           mask = (False, [[True, False, True],
682: (45)                             [False, False, True]],
683: (37)                               False),
684: (29)                               dtype = "int, (2,3)float, float")
685: (8)                               control = "(0, [[--, 0.0, --], [0.0, 0.0, --]], 0.0)"
686: (8)                               assert_equal(str(t_2d0), control)
687: (4)             def test_flatten_structured_array(self):
688: (8)                 ndtype = [('a', int), ('b', float)]
689: (8)                 a = np.array([(1, 1), (2, 2)], dtype=ndtype)
690: (8)                 test = flatten_structured_array(a)
691: (8)                 control = np.array([[1., 1.], [2., 2.]], dtype=float)
692: (8)                 assert_equal(test, control)
693: (8)                 assert_equal(test.dtype, control.dtype)
694: (8)                 a = array([(1, 1), (2, 2)], mask=[(0, 1), (1, 0)], dtype=ndtype)
695: (8)                 test = flatten_structured_array(a)
696: (8)                 control = array([[1., 1.], [2., 2.]],
697: (24)                   mask=[[0, 1], [1, 0]], dtype=float)
698: (8)                 assert_equal(test, control)
699: (8)                 assert_equal(test.dtype, control.dtype)
700: (8)                 assert_equal(test.mask, control.mask)
701: (8)                 ndtype = [('a', int), ('b', [(('ba', int), ('bb', float))])]
702: (8)                 a = array([(1, (1, 1.1)), (2, (2, 2.2))],
703: (18)                   mask=[(0, (1, 0)), (1, (0, 1))], dtype=ndtype)
704: (8)                 test = flatten_structured_array(a)
705: (8)                 control = array([[1., 1., 1.1], [2., 2., 2.2]],
706: (24)                   mask=[[0, 1, 0], [1, 0, 1]], dtype=float)
707: (8)                 assert_equal(test, control)
708: (8)                 assert_equal(test.dtype, control.dtype)
709: (8)                 assert_equal(test.mask, control.mask)
710: (8)                 ndtype = [('a', int), ('b', float)]
711: (8)                 a = np.array([(1, 1), (2, 2)], dtype=ndtype)
712: (8)                 test = flatten_structured_array(a)
713: (8)                 control = np.array([[1., 1.], [2., 2.]], dtype=float)
714: (8)                 assert_equal(test, control)
715: (8)                 assert_equal(test.dtype, control.dtype)
716: (4)             def test_void0d(self):
717: (8)                 ndtype = [('a', int), ('b', int)]
718: (8)                 a = np.array([(1, 2)], dtype=ndtype)[0]
719: (8)                 f = mvoid(a)
720: (8)                 assert_(isinstance(f, mvoid))

```

```

721: (8)                                a = masked_array([(1, 2)], mask=[(1, 0)], dtype=ndtype)[0]
722: (8)                                assert_(isinstance(a, mvoid))
723: (8)                                a = masked_array([(1, 2), (1, 2)], mask=[(1, 0), (0, 0)],
724: (8)                                    dtype=ndtype)
725: (8)                                f = mvoid(a._data[0], a._mask[0])
726: (4)                                assert_(isinstance(f, mvoid))
727: (8)                                def test_mvoid_getitem(self):
728: (8)                                    ndtype = [('a', int), ('b', int)]
729: (25)                                   a = masked_array([(1, 2), (3, 4)], mask=[(0, 0), (1, 0)],
730: (8)                                         dtype=ndtype)
731: (8)                                   f = a[0]
732: (8)                                   assert_(isinstance(f, mvoid))
733: (8)                                   assert_equal((f[0], f['a']), (1, 1))
734: (8)                                   assert_equal(f['b'], 2)
735: (8)                                   f = a[1]
736: (8)                                   assert_(isinstance(f, mvoid))
737: (8)                                   assert_(f[0] is masked)
738: (8)                                   assert_(f['a'] is masked)
739: (8)                                   assert_equal(f[1], 4)
740: (25)                                  A = masked_array(data=[[0,1],]),
741: (25)                                      mask=[[True, False],),
742: (8)                                         dtype=[("A", ">i2", (2,))])
743: (8)                                   assert_equal(A[0]["A"], A["A"][0])
744: (25)                                   assert_equal(A[0]["A"], masked_array(data=[0, 1],
745: (4)                                         mask=[True, False], dtype=">i2"))
746: (8)                                def test_mvoid_iter(self):
747: (8)                                    ndtype = [('a', int), ('b', int)]
748: (25)                                   a = masked_array([(1, 2), (3, 4)], mask=[(0, 0), (1, 0)],
749: (8)                                         dtype=ndtype)
750: (8)                                   assert_equal(list(a[0]), [1, 2])
751: (4)                                   assert_equal(list(a[1]), [masked, 4])
752: (8)                                def test_mvoid_print(self):
753: (8)                                    mx = array([(1, 1), (2, 2)], dtype=[('a', int), ('b', int)])
754: (8)                                    assert_equal(str(mx[0]), "(1, 1)")
755: (8)                                    mx['b'][0] = masked
756: (8)                                    ini_display = masked_print_option._display
757: (8)                                    masked_print_option.set_display("-X-")
758: (12)                                   try:
759: (12)                                       assert_equal(str(mx[0]), "(1, -X-)")
760: (8)                                       assert_equal(repr(mx[0]), "(1, -X-)")
761: (12)                                   finally:
762: (8)                                       masked_print_option.set_display(ini_display)
763: (8)                                   mx = array([(1,), (2,)], dtype=[('a', 'O')])
764: (4)                                   assert_equal(str(mx[0]), "(1,)")
765: (8)                                def test_mvoid_multidim_print(self):
766: (28)                                   t_ma = masked_array(data = [[1, 2, 3],]),
767: (28)                                       mask = [[False, True, False],),
768: (28)                                       fill_value = ([999999, 999999, 999999],),
769: (8)                                         dtype = [('a', '<i4', (3,))])
770: (8)                                   assert_(str(t_ma[0]) == "[1, --, 3]")
771: (8)                                   assert_(repr(t_ma[0]) == "[1, --, 3]")
772: (28)                                   t_2d = masked_array(data = [[[1, 2], [3,4]]],),
773: (28)                                       mask = [[[False, True], [True, False]],),
774: (8)                                         dtype = [('a', '<i4', (2,2))])
775: (8)                                   assert_(str(t_2d[0]) == "[[1, --], [--, 4]]")
776: (8)                                   assert_(repr(t_2d[0]) == "[[1, --], [--, 4]]")
777: (28)                                   t_0d = masked_array(data = [(1,2)],
778: (28)                                       mask = [(True, False)],
779: (8)                                         dtype = [('a', '<i4'), ('b', '<i4')])
780: (8)                                   assert_(str(t_0d[0]) == "(--, 2)")
781: (8)                                   assert_(repr(t_0d[0]) == "(--, 2)")
782: (28)                                   t_2d = masked_array(data = [[[1, 2], [3,4]], 1]),
783: (28)                                       mask = [[[False, True], [True, False]], False]),
784: (8)                                         dtype = [('a', '<i4', (2,2)), ('b', float)])
785: (8)                                   assert_(str(t_2d[0]) == "[[1, --], [--, 4]], 1.0]")
786: (8)                                   assert_(repr(t_2d[0]) == "[[1, --], [--, 4]], 1.0]")
787: (28)                                   t_ne = masked_array(data=[(1, (1, 1))],
788: (28)                                       mask=[(True, (True, False))],
                                         dtype = [('a', '<i4'), ('b', 'i4,i4')])
```

```

789: (8)             assert_(str(t_ne[0]) == "(--, (--, 1))")
790: (8)             assert_(repr(t_ne[0]) == "(--, (--, 1))")
791: (4)             def test_object_with_array(self):
792: (8)                 mx1 = masked_array([1.], mask=[True])
793: (8)                 mx2 = masked_array([1., 2.])
794: (8)                 mx = masked_array([mx1, mx2], mask=[False, True], dtype=object)
795: (8)                 assert_(mx[0] is mx1)
796: (8)                 assert_(mx[1] is not mx2)
797: (8)                 assert_(np.all(mx[1].data == mx2.data))
798: (8)                 assert_(np.all(mx[1].mask))
799: (8)                 mx[1].data[0] = 0.
800: (8)                 assert_(mx2[0] == 0.)
801: (0)             class TestMaskedArrayArithmetic:
802: (4)                 def setup_method(self):
803: (8)                     x = np.array([1., 1., 1., -2., pi/2.0, 4., 5., -10., 10., 1., 2., 3.])
804: (8)                     y = np.array([5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.])
805: (8)                     a10 = 10.
806: (8)                     m1 = [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
807: (8)                     m2 = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1]
808: (8)                     xm = masked_array(x, mask=m1)
809: (8)                     ym = masked_array(y, mask=m2)
810: (8)                     z = np.array([-0.5, 0., 0.5, 0.8])
811: (8)                     zm = masked_array(z, mask=[0, 1, 0, 0])
812: (8)                     xf = np.where(m1, 1e+20, x)
813: (8)                     xm.set_fill_value(1e+20)
814: (8)                     self.d = (x, y, a10, m1, m2, xm, ym, z, zm, xf)
815: (8)                     self.err_status = np.geterr()
816: (8)                     np.seterr(divide='ignore', invalid='ignore')
817: (4)                 def teardown_method(self):
818: (8)                     np.seterr(**self.err_status)
819: (4)                 def test_basic_arithmetic(self):
820: (8)                     (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
821: (8)                     a2d = array([[1, 2], [0, 4]])
822: (8)                     a2dm = masked_array(a2d, [[0, 0], [1, 0]])
823: (8)                     assert_equal(a2d * a2d, a2d * a2dm)
824: (8)                     assert_equal(a2d + a2d, a2d + a2dm)
825: (8)                     assert_equal(a2d - a2d, a2d - a2dm)
826: (8)                     for s in [(12,), (4, 3), (2, 6)]:
827: (12)                         x = x.reshape(s)
828: (12)                         y = y.reshape(s)
829: (12)                         xm = xm.reshape(s)
830: (12)                         ym = ym.reshape(s)
831: (12)                         xf = xf.reshape(s)
832: (12)                         assert_equal(-x, -xm)
833: (12)                         assert_equal(x + y, xm + ym)
834: (12)                         assert_equal(x - y, xm - ym)
835: (12)                         assert_equal(x * y, xm * ym)
836: (12)                         assert_equal(x / y, xm / ym)
837: (12)                         assert_equal(a10 + y, a10 + ym)
838: (12)                         assert_equal(a10 - y, a10 - ym)
839: (12)                         assert_equal(a10 * y, a10 * ym)
840: (12)                         assert_equal(a10 / y, a10 / ym)
841: (12)                         assert_equal(x + a10, xm + a10)
842: (12)                         assert_equal(x - a10, xm - a10)
843: (12)                         assert_equal(x * a10, xm * a10)
844: (12)                         assert_equal(x / a10, xm / a10)
845: (12)                         assert_equal(x ** 2, xm ** 2)
846: (12)                         assert_equal(abs(x) ** 2.5, abs(xm) ** 2.5)
847: (12)                         assert_equal(x ** y, xm ** ym)
848: (12)                         assert_equal(np.add(x, y), add(xm, ym))
849: (12)                         assert_equal(np.subtract(x, y), subtract(xm, ym))
850: (12)                         assert_equal(np.multiply(x, y), multiply(xm, ym))
851: (12)                         assert_equal(np.divide(x, y), divide(xm, ym))
852: (4)                 def test_divide_on_different_shapes(self):
853: (8)                     x = arange(6, dtype=float)
854: (8)                     x.shape = (2, 3)
855: (8)                     y = arange(3, dtype=float)
856: (8)                     z = x / y
857: (8)                     assert_equal(z, [[-1., 1., 1.], [-1., 4., 2.5]])

```

```

858: (8)             assert_equal(z.mask, [[1, 0, 0], [1, 0, 0]])
859: (8)             z = x / y[None,:]
860: (8)             assert_equal(z, [[-1., 1., 1.], [-1., 4., 2.5]])
861: (8)             assert_equal(z.mask, [[1, 0, 0], [1, 0, 0]])
862: (8)             y = arange(2, dtype=float)
863: (8)             z = x / y[:, None]
864: (8)             assert_equal(z, [[-1., -1., -1.], [3., 4., 5.]])
865: (8)             assert_equal(z.mask, [[1, 1, 1], [0, 0, 0]])
866: (4)             def test_mixed_arithmetic(self):
867: (8)                 na = np.array([1])
868: (8)                 ma = array([1])
869: (8)                 assert_(isinstance(na + ma, MaskedArray))
870: (8)                 assert_(isinstance(ma + na, MaskedArray))
871: (4)             def test_limits_arithmetic(self):
872: (8)                 tiny = np.finfo(float).tiny
873: (8)                 a = array([tiny, 1. / tiny, 0.])
874: (8)                 assert_equal(getmaskarray(a / 2), [0, 0, 0])
875: (8)                 assert_equal(getmaskarray(2 / a), [1, 0, 1])
876: (4)             def test_masked_singleton_arithmetic(self):
877: (8)                 xm = array(0, mask=1)
878: (8)                 assert_((1 / array(0)).mask)
879: (8)                 assert_((1 + xm).mask)
880: (8)                 assert_((-xm).mask)
881: (8)                 assert_(maximum(xm, xm).mask)
882: (8)                 assert_(minimum(xm, xm).mask)
883: (4)             def test_masked_singleton_equality(self):
884: (8)                 a = array([1, 2, 3], mask=[1, 1, 0])
885: (8)                 assert_((a[0] == 0) is masked)
886: (8)                 assert_((a[0] != 0) is masked)
887: (8)                 assert_equal((a[-1] == 0), False)
888: (8)                 assert_equal((a[-1] != 0), True)
889: (4)             def test_arithmetic_with_masked_singleton(self):
890: (8)                 x = masked_array([1, 2])
891: (8)                 y = x * masked
892: (8)                 assert_equal(y.shape, x.shape)
893: (8)                 assert_equal(y._mask, [True, True])
894: (8)                 y = x[0] * masked
895: (8)                 assert_(y is masked)
896: (8)                 y = x + masked
897: (8)                 assert_equal(y.shape, x.shape)
898: (8)                 assert_equal(y._mask, [True, True])
899: (4)             def test_arithmetic_with_masked_singleton_on_1d_singleton(self):
900: (8)                 x = masked_array([1, ])
901: (8)                 y = x + masked
902: (8)                 assert_equal(y.shape, x.shape)
903: (8)                 assert_equal(y.mask, [True, ])
904: (4)             def test_scalar_arithmetic(self):
905: (8)                 x = array(0, mask=0)
906: (8)                 assert_equal(x.filled().ctypes.data, x.ctypes.data)
907: (8)                 xm = array((0, 0)) / 0.
908: (8)                 assert_equal(xm.shape, (2,))
909: (8)                 assert_equal(xm.mask, [1, 1])
910: (4)             def test_basic_ufuncs(self):
911: (8)                 (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
912: (8)                 assert_equal(np.cos(x), cos(xm))
913: (8)                 assert_equal(np.cosh(x), cosh(xm))
914: (8)                 assert_equal(np.sin(x), sin(xm))
915: (8)                 assert_equal(np.sinh(x), sinh(xm))
916: (8)                 assert_equal(np.tan(x), tan(xm))
917: (8)                 assert_equal(np.tanh(x), tanh(xm))
918: (8)                 assert_equal(np.sqrt(abs(x)), sqrt(xm))
919: (8)                 assert_equal(np.log(abs(x)), log(xm))
920: (8)                 assert_equal(np.log10(abs(x)), log10(xm))
921: (8)                 assert_equal(np.exp(x), exp(xm))
922: (8)                 assert_equal(np.arcsin(z), arcsin(zm))
923: (8)                 assert_equal(np.arccos(z), arccos(zm))
924: (8)                 assert_equal(np.arctan(z), arctan(zm))
925: (8)                 assert_equal(np.arctan2(x, y), arctan2(xm, ym))
926: (8)                 assert_equal(np.absolute(x), absolute(xm))

```

```

927: (8) assert_equal(np.angle(x + 1j*y), angle(xm + 1j*ym))
928: (8) assert_equal(np.angle(x + 1j*y, deg=True), angle(xm + 1j*ym,
929: (8) deg=True))
930: (8) assert_equal(np.equal(x, y), equal(xm, ym))
931: (8) assert_equal(np.not_equal(x, y), not_equal(xm, ym))
932: (8) assert_equal(np.less(x, y), less(xm, ym))
933: (8) assert_equal(np.greater(x, y), greater(xm, ym))
934: (8) assert_equal(np.less_equal(x, y), less_equal(xm, ym))
935: (8) assert_equal(np.greater_equal(x, y), greater_equal(xm, ym))
936: (4) assert_equal(np.conjugate(x), conjugate(xm))
937: (8) def test_count_func(self):
938: (8)     assert_equal(1, count(1))
939: (8)     assert_equal(0, array(1, mask=[1]))
940: (8)     ott = array([0., 1., 2., 3.], mask=[1, 0, 0, 0])
941: (8)     res = count(ott)
942: (8)     assert_(res.dtype.type is np.intp)
943: (8)     assert_equal(3, res)
944: (8)     ott = ott.reshape((2, 2))
945: (8)     res = count(ott)
946: (8)     assert_(res.dtype.type is np.intp)
947: (8)     assert_equal(3, res)
948: (8)     res = count(ott, 0)
949: (8)     assert_(isinstance(res, ndarray))
950: (8)     assert_equal([1, 2], res)
951: (8)     assert_(getmask(res) is nomask)
952: (8)     ott = array([0., 1., 2., 3.])
953: (8)     res = count(ott, 0)
954: (8)     assert_(isinstance(res, ndarray))
955: (8)     assert_(res.dtype.type is np.intp)
956: (4)     assert_raises(np.AxisError, ott.count, axis=1)
957: (8) def test_count_on_python_builtins(self):
958: (8)     assert_equal(3, count([1, 2, 3]))
959: (4)     assert_equal(2, count((1, 2)))
960: (8) def test_minmax_func(self):
961: (8)     (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
962: (8)     xr = np.ravel(x)
963: (8)     xmr = ravel(xm)
964: (8)     assert_equal(max(xr), maximum.reduce(xmr))
965: (8)     assert_equal(min(xr), minimum.reduce(xmr))
966: (8)     assert_equal(minimum([1, 2, 3], [4, 0, 9]), [1, 0, 3])
967: (8)     assert_equal(maximum([1, 2, 3], [4, 0, 9]), [4, 2, 9])
968: (8)     x = arange(5)
969: (8)     y = arange(5) - 2
970: (8)     x[3] = masked
971: (8)     y[0] = masked
972: (8)     assert_equal(minimum(x, y), where(less(x, y), x, y))
973: (8)     assert_equal(maximum(x, y), where(greater(x, y), x, y))
974: (8)     assert_(minimum.reduce(x) == 0)
975: (8)     assert_(maximum.reduce(x) == 4)
976: (8)     x = arange(4).reshape(2, 2)
977: (8)     x[-1, -1] = masked
978: (4)     assert_equal(maximum.reduce(x, axis=None), 2)
979: (8) def test_minimunmaximum_func(self):
980: (8)     a = np.ones((2, 2))
981: (8)     aminimum = minimum(a, a)
982: (8)     assert_(isinstance(aminimum, MaskedArray))
983: (8)     assert_equal(aminimum, np.minimum(a, a))
984: (8)     aminimum = minimum.outer(a, a)
985: (8)     assert_(isinstance(aminimum, MaskedArray))
986: (8)     assert_equal(aminimum, np.minimum.outer(a, a))
987: (8)     amaximum = maximum(a, a)
988: (8)     assert_(isinstance(amaximum, MaskedArray))
989: (8)     assert_equal(amaximum, np.maximum(a, a))
990: (8)     amaximum = maximum.outer(a, a)
991: (8)     assert_(isinstance(amaximum, MaskedArray))
992: (4)     assert_equal(amaximum, np.maximum.outer(a, a))
993: (8) def test_minmax_reduce(self):
994: (8)     a = array([1, 2, 3], mask=[False, False, False])

```

```

995: (8)             assert_equal(b, 3)
996: (4)             def test_minmax_funcs_with_output(self):
997: (8)                 mask = np.random.rand(12).round()
998: (8)                 xm = array(np.random.uniform(0, 10, 12), mask=mask)
999: (8)                 xm.shape = (3, 4)
1000: (8)                 for funcname in ('min', 'max'):
1001: (12)                     npfunc = getattr(np, funcname)
1002: (12)                     mafunc = getattr(numpy.ma.core, funcname)
1003: (12)                     nout = np.empty((4,), dtype=int)
1004: (12)                     try:
1005: (16)                         result = npfunc(xm, axis=0, out=nout)
1006: (12)                     except MaskError:
1007: (16)                         pass
1008: (12)                         nout = np.empty((4,), dtype=float)
1009: (12)                         result = npfunc(xm, axis=0, out=nout)
1010: (12)                         assert_(result is nout)
1011: (12)                         nout.fill(-999)
1012: (12)                         result = mafunc(xm, axis=0, out=nout)
1013: (12)                         assert_(result is nout)
1014: (4)             def test_minmax_methods(self):
1015: (8)                 (_, _, _, _, _, xm, _, _, _, _) = self.d
1016: (8)                 xm.shape = (xm.size,)
1017: (8)                 assert_equal(xm.max(), 10)
1018: (8)                 assert_(xm[0].max() is masked)
1019: (8)                 assert_(xm[0].max(0) is masked)
1020: (8)                 assert_(xm[0].max(-1) is masked)
1021: (8)                 assert_equal(xm.min(), -10.)
1022: (8)                 assert_(xm[0].min() is masked)
1023: (8)                 assert_(xm[0].min(0) is masked)
1024: (8)                 assert_(xm[0].min(-1) is masked)
1025: (8)                 assert_equal(xm.ptp(), 20.)
1026: (8)                 assert_(xm[0].ptp() is masked)
1027: (8)                 assert_(xm[0].ptp(0) is masked)
1028: (8)                 assert_(xm[0].ptp(-1) is masked)
1029: (8)                 x = array([1, 2, 3], mask=True)
1030: (8)                 assert_(x.min() is masked)
1031: (8)                 assert_(x.max() is masked)
1032: (8)                 assert_(x.ptp() is masked)
1033: (4)             def test_minmax_dtypes(self):
1034: (8)                 x = np.array([1., 1., 1., -2., pi/2.0, 4., 5., -10., 10., 1., 2., 3.])
1035: (8)                 a10 = 10.
1036: (8)                 an10 = -10.0
1037: (8)                 m1 = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
1038: (8)                 xm = masked_array(x, mask=m1)
1039: (8)                 xm.set_fill_value(1e+20)
1040: (8)                 float_dtotypes = [np.float16, np.float32, np.float64, np.longdouble,
1041: (24)                               np.complex64, np.complex128, np.clongdouble]
1042: (8)                 for float_dtype in float_dtotypes:
1043: (12)                     assert_equal(masked_array(x, mask=m1, dtype=float_dtype).max(),
1044: (25)                           float_dtype(a10))
1045: (12)                     assert_equal(masked_array(x, mask=m1, dtype=float_dtype).min(),
1046: (25)                           float_dtype(an10))
1047: (8)                     assert_equal(xm.min(), an10)
1048: (8)                     assert_equal(xm.max(), a10)
1049: (8)                     for float_dtype in float_dtotypes[:4]:
1050: (12)                         assert_equal(masked_array(x, mask=m1, dtype=float_dtype).max(),
1051: (25)                           float_dtype(a10))
1052: (12)                         assert_equal(masked_array(x, mask=m1, dtype=float_dtype).min(),
1053: (25)                           float_dtype(an10))
1054: (8)                     for float_dtype in float_dtotypes[-3:]:
1055: (12)                         ym = masked_array([1e20+1j, 1e20-2j, 1e20-1j], mask=[0, 1, 0],
1056: (26)                           dtype=float_dtype)
1057: (12)                         assert_equal(ym.min(), float_dtype(1e20-1j))
1058: (12)                         assert_equal(ym.max(), float_dtype(1e20+1j))
1059: (12)                         zm = masked_array([np.inf+2j, np.inf+3j, -np.inf-1j], mask=[0, 1,
0],
1060: (30)                           dtype=float_dtype)
1061: (12)                         assert_equal(zm.min(), float_dtype(-np.inf-1j))
1062: (12)                         assert_equal(zm.max(), float_dtype(np.inf+2j))

```

```

1063: (12)             cmax = np.inf - 1j * np.finfo(np.float64).max
1064: (12)             assert masked_array([-cmax, 0], mask=[0, 1]).max() == -cmax
1065: (12)             assert masked_array([cmax, 0], mask=[0, 1]).min() == cmax
1066: (4)              def test_addsumprod(self):
1067: (8)                (x, y, a10, m1, m2, xm, ym, zm, xf) = self.d
1068: (8)                assert_equal(np.add.reduce(x), add.reduce(x))
1069: (8)                assert_equal(np.add.accumulate(x), add.accumulate(x))
1070: (8)                assert_equal(4, sum(array(4), axis=0))
1071: (8)                assert_equal(4, sum(array(4), axis=0))
1072: (8)                assert_equal(np.sum(x, axis=0), sum(x, axis=0))
1073: (8)                assert_equal(np.sum(filled(xm, 0), axis=0), sum(xm, axis=0))
1074: (8)                assert_equal(np.sum(x, 0), sum(x, 0))
1075: (8)                assert_equal(np.prod(x, axis=0), product(x, axis=0))
1076: (8)                assert_equal(np.prod(x, 0), product(x, 0))
1077: (8)                assert_equal(np.prod(filled(xm, 1), axis=0), product(xm, axis=0))
1078: (8)                s = (3, 4)
1079: (8)                x.shape = y.shape = xm.shape = ym.shape = s
1080: (8)                if len(s) > 1:
1081: (12)                  assert_equal(np.concatenate((x, y), 1), concatenate((xm, ym), 1))
1082: (12)                  assert_equal(np.add.reduce(x, 1), add.reduce(x, 1))
1083: (12)                  assert_equal(np.sum(x, 1), sum(x, 1))
1084: (12)                  assert_equal(np.prod(x, 1), product(x, 1))
1085: (4)              def test_binops_d2D(self):
1086: (8)                a = array([[1.], [2.], [3.]], mask=[[False], [True], [True]])
1087: (8)                b = array([[2., 3.], [4., 5.], [6., 7.]])
1088: (8)                test = a * b
1089: (8)                control = array([[2., 3.], [2., 2.], [3., 3.]],
1090: (24)                  mask=[[0, 0], [1, 1], [1, 1]])
1091: (8)                assert_equal(test, control)
1092: (8)                assert_equal(test.data, control.data)
1093: (8)                assert_equal(test.mask, control.mask)
1094: (8)                test = b * a
1095: (8)                control = array([[2., 3.], [4., 5.], [6., 7.]],
1096: (24)                  mask=[[0, 0], [1, 1], [1, 1]])
1097: (8)                assert_equal(test, control)
1098: (8)                assert_equal(test.data, control.data)
1099: (8)                assert_equal(test.mask, control.mask)
1100: (8)                a = array([[1.], [2.], [3.]])
1101: (8)                b = array([[2., 3.], [4., 5.], [6., 7.]],
1102: (18)                  mask=[[0, 0], [0, 0], [0, 1]])
1103: (8)                test = a * b
1104: (8)                control = array([[2, 3], [8, 10], [18, 3]],
1105: (24)                  mask=[[0, 0], [0, 0], [0, 1]])
1106: (8)                assert_equal(test, control)
1107: (8)                assert_equal(test.data, control.data)
1108: (8)                assert_equal(test.mask, control.mask)
1109: (8)                test = b * a
1110: (8)                control = array([[2, 3], [8, 10], [18, 7]],
1111: (24)                  mask=[[0, 0], [0, 0], [0, 1]])
1112: (8)                assert_equal(test, control)
1113: (8)                assert_equal(test.data, control.data)
1114: (8)                assert_equal(test.mask, control.mask)
1115: (4)              def test_domainated_binops_d2D(self):
1116: (8)                a = array([[1.], [2.], [3.]], mask=[[False], [True], [True]])
1117: (8)                b = array([[2., 3.], [4., 5.], [6., 7.]])
1118: (8)                test = a / b
1119: (8)                control = array([[1. / 2., 1. / 3.], [2., 2.], [3., 3.]],
1120: (24)                  mask=[[0, 0], [1, 1], [1, 1]])
1121: (8)                assert_equal(test, control)
1122: (8)                assert_equal(test.data, control.data)
1123: (8)                assert_equal(test.mask, control.mask)
1124: (8)                test = b / a
1125: (8)                control = array([[2. / 1., 3. / 1.], [4., 5.], [6., 7.]],
1126: (24)                  mask=[[0, 0], [1, 1], [1, 1]])
1127: (8)                assert_equal(test, control)
1128: (8)                assert_equal(test.data, control.data)
1129: (8)                assert_equal(test.mask, control.mask)
1130: (8)                a = array([[1.], [2.], [3.]])
1131: (8)                b = array([[2., 3.], [4., 5.], [6., 7.]],
```

```

1132: (18)                         mask=[[0, 0], [0, 0], [0, 1]])
1133: (8)                          test = a / b
1134: (8)                          control = array([[1. / 2, 1. / 3], [2. / 4, 2. / 5], [3. / 6, 3]],
1135: (24)                                         mask=[[0, 0], [0, 0], [0, 1]])
1136: (8)                          assert_equal(test, control)
1137: (8)                          assert_equal(test.data, control.data)
1138: (8)                          assert_equal(test.mask, control.mask)
1139: (8)                          test = b / a
1140: (8)                          control = array([[2 / 1., 3 / 1.], [4 / 2., 5 / 2.], [6 / 3., 7]],
1141: (24)                                         mask=[[0, 0], [0, 0], [0, 1]])
1142: (8)                          assert_equal(test, control)
1143: (8)                          assert_equal(test.data, control.data)
1144: (8)                          assert_equal(test.mask, control.mask)
1145: (4)                           def test_noshrinking(self):
1146: (8)                             a = masked_array([1., 2., 3.], mask=[False, False, False],
1147: (25)                                         shrink=False)
1148: (8)                             b = a + 1
1149: (8)                             assert_equal(b.mask, [0, 0, 0])
1150: (8)                             a += 1
1151: (8)                             assert_equal(a.mask, [0, 0, 0])
1152: (8)                             b = a / 1.
1153: (8)                             assert_equal(b.mask, [0, 0, 0])
1154: (8)                             a /= 1.
1155: (8)                             assert_equal(a.mask, [0, 0, 0])
1156: (4)                           def test_ufunc_nomask(self):
1157: (8)                             m = np.ma.array([1])
1158: (8)                             assert_equal(np.true_divide(m, 5).mask.shape, ())
1159: (4)                           def test_noshink_on_creation(self):
1160: (8)                             a = np.ma.masked_values([1., 2.5, 3.1], 1.5, shrink=False)
1161: (8)                             assert_equal(a.mask, [0, 0, 0])
1162: (4)                           def test_mod(self):
1163: (8)                             (x, y, a10, m1, m2, xm, ym, z, zm, xf) = self.d
1164: (8)                             assert_equal(mod(x, y), mod(xm, ym))
1165: (8)                             test = mod(ym, xm)
1166: (8)                             assert_equal(test, np.mod(ym, xm))
1167: (8)                             assert_equal(test.mask, mask_or(xm.mask, ym.mask))
1168: (8)                             test = mod(xm, ym)
1169: (8)                             assert_equal(test, np.mod(xm, ym))
1170: (8)                             assert_equal(test.mask, mask_or(mask_or(xm.mask, ym.mask), (ym == 0)))
1171: (4)                           def test_TakeTransposeInnerOuter(self):
1172: (8)                             x = arange(24)
1173: (8)                             y = np.arange(24)
1174: (8)                             x[5:6] = masked
1175: (8)                             x = x.reshape(2, 3, 4)
1176: (8)                             y = y.reshape(2, 3, 4)
1177: (8)                             assert_equal(np.transpose(y, (2, 0, 1)), transpose(x, (2, 0, 1)))
1178: (8)                             assert_equal(np.take(y, (2, 0, 1), 1), take(x, (2, 0, 1), 1))
1179: (8)                             assert_equal(np.inner(filled(x, 0), filled(y, 0)),
1180: (21)                                         inner(x, y))
1181: (8)                             assert_equal(np.outer(filled(x, 0), filled(y, 0)),
1182: (21)                                         outer(x, y))
1183: (8)                             y = array(['abc', 1, 'def', 2, 3], object)
1184: (8)                             y[2] = masked
1185: (8)                             t = take(y, [0, 3, 4])
1186: (8)                             assert_(t[0] == 'abc')
1187: (8)                             assert_(t[1] == 2)
1188: (8)                             assert_(t[2] == 3)
1189: (4)                           def test_imag_real(self):
1190: (8)                             xx = array([1 + 10j, 20 + 2j], mask=[1, 0])
1191: (8)                             assert_equal(xx.imag, [10, 2])
1192: (8)                             assert_equal(xx.imag.filled(), [1e+20, 2])
1193: (8)                             assert_equal(xx.imag.dtype, xx._data.imag.dtype)
1194: (8)                             assert_equal(xx.real, [1, 20])
1195: (8)                             assert_equal(xx.real.filled(), [1e+20, 20])
1196: (8)                             assert_equal(xx.real.dtype, xx._data.real.dtype)
1197: (4)                           def test_methods_with_output(self):
1198: (8)                             xm = array(np.random.uniform(0, 10, 12)).reshape(3, 4)
1199: (8)                             xm[:, 0] = xm[0] = xm[-1, -1] = masked
1200: (8)                             funclist = ('sum', 'prod', 'var', 'std', 'max', 'min', 'ptp', 'mean', )

```

```

1201: (8)           for funcname in funclist:
1202: (12)             npfunc = getattr(np, funcname)
1203: (12)             xmmeth = getattr(xm, funcname)
1204: (12)             output = np.empty(4, dtype=float)
1205: (12)             output.fill(-9999)
1206: (12)             result = npfunc(xm, axis=0, out=output)
1207: (12)             assert_(result is output)
1208: (12)             assert_equal(result, xmmeth(axis=0, out=output))
1209: (12)             output = empty(4, dtype=int)
1210: (12)             result = xmmeth(axis=0, out=output)
1211: (12)             assert_(result is output)
1212: (12)             assert_(output[0] is masked)
1213: (4)           def test_eq_on_structured(self):
1214: (8)             ndtype = [('A', int), ('B', int)]
1215: (8)             a = array([(1, 1), (2, 2)], mask=[(0, 1), (0, 0)], dtype=ndtype)
1216: (8)             test = (a == a)
1217: (8)             assert_equal(test.data, [True, True])
1218: (8)             assert_equal(test.mask, [False, False])
1219: (8)             assert_(test.fill_value == True)
1220: (8)             test = (a == a[0])
1221: (8)             assert_equal(test.data, [True, False])
1222: (8)             assert_equal(test.mask, [False, False])
1223: (8)             assert_(test.fill_value == True)
1224: (8)             b = array([(1, 1), (2, 2)], mask=[(1, 0), (0, 0)], dtype=ndtype)
1225: (8)             test = (a == b)
1226: (8)             assert_equal(test.data, [False, True])
1227: (8)             assert_equal(test.mask, [True, False])
1228: (8)             assert_(test.fill_value == True)
1229: (8)             test = (a[0] == b)
1230: (8)             assert_equal(test.data, [False, False])
1231: (8)             assert_equal(test.mask, [True, False])
1232: (8)             assert_(test.fill_value == True)
1233: (8)             b = array([(1, 1), (2, 2)], mask=[(0, 1), (1, 0)], dtype=ndtype)
1234: (8)             test = (a == b)
1235: (8)             assert_equal(test.data, [True, True])
1236: (8)             assert_equal(test.mask, [False, False])
1237: (8)             assert_(test.fill_value == True)
1238: (8)             ndtype = [('A', int), ('B', [('BA', int), ('BB', int)])]
1239: (8)             a = array([(1, (1, 1)), (2, (2, 2))],
1240: (19)                 [(3, (3, 3)), (4, (4, 4))]],
1241: (18)                 mask=[[(0, (1, 0)), (0, (0, 1))],
1242: (24)                   [(1, (0, 0)), (1, (1, 1))]], dtype=ndtype)
1243: (8)             test = (a[0, 0] == a)
1244: (8)             assert_equal(test.data, [[True, False], [False, False]])
1245: (8)             assert_equal(test.mask, [[False, False], [False, True]])
1246: (8)             assert_(test.fill_value == True)
1247: (4)           def test_ne_on_structured(self):
1248: (8)             ndtype = [('A', int), ('B', int)]
1249: (8)             a = array([(1, 1), (2, 2)], mask=[(0, 1), (0, 0)], dtype=ndtype)
1250: (8)             test = (a != a)
1251: (8)             assert_equal(test.data, [False, False])
1252: (8)             assert_equal(test.mask, [False, False])
1253: (8)             assert_(test.fill_value == True)
1254: (8)             test = (a != a[0])
1255: (8)             assert_equal(test.data, [False, True])
1256: (8)             assert_equal(test.mask, [False, False])
1257: (8)             assert_(test.fill_value == True)
1258: (8)             b = array([(1, 1), (2, 2)], mask=[(1, 0), (0, 0)], dtype=ndtype)
1259: (8)             test = (a != b)
1260: (8)             assert_equal(test.data, [True, False])
1261: (8)             assert_equal(test.mask, [True, False])
1262: (8)             assert_(test.fill_value == True)
1263: (8)             test = (a[0] != b)
1264: (8)             assert_equal(test.data, [True, True])
1265: (8)             assert_equal(test.mask, [True, False])
1266: (8)             assert_(test.fill_value == True)
1267: (8)             b = array([(1, 1), (2, 2)], mask=[(0, 1), (1, 0)], dtype=ndtype)
1268: (8)             test = (a != b)
1269: (8)             assert_equal(test.data, [False, False])

```

```

1270: (8)             assert_equal(test.mask, [False, False])
1271: (8)             assert_(test.fill_value == True)
1272: (8)             ndtype = [('A', int), ('B', [(('BA', int), ('BB', int))])]
1273: (8)             a = array([[[(1, (1, 1)), (2, (2, 2))],
1274: (19)                         [(3, (3, 3)), (4, (4, 4))]],
1275: (18)                         mask=[[((0, (1, 0)), (0, (0, 1))),
1276: (24)                           [(1, (0, 0)), (1, (1, 1))]]], dtype=ndtype)
1277: (8)             test = (a[0, 0] != a)
1278: (8)             assert_equal(test.data, [[False, True], [True, True]])
1279: (8)             assert_equal(test.mask, [[False, False], [False, True]])
1280: (8)             assert_(test.fill_value == True)
1281: (4)             def test_eq_ne_structured_with_non_masked(self):
1282: (8)                 a = array([(1, 1), (2, 2), (3, 4)],
1283: (18)                     mask=[(0, 1), (0, 0), (1, 1)], dtype='i4,i4')
1284: (8)                 eq = a == a.data
1285: (8)                 ne = a.data != a
1286: (8)                 assert_(np.all(eq))
1287: (8)                 assert_(not np.any(ne))
1288: (8)                 expected_mask = a.mask == np.ones((), a.mask.dtype)
1289: (8)                 assert_array_equal(eq.mask, expected_mask)
1290: (8)                 assert_array_equal(ne.mask, expected_mask)
1291: (8)                 assert_equal(eq.data, [True, True, False])
1292: (8)                 assert_array_equal(eq.data, ~ne.data)
1293: (4)             def test_eq_ne_structured_extra(self):
1294: (8)                 dt = np.dtype('i4,i4')
1295: (8)                 for m1 in (mvoid((1, 2), mask=(0, 0), dtype=dt),
1296: (19)                               mvoid((1, 2), mask=(0, 1), dtype=dt),
1297: (19)                               mvoid((1, 2), mask=(1, 0), dtype=dt),
1298: (19)                               mvoid((1, 2), mask=(1, 1), dtype=dt)):
1299: (12)                   ma1 = m1.view(MaskedArray)
1300: (12)                   r1 = ma1.view('2i4')
1301: (12)                   for m2 in (np.array((1, 1), dtype=dt),
1302: (23)                         mvoid((1, 1), dtype=dt),
1303: (23)                         mvoid((1, 0), mask=(0, 1), dtype=dt),
1304: (23)                         mvoid((3, 2), mask=(0, 1), dtype=dt)):
1305: (16)                     ma2 = m2.view(MaskedArray)
1306: (16)                     r2 = ma2.view('2i4')
1307: (16)                     eq_expected = (r1 == r2).all()
1308: (16)                     assert_equal(m1 == m2, eq_expected)
1309: (16)                     assert_equal(m2 == m1, eq_expected)
1310: (16)                     assert_equal(ma1 == m2, eq_expected)
1311: (16)                     assert_equal(m1 == ma2, eq_expected)
1312: (16)                     assert_equal(ma1 == ma2, eq_expected)
1313: (16)                     el_by_el = [m1[name] == m2[name] for name in dt.names]
1314: (16)                     assert_equal(array(el_by_el, dtype=bool).all(), eq_expected)
1315: (16)                     ne_expected = (r1 != r2).any()
1316: (16)                     assert_equal(m1 != m2, ne_expected)
1317: (16)                     assert_equal(m2 != m1, ne_expected)
1318: (16)                     assert_equal(ma1 != m2, ne_expected)
1319: (16)                     assert_equal(m1 != ma2, ne_expected)
1320: (16)                     assert_equal(ma1 != ma2, ne_expected)
1321: (16)                     el_by_el = [m1[name] != m2[name] for name in dt.names]
1322: (16)                     assert_equal(array(el_by_el, dtype=bool).any(), ne_expected)
1323: (4)             @pytest.mark.parametrize('dt', ['S', 'U'])
1324: (4)             @pytest.mark.parametrize('fill', [None, 'A'])
1325: (4)             def test_eq_for_strings(self, dt, fill):
1326: (8)                 a = array(['a', 'b'], dtype=dt, mask=[0, 1], fill_value=fill)
1327: (8)                 test = (a == a)
1328: (8)                 assert_equal(test.data, [True, True])
1329: (8)                 assert_equal(test.mask, [False, True])
1330: (8)                 assert_(test.fill_value == True)
1331: (8)                 test = (a == a[0])
1332: (8)                 assert_equal(test.data, [True, False])
1333: (8)                 assert_equal(test.mask, [False, True])
1334: (8)                 assert_(test.fill_value == True)
1335: (8)                 b = array(['a', 'b'], dtype=dt, mask=[1, 0], fill_value=fill)
1336: (8)                 test = (a == b)
1337: (8)                 assert_equal(test.data, [False, False])
1338: (8)                 assert_equal(test.mask, [True, True])

```

```

1339: (8)             assert_(test.fill_value == True)
1340: (8)             test = (a[0] == b)
1341: (8)             assert_equal(test.data, [False, False])
1342: (8)             assert_equal(test.mask, [True, False])
1343: (8)             assert_(test.fill_value == True)
1344: (8)             test = (b == a[0])
1345: (8)             assert_equal(test.data, [False, False])
1346: (8)             assert_equal(test.mask, [True, False])
1347: (8)             assert_(test.fill_value == True)
1348: (4) @pytest.mark.parametrize('dt', ['S', 'U'])
1349: (4) @pytest.mark.parametrize('fill', [None, 'A'])
1350: (4) def test_ne_for_strings(self, dt, fill):
1351: (8)     a = array(['a', 'b'], dtype=dt, mask=[0, 1], fill_value=fill)
1352: (8)     test = (a != a)
1353: (8)     assert_equal(test.data, [False, False])
1354: (8)     assert_equal(test.mask, [False, True])
1355: (8)     assert_(test.fill_value == True)
1356: (8)     test = (a != a[0])
1357: (8)     assert_equal(test.data, [False, True])
1358: (8)     assert_equal(test.mask, [False, True])
1359: (8)     assert_(test.fill_value == True)
1360: (8)     b = array(['a', 'b'], dtype=dt, mask=[1, 0], fill_value=fill)
1361: (8)     test = (a != b)
1362: (8)     assert_equal(test.data, [True, True])
1363: (8)     assert_equal(test.mask, [True, True])
1364: (8)     assert_(test.fill_value == True)
1365: (8)     test = (a[0] != b)
1366: (8)     assert_equal(test.data, [True, True])
1367: (8)     assert_equal(test.mask, [True, False])
1368: (8)     assert_(test.fill_value == True)
1369: (8)     test = (b != a[0])
1370: (8)     assert_equal(test.data, [True, True])
1371: (8)     assert_equal(test.mask, [True, False])
1372: (8)     assert_(test.fill_value == True)
1373: (4) @pytest.mark.parametrize('dt1', num_dts, ids=num_ids)
1374: (4) @pytest.mark.parametrize('dt2', num_dts, ids=num_ids)
1375: (4) @pytest.mark.parametrize('fill', [None, 1])
1376: (4) def test_eq_for_numeric(self, dt1, dt2, fill):
1377: (8)     a = array([0, 1], dtype=dt1, mask=[0, 1], fill_value=fill)
1378: (8)     test = (a == a)
1379: (8)     assert_equal(test.data, [True, True])
1380: (8)     assert_equal(test.mask, [False, True])
1381: (8)     assert_(test.fill_value == True)
1382: (8)     test = (a == a[0])
1383: (8)     assert_equal(test.data, [True, False])
1384: (8)     assert_equal(test.mask, [False, True])
1385: (8)     assert_(test.fill_value == True)
1386: (8)     b = array([0, 1], dtype=dt2, mask=[1, 0], fill_value=fill)
1387: (8)     test = (a == b)
1388: (8)     assert_equal(test.data, [False, False])
1389: (8)     assert_equal(test.mask, [True, True])
1390: (8)     assert_(test.fill_value == True)
1391: (8)     test = (a[0] == b)
1392: (8)     assert_equal(test.data, [False, False])
1393: (8)     assert_equal(test.mask, [True, False])
1394: (8)     assert_(test.fill_value == True)
1395: (8)     test = (b == a[0])
1396: (8)     assert_equal(test.data, [False, False])
1397: (8)     assert_equal(test.mask, [True, False])
1398: (8)     assert_(test.fill_value == True)
1399: (4) @pytest.mark.parametrize("op", [operator.eq, operator.lt])
1400: (4) def test_eq_broadcast_with_unmasked(self, op):
1401: (8)     a = array([0, 1], mask=[0, 1])
1402: (8)     b = np.arange(10).reshape(5, 2)
1403: (8)     result = op(a, b)
1404: (8)     assert_(result.mask.shape == b.shape)
1405: (8)     assert_equal(result.mask, np.zeros(b.shape, bool) | a.mask)
1406: (4) @pytest.mark.parametrize("op", [operator.eq, operator.gt])
1407: (4) def test_comp_no_mask_not_broadcast(self, op):

```

```

1408: (8)         a = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
1409: (8)         result = op(a, 3)
1410: (8)         assert_(not result.mask.shape)
1411: (8)         assert_(result.mask is nomask)
1412: (4) @pytest.mark.parametrize('dt1', num_dts, ids=num_ids)
1413: (4) @pytest.mark.parametrize('dt2', num_dts, ids=num_ids)
1414: (4) @pytest.mark.parametrize('fill', [None, 1])
1415: (4) def test_ne_for_numeric(self, dt1, dt2, fill):
1416: (8)     a = array([0, 1], dtype=dt1, mask=[0, 1], fill_value=fill)
1417: (8)     test = (a != a)
1418: (8)     assert_equal(test.data, [False, False])
1419: (8)     assert_equal(test.mask, [False, True])
1420: (8)     assert_(test.fill_value == True)
1421: (8)     test = (a != a[0])
1422: (8)     assert_equal(test.data, [False, True])
1423: (8)     assert_equal(test.mask, [False, True])
1424: (8)     assert_(test.fill_value == True)
1425: (8)     b = array([0, 1], dtype=dt2, mask=[1, 0], fill_value=fill)
1426: (8)     test = (a != b)
1427: (8)     assert_equal(test.data, [True, True])
1428: (8)     assert_equal(test.mask, [True, True])
1429: (8)     assert_(test.fill_value == True)
1430: (8)     test = (a[0] != b)
1431: (8)     assert_equal(test.data, [True, True])
1432: (8)     assert_equal(test.mask, [True, False])
1433: (8)     assert_(test.fill_value == True)
1434: (8)     test = (b != a[0])
1435: (8)     assert_equal(test.data, [True, True])
1436: (8)     assert_equal(test.mask, [True, False])
1437: (8)     assert_(test.fill_value == True)
1438: (4) @pytest.mark.parametrize('dt1', num_dts, ids=num_ids)
1439: (4) @pytest.mark.parametrize('dt2', num_dts, ids=num_ids)
1440: (4) @pytest.mark.parametrize('fill', [None, 1])
1441: (4) @pytest.mark.parametrize('op',
1442: (12)             [operator.le, operator.lt, operator.ge, operator.gt])
1443: (4) def test_comparisons_for_numeric(self, op, dt1, dt2, fill):
1444: (8)     a = array([0, 1], dtype=dt1, mask=[0, 1], fill_value=fill)
1445: (8)     test = op(a, a)
1446: (8)     assert_equal(test.data, op(a._data, a._data))
1447: (8)     assert_equal(test.mask, [False, True])
1448: (8)     assert_(test.fill_value == True)
1449: (8)     test = op(a, a[0])
1450: (8)     assert_equal(test.data, op(a._data, a._data[0]))
1451: (8)     assert_equal(test.mask, [False, True])
1452: (8)     assert_(test.fill_value == True)
1453: (8)     b = array([0, 1], dtype=dt2, mask=[1, 0], fill_value=fill)
1454: (8)     test = op(a, b)
1455: (8)     assert_equal(test.data, op(a._data, b._data))
1456: (8)     assert_equal(test.mask, [True, True])
1457: (8)     assert_(test.fill_value == True)
1458: (8)     test = op(a[0], b)
1459: (8)     assert_equal(test.data, op(a._data[0], b._data))
1460: (8)     assert_equal(test.mask, [True, False])
1461: (8)     assert_(test.fill_value == True)
1462: (8)     test = op(b, a[0])
1463: (8)     assert_equal(test.data, op(b._data, a._data[0]))
1464: (8)     assert_equal(test.mask, [True, False])
1465: (8)     assert_(test.fill_value == True)
1466: (4) @pytest.mark.parametrize('op',
1467: (12)             [operator.le, operator.lt, operator.ge, operator.gt])
1468: (4) @pytest.mark.parametrize('fill', [None, "N/A"])
1469: (4) def test_comparisons_strings(self, op, fill):
1470: (8)     ma1 = masked_array(["a", "b", "cde"], mask=[0, 1, 0], fill_value=fill)
1471: (8)     ma2 = masked_array(["cde", "b", "a"], mask=[0, 1, 0], fill_value=fill)
1472: (8)     assert_equal(op(ma1, ma2)._data, op(ma1._data, ma2._data))
1473: (4) def test_eq_with_None(self):
1474: (8)     with suppress_warnings() as sup:
1475: (12)         sup.filter(FutureWarning, "Comparison to `None`")
1476: (12)         a = array([None, 1], mask=[0, 1])

```

```

1477: (12) assert_equal(a == None, array([True, False], mask=[0, 1]))
1478: (12) assert_equal(a.data == None, [True, False])
1479: (12) assert_equal(a != None, array([False, True], mask=[0, 1]))
1480: (12) a = array([None, 1], mask=False)
1481: (12) assert_equal(a == None, [True, False])
1482: (12) assert_equal(a != None, [False, True])
1483: (12) a = array([None, 2], mask=True)
1484: (12) assert_equal(a == None, array([False, True], mask=True))
1485: (12) assert_equal(a != None, array([True, False], mask=True))
1486: (12) a = masked
1487: (12) assert_equal(a == None, masked)
1488: (4) def test_eq_with_scalar(self):
1489: (8)     a = array(1)
1490: (8)     assert_equal(a == 1, True)
1491: (8)     assert_equal(a == 0, False)
1492: (8)     assert_equal(a != 1, False)
1493: (8)     assert_equal(a != 0, True)
1494: (8)     b = array(1, mask=True)
1495: (8)     assert_equal(b == 0, masked)
1496: (8)     assert_equal(b == 1, masked)
1497: (8)     assert_equal(b != 0, masked)
1498: (8)     assert_equal(b != 1, masked)
1499: (4) def test_eq_different_dimensions(self):
1500: (8)     m1 = array([1, 1], mask=[0, 1])
1501: (8)     for m2 in (array([[0, 1], [1, 2]]),
1502: (19)                 np.array([[0, 1], [1, 2]])):
1503: (12)         test = (m1 == m2)
1504: (12)         assert_equal(test.data, [[False, False],
1505: (37)                                         [True, False]])
1506: (12)         assert_equal(test.mask, [[False, True],
1507: (37)                                         [False, True]])
1508: (4) def test_numpyarithmetic(self):
1509: (8)     a = masked_array([-1, 0, 1, 2, 3], mask=[0, 0, 0, 0, 1])
1510: (8)     control = masked_array([np.nan, np.nan, 0, np.log(2), -1],
1511: (31)                     mask=[1, 1, 0, 0, 1])
1512: (8)     test = log(a)
1513: (8)     assert_equal(test, control)
1514: (8)     assert_equal(test.mask, control.mask)
1515: (8)     assert_equal(a.mask, [0, 0, 0, 0, 1])
1516: (8)     test = np.log(a)
1517: (8)     assert_equal(test, control)
1518: (8)     assert_equal(test.mask, control.mask)
1519: (8)     assert_equal(a.mask, [0, 0, 0, 0, 1])
1520: (0) class TestMaskedArrayAttributes:
1521: (4)     def test_keepmask(self):
1522: (8)         x = masked_array([1, 2, 3], mask=[1, 0, 0])
1523: (8)         mx = masked_array(x)
1524: (8)         assert_equal(mx.mask, x.mask)
1525: (8)         mx = masked_array(x, mask=[0, 1, 0], keep_mask=False)
1526: (8)         assert_equal(mx.mask, [0, 1, 0])
1527: (8)         mx = masked_array(x, mask=[0, 1, 0], keep_mask=True)
1528: (8)         assert_equal(mx.mask, [1, 1, 0])
1529: (8)         mx = masked_array(x, mask=[0, 1, 0])
1530: (8)         assert_equal(mx.mask, [1, 1, 0])
1531: (4)     def test_hardmask(self):
1532: (8)         d = arange(5)
1533: (8)         n = [0, 0, 0, 1, 1]
1534: (8)         m = make_mask(n)
1535: (8)         xh = array(d, mask=m, hard_mask=True)
1536: (8)         xs = array(d, mask=m, hard_mask=False, copy=True)
1537: (8)         xh[[1, 4]] = [10, 40]
1538: (8)         xs[[1, 4]] = [10, 40]
1539: (8)         assert_equal(xh._data, [0, 10, 2, 3, 4])
1540: (8)         assert_equal(xs._data, [0, 10, 2, 3, 40])
1541: (8)         assert_equal(xs.mask, [0, 0, 0, 1, 0])
1542: (8)         assert_(xh._hardmask)
1543: (8)         assert_(not xs._hardmask)
1544: (8)         xh[1:4] = [10, 20, 30]
1545: (8)         xs[1:4] = [10, 20, 30]

```

```

1546: (8) assert_equal(xh._data, [0, 10, 20, 3, 4])
1547: (8) assert_equal(xs._data, [0, 10, 20, 30, 40])
1548: (8) assert_equal(xs.mask, nomask)
1549: (8) xh[0] = masked
1550: (8) xs[0] = masked
1551: (8) assert_equal(xh.mask, [1, 0, 0, 1, 1])
1552: (8) assert_equal(xs.mask, [1, 0, 0, 0, 0])
1553: (8) xh[:] = 1
1554: (8) xs[:] = 1
1555: (8) assert_equal(xh._data, [0, 1, 1, 3, 4])
1556: (8) assert_equal(xs._data, [1, 1, 1, 1, 1])
1557: (8) assert_equal(xh.mask, [1, 0, 0, 1, 1])
1558: (8) assert_equal(xs.mask, nomask)
1559: (8) xh.soften_mask()
1560: (8) xh[:] = arange(5)
1561: (8) assert_equal(xh._data, [0, 1, 2, 3, 4])
1562: (8) assert_equal(xh.mask, nomask)
1563: (8) xh.harden_mask()
1564: (8) xh[xh < 3] = masked
1565: (8) assert_equal(xh._data, [0, 1, 2, 3, 4])
1566: (8) assert_equal(xh._mask, [1, 1, 1, 0, 0])
1567: (8) xh[filled(xh > 1, False)] = 5
1568: (8) assert_equal(xh._data, [0, 1, 2, 5, 5])
1569: (8) assert_equal(xh._mask, [1, 1, 1, 0, 0])
1570: (8) xh = array([[1, 2], [3, 4]], mask=[[1, 0], [0, 0]], hard_mask=True)
1571: (8) xh[0] = 0
1572: (8) assert_equal(xh._data, [[1, 0], [3, 4]])
1573: (8) assert_equal(xh._mask, [[1, 0], [0, 0]])
1574: (8) xh[-1, -1] = 5
1575: (8) assert_equal(xh._data, [[1, 0], [3, 5]])
1576: (8) assert_equal(xh._mask, [[1, 0], [0, 0]])
1577: (8) xh[filled(xh < 5, False)] = 2
1578: (8) assert_equal(xh._data, [[1, 2], [2, 5]])
1579: (8) assert_equal(xh._mask, [[1, 0], [0, 0]])
1580: (4) def test_hardmask_again(self):
1581: (8) d = arange(5)
1582: (8) n = [0, 0, 0, 1, 1]
1583: (8) m = make_mask(n)
1584: (8) xh = array(d, mask=m, hard_mask=True)
1585: (8) xh[4:5] = 999
1586: (8) xh[0:1] = 999
1587: (8) assert_equal(xh._data, [999, 1, 2, 3, 4])
1588: (4) def test_hardmask_oncemore_yay(self):
1589: (8) a = array([1, 2, 3], mask=[1, 0, 0])
1590: (8) b = a.harden_mask()
1591: (8) assert_equal(a, b)
1592: (8) b[0] = 0
1593: (8) assert_equal(a, b)
1594: (8) assert_equal(b, array([1, 2, 3], mask=[1, 0, 0]))
1595: (8) a = b.soften_mask()
1596: (8) a[0] = 0
1597: (8) assert_equal(a, b)
1598: (8) assert_equal(b, array([0, 2, 3], mask=[0, 0, 0]))
1599: (4) def test_smallmask(self):
1600: (8) a = arange(10)
1601: (8) a[1] = masked
1602: (8) a[1] = 1
1603: (8) assert_equal(a._mask, nomask)
1604: (8) a = arange(10)
1605: (8) a._smallmask = False
1606: (8) a[1] = masked
1607: (8) a[1] = 1
1608: (8) assert_equal(a._mask, zeros(10))
1609: (4) def test_shrink_mask(self):
1610: (8) a = array([1, 2, 3], mask=[0, 0, 0])
1611: (8) b = a.shrink_mask()
1612: (8) assert_equal(a, b)
1613: (8) assert_equal(a.mask, nomask)
1614: (8) a = np.ma.array([(1, 2.0)], [(‘a’, int), (‘b’, float)])

```

```

1615: (8)             b = a.copy()
1616: (8)             a.shrink_mask()
1617: (8)             assert_equal(a.mask, b.mask)
1618: (4)             def test_flat(self):
1619: (8)                 x = array([(1, 1.1, 'one'), (2, 2.2, 'two'), (3, 3.3, 'thr')],
1620: (19)                         [(4, 4.4, 'fou'), (5, 5.5, 'fiv'), (6, 6.6, 'six')]], 
1621: (18)                         dtype=[('a', int), ('b', float), ('c', '|S8')])
1622: (8)                 x['a'][0, 1] = masked
1623: (8)                 x['b'][1, 0] = masked
1624: (8)                 x['c'][0, 2] = masked
1625: (8)                 x[-1, -1] = masked
1626: (8)                 xflat = x.flat
1627: (8)                 assert_equal(xflat[0], x[0, 0])
1628: (8)                 assert_equal(xflat[1], x[0, 1])
1629: (8)                 assert_equal(xflat[2], x[0, 2])
1630: (8)                 assert_equal(xflat[:3], x[0])
1631: (8)                 assert_equal(xflat[3], x[1, 0])
1632: (8)                 assert_equal(xflat[4], x[1, 1])
1633: (8)                 assert_equal(xflat[5], x[1, 2])
1634: (8)                 assert_equal(xflat[3:], x[1])
1635: (8)                 assert_equal(xflat[-1], x[-1, -1])
1636: (8)                 i = 0
1637: (8)                 j = 0
1638: (8)                 for xf in xflat:
1639: (12)                     assert_equal(xf, x[j, i])
1640: (12)                     i += 1
1641: (12)                     if i >= x.shape[-1]:
1642: (16)                         i = 0
1643: (16)                         j += 1
1644: (4)             def test_assign_dtype(self):
1645: (8)                 a = np.zeros(4, dtype='f4,i4')
1646: (8)                 m = np.ma.array(a)
1647: (8)                 m.dtype = np.dtype('f4')
1648: (8)                 repr(m) # raises?
1649: (8)                 assert_equal(m.dtype, np.dtype('f4'))
1650: (8)             def assign():
1651: (12)                 m = np.ma.array(a)
1652: (12)                 m.dtype = np.dtype('f8')
1653: (8)                 assert_raises(ValueError, assign)
1654: (8)                 b = a.view(dtype='f4', type=np.ma.MaskedArray) # raises?
1655: (8)                 assert_equal(b.dtype, np.dtype('f4'))
1656: (8)                 a = np.zeros(4, dtype='f4')
1657: (8)                 m = np.ma.array(a)
1658: (8)                 m.dtype = np.dtype('f4,i4')
1659: (8)                 assert_equal(m.dtype, np.dtype('f4,i4'))
1660: (8)                 assert_equal(m._mask, np.ma.nomask)
1661: (0)             class TestFillingValues:
1662: (4)                 def test_check_on_scalar(self):
1663: (8)                     _check_fill_value = np.ma.core._check_fill_value
1664: (8)                     fval = _check_fill_value(0, int)
1665: (8)                     assert_equal(fval, 0)
1666: (8)                     fval = _check_fill_value(None, int)
1667: (8)                     assert_equal(fval, default_fill_value(0))
1668: (8)                     fval = _check_fill_value(0, "|S3")
1669: (8)                     assert_equal(fval, b"0")
1670: (8)                     fval = _check_fill_value(None, "|S3")
1671: (8)                     assert_equal(fval, default_fill_value(b"camelot!"))
1672: (8)                     assert_raises(TypeError, _check_fill_value, 1e+20, int)
1673: (8)                     assert_raises(TypeError, _check_fill_value, 'stuff', int)
1674: (4)                 def test_check_on_fields(self):
1675: (8)                     _check_fill_value = np.ma.core._check_fill_value
1676: (8)                     ndtype = [('a', int), ('b', float), ('c', '|S3')]
1677: (8)                     fval = _check_fill_value([-999, -12345678.9, "???"], ndtype)
1678: (8)                     assert_(isinstance(fval, ndarray))
1679: (8)                     assert_equal(fval.item(), [-999, -12345678.9, b"???"])
1680: (8)                     fval = _check_fill_value(None, ndtype)
1681: (8)                     assert_(isinstance(fval, ndarray))
1682: (8)                     assert_equal(fval.item(), [default_fill_value(0),
1683: (35)                                     default_fill_value(0.,),

```

```

1684: (35)                                         asbytes(default_fill_value("0"))])
1685: (8)                                          fill_val = np.array((-999, -12345678.9, "???"), dtype=ndtype)
1686: (8)                                          fval = _check_fill_value(fill_val, ndtype)
1687: (8)                                          assert_(isinstance(fval, ndarray))
1688: (8)                                          assert_equal(fval.item(), [-999, -12345678.9, b"???"])
1689: (8)                                          fill_val = np.array((-999, -12345678.9, "???"),
1690: (28)                                              dtype=[("A", int), ("B", float), ("C", "|S3")])
1691: (8)                                          fval = _check_fill_value(fill_val, ndtype)
1692: (8)                                          assert_(isinstance(fval, ndarray))
1693: (8)                                          assert_equal(fval.item(), [-999, -12345678.9, b"???"])
1694: (8)                                          fill_val = np.ndarray(shape=(1,), dtype=object)
1695: (8)                                          fill_val[0] = (-999, -12345678.9, b"???")
1696: (8)                                          fval = _check_fill_value(fill_val, object)
1697: (8)                                          assert_(isinstance(fval, ndarray))
1698: (8)                                          assert_equal(fval.item(), [-999, -12345678.9, b"???"])
1699: (8)                                          ndtype = [("a", int)]
1700: (8)                                          fval = _check_fill_value(-999999999, ndtype)
1701: (8)                                          assert_(isinstance(fval, ndarray))
1702: (8)                                          assert_equal(fval.item(), (-999999999,))
1703: (4) def test_fillvalue_conversion(self):
1704: (8)     a = array([b'3', b'4', b'5'])
1705: (8)     a._optinfo.update({'comment':'updated!"'})
1706: (8)     b = array(a, dtype=int)
1707: (8)     assert_equal(b._data, [3, 4, 5])
1708: (8)     assert_equal(b.fill_value, default_fill_value(0))
1709: (8)     b = array(a, dtype=float)
1710: (8)     assert_equal(b._data, [3, 4, 5])
1711: (8)     assert_equal(b.fill_value, default_fill_value(0.))
1712: (8)     b = a.astype(int)
1713: (8)     assert_equal(b._data, [3, 4, 5])
1714: (8)     assert_equal(b.fill_value, default_fill_value(0))
1715: (8)     assert_equal(b._optinfo['comment'], "updated!")
1716: (8)     b = a.astype([('a', '|S3')])
1717: (8)     assert_equal(b['a']._data, a._data)
1718: (8)     assert_equal(b['a'].fill_value, a.fill_value)
1719: (4) def test_default_fill_value(self):
1720: (8)     f1 = default_fill_value(1.)
1721: (8)     f2 = default_fill_value(np.array(1.))
1722: (8)     f3 = default_fill_value(np.array(1.).dtype)
1723: (8)     assert_equal(f1, f2)
1724: (8)     assert_equal(f1, f3)
1725: (4) def test_default_fill_value_structured(self):
1726: (8)     fields = array([(1, 1, 1)],
1727: (22)                                     dtype=[('i', int), ('s', '|S8'), ('f', float)])
1728: (8)     f1 = default_fill_value(fields)
1729: (8)     f2 = default_fill_value(fields.dtype)
1730: (8)     expected = np.array((default_fill_value(0),
1731: (29)                                     default_fill_value('0'),
1732: (29)                                     default_fill_value(0.)), dtype=fields.dtype)
1733: (8)     assert_equal(f1, expected)
1734: (8)     assert_equal(f2, expected)
1735: (4) def test_default_fill_value_void(self):
1736: (8)     dt = np.dtype([('v', 'V7')])
1737: (8)     f = default_fill_value(dt)
1738: (8)     assert_equal(f['v'], np.array(default_fill_value(dt['v']), dt['v']))
1739: (4) def test_fillvalue(self):
1740: (8)     data = masked_array([1, 2, 3], fill_value=-999)
1741: (8)     series = data[[0, 2, 1]]
1742: (8)     assert_equal(series._fill_value, data._fill_value)
1743: (8)     mtype = [('f', float), ('s', '|S3')]
1744: (8)     x = array([(1, 'a'), (2, 'b'), (pi, 'pi')], dtype=mtype)
1745: (8)     x.fill_value = 999
1746: (8)     assert_equal(x.fill_value.item(), [999., b'999'])
1747: (8)     assert_equal(x['f'].fill_value, 999)
1748: (8)     assert_equal(x['s'].fill_value, b'999')
1749: (8)     x.fill_value = (9, '???')
1750: (8)     assert_equal(x.fill_value.item(), (9, b'???''))
1751: (8)     assert_equal(x['f'].fill_value, 9)
1752: (8)     assert_equal(x['s'].fill_value, b'???'')

```

```

1753: (8)             x = array([1, 2, 3.1])
1754: (8)             x.fill_value = 999
1755: (8)             assert_equal(np.asarray(x.fill_value).dtype, float)
1756: (8)             assert_equal(x.fill_value, 999.)
1757: (8)             assert_equal(x._fill_value, np.array(999.))
1758: (4)              def test_subarray_fillvalue(self):
1759: (8)                  fields = array([(1, 1, 1)],
1760: (22)                      dtype=[('i', int), ('s', '|S8'), ('f', float)])
1761: (8)                  with suppress_warnings() as sup:
1762: (12)                      sup.filter(FutureWarning, "NumPy has detected")
1763: (12)                      subfields = fields[['i', 'f']]
1764: (12)                      assert_equal(tuple(subfields.fill_value), (999999, 1.e+20))
1765: (12)                      subfields[1:] == subfields[:-1]
1766: (4)              def test_fillvalue_exotic_dtype(self):
1767: (8)                  _check_fill_value = np.ma.core._check_fill_value
1768: (8)                  ndtype = [('i', int), ('s', '|S8'), ('f', float)]
1769: (8)                  control = np.array((default_fill_value(0),
1770: (28)                      default_fill_value('0'),
1771: (28)                      default_fill_value(0.),),
1772: (27)                          dtype=ndtype)
1773: (8)                  assert_equal(_check_fill_value(None, ndtype), control)
1774: (8)                  ndtype = [('f0', float, (2, 2))]
1775: (8)                  control = np.array((default_fill_value(0.),),
1776: (27)                      dtype=[('f0', float)]).astype(ndtype)
1777: (8)                  assert_equal(_check_fill_value(None, ndtype), control)
1778: (8)                  control = np.array((0,), dtype=[('f0', float)]).astype(ndtype)
1779: (8)                  assert_equal(_check_fill_value(0, ndtype), control)
1780: (8)                  ndtype = np.dtype("int, (2,3)float, float")
1781: (8)                  control = np.array((default_fill_value(0),
1782: (28)                      default_fill_value(0.),
1783: (28)                      default_fill_value(0.),),
1784: (27)                          dtype="int, float, float").astype(ndtype)
1785: (8)                  test = _check_fill_value(None, ndtype)
1786: (8)                  assert_equal(test, control)
1787: (8)                  control = np.array((0, 0, 0), dtype="int, float,
float").astype(ndtype)
1788: (8)                  assert_equal(_check_fill_value(0, ndtype), control)
1789: (8)                  M = masked_array(control)
1790: (8)                  assert_equal(M["f1"].fill_value.ndim, 0)
1791: (4)              def test_fillvalue_datetime_timedelta(self):
1792: (8)                  for timecode in ("as", "fs", "ps", "ns", "us", "ms", "s", "m",
1793: (25)                      "h", "D", "W", "M", "Y"):
1794: (12)                      control = numpy.datetime64("NaT", timecode)
1795: (12)                      test = default_fill_value(numpy.dtype("<M8[" + timecode + "]"))
1796: (12)                      np.testing.assert_equal(test, control)
1797: (12)                      control = numpy.timedelta64("NaT", timecode)
1798: (12)                      test = default_fill_value(numpy.dtype("<m8[" + timecode + "]"))
1799: (12)                      np.testing.assert_equal(test, control)
1800: (4)              def test_extremum_fill_value(self):
1801: (8)                  a = array([(1, (2, 3)), (4, (5, 6))],
1802: (18)                      dtype=[('A', int), ('B', [('BA', int), ('BB', int)])])
1803: (8)                  test = a.fill_value
1804: (8)                  assert_equal(test.dtype, a.dtype)
1805: (8)                  assert_equal(test['A'], default_fill_value(a['A']))
1806: (8)                  assert_equal(test['B']['BA'], default_fill_value(a['B']['BA']))
1807: (8)                  assert_equal(test['B']['BB'], default_fill_value(a['B']['BB']))
1808: (8)                  test = minimum_fill_value(a)
1809: (8)                  assert_equal(test.dtype, a.dtype)
1810: (8)                  assert_equal(test[0], minimum_fill_value(a['A']))
1811: (8)                  assert_equal(test[1][0], minimum_fill_value(a['B']['BA']))
1812: (8)                  assert_equal(test[1][1], minimum_fill_value(a['B']['BB']))
1813: (8)                  assert_equal(test[1], minimum_fill_value(a['B']))
1814: (8)                  test = maximum_fill_value(a)
1815: (8)                  assert_equal(test.dtype, a.dtype)
1816: (8)                  assert_equal(test[0], maximum_fill_value(a['A']))
1817: (8)                  assert_equal(test[1][0], maximum_fill_value(a['B']['BA']))
1818: (8)                  assert_equal(test[1][1], maximum_fill_value(a['B']['BB']))
1819: (8)                  assert_equal(test[1], maximum_fill_value(a['B']))
1820: (4)              def test_extremum_fill_value_subdtype(self):

```

```

1821: (8)         a = array(([2, 3, 4],), dtype=[('value', np.int8, 3)])
1822: (8)         test = minimum_fill_value(a)
1823: (8)         assert_equal(test.dtype, a.dtype)
1824: (8)         assert_equal(test[0], np.full(3, minimum_fill_value(a['value'])))
1825: (8)         test = maximum_fill_value(a)
1826: (8)         assert_equal(test.dtype, a.dtype)
1827: (8)         assert_equal(test[0], np.full(3, maximum_fill_value(a['value'])))
1828: (4)         def test_fillvalue_individual_fields(self):
1829: (8)             ndtype = [('a', int), ('b', int)]
1830: (8)             a = array(list(zip([1, 2, 3], [4, 5, 6])),
1831: (18)                 fill_value=(-999, -999), dtype=ndtype)
1832: (8)             aa = a['a']
1833: (8)             aa.set_fill_value(10)
1834: (8)             assert_equal(aa._fill_value, np.array(10))
1835: (8)             assert_equal(tuple(a.fill_value), (10, -999))
1836: (8)             a.fill_value['b'] = -10
1837: (8)             assert_equal(tuple(a.fill_value), (10, -10))
1838: (8)             t = array(list(zip([1, 2, 3], [4, 5, 6])), dtype=ndtype)
1839: (8)             tt = t['a']
1840: (8)             tt.set_fill_value(10)
1841: (8)             assert_equal(tt._fill_value, np.array(10))
1842: (8)             assert_equal(tuple(t.fill_value), (10, default_fill_value(0)))
1843: (4)         def test_fillvalue_implicit_structured_array(self):
1844: (8)             ndtype = ('b', float)
1845: (8)             adtype = ('a', float)
1846: (8)             a = array([(1.,), (2.,)], mask=[(False,), (False,)],
1847: (18)                 fill_value=(np.nan,), dtype=np.dtype([adtype]))
1848: (8)             b = empty(a.shape, dtype=[adtype, ndtype])
1849: (8)             b['a'] = a['a']
1850: (8)             b['a'].set_fill_value(a['a'].fill_value)
1851: (8)             f = b._fill_value[()]
1852: (8)             assert_(np.isnan(f[0]))
1853: (8)             assert_equal(f[-1], default_fill_value(1.))
1854: (4)         def test_fillvalue_as_arguments(self):
1855: (8)             a = empty(3, fill_value=999.)
1856: (8)             assert_equal(a.fill_value, 999.)
1857: (8)             a = ones(3, fill_value=999., dtype=float)
1858: (8)             assert_equal(a.fill_value, 999.)
1859: (8)             a = zeros(3, fill_value=0., dtype=complex)
1860: (8)             assert_equal(a.fill_value, 0.)
1861: (8)             a = identity(3, fill_value=0., dtype=complex)
1862: (8)             assert_equal(a.fill_value, 0.)
1863: (4)         def test_shape_argument(self):
1864: (8)             a = empty(shape=(3, ))
1865: (8)             assert_equal(a.shape, (3, ))
1866: (8)             a = ones(shape=(3, ), dtype=float)
1867: (8)             assert_equal(a.shape, (3, ))
1868: (8)             a = zeros(shape=(3, ), dtype=complex)
1869: (8)             assert_equal(a.shape, (3, ))
1870: (4)         def test_fillvalue_in_view(self):
1871: (8)             x = array([1, 2, 3], fill_value=1, dtype=np.int64)
1872: (8)             y = x.view()
1873: (8)             assert_(y.fill_value == 1)
1874: (8)             y = x.view(MaskedArray)
1875: (8)             assert_(y.fill_value == 1)
1876: (8)             y = x.view(type=MaskedArray)
1877: (8)             assert_(y.fill_value == 1)
1878: (8)             y = x.view(np.ndarray)
1879: (8)             y = x.view(type=np.ndarray)
1880: (8)             y = x.view(MaskedArray, fill_value=2)
1881: (8)             assert_(y.fill_value == 2)
1882: (8)             y = x.view(type=MaskedArray, fill_value=2)
1883: (8)             assert_(y.fill_value == 2)
1884: (8)             y = x.view(dtype=np.int32)
1885: (8)             assert_(y.fill_value == 999999)
1886: (4)         def test_fillvalue_bytes_or_str(self):
1887: (8)             a = empty(shape=(3, ), dtype="(2)3S,(2)3U")
1888: (8)             assert_equal(a["f0"].fill_value, default_fill_value(b"spam"))
1889: (8)             assert_equal(a["f1"].fill_value, default_fill_value("eggs"))

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1890: (0)
1891: (4)
1892: (8)
1893: (18)
1894: (8)
1895: (8)
1896: (4)
1897: (8)
1898: (4)
1899: (8)
1900: (18)
1901: (18)
1902: (18)
1903: (18)
1904: (18)
1905: (18)
1906: (18)
1907: (18)
1908: (18)
1909: (18)
1910: (18)
1911: (18)
1912: (18)
1913: (18)
1914: (18)
1915: (18)
1916: (12)
1917: (16)
1918: (12)
1919: (16)
1920: (12)
1921: (12)
1922: (12)
1923: (12)
1924: (12)
1925: (12)
1926: (4)
1927: (8)
1928: (8)
1929: (8)
1930: (8)
1931: (8)
1932: (8)
1933: (4)
1934: (8)
1935: (8)
1936: (8)
1937: (8)
1938: (8)
1939: (8)
1940: (8)
1941: (8)
1942: (4)
1943: (8)
1944: (8)
1945: (8)
1946: (31)
1947: (8)
1948: (8)
1949: (8)
1950: (4)
1951: (8)
1952: (8)
1953: (8)
1954: (8)
1955: (12)
1956: (12)
1957: (16)
1958: (12)

class TestUfuncs:
    def setup_method(self):
        self.d = (array([1.0, 0, -1, pi / 2] * 2, mask=[0, 1] + [0] * 6),
                  array([1.0, 0, -1, pi / 2] * 2, mask=[1, 0] + [0] * 6),)
        self.err_status = np.geterr()
        np.seterr(divide='ignore', invalid='ignore')
    def teardown_method(self):
        np.seterr(**self.err_status)
    def test_testUfuncRegression(self):
        for f in ['sqrt', 'log', 'log10', 'exp', 'conjugate',
                  'sin', 'cos', 'tan',
                  'arcsin', 'arccos', 'arctan',
                  'sinh', 'cosh', 'tanh',
                  'arcsinh',
                  'arccosh',
                  'arctanh',
                  'absolute', 'fabs', 'negative',
                  'floor', 'ceil',
                  'logical_not',
                  'add', 'subtract', 'multiply',
                  'divide', 'true_divide', 'floor_divide',
                  'remainder', 'fmod', 'hypot', 'arctan2',
                  'equal', 'not_equal', 'less_equal', 'greater_equal',
                  'less', 'greater',
                  'logical_and', 'logical_or', 'logical_xor',
                  ]:
            try:
                uf = getattr(umath, f)
            except AttributeError:
                uf = getattr(fromnumeric, f)
            mf = getattr(numpy.ma.core, f)
            args = self.d[:uf.nin]
            ur = uf(*args)
            mr = mf(*args)
            assert_equal(ur.filled(0), mr.filled(0), f)
            assert_mask_equal(ur.mask, mr.mask, err_msg=f)
    def test_reduce(self):
        a = self.d[0]
        assert_(not alltrue(a, axis=0))
        assert_(sometrue(a, axis=0))
        assert_equal(sum(a[:3], axis=0), 0)
        assert_equal(product(a, axis=0), 0)
        assert_equal(add.reduce(a), pi)
    def test_minmax(self):
        a = arange(1, 13).reshape(3, 4)
        amask = masked_where(a < 5, a)
        assert_equal(amask.max(), a.max())
        assert_equal(amask.min(), 5)
        assert_equal(amask.max(0), a.max(0))
        assert_equal(amask.min(0), [5, 6, 7, 8])
        assert_(amask.max(1)[0].mask)
        assert_(amask.min(1)[0].mask)
    def test_ndarray_mask(self):
        a = masked_array([-1, 0, 1, 2, 3], mask=[0, 0, 0, 0, 1])
        test = np.sqrt(a)
        control = masked_array([-1, 0, 1, np.sqrt(2), -1],
                               mask=[1, 0, 0, 0, 1])
        assert_equal(test, control)
        assert_equal(test.mask, control.mask)
        assert_(not isinstance(test.mask, MaskedArray))
    def test_treatment_of_NotImplemented(self):
        a = masked_array([1., 2.], mask=[1, 0])
        assert_raises(TypeError, operator.mul, a, "abc")
        assert_raises(TypeError, operator.truediv, a, "abc")
    class MyClass:
        __array_priority__ = a.__array_priority__ + 1
        def __mul__(self, other):
            return "My mul"
        def __rmul__(self, other):

```

```

1959: (16)                     return "My rmul"
1960: (8)                      me = MyClass()
1961: (8)                      assert_(me * a == "My mul")
1962: (8)                      assert_(a * me == "My rmul")
1963: (8)                      class MyClass2:
1964: (12)                        __array_priority__ = 100
1965: (12)                        def __mul__(self, other):
1966: (16)                          return "Me2mul"
1967: (12)                        def __rmul__(self, other):
1968: (16)                          return "Me2rmul"
1969: (12)                        def __rdiv__(self, other):
1970: (16)                          return "Me2rdiv"
1971: (12)                        __rtruediv__ = __rdiv__
1972: (8)                      me_too = MyClass2()
1973: (8)                      assert_(a.__mul__(me_too) is NotImplemented)
1974: (8)                      assert_(all(multiply.outer(a, me_too) == "Me2rmul"))
1975: (8)                      assert_(a.__truediv__(me_too) is NotImplemented)
1976: (8)                      assert_(me_too * a == "Me2mul")
1977: (8)                      assert_(a * me_too == "Me2rmul")
1978: (8)                      assert_(a / me_too == "Me2rdiv")
1979: (4)                      def test_no_masked_nan_warnings(self):
1980: (8)                        m = np.ma.array([0.5, np.nan], mask=[0,1])
1981: (8)                        with warnings.catch_warnings():
1982: (12)                          warnings.filterwarnings("error")
1983: (12)                          exp(m)
1984: (12)                          add(m, 1)
1985: (12)                          m > 0
1986: (12)                          sqrt(m)
1987: (12)                          log(m)
1988: (12)                          tan(m)
1989: (12)                          arcsin(m)
1990: (12)                          arccos(m)
1991: (12)                          arccosh(m)
1992: (12)                          divide(m, 2)
1993: (12)                          allclose(m, 0.5)
1994: (0)                      class TestMaskedArrayInPlaceArithmetic:
1995: (4)                        def setup_method(self):
1996: (8)                          x = arange(10)
1997: (8)                          y = arange(10)
1998: (8)                          xm = arange(10)
1999: (8)                          xm[2] = masked
2000: (8)                          self.intdata = (x, y, xm)
2001: (8)                          self.floatdata = (x.astype(float), y.astype(float), xm.astype(float))
2002: (8)                          self.othertypes = np.typecodes['AllInteger'] +
np.typecodes['AllFloat']
2003: (8)                          self.othertypes = [np.dtype(_.type) for _ in self.othertypes]
2004: (8)                          self.uint8data =
2005: (12)                            x.astype(np.uint8),
2006: (12)                            y.astype(np.uint8),
2007: (12)                            xm.astype(np.uint8)
2008: (8)                          )
2009: (4)                          def test_inplace_addition_scalar(self):
2010: (8)                            (x, y, xm) = self.intdata
2011: (8)                            xm[2] = masked
2012: (8)                            x += 1
2013: (8)                            assert_equal(x, y + 1)
2014: (8)                            xm += 1
2015: (8)                            assert_equal(xm, y + 1)
2016: (8)                            (x, _, xm) = self.floatdata
2017: (8)                            id1 = x.data.ctypes.data
2018: (8)                            x += 1.
2019: (8)                            assert_(id1 == x.data.ctypes.data)
2020: (8)                            assert_equal(x, y + 1.)
2021: (4)                          def test_inplace_addition_array(self):
2022: (8)                            (x, y, xm) = self.intdata
2023: (8)                            m = xm.mask
2024: (8)                            a = arange(10, dtype=np.int16)
2025: (8)                            a[-1] = masked
2026: (8)                            x += a

```

```

2027: (8)             xm += a
2028: (8)             assert_equal(x, y + a)
2029: (8)             assert_equal(xm, y + a)
2030: (8)             assert_equal(xm.mask, mask_or(m, a.mask))
2031: (4)             def test_inplace_subtraction_scalar(self):
2032: (8)                 (x, y, xm) = self.intdata
2033: (8)                 x -= 1
2034: (8)                 assert_equal(x, y - 1)
2035: (8)                 xm -= 1
2036: (8)                 assert_equal(xm, y - 1)
2037: (4)             def test_inplace_subtraction_array(self):
2038: (8)                 (x, y, xm) = self.floatdata
2039: (8)                 m = xm.mask
2040: (8)                 a = arange(10, dtype=float)
2041: (8)                 a[-1] = masked
2042: (8)                 x -= a
2043: (8)                 xm -= a
2044: (8)                 assert_equal(x, y - a)
2045: (8)                 assert_equal(xm, y - a)
2046: (8)                 assert_equal(xm.mask, mask_or(m, a.mask))
2047: (4)             def test_inplace_multiplication_scalar(self):
2048: (8)                 (x, y, xm) = self.floatdata
2049: (8)                 x *= 2.0
2050: (8)                 assert_equal(x, y * 2)
2051: (8)                 xm *= 2.0
2052: (8)                 assert_equal(xm, y * 2)
2053: (4)             def test_inplace_multiplication_array(self):
2054: (8)                 (x, y, xm) = self.floatdata
2055: (8)                 m = xm.mask
2056: (8)                 a = arange(10, dtype=float)
2057: (8)                 a[-1] = masked
2058: (8)                 x *= a
2059: (8)                 xm *= a
2060: (8)                 assert_equal(x, y * a)
2061: (8)                 assert_equal(xm, y * a)
2062: (8)                 assert_equal(xm.mask, mask_or(m, a.mask))
2063: (4)             def test_inplace_division_scalar_int(self):
2064: (8)                 (x, y, xm) = self.intdata
2065: (8)                 x = arange(10) * 2
2066: (8)                 xm = arange(10) * 2
2067: (8)                 xm[2] = masked
2068: (8)                 x // 2
2069: (8)                 assert_equal(x, y)
2070: (8)                 xm // 2
2071: (8)                 assert_equal(xm, y)
2072: (4)             def test_inplace_division_scalar_float(self):
2073: (8)                 (x, y, xm) = self.floatdata
2074: (8)                 x /= 2.0
2075: (8)                 assert_equal(x, y / 2.0)
2076: (8)                 xm /= arange(10)
2077: (8)                 assert_equal(xm, ones((10,)))
2078: (4)             def test_inplace_division_array_float(self):
2079: (8)                 (x, y, xm) = self.floatdata
2080: (8)                 m = xm.mask
2081: (8)                 a = arange(10, dtype=float)
2082: (8)                 a[-1] = masked
2083: (8)                 x /= a
2084: (8)                 xm /= a
2085: (8)                 assert_equal(x, y / a)
2086: (8)                 assert_equal(xm, y / a)
2087: (8)                 assert_equal(xm.mask, mask_or(mask_or(m, a.mask), (a == 0)))
2088: (4)             def test_inplace_division_misc(self):
2089: (8)                 x = [1., 1., 1., -2., pi / 2., 4., 5., -10., 10., 1., 2., 3.]
2090: (8)                 y = [5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.]
2091: (8)                 m1 = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
2092: (8)                 m2 = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1]
2093: (8)                 xm = masked_array(x, mask=m1)
2094: (8)                 ym = masked_array(y, mask=m2)
2095: (8)                 z = xm / ym

```

```

2096: (8)             assert_equal(z._mask, [1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1])
2097: (8)             assert_equal(z._data,
2098: (21)                 [1., 1., 1., -1., -pi / 2., 4., 5., 1., 1., 1., 2., 3.])
2099: (8)             xm = xm.copy()
2100: (8)             xm /= ym
2101: (8)             assert_equal(xm._mask, [1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1])
2102: (8)             assert_equal(z._data,
2103: (21)                 [1., 1., 1., -1., -pi / 2., 4., 5., 1., 1., 1., 2., 3.])
2104: (4)             def test_datafriendly_add(self):
2105: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2106: (8)                 xx = x + 1
2107: (8)                 assert_equal(xx.data, [2, 3, 3])
2108: (8)                 assert_equal(xx.mask, [0, 0, 1])
2109: (8)                 x += 1
2110: (8)                 assert_equal(x.data, [2, 3, 3])
2111: (8)                 assert_equal(x.mask, [0, 0, 1])
2112: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2113: (8)                 xx = x + array([1, 2, 3], mask=[1, 0, 0])
2114: (8)                 assert_equal(xx.data, [1, 4, 3])
2115: (8)                 assert_equal(xx.mask, [1, 0, 1])
2116: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2117: (8)                 x += array([1, 2, 3], mask=[1, 0, 0])
2118: (8)                 assert_equal(x.data, [1, 4, 3])
2119: (8)                 assert_equal(x.mask, [1, 0, 1])
2120: (4)             def test_datafriendly_sub(self):
2121: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2122: (8)                 xx = x - 1
2123: (8)                 assert_equal(xx.data, [0, 1, 3])
2124: (8)                 assert_equal(xx.mask, [0, 0, 1])
2125: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2126: (8)                 x -= 1
2127: (8)                 assert_equal(x.data, [0, 1, 3])
2128: (8)                 assert_equal(x.mask, [0, 0, 1])
2129: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2130: (8)                 xx = x - array([1, 2, 3], mask=[1, 0, 0])
2131: (8)                 assert_equal(xx.data, [1, 0, 3])
2132: (8)                 assert_equal(xx.mask, [1, 0, 1])
2133: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2134: (8)                 x -= array([1, 2, 3], mask=[1, 0, 0])
2135: (8)                 assert_equal(x.data, [1, 0, 3])
2136: (8)                 assert_equal(x.mask, [1, 0, 1])
2137: (4)             def test_datafriendly_mul(self):
2138: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2139: (8)                 xx = x * 2
2140: (8)                 assert_equal(xx.data, [2, 4, 3])
2141: (8)                 assert_equal(xx.mask, [0, 0, 1])
2142: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2143: (8)                 x *= 2
2144: (8)                 assert_equal(x.data, [2, 4, 3])
2145: (8)                 assert_equal(x.mask, [0, 0, 1])
2146: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2147: (8)                 xx = x * array([10, 20, 30], mask=[1, 0, 0])
2148: (8)                 assert_equal(xx.data, [1, 40, 3])
2149: (8)                 assert_equal(xx.mask, [1, 0, 1])
2150: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2151: (8)                 x *= array([10, 20, 30], mask=[1, 0, 0])
2152: (8)                 assert_equal(x.data, [1, 40, 3])
2153: (8)                 assert_equal(x.mask, [1, 0, 1])
2154: (4)             def test_datafriendly_div(self):
2155: (8)                 x = array([1, 2, 3], mask=[0, 0, 1])
2156: (8)                 xx = x / 2.
2157: (8)                 assert_equal(xx.data, [1 / 2., 2 / 2., 3])
2158: (8)                 assert_equal(xx.mask, [0, 0, 1])
2159: (8)                 x = array([1., 2., 3.], mask=[0, 0, 1])
2160: (8)                 x /= 2.
2161: (8)                 assert_equal(x.data, [1 / 2., 2 / 2., 3])
2162: (8)                 assert_equal(x.mask, [0, 0, 1])
2163: (8)                 x = array([1., 2., 3.], mask=[0, 0, 1])
2164: (8)                 xx = x / array([10., 20., 30.], mask=[1, 0, 0])

```

```

2165: (8)             assert_equal(xx.data, [1., 2. / 20., 3.])
2166: (8)             assert_equal(xx.mask, [1, 0, 1])
2167: (8)             x = array([1., 2., 3.], mask=[0, 0, 1])
2168: (8)             x /= array([10., 20., 30.], mask=[1, 0, 0])
2169: (8)             assert_equal(x.data, [1., 2 / 20., 3.])
2170: (8)             assert_equal(x.mask, [1, 0, 1])
2171: (4)             def test_datafriendly_pow(self):
2172: (8)                 x = array([1., 2., 3.], mask=[0, 0, 1])
2173: (8)                 xx = x ** 2.5
2174: (8)                 assert_equal(xx.data, [1., 2. ** 2.5, 3.])
2175: (8)                 assert_equal(xx.mask, [0, 0, 1])
2176: (8)                 x **= 2.5
2177: (8)                 assert_equal(x.data, [1., 2. ** 2.5, 3.])
2178: (8)                 assert_equal(x.mask, [0, 0, 1])
2179: (4)             def test_datafriendly_add_arrays(self):
2180: (8)                 a = array([[1, 1], [3, 3]])
2181: (8)                 b = array([1, 1], mask=[0, 0])
2182: (8)                 a += b
2183: (8)                 assert_equal(a, [[2, 2], [4, 4]])
2184: (8)                 if a.mask is not nomask:
2185: (12)                     assert_equal(a.mask, [[0, 0], [0, 0]])
2186: (8)                 a = array([[1, 1], [3, 3]])
2187: (8)                 b = array([1, 1], mask=[0, 1])
2188: (8)                 a += b
2189: (8)                 assert_equal(a, [[2, 2], [4, 4]])
2190: (8)                 assert_equal(a.mask, [[0, 1], [0, 1]])
2191: (4)             def test_datafriendly_sub_arrays(self):
2192: (8)                 a = array([[1, 1], [3, 3]])
2193: (8)                 b = array([1, 1], mask=[0, 0])
2194: (8)                 a -= b
2195: (8)                 assert_equal(a, [[0, 0], [2, 2]])
2196: (8)                 if a.mask is not nomask:
2197: (12)                     assert_equal(a.mask, [[0, 0], [0, 0]])
2198: (8)                 a = array([[1, 1], [3, 3]])
2199: (8)                 b = array([1, 1], mask=[0, 1])
2200: (8)                 a -= b
2201: (8)                 assert_equal(a, [[0, 0], [2, 2]])
2202: (8)                 assert_equal(a.mask, [[0, 1], [0, 1]])
2203: (4)             def test_datafriendly_mul_arrays(self):
2204: (8)                 a = array([[1, 1], [3, 3]])
2205: (8)                 b = array([1, 1], mask=[0, 0])
2206: (8)                 a *= b
2207: (8)                 assert_equal(a, [[1, 1], [3, 3]])
2208: (8)                 if a.mask is not nomask:
2209: (12)                     assert_equal(a.mask, [[0, 0], [0, 0]])
2210: (8)                 a = array([[1, 1], [3, 3]])
2211: (8)                 b = array([1, 1], mask=[0, 1])
2212: (8)                 a *= b
2213: (8)                 assert_equal(a, [[1, 1], [3, 3]])
2214: (8)                 assert_equal(a.mask, [[0, 1], [0, 1]])
2215: (4)             def test_inplace_addition_scalar_type(self):
2216: (8)                 for t in self.othertypes:
2217: (12)                     with warnings.catch_warnings():
2218: (16)                         warnings.filterwarnings("error")
2219: (16)                         (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2220: (16)                         xm[2] = masked
2221: (16)                         x += t(1)
2222: (16)                         assert_equal(x, y + t(1))
2223: (16)                         xm += t(1)
2224: (16)                         assert_equal(xm, y + t(1))
2225: (4)             def test_inplace_addition_array_type(self):
2226: (8)                 for t in self.othertypes:
2227: (12)                     with warnings.catch_warnings():
2228: (16)                         warnings.filterwarnings("error")
2229: (16)                         (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2230: (16)                         m = xm.mask
2231: (16)                         a = arange(10, dtype=t)
2232: (16)                         a[-1] = masked
2233: (16)                         x += a

```

```

2234: (16)             xm += a
2235: (16)             assert_equal(x, y + a)
2236: (16)             assert_equal(xm, y + a)
2237: (16)             assert_equal(xm.mask, mask_or(m, a.mask))
2238: (4)              def test_inplace_subtraction_scalar_type(self):
2239: (8)                for t in self.othertypes:
2240: (12)                  with warnings.catch_warnings():
2241: (16)                      warnings.filterwarnings("error")
2242: (16)                      (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2243: (16)                      x -= t(1)
2244: (16)                      assert_equal(x, y - t(1))
2245: (16)                      xm -= t(1)
2246: (16)                      assert_equal(xm, y - t(1))
2247: (4)              def test_inplace_subtraction_array_type(self):
2248: (8)                for t in self.othertypes:
2249: (12)                  with warnings.catch_warnings():
2250: (16)                      warnings.filterwarnings("error")
2251: (16)                      (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2252: (16)                      m = xm.mask
2253: (16)                      a = arange(10, dtype=t)
2254: (16)                      a[-1] = masked
2255: (16)                      x -= a
2256: (16)                      xm -= a
2257: (16)                      assert_equal(x, y - a)
2258: (16)                      assert_equal(xm, y - a)
2259: (16)                      assert_equal(xm.mask, mask_or(m, a.mask))
2260: (4)              def test_inplace_multiplication_scalar_type(self):
2261: (8)                for t in self.othertypes:
2262: (12)                  with warnings.catch_warnings():
2263: (16)                      warnings.filterwarnings("error")
2264: (16)                      (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2265: (16)                      x *= t(2)
2266: (16)                      assert_equal(x, y * t(2))
2267: (16)                      xm *= t(2)
2268: (16)                      assert_equal(xm, y * t(2))
2269: (4)              def test_inplace_multiplication_array_type(self):
2270: (8)                for t in self.othertypes:
2271: (12)                  with warnings.catch_warnings():
2272: (16)                      warnings.filterwarnings("error")
2273: (16)                      (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2274: (16)                      m = xm.mask
2275: (16)                      a = arange(10, dtype=t)
2276: (16)                      a[-1] = masked
2277: (16)                      x *= a
2278: (16)                      xm *= a
2279: (16)                      assert_equal(x, y * a)
2280: (16)                      assert_equal(xm, y * a)
2281: (16)                      assert_equal(xm.mask, mask_or(m, a.mask))
2282: (4)              def test_inplace_floor_division_scalar_type(self):
2283: (8)                unsupported = {np.dtype(t).type for t in np.typecodes["Complex"]}
2284: (8)                for t in self.othertypes:
2285: (12)                  with warnings.catch_warnings():
2286: (16)                      warnings.filterwarnings("error")
2287: (16)                      (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2288: (16)                      x = arange(10, dtype=t) * t(2)
2289: (16)                      xm = arange(10, dtype=t) * t(2)
2290: (16)                      xm[2] = masked
2291: (16)                      try:
2292: (20)                          x // t(2)
2293: (20)                          xm // t(2)
2294: (20)                          assert_equal(x, y)
2295: (20)                          assert_equal(xm, y)
2296: (16)                      except TypeError:
2297: (20)                          msg = f"Supported type {t} throwing TypeError"
2298: (20)                          assert t in unsupported, msg
2299: (4)              def test_inplace_floor_division_array_type(self):
2300: (8)                unsupported = {np.dtype(t).type for t in np.typecodes["Complex"]}
2301: (8)                for t in self.othertypes:
2302: (12)                  with warnings.catch_warnings():

```

```

2303: (16)             warnings.filterwarnings("error")
2304: (16)             (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2305: (16)             m = xm.mask
2306: (16)             a = arange(10, dtype=t)
2307: (16)             a[-1] = masked
2308: (16)             try:
2309: (20)                 x //= a
2310: (20)                 xm //= a
2311: (20)                 assert_equal(x, y // a)
2312: (20)                 assert_equal(xm, y // a)
2313: (20)                 assert_equal(
2314: (24)                     xm.mask,
2315: (24)                     mask_or(mask_or(m, a.mask), (a == t(0))))
2316: (20)             )
2317: (16)             except TypeError:
2318: (20)                 msg = f"Supported type {t} throwing TypeError"
2319: (20)                 assert t in unsupported, msg
2320: (4)                  def test_inplace_division_scalar_type(self):
2321: (8)                      for t in self.othertypes:
2322: (12)                          with suppress_warnings() as sup:
2323: (16)                            sup.record(UserWarning)
2324: (16)                            (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2325: (16)                            x = arange(10, dtype=t) * t(2)
2326: (16)                            xm = arange(10, dtype=t) * t(2)
2327: (16)                            xm[2] = masked
2328: (16)                            try:
2329: (20)                                x /= t(2)
2330: (20)                                assert_equal(x, y)
2331: (16)                            except (DeprecationWarning, TypeError) as e:
2332: (20)                                warnings.warn(str(e), stacklevel=1)
2333: (16)                            try:
2334: (20)                                xm /= t(2)
2335: (20)                                assert_equal(xm, y)
2336: (16)                            except (DeprecationWarning, TypeError) as e:
2337: (20)                                warnings.warn(str(e), stacklevel=1)
2338: (16)                            if issubclass(t, np.integer):
2339: (20)                                assert_equal(len(sup.log), 2, f'Failed on type={t}.')
2340: (16)                            else:
2341: (20)                                assert_equal(len(sup.log), 0, f'Failed on type={t}.')
2342: (4)                  def test_inplace_division_array_type(self):
2343: (8)                      for t in self.othertypes:
2344: (12)                          with suppress_warnings() as sup:
2345: (16)                            sup.record(UserWarning)
2346: (16)                            (x, y, xm) = (_.astype(t) for _ in self.uint8data)
2347: (16)                            m = xm.mask
2348: (16)                            a = arange(10, dtype=t)
2349: (16)                            a[-1] = masked
2350: (16)                            try:
2351: (20)                                x /= a
2352: (20)                                assert_equal(x, y / a)
2353: (16)                            except (DeprecationWarning, TypeError) as e:
2354: (20)                                warnings.warn(str(e), stacklevel=1)
2355: (16)                            try:
2356: (20)                                xm /= a
2357: (20)                                assert_equal(xm, y / a)
2358: (20)                                assert_equal(
2359: (24)                                    xm.mask,
2360: (24)                                    mask_or(mask_or(m, a.mask), (a == t(0))))
2361: (20)                            )
2362: (16)                            except (DeprecationWarning, TypeError) as e:
2363: (20)                                warnings.warn(str(e), stacklevel=1)
2364: (16)                            if issubclass(t, np.integer):
2365: (20)                                assert_equal(len(sup.log), 2, f'Failed on type={t}.')
2366: (16)                            else:
2367: (20)                                assert_equal(len(sup.log), 0, f'Failed on type={t}.')
2368: (4)                  def test_inplace_pow_type(self):
2369: (8)                      for t in self.othertypes:
2370: (12)                          with warnings.catch_warnings():
2371: (16)                              warnings.filterwarnings("error")

```

```

2372: (16)             x = array([1, 2, 3], mask=[0, 0, 1], dtype=t)
2373: (16)             xx = x ** t(2)
2374: (16)             xx_r = array([1, 2 ** 2, 3], mask=[0, 0, 1], dtype=t)
2375: (16)             assert_equal(xx.data, xx_r.data)
2376: (16)             assert_equal(xx.mask, xx_r.mask)
2377: (16)             x **= t(2)
2378: (16)             assert_equal(x.data, xx_r.data)
2379: (16)             assert_equal(x.mask, xx_r.mask)
2380: (0)              class TestMaskedArrayMethods:
2381: (4)                def setup_method(self):
2382: (8)                  x = np.array([8.375, 7.545, 8.828, 8.5, 1.757, 5.928,
2383: (22)                      8.43, 7.78, 9.865, 5.878, 8.979, 4.732,
2384: (22)                      3.012, 6.022, 5.095, 3.116, 5.238, 3.957,
2385: (22)                      6.04, 9.63, 7.712, 3.382, 4.489, 6.479,
2386: (22)                      7.189, 9.645, 5.395, 4.961, 9.894, 2.893,
2387: (22)                      7.357, 9.828, 6.272, 3.758, 6.693, 0.993])
2388: (8)                  X = x.reshape(6, 6)
2389: (8)                  XX = x.reshape(3, 2, 2, 3)
2390: (8)                  m = np.array([0, 1, 0, 1, 0, 0,
2391: (21)                      1, 0, 1, 1, 0, 1,
2392: (21)                      0, 0, 0, 1, 0, 1,
2393: (21)                      0, 0, 0, 1, 1, 1,
2394: (21)                      1, 0, 0, 1, 0, 0,
2395: (21)                      0, 0, 1, 0, 1, 0])
2396: (8)                  mx = array(data=x, mask=m)
2397: (8)                  mX = array(data=X, mask=m.reshape(X.shape))
2398: (8)                  mXX = array(data=XX, mask=m.reshape(XX.shape))
2399: (8)                  m2 = np.array([1, 1, 0, 1, 0, 0,
2400: (22)                      1, 1, 1, 1, 0, 1,
2401: (22)                      0, 0, 1, 1, 0, 1,
2402: (22)                      0, 0, 0, 1, 1, 1,
2403: (22)                      1, 0, 0, 1, 1, 0,
2404: (22)                      0, 0, 1, 0, 1, 1])
2405: (8)                  m2x = array(data=x, mask=m2)
2406: (8)                  m2X = array(data=X, mask=m2.reshape(X.shape))
2407: (8)                  m2XX = array(data=XX, mask=m2.reshape(XX.shape))
2408: (8)                  self.d = (x, X, XX, m, mx, mX, mXX, m2x, m2X, m2XX)
2409: (4)                  def test_generic_methods(self):
2410: (8)                      a = array([1, 3, 2])
2411: (8)                      assert_equal(a.any(), a._data.any())
2412: (8)                      assert_equal(a.all(), a._data.all())
2413: (8)                      assert_equal(a.argmax(), a._data.argmax())
2414: (8)                      assert_equal(a.argmin(), a._data.argmin())
2415: (8)                      assert_equal(a.choose(0, 1, 2, 3, 4), a._data.choose(0, 1, 2, 3, 4))
2416: (8)                      assert_equal(a.compress([1, 0, 1]), a._data.compress([1, 0, 1]))
2417: (8)                      assert_equal(a.conj(), a._data.conj())
2418: (8)                      assert_equal(a.conjugate(), a._data.conjugate())
2419: (8)                      m = array([[1, 2], [3, 4]])
2420: (8)                      assert_equal(m.diagonal(), m._data.diagonal())
2421: (8)                      assert_equal(a.sum(), a._data.sum())
2422: (8)                      assert_equal(a.take([1, 2]), a._data.take([1, 2]))
2423: (8)                      assert_equal(m.transpose(), m._data.transpose())
2424: (4)                  def test_allclose(self):
2425: (8)                      a = np.random.rand(10)
2426: (8)                      b = a + np.random.rand(10) * 1e-8
2427: (8)                      assert_(allclose(a, b))
2428: (8)                      a[0] = np.inf
2429: (8)                      assert_(not allclose(a, b))
2430: (8)                      b[0] = np.inf
2431: (8)                      assert_(allclose(a, b))
2432: (8)                      a = masked_array(a)
2433: (8)                      a[-1] = masked
2434: (8)                      assert_(allclose(a, b, masked_equal=True))
2435: (8)                      assert_(not allclose(a, b, masked_equal=False))
2436: (8)                      a *= 1e-8
2437: (8)                      a[0] = 0
2438: (8)                      assert_(allclose(a, 0, masked_equal=True))
2439: (8)                      a = masked_array([np.iinfo(np.int_).min], dtype=np.int_)
2440: (8)                      assert_(allclose(a, a))

```

```

2441: (4)             def test_allclose_timedelta(self):
2442: (8)               a = np.array([[1, 2, 3, 4]], dtype="m8[ns]")
2443: (8)               assert allclose(a, a, atol=0)
2444: (8)               assert allclose(a, a, atol=np.timedelta64(1, "ns"))
2445: (4)             def test_allany(self):
2446: (8)               x = np.array([[0.13, 0.26, 0.90],
2447: (22)                   [0.28, 0.33, 0.63],
2448: (22)                   [0.31, 0.87, 0.70]])
2449: (8)               m = np.array([[True, False, False],
2450: (22)                   [False, False, False],
2451: (22)                   [True, True, False]], dtype=np.bool_)
2452: (8)               mx = masked_array(x, mask=m)
2453: (8)               mxbig = (mx > 0.5)
2454: (8)               mxsmall = (mx < 0.5)
2455: (8)               assert_(not mxbig.all())
2456: (8)               assert_(mxbig.any())
2457: (8)               assert_equal(mxbig.all(0), [False, False, True])
2458: (8)               assert_equal(mxbig.all(1), [False, False, True])
2459: (8)               assert_equal(mxbig.any(0), [False, False, True])
2460: (8)               assert_equal(mxbig.any(1), [True, True, True])
2461: (8)               assert_(not mxsmall.all())
2462: (8)               assert_(mxsmall.any())
2463: (8)               assert_equal(mxsmall.all(0), [True, True, False])
2464: (8)               assert_equal(mxsmall.all(1), [False, False, False])
2465: (8)               assert_equal(mxsmall.any(0), [True, True, False])
2466: (8)               assert_equal(mxsmall.any(1), [True, True, False])
2467: (4)             def test_allany_oddities(self):
2468: (8)               store = empty(), dtype=bool)
2469: (8)               full = array([1, 2, 3], mask=True)
2470: (8)               assert_(full.all() is masked)
2471: (8)               full.all(out=store)
2472: (8)               assert_(store)
2473: (8)               assert_(store._mask, True)
2474: (8)               assert_(store is not masked)
2475: (8)               store = empty(), dtype=bool)
2476: (8)               assert_(full.any() is masked)
2477: (8)               full.any(out=store)
2478: (8)               assert_(not store)
2479: (8)               assert_(store._mask, True)
2480: (8)               assert_(store is not masked)
2481: (4)             def test_argmax_argmin(self):
2482: (8)               (x, X, XX, m, mx, mX, mXX, m2x, m2X, m2XX) = self.d
2483: (8)               assert_equal(mx.argmin(), 35)
2484: (8)               assert_equal(mX.argmin(), 35)
2485: (8)               assert_equal(m2x.argmin(), 4)
2486: (8)               assert_equal(m2X.argmin(), 4)
2487: (8)               assert_equal(mx.argmax(), 28)
2488: (8)               assert_equal(mX.argmax(), 28)
2489: (8)               assert_equal(m2x.argmax(), 31)
2490: (8)               assert_equal(m2X.argmax(), 31)
2491: (8)               assert_equal(mX.argmin(0), [2, 2, 2, 5, 0, 5])
2492: (8)               assert_equal(m2X.argmin(0), [2, 2, 4, 5, 0, 4])
2493: (8)               assert_equal(mX.argmax(0), [0, 5, 0, 5, 4, 0])
2494: (8)               assert_equal(m2X.argmax(0), [5, 5, 0, 5, 1, 0])
2495: (8)               assert_equal(mX.argmin(1), [4, 1, 0, 0, 5, 5, ])
2496: (8)               assert_equal(m2X.argmin(1), [4, 4, 0, 0, 5, 3])
2497: (8)               assert_equal(mX.argmax(1), [2, 4, 1, 1, 4, 1])
2498: (8)               assert_equal(m2X.argmax(1), [2, 4, 1, 1, 1, 1])
2499: (4)             def test_clip(self):
2500: (8)               x = np.array([8.375, 7.545, 8.828, 8.5, 1.757, 5.928,
2501: (22)                   8.43, 7.78, 9.865, 5.878, 8.979, 4.732,
2502: (22)                   3.012, 6.022, 5.095, 3.116, 5.238, 3.957,
2503: (22)                   6.04, 9.63, 7.712, 3.382, 4.489, 6.479,
2504: (22)                   7.189, 9.645, 5.395, 4.961, 9.894, 2.893,
2505: (22)                   7.357, 9.828, 6.272, 3.758, 6.693, 0.993])
2506: (8)               m = np.array([0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1,
2507: (22)                   0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1,
2508: (22)                   1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0])
2509: (8)               mx = array(x, mask=m)

```

```

2510: (8)             clipped = mx.clip(2, 8)
2511: (8)             assert_equal(clipped.mask, mx.mask)
2512: (8)             assert_equal(clipped._data, x.clip(2, 8))
2513: (8)             assert_equal(clipped._data, mx._data.clip(2, 8))
2514: (4)             def test_clip_out(self):
2515: (8)                 a = np.arange(10)
2516: (8)                 m = np.ma.MaskedArray(a, mask=[0, 1] * 5)
2517: (8)                 m.clip(0, 5, out=m)
2518: (8)                 assert_equal(m.mask, [0, 1] * 5)
2519: (4)             def test_compress(self):
2520: (8)                 a = masked_array([1., 2., 3., 4., 5.], fill_value=9999)
2521: (8)                 condition = (a > 1.5) & (a < 3.5)
2522: (8)                 assert_equal(a.compress(condition), [2., 3.])
2523: (8)                 a[[2, 3]] = masked
2524: (8)                 b = a.compress(condition)
2525: (8)                 assert_equal(b._data, [2., 3.])
2526: (8)                 assert_equal(b._mask, [0, 1])
2527: (8)                 assert_equal(b.fill_value, 9999)
2528: (8)                 assert_equal(b, a[condition])
2529: (8)                 condition = (a < 4.)
2530: (8)                 b = a.compress(condition)
2531: (8)                 assert_equal(b._data, [1., 2., 3.])
2532: (8)                 assert_equal(b._mask, [0, 0, 1])
2533: (8)                 assert_equal(b.fill_value, 9999)
2534: (8)                 assert_equal(b, a[condition])
2535: (8)                 a = masked_array([[10, 20, 30], [40, 50, 60]],
2536: (25)                   mask=[[0, 0, 1], [1, 0, 0]])
2537: (8)                 b = a.compress(a.ravel() >= 22)
2538: (8)                 assert_equal(b._data, [30, 40, 50, 60])
2539: (8)                 assert_equal(b._mask, [1, 1, 0, 0])
2540: (8)                 x = np.array([3, 1, 2])
2541: (8)                 b = a.compress(x >= 2, axis=1)
2542: (8)                 assert_equal(b._data, [[10, 30], [40, 60]])
2543: (8)                 assert_equal(b._mask, [[0, 1], [1, 0]])
2544: (4)             def test_compressed(self):
2545: (8)                 a = array([1, 2, 3, 4], mask=[0, 0, 0, 0])
2546: (8)                 b = a.compressed()
2547: (8)                 assert_equal(b, a)
2548: (8)                 a[0] = masked
2549: (8)                 b = a.compressed()
2550: (8)                 assert_equal(b, [2, 3, 4])
2551: (4)             def test_empty(self):
2552: (8)                 datatype = [('a', int), ('b', float), ('c', '|S8')]
2553: (8)                 a = masked_array([(1, 1.1, '1.1'), (2, 2.2, '2.2'), (3, 3.3, '3.3')], 
2554: (25)                   dtype=datatype)
2555: (8)                 assert_equal(len(a.fill_value.item()), len(datatype))
2556: (8)                 b = empty_like(a)
2557: (8)                 assert_equal(b.shape, a.shape)
2558: (8)                 assert_equal(b.fill_value, a.fill_value)
2559: (8)                 b = empty(len(a), dtype=datatype)
2560: (8)                 assert_equal(b.shape, a.shape)
2561: (8)                 assert_equal(b.fill_value, a.fill_value)
2562: (8)                 a = masked_array([1, 2, 3], mask=[False, True, False])
2563: (8)                 b = empty_like(a)
2564: (8)                 assert_(not np.may_share_memory(a.mask, b.mask))
2565: (8)                 b = a.view(masked_array)
2566: (8)                 assert_(np.may_share_memory(a.mask, b.mask))
2567: (4)             def test_zeros(self):
2568: (8)                 datatype = [('a', int), ('b', float), ('c', '|S8')]
2569: (8)                 a = masked_array([(1, 1.1, '1.1'), (2, 2.2, '2.2'), (3, 3.3, '3.3')], 
2570: (25)                   dtype=datatype)
2571: (8)                 assert_equal(len(a.fill_value.item()), len(datatype))
2572: (8)                 b = zeros(len(a), dtype=datatype)
2573: (8)                 assert_equal(b.shape, a.shape)
2574: (8)                 assert_equal(b.fill_value, a.fill_value)
2575: (8)                 b = zeros_like(a)
2576: (8)                 assert_equal(b.shape, a.shape)
2577: (8)                 assert_equal(b.fill_value, a.fill_value)
2578: (8)                 a = masked_array([1, 2, 3], mask=[False, True, False])

```

```

2579: (8)             b = zeros_like(a)
2580: (8)             assert_(not np.may_share_memory(a.mask, b.mask))
2581: (8)             b = a.view()
2582: (8)             assert_(np.may_share_memory(a.mask, b.mask))
2583: (4)             def test_ones(self):
2584: (8)                 datatype = [('a', int), ('b', float), ('c', '|S8')]
2585: (8)                 a = masked_array([(1, 1.1, '1.1'), (2, 2.2, '2.2'), (3, 3.3, '3.3')], 
2586: (25)                   dtype=datatype)
2587: (8)                 assert_equal(len(a.fill_value.item()), len(datatype))
2588: (8)                 b = ones(len(a), dtype=datatype)
2589: (8)                 assert_equal(b.shape, a.shape)
2590: (8)                 assert_equal(b.fill_value, a.fill_value)
2591: (8)                 b = ones_like(a)
2592: (8)                 assert_equal(b.shape, a.shape)
2593: (8)                 assert_equal(b.fill_value, a.fill_value)
2594: (8)                 a = masked_array([1, 2, 3], mask=[False, True, False])
2595: (8)                 b = ones_like(a)
2596: (8)                 assert_(not np.may_share_memory(a.mask, b.mask))
2597: (8)                 b = a.view()
2598: (8)                 assert_(np.may_share_memory(a.mask, b.mask))
2599: (4)             @suppress_copy_mask_on_assignment
2600: (4)             def test_put(self):
2601: (8)                 d = arange(5)
2602: (8)                 n = [0, 0, 0, 1, 1]
2603: (8)                 m = make_mask(n)
2604: (8)                 x = array(d, mask=m)
2605: (8)                 assert_(x[3] is masked)
2606: (8)                 assert_(x[4] is masked)
2607: (8)                 x[[1, 4]] = [10, 40]
2608: (8)                 assert_(x[3] is masked)
2609: (8)                 assert_(x[4] is not masked)
2610: (8)                 assert_equal(x, [0, 10, 2, -1, 40])
2611: (8)                 x = masked_array(arange(10), mask=[1, 0, 0, 0, 0] * 2)
2612: (8)                 i = [0, 2, 4, 6]
2613: (8)                 x.put(i, [6, 4, 2, 0])
2614: (8)                 assert_equal(x, asarray([6, 1, 4, 3, 2, 5, 0, 7, 8, 9, ]))
2615: (8)                 assert_equal(x.mask, [0, 0, 0, 0, 1, 0, 0, 0, 0])
2616: (8)                 x.put(i, masked_array([0, 2, 4, 6], [1, 0, 1, 0]))
2617: (8)                 assert_array_equal(x, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ])
2618: (8)                 assert_equal(x.mask, [1, 0, 0, 0, 1, 1, 0, 0, 0])
2619: (8)                 x = masked_array(arange(10), mask=[1, 0, 0, 0, 0] * 2)
2620: (8)                 put(x, i, [6, 4, 2, 0])
2621: (8)                 assert_equal(x, asarray([6, 1, 4, 3, 2, 5, 0, 7, 8, 9, ]))
2622: (8)                 assert_equal(x.mask, [0, 0, 0, 0, 0, 1, 0, 0, 0])
2623: (8)                 put(x, i, masked_array([0, 2, 4, 6], [1, 0, 1, 0]))
2624: (8)                 assert_array_equal(x, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ])
2625: (8)                 assert_equal(x.mask, [1, 0, 0, 0, 1, 1, 0, 0, 0])
2626: (4)             def test_put_nomask(self):
2627: (8)                 x = zeros(10)
2628: (8)                 z = array([3., -1.], mask=[False, True])
2629: (8)                 x.put([1, 2], z)
2630: (8)                 assert_(x[0] is not masked)
2631: (8)                 assert_equal(x[0], 0)
2632: (8)                 assert_(x[1] is not masked)
2633: (8)                 assert_equal(x[1], 3)
2634: (8)                 assert_(x[2] is masked)
2635: (8)                 assert_(x[3] is not masked)
2636: (8)                 assert_equal(x[3], 0)
2637: (4)             def test_put_hardmask(self):
2638: (8)                 d = arange(5)
2639: (8)                 n = [0, 0, 0, 1, 1]
2640: (8)                 m = make_mask(n)
2641: (8)                 xh = array(d + 1, mask=m, hard_mask=True, copy=True)
2642: (8)                 xh.put([4, 2, 0, 1, 3], [1, 2, 3, 4, 5])
2643: (8)                 assert_equal(xh._data, [3, 4, 2, 4, 5])
2644: (4)             def test_putmask(self):
2645: (8)                 x = arange(6) + 1
2646: (8)                 mx = array(x, mask=[0, 0, 0, 1, 1, 1])
2647: (8)                 mask = [0, 0, 1, 0, 0, 1]

```

```
2648: (8)          xx = x.copy()
2649: (8)          putmask(xx, mask, 99)
2650: (8)          assert_equal(xx, [1, 2, 99, 4, 5, 99])
2651: (8)          mxx = mx.copy()
2652: (8)          putmask(mxx, mask, 99)
2653: (8)          assert_equal(mxx._data, [1, 2, 99, 4, 5, 99])
2654: (8)          assert_equal(mxx._mask, [0, 0, 0, 1, 1, 0])
2655: (8)          values = array([10, 20, 30, 40, 50, 60], mask=[1, 1, 1, 0, 0, 0])
2656: (8)          xx = x.copy()
2657: (8)          putmask(xx, mask, values)
2658: (8)          assert_equal(xx._data, [1, 2, 30, 4, 5, 60])
2659: (8)          assert_equal(xx._mask, [0, 0, 1, 0, 0, 0])
2660: (8)          mxx = mx.copy()
2661: (8)          putmask(mxx, mask, values)
2662: (8)          assert_equal(mxx._data, [1, 2, 30, 4, 5, 60])
2663: (8)          assert_equal(mxx._mask, [0, 0, 1, 1, 1, 0])
2664: (8)          mxx = mx.copy()
2665: (8)          mxx.harden_mask()
2666: (8)          putmask(mxx, mask, values)
2667: (8)          assert_equal(mxx, [1, 2, 30, 4, 5, 60])
2668: (4)          def test_ravel(self):
2669: (8)          a = array([[1, 2, 3, 4, 5]], mask=[[0, 1, 0, 0, 0]])
2670: (8)          aravel = a.ravel()
2671: (8)          assert_equal(aravel._mask.shape, aravel.shape)
2672: (8)          a = array([0, 0], mask=[1, 1])
2673: (8)          aravel = a.ravel()
2674: (8)          assert_equal(aravel._mask.shape, a.shape)
2675: (8)          a = array([1, 2, 3, 4], mask=[0, 0, 0, 0], shrink=False)
2676: (8)          assert_equal(a.ravel()._mask, [0, 0, 0, 0])
2677: (8)          a.fill_value = -99
2678: (8)          a.shape = (2, 2)
2679: (8)          ar = a.ravel()
2680: (8)          assert_equal(ar._mask, [0, 0, 0, 0])
2681: (8)          assert_equal(ar._data, [1, 2, 3, 4])
2682: (8)          assert_equal(ar.fill_value, -99)
2683: (8)          assert_equal(a.ravel(order='C'), [1, 2, 3, 4])
2684: (8)          assert_equal(a.ravel(order='F'), [1, 3, 2, 4])
2685: (4)          @pytest.mark.parametrize("order", "AKCF")
2686: (4)          @pytest.mark.parametrize("data_order", "CF")
2687: (4)          def test_ravel_order(self, order, data_order):
2688: (8)          arr = np.ones((5, 10), order=data_order)
2689: (8)          arr[0, :] = 0
2690: (8)          mask = np.ones((10, 5), dtype=bool, order=data_order).T
2691: (8)          mask[0, :] = False
2692: (8)          x = array(arr, mask=mask)
2693: (8)          assert x._data.flags.fnc != x._mask.flags.fnc
2694: (8)          assert (x.filled(0) == 0).all()
2695: (8)          raveled = x.ravel(order)
2696: (8)          assert (raveled.filled(0) == 0).all()
2697: (8)          assert_array_equal(arr.ravel(order), x.ravel(order)._data)
2698: (4)          def test_reshape(self):
2699: (8)          x = arange(4)
2700: (8)          x[0] = masked
2701: (8)          y = x.reshape(2, 2)
2702: (8)          assert_equal(y.shape, (2, 2,))
2703: (8)          assert_equal(y._mask.shape, (2, 2,))
2704: (8)          assert_equal(x.shape, (4,))
2705: (8)          assert_equal(x._mask.shape, (4,))
2706: (4)          def test_sort(self):
2707: (8)          x = array([1, 4, 2, 3], mask=[0, 1, 0, 0], dtype=np.uint8)
2708: (8)          sortedx = sort(x)
2709: (8)          assert_equal(sortedx._data, [1, 2, 3, 4])
2710: (8)          assert_equal(sortedx._mask, [0, 0, 0, 1])
2711: (8)          sortedx = sort(x, endwith=False)
2712: (8)          assert_equal(sortedx._data, [4, 1, 2, 3])
2713: (8)          assert_equal(sortedx._mask, [1, 0, 0, 0])
2714: (8)          x.sort()
2715: (8)          assert_equal(x._data, [1, 2, 3, 4])
2716: (8)          assert_equal(x._mask, [0, 0, 0, 1])
```

```

2717: (8)           x = array([1, 4, 2, 3], mask=[0, 1, 0, 0], dtype=np.uint8)
2718: (8)           x.sort(endwith=False)
2719: (8)           assert_equal(x._data, [4, 1, 2, 3])
2720: (8)           assert_equal(x._mask, [1, 0, 0, 0])
2721: (8)           x = [1, 4, 2, 3]
2722: (8)           sortedx = sort(x)
2723: (8)           assert_(not isinstance(sorted, MaskedArray))
2724: (8)           x = array([0, 1, -1, -2, 2], mask=nomask, dtype=np.int8)
2725: (8)           sortedx = sort(x, endwith=False)
2726: (8)           assert_equal(sortedx._data, [-2, -1, 0, 1, 2])
2727: (8)           x = array([0, 1, -1, -2, 2], mask=[0, 1, 0, 0, 1], dtype=np.int8)
2728: (8)           sortedx = sort(x, endwith=False)
2729: (8)           assert_equal(sortedx._data, [1, 2, -2, -1, 0])
2730: (8)           assert_equal(sortedx._mask, [1, 1, 0, 0, 0])
2731: (8)           x = array([0, -1], dtype=np.int8)
2732: (8)           sortedx = sort(x, kind="stable")
2733: (8)           assert_equal(sortedx, array([-1, 0], dtype=np.int8))
2734: (4) def test_stable_sort(self):
2735: (8)     x = array([1, 2, 3, 1, 2, 3], dtype=np.uint8)
2736: (8)     expected = array([0, 3, 1, 4, 2, 5])
2737: (8)     computed = argsort(x, kind='stable')
2738: (8)     assert_equal(computed, expected)
2739: (4) def test_argsort_matches_sort(self):
2740: (8)     x = array([1, 4, 2, 3], mask=[0, 1, 0, 0], dtype=np.uint8)
2741: (8)     for kwargs in [dict(),
2742: (23)                 dict(endwith=True),
2743: (23)                 dict(endwith=False),
2744: (23)                 dict(fill_value=2),
2745: (23)                 dict(fill_value=2, endwith=True),
2746: (23)                 dict(fill_value=2, endwith=False)]:
2747: (12)         sortedx = sort(x, **kwargs)
2748: (12)         argsortedx = x[argsort(x, **kwargs)]
2749: (12)         assert_equal(sortedx._data, argsortedx._data)
2750: (12)         assert_equal(sortedx._mask, argsortedx._mask)
2751: (4) def test_sort_2d(self):
2752: (8)     a = masked_array([[8, 4, 1], [2, 0, 9]])
2753: (8)     a.sort(0)
2754: (8)     assert_equal(a, [[2, 0, 1], [8, 4, 9]])
2755: (8)     a = masked_array([[8, 4, 1], [2, 0, 9]])
2756: (8)     a.sort(1)
2757: (8)     assert_equal(a, [[1, 4, 8], [0, 2, 9]])
2758: (8)     a = masked_array([[8, 4, 1], [2, 0, 9]], mask=[[1, 0, 0], [0, 0, 1]])
2759: (8)     a.sort(0)
2760: (8)     assert_equal(a, [[2, 0, 1], [8, 4, 9]])
2761: (8)     assert_equal(a._mask, [[0, 0, 0], [1, 0, 1]])
2762: (8)     a = masked_array([[8, 4, 1], [2, 0, 9]], mask=[[1, 0, 0], [0, 0, 1]])
2763: (8)     a.sort(1)
2764: (8)     assert_equal(a, [[1, 4, 8], [0, 2, 9]])
2765: (8)     assert_equal(a._mask, [[0, 0, 1], [0, 0, 1]])
2766: (8)     a = masked_array([[[7, 8, 9], [4, 5, 6], [1, 2, 3]],
2767: (26)                   [[1, 2, 3], [7, 8, 9], [4, 5, 6]],
2768: (26)                   [[7, 8, 9], [1, 2, 3], [4, 5, 6]],
2769: (26)                   [[4, 5, 6], [1, 2, 3], [7, 8, 9]]])
2770: (8)     a[a % 4 == 0] = masked
2771: (8)     am = a.copy()
2772: (8)     an = a.filled(99)
2773: (8)     am.sort(0)
2774: (8)     an.sort(0)
2775: (8)     assert_equal(am, an)
2776: (8)     am = a.copy()
2777: (8)     an = a.filled(99)
2778: (8)     am.sort(1)
2779: (8)     an.sort(1)
2780: (8)     assert_equal(am, an)
2781: (8)     am = a.copy()
2782: (8)     an = a.filled(99)
2783: (8)     am.sort(2)
2784: (8)     an.sort(2)
2785: (8)     assert_equal(am, an)

```

```

2786: (4)
2787: (8)
2788: (12)
2789: (12)
2790: (12)
2791: (8)
2792: (12)
2793: (12)
2794: (12)
2795: (8)
2796: (12)
2797: (12)
2798: (12)
2799: (8)
2800: (8)
2801: (8)
2802: (8)
2803: (8)
2804: (8)
2805: (8)
2806: (8)
2807: (8)
2808: (8)
2809: (4)
2810: (8)
2811: (8)
2812: (4)
2813: (8)
2814: (8)
2815: (8)
2816: (8)
2817: (8)
2818: (8)
2819: (8)
2820: (8)
2821: (8)
2822: (8)
2823: (8)
2824: (8)
2825: (8)
2826: (8)
2827: (8)
2828: (8)
2829: (4)
2830: (8)
2831: (22)
2832: (22)
2833: (22)
2834: (22)
2835: (22)
2836: (8)
2837: (22)
2838: (22)
2839: (22)
2840: (22)
2841: (22)
2842: (8)
2843: (8)
2844: (8)
2845: (8)
2846: (8)
2847: (8)
2848: (4)
2849: (8)
2850: (8)
2851: (8)
2852: (8)
2853: (21)
2854: (8)

        def test_sort_flexible(self):
            a = array(
                data=[(3, 3), (3, 2), (2, 2), (2, 1), (1, 0), (1, 1), (1, 2)],
                mask=[(0, 0), (0, 1), (0, 0), (0, 0), (1, 0), (0, 0), (0, 0)],
                dtype=[('A', int), ('B', int)])
            mask_last = array(
                data=[(1, 1), (1, 2), (2, 1), (2, 2), (3, 3), (3, 2), (1, 0)],
                mask=[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 1), (1, 0)],
                dtype=[('A', int), ('B', int)])
            mask_first = array(
                data=[(1, 0), (1, 1), (1, 2), (2, 1), (2, 2), (3, 2), (3, 3)],
                mask=[(1, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 1), (0, 0)],
                dtype=[('A', int), ('B', int)])
            test = sort(a)
            assert_equal(test, mask_last)
            assert_equal(test.mask, mask_last.mask)
            test = sort(a, endwith=False)
            assert_equal(test, mask_first)
            assert_equal(test.mask, mask_first.mask)
            dt = np.dtype([('v', int, 2)])
            a = a.view(dt)
            test = sort(a)
            test = sort(a, endwith=False)
        def test_argsort(self):
            a = array([1, 5, 2, 4, 3], mask=[1, 0, 0, 1, 0])
            assert_equal(np.argsort(a), argsort(a))
        def test_squeeze(self):
            data = masked_array([[1, 2, 3]])
            assert_equal(data.squeeze(), [1, 2, 3])
            data = masked_array([[1, 2, 3]], mask=[[1, 1, 1]])
            assert_equal(data.squeeze(), [1, 2, 3])
            assert_equal(data.squeeze().__mask__, [1, 1, 1])
            arr = np.array([[1]])
            arr_sq = arr.squeeze()
            assert_equal(arr_sq, 1)
            arr_sq[...] = 2
            assert_equal(arr[0, 0], 2)
            m_arr = masked_array([[1]], mask=True)
            m_arr_sq = m_arr.squeeze()
            assert_(m_arr_sq is not np.ma.masked)
            assert_equal(m_arr_sq.__mask__, True)
            m_arr_sq[...] = 2
            assert_equal(m_arr[0, 0], 2)
        def test_swapaxes(self):
            x = np.array([8.375, 7.545, 8.828, 8.5, 1.757, 5.928,
                         8.43, 7.78, 9.865, 5.878, 8.979, 4.732,
                         3.012, 6.022, 5.095, 3.116, 5.238, 3.957,
                         6.04, 9.63, 7.712, 3.382, 4.489, 6.479,
                         7.189, 9.645, 5.395, 4.961, 9.894, 2.893,
                         7.357, 9.828, 6.272, 3.758, 6.693, 0.993])
            m = np.array([0, 1, 0, 1, 0, 0,
                         1, 0, 1, 1, 0, 1,
                         0, 0, 0, 1, 0, 1,
                         0, 0, 0, 1, 1, 1,
                         1, 0, 0, 1, 0, 0,
                         0, 0, 1, 0, 1, 0])
            mX = array(x, mask=m).reshape(6, 6)
            mXX = mX.reshape(3, 2, 2, 3)
            mXswapped = mX.swapaxes(0, 1)
            assert_equal(mXswapped[-1], mX[:, -1])
            mXXswapped = mXX.swapaxes(0, 2)
            assert_equal(mXXswapped.shape, (2, 2, 3, 3))
        def test_take(self):
            x = masked_array([10, 20, 30, 40], [0, 1, 0, 1])
            assert_equal(x.take([0, 0, 3]), masked_array([10, 10, 40], [0, 0, 1]))
            assert_equal(x.take([0, 0, 3]), x[[0, 0, 3]])
            assert_equal(x.take([[0, 1], [0, 1]]),
                         masked_array([[10, 20], [10, 20]], [[0, 1], [0, 1]]))
            assert_(x[1] is np.ma.masked)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

2855: (8) assert_(x.take(1) is np.ma.masked)
2856: (8) x = array([[10, 20, 30], [40, 50, 60]], mask=[[0, 0, 1], [1, 0, 0, ]])
2857: (8) assert_equal(x.take([0, 2], axis=1),
2858: (21)           array([[10, 30], [40, 60]], mask=[[0, 1], [1, 0]]))
2859: (8) assert_equal(take(x, [0, 2], axis=1),
2860: (21)           array([[10, 30], [40, 60]], mask=[[0, 1], [1, 0]]))
2861: (4) def test_take_masked_indices(self):
2862: (8)     a = np.array((40, 18, 37, 9, 22))
2863: (8)     indices = np.arange(3)[None,:] + np.arange(5)[:, None]
2864: (8)     mindices = array(indices, mask=(indices >= len(a)))
2865: (8)     test = take(a, mindices, mode='clip')
2866: (8)     ctrl = array([[40, 18, 37],
2867: (22)           [18, 37, 9],
2868: (22)           [37, 9, 22],
2869: (22)           [9, 22, 22],
2870: (22)           [22, 22, 22]])
2871: (8)     assert_equal(test, ctrl)
2872: (8)     test = take(a, mindices)
2873: (8)     ctrl = array([[40, 18, 37],
2874: (22)           [18, 37, 9],
2875: (22)           [37, 9, 22],
2876: (22)           [9, 22, 40],
2877: (22)           [22, 40, 40]])
2878: (8)     ctrl[3, 2] = ctrl[4, 1] = ctrl[4, 2] = masked
2879: (8)     assert_equal(test, ctrl)
2880: (8)     assert_equal(test.mask, ctrl.mask)
2881: (8)     a = array((40, 18, 37, 9, 22), mask=(0, 1, 0, 0, 0))
2882: (8)     test = take(a, mindices)
2883: (8)     ctrl[0, 1] = ctrl[1, 0] = masked
2884: (8)     assert_equal(test, ctrl)
2885: (8)     assert_equal(test.mask, ctrl.mask)
2886: (4) def test_tololist(self):
2887: (8)     x = array(np.arange(12))
2888: (8)     x[[1, -2]] = masked
2889: (8)     xlist = x.tolist()
2890: (8)     assert_(xlist[1] is None)
2891: (8)     assert_(xlist[-2] is None)
2892: (8)     x.shape = (3, 4)
2893: (8)     xlist = x.tolist()
2894: (8)     ctrl = [[0, None, 2, 3], [4, 5, 6, 7], [8, 9, None, 11]]
2895: (8)     assert_equal(xlist[0], [0, None, 2, 3])
2896: (8)     assert_equal(xlist[1], [4, 5, 6, 7])
2897: (8)     assert_equal(xlist[2], [8, 9, None, 11])
2898: (8)     assert_equal(xlist, ctrl)
2899: (8)     x = array(list(zip([1, 2, 3],
2900: (27)           [1.1, 2.2, 3.3],
2901: (27)           ['one', 'two', 'thr'])),
2902: (18)           dtype=[('a', int), ('b', float), ('c', '|S8')])
```

```

2903: (8)     x[-1] = masked
2904: (8)     assert_equal(x.tolist(),
2905: (21)           [(1, 1.1, b'one'),
2906: (22)             (2, 2.2, b'two'),
2907: (22)             (None, None, None)])
2908: (8)     a = array([(1, 2,), (3, 4)], mask=[(0, 1), (0, 0)],
2909: (18)           dtype=[('a', int), ('b', int)])
2910: (8)     test = a.tolist()
2911: (8)     assert_equal(test, [[1, None], [3, 4]])
2912: (8)     a = a[0]
2913: (8)     test = a.tolist()
2914: (8)     assert_equal(test, [1, None])
2915: (4) def test_tololist_specialcase(self):
2916: (8)     a = array([(0, 1), (2, 3)], dtype=[('a', int), ('b', int)])
2917: (8)     for entry in a:
2918: (12)         for item in entry.tolist():
2919: (16)             assert_(not isinstance(item, np.generic))
2920: (8)             a.mask[0] = (0, 1)
2921: (8)             for entry in a:
2922: (12)                 for item in entry.tolist():
2923: (16)                     assert_(not isinstance(item, np.generic))
```

```

2924: (4)             def test_toflex(self):
2925: (8)                 data = arange(10)
2926: (8)                 record = data.toflex()
2927: (8)                 assert_equal(record['_data'], data._data)
2928: (8)                 assert_equal(record['_mask'], data._mask)
2929: (8)                 data[[0, 1, 2, -1]] = masked
2930: (8)                 record = data.toflex()
2931: (8)                 assert_equal(record['_data'], data._data)
2932: (8)                 assert_equal(record['_mask'], data._mask)
2933: (8)                 ndtype = [('i', int), ('s', '|S3'), ('f', float)]
2934: (8)                 data = array([(i, s, f) for (i, s, f) in zip(np.arange(10),
2935: (53)                               'ABCDEFGHIJKLM',
2936: (53)                               np.random.rand(10))],
2937: (21)                               dtype=ndtype)
2938: (8)                 data[[0, 1, 2, -1]] = masked
2939: (8)                 record = data.toflex()
2940: (8)                 assert_equal(record['_data'], data._data)
2941: (8)                 assert_equal(record['_mask'], data._mask)
2942: (8)                 ndtype = np.dtype("int, (2,3)float, float")
2943: (8)                 data = array([(i, f, ff) for (i, f, ff) in zip(np.arange(10),
2944: (55)                               np.random.rand(10),
2945: (55)                               np.random.rand(10))],
2946: (21)                               dtype=ndtype)
2947: (8)                 data[[0, 1, 2, -1]] = masked
2948: (8)                 record = data.toflex()
2949: (8)                 assert_equal_records(record['_data'], data._data)
2950: (8)                 assert_equal_records(record['_mask'], data._mask)
2951: (4)             def test_fromflex(self):
2952: (8)                 a = array([1, 2, 3])
2953: (8)                 test = fromflex(a.toflex())
2954: (8)                 assert_equal(test, a)
2955: (8)                 assert_equal(test.mask, a.mask)
2956: (8)                 a = array([1, 2, 3], mask=[0, 0, 1])
2957: (8)                 test = fromflex(a.toflex())
2958: (8)                 assert_equal(test, a)
2959: (8)                 assert_equal(test.mask, a.mask)
2960: (8)                 a = array([(1, 1.), (2, 2.), (3, 3.)], mask=[(1, 0), (0, 0), (0, 1)],
2961: (18)                               dtype=[('A', int), ('B', float)])
2962: (8)                 test = fromflex(a.toflex())
2963: (8)                 assert_equal(test, a)
2964: (8)                 assert_equal(test.data, a.data)
2965: (4)             def test_arraymethod(self):
2966: (8)                 marray = masked_array([[1, 2, 3, 4, 5]], mask=[0, 0, 1, 0, 0])
2967: (8)                 control = masked_array([[1], [2], [3], [4], [5]],
2968: (31)                               mask=[0, 0, 1, 0, 0])
2969: (8)                 assert_equal(marray.T, control)
2970: (8)                 assert_equal(marray.transpose(), control)
2971: (8)                 assert_equal(MaskedArray.cumsum(marray.T, 0), control.cumsum(0))
2972: (4)             def test_arraymethod_0d(self):
2973: (8)                 x = np.ma.array(42, mask=True)
2974: (8)                 assert_equal(x.T.mask, x.mask)
2975: (8)                 assert_equal(x.T.data, x.data)
2976: (4)             def test_transpose_view(self):
2977: (8)                 x = np.ma.array([[1, 2, 3], [4, 5, 6]])
2978: (8)                 x[0,1] = np.ma.masked
2979: (8)                 xt = x.T
2980: (8)                 xt[1,0] = 10
2981: (8)                 xt[0,1] = np.ma.masked
2982: (8)                 assert_equal(x.data, xt.T.data)
2983: (8)                 assert_equal(x.mask, xt.T.mask)
2984: (4)             def test_diagonal_view(self):
2985: (8)                 x = np.ma.zeros((3,3))
2986: (8)                 x[0,0] = 10
2987: (8)                 x[1,1] = np.ma.masked
2988: (8)                 x[2,2] = 20
2989: (8)                 xd = x.diagonal()
2990: (8)                 x[1,1] = 15
2991: (8)                 assert_equal(xd.mask, x.diagonal().mask)
2992: (8)                 assert_equal(xd.data, x.diagonal().data)

```

```

2993: (0)
2994: (4)
2995: (8)
2996: (22)
2997: (22)
2998: (22)
2999: (22)
3000: (22)
3001: (8)
3002: (8)
3003: (8)
3004: (21)
3005: (21)
3006: (21)
3007: (21)
3008: (21)
3009: (8)
3010: (8)
3011: (8)
3012: (8)
3013: (22)
3014: (22)
3015: (22)
3016: (22)
3017: (22)
3018: (8)
3019: (8)
3020: (8)
3021: (8)
3022: (4)
3023: (8)
3024: (8)
3025: (8)
3026: (8)
3027: (8)
3028: (8)
3029: (8)
3030: (8)
3031: (8)
3032: (4)
3033: (8)
3034: (8)
3035: (8)
3036: (12)
3037: (12)
3038: (12)
3039: (12)
3040: (12)
3041: (12)
3042: (12)
3043: (12)
3044: (12)
3045: (12)
3046: (4)
3047: (8)
3048: (8)
3049: (8)
3050: (8)
3051: (8)
3052: (8)
3053: (12)
3054: (8)
3055: (12)
3056: (8)
3057: (8)
3058: (4)
3059: (8)
3060: (8)
3061: (8)

    class TestMaskedArrayMathMethods:
        def setup_method(self):
            x = np.array([8.375, 7.545, 8.828, 8.5, 1.757, 5.928,
                         8.43, 7.78, 9.865, 5.878, 8.979, 4.732,
                         3.012, 6.022, 5.095, 3.116, 5.238, 3.957,
                         6.04, 9.63, 7.712, 3.382, 4.489, 6.479,
                         7.189, 9.645, 5.395, 4.961, 9.894, 2.893,
                         7.357, 9.828, 6.272, 3.758, 6.693, 0.993])
            X = x.reshape(6, 6)
            XX = x.reshape(3, 2, 2, 3)
            m = np.array([0, 1, 0, 1, 0, 0,
                          1, 0, 1, 1, 0, 1,
                          0, 0, 0, 1, 0, 1,
                          0, 0, 0, 1, 1, 1,
                          1, 0, 0, 1, 0, 0,
                          0, 0, 1, 0, 1, 0])
            mx = array(data=x, mask=m)
            mX = array(data=X, mask=m.reshape(X.shape))
            mXX = array(data=XX, mask=m.reshape(XX.shape))
            m2 = np.array([1, 1, 0, 1, 0, 0,
                           1, 1, 1, 1, 0, 1,
                           0, 0, 1, 1, 0, 1,
                           0, 0, 0, 1, 1, 1,
                           1, 0, 0, 1, 1, 0,
                           0, 0, 1, 0, 1, 1])
            m2x = array(data=x, mask=m2)
            m2X = array(data=X, mask=m2.reshape(X.shape))
            m2XX = array(data=XX, mask=m2.reshape(XX.shape))
            self.d = (x, X, XX, m, mx, mX, mXX, m2x, m2X, m2XX)

        def test_cumsumprod(self):
            (x, X, XX, m, mx, mX, mXX, m2x, m2X, m2XX) = self.d
            mXcp = mX.cumsum(0)
            assert_equal(mXcp._data, mX.filled(0).cumsum(0))
            mXcp = mX.cumsum(1)
            assert_equal(mXcp._data, mX.filled(0).cumsum(1))
            mXcp = mX.cumprod(0)
            assert_equal(mXcp._data, mX.filled(1).cumprod(0))
            mXcp = mX.cumprod(1)
            assert_equal(mXcp._data, mX.filled(1).cumprod(1))

        def test_cumsumprod_with_output(self):
            xm = array(np.random.uniform(0, 10, 12)).reshape(3, 4)
            xm[:, 0] = xm[0] = xm[-1, -1] = masked
            for funcname in ('cumsum', 'cumprod'):
                npfunc = getattr(np, funcname)
                xmmeth = getattr(xm, funcname)
                output = np.empty((3, 4), dtype=float)
                output.fill(-9999)
                result = npfunc(xm, axis=0, out=output)
                assert_(result is output)
                assert_equal(result, xmmeth(axis=0, out=output))
                output = empty((3, 4), dtype=int)
                result = xmmeth(axis=0, out=output)
                assert_(result is output)

        def test_ptp(self):
            (x, X, XX, m, mx, mX, mXX, m2x, m2X, m2XX) = self.d
            (n, m) = X.shape
            assert_equal(mx.ptp(), mx.compressed().ptp())
            rows = np.zeros(n, float)
            cols = np.zeros(m, float)
            for k in range(m):
                cols[k] = mX[:, k].compressed().ptp()
            for k in range(n):
                rows[k] = mX[k].compressed().ptp()
            assert_equal(mX.ptp(0), cols)
            assert_equal(mX.ptp(1), rows)

        def test_add_object(self):
            x = masked_array(['a', 'b'], mask=[1, 0], dtype=object)
            y = x + 'x'
            assert_equal(y[1], 'bx')

```

```

3062: (8)             assert_(y.mask[0])
3063: (4)             def test_sum_object(self):
3064: (8)                 a = masked_array([1, 2, 3], mask=[1, 0, 0], dtype=object)
3065: (8)                 assert_equal(a.sum(), 5)
3066: (8)                 a = masked_array([[1, 2, 3], [4, 5, 6]], dtype=object)
3067: (8)                 assert_equal(a.sum(axis=0), [5, 7, 9])
3068: (4)             def test_prod_object(self):
3069: (8)                 a = masked_array([1, 2, 3], mask=[1, 0, 0], dtype=object)
3070: (8)                 assert_equal(a.prod(), 2 * 3)
3071: (8)                 a = masked_array([[1, 2, 3], [4, 5, 6]], dtype=object)
3072: (8)                 assert_equal(a.prod(axis=0), [4, 10, 18])
3073: (4)             def test_meananom_object(self):
3074: (8)                 a = masked_array([1, 2, 3], dtype=object)
3075: (8)                 assert_equal(a.mean(), 2)
3076: (8)                 assert_equal(a.anom(), [-1, 0, 1])
3077: (4)             def test_anom_shape(self):
3078: (8)                 a = masked_array([1, 2, 3])
3079: (8)                 assert_equal(a.anom().shape, a.shape)
3080: (8)                 a.mask = True
3081: (8)                 assert_equal(a.anom().shape, a.shape)
3082: (8)                 assert_(np.ma.is_masked(a.anom()))
3083: (4)             def test_anom(self):
3084: (8)                 a = masked_array(np.arange(1, 7).reshape(2, 3))
3085: (8)                 assert_almost_equal(a.anom(),
3086: (28)                             [[-2.5, -1.5, -0.5], [0.5, 1.5, 2.5]])
3087: (8)                 assert_almost_equal(a.anom(axis=0),
3088: (28)                             [[-1.5, -1.5, -1.5], [1.5, 1.5, 1.5]])
3089: (8)                 assert_almost_equal(a.anom(axis=1),
3090: (28)                             [[-1., 0., 1.], [-1., 0., 1.]])
3091: (8)                 a.mask = [[0, 0, 1], [0, 1, 0]]
3092: (8)                 mval = -99
3093: (8)                 assert_almost_equal(a.anom().filled(mval),
3094: (28)                             [[-2.25, -1.25, mval], [0.75, mval, 2.75]])
3095: (8)                 assert_almost_equal(a.anom(axis=0).filled(mval),
3096: (28)                             [[-1.5, 0.0, mval], [1.5, mval, 0.0]])
3097: (8)                 assert_almost_equal(a.anom(axis=1).filled(mval),
3098: (28)                             [[-0.5, 0.5, mval], [-1.0, mval, 1.0]])
3099: (4)             def test_trace(self):
3100: (8)                 (x, X, XX, m, mx, mXX, m2x, m2X, m2XX) = self.d
3101: (8)                 mXdiag = mX.diagonal()
3102: (8)                 assert_equal(mX.trace(), mX.diagonal().compressed().sum())
3103: (8)                 assert_almost_equal(mX.trace(),
3104: (28)                             X.trace() - sum(mXdiag.mask * X.diagonal(),
3105: (44)                               axis=0))
3106: (8)                 assert_equal(np.trace(mX), mX.trace())
3107: (8)                 arr = np.arange(2*4*4).reshape(2,4,4)
3108: (8)                 m_arr = np.ma.masked_array(arr, False)
3109: (8)                 assert_equal(arr.trace(axis1=1, axis2=2), m_arr.trace(axis1=1,
axis2=2))
3110: (4)             def test_dot(self):
3111: (8)                 (x, X, XX, m, mx, mX, mXX, m2x, m2X, m2XX) = self.d
3112: (8)                 fx = mx.filled(0)
3113: (8)                 r = mx.dot(mx)
3114: (8)                 assert_almost_equal(r.filled(0), fx.dot(fx))
3115: (8)                 assert_(r.mask is nomask)
3116: (8)                 fX = mX.filled(0)
3117: (8)                 r = mX.dot(mX)
3118: (8)                 assert_almost_equal(r.filled(0), fX.dot(fX))
3119: (8)                 assert_(r.mask[1,3])
3120: (8)                 r1 = empty_like(r)
3121: (8)                 mX.dot(mX, out=r1)
3122: (8)                 assert_almost_equal(r, r1)
3123: (8)                 mYY = mXX.swapaxes(-1, -2)
3124: (8)                 fXX, fYY = mXX.filled(0), mYY.filled(0)
3125: (8)                 r = mXX.dot(mYY)
3126: (8)                 assert_almost_equal(r.filled(0), fXX.dot(fYY))
3127: (8)                 r1 = empty_like(r)
3128: (8)                 mXX.dot(mYY, out=r1)
3129: (8)                 assert_almost_equal(r, r1)

```

```

3130: (4)
3131: (8)
3132: (8)
3133: (8)
3134: (8)
3135: (8)
3136: (8)
3137: (4)
3138: (8)
3139: (8)
3140: (8)
3141: (8)
3142: (8)
3143: (8)
3144: (8)
3145: (8)
3146: (8)
3147: (8)
3148: (8)
3149: (4)
3150: (8)
3151: (8)
3152: (8)
3153: (8)
3154: (28)
3155: (8)
3156: (28)
3157: (8)
3158: (8)
3159: (8)
3160: (8)
3161: (28)
3162: (8)
3163: (28)
3164: (8)
3165: (12)
3166: (12)
3167: (12)
3168: (32)
3169: (4)
3170: (4)
3171: (8)
3172: (8)
3173: (8)
3174: (8)
3175: (12)
3176: (12)
3177: (12)
3178: (12)
3179: (12)
3180: (12)
3181: (12)
3182: (12)
3183: (12)
3184: (8)
3185: (8)
3186: (8)
3187: (12)
3188: (12)
3189: (12)
3190: (12)
3191: (12)
3192: (12)
3193: (12)
3194: (12)
3195: (12)
3196: (4)
3197: (8)
3198: (8)

        def test_dot_shape_mismatch(self):
            x = masked_array([[1,2],[3,4]], mask=[[0,1],[0,0]])
            y = masked_array([[1,2],[3,4]], mask=[[0,1],[0,0]])
            z = masked_array([[0,1],[3,3]])
            x.dot(y, out=z)
            assert_almost_equal(z.filled(0), [[1, 0], [15, 16]])
            assert_almost_equal(z.mask, [[0, 1], [0, 0]])

        def test_varmean_nomask(self):
            foo = array([1,2,3,4], dtype='f8')
            bar = array([1,2,3,4], dtype='f8')
            assert_equal(type(foo.mean()), np.float64)
            assert_equal(type(foo.var()), np.float64)
            assert((foo.mean() == bar.mean()) is np.bool_(True))
            foo = array(np.arange(16).reshape((4,4)), dtype='f8')
            bar = empty(4, dtype='f4')
            assert_equal(type(foo.mean(axis=1)), MaskedArray)
            assert_equal(type(foo.var(axis=1)), MaskedArray)
            assert_(foo.mean(axis=1, out=bar) is bar)
            assert_(foo.var(axis=1, out=bar) is bar)

        def test_varstd(self):
            (x, X, XX, m, mx, mXX, m2x, m2X, m2XX) = self.d
            assert_almost_equal(mX.var(axis=None), mX.compressed().var())
            assert_almost_equal(mX.std(axis=None), mX.compressed().std())
            assert_almost_equal(mX.std(axis=None, ddof=1),
                                mX.compressed().std(ddof=1))
            assert_almost_equal(mX.var(axis=None, ddof=1),
                                mX.compressed().var(ddof=1))
            assert_equal(mXX.var(axis=3).shape, XX.var(axis=3).shape)
            assert_equal(mX.var().shape, X.var().shape)
            (mXvar0, mXvar1) = (mX.var(axis=0), mX.var(axis=1))
            assert_almost_equal(mX.var(axis=None, ddof=2),
                                mX.compressed().var(ddof=2))
            assert_almost_equal(mX.std(axis=None, ddof=2),
                                mX.compressed().std(ddof=2))
            for k in range(6):
                assert_almost_equal(mXvar1[k], mX[k].compressed().var())
                assert_almost_equal(mXvar0[k], mX[:, k].compressed().var())
                assert_almost_equal(np.sqrt(mXvar0[k]),
                                    mX[:, k].compressed().std())

    @suppress_copy_mask_on_assignment
    def test_varstd_specialcases(self):
        nout = np.array(-1, dtype=float)
        mout = array(-1, dtype=float)
        x = array(arange(10), mask=True)
        for methodname in ('var', 'std'):
            method = getattr(x, methodname)
            assert_(method() is masked)
            assert_(method(0) is masked)
            assert_(method(-1) is masked)
            method(out=mout)
            assert_(mout is not masked)
            assert_equal(mout.mask, True)
            method(out=nout)
            assert_(np.isnan(nout))
        x = array(arange(10), mask=True)
        x[-1] = 9
        for methodname in ('var', 'std'):
            method = getattr(x, methodname)
            assert_(method(ddof=1) is masked)
            assert_(method(0, ddof=1) is masked)
            assert_(method(-1, ddof=1) is masked)
            method(out=mout, ddof=1)
            assert_(mout is not masked)
            assert_equal(mout.mask, True)
            method(out=nout, ddof=1)
            assert_(np.isnan(nout))

    def test_varstd_ddof(self):
        a = array([[1, 1, 0], [1, 1, 0]], mask=[[0, 0, 1], [0, 0, 1]])
        test = a.std(axis=0, ddof=0)

```

```

3199: (8)             assert_equal(test.filled(0), [0, 0, 0])
3200: (8)             assert_equal(test.mask, [0, 0, 1])
3201: (8)             test = a.std(axis=0, ddof=1)
3202: (8)             assert_equal(test.filled(0), [0, 0, 0])
3203: (8)             assert_equal(test.mask, [0, 0, 1])
3204: (8)             test = a.std(axis=0, ddof=2)
3205: (8)             assert_equal(test.filled(0), [0, 0, 0])
3206: (8)             assert_equal(test.mask, [1, 1, 1])
3207: (4)             def test_diag(self):
3208: (8)                 x = arange(9).reshape((3, 3))
3209: (8)                 x[1, 1] = masked
3210: (8)                 out = np.diag(x)
3211: (8)                 assert_equal(out, [0, 4, 8])
3212: (8)                 out = diag(x)
3213: (8)                 assert_equal(out, [0, 4, 8])
3214: (8)                 assert_equal(out.mask, [0, 1, 0])
3215: (8)                 out = diag(out)
3216: (8)                 control = array([[0, 0, 0], [0, 4, 0], [0, 0, 8]],
3217: (24)                   mask=[[0, 0, 0], [0, 1, 0], [0, 0, 0]])
3218: (8)                 assert_equal(out, control)
3219: (4)             def test_axis_methods_nomask(self):
3220: (8)                 a = array([[1, 2, 3], [4, 5, 6]])
3221: (8)                 assert_equal(a.sum(0), [5, 7, 9])
3222: (8)                 assert_equal(a.sum(-1), [6, 15])
3223: (8)                 assert_equal(a.sum(1), [6, 15])
3224: (8)                 assert_equal(a.prod(0), [4, 10, 18])
3225: (8)                 assert_equal(a.prod(-1), [6, 120])
3226: (8)                 assert_equal(a.prod(1), [6, 120])
3227: (8)                 assert_equal(a.min(0), [1, 2, 3])
3228: (8)                 assert_equal(a.min(-1), [1, 4])
3229: (8)                 assert_equal(a.min(1), [1, 4])
3230: (8)                 assert_equal(a.max(0), [4, 5, 6])
3231: (8)                 assert_equal(a.max(-1), [3, 6])
3232: (8)                 assert_equal(a.max(1), [3, 6])
3233: (4)             @requires_memory(free_bytes=2 * 10000 * 1000 * 2)
3234: (4)             def test_mean_overflow(self):
3235: (8)                 a = masked_array(np.full((10000, 10000), 65535, dtype=np.uint16),
3236: (25)                   mask=np.zeros((10000, 10000)))
3237: (8)                 assert_equal(a.mean(), 65535.0)
3238: (4)             def test_diff_with_prepend(self):
3239: (8)                 x = np.array([1, 2, 2, 3, 4, 2, 1, 1])
3240: (8)                 a = np.ma.masked_equal(x[3:], value=2)
3241: (8)                 a_prep = np.ma.masked_equal(x[:3], value=2)
3242: (8)                 diff1 = np.ma.diff(a, prepend=a_prep, axis=0)
3243: (8)                 b = np.ma.masked_equal(x, value=2)
3244: (8)                 diff2 = np.ma.diff(b, axis=0)
3245: (8)                 assert_(np.ma.allequal(diff1, diff2))
3246: (4)             def test_diff_with_append(self):
3247: (8)                 x = np.array([1, 2, 2, 3, 4, 2, 1, 1])
3248: (8)                 a = np.ma.masked_equal(x[:3], value=2)
3249: (8)                 a_app = np.ma.masked_equal(x[3:], value=2)
3250: (8)                 diff1 = np.ma.diff(a, append=a_app, axis=0)
3251: (8)                 b = np.ma.masked_equal(x, value=2)
3252: (8)                 diff2 = np.ma.diff(b, axis=0)
3253: (8)                 assert_(np.ma.allequal(diff1, diff2))
3254: (4)             def test_diff_with_dim_0(self):
3255: (8)                 with pytest.raises(
3256: (12)                     ValueError,
3257: (12)                     match="diff requires input that is at least one dimensional"
3258: (12)                     ):
3259: (12)                     np.ma.diff(np.array(1))
3260: (4)             def test_diff_with_n_0(self):
3261: (8)                 a = np.ma.masked_equal([1, 2, 2, 3, 4, 2, 1, 1], value=2)
3262: (8)                 diff = np.ma.diff(a, n=0, axis=0)
3263: (8)                 assert_(np.ma.allequal(a, diff))
3264: (0)             class TestMaskedArrayMathMethodsComplex:
3265: (4)                 def setup_method(self):
3266: (8)                     x = np.array([8.375j, 7.545j, 8.828j, 8.5j, 1.757j, 5.928,
3267: (22)                       8.43, 7.78, 9.865, 5.878, 8.979, 4.732,

```

```

3268: (22)           3.012, 6.022, 5.095, 3.116, 5.238, 3.957,
3269: (22)           6.04, 9.63, 7.712, 3.382, 4.489, 6.479j,
3270: (22)           7.189j, 9.645, 5.395, 4.961, 9.894, 2.893,
3271: (22)           7.357, 9.828, 6.272, 3.758, 6.693, 0.993j])
3272: (8)            X = x.reshape(6, 6)
3273: (8)            XX = x.reshape(3, 2, 2, 3)
3274: (8)            m = np.array([0, 1, 0, 1, 0, 0,
3275: (21)              1, 0, 1, 1, 0, 1,
3276: (21)              0, 0, 0, 1, 0, 1,
3277: (21)              0, 0, 0, 1, 1, 1,
3278: (21)              1, 0, 0, 1, 0, 0,
3279: (21)              0, 0, 1, 0, 1, 0])
3280: (8)            mx = array(data=x, mask=m)
3281: (8)            mX = array(data=X, mask=m.reshape(X.shape))
3282: (8)            mXX = array(data=XX, mask=m.reshape(XX.shape))
3283: (8)            m2 = np.array([1, 1, 0, 1, 0, 0,
3284: (22)              1, 1, 1, 1, 0, 1,
3285: (22)              0, 0, 1, 1, 0, 1,
3286: (22)              0, 0, 0, 1, 1, 1,
3287: (22)              1, 0, 0, 1, 1, 0,
3288: (22)              0, 0, 1, 0, 1, 1])
3289: (8)            m2x = array(data=x, mask=m2)
3290: (8)            m2X = array(data=X, mask=m2.reshape(X.shape))
3291: (8)            m2XX = array(data=XX, mask=m2.reshape(XX.shape))
3292: (8)            self.d = (x, X, XX, m, mx, mX, mXX, m2x, m2X, m2XX)
3293: (4)             def test_varstd(self):
3294: (8)               (x, X, XX, m, mx, mX, mXX, m2x, m2X, m2XX) = self.d
3295: (8)               assert_almost_equal(mX.var(axis=None), mX.compressed().var())
3296: (8)               assert_almost_equal(mX.std(axis=None), mX.compressed().std())
3297: (8)               assert_equal(mXX.var(axis=3).shape, XX.var(axis=3).shape)
3298: (8)               assert_equal(mX.var().shape, X.var().shape)
3299: (8)               (mXvar0, mXvar1) = (mX.var(axis=0), mX.var(axis=1))
3300: (8)               assert_almost_equal(mX.var(axis=None, ddof=2),
3301: (28)                 mX.compressed().var(ddof=2))
3302: (8)               assert_almost_equal(mX.std(axis=None, ddof=2),
3303: (28)                 mX.compressed().std(ddof=2))
3304: (8)               for k in range(6):
3305: (12)                 assert_almost_equal(mXvar1[k], mX[k].compressed().var())
3306: (12)                 assert_almost_equal(mXvar0[k], mX[:, k].compressed().var())
3307: (12)                 assert_almost_equal(np.sqrt(mXvar0[k]),
3308: (32)                   mX[:, k].compressed().std())
3309: (0)              class TestMaskedArrayFunctions:
3310: (4)                def setup_method(self):
3311: (8)                  x = np.array([1., 1., 1., -2., pi/2.0, 4., 5., -10., 10., 1., 2., 3.])
3312: (8)                  y = np.array([5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.])
3313: (8)                  m1 = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
3314: (8)                  m2 = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1]
3315: (8)                  xm = masked_array(x, mask=m1)
3316: (8)                  ym = masked_array(y, mask=m2)
3317: (8)                  xm.set_fill_value(1e+20)
3318: (8)                  self.info = (xm, ym)
3319: (4)                def test_masked_where_bool(self):
3320: (8)                  x = [1, 2]
3321: (8)                  y = masked_where(False, x)
3322: (8)                  assert_equal(y, [1, 2])
3323: (8)                  assert_equal(y[1], 2)
3324: (4)                def test_masked_equal_wlist(self):
3325: (8)                  x = [1, 2, 3]
3326: (8)                  mx = masked_equal(x, 3)
3327: (8)                  assert_equal(mx, x)
3328: (8)                  assert_equal(mx._mask, [0, 0, 1])
3329: (8)                  mx = masked_not_equal(x, 3)
3330: (8)                  assert_equal(mx, x)
3331: (8)                  assert_equal(mx._mask, [1, 1, 0])
3332: (4)                def test_masked_equal_fill_value(self):
3333: (8)                  x = [1, 2, 3]
3334: (8)                  mx = masked_equal(x, 3)
3335: (8)                  assert_equal(mx._mask, [0, 0, 1])
3336: (8)                  assert_equal(mx.fill_value, 3)

```

```

3337: (4) def test_masked_where_condition(self):
3338: (8)     x = array([1., 2., 3., 4., 5.])
3339: (8)     x[2] = masked
3340: (8)     assert_equal(masked_where(greater(x, 2), x), masked_greater(x, 2))
3341: (8)     assert_equal(masked_where(greater_equal(x, 2), x),
3342: (21)             masked_greater_equal(x, 2))
3343: (8)     assert_equal(masked_where(less(x, 2), x), masked_less(x, 2))
3344: (8)     assert_equal(masked_where(less_equal(x, 2), x),
3345: (21)             masked_less_equal(x, 2))
3346: (8)     assert_equal(masked_where(not_equal(x, 2), x), masked_not_equal(x, 2))
3347: (8)     assert_equal(masked_where(equal(x, 2), x), masked_equal(x, 2))
3348: (8)     assert_equal(masked_where(not_equal(x, 2), x), masked_not_equal(x, 2))
3349: (8)     assert_equal(masked_where([1, 1, 0, 0, 0], [1, 2, 3, 4, 5]),
3350: (21)             [99, 99, 3, 4, 5])
3351: (4) def test_masked_where_oddities(self):
3352: (8)     atest = ones((10, 10, 10), dtype=float)
3353: (8)     btest = zeros(atest.shape, MaskType)
3354: (8)     ctest = masked_where(btest, atest)
3355: (8)     assert_equal(atest, ctest)
3356: (4) def test_masked_where_shape_constraint(self):
3357: (8)     a = arange(10)
3358: (8)     with assert_raises(IndexError):
3359: (12)         masked_equal(1, a)
3360: (8)     test = masked_equal(a, 1)
3361: (8)     assert_equal(test.mask, [0, 1, 0, 0, 0, 0, 0, 0, 0, 0])
3362: (4) def test_masked_where_structured(self):
3363: (8)     a = np.zeros(10, dtype=[("A", "<f2"), ("B", "<f4")])
3364: (8)     with np.errstate(over="ignore"):
3365: (12)         am = np.ma.masked_where(a["A"] < 5, a)
3366: (8)         assert_equal(am.mask.dtype.names, am.dtype.names)
3367: (8)         assert_equal(am["A"],
3368: (20)             np.ma.masked_array(np.zeros(10), np.ones(10)))
3369: (4) def test_masked_where_mismatch(self):
3370: (8)     x = np.arange(10)
3371: (8)     y = np.arange(5)
3372: (8)     assert_raises(IndexError, np.ma.masked_where, y > 6, x)
3373: (4) def test_masked_otherfunctions(self):
3374: (8)     assert_equal(masked_inside(list(range(5)), 1, 3),
3375: (21)             [0, 199, 199, 199, 4])
3376: (8)     assert_equal(masked_outside(list(range(5)), 1, 3), [199, 1, 2, 3,
3377: (199])
3378: (8)     assert_equal(masked_inside(array(list(range(5))),
3379: (41)             mask=[1, 0, 0, 0, 0]), 1, 3).mask,
3380: (21)     [1, 1, 1, 1, 0])
3381: (8)     assert_equal(masked_outside(array(list(range(5))),
3382: (42)             mask=[0, 1, 0, 0, 0]), 1, 3).mask,
3383: (21)     [1, 1, 0, 0, 1])
3384: (8)     assert_equal(masked_equal(array(list(range(5))),
3385: (40)             mask=[1, 0, 0, 0, 0]), 2).mask,
3386: (21)     [1, 0, 1, 0, 0])
3387: (8)     assert_equal(masked_not_equal(array([2, 2, 1, 2, 1]),
3388: (44)             mask=[1, 0, 0, 0, 0]), 2).mask,
3389: (21)     [1, 0, 1, 0, 1])
3390: (4) def test_round(self):
3391: (8)     a = array([1.23456, 2.34567, 3.45678, 4.56789, 5.67890],
3392: (18)             mask=[0, 1, 0, 0, 0])
3393: (8)     assert_equal(a.round(), [1., 2., 3., 5., 6.])
3394: (8)     assert_equal(a.round(1), [1.2, 2.3, 3.5, 4.6, 5.7])
3395: (8)     assert_equal(a.round(3), [1.235, 2.346, 3.457, 4.568, 5.679])
3396: (8)     b = empty_like(a)
3397: (8)     a.round(out=b)
3398: (8)     assert_equal(b, [1., 2., 3., 5., 6.])
3399: (8)     x = array([1., 2., 3., 4., 5.])
3400: (8)     c = array([1, 1, 1, 0, 0])
3401: (8)     x[2] = masked
3402: (8)     z = where(c, x, -x)
3403: (8)     assert_equal(z, [1., 2., 0., -4., -5])
3404: (8)     c[0] = masked
3405: (8)     z = where(c, x, -x)

```

```

3405: (8)             assert_equal(z, [1., 2., 0., -4., -5])
3406: (8)             assert_(z[0] is masked)
3407: (8)             assert_(z[1] is not masked)
3408: (8)             assert_(z[2] is masked)
3409: (4)             def test_round_with_output(self):
3410: (8)                 xm = array(np.random.uniform(0, 10, 12)).reshape(3, 4)
3411: (8)                 xm[:, 0] = xm[0] = xm[-1, -1] = masked
3412: (8)                 output = np.empty((3, 4), dtype=float)
3413: (8)                 output.fill(-9999)
3414: (8)                 result = np.round(xm, decimals=2, out=output)
3415: (8)                 assert_(result is output)
3416: (8)                 assert_equal(result, xm.round(decimals=2, out=output))
3417: (8)                 output = empty((3, 4), dtype=float)
3418: (8)                 result = xm.round(decimals=2, out=output)
3419: (8)                 assert_(result is output)
3420: (4)             def test_round_with_scalar(self):
3421: (8)                 a = array(1.1, mask=[False])
3422: (8)                 assert_equal(a.round(), 1)
3423: (8)                 a = array(1.1, mask=[True])
3424: (8)                 assert_(a.round() is masked)
3425: (8)                 a = array(1.1, mask=[False])
3426: (8)                 output = np.empty(1, dtype=float)
3427: (8)                 output.fill(-9999)
3428: (8)                 a.round(out=output)
3429: (8)                 assert_equal(output, 1)
3430: (8)                 a = array(1.1, mask=[False])
3431: (8)                 output = array(-9999., mask=[True])
3432: (8)                 a.round(out=output)
3433: (8)                 assert_equal(output[()], 1)
3434: (8)                 a = array(1.1, mask=[True])
3435: (8)                 output = array(-9999., mask=[False])
3436: (8)                 a.round(out=output)
3437: (8)                 assert_(output[()] is masked)
3438: (4)             def test_identity(self):
3439: (8)                 a = identity(5)
3440: (8)                 assert_(isinstance(a, MaskedArray))
3441: (8)                 assert_equal(a, np.identity(5))
3442: (4)             def test_power(self):
3443: (8)                 x = -1.1
3444: (8)                 assert_almost_equal(power(x, 2.), 1.21)
3445: (8)                 assert_(power(x, masked) is masked)
3446: (8)                 x = array([-1.1, -1.1, 1.1, 1.1, 0.])
3447: (8)                 b = array([0.5, 2., 0.5, 2., -1.], mask=[0, 0, 0, 0, 1])
3448: (8)                 y = power(x, b)
3449: (8)                 assert_almost_equal(y, [0, 1.21, 1.04880884817, 1.21, 0.])
3450: (8)                 assert_equal(y._mask, [1, 0, 0, 0, 1])
3451: (8)                 b.mask = nomask
3452: (8)                 y = power(x, b)
3453: (8)                 assert_equal(y._mask, [1, 0, 0, 0, 1])
3454: (8)                 z = x ** b
3455: (8)                 assert_equal(z._mask, y._mask)
3456: (8)                 assert_almost_equal(z, y)
3457: (8)                 assert_almost_equal(z._data, y._data)
3458: (8)                 x **= b
3459: (8)                 assert_equal(x._mask, y._mask)
3460: (8)                 assert_almost_equal(x, y)
3461: (8)                 assert_almost_equal(x._data, y._data)
3462: (4)             def test_power_with_broadcasting(self):
3463: (8)                 a2 = np.array([[1., 2., 3.], [4., 5., 6.]])
3464: (8)                 a2m = array(a2, mask=[[1, 0, 0], [0, 0, 1]])
3465: (8)                 b1 = np.array([2, 4, 3])
3466: (8)                 b2 = np.array([b1, b1])
3467: (8)                 b2m = array(b2, mask=[[0, 1, 0], [0, 1, 0]])
3468: (8)                 ctrl = array([[1 ** 2, 2 ** 4, 3 ** 3], [4 ** 2, 5 ** 4, 6 ** 3]], 
3469: (21)                               mask=[[1, 1, 0], [0, 1, 1]])
3470: (8)                 test = a2m ** b2m
3471: (8)                 assert_equal(test, ctrl)
3472: (8)                 assert_equal(test.mask, ctrl.mask)
3473: (8)                 test = a2m ** b2

```

```

3474: (8)             assert_equal(test, ctrl)
3475: (8)             assert_equal(test.mask, a2m.mask)
3476: (8)             test = a2 ** b2m
3477: (8)             assert_equal(test, ctrl)
3478: (8)             assert_equal(test.mask, b2m.mask)
3479: (8)             ctrl = array([[2 ** 2, 4 ** 4, 3 ** 3], [2 ** 2, 4 ** 4, 3 ** 3]], mask=[[0, 1, 0], [0, 1, 0]])
3480: (21)            test = b1 ** b2m
3481: (8)             assert_equal(test, ctrl)
3482: (8)             assert_equal(test.mask, ctrl.mask)
3483: (8)             test = b2m ** b1
3484: (8)             assert_equal(test, ctrl)
3485: (8)             assert_equal(test.mask, ctrl.mask)
3486: (8)             assert_equal(test.mask, ctrl.mask)
3487: (4) @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
3488: (4) def test_where(self):
3489: (8)     x = np.array([1., 1., 1., -2., pi/2.0, 4., 5., -10., 10., 1., 2., 3.])
3490: (8)     y = np.array([5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.])
3491: (8)     m1 = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
3492: (8)     m2 = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1]
3493: (8)     xm = masked_array(x, mask=m1)
3494: (8)     ym = masked_array(y, mask=m2)
3495: (8)     xm.set_fill_value(1e+20)
3496: (8)     d = where(xm > 2, xm, -9)
3497: (8)     assert_equal(d, [-9., -9., -9., -9., -9., 4., -9., -9., 10., -9., -9., 3.])
3498: (25)            assert_equal(d._mask, xm._mask)
3499: (8)     d = where(xm > 2, -9, ym)
3500: (8)     assert_equal(d, [5., 0., 3., 2., -1., -9., -9., -10., -9., 1., 0., -9.])
3501: (8)     assert_equal(d._mask, [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0])
3502: (25)            d = where(xm > 2, xm, masked)
3503: (8)     assert_equal(d, [-9., -9., -9., -9., -9., 4., -9., -9., 10., -9., -9., 3.])
3504: (8)     assert_equal(d._mask, [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0])
3505: (8)     tmp = xm._mask.copy()
3506: (25)            tmp[(xm <= 2).filled(True)] = True
3507: (8)     assert_equal(d._mask, tmp)
3508: (8)     with np.errstate(invalid="warn"):
3509: (8)         with pytest.warns(RuntimeWarning, match="invalid value"):
3510: (8)             ixm = xm.astype(int)
3511: (12)            d = where(ixm > 2, ixm, masked)
3512: (16)            assert_equal(d, [-9, -9, -9, -9, -9, 4, -9, -9, 10, -9, -9, 3])
3513: (8)             assert_equal(d.dtype, ixm.dtype)
3514: (8)     def test_where_object(self):
3515: (8)         a = np.array(None)
3516: (4)         b = masked_array(None)
3517: (8)         r = b.copy()
3518: (8)         assert_equal(np.ma.where(True, a, a), r)
3519: (8)         assert_equal(np.ma.where(True, b, b), r)
3520: (8)     def test_where_with_masked_choice(self):
3521: (8)         x = arange(10)
3522: (4)         x[3] = masked
3523: (8)         c = x >= 8
3524: (8)         z = where(c, x, masked)
3525: (8)         assert_(z.dtype is x.dtype)
3526: (8)         assert_(z[3] is masked)
3527: (8)         assert_(z[4] is masked)
3528: (8)         assert_(z[7] is masked)
3529: (8)         assert_(z[8] is not masked)
3530: (8)         assert_(z[9] is not masked)
3531: (8)         assert_equal(x, z)
3532: (8)         z = where(c, masked, x)
3533: (8)         assert_(z.dtype is x.dtype)
3534: (8)         assert_(z[3] is masked)
3535: (8)         assert_(z[4] is not masked)
3536: (8)         assert_(z[7] is not masked)
3537: (8)         assert_(z[8] is masked)
3538: (8)         assert_(z[9] is masked)
3539: (8)     def test_where_with_masked_condition(self):
3540: (8)         x = array([1., 2., 3., 4., 5.])

```

```

3543: (8)             c = array([1, 1, 1, 0, 0])
3544: (8)             x[2] = masked
3545: (8)             z = where(c, x, -x)
3546: (8)             assert_equal(z, [1., 2., 0., -4., -5])
3547: (8)             c[0] = masked
3548: (8)             z = where(c, x, -x)
3549: (8)             assert_equal(z, [1., 2., 0., -4., -5])
3550: (8)             assert_(z[0] is masked)
3551: (8)             assert_(z[1] is not masked)
3552: (8)             assert_(z[2] is masked)
3553: (8)             x = arange(1, 6)
3554: (8)             x[-1] = masked
3555: (8)             y = arange(1, 6) * 10
3556: (8)             y[2] = masked
3557: (8)             c = array([1, 1, 1, 0, 0], mask=[1, 0, 0, 0, 0])
3558: (8)             cm = c.filled(1)
3559: (8)             z = where(c, x, y)
3560: (8)             zm = where(cm, x, y)
3561: (8)             assert_equal(z, zm)
3562: (8)             assert_(getmask(zm) is nomask)
3563: (8)             assert_equal(zm, [1, 2, 3, 40, 50])
3564: (8)             z = where(c, masked, 1)
3565: (8)             assert_equal(z, [99, 99, 99, 1, 1])
3566: (8)             z = where(c, 1, masked)
3567: (8)             assert_equal(z, [99, 1, 1, 99, 99])
3568: (4)              def test_where_type(self):
3569: (8)                  x = np.arange(4, dtype=np.int32)
3570: (8)                  y = np.arange(4, dtype=np.float32) * 2.2
3571: (8)                  test = where(x > 1.5, y, x).dtype
3572: (8)                  control = np.result_type(np.int32, np.float32)
3573: (8)                  assert_equal(test, control)
3574: (4)              def test_where_broadcast(self):
3575: (8)                  x = np.arange(9).reshape(3, 3)
3576: (8)                  y = np.zeros(3)
3577: (8)                  core = np.where([1, 0, 1], x, y)
3578: (8)                  ma = where([1, 0, 1], x, y)
3579: (8)                  assert_equal(core, ma)
3580: (8)                  assert_equal(core.dtype, ma.dtype)
3581: (4)              def test_where_structured(self):
3582: (8)                  dt = np.dtype([('a', int), ('b', int)])
3583: (8)                  x = np.array([(1, 2), (3, 4), (5, 6)], dtype=dt)
3584: (8)                  y = np.array((10, 20), dtype=dt)
3585: (8)                  core = np.where([0, 1, 1], x, y)
3586: (8)                  ma = np.where([0, 1, 1], x, y)
3587: (8)                  assert_equal(core, ma)
3588: (8)                  assert_equal(core.dtype, ma.dtype)
3589: (4)              def test_where_structured_masked(self):
3590: (8)                  dt = np.dtype([('a', int), ('b', int)])
3591: (8)                  x = np.array([(1, 2), (3, 4), (5, 6)], dtype=dt)
3592: (8)                  ma = where([0, 1, 1], x, masked)
3593: (8)                  expected = masked_where([1, 0, 0], x)
3594: (8)                  assert_equal(ma.dtype, expected.dtype)
3595: (8)                  assert_equal(ma, expected)
3596: (8)                  assert_equal(ma.mask, expected.mask)
3597: (4)              def test_masked_invalid_error(self):
3598: (8)                  a = np.arange(5, dtype=object)
3599: (8)                  a[3] = np.PINF
3600: (8)                  a[2] = np.NaN
3601: (8)                  with pytest.raises(TypeError,
3602: (27)                      match="not supported for the input types"):
3603: (12)                  np.ma.masked_invalid(a)
3604: (4)              def test_masked_invalid_pandas(self):
3605: (8)                  class Series():
3606: (12)                      _data = "nonsense"
3607: (12)                      def __array__(self):
3608: (16)                          return np.array([5, np.nan, np.inf])
3609: (8)                      arr = np.ma.masked_invalid(Series())
3610: (8)                      assert_array_equal(arr._data, np.array(Series()))
3611: (8)                      assert_array_equal(arr._mask, [False, True, True])

```

```

3612: (4) @pytest.mark.parametrize("copy", [True, False])
3613: (4) def test_masked_invalid_full_mask(self, copy):
3614: (8)     a = np.ma.array([1, 2, 3, 4])
3615: (8)     assert a._mask is nomask
3616: (8)     res = np.ma.masked_invalid(a, copy=copy)
3617: (8)     assert res.mask is not nomask
3618: (8)     assert a.mask is nomask
3619: (8)     assert np.may_share_memory(a._data, res._data) != copy
3620: (4) def test_choose(self):
3621: (8)     choices = [[0, 1, 2, 3], [10, 11, 12, 13],
3622: (19)             [20, 21, 22, 23], [30, 31, 32, 33]]
3623: (8)     chosen = choose([2, 3, 1, 0], choices)
3624: (8)     assert_equal(chosen, array([20, 31, 12, 3]))
3625: (8)     chosen = choose([2, 4, 1, 0], choices, mode='clip')
3626: (8)     assert_equal(chosen, array([20, 31, 12, 3]))
3627: (8)     chosen = choose([2, 4, 1, 0], choices, mode='wrap')
3628: (8)     assert_equal(chosen, array([20, 1, 12, 3]))
3629: (8)     indices_ = array([2, 4, 1, 0], mask=[1, 0, 0, 1])
3630: (8)     chosen = choose(indices_, choices, mode='wrap')
3631: (8)     assert_equal(chosen, array([99, 1, 12, 99]))
3632: (8)     assert_equal(chosen.mask, [1, 0, 0, 1])
3633: (8)     choices = array(choices, mask=[[0, 0, 0, 1], [1, 1, 0, 1],
3634: (39)             [1, 0, 0, 0], [0, 0, 0, 0]])
3635: (8)     indices_ = [2, 3, 1, 0]
3636: (8)     chosen = choose(indices_, choices, mode='wrap')
3637: (8)     assert_equal(chosen, array([20, 31, 12, 3]))
3638: (8)     assert_equal(chosen.mask, [1, 0, 0, 1])
3639: (4) def test_choose_with_out(self):
3640: (8)     choices = [[0, 1, 2, 3], [10, 11, 12, 13],
3641: (19)             [20, 21, 22, 23], [30, 31, 32, 33]]
3642: (8)     store = empty(4, dtype=int)
3643: (8)     chosen = choose([2, 3, 1, 0], choices, out=store)
3644: (8)     assert_equal(store, array([20, 31, 12, 3]))
3645: (8)     assert_(store is chosen)
3646: (8)     store = empty(4, dtype=int)
3647: (8)     indices_ = array([2, 3, 1, 0], mask=[1, 0, 0, 1])
3648: (8)     chosen = choose(indices_, choices, mode='wrap', out=store)
3649: (8)     assert_equal(store, array([99, 31, 12, 99]))
3650: (8)     assert_equal(store.mask, [1, 0, 0, 1])
3651: (8)     choices = array(choices, mask=[[0, 0, 0, 1], [1, 1, 0, 1],
3652: (39)             [1, 0, 0, 0], [0, 0, 0, 0]])
3653: (8)     indices_ = [2, 3, 1, 0]
3654: (8)     store = empty(4, dtype=int).view(ndarray)
3655: (8)     chosen = choose(indices_, choices, mode='wrap', out=store)
3656: (8)     assert_equal(store, array([999999, 31, 12, 999999]))
3657: (4) def test_reshape(self):
3658: (8)     a = arange(10)
3659: (8)     a[0] = masked
3660: (8)     b = a.reshape((5, 2))
3661: (8)     assert_equal(b.shape, (5, 2))
3662: (8)     assert_(b.flags['C'])
3663: (8)     b = a.reshape(5, 2)
3664: (8)     assert_equal(b.shape, (5, 2))
3665: (8)     assert_(b.flags['C'])
3666: (8)     b = a.reshape((5, 2), order='F')
3667: (8)     assert_equal(b.shape, (5, 2))
3668: (8)     assert_(b.flags['F'])
3669: (8)     b = a.reshape(5, 2, order='F')
3670: (8)     assert_equal(b.shape, (5, 2))
3671: (8)     assert_(b.flags['F'])
3672: (8)     c = np.reshape(a, (2, 5))
3673: (8)     assert_(isinstance(c, MaskedArray))
3674: (8)     assert_equal(c.shape, (2, 5))
3675: (8)     assert_(c[0, 0] is masked)
3676: (8)     assert_(c.flags['C'])
3677: (4) def test_make_mask_descr(self):
3678: (8)     ntype = [('a', float), ('b', float)]
3679: (8)     test = make_mask_descr(ntype)
3680: (8)     assert_equal(test, [('a', bool), ('b', bool)])

```

```

3681: (8)             assert_(test is make_mask_descr(test))
3682: (8)             ntype = (float, 2)
3683: (8)             test = make_mask_descr(ntype)
3684: (8)             assert_equal(test, (bool, 2))
3685: (8)             assert_(test is make_mask_descr(test))
3686: (8)             ntype = float
3687: (8)             test = make_mask_descr(ntype)
3688: (8)             assert_equal(test, np.dtype(bool))
3689: (8)             assert_(test is make_mask_descr(test))
3690: (8)             ntype = [('a', float), ('b', [('ba', float), ('bb', float)))]
3691: (8)             test = make_mask_descr(ntype)
3692: (8)             control = np.dtype([('a', 'b1'), ('b', [('ba', 'b1'), ('bb', 'b1')))])
3693: (8)             assert_equal(test, control)
3694: (8)             assert_(test is make_mask_descr(test))
3695: (8)             ntype = [('a', (float, 2))]
3696: (8)             test = make_mask_descr(ntype)
3697: (8)             assert_equal(test, np.dtype([('a', (bool, 2))]))
3698: (8)             assert_(test is make_mask_descr(test))
3699: (8)             ntype = [((('A', 'a')), float)]
3700: (8)             test = make_mask_descr(ntype)
3701: (8)             assert_equal(test, np.dtype([('A', 'a'), bool]))
3702: (8)             assert_(test is make_mask_descr(test))
3703: (8)             base_type = np.dtype([('a', int, 3)])
3704: (8)             base_mtype = make_mask_descr(base_type)
3705: (8)             sub_type = np.dtype([('a', int), ('b', base_mtype)])
3706: (8)             test = make_mask_descr(sub_type)
3707: (8)             assert_equal(test, np.dtype([('a', bool), ('b', [('a', bool, 3)))]))
3708: (8)             assert_(test.fields['b'][0] is base_mtype)
3709: (4)             def test_make_mask(self):
3710: (8)                 mask = [0, 1]
3711: (8)                 test = make_mask(mask)
3712: (8)                 assert_equal(test.dtype, MaskType)
3713: (8)                 assert_equal(test, [0, 1])
3714: (8)                 mask = np.array([0, 1], dtype=bool)
3715: (8)                 test = make_mask(mask)
3716: (8)                 assert_equal(test.dtype, MaskType)
3717: (8)                 assert_equal(test, [0, 1])
3718: (8)                 mdtype = [('a', bool), ('b', bool)]
3719: (8)                 mask = np.array([(0, 0), (0, 1)], dtype=mdtype)
3720: (8)                 test = make_mask(mask)
3721: (8)                 assert_equal(test.dtype, MaskType)
3722: (8)                 assert_equal(test, [1, 1])
3723: (8)                 mdtype = [('a', bool), ('b', bool)]
3724: (8)                 mask = np.array([(0, 0), (0, 1)], dtype=mdtype)
3725: (8)                 test = make_mask(mask, dtype=mask.dtype)
3726: (8)                 assert_equal(test.dtype, mdtype)
3727: (8)                 assert_equal(test, mask)
3728: (8)                 mdtype = [('a', float), ('b', float)]
3729: (8)                 bdtype = [('a', bool), ('b', bool)]
3730: (8)                 mask = np.array([(0, 0), (0, 1)], dtype=mdtype)
3731: (8)                 test = make_mask(mask, dtype=mask.dtype)
3732: (8)                 assert_equal(test.dtype, bdtype)
3733: (8)                 assert_equal(test, np.array([(0, 0), (0, 1)], dtype=bdtype))
3734: (8)                 mask = np.array((False, True), dtype='?')
3735: (8)                 assert_(isinstance(mask, np.void))
3736: (8)                 test = make_mask(mask, dtype=mask.dtype)
3737: (8)                 assert_equal(test, mask)
3738: (8)                 assert_(test is not mask)
3739: (8)                 mask = np.array((0, 1), dtype='i4,i4')
3740: (8)                 test2 = make_mask(mask, dtype=mask.dtype)
3741: (8)                 assert_equal(test2, test)
3742: (8)                 bools = [True, False]
3743: (8)                 dtypes = [MaskType, float]
3744: (8)                 msgformat = 'copy=%s, shrink=%s, dtype=%s'
3745: (8)                 for cpy, shr, dt in itertools.product(bools, bools, dtypes):
3746: (12)                     res = make_mask(nomask, copy=cpy, shrink=shr, dtype=dt)
3747: (12)                     assert_(res is nomask, msgformat % (cpy, shr, dt))
3748: (4)             def test_mask_or(self):
3749: (8)                 mtype = [('a', bool), ('b', bool)]

```

```

3750: (8)             mask = np.array([(0, 0), (0, 1), (1, 0), (0, 0)], dtype=mtype)
3751: (8)             test = mask_or(mask, nomask)
3752: (8)             assert_equal(test, mask)
3753: (8)             test = mask_or(nomask, mask)
3754: (8)             assert_equal(test, mask)
3755: (8)             test = mask_or(mask, False)
3756: (8)             assert_equal(test, mask)
3757: (8)             other = np.array([(0, 1), (0, 1), (0, 1), (0, 1)], dtype=mtype)
3758: (8)             test = mask_or(mask, other)
3759: (8)             control = np.array([(0, 1), (0, 1), (1, 1), (0, 1)], dtype=mtype)
3760: (8)             assert_equal(test, control)
3761: (8)             othertype = [('A', bool), ('B', bool)]
3762: (8)             other = np.array([(0, 1), (0, 1), (0, 1), (0, 1)], dtype=othertype)
3763: (8)             try:
3764: (12)                 test = mask_or(mask, other)
3765: (8)             except ValueError:
3766: (12)                 pass
3767: (8)             dtype = [('a', bool), ('b', [('ba', bool), ('bb', bool)])]
3768: (8)             amask = np.array([(0, (1, 0)), (0, (1, 0))], dtype=dtype)
3769: (8)             bmask = np.array([(1, (0, 1)), (0, (0, 0))], dtype=dtype)
3770: (8)             cntrl = np.array([(1, (1, 1)), (0, (1, 0))], dtype=dtype)
3771: (8)             assert_equal(mask_or(amask, bmask), cntrl)
3772: (4)             def test_flatten_mask(self):
3773: (8)                 mask = np.array([0, 0, 1], dtype=bool)
3774: (8)                 assert_equal(flatten_mask(mask), mask)
3775: (8)                 mask = np.array([(0, 0), (0, 1)], dtype=[('a', bool), ('b', bool)])
3776: (8)                 test = flatten_mask(mask)
3777: (8)                 control = np.array([0, 0, 0, 1], dtype=bool)
3778: (8)                 assert_equal(test, control)
3779: (8)                 mdtype = [('a', bool), ('b', [('ba', bool), ('bb', bool)])]
3780: (8)                 data = [(0, (0, 0)), (0, (0, 1))]
3781: (8)                 mask = np.array(data, dtype=mdtype)
3782: (8)                 test = flatten_mask(mask)
3783: (8)                 control = np.array([0, 0, 0, 0, 0, 1], dtype=bool)
3784: (8)                 assert_equal(test, control)
3785: (4)             def test_on_ndarray(self):
3786: (8)                 a = np.array([1, 2, 3, 4])
3787: (8)                 m = array(a, mask=False)
3788: (8)                 test = anom(a)
3789: (8)                 assert_equal(test, m.anom())
3790: (8)                 test = reshape(a, (2, 2))
3791: (8)                 assert_equal(test, m.reshape(2, 2))
3792: (4)             def test_compress(self):
3793: (8)                 arr = np.arange(8)
3794: (8)                 arr.shape = 4, 2
3795: (8)                 cond = np.array([True, False, True, True])
3796: (8)                 control = arr[[0, 2, 3]]
3797: (8)                 test = np.ma.compress(cond, arr, axis=0)
3798: (8)                 assert_equal(test, control)
3799: (8)                 marr = np.ma.array(arr)
3800: (8)                 test = np.ma.compress(cond, marr, axis=0)
3801: (8)                 assert_equal(test, control)
3802: (4)             def test_compressed(self):
3803: (8)                 a = np.ma.array([1, 2])
3804: (8)                 test = np.ma.compressed(a)
3805: (8)                 assert_(type(test) is np.ndarray)
3806: (8)                 class A(np.ndarray):
3807: (12)                     pass
3808: (8)                     a = np.ma.array(A(shape=0))
3809: (8)                     test = np.ma.compressed(a)
3810: (8)                     assert_(type(test) is A)
3811: (8)                     test = np.ma.compressed([[1],[2]])
3812: (8)                     assert_equal(test.ndim, 1)
3813: (8)                     test = np.ma.compressed([[[[1]]]])
3814: (8)                     assert_equal(test.ndim, 1)
3815: (8)                     class M(MaskedArray):
3816: (12)                         pass
3817: (8)                         test = np.ma.compressed(M([[[]], [[]]]))
3818: (8)                         assert_equal(test.ndim, 1)

```

```

3819: (8)
3820: (12)
3821: (16)
3822: (8)
3823: (8)
3824: (4)
3825: (8)
3826: (8)
3827: (8)
3828: (8)
3829: (8)
3830: (8)
3831: (8)
3832: (8)
3833: (8)
3834: (8)
3835: (8)
3836: (8)
3837: (8)
3838: (8)
3839: (0)
3840: (4)
3841: (8)
3842: (8)
3843: (8)
3844: (8)
3845: (8)
3846: (8)
3847: (8)
3848: (8)
3849: (4)
3850: (8)
3851: (8)
3852: (8)
3853: (8)
3854: (8)
3855: (8)
3856: (8)
3857: (8)
3858: (8)
3859: (8)
3860: (8)
3861: (8)
3862: (29)
3863: (38)
3864: (4)
3865: (8)
3866: (8)
3867: (8)
3868: (8)
3869: (8)
3870: (8)
3871: (8)
3872: (8)
3873: (8)
3874: (21)
3875: (4)
3876: (8)
3877: (8)
3878: (8)
3879: (8)
3880: (8)
3881: (8)
3882: (8)
3883: (8)
3884: (8)
3885: (21)
3886: (4)
3887: (8)

        class M(MaskedArray):
            def compressed(self):
                return 42
            test = np.ma.compressed(M([[], []]))
            assert_equal(test, 42)
        def test_convolve(self):
            a = masked_equal(np.arange(5), 2)
            b = np.array([1, 1])
            test = np.ma.convolve(a, b)
            assert_equal(test, masked_equal([0, 1, -1, -1, 7, 4], -1))
            test = np.ma.convolve(a, b, propagate_mask=False)
            assert_equal(test, masked_equal([0, 1, 1, 3, 7, 4], -1))
            test = np.ma.convolve([1, 1], [1, 1, 1])
            assert_equal(test, masked_equal([1, 2, 2, 1], -1))
            a = [1, 1]
            b = masked_equal([1, -1, -1, 1], -1)
            test = np.ma.convolve(a, b, propagate_mask=False)
            assert_equal(test, masked_equal([1, 1, -1, 1, 1], -1))
            test = np.ma.convolve(a, b, propagate_mask=True)
            assert_equal(test, masked_equal([-1, -1, -1, -1, -1], -1))

    class TestMaskedFields:
        def setup_method(self):
            ilist = [1, 2, 3, 4, 5]
            flist = [1.1, 2.2, 3.3, 4.4, 5.5]
            slist = ['one', 'two', 'three', 'four', 'five']
            ddtype = [('a', int), ('b', float), ('c', '|S8')]
            mdtype = [('a', bool), ('b', bool), ('c', bool)]
            mask = [0, 1, 0, 0, 1]
            base = array(list(zip(ilist, flist, slist)), mask=mask, dtype=ddtype)
            self.data = dict(base=base, mask=mask, ddtype=ddtype, mdtype=mdtype)
        def test_set_records_masks(self):
            base = self.data['base']
            mdtype = self.data['mdtype']
            base.mask = nomask
            assert_equal_records(base._mask, np.zeros(base.shape, dtype=mdtype))
            base.mask = masked
            assert_equal_records(base._mask, np.ones(base.shape, dtype=mdtype))
            base.mask = False
            assert_equal_records(base._mask, np.zeros(base.shape, dtype=mdtype))
            base.mask = True
            assert_equal_records(base._mask, np.ones(base.shape, dtype=mdtype))
            base.mask = [0, 0, 0, 1, 1]
            assert_equal_records(base._mask,
                                np.array([(x, x, x) for x in [0, 0, 0, 1, 1]]),
                                dtype=mdtype)
        def test_set_record_element(self):
            base = self.data['base']
            (base_a, base_b, base_c) = (base['a'], base['b'], base['c'])
            base[0] = (pi, pi, 'pi')
            assert_equal(base_a.dtype, int)
            assert_equal(base_a._data, [3, 2, 3, 4, 5])
            assert_equal(base_b.dtype, float)
            assert_equal(base_b._data, [pi, 2.2, 3.3, 4.4, 5.5])
            assert_equal(base_c.dtype, '|S8')
            assert_equal(base_c._data,
                        [b'pi', b'two', b'three', b'four', b'five'])
        def test_set_record_slice(self):
            base = self.data['base']
            (base_a, base_b, base_c) = (base['a'], base['b'], base['c'])
            base[:3] = (pi, pi, 'pi')
            assert_equal(base_a.dtype, int)
            assert_equal(base_a._data, [3, 3, 3, 4, 5])
            assert_equal(base_b.dtype, float)
            assert_equal(base_b._data, [pi, pi, pi, 4.4, 5.5])
            assert_equal(base_c.dtype, '|S8')
            assert_equal(base_c._data,
                        [b'pi', b'pi', b'pi', b'four', b'five'])
        def test_mask_element(self):
            "Check record access"

```

```

3888: (8)             base = self.data['base']
3889: (8)             base[0] = masked
3890: (8)             for n in ('a', 'b', 'c'):
3891: (12)                assert_equal(base[n].mask, [1, 1, 0, 0, 1])
3892: (12)                assert_equal(base[n]._data, base._data[n])
3893: (4)             def test_getmaskarray(self):
3894: (8)                 ndtype = [('a', int), ('b', float)]
3895: (8)                 test = empty(3, dtype=ndtype)
3896: (8)                 assert_equal(getmaskarray(test),
3897: (21)                     np.array([(0, 0), (0, 0), (0, 0)],
3898: (30)                         dtype=[('a', '|b1'), ('b', '|b1')]))
3899: (8)                 test[:] = masked
3900: (8)                 assert_equal(getmaskarray(test),
3901: (21)                     np.array([(1, 1), (1, 1), (1, 1)],
3902: (30)                         dtype=[('a', '|b1'), ('b', '|b1')]))
3903: (4)             def test_view(self):
3904: (8)                 iterator = list(zip(np.arange(10), np.random.rand(10)))
3905: (8)                 data = np.array(iterator)
3906: (8)                 a = array(iterator, dtype=[('a', float), ('b', float)])
3907: (8)                 a.mask[0] = (1, 0)
3908: (8)                 controlmask = np.array([1] + 19 * [0], dtype=bool)
3909: (8)                 test = a.view(float)
3910: (8)                 assert_equal(test, data.ravel())
3911: (8)                 assert_equal(test.mask, controlmask)
3912: (8)                 test = a.view((float, 2))
3913: (8)                 assert_equal(test, data)
3914: (8)                 assert_equal(test.mask, controlmask.reshape(-1, 2))
3915: (4)             def test_getitem(self):
3916: (8)                 ndtype = [('a', float), ('b', float)]
3917: (8)                 a = array(list(zip(np.random.rand(10), np.arange(10))), dtype=ndtype)
3918: (8)                 a.mask = np.array(list(zip([0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
3919: (35)                               [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0])),
3920: (26)                               dtype=[('a', bool), ('b', bool)])
3921: (8)             def _test_index(i):
3922: (12)                 assert_equal(type(a[i]), mvoid)
3923: (12)                 assert_equal_records(a[i]._data, a._data[i])
3924: (12)                 assert_equal_records(a[i]._mask, a._mask[i])
3925: (12)                 assert_equal(type(a[i, ...]), MaskedArray)
3926: (12)                 assert_equal_records(a[i,...]._data, a._data[i,...])
3927: (12)                 assert_equal_records(a[i,...]._mask, a._mask[i,...])
3928: (8)                 _test_index(1) # No mask
3929: (8)                 _test_index(0) # One element masked
3930: (8)                 _test_index(-2) # All element masked
3931: (4)             def test_setitem(self):
3932: (8)                 ndtype = np.dtype([('a', float), ('b', int)])
3933: (8)                 ma = np.ma.MaskedArray([(1.0, 1), (2.0, 2)], dtype=ndtype)
3934: (8)                 ma['a'][1] = 3.0
3935: (8)                 assert_equal(ma['a'], np.array([1.0, 3.0]))
3936: (8)                 ma[1]['a'] = 4.0
3937: (8)                 assert_equal(ma['a'], np.array([1.0, 4.0]))
3938: (8)                 mdtype = np.dtype([('a', bool), ('b', bool)])
3939: (8)                 control = np.array([(False, True), (True, True)], dtype=mdtype)
3940: (8)                 a = np.ma.masked_all((2,), dtype=ndtype)
3941: (8)                 a['a'][0] = 2
3942: (8)                 assert_equal(a.mask, control)
3943: (8)                 a = np.ma.masked_all((2,), dtype=ndtype)
3944: (8)                 a[0]['a'] = 2
3945: (8)                 assert_equal(a.mask, control)
3946: (8)                 control = np.array([(True, True), (True, True)], dtype=mdtype)
3947: (8)                 a = np.ma.masked_all((2,), dtype=ndtype)
3948: (8)                 a.harden_mask()
3949: (8)                 a['a'][0] = 2
3950: (8)                 assert_equal(a.mask, control)
3951: (8)                 a = np.ma.masked_all((2,), dtype=ndtype)
3952: (8)                 a.harden_mask()
3953: (8)                 a[0]['a'] = 2
3954: (8)                 assert_equal(a.mask, control)
3955: (4)             def test_setitem_scalar(self):
3956: (8)                 mask_0d = np.ma.masked_array(1, mask=True)

```

```

3957: (8)             arr = np.ma.arange(3)
3958: (8)             arr[0] = mask_0d
3959: (8)             assert_array_equal(arr.mask, [True, False, False])
3960: (4)             def test_element_len(self):
3961: (8)                 for rec in self.data['base']:
3962: (12)                     assert_equal(len(rec), len(self.data['ddtype']))
3963: (0)             class TestMaskedObjectArray:
3964: (4)                 def test_getitem(self):
3965: (8)                     arr = np.ma.array([None, None])
3966: (8)                     for dt in [float, object]:
3967: (12)                         a0 = np.eye(2).astype(dt)
3968: (12)                         a1 = np.eye(3).astype(dt)
3969: (12)                         arr[0] = a0
3970: (12)                         arr[1] = a1
3971: (12)                         assert_(arr[0] is a0)
3972: (12)                         assert_(arr[1] is a1)
3973: (12)                         assert_(isinstance(arr[0,...], MaskedArray))
3974: (12)                         assert_(isinstance(arr[1,...], MaskedArray))
3975: (12)                         assert_(arr[0,...][()] is a0)
3976: (12)                         assert_(arr[1,...][()] is a1)
3977: (12)                         arr[0] = np.ma.masked
3978: (12)                         assert_(arr[1] is a1)
3979: (12)                         assert_(isinstance(arr[0,...], MaskedArray))
3980: (12)                         assert_(isinstance(arr[1,...], MaskedArray))
3981: (12)                         assert_equal(arr[0,...].mask, True)
3982: (12)                         assert_(arr[1,...][()] is a1)
3983: (12)                         assert_equal(arr[0].data, a0)
3984: (12)                         assert_equal(arr[0].mask, True)
3985: (12)                         assert_equal(arr[0,...][()].data, a0)
3986: (12)                         assert_equal(arr[0,...][()].mask, True)
3987: (4)             def test_nested_ma(self):
3988: (8)                 arr = np.ma.array([None, None])
3989: (8)                 arr[0,...] = np.array([np.ma.masked], object)[0,...]
3990: (8)                 assert_(arr.data[0] is np.ma.masked)
3991: (8)                 assert_(arr[0] is np.ma.masked)
3992: (8)                 arr[0] = np.ma.masked
3993: (8)                 assert_(arr[0] is np.ma.masked)
3994: (0)             class TestMaskedView:
3995: (4)                 def setup_method(self):
3996: (8)                     iterator = list(zip(np.arange(10), np.random.rand(10)))
3997: (8)                     data = np.array(iterator)
3998: (8)                     a = array(iterator, dtype=[('a', float), ('b', float)])
3999: (8)                     a.mask[0] = (1, 0)
4000: (8)                     controlmask = np.array([1] + 19 * [0], dtype=bool)
4001: (8)                     self.data = (data, a, controlmask)
4002: (4)             def test_view_to_nothing(self):
4003: (8)                 (data, a, controlmask) = self.data
4004: (8)                 test = a.view()
4005: (8)                 assert_(isinstance(test, MaskedArray))
4006: (8)                 assert_equal(test._data, a._data)
4007: (8)                 assert_equal(test._mask, a._mask)
4008: (4)             def test_view_to_type(self):
4009: (8)                 (data, a, controlmask) = self.data
4010: (8)                 test = a.view(np.ndarray)
4011: (8)                 assert_(not isinstance(test, MaskedArray))
4012: (8)                 assert_equal(test, a._data)
4013: (8)                 assert_equal_records(test, data.view(a.dtype).squeeze())
4014: (4)             def test_view_to_simple_dtype(self):
4015: (8)                 (data, a, controlmask) = self.data
4016: (8)                 test = a.view(float)
4017: (8)                 assert_(isinstance(test, MaskedArray))
4018: (8)                 assert_equal(test, data.ravel())
4019: (8)                 assert_equal(test.mask, controlmask)
4020: (4)             def test_view_to_flexible_dtype(self):
4021: (8)                 (data, a, controlmask) = self.data
4022: (8)                 test = a.view([('A', float), ('B', float)])
4023: (8)                 assert_equal(test.mask.dtype.names, ('A', 'B'))
4024: (8)                 assert_equal(test['A'], a['a'])
4025: (8)                 assert_equal(test['B'], a['b'])

```

```

4026: (8)             test = a[0].view([('A', float), ('B', float)])
4027: (8)             assert_(isinstance(test, MaskedArray))
4028: (8)             assert_equal(test.mask.dtype.names, ('A', 'B'))
4029: (8)             assert_equal(test['A'], a['a'][0])
4030: (8)             assert_equal(test['B'], a['b'][0])
4031: (8)             test = a[-1].view([('A', float), ('B', float)])
4032: (8)             assert_(isinstance(test, MaskedArray))
4033: (8)             assert_equal(test.dtype.names, ('A', 'B'))
4034: (8)             assert_equal(test['A'], a['a'][-1])
4035: (8)             assert_equal(test['B'], a['b'][-1])
4036: (4)             def test_view_to_subtype(self):
4037: (8)                 (data, a, controlmask) = self.data
4038: (8)                 test = a.view((float, 2))
4039: (8)                 assert_(isinstance(test, MaskedArray))
4040: (8)                 assert_equal(test, data)
4041: (8)                 assert_equal(test.mask, controlmask.reshape(-1, 2))
4042: (8)                 test = a[0].view((float, 2))
4043: (8)                 assert_(isinstance(test, MaskedArray))
4044: (8)                 assert_equal(test, data[0])
4045: (8)                 assert_equal(test.mask, (1, 0))
4046: (8)                 test = a[-1].view((float, 2))
4047: (8)                 assert_(isinstance(test, MaskedArray))
4048: (8)                 assert_equal(test, data[-1])
4049: (4)             def test_view_to_dtype_and_type(self):
4050: (8)                 (data, a, controlmask) = self.data
4051: (8)                 test = a.view((float, 2), np.recarray)
4052: (8)                 assert_equal(test, data)
4053: (8)                 assert_(isinstance(test, np.recarray))
4054: (8)                 assert_(not isinstance(test, MaskedArray))
4055: (0)             class TestOptionalArgs:
4056: (4)                 def test_ndarrayfuncs(self):
4057: (8)                     d = np.arange(24.0).reshape((2,3,4))
4058: (8)                     m = np.zeros(24, dtype=bool).reshape((2,3,4))
4059: (8)                     m[:, :, -1] = True
4060: (8)                     a = np.ma.array(d, mask=m)
4061: (8)                     def testaxis(f, a, d):
4062: (12)                         numpy_f = numpy.__getattribute__(f)
4063: (12)                         ma_f = np.ma.__getattribute__(f)
4064: (12)                         assert_equal(ma_f(a, axis=1)[..., :-1], numpy_f(d[..., :-1]),
4065: (12)                                         axis=1))
4066: (25)                         assert_equal(ma_f(a, axis=(0,1))[..., :-1],
4067: (8)                                         numpy_f(d[...,-1], axis=(0,1)))
4068: (12)                     def testkeepdims(f, a, d):
4069: (12)                         numpy_f = numpy.__getattribute__(f)
4070: (12)                         ma_f = np.ma.__getattribute__(f)
4071: (25)                         assert_equal(ma_f(a, keepdims=True).shape,
4072: (12)                                         numpy_f(d, keepdims=True).shape)
4073: (25)                         assert_equal(ma_f(a, keepdims=False).shape,
4074: (12)                                         numpy_f(d, keepdims=False).shape)
4075: (25)                         assert_equal(ma_f(a, axis=1, keepdims=True)[..., :-1],
4076: (12)                                         numpy_f(d[...,-1], axis=1, keepdims=True))
4077: (25)                         assert_equal(ma_f(a, axis=(0,1), keepdims=True)[..., :-1],
4078: (8)                                         numpy_f(d[...,-1], axis=(0,1), keepdims=True))
4079: (12)                     for f in ['sum', 'prod', 'mean', 'var', 'std']:
4080: (12)                         testaxis(f, a, d)
4081: (8)                         testkeepdims(f, a, d)
4082: (12)                     for f in ['min', 'max']:
4083: (8)                         testaxis(f, a, d)
4084: (8)                         d = (np.arange(24).reshape((2,3,4))%2 == 0)
4085: (8)                         a = np.ma.array(d, mask=m)
4086: (12)                         for f in ['all', 'any']:
4087: (12)                             testaxis(f, a, d)
4088: (4)                             testkeepdims(f, a, d)
4089: (8)                         def test_count(self):
4090: (8)                             d = np.arange(24.0).reshape((2,3,4))
4091: (8)                             m = np.zeros(24, dtype=bool).reshape((2,3,4))
4092: (8)                             m[:, 0, :] = True
4093: (8)                             a = np.ma.array(d, mask=m)
4094: (8)                             assert_equal(count(a), 16)

```

```

4094: (8) assert_equal(count(a, axis=1), 2*ones((2,4)))
4095: (8) assert_equal(count(a, axis=(0,1)), 4*ones((4,)))
4096: (8) assert_equal(count(a, keepdims=True), 16*ones((1,1,1)))
4097: (8) assert_equal(count(a, axis=1, keepdims=True), 2*ones((2,1,4)))
4098: (8) assert_equal(count(a, axis=(0,1), keepdims=True), 4*ones((1,1,4)))
4099: (8) assert_equal(count(a, axis=-2), 2*ones((2,4)))
4100: (8) assert_raises(ValueError, count, a, axis=(1,1))
4101: (8) assert_raises(np.AxisError, count, a, axis=3)
4102: (8) a = np.ma.array(d, mask=nomask)
4103: (8) assert_equal(count(a), 24)
4104: (8) assert_equal(count(a, axis=1), 3*ones((2,4)))
4105: (8) assert_equal(count(a, axis=(0,1)), 6*ones((4,)))
4106: (8) assert_equal(count(a, keepdims=True), 24*ones((1,1,1)))
4107: (8) assert_equal(np.ndim(count(a, keepdims=True)), 3)
4108: (8) assert_equal(count(a, axis=1, keepdims=True), 3*ones((2,1,4)))
4109: (8) assert_equal(count(a, axis=(0,1), keepdims=True), 6*ones((1,1,4)))
4110: (8) assert_equal(count(a, axis=-2), 3*ones((2,4)))
4111: (8) assert_raises(ValueError, count, a, axis=(1,1))
4112: (8) assert_raises(np.AxisError, count, a, axis=3)
4113: (8) assert_equal(count(np.ma.masked), 0)
4114: (8) assert_raises(np.AxisError, count, np.ma.array(1), axis=1)
4115: (0) class TestMaskedConstant:
4116: (4)     def _do_add_test(self, add):
4117: (8)         assert_(add(np.ma.masked, 1) is np.ma.masked)
4118: (8)         vector = np.array([1, 2, 3])
4119: (8)         result = add(np.ma.masked, vector)
4120: (8)         assert_(result is not np.ma.masked)
4121: (8)         assert_(not isinstance(result, np.ma.core.MaskedConstant))
4122: (8)         assert_equal(result.shape, vector.shape)
4123: (8)         assert_equal(np.ma.getmask(result), np.ones(vector.shape, dtype=bool))
4124: (4)     def test_ufunc(self):
4125: (8)         self._do_add_test(np.add)
4126: (4)     def test_operator(self):
4127: (8)         self._do_add_test(lambda a, b: a + b)
4128: (4)     def test_ctor(self):
4129: (8)         m = np.ma.array(np.ma.masked)
4130: (8)         assert_(not isinstance(m, np.ma.core.MaskedConstant))
4131: (8)         assert_(m is not np.ma.masked)
4132: (4)     def test_repr(self):
4133: (8)         assert_equal(repr(np.ma.masked), 'masked')
4134: (8)         masked2 = np.ma.MaskedArray.__new__(np.ma.core.MaskedConstant)
4135: (8)         assert_not_equal(repr(masked2), 'masked')
4136: (4)     def test_pickle(self):
4137: (8)         from io import BytesIO
4138: (8)         for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
4139: (12)             with BytesIO() as f:
4140: (16)                 pickle.dump(np.ma.masked, f, protocol=proto)
4141: (16)                 f.seek(0)
4142: (16)                 res = pickle.load(f)
4143: (12)                 assert_(res is np.ma.masked)
4144: (4)     def test_copy(self):
4145: (8)         assert_equal(
4146: (12)             np.ma.masked.copy() is np.ma.masked,
4147: (12)             np.True_.copy() is np.True_)
4148: (4)     def test_copy(self):
4149: (8)         import copy
4150: (8)         assert_(
4151: (12)             copy.copy(np.ma.masked) is np.ma.masked)
4152: (4)     def test_deepcopy(self):
4153: (8)         import copy
4154: (8)         assert_(
4155: (12)             copy.deepcopy(np.ma.masked) is np.ma.masked)
4156: (4)     def test_immutable(self):
4157: (8)         orig = np.ma.masked
4158: (8)         assert_raises(np.ma.core.MaskError, operator.setitem, orig, (), 1)
4159: (8)         assert_raises(ValueError, operator.setitem, orig.data, (), 1)
4160: (8)         assert_raises(ValueError, operator.setitem, orig.mask, (), False)
4161: (8)         view = np.ma.masked.view(np.ma.MaskedArray)
4162: (8)         assert_raises(ValueError, operator.setitem, view, (), 1)

```

```

4163: (8)             assert_raises(ValueError, operator.setitem, view.data, (), 1)
4164: (8)             assert_raises(ValueError, operator.setitem, view.mask, (), False)
4165: (4)             def test_coercion_int(self):
4166: (8)                 a_i = np.zeros((), int)
4167: (8)                 assert_raises(MaskError, operator.setitem, a_i, (), np.ma.masked)
4168: (8)                 assert_raises(MaskError, int, np.ma.masked)
4169: (4)             def test_coercion_float(self):
4170: (8)                 a_f = np.zeros((), float)
4171: (8)                 assert_warns(UserWarning, operator.setitem, a_f, (), np.ma.masked)
4172: (8)                 assert_(np.isnan(a_f[()]))
4173: (4)             @pytest.mark.xfail(reason="See gh-9750")
4174: (4)             def test_coercion_unicode(self):
4175: (8)                 a_u = np.zeros((), 'U10')
4176: (8)                 a_u[()] = np.ma.masked
4177: (8)                 assert_equal(a_u[()], '--')
4178: (4)             @pytest.mark.xfail(reason="See gh-9750")
4179: (4)             def test_coercion_bytes(self):
4180: (8)                 a_b = np.zeros((), 'S10')
4181: (8)                 a_b[()] = np.ma.masked
4182: (8)                 assert_equal(a_b[()], b'--')
4183: (4)             def test_subclass(self):
4184: (8)                 class Sub(type(np.ma.masked)): pass
4185: (8)                 a = Sub()
4186: (8)                 assert_(a is Sub())
4187: (8)                 assert_(a is not np.ma.masked)
4188: (8)                 assert_not_equal(repr(a), 'masked')
4189: (4)             def test_attributes_READONLY(self):
4190: (8)                 assert_raises(AttributeError, setattr, np.ma.masked, 'shape', (1,))
4191: (8)                 assert_raises(AttributeError, setattr, np.ma.masked, 'dtype',
np.int64)
4192: (0)             class TestMaskedWhereAliases:
4193: (4)                 def test_masked_values(self):
4194: (8)                     res = masked_values(np.array([-32768.0]), np.int16(-32768))
4195: (8)                     assert_equal(res.mask, [True])
4196: (8)                     res = masked_values(np.inf, np.inf)
4197: (8)                     assert_equal(res.mask, True)
4198: (8)                     res = np.ma.masked_values(np.inf, -np.inf)
4199: (8)                     assert_equal(res.mask, False)
4200: (8)                     res = np.ma.masked_values([1, 2, 3, 4], 5, shrink=True)
4201: (8)                     assert_(res.mask is np.ma.nomask)
4202: (8)                     res = np.ma.masked_values([1, 2, 3, 4], 5, shrink=False)
4203: (8)                     assert_equal(res.mask, [False] * 4)
4204: (0)             def test_masked_array():
4205: (4)                 a = np.ma.array([0, 1, 2, 3], mask=[0, 0, 1, 0])
4206: (4)                 assert_equal(np.argwhere(a), [[1], [3]])
4207: (0)             def test_masked_array_no_copy():
4208: (4)                 a = np.ma.array([1, 2, 3, 4])
4209: (4)                 _ = np.ma.masked_where(a == 3, a, copy=False)
4210: (4)                 assert_array_equal(a.mask, [False, False, True, False])
4211: (4)                 a = np.ma.array([1, 2, 3, 4], mask=[1, 0, 0, 0])
4212: (4)                 _ = np.ma.masked_where(a == 3, a, copy=False)
4213: (4)                 assert_array_equal(a.mask, [True, False, True, False])
4214: (4)                 a = np.ma.array([np.inf, 1, 2, 3, 4])
4215: (4)                 _ = np.ma.masked_invalid(a, copy=False)
4216: (4)                 assert_array_equal(a.mask, [True, False, False, False, False])
4217: (0)             def test_append_masked_array():
4218: (4)                 a = np.ma.masked_equal([1, 2, 3], value=2)
4219: (4)                 b = np.ma.masked_equal([4, 3, 2], value=2)
4220: (4)                 result = np.ma.append(a, b)
4221: (4)                 expected_data = [1, 2, 3, 4, 3, 2]
4222: (4)                 expected_mask = [False, True, False, False, False, True]
4223: (4)                 assert_array_equal(result.data, expected_data)
4224: (4)                 assert_array_equal(result.mask, expected_mask)
4225: (4)                 a = np.ma.masked_all((2, 2))
4226: (4)                 b = np.ma.ones((3, 1))
4227: (4)                 result = np.ma.append(a, b)
4228: (4)                 expected_data = [1] * 3
4229: (4)                 expected_mask = [True] * 4 + [False] * 3
4230: (4)                 assert_array_equal(result.data[-3], expected_data)

```

```

4231: (4)             assert_array_equal(result.mask, expected_mask)
4232: (4)             result = np.ma.append(a, b, axis=None)
4233: (4)             assert_array_equal(result.data[-3], expected_data)
4234: (4)             assert_array_equal(result.mask, expected_mask)
4235: (0)             def test_append_masked_array_along_axis():
4236: (4)                 a = np.ma.masked_equal([1,2,3], value=2)
4237: (4)                 b = np.ma.masked_values([[4, 5, 6], [7, 8, 9]], 7)
4238: (4)                 assert_raises(ValueError, np.ma.append, a, b, axis=0)
4239: (4)                 result = np.ma.append(a[np.newaxis,:], b, axis=0)
4240: (4)                 expected = np.ma.arange(1, 10)
4241: (4)                 expected[[1, 6]] = np.ma.masked
4242: (4)                 expected = expected.reshape((3,3))
4243: (4)                 assert_array_equal(result.data, expected.data)
4244: (4)                 assert_array_equal(result.mask, expected.mask)
4245: (0)             def test_default_fill_value_complex():
4246: (4)                 assert_(default_fill_value(1 + 1j) == 1.e20 + 0.0j)
4247: (0)             def test_ufunc_with_output():
4248: (4)                 x = array([1., 2., 3.], mask=[0, 0, 1])
4249: (4)                 y = np.add(x, 1., out=x)
4250: (4)                 assert_(y is x)
4251: (0)             def test_ufunc_with_out_varied():
4252: (4)                 """ Test that masked arrays are immune to gh-10459 """
4253: (4)                 a          = array([ 1, 2, 3], mask=[1, 0, 0])
4254: (4)                 b          = array([10, 20, 30], mask=[1, 0, 0])
4255: (4)                 out        = array([ 0, 0, 0], mask=[0, 0, 1])
4256: (4)                 expected   = array([11, 22, 33], mask=[1, 0, 0])
4257: (4)                 out_pos    = out.copy()
4258: (4)                 res_pos    = np.add(a, b, out_pos)
4259: (4)                 out_kw     = out.copy()
4260: (4)                 res_kw     = np.add(a, b, out=out_kw)
4261: (4)                 out_tup    = out.copy()
4262: (4)                 res_tup    = np.add(a, b, out=(out_tup,))
4263: (4)                 assert_equal(res_kw.mask, expected.mask)
4264: (4)                 assert_equal(res_kw.data, expected.data)
4265: (4)                 assert_equal(res_tup.mask, expected.mask)
4266: (4)                 assert_equal(res_tup.data, expected.data)
4267: (4)                 assert_equal(res_pos.mask, expected.mask)
4268: (4)                 assert_equal(res_pos.data, expected.data)
4269: (0)             def test_astype_mask_ordering():
4270: (4)                 descr = np.dtype([('v', int, 3), ('x', [('y', float)])])
4271: (4)                 x = array([
4272: (8)                     ([[1, 2, 3], (1.0,)),  ([[1, 2, 3], (2.0,))],
4273: (8)                     ([[1, 2, 3], (3.0,)),  ([[1, 2, 3], (4.0,))]], dtype=descr)
4274: (4)                 x[0]['v'][0] = np.ma.masked
4275: (4)                 x_a = x.astype(descr)
4276: (4)                 assert x_a.dtype.names == np.dtype(descr).names
4277: (4)                 assert x_a.mask.dtype.names == np.dtype(descr).names
4278: (4)                 assert_equal(x, x_a)
4279: (4)                 assert_(x is x.astype(x.dtype, copy=False))
4280: (4)                 assert_equal(type(x.astype(x.dtype, subok=False)), np.ndarray)
4281: (4)                 x_f = x.astype(x.dtype, order='F')
4282: (4)                 assert_(x_f.flags.f_contiguous)
4283: (4)                 assert_(x_f.mask.flags.f_contiguous)
4284: (4)                 x_a2 = np.array(x, dtype=descr, subok=True)
4285: (4)                 assert x_a2.dtype.names == np.dtype(descr).names
4286: (4)                 assert x_a2.mask.dtype.names == np.dtype(descr).names
4287: (4)                 assert_equal(x, x_a2)
4288: (4)                 assert_(x is np.array(x, dtype=descr, copy=False, subok=True))
4289: (4)                 x_f2 = np.array(x, dtype=x.dtype, order='F', subok=True)
4290: (4)                 assert_(x_f2.flags.f_contiguous)
4291: (4)                 assert_(x_f2.mask.flags.f_contiguous)
4292: (0)             @pytest.mark.parametrize('dt1', num_dts, ids=num_ids)
4293: (0)             @pytest.mark.parametrize('dt2', num_dts, ids=num_ids)
4294: (0)             @pytest.mark.filterwarnings('ignore::numpy.ComplexWarning')
4295: (0)             def test_astype_basic(dt1, dt2):
4296: (4)                 src = np.ma.array(ones(3, dt1), fill_value=1)
4297: (4)                 dst = src.astype(dt2)
4298: (4)                 assert_(src.fill_value == 1)
4299: (4)                 assert_(src.dtype == dt1)

```

```

4300: (4)             assert_(src.fill_value.dtype == dt1)
4301: (4)             assert_(dst.fill_value == 1)
4302: (4)             assert_(dst.dtype == dt2)
4303: (4)             assert_(dst.fill_value.dtype == dt2)
4304: (4)             assert_equal(src, dst)
4305: (0)              def test_fieldless_void():
4306: (4)                  dt = np.dtype([]) # a void dtype with no fields
4307: (4)                  x = np.empty(4, dt)
4308: (4)                  mx = np.ma.array(x)
4309: (4)                  assert_equal(mx.dtype, x.dtype)
4310: (4)                  assert_equal(mx.shape, x.shape)
4311: (4)                  mx = np.ma.array(x, mask=x)
4312: (4)                  assert_equal(mx.dtype, x.dtype)
4313: (4)                  assert_equal(mx.shape, x.shape)
4314: (0)              def test_mask_shape_assignment_does_not_break_masked():
4315: (4)                  a = np.ma.masked
4316: (4)                  b = np.ma.array(1, mask=a.mask)
4317: (4)                  b.shape = (1,)
4318: (4)                  assert_equal(a.mask.shape, ())
4319: (0)          @pytest.mark.skipif(sys.flags.optimize > 1,
4320: (20)                      reason="no docstrings present to inspect when
PYTHONOPTIMIZE/Py_OptimizeFlag > 1")
4321: (0)          def test_doc_note():
4322: (4)              def method(self):
4323: (8)                  """This docstring
4324: (8)                  Has multiple lines
4325: (8)                  And notes
4326: (8)                  Notes
4327: (8)                  -----
4328: (8)                  original note
4329: (8)                  """
4330: (8)                  pass
4331: (4)              expected_doc = """This docstring
4332: (0)          Has multiple lines
4333: (0)          And notes
4334: (0)          Notes
4335: (0)          -----
4336: (0)          note
4337: (0)          original note"""
4338: (4)          assert_equal(np.ma.core.doc_note(method.__doc__), expected_doc)
4339: (0)          def test_gh_22556():
4340: (4)              source = np.ma.array([0, [0, 1, 2]], dtype=object)
4341: (4)              deepcopy = copy.deepcopy(source)
4342: (4)              deepcopy[1].append('this should not appear in source')
4343: (4)              assert len(source[1]) == 3
4344: (0)          def test_gh_21022():
4345: (4)              source = np.ma.masked_array(data=[-1, -1], mask=True, dtype=np.float64)
4346: (4)              axis = np.array(0)
4347: (4)              result = np.prod(source, axis=axis, keepdims=False)
4348: (4)              result = np.ma.masked_array(result,
4349: (32)                                mask=np.ones(result.shape, dtype=np.bool_))
4350: (4)              array = np.ma.masked_array(data=-1, mask=True, dtype=np.float64)
4351: (4)              copy.deepcopy(array)
4352: (4)              copy.deepcopy(result)
4353: (0)          def test_deepcopy_2d_obj():
4354: (4)              source = np.ma.array([[0, "dog"],
4355: (26)                            [1, 1],
4356: (26)                            [[1, 2], "cat"]],
4357: (24)                            mask=[[0, 1],
4358: (30)                              [0, 0],
4359: (30)                              [0, 0]],
4360: (24)                            dtype=object)
4361: (4)              deepcopy = copy.deepcopy(source)
4362: (4)              deepcopy[2, 0].extend(['this should not appear in source', 3])
4363: (4)              assert len(source[2, 0]) == 2
4364: (4)              assert len(deepcopy[2, 0]) == 4
4365: (4)              assert_equal(deepcopy._mask, source._mask)
4366: (4)              deepcopy._mask[0, 0] = 1
4367: (4)              assert source._mask[0, 0] == 0

```

```

4368: (0)     def test_deepcopy_0d_obj():
4369: (4)         source = np.ma.array(0, mask=[0], dtype=object)
4370: (4)         deepcopy = copy.deepcopy(source)
4371: (4)         deepcopy[...] = 17
4372: (4)         assert_equal(source, 0)
4373: (4)         assert_equal(deepcopy, 17)

```

---

File 269 - test\_extras.py:

```

1: (0)             """Tests suite for MaskedArray.
2: (0)             Adapted from the original test_ma by Pierre Gerard-Marchant
3: (0)             :author: Pierre Gerard-Marchant
4: (0)             :contact: pierregm_at_uga_dot_edu
5: (0)             :version: $Id: test_extras.py 3473 2007-10-29 15:18:13Z jarrod.millman $
6: (0)             """
7: (0)             import warnings
8: (0)             import itertools
9: (0)             import pytest
10: (0)            import numpy as np
11: (0)            from numpy.core.numeric import normalize_axis_tuple
12: (0)            from numpy.testing import (
13: (4)                assert_warnings, suppress_warnings
14: (4)            )
15: (0)            from numpy.ma.testutils import (
16: (4)                assert_, assert_array_equal, assert_equal, assert_almost_equal
17: (4)            )
18: (0)            from numpy.ma.core import (
19: (4)                array, arange, masked, MaskedArray, masked_array, getmaskarray, shape,
20: (4)                nomask, ones, zeros, count
21: (4)            )
22: (0)            from numpy.ma.extras import (
23: (4)                atleast_1d, atleast_2d, atleast_3d, mr_, dot, polyfit, cov, corrcoef,
24: (4)                median, average, unique, setxor1d, setdiff1d, union1d, intersect1d, in1d,
25: (4)                ediff1d, apply_over_axes, apply_along_axis, compress_nd, compress_rowcols,
26: (4)                mask_rowcols, clump_masked, clump_unmasked, flatnotmasked_contiguous,
27: (4)                notmasked_contiguous, notmasked_edges, masked_all, masked_all_like, isin,
28: (4)                diagflat, ndenumerate, stack, vstack
29: (4)            )
30: (0)            class TestGeneric:
31: (4)                def test_masked_all(self):
32: (8)                    test = masked_all((2,), dtype=float)
33: (8)                    control = array([1, 1], mask=[1, 1], dtype=float)
34: (8)                    assert_equal(test, control)
35: (8)                    dt = np.dtype({'names': ['a', 'b'], 'formats': ['f', 'f']})
36: (8)                    test = masked_all((2,), dtype=dt)
37: (8)                    control = array([(0, 0), (0, 0)], mask=[(1, 1), (1, 1)], dtype=dt)
38: (8)                    assert_equal(test, control)
39: (8)                    test = masked_all((2, 2), dtype=dt)
40: (8)                    control = array([(0, 0), (0, 0)], [(0, 0), (0, 0)],
41: (24)                        mask=[[(1, 1), (1, 1)], [(1, 1), (1, 1)]],
42: (24)                        dtype=dt)
43: (8)                    assert_equal(test, control)
44: (8)                    dt = np.dtype([('a', 'f'), ('b', [('ba', 'f'), ('bb', 'f')])])
45: (8)                    test = masked_all((2,), dtype=dt)
46: (8)                    control = array([(1, (1, 1)), (1, (1, 1))],
47: (24)                        mask=[(1, (1, 1)), (1, (1, 1))], dtype=dt)
48: (8)                    assert_equal(test, control)
49: (8)                    test = masked_all((2,), dtype=dt)
50: (8)                    control = array([(1, (1, 1)), (1, (1, 1))],
51: (24)                        mask=[(1, (1, 1)), (1, (1, 1))], dtype=dt)
52: (8)                    assert_equal(test, control)
53: (8)                    test = masked_all((1, 1), dtype=dt)
54: (8)                    control = array([(1, (1, 1))], mask=[(1, (1, 1))], dtype=dt)
55: (8)                    assert_equal(test, control)
56: (4)                    def test_masked_all_with_object_nested(self):
57: (8)                        my_dtype = np.dtype([('b', [(('c', object)], (1,))]))
58: (8)                        masked_arr = np.ma.masked_all((1,), my_dtype)

```

```

59: (8)             assert_equal(type(masked_arr['b']), np.ma.core.MaskedArray)
60: (8)             assert_equal(type(masked_arr['b']['c']), np.ma.core.MaskedArray)
61: (8)             assert_equal(len(masked_arr['b']['c']), 1)
62: (8)             assert_equal(masked_arr['b']['c'].shape, (1, 1))
63: (8)             assert_equal(masked_arr['b']['c']._fill_value.shape, ())
64: (4)             def test_masked_all_with_object(self):
65: (8)                 my_dtype = np.dtype([('b', (object, (1,)))])
66: (8)                 masked_arr = np.ma.masked_all((1,), my_dtype)
67: (8)                 assert_equal(type(masked_arr['b']), np.ma.core.MaskedArray)
68: (8)                 assert_equal(len(masked_arr['b']), 1)
69: (8)                 assert_equal(masked_arr['b'].shape, (1, 1))
70: (8)                 assert_equal(masked_arr['b']._fill_value.shape, ())
71: (4)             def test_masked_all_like(self):
72: (8)                 base = array([1, 2], dtype=float)
73: (8)                 test = masked_all_like(base)
74: (8)                 control = array([1, 1], mask=[1, 1], dtype=float)
75: (8)                 assert_equal(test, control)
76: (8)                 dt = np.dtype({'names': ['a', 'b'], 'formats': ['f', 'f']})
77: (8)                 base = array([(0, 0), (0, 0)], mask=[(1, 1), (1, 1)], dtype=dt)
78: (8)                 test = masked_all_like(base)
79: (8)                 control = array([(10, 10), (10, 10)], mask=[(1, 1), (1, 1)], dtype=dt)
80: (8)                 assert_equal(test, control)
81: (8)                 dt = np.dtype([('a', 'f'), ('b', [('ba', 'f'), ('bb', 'f')])])
82: (8)                 control = array([(1, (1, 1)), (1, (1, 1))],
83: (24)                               mask=[(1, (1, 1)), (1, (1, 1))], dtype=dt)
84: (8)                 test = masked_all_like(control)
85: (8)                 assert_equal(test, control)
86: (4)             def check_clump(self, f):
87: (8)                 for i in range(1, 7):
88: (12)                     for j in range(2**i):
89: (16)                         k = np.arange(i, dtype=int)
90: (16)                         ja = np.full(i, j, dtype=int)
91: (16)                         a = masked_array(2**k)
92: (16)                         a.mask = (ja & (2**k)) != 0
93: (16)                         s = 0
94: (16)                         for sl in f(a):
95: (20)                             s += a.data[sl].sum()
96: (16)                         if f == clump_unmasked:
97: (20)                             assert_equal(a.compressed().sum(), s)
98: (16)                         else:
99: (20)                             a.mask = ~a.mask
100: (20)                            assert_equal(a.compressed().sum(), s)
101: (4)             def test_clump_masked(self):
102: (8)                 a = masked_array(np.arange(10))
103: (8)                 a[[0, 1, 2, 6, 8, 9]] = masked
104: (8)                 test = clump_masked(a)
105: (8)                 control = [slice(0, 3), slice(6, 7), slice(8, 10)]
106: (8)                 assert_equal(test, control)
107: (8)                 self.check_clump(clump_masked)
108: (4)             def test_clump_unmasked(self):
109: (8)                 a = masked_array(np.arange(10))
110: (8)                 a[[0, 1, 2, 6, 8, 9]] = masked
111: (8)                 test = clump_unmasked(a)
112: (8)                 control = [slice(3, 6), slice(7, 8), ]
113: (8)                 assert_equal(test, control)
114: (8)                 self.check_clump(clump_unmasked)
115: (4)             def test_flatnotmasked_contiguous(self):
116: (8)                 a = arange(10)
117: (8)                 test = flatnotmasked_contiguous(a)
118: (8)                 assert_equal(test, [slice(0, a.size)])
119: (8)                 a.mask = np.zeros(10, dtype=bool)
120: (8)                 assert_equal(test, [slice(0, a.size)])
121: (8)                 a[(a < 3) | (a > 8) | (a == 5)] = masked
122: (8)                 test = flatnotmasked_contiguous(a)
123: (8)                 assert_equal(test, [slice(3, 5), slice(6, 9)])
124: (8)                 a[:] = masked
125: (8)                 test = flatnotmasked_contiguous(a)
126: (8)                 assert_equal(test, [])
127: (0)             class TestAverage:

```

```

128: (4)
129: (8)     def test_testAverage1(self):
130: (8)         ott = array([0., 1., 2., 3.], mask=[True, False, False, False])
131: (8)         assert_equal(2.0, average(ott, axis=0))
132: (8)         assert_equal(2.0, average(ott, weights=[1., 1., 2., 1.]))
133: (8)         result, wts = average(ott, weights=[1., 1., 2., 1.], returned=True)
134: (8)         assert_equal(2.0, result)
135: (8)         assert_(wts == 4.0)
136: (8)         ott[:] = masked
137: (8)         assert_equal(average(ott, axis=0).mask, [True])
138: (8)         ott = array([0., 1., 2., 3.], mask=[True, False, False, False])
139: (8)         ott[:, 1] = masked
140: (8)         assert_equal(average(ott, axis=0), [2.0, 0.0])
141: (8)         assert_equal(average(ott, axis=1).mask[0], [True])
142: (8)         assert_equal([2., 0.], average(ott, axis=0))
143: (8)         result, wts = average(ott, axis=0, returned=True)
144: (8)         assert_equal(wts, [1., 0.])
145: (4)     def test_testAverage2(self):
146: (8)         w1 = [0, 1, 1, 1, 1, 0]
147: (8)         w2 = [[0, 1, 1, 1, 1, 0], [1, 0, 0, 0, 0, 1]]
148: (8)         x = arange(6, dtype=np.float_)
149: (8)         assert_equal(average(x, axis=0), 2.5)
150: (8)         assert_equal(average(x, axis=0, weights=w1), 2.5)
151: (8)         y = array([arange(6, dtype=np.float_), 2.0 * arange(6)])
152: (8)         assert_equal(average(y, None), np.add.reduce(np.arange(6)) * 3. / 12.)
153: (8)         assert_equal(average(y, axis=0), np.arange(6) * 3. / 2.)
154: (8)         assert_equal(average(y, axis=1),
155: (21)             [average(x, axis=0), average(x, axis=0) * 2.0])
156: (8)         assert_equal(average(y, None, weights=w2), 20. / 6.)
157: (8)         assert_equal(average(y, axis=0, weights=w2),
158: (21)             [0., 1., 2., 3., 4., 10.])
159: (8)         assert_equal(average(y, axis=1),
160: (21)             [average(x, axis=0), average(x, axis=0) * 2.0])
161: (8)         m1 = zeros(6)
162: (8)         m2 = [0, 0, 1, 1, 0, 0]
163: (8)         m3 = [[0, 0, 1, 1, 0, 0], [0, 1, 1, 1, 1, 0]]
164: (8)         m4 = ones(6)
165: (8)         m5 = [0, 1, 1, 1, 1, 1]
166: (8)         assert_equal(average(masked_array(x, m1), axis=0), 2.5)
167: (8)         assert_equal(average(masked_array(x, m2), axis=0), 2.5)
168: (8)         assert_equal(average(masked_array(x, m4), axis=0).mask, [True])
169: (8)         assert_equal(average(masked_array(x, m5), axis=0), 0.0)
170: (8)         assert_equal(count(average(masked_array(x, m4), axis=0)), 0)
171: (8)         z = masked_array(y, m3)
172: (8)         assert_equal(average(z, None), 20. / 6.)
173: (8)         assert_equal(average(z, axis=0), [0., 1., 99., 99., 4.0, 7.5])
174: (8)         assert_equal(average(z, axis=1), [2.5, 5.0])
175: (8)         assert_equal(average(z, axis=0, weights=w2),
176: (21)             [0., 1., 99., 99., 4.0, 10.0])
177: (4)     def test_testAverage3(self):
178: (8)         a = arange(6)
179: (8)         b = arange(6) * 3
180: (8)         r1, w1 = average([[a, b], [b, a]], axis=1, returned=True)
181: (8)         assert_equal(shape(r1), shape(w1))
182: (8)         assert_equal(r1.shape, w1.shape)
183: (8)         r2, w2 = average(ones((2, 2, 3)), axis=0, weights=[3, 1],
184: (8)             returned=True)
185: (8)         assert_equal(shape(w2), shape(r2))
186: (8)         r2, w2 = average(ones((2, 2, 3)), returned=True)
187: (8)         assert_equal(shape(w2), shape(r2))
188: (8)         r2, w2 = average(ones((2, 2, 3)), weights=ones((2, 2, 3)),
189: (8)             returned=True)
190: (8)         assert_equal(shape(w2), shape(r2))
191: (8)         a2d = array([[1, 2], [0, 4]], float)
192: (8)         a2dm = masked_array(a2d, [[False, False], [True, False]])
193: (8)         a2da = average(a2d, axis=0)
194: (8)         assert_equal(a2da, [0.5, 3.0])

```

```

195: (8)           a2dma = average(a2dm, axis=None)
196: (8)           assert_equal(a2dma, 7. / 3.)
197: (8)           a2dma = average(a2dm, axis=1)
198: (8)           assert_equal(a2dma, [1.5, 4.0])
199: (4)           def test_testAverage4(self):
200: (8)             x = np.array([2, 3, 4]).reshape(3, 1)
201: (8)             b = np.ma.array(x, mask=[[False], [False], [True]])
202: (8)             w = np.array([4, 5, 6]).reshape(3, 1)
203: (8)             actual = average(b, weights=w, axis=1, keepdims=True)
204: (8)             desired = masked_array([[2.], [3.], [4.]], [[False], [False], [True]])
205: (8)             assert_equal(actual, desired)
206: (4)           def test_onintegers_with_mask(self):
207: (8)             a = average(array([1, 2]))
208: (8)             assert_equal(a, 1.5)
209: (8)             a = average(array([1, 2, 3, 4], mask=[False, False, True, True]))
210: (8)             assert_equal(a, 1.5)
211: (4)           def test_complex(self):
212: (8)             mask = np.array([[0, 0, 0, 1, 0],
213: (25)                           [0, 1, 0, 0, 0]], dtype=bool)
214: (8)             a = masked_array([[0, 1+2j, 3+4j, 5+6j, 7+8j],
215: (26)                           [9j, 0+1j, 2+3j, 4+5j, 7+7j]],
216: (25)                           mask=mask)
217: (8)             av = average(a)
218: (8)             expected = np.average(a.compressed())
219: (8)             assert_almost_equal(av.real, expected.real)
220: (8)             assert_almost_equal(av.imag, expected.imag)
221: (8)             av0 = average(a, axis=0)
222: (8)             expected0 = average(a.real, axis=0) + average(a.imag, axis=0)*1j
223: (8)             assert_almost_equal(av0.real, expected0.real)
224: (8)             assert_almost_equal(av0.imag, expected0.imag)
225: (8)             av1 = average(a, axis=1)
226: (8)             expected1 = average(a.real, axis=1) + average(a.imag, axis=1)*1j
227: (8)             assert_almost_equal(av1.real, expected1.real)
228: (8)             assert_almost_equal(av1.imag, expected1.imag)
229: (8)             wts = np.array([[0.5, 1.0, 2.0, 1.0, 0.5],
230: (24)                           [1.0, 1.0, 1.0, 1.0, 1.0]])
231: (8)             wav = average(a, weights=wts)
232: (8)             expected = np.average(a.compressed(), weights=wts[~mask])
233: (8)             assert_almost_equal(wav.real, expected.real)
234: (8)             assert_almost_equal(wav.imag, expected.imag)
235: (8)             wav0 = average(a, weights=wts, axis=0)
236: (8)             expected0 = (average(a.real, weights=wts, axis=0) +
237: (21)                           average(a.imag, weights=wts, axis=0)*1j)
238: (8)             assert_almost_equal(wav0.real, expected0.real)
239: (8)             assert_almost_equal(wav0.imag, expected0.imag)
240: (8)             wav1 = average(a, weights=wts, axis=1)
241: (8)             expected1 = (average(a.real, weights=wts, axis=1) +
242: (21)                           average(a.imag, weights=wts, axis=1)*1j)
243: (8)             assert_almost_equal(wav1.real, expected1.real)
244: (8)             assert_almost_equal(wav1.imag, expected1.imag)
245: (4)           @pytest.mark.parametrize(
246: (8)             'x, axis, expected_avg, weights, expected_wavg, expected_wsum',
247: (8)             '[[[1, 2, 3], None, [2.0], [3, 4, 1], [1.75], [8.0]],
248: (9)                           [[[1, 2, 5], [1, 6, 11]], 0, [[1.0, 4.0, 8.0]],
249: (10)                             [1, 3], [[1.0, 5.0, 9.5]], [[4, 4, 4]]]]',
250: (4)
251: (4)           )
252: (28)           def test_basic_keepdims(self, x, axis, expected_avg,
253: (8)                         weights, expected_wavg, expected_wsum):
254: (8)             avg = np.ma.average(x, axis=axis, keepdims=True)
255: (8)             assert avg.shape == np.shape(expected_avg)
256: (8)             assert_array_equal(avg, expected_avg)
257: (8)             wavg = np.ma.average(x, axis=axis, weights=weights, keepdims=True)
258: (8)             assert wavg.shape == np.shape(expected_wavg)
259: (8)             assert_array_equal(wavg, expected_wavg)
260: (35)             wavg, wsum = np.ma.average(x, axis=axis, weights=weights,
261: (8)                           returned=True, keepdims=True)
262: (8)             assert wavg.shape == np.shape(expected_wavg)
263: (8)             assert_array_equal(wavg, expected_wavg)

```

```

264: (8)             assert_array_equal(wsum, expected_wsum)
265: (4)             def test_masked_weights(self):
266: (8)                 a = np.ma.array(np.arange(9).reshape(3, 3),
267: (24)                               mask=[[1, 0, 0], [1, 0, 0], [0, 0, 0]])
268: (8)                 weights_unmasked = masked_array([5, 28, 31], mask=False)
269: (8)                 weights_masked = masked_array([5, 28, 31], mask=[1, 0, 0])
270: (8)                 avg_unmasked = average(a, axis=0,
271: (31)                               weights=weights_unmasked, returned=False)
272: (8)                 expected_unmasked = np.array([6.0, 5.21875, 6.21875])
273: (8)                 assert_almost_equal(avg_unmasked, expected_unmasked)
274: (8)                 avg_masked = average(a, axis=0, weights=weights_masked,
275: (8)                               expected_masked = np.array([6.0, 5.576271186440678,
276: (8)                               6.576271186440678])
277: (8)
278: (19)
279: (8)
280: (8)
281: (8)
282: (12)
283: (12)
284: (12)
285: (8)
286: (12)
287: (12)
288: (12)
289: (8)
290: (8)
291: (8)
292: (30)
293: (8)
294: (8)
295: (0)             class TestConcatenator:
296: (4)                 def test_1d(self):
297: (8)                     assert_array_equal(mr_[1, 2, 3, 4, 5, 6], array([1, 2, 3, 4, 5, 6]))
298: (8)                     b = ones(5)
299: (8)                     m = [1, 0, 0, 0, 0]
300: (8)                     d = masked_array(b, mask=m)
301: (8)                     c = mr_[d, 0, 0, d]
302: (8)                     assert_(isinstance(c, MaskedArray))
303: (8)                     assert_array_equal(c, [1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1])
304: (8)                     assert_array_equal(c.mask, mr_[m, 0, 0, m])
305: (4)                 def test_2d(self):
306: (8)                     a_1 = np.random.rand(5, 5)
307: (8)                     a_2 = np.random.rand(5, 5)
308: (8)                     m_1 = np.round(np.random.rand(5, 5), 0)
309: (8)                     m_2 = np.round(np.random.rand(5, 5), 0)
310: (8)                     b_1 = masked_array(a_1, mask=m_1)
311: (8)                     b_2 = masked_array(a_2, mask=m_2)
312: (8)                     d = mr_['1', b_1, b_2]
313: (8)                     assert_(d.shape == (5, 10))
314: (8)                     assert_array_equal(d[:, :5], b_1)
315: (8)                     assert_array_equal(d[:, 5:], b_2)
316: (8)                     assert_array_equal(d.mask, np.r_['1', m_1, m_2])
317: (8)                     d = mr_[b_1, b_2]
318: (8)                     assert_(d.shape == (10, 5))
319: (8)                     assert_array_equal(d[:5, :], b_1)
320: (8)                     assert_array_equal(d[5:, :], b_2)
321: (8)                     assert_array_equal(d.mask, np.r_[m_1, m_2])
322: (4)                 def test_masked_constant(self):
323: (8)                     actual = mr_[np.ma.masked, 1]
324: (8)                     assert_equal(actual.mask, [True, False])
325: (8)                     assert_equal(actual.data[1], 1)
326: (8)                     actual = mr_[[1, 2], np.ma.masked]
327: (8)                     assert_equal(actual.mask, [False, False, True])
328: (8)                     assert_equal(actual.data[:2], [1, 2])
329: (0)             class TestNotMasked:
330: (4)                 def test_edges(self):

```

```

331: (8)           data = masked_array(np.arange(25).reshape(5, 5),
332: (28)             mask=[[0, 0, 1, 0, 0],
333: (34)               [0, 0, 0, 1, 1],
334: (34)               [1, 1, 0, 0, 0],
335: (34)               [0, 0, 0, 0, 0],
336: (34)               [1, 1, 1, 0, 0]],)
337: (8)           test = notmasked_edges(data, None)
338: (8)           assert_equal(test, [0, 24])
339: (8)           test = notmasked_edges(data, 0)
340: (8)           assert_equal(test[0], [(0, 0, 1, 0, 0), (0, 1, 2, 3, 4)])
341: (8)           assert_equal(test[1], [(3, 3, 3, 4, 4), (0, 1, 2, 3, 4)])
342: (8)           test = notmasked_edges(data, 1)
343: (8)           assert_equal(test[0], [(0, 1, 2, 3, 4), (0, 0, 2, 0, 3)])
344: (8)           assert_equal(test[1], [(0, 1, 2, 3, 4), (4, 2, 4, 4, 4)])
345: (8)           test = notmasked_edges(data.data, None)
346: (8)           assert_equal(test, [0, 24])
347: (8)           test = notmasked_edges(data.data, 0)
348: (8)           assert_equal(test[0], [(0, 0, 0, 0, 0), (0, 1, 2, 3, 4)])
349: (8)           assert_equal(test[1], [(4, 4, 4, 4, 4), (0, 1, 2, 3, 4)])
350: (8)           test = notmasked_edges(data.data, -1)
351: (8)           assert_equal(test[0], [(0, 1, 2, 3, 4), (0, 0, 0, 0, 0)])
352: (8)           assert_equal(test[1], [(0, 1, 2, 3, 4), (4, 4, 4, 4, 4)])
353: (8)           data[-2] = masked
354: (8)           test = notmasked_edges(data, 0)
355: (8)           assert_equal(test[0], [(0, 0, 1, 0, 0), (0, 1, 2, 3, 4)])
356: (8)           assert_equal(test[1], [(1, 1, 2, 4, 4), (0, 1, 2, 3, 4)])
357: (8)           test = notmasked_edges(data, -1)
358: (8)           assert_equal(test[0], [(0, 1, 2, 4), (0, 0, 2, 3)])
359: (8)           assert_equal(test[1], [(0, 1, 2, 4), (4, 2, 4, 4)])
360: (4)           def test_contiguous(self):
361: (8)             a = masked_array(np.arange(24).reshape(3, 8),
362: (25)               mask=[[0, 0, 0, 0, 1, 1, 1, 1],
363: (31)                 [1, 1, 1, 1, 1, 1, 1, 1],
364: (31)                 [0, 0, 0, 0, 0, 0, 1, 0]])
365: (8)             tmp = notmasked_contiguous(a, None)
366: (8)             assert_equal(tmp, [
367: (12)               slice(0, 4, None),
368: (12)               slice(16, 22, None),
369: (12)               slice(23, 24, None)
370: (8)             ])
371: (8)             tmp = notmasked_contiguous(a, 0)
372: (8)             assert_equal(tmp, [
373: (12)               [slice(0, 1, None), slice(2, 3, None)],
374: (12)               [slice(0, 1, None), slice(2, 3, None)],
375: (12)               [slice(0, 1, None), slice(2, 3, None)],
376: (12)               [slice(0, 1, None), slice(2, 3, None)],
377: (12)               [slice(2, 3, None)],
378: (12)               [slice(2, 3, None)],
379: (12)               [],
380: (12)               [slice(2, 3, None)]
381: (8)             ])
382: (8)             tmp = notmasked_contiguous(a, 1)
383: (8)             assert_equal(tmp, [
384: (12)               [slice(0, 4, None)],
385: (12)               [],
386: (12)               [slice(0, 6, None), slice(7, 8, None)]
387: (8)             ])
388: (0)           class TestCompressFunctions:
389: (4)             def test_compress_nd(self):
390: (8)               x = np.array(list(range(3*4*5))).reshape(3, 4, 5)
391: (8)               m = np.zeros((3,4,5)).astype(bool)
392: (8)               m[1,1,1] = True
393: (8)               x = array(x, mask=m)
394: (8)               a = compress_nd(x)
395: (8)               assert_equal(a, [[[ 0,  2,  3,  4],
396: (26)                 [10, 12, 13, 14],
397: (26)                 [15, 17, 18, 19]],
398: (25)                   [[40, 42, 43, 44],
399: (26)                     [50, 52, 53, 54],
```

```

400: (26) [55, 57, 58, 59]]])
401: (8) a = compress_nd(x, 0)
402: (8) assert_equal(a, [[[ 0, 1, 2, 3, 4],
403: (26) [ 5, 6, 7, 8, 9],
404: (26) [10, 11, 12, 13, 14],
405: (26) [15, 16, 17, 18, 19]],
406: (25) [[40, 41, 42, 43, 44],
407: (26) [45, 46, 47, 48, 49],
408: (26) [50, 51, 52, 53, 54],
409: (26) [55, 56, 57, 58, 59]]])
410: (8) a = compress_nd(x, 1)
411: (8) assert_equal(a, [[[ 0, 1, 2, 3, 4],
412: (26) [10, 11, 12, 13, 14],
413: (26) [15, 16, 17, 18, 19]],
414: (25) [[20, 21, 22, 23, 24],
415: (26) [30, 31, 32, 33, 34],
416: (26) [35, 36, 37, 38, 39]],
417: (25) [[40, 41, 42, 43, 44],
418: (26) [50, 51, 52, 53, 54],
419: (26) [55, 56, 57, 58, 59]]])
420: (8) a2 = compress_nd(x, (1,))
421: (8) a3 = compress_nd(x, -2)
422: (8) a4 = compress_nd(x, (-2,))
423: (8) assert_equal(a, a2)
424: (8) assert_equal(a, a3)
425: (8) assert_equal(a, a4)
426: (8) a = compress_nd(x, 2)
427: (8) assert_equal(a, [[[ 0, 2, 3, 4],
428: (26) [ 5, 7, 8, 9],
429: (26) [10, 12, 13, 14],
430: (26) [15, 17, 18, 19]],
431: (25) [[20, 22, 23, 24],
432: (26) [25, 27, 28, 29],
433: (26) [30, 32, 33, 34],
434: (26) [35, 37, 38, 39]],
435: (25) [[40, 42, 43, 44],
436: (26) [45, 47, 48, 49],
437: (26) [50, 52, 53, 54],
438: (26) [55, 57, 58, 59]]])
439: (8) a2 = compress_nd(x, (2,))
440: (8) a3 = compress_nd(x, -1)
441: (8) a4 = compress_nd(x, (-1,))
442: (8) assert_equal(a, a2)
443: (8) assert_equal(a, a3)
444: (8) assert_equal(a, a4)
445: (8) a = compress_nd(x, (0, 1))
446: (8) assert_equal(a, [[[ 0, 1, 2, 3, 4],
447: (26) [10, 11, 12, 13, 14],
448: (26) [15, 16, 17, 18, 19]],
449: (25) [[40, 41, 42, 43, 44],
450: (26) [50, 51, 52, 53, 54],
451: (26) [55, 56, 57, 58, 59]]])
452: (8) a2 = compress_nd(x, (0, -2))
453: (8) assert_equal(a, a2)
454: (8) a = compress_nd(x, (1, 2))
455: (8) assert_equal(a, [[[ 0, 2, 3, 4],
456: (26) [10, 12, 13, 14],
457: (26) [15, 17, 18, 19]],
458: (25) [[20, 22, 23, 24],
459: (26) [30, 32, 33, 34],
460: (26) [35, 37, 38, 39]],
461: (25) [[40, 42, 43, 44],
462: (26) [50, 52, 53, 54],
463: (26) [55, 57, 58, 59]]])
464: (8) a2 = compress_nd(x, (-2, 2))
465: (8) a3 = compress_nd(x, (1, -1))
466: (8) a4 = compress_nd(x, (-2, -1))
467: (8) assert_equal(a, a2)
468: (8) assert_equal(a, a3)

```

```

469: (8)             assert_equal(a, a4)
470: (8)             a = compress_nd(x, (0, 2))
471: (8)             assert_equal(a, [[[ 0,  2,  3,  4],
472: (26)                     [ 5,  7,  8,  9],
473: (26)                     [10, 12, 13, 14],
474: (26)                     [15, 17, 18, 19]],
475: (25)                     [[40, 42, 43, 44],
476: (26)                     [45, 47, 48, 49],
477: (26)                     [50, 52, 53, 54],
478: (26)                     [55, 57, 58, 59]]])
479: (8)             a2 = compress_nd(x, (0, -1))
480: (8)             assert_equal(a, a2)
481: (4)             def test_compress_rowcols(self):
482: (8)                 x = array(np.arange(9).reshape(3, 3),
483: (18)                     mask=[[1, 0, 0], [0, 0, 0], [0, 0, 0]])
484: (8)                 assert_equal(compress_rowcols(x), [[4, 5], [7, 8]])
485: (8)                 assert_equal(compress_rowcols(x, 0), [[3, 4, 5], [6, 7, 8]])
486: (8)                 assert_equal(compress_rowcols(x, 1), [[1, 2], [4, 5], [7, 8]])
487: (8)                 x = array(x._data, mask=[[0, 0, 0], [0, 1, 0], [0, 0, 0]])
488: (8)                 assert_equal(compress_rowcols(x), [[0, 2], [6, 8]])
489: (8)                 assert_equal(compress_rowcols(x, 0), [[0, 1, 2], [6, 7, 8]])
490: (8)                 assert_equal(compress_rowcols(x, 1), [[0, 2], [3, 5], [6, 8]])
491: (8)                 x = array(x._data, mask=[[1, 0, 0], [0, 1, 0], [0, 0, 0]])
492: (8)                 assert_equal(compress_rowcols(x), [[8]])
493: (8)                 assert_equal(compress_rowcols(x, 0), [[6, 7, 8]])
494: (8)                 assert_equal(compress_rowcols(x, 1), [[2], [5], [8]])
495: (8)                 x = array(x._data, mask=[[1, 0, 0], [0, 1, 0], [0, 0, 1]])
496: (8)                 assert_equal(compress_rowcols(x).size, 0)
497: (8)                 assert_equal(compress_rowcols(x, 0).size, 0)
498: (8)                 assert_equal(compress_rowcols(x, 1).size, 0)
499: (4)             def test_mask_rowcols(self):
500: (8)                 x = array(np.arange(9).reshape(3, 3),
501: (18)                     mask=[[1, 0, 0], [0, 0, 0], [0, 0, 0]])
502: (8)                 assert_equal(mask_rowcols(x).mask,
503: (21)                     [[1, 1, 1], [1, 0, 0], [1, 0, 0]])
504: (8)                 assert_equal(mask_rowcols(x, 0).mask,
505: (21)                     [[1, 1, 1], [0, 0, 0], [0, 0, 0]])
506: (8)                 assert_equal(mask_rowcols(x, 1).mask,
507: (21)                     [[1, 0, 0], [1, 0, 0], [1, 0, 0]])
508: (8)                 x = array(x._data, mask=[[0, 0, 0], [0, 1, 0], [0, 0, 0]])
509: (8)                 assert_equal(mask_rowcols(x).mask,
510: (21)                     [[0, 1, 0], [1, 1, 1], [0, 1, 0]])
511: (8)                 assert_equal(mask_rowcols(x, 0).mask,
512: (21)                     [[0, 0, 0], [1, 1, 1], [0, 0, 0]])
513: (8)                 assert_equal(mask_rowcols(x, 1).mask,
514: (21)                     [[0, 1, 0], [0, 1, 0], [0, 1, 0]])
515: (8)                 x = array(x._data, mask=[[1, 0, 0], [0, 1, 0], [0, 0, 0]])
516: (8)                 assert_equal(mask_rowcols(x).mask,
517: (21)                     [[1, 1, 1], [1, 1, 1], [1, 1, 1]])
518: (8)                 assert_equal(mask_rowcols(x, 0).mask,
519: (21)                     [[1, 1, 1], [1, 1, 1], [0, 0, 0]])
520: (8)                 assert_equal(mask_rowcols(x, 1).mask,
521: (21)                     [[1, 1, 0], [1, 1, 0], [1, 1, 0]])
522: (8)                 x = array(x._data, mask=[[1, 0, 0], [0, 1, 0], [0, 0, 1]])
523: (8)                 assert_(mask_rowcols(x).all() is masked)
524: (8)                 assert_(mask_rowcols(x, 0).all() is masked)
525: (8)                 assert_(mask_rowcols(x, 1).all() is masked)
526: (8)                 assert_(mask_rowcols(x).mask.all())
527: (8)                 assert_(mask_rowcols(x, 0).mask.all())
528: (8)                 assert_(mask_rowcols(x, 1).mask.all())
529: (4)             @pytest.mark.parametrize("axis", [None, 0, 1])
530: (4)             @pytest.mark.parametrize(["func", "rowcols_axis"],
531: (29)                     [(np.ma.mask_rows, 0), (np.ma.mask_cols, 1)])
532: (4)             def test_mask_row_cols_axis_deprecation(self, axis, func, rowcols_axis):
533: (8)                 x = array(np.arange(9).reshape(3, 3),
534: (18)                     mask=[[1, 0, 0], [0, 0, 0], [0, 0, 0]])
535: (8)                 with assert_warnings(DeprecationWarning):
536: (12)                     res = func(x, axis=axis)
537: (12)                     assert_equal(res, mask_rowcols(x, rowcols_axis))

```

```

538: (4)          def test_dot(self):
539: (8)          n = np.arange(1, 7)
540: (8)          m = [1, 0, 0, 0, 0, 0]
541: (8)          a = masked_array(n, mask=m).reshape(2, 3)
542: (8)          b = masked_array(n, mask=m).reshape(3, 2)
543: (8)          c = dot(a, b, strict=True)
544: (8)          assert_equal(c.mask, [[1, 1], [1, 0]])
545: (8)          c = dot(b, a, strict=True)
546: (8)          assert_equal(c.mask, [[1, 1, 1], [1, 0, 0], [1, 0, 0]])
547: (8)          c = dot(a, b, strict=False)
548: (8)          assert_equal(c, np.dot(a.filled(0), b.filled(0)))
549: (8)          c = dot(b, a, strict=False)
550: (8)          assert_equal(c, np.dot(b.filled(0), a.filled(0)))
551: (8)          m = [0, 0, 0, 0, 0, 1]
552: (8)          a = masked_array(n, mask=m).reshape(2, 3)
553: (8)          b = masked_array(n, mask=m).reshape(3, 2)
554: (8)          c = dot(a, b, strict=True)
555: (8)          assert_equal(c.mask, [[0, 1], [1, 1]])
556: (8)          c = dot(b, a, strict=True)
557: (8)          assert_equal(c.mask, [[0, 0, 1], [0, 0, 1], [1, 1, 1]])
558: (8)          c = dot(a, b, strict=False)
559: (8)          assert_equal(c, np.dot(a.filled(0), b.filled(0)))
560: (8)          assert_equal(c, dot(a, b))
561: (8)          c = dot(b, a, strict=False)
562: (8)          assert_equal(c, np.dot(b.filled(0), a.filled(0)))
563: (8)          m = [0, 0, 0, 0, 0, 0]
564: (8)          a = masked_array(n, mask=m).reshape(2, 3)
565: (8)          b = masked_array(n, mask=m).reshape(3, 2)
566: (8)          c = dot(a, b)
567: (8)          assert_equal(c.mask, nomask)
568: (8)          c = dot(b, a)
569: (8)          assert_equal(c.mask, nomask)
570: (8)          a = masked_array(n, mask=[1, 0, 0, 0, 0, 0]).reshape(2, 3)
571: (8)          b = masked_array(n, mask=[0, 0, 0, 0, 0, 0]).reshape(3, 2)
572: (8)          c = dot(a, b, strict=True)
573: (8)          assert_equal(c.mask, [[1, 1], [0, 0]])
574: (8)          c = dot(a, b, strict=False)
575: (8)          assert_equal(c, np.dot(a.filled(0), b.filled(0)))
576: (8)          c = dot(b, a, strict=True)
577: (8)          assert_equal(c.mask, [[1, 0, 0], [1, 0, 0], [1, 0, 0]])
578: (8)          c = dot(b, a, strict=False)
579: (8)          assert_equal(c, np.dot(b.filled(0), a.filled(0)))
580: (8)          a = masked_array(n, mask=[0, 0, 0, 0, 0, 1]).reshape(2, 3)
581: (8)          b = masked_array(n, mask=[0, 0, 0, 0, 0, 0]).reshape(3, 2)
582: (8)          c = dot(a, b, strict=True)
583: (8)          assert_equal(c.mask, [[0, 0], [1, 1]])
584: (8)          c = dot(a, b)
585: (8)          assert_equal(c, np.dot(a.filled(0), b.filled(0)))
586: (8)          c = dot(b, a, strict=True)
587: (8)          assert_equal(c.mask, [[0, 0, 1], [0, 0, 1], [0, 0, 1]])
588: (8)          c = dot(b, a, strict=False)
589: (8)          assert_equal(c, np.dot(b.filled(0), a.filled(0)))
590: (8)          a = masked_array(n, mask=[0, 0, 0, 0, 0, 1]).reshape(2, 3)
591: (8)          b = masked_array(n, mask=[0, 0, 1, 0, 0, 0]).reshape(3, 2)
592: (8)          c = dot(a, b, strict=True)
593: (8)          assert_equal(c.mask, [[1, 0], [1, 1]])
594: (8)          c = dot(a, b, strict=False)
595: (8)          assert_equal(c, np.dot(a.filled(0), b.filled(0)))
596: (8)          c = dot(b, a, strict=True)
597: (8)          assert_equal(c.mask, [[0, 0, 1], [1, 1, 1], [0, 0, 1]])
598: (8)          c = dot(b, a, strict=False)
599: (8)          assert_equal(c, np.dot(b.filled(0), a.filled(0)))
600: (8)          a = masked_array(np.arange(8).reshape(2, 2, 2),
601: (25)                      mask=[[1, 0], [0, 0], [[0, 0], [0, 0]]])
602: (8)          b = masked_array(np.arange(8).reshape(2, 2, 2),
603: (25)                      mask=[[0, 0], [0, 0], [[0, 0], [0, 1]]])
604: (8)          c = dot(a, b, strict=True)
605: (8)          assert_equal(c.mask,
606: (21)                      [[[1, 1], [1, 1]], [[0, 0], [0, 1]]]),

```

```

607: (22)                                [[[0, 0], [0, 1]], [[0, 0], [0, 1]]])
608: (8)       c = dot(a, b, strict=False)
609: (8)       assert_equal(c.mask,
610: (21)           [[[0, 0], [0, 1]], [[0, 0], [0, 0]]],
611: (22)           [[[0, 0], [0, 0]], [[0, 0], [0, 0]]])
612: (8)       c = dot(b, a, strict=True)
613: (8)       assert_equal(c.mask,
614: (21)           [[[1, 0], [0, 0]], [[1, 0], [0, 0]]],
615: (22)           [[[1, 0], [0, 0]], [[1, 1], [1, 1]]])
616: (8)       c = dot(b, a, strict=False)
617: (8)       assert_equal(c.mask,
618: (21)           [[[0, 0], [0, 0]], [[0, 0], [0, 0]]],
619: (22)           [[[0, 0], [0, 0]], [[1, 0], [0, 0]]])
620: (8)   a = masked_array(np.arange(8).reshape(2, 2, 2),
621: (25)           mask=[[1, 0], [0, 0], [[0, 0], [0, 0]]])
622: (8)   b = 5.
623: (8)   c = dot(a, b, strict=True)
624: (8)   assert_equal(c.mask, [[1, 0], [0, 0], [[0, 0], [0, 0]]])
625: (8)   c = dot(a, b, strict=False)
626: (8)   assert_equal(c.mask, [[1, 0], [0, 0], [[0, 0], [0, 0]]])
627: (8)   c = dot(b, a, strict=True)
628: (8)   assert_equal(c.mask, [[1, 0], [0, 0], [[0, 0], [0, 0]]])
629: (8)   c = dot(b, a, strict=False)
630: (8)   assert_equal(c.mask, [[1, 0], [0, 0], [[0, 0], [0, 0]]])
631: (8)   a = masked_array(np.arange(8).reshape(2, 2, 2),
632: (25)           mask=[[1, 0], [0, 0], [[0, 0], [0, 0]]])
633: (8)   b = masked_array(np.arange(2), mask=[0, 1])
634: (8)   c = dot(a, b, strict=True)
635: (8)   assert_equal(c.mask, [[1, 1], [1, 1]])
636: (8)   c = dot(a, b, strict=False)
637: (8)   assert_equal(c.mask, [[1, 0], [0, 0]])
638: (4) def test_dot_returns_maskedarray(self):
639: (8)   a = np.eye(3)
640: (8)   b = array(a)
641: (8)   assert_(type(dot(a, a)) is MaskedArray)
642: (8)   assert_(type(dot(a, b)) is MaskedArray)
643: (8)   assert_(type(dot(b, a)) is MaskedArray)
644: (8)   assert_(type(dot(b, b)) is MaskedArray)
645: (4) def test_dot_out(self):
646: (8)   a = array(np.eye(3))
647: (8)   out = array(np.zeros((3, 3)))
648: (8)   res = dot(a, a, out=out)
649: (8)   assert_(res is out)
650: (8)   assert_equal(a, res)
651: (0) class TestApplyAlongAxis:
652: (4)     def test_3d(self):
653: (8)       a = arange(12.).reshape(2, 2, 3)
654: (8)       def myfunc(b):
655: (12)           return b[1]
656: (8)       xa = apply_along_axis(myfunc, 2, a)
657: (8)       assert_equal(xa, [[1, 4], [7, 10]])
658: (4)     def test_3d_kwargs(self):
659: (8)       a = arange(12).reshape(2, 2, 3)
660: (8)       def myfunc(b, offset=0):
661: (12)           return b[1+offset]
662: (8)       xa = apply_along_axis(myfunc, 2, a, offset=1)
663: (8)       assert_equal(xa, [[2, 5], [8, 11]])
664: (0) class TestApplyOverAxes:
665: (4)     def test_basic(self):
666: (8)       a = arange(24).reshape(2, 3, 4)
667: (8)       test = apply_over_axes(np.sum, a, [0, 2])
668: (8)       ctrl = np.array([[60], [92], [124]])
669: (8)       assert_equal(test, ctrl)
670: (8)       a[(a % 2).astype(bool)] = masked
671: (8)       test = apply_over_axes(np.sum, a, [0, 2])
672: (8)       ctrl = np.array([[28], [44], [60]])
673: (8)       assert_equal(test, ctrl)
674: (0) class TestMedian:
675: (4)     def test_pytype(self):

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

676: (8)           r = np.ma.median([[np.inf, np.inf], [np.inf, np.inf]], axis=-1)
677: (8)           assert_equal(r, np.inf)
678: (4)           def test_inf(self):
679: (8)             r = np.ma.median(np.ma.masked_array([[np.inf, np.inf],
680: (45)                           [np.inf, np.inf]]], axis=-1)
681: (8)             assert_equal(r, np.inf)
682: (8)             r = np.ma.median(np.ma.masked_array([[np.inf, np.inf],
683: (45)                           [np.inf, np.inf]]], axis=None)
684: (8)             assert_equal(r, np.inf)
685: (8)             r = np.ma.median(np.ma.masked_array([[np.inf, np.inf],
686: (45)                           [np.inf, np.inf]]], mask=True),
687: (25)                         axis=-1)
688: (8)             assert_equal(r.mask, True)
689: (8)             r = np.ma.median(np.ma.masked_array([[np.inf, np.inf],
690: (45)                           [np.inf, np.inf]]], mask=True),
691: (25)                         axis=None)
692: (8)             assert_equal(r.mask, True)
693: (4)           def test_non_masked(self):
694: (8)             x = np.arange(9)
695: (8)             assert_equal(np.ma.median(x), 4.)
696: (8)             assert_(type(np.ma.median(x)) is not MaskedArray)
697: (8)             x = range(8)
698: (8)             assert_equal(np.ma.median(x), 3.5)
699: (8)             assert_(type(np.ma.median(x)) is not MaskedArray)
700: (8)             x = 5
701: (8)             assert_equal(np.ma.median(x), 5.)
702: (8)             assert_(type(np.ma.median(x)) is not MaskedArray)
703: (8)             x = np.arange(9 * 8).reshape(9, 8)
704: (8)             assert_equal(np.ma.median(x, axis=0), np.median(x, axis=0))
705: (8)             assert_equal(np.ma.median(x, axis=1), np.median(x, axis=1))
706: (8)             assert_(np.ma.median(x, axis=1) is not MaskedArray)
707: (8)             x = np.arange(9 * 8.).reshape(9, 8)
708: (8)             assert_equal(np.ma.median(x, axis=0), np.median(x, axis=0))
709: (8)             assert_equal(np.ma.median(x, axis=1), np.median(x, axis=1))
710: (8)             assert_(np.ma.median(x, axis=1) is not MaskedArray)
711: (4)           def test_docstring_examples(self):
712: (8)             "test the examples given in the docstring of ma.median"
713: (8)             x = array(np.arange(8), mask=[0]*4 + [1]*4)
714: (8)             assert_equal(np.ma.median(x), 1.5)
715: (8)             assert_equal(np.ma.median(x).shape, (), "shape mismatch")
716: (8)             assert_(type(np.ma.median(x)) is not MaskedArray)
717: (8)             x = array(np.arange(10).reshape(2, 5), mask=[0]*6 + [1]*4)
718: (8)             assert_equal(np.ma.median(x), 2.5)
719: (8)             assert_equal(np.ma.median(x).shape, (), "shape mismatch")
720: (8)             assert_(type(np.ma.median(x)) is not MaskedArray)
721: (8)             ma_x = np.ma.median(x, axis=-1, overwrite_input=True)
722: (8)             assert_equal(ma_x, [2., 5.])
723: (8)             assert_equal(ma_x.shape, (2,), "shape mismatch")
724: (8)             assert_(type(ma_x) is MaskedArray)
725: (4)           def test_axis_argument_errors(self):
726: (8)             msg = "mask = %s, ndim = %s, axis = %s, overwrite_input = %s"
727: (8)             for ndmin in range(5):
728: (12)               for mask in [False, True]:
729: (16)                 x = array(1, ndmin=ndmin, mask=mask)
730: (16)                 args = itertools.product(range(-ndmin, ndmin), [False, True])
731: (16)                 for axis, over in args:
732: (20)                   try:
733: (24)                     np.ma.median(x, axis=axis, overwrite_input=over)
734: (20)                   except Exception:
735: (24)                     raise AssertionError(msg % (mask, ndmin, axis, over))
736: (16)                 args = itertools.product([- (ndmin + 1), ndmin], [False, True])
737: (16)                 for axis, over in args:
738: (20)                   try:
739: (24)                     np.ma.median(x, axis=axis, overwrite_input=over)
740: (20)                   except np.AxisError:
741: (24)                     pass
742: (20)                   else:
743: (24)                     raise AssertionError(msg % (mask, ndmin, axis, over))
744: (4)           def test_masked_0d(self):

```

```

745: (8)             x = array(1, mask=False)
746: (8)             assert_equal(np.ma.median(x), 1)
747: (8)             x = array(1, mask=True)
748: (8)             assert_equal(np.ma.median(x), np.ma.masked)
749: (4)             def test_masked_1d(self):
750: (8)                 x = array(np.arange(5), mask=True)
751: (8)                 assert_equal(np.ma.median(x), np.ma.masked)
752: (8)                 assert_equal(np.ma.median(x).shape, (), "shape mismatch")
753: (8)                 assert_(type(np.ma.median(x)) is np.ma.core.MaskedConstant)
754: (8)                 x = array(np.arange(5), mask=False)
755: (8)                 assert_equal(np.ma.median(x), 2.)
756: (8)                 assert_equal(np.ma.median(x).shape, (), "shape mismatch")
757: (8)                 assert_(type(np.ma.median(x)) is not MaskedArray)
758: (8)                 x = array(np.arange(5), mask=[0,1,0,0,0])
759: (8)                 assert_equal(np.ma.median(x), 2.5)
760: (8)                 assert_equal(np.ma.median(x).shape, (), "shape mismatch")
761: (8)                 assert_(type(np.ma.median(x)) is not MaskedArray)
762: (8)                 x = array(np.arange(5), mask=[0,1,1,1,1])
763: (8)                 assert_equal(np.ma.median(x), 0.)
764: (8)                 assert_equal(np.ma.median(x).shape, (), "shape mismatch")
765: (8)                 assert_(type(np.ma.median(x)) is not MaskedArray)
766: (8)                 x = array(np.arange(5), mask=[0,1,1,0,0])
767: (8)                 assert_equal(np.ma.median(x), 3.)
768: (8)                 assert_equal(np.ma.median(x).shape, (), "shape mismatch")
769: (8)                 assert_(type(np.ma.median(x)) is not MaskedArray)
770: (8)                 x = array(np.arange(5.), mask=[0,1,1,0,0])
771: (8)                 assert_equal(np.ma.median(x), 3.)
772: (8)                 assert_equal(np.ma.median(x).shape, (), "shape mismatch")
773: (8)                 assert_(type(np.ma.median(x)) is not MaskedArray)
774: (8)                 x = array(np.arange(6), mask=[0,1,1,1,1,0])
775: (8)                 assert_equal(np.ma.median(x), 2.5)
776: (8)                 assert_equal(np.ma.median(x).shape, (), "shape mismatch")
777: (8)                 assert_(type(np.ma.median(x)) is not MaskedArray)
778: (8)                 x = array(np.arange(6.), mask=[0,1,1,1,1,0])
779: (8)                 assert_equal(np.ma.median(x), 2.5)
780: (8)                 assert_equal(np.ma.median(x).shape, (), "shape mismatch")
781: (8)                 assert_(type(np.ma.median(x)) is not MaskedArray)
782: (4)                 def test_1d_shape_consistency(self):
783: (8)                     assert_equal(np.ma.median(array([1,2,3],mask=[0,0,0])).shape,
784: (21)                               np.ma.median(array([1,2,3],mask=[0,1,0])).shape )
785: (4)                 def test_2d(self):
786: (8)                     (n, p) = (101, 30)
787: (8)                     x = masked_array(np.linspace(-1., 1., n,))
788: (8)                     x[:10] = x[-10:] = masked
789: (8)                     z = masked_array(np.empty((n, p), dtype=float))
790: (8)                     z[:, 0] = x[:, :]
791: (8)                     idx = np.arange(len(x))
792: (8)                     for i in range(1, p):
793: (12)                         np.random.shuffle(idx)
794: (12)                         z[:, i] = x[idx]
795: (8)                         assert_equal(median(z[:, 0]), 0)
796: (8)                         assert_equal(median(z), 0)
797: (8)                         assert_equal(median(z, axis=0), np.zeros(p))
798: (8)                         assert_equal(median(z.T, axis=1), np.zeros(p))
799: (4)                 def test_2d_waxis(self):
800: (8)                     x = masked_array(np.arange(30).reshape(10, 3))
801: (8)                     x[:3] = x[-3:] = masked
802: (8)                     assert_equal(median(x), 14.5)
803: (8)                     assert_(type(np.ma.median(x)) is not MaskedArray)
804: (8)                     assert_equal(median(x, axis=0), [13.5, 14.5, 15.5])
805: (8)                     assert_(type(np.ma.median(x, axis=0)) is MaskedArray)
806: (8)                     assert_equal(median(x, axis=1), [0, 0, 0, 10, 13, 16, 19, 0, 0, 0])
807: (8)                     assert_(type(np.ma.median(x, axis=1)) is MaskedArray)
808: (8)                     assert_equal(median(x, axis=1).mask, [1, 1, 1, 0, 0, 0, 0, 1, 1, 1])
809: (4)                 def test_3d(self):
810: (8)                     x = np.ma.arange(24).reshape(3, 4, 2)
811: (8)                     x[x % 3 == 0] = masked
812: (8)                     assert_equal(median(x, 0), [[12, 9], [6, 15], [12, 9], [18, 15]])
813: (8)                     x.shape = (4, 3, 2)

```

```

814: (8)             assert_equal(median(x, 0), [[99, 10], [11, 99], [13, 14]])
815: (8)             x = np.ma.arange(24).reshape(4, 3, 2)
816: (8)             x[x % 5 == 0] = masked
817: (8)             assert_equal(median(x, 0), [[12, 10], [8, 9], [16, 17]])
818: (4)             def test_neg_axis(self):
819: (8)                 x = masked_array(np.arange(30).reshape(10, 3))
820: (8)                 x[:3] = x[-3:] = masked
821: (8)                 assert_equal(median(x, axis=-1), median(x, axis=1))
822: (4)             def test_out_1d(self):
823: (8)                 for v in (30, 30., 31, 31.):
824: (12)                     x = masked_array(np.arange(v))
825: (12)                     x[:3] = x[-3:] = masked
826: (12)                     out = masked_array(np.ones(()))
827: (12)                     r = median(x, out=out)
828: (12)                     if v == 30:
829: (16)                         assert_equal(out, 14.5)
830: (12)                     else:
831: (16)                         assert_equal(out, 15.)
832: (12)                     assert_(r is out)
833: (12)                     assert_(type(r) is MaskedArray)
834: (4)             def test_out(self):
835: (8)                 for v in (40, 40., 30, 30.):
836: (12)                     x = masked_array(np.arange(v).reshape(10, -1))
837: (12)                     x[:3] = x[-3:] = masked
838: (12)                     out = masked_array(np.ones(10))
839: (12)                     r = median(x, axis=1, out=out)
840: (12)                     if v == 30:
841: (16)                         e = masked_array([0.]*3 + [10, 13, 16, 19] + [0.]*3,
842: (33)                             mask=[True] * 3 + [False] * 4 + [True] * 3)
843: (12)                     else:
844: (16)                         e = masked_array([0.]*3 + [13.5, 17.5, 21.5, 25.5] + [0.]*3,
845: (33)                             mask=[True]*3 + [False]*4 + [True]*3)
846: (12)                     assert_equal(r, e)
847: (12)                     assert_(r is out)
848: (12)                     assert_(type(r) is MaskedArray)
849: (4)             @pytest.mark.parametrize(
850: (8)                 argnames='axis',
851: (8)                 argvalues=[
852: (12)                     None,
853: (12)                     1,
854: (12)                     (1, ),
855: (12)                     (0, 1),
856: (12)                     (-3, -1),
857: (8)                 ]
858: (4)             )
859: (4)             def test_kepdims_out(self, axis):
860: (8)                 mask = np.zeros((3, 5, 7, 11), dtype=bool)
861: (8)                 w = np.random.random((4, 200)) * np.array(mask.shape)[:, None]
862: (8)                 w = w.astype(np.intp)
863: (8)                 mask[tuple(w)] = np.nan
864: (8)                 d = masked_array(np.ones(mask.shape), mask=mask)
865: (8)                 if axis is None:
866: (12)                     shape_out = (1,) * d.ndim
867: (8)                 else:
868: (12)                     axis_norm = normalize_axis_tuple(axis, d.ndim)
869: (12)                     shape_out = tuple(
870: (16)                         1 if i in axis_norm else d.shape[i] for i in range(d.ndim))
871: (8)                     out = masked_array(np.empty(shape_out))
872: (8)                     result = median(d, axis=axis, keepdims=True, out=out)
873: (8)                     assert result is out
874: (8)                     assert_equal(result.shape, shape_out)
875: (4)             def test_single_non_masked_value_on_axis(self):
876: (8)                 data = [[1., 0.],
877: (16)                     [0., 3.],
878: (16)                     [0., 0.]]
879: (8)                 masked_arr = np.ma.masked_equal(data, 0)
880: (8)                 expected = [1., 3.]
881: (8)                 assert_array_equal(np.ma.median(masked_arr, axis=0),
882: (27)                               expected)

```

```

883: (4)
884: (8)
885: (12)
886: (12)
887: (12)
888: (12)
889: (12)
890: (12)
891: (12)
892: (12)
893: (12)
894: (12)
895: (12)
896: (12)
897: (12)
898: (12)
899: (12)
900: (12)
901: (12)
902: (8)
903: (8)
904: (8)
905: (8)
906: (8)
907: (4)
908: (8)
909: (8)
910: (8)
911: (8)
912: (8)
913: (8)
914: (8)
915: (8)
916: (8)
917: (4)
918: (8)
919: (8)
920: (8)
921: (8)
922: (8)
923: (8)
924: (8)
925: (8)
926: (8)
927: (8)
928: (8)
929: (8)
930: (8)
931: (8)
932: (8)
933: (8)
934: (8)
935: (8)
936: (8)
937: (8)
938: (8)
939: (8)
940: (8)
941: (8)
942: (4)
943: (8)
944: (8)
945: (8)
946: (8)
947: (8)
948: (8)
949: (4)
950: (8)
951: (12)

def test_nan(self):
    for mask in (False, np.zeros(6, dtype=bool)):
        dm = np.ma.array([[1, np.nan, 3], [1, 2, 3]])
        dm.mask = mask
        r = np.ma.median(dm, axis=None)
        assert_(np.isscalar(r))
        assert_array_equal(r, np.nan)
        r = np.ma.median(dm.ravel(), axis=0)
        assert_(np.isscalar(r))
        assert_array_equal(r, np.nan)
        r = np.ma.median(dm, axis=0)
        assert_equal(type(r), MaskedArray)
        assert_array_equal(r, [1, np.nan, 3])
        r = np.ma.median(dm, axis=1)
        assert_equal(type(r), MaskedArray)
        assert_array_equal(r, [np.nan, 2])
        r = np.ma.median(dm, axis=-1)
        assert_equal(type(r), MaskedArray)
        assert_array_equal(r, [np.nan, 2])
        dm = np.ma.array([[1, np.nan, 3], [1, 2, 3]])
        dm[:, 2] = np.ma.masked
        assert_array_equal(np.ma.median(dm, axis=None), np.nan)
        assert_array_equal(np.ma.median(dm, axis=0), [1, np.nan, 3])
        assert_array_equal(np.ma.median(dm, axis=1), [np.nan, 1.5])
def test_out_nan(self):
    o = np.ma.masked_array(np.zeros((4,)))
    d = np.ma.masked_array(np.ones((3, 4)))
    d[2, 1] = np.nan
    d[2, 2] = np.ma.masked
    assert_equal(np.ma.median(d, 0, out=o), o)
    o = np.ma.masked_array(np.zeros((3,)))
    assert_equal(np.ma.median(d, 1, out=o), o)
    o = np.ma.masked_array(np.zeros(()))
    assert_equal(np.ma.median(d, out=o), o)
def test_nan_behavior(self):
    a = np.ma.masked_array(np.arange(24, dtype=float))
    a[::3] = np.ma.masked
    a[2] = np.nan
    assert_array_equal(np.ma.median(a), np.nan)
    assert_array_equal(np.ma.median(a, axis=0), np.nan)
    a = np.ma.masked_array(np.arange(24, dtype=float).reshape(2, 3, 4))
    a.mask = np.arange(a.size) % 2 == 1
    aorig = a.copy()
    a[1, 2, 3] = np.nan
    a[1, 1, 2] = np.nan
    assert_array_equal(np.ma.median(a), np.nan)
    assert_(np.isscalar(np.ma.median(a)))
    b = np.ma.median(aorig, axis=0)
    b[2, 3] = np.nan
    b[1, 2] = np.nan
    assert_equal(np.ma.median(a, 0), b)
    b = np.ma.median(aorig, axis=1)
    b[1, 3] = np.nan
    b[1, 2] = np.nan
    assert_equal(np.ma.median(a, 1), b)
    b = np.ma.median(aorig, axis=(0, 2))
    b[1] = np.nan
    b[2] = np.nan
    assert_equal(np.ma.median(a, (0, 2)), b)
def test_ambiguous_fill(self):
    a = np.array([[3, 3, 255], [3, 3, 255]], dtype=np.uint8)
    a = np.ma.masked_array(a, mask=a == 3)
    assert_array_equal(np.ma.median(a, axis=1), 255)
    assert_array_equal(np.ma.median(a, axis=1).mask, False)
    assert_array_equal(np.ma.median(a, axis=0), a[0])
    assert_array_equal(np.ma.median(a), 255)
def test_special(self):
    for inf in [np.inf, -np.inf]:
        a = np.array([[inf, np.nan], [np.nan, np.nan]])

```

```

952: (12)             a = np.ma.masked_array(a, mask=np.isnan(a))
953: (12)             assert_equal(np.ma.median(a, axis=0), [inf, np.nan])
954: (12)             assert_equal(np.ma.median(a, axis=1), [inf, np.nan])
955: (12)             assert_equal(np.ma.median(a), inf)
956: (12)             a = np.array([[np.nan, np.nan, inf], [np.nan, np.nan, inf]])
957: (12)             a = np.ma.masked_array(a, mask=np.isnan(a))
958: (12)             assert_array_equal(np.ma.median(a, axis=1), inf)
959: (12)             assert_array_equal(np.ma.median(a, axis=1).mask, False)
960: (12)             assert_array_equal(np.ma.median(a, axis=0), a[0])
961: (12)             assert_array_equal(np.ma.median(a), inf)
962: (12)             a = np.array([[inf, inf], [inf, inf]])
963: (12)             assert_equal(np.ma.median(a), inf)
964: (12)             assert_equal(np.ma.median(a, axis=0), inf)
965: (12)             assert_equal(np.ma.median(a, axis=1), inf)
966: (12)             a = np.array([[inf, 7, -inf, -9],
967: (26)                         [-10, np.nan, np.nan, 5],
968: (26)                         [4, np.nan, np.nan, inf]],
969: (26)                         dtype=np.float32)
970: (12)             a = np.ma.masked_array(a, mask=np.isnan(a))
971: (12)             if inf > 0:
972: (16)                 assert_equal(np.ma.median(a, axis=0), [4., 7., -inf, 5.])
973: (16)                 assert_equal(np.ma.median(a), 4.5)
974: (12)             else:
975: (16)                 assert_equal(np.ma.median(a, axis=0), [-10., 7., -inf, -9.])
976: (16)                 assert_equal(np.ma.median(a), -2.5)
977: (12)             assert_equal(np.ma.median(a, axis=1), [-1., -2.5, inf])
978: (12)             for i in range(0, 10):
979: (16)                 for j in range(1, 10):
980: (20)                     a = np.array([[([np.nan] * i) + ([inf] * j)] * 2)
981: (20)                     a = np.ma.masked_array(a, mask=np.isnan(a))
982: (20)                     assert_equal(np.ma.median(a), inf)
983: (20)                     assert_equal(np.ma.median(a, axis=1), inf)
984: (20)                     assert_equal(np.ma.median(a, axis=0),
985: (33)                         ([np.nan] * i) + [inf] * j)
986: (4)             def test_empty(self):
987: (8)                 a = np.ma.masked_array(np.array([], dtype=float))
988: (8)                 with suppress_warnings() as w:
989: (12)                     w.record(RuntimeWarning)
990: (12)                     assert_array_equal(np.ma.median(a), np.nan)
991: (12)                     assert_(w.log[0].category is RuntimeWarning)
992: (8)                 a = np.ma.masked_array(np.array([], dtype=float, ndmin=3))
993: (8)                 with suppress_warnings() as w:
994: (12)                     w.record(RuntimeWarning)
995: (12)                     warnings.filterwarnings('always', '', RuntimeWarning)
996: (12)                     assert_array_equal(np.ma.median(a), np.nan)
997: (12)                     assert_(w.log[0].category is RuntimeWarning)
998: (8)                 b = np.ma.masked_array(np.array([], dtype=float, ndmin=2))
999: (8)                 assert_equal(np.ma.median(a, axis=0), b)
1000: (8)                 assert_equal(np.ma.median(a, axis=1), b)
1001: (8)                 b = np.ma.masked_array(np.array(np.nan, dtype=float, ndmin=2))
1002: (8)                 with warnings.catch_warnings(record=True) as w:
1003: (12)                     warnings.filterwarnings('always', '', RuntimeWarning)
1004: (12)                     assert_equal(np.ma.median(a, axis=2), b)
1005: (12)                     assert_(w[0].category is RuntimeWarning)
1006: (4)             def test_object(self):
1007: (8)                 o = np.ma.masked_array(np.arange(7.))
1008: (8)                 assert_(type(np.ma.median(o.astype(object))), float)
1009: (8)                 o[2] = np.nan
1010: (8)                 assert_(type(np.ma.median(o.astype(object))), float)
1011: (0)             class TestCov:
1012: (4)                 def setup_method(self):
1013: (8)                     self.data = array(np.random.rand(12))
1014: (4)                 def test_1d_without_missing(self):
1015: (8)                     x = self.data
1016: (8)                     assert_almost_equal(np.cov(x), cov(x))
1017: (8)                     assert_almost_equal(np.cov(x, rowvar=False), cov(x, rowvar=False))
1018: (8)                     assert_almost_equal(np.cov(x, rowvar=False, bias=True),
1019: (28)                         cov(x, rowvar=False, bias=True))
1020: (4)                 def test_2d_without_missing(self):

```

```

1021: (8)           x = self.data.reshape(3, 4)
1022: (8)           assert_almost_equal(np.cov(x), cov(x))
1023: (8)           assert_almost_equal(np.cov(x, rowvar=False), cov(x, rowvar=False))
1024: (8)           assert_almost_equal(np.cov(x, rowvar=False, bias=True),
1025:                               cov(x, rowvar=False, bias=True))
1026: (4)           def test_1d_with_missing(self):
1027: (8)             x = self.data
1028: (8)             x[-1] = masked
1029: (8)             x -= x.mean()
1030: (8)             nx = x.compressed()
1031: (8)             assert_almost_equal(np.cov(nx), cov(x))
1032: (8)             assert_almost_equal(np.cov(nx, rowvar=False), cov(x, rowvar=False))
1033: (8)             assert_almost_equal(np.cov(nx, rowvar=False, bias=True),
1034:                               cov(x, rowvar=False, bias=True))
1035: (8)             try:
1036: (12)               cov(x, allow_masked=False)
1037: (8)             except ValueError:
1038: (12)               pass
1039: (8)             nx = x[1:-1]
1040: (8)             assert_almost_equal(np.cov(nx, nx[::-1]), cov(x, x[::-1]))
1041: (8)             assert_almost_equal(np.cov(nx, nx[::-1], rowvar=False),
1042:                               cov(x, x[::-1], rowvar=False))
1043: (8)             assert_almost_equal(np.cov(nx, nx[::-1], rowvar=False, bias=True),
1044:                               cov(x, x[::-1], rowvar=False, bias=True))
1045: (4)           def test_2d_with_missing(self):
1046: (8)             x = self.data
1047: (8)             x[-1] = masked
1048: (8)             x = x.reshape(3, 4)
1049: (8)             valid = np.logical_not(getmaskarray(x)).astype(int)
1050: (8)             frac = np.dot(valid, valid.T)
1051: (8)             xf = (x - x.mean(1)[:, None]).filled(0)
1052: (8)             assert_almost_equal(cov(x),
1053:                               np.cov(xf) * (x.shape[1] - 1) / (frac - 1.))
1054: (8)             assert_almost_equal(cov(x, bias=True),
1055:                               np.cov(xf, bias=True) * x.shape[1] / frac)
1056: (8)             frac = np.dot(valid.T, valid)
1057: (8)             xf = (x - x.mean(0)).filled(0)
1058: (8)             assert_almost_equal(cov(x, rowvar=False),
1059:                               (np.cov(xf, rowvar=False) *
1060:                                (x.shape[0] - 1) / (frac - 1.)))
1061: (8)             assert_almost_equal(cov(x, rowvar=False, bias=True),
1062:                               (np.cov(xf, rowvar=False, bias=True) *
1063:                                x.shape[0] / frac))
1064: (0)           class TestCorrcoef:
1065: (4)             def setup_method(self):
1066: (8)               self.data = array(np.random.rand(12))
1067: (8)               self.data2 = array(np.random.rand(12))
1068: (4)             def test_ddof(self):
1069: (8)               x, y = self.data, self.data2
1070: (8)               expected = np.corrcoef(x)
1071: (8)               expected2 = np.corrcoef(x, y)
1072: (8)               with suppress_warnings() as sup:
1073: (12)                 warnings.simplefilter("always")
1074: (12)                 assert_warns(DeprecationWarning, corrcoef, x, ddof=-1)
1075: (12)                 sup.filter(DeprecationWarning, "bias and ddof have no effect")
1076: (12)                 assert_almost_equal(np.corrcoef(x, ddof=0), corrcoef(x, ddof=0))
1077: (12)                 assert_almost_equal(corrcoef(x, ddof=-1), expected)
1078: (12)                 assert_almost_equal(corrcoef(x, y, ddof=-1), expected2)
1079: (12)                 assert_almost_equal(corrcoef(x, ddof=3), expected)
1080: (12)                 assert_almost_equal(corrcoef(x, y, ddof=3), expected2)
1081: (4)             def test_bias(self):
1082: (8)               x, y = self.data, self.data2
1083: (8)               expected = np.corrcoef(x)
1084: (8)               with suppress_warnings() as sup:
1085: (12)                 warnings.simplefilter("always")
1086: (12)                 assert_warns(DeprecationWarning, corrcoef, x, y, True, False)
1087: (12)                 assert_warns(DeprecationWarning, corrcoef, x, y, True, True)
1088: (12)                 assert_warns(DeprecationWarning, corrcoef, x, bias=False)
1089: (12)                 sup.filter(DeprecationWarning, "bias and ddof have no effect")

```

```

1090: (12)             assert_almost_equal(corrcoef(x, bias=1), expected)
1091: (4) def test_1d_without_missing(self):
1092: (8)     x = self.data
1093: (8)     assert_almost_equal(np.corrcoef(x), corrcoef(x))
1094: (8)     assert_almost_equal(np.corrcoef(x, rowvar=False),
1095: (28)                 corrcoef(x, rowvar=False))
1096: (8)     with suppress_warnings() as sup:
1097: (12)         sup.filter(DeprecationWarning, "bias and ddof have no effect")
1098: (12)         assert_almost_equal(np.corrcoef(x, rowvar=False, bias=True),
1099: (32)                 corrcoef(x, rowvar=False, bias=True))
1100: (4) def test_2d_without_missing(self):
1101: (8)     x = self.data.reshape(3, 4)
1102: (8)     assert_almost_equal(np.corrcoef(x), corrcoef(x))
1103: (8)     assert_almost_equal(np.corrcoef(x, rowvar=False),
1104: (28)                 corrcoef(x, rowvar=False))
1105: (8)     with suppress_warnings() as sup:
1106: (12)         sup.filter(DeprecationWarning, "bias and ddof have no effect")
1107: (12)         assert_almost_equal(np.corrcoef(x, rowvar=False, bias=True),
1108: (32)                 corrcoef(x, rowvar=False, bias=True))
1109: (4) def test_1d_with_missing(self):
1110: (8)     x = self.data
1111: (8)     x[-1] = masked
1112: (8)     x -= x.mean()
1113: (8)     nx = x.compressed()
1114: (8)     assert_almost_equal(np.corrcoef(nx), corrcoef(x))
1115: (8)     assert_almost_equal(np.corrcoef(nx, rowvar=False),
1116: (28)                 corrcoef(x, rowvar=False))
1117: (8)     with suppress_warnings() as sup:
1118: (12)         sup.filter(DeprecationWarning, "bias and ddof have no effect")
1119: (12)         assert_almost_equal(np.corrcoef(nx, rowvar=False, bias=True),
1120: (32)                 corrcoef(x, rowvar=False, bias=True))
1121: (8) try:
1122: (12)     corrcoef(x, allow_masked=False)
1123: (8) except ValueError:
1124: (12)     pass
1125: (8)     nx = x[1:-1]
1126: (8)     assert_almost_equal(np.corrcoef(nx, nx[::-1]), corrcoef(x, x[::-1]))
1127: (8)     assert_almost_equal(np.corrcoef(nx, nx[::-1], rowvar=False),
1128: (28)                 corrcoef(x, x[::-1], rowvar=False))
1129: (8)     with suppress_warnings() as sup:
1130: (12)         sup.filter(DeprecationWarning, "bias and ddof have no effect")
1131: (12)         assert_almost_equal(np.corrcoef(nx, nx[::-1]),
1132: (32)                 corrcoef(x, x[::-1], bias=1))
1133: (12)         assert_almost_equal(np.corrcoef(nx, nx[::-1]),
1134: (32)                 corrcoef(x, x[::-1], ddof=2))
1135: (4) def test_2d_with_missing(self):
1136: (8)     x = self.data
1137: (8)     x[-1] = masked
1138: (8)     x = x.reshape(3, 4)
1139: (8)     test = corrcoef(x)
1140: (8)     control = np.corrcoef(x)
1141: (8)     assert_almost_equal(test[:-1, :-1], control[:-1, :-1])
1142: (8)     with suppress_warnings() as sup:
1143: (12)         sup.filter(DeprecationWarning, "bias and ddof have no effect")
1144: (12)         assert_almost_equal(corrcoef(x, ddof=-2)[:-1, :-1],
1145: (32)                 control[:-1, :-1])
1146: (12)         assert_almost_equal(corrcoef(x, ddof=3)[:-1, :-1],
1147: (32)                 control[:-1, :-1])
1148: (12)         assert_almost_equal(corrcoef(x, bias=1)[:-1, :-1],
1149: (32)                 control[:-1, :-1])
1150: (0) class TestPolynomial:
1151: (4)     def test_polyfit(self):
1152: (8)         x = np.random.rand(10)
1153: (8)         y = np.random.rand(20).reshape(-1, 2)
1154: (8)         assert_almost_equal(polyfit(x, y, 3), np.polyfit(x, y, 3))
1155: (8)         x = x.view(MaskedArray)
1156: (8)         x[0] = masked
1157: (8)         y = y.view(MaskedArray)
1158: (8)         y[0, 0] = y[-1, -1] = masked

```

```

1159: (8)             (C, R, K, S, D) = polyfit(x, y[:, 0], 3, full=True)
1160: (8)             (c, r, k, s, d) = np.polyfit(x[1:], y[1:, 0].compressed(), 3,
1161: (37)                         full=True)
1162: (8)             for (a, a_) in zip((C, R, K, S, D), (c, r, k, s, d)):
1163: (12)                 assert_almost_equal(a, a_)
1164: (8)             (C, R, K, S, D) = polyfit(x, y[:, -1], 3, full=True)
1165: (8)             (c, r, k, s, d) = np.polyfit(x[1:-1], y[1:-1, -1], 3, full=True)
1166: (8)             for (a, a_) in zip((C, R, K, S, D), (c, r, k, s, d)):
1167: (12)                 assert_almost_equal(a, a_)
1168: (8)             (C, R, K, S, D) = polyfit(x, y, 3, full=True)
1169: (8)             (c, r, k, s, d) = np.polyfit(x[1:-1], y[1:-1, :], 3, full=True)
1170: (8)             for (a, a_) in zip((C, R, K, S, D), (c, r, k, s, d)):
1171: (12)                 assert_almost_equal(a, a_)
1172: (8)             w = np.random.rand(10) + 1
1173: (8)             wo = w.copy()
1174: (8)             xs = x[1:-1]
1175: (8)             ys = y[1:-1]
1176: (8)             ws = w[1:-1]
1177: (8)             (C, R, K, S, D) = polyfit(x, y, 3, full=True, w=w)
1178: (8)             (c, r, k, s, d) = np.polyfit(xs, ys, 3, full=True, w=ws)
1179: (8)             assert_equal(w, wo)
1180: (8)             for (a, a_) in zip((C, R, K, S, D), (c, r, k, s, d)):
1181: (12)                 assert_almost_equal(a, a_)
1182: (4)             def test_polyfit_with_masked_NaN(self):
1183: (8)                 x = np.random.rand(10)
1184: (8)                 y = np.random.rand(20).reshape(-1, 2)
1185: (8)                 x[0] = np.nan
1186: (8)                 y[-1,-1] = np.nan
1187: (8)                 x = x.view(MaskedArray)
1188: (8)                 y = y.view(MaskedArray)
1189: (8)                 x[0] = masked
1190: (8)                 y[-1,-1] = masked
1191: (8)                 (C, R, K, S, D) = polyfit(x, y, 3, full=True)
1192: (8)                 (c, r, k, s, d) = np.polyfit(x[1:-1], y[1:-1, :], 3, full=True)
1193: (8)                 for (a, a_) in zip((C, R, K, S, D), (c, r, k, s, d)):
1194: (12)                     assert_almost_equal(a, a_)
1195: (0)             class TestArraySetOps:
1196: (4)                 def test_unique_onlist(self):
1197: (8)                     data = [1, 1, 1, 2, 2, 3]
1198: (8)                     test = unique(data, return_index=True, return_inverse=True)
1199: (8)                     assert_(isinstance(test[0], MaskedArray))
1200: (8)                     assert_equal(test[0], masked_array([1, 2, 3], mask=[0, 0, 0]))
1201: (8)                     assert_equal(test[1], [0, 3, 5])
1202: (8)                     assert_equal(test[2], [0, 0, 1, 1, 2])
1203: (4)                 def test_unique_onmaskedarray(self):
1204: (8)                     data = masked_array([1, 1, 1, 2, 2, 3], mask=[0, 0, 1, 0, 1, 0])
1205: (8)                     test = unique(data, return_index=True, return_inverse=True)
1206: (8)                     assert_equal(test[0], masked_array([1, 2, 3, -1], mask=[0, 0, 0, 1]))
1207: (8)                     assert_equal(test[1], [0, 3, 5, 2])
1208: (8)                     assert_equal(test[2], [0, 0, 1, 3, 2])
1209: (8)                     data.fill_value = 3
1210: (8)                     data = masked_array(data=[1, 1, 1, 2, 2, 3],
1211: (28)                                     mask=[0, 0, 1, 0, 1, 0], fill_value=3)
1212: (8)                     test = unique(data, return_index=True, return_inverse=True)
1213: (8)                     assert_equal(test[0], masked_array([1, 2, 3, -1], mask=[0, 0, 0, 1]))
1214: (8)                     assert_equal(test[1], [0, 3, 5, 2])
1215: (8)                     assert_equal(test[2], [0, 0, 1, 3, 2])
1216: (4)                 def test_unique_allmasked(self):
1217: (8)                     data = masked_array([1, 1, 1], mask=True)
1218: (8)                     test = unique(data, return_index=True, return_inverse=True)
1219: (8)                     assert_equal(test[0], masked_array([1, ], mask=[True]))
1220: (8)                     assert_equal(test[1], [0])
1221: (8)                     assert_equal(test[2], [0, 0, 0])
1222: (8)                     data = masked
1223: (8)                     test = unique(data, return_index=True, return_inverse=True)
1224: (8)                     assert_equal(test[0], masked_array(masked))
1225: (8)                     assert_equal(test[1], [0])
1226: (8)                     assert_equal(test[2], [0])
1227: (4)             def test_ediff1d(self):

```

```

1228: (8)           x = masked_array(np.arange(5), mask=[1, 0, 0, 0, 1])
1229: (8)           control = array([1, 1, 1, 4], mask=[1, 0, 0, 1])
1230: (8)           test = ediff1d(x)
1231: (8)           assert_equal(test, control)
1232: (8)           assert_equal(test.filled(0), control.filled(0))
1233: (8)           assert_equal(test.mask, control.mask)
1234: (4) def test_ediff1d_tobegin(self):
1235: (8)           x = masked_array(np.arange(5), mask=[1, 0, 0, 0, 1])
1236: (8)           test = ediff1d(x, to_begin=masked)
1237: (8)           control = array([0, 1, 1, 1, 4], mask=[1, 1, 0, 0, 1])
1238: (8)           assert_equal(test, control)
1239: (8)           assert_equal(test.filled(0), control.filled(0))
1240: (8)           assert_equal(test.mask, control.mask)
1241: (8)           test = ediff1d(x, to_begin=[1, 2, 3])
1242: (8)           control = array([1, 2, 3, 1, 1, 4], mask=[0, 0, 0, 1, 0, 0, 1])
1243: (8)           assert_equal(test, control)
1244: (8)           assert_equal(test.filled(0), control.filled(0))
1245: (8)           assert_equal(test.mask, control.mask)
1246: (4) def test_ediff1d_toend(self):
1247: (8)           x = masked_array(np.arange(5), mask=[1, 0, 0, 0, 1])
1248: (8)           test = ediff1d(x, to_end=masked)
1249: (8)           control = array([1, 1, 1, 4, 0], mask=[1, 0, 0, 1, 1])
1250: (8)           assert_equal(test, control)
1251: (8)           assert_equal(test.filled(0), control.filled(0))
1252: (8)           assert_equal(test.mask, control.mask)
1253: (8)           test = ediff1d(x, to_end=[1, 2, 3])
1254: (8)           control = array([1, 1, 1, 4, 1, 2, 3], mask=[1, 0, 0, 1, 0, 0, 0])
1255: (8)           assert_equal(test, control)
1256: (8)           assert_equal(test.filled(0), control.filled(0))
1257: (8)           assert_equal(test.mask, control.mask)
1258: (4) def test_ediff1d_tobegin_toend(self):
1259: (8)           x = masked_array(np.arange(5), mask=[1, 0, 0, 0, 1])
1260: (8)           test = ediff1d(x, to_end=masked, to_begin=masked)
1261: (8)           control = array([0, 1, 1, 1, 4, 0], mask=[1, 1, 0, 0, 1, 1])
1262: (8)           assert_equal(test, control)
1263: (8)           assert_equal(test.filled(0), control.filled(0))
1264: (8)           assert_equal(test.mask, control.mask)
1265: (8)           test = ediff1d(x, to_end=[1, 2, 3], to_begin=masked)
1266: (8)           control = array([0, 1, 1, 1, 4, 1, 2, 3],
1267: (24)                           mask=[1, 1, 0, 0, 1, 0, 0, 0])
1268: (8)           assert_equal(test, control)
1269: (8)           assert_equal(test.filled(0), control.filled(0))
1270: (8)           assert_equal(test.mask, control.mask)
1271: (4) def test_ediff1d_ndarray(self):
1272: (8)           x = np.arange(5)
1273: (8)           test = ediff1d(x)
1274: (8)           control = array([1, 1, 1, 1], mask=[0, 0, 0, 0])
1275: (8)           assert_equal(test, control)
1276: (8)           assert_(isinstance(test, MaskedArray))
1277: (8)           assert_equal(test.filled(0), control.filled(0))
1278: (8)           assert_equal(test.mask, control.mask)
1279: (8)           test = ediff1d(x, to_end=masked, to_begin=masked)
1280: (8)           control = array([0, 1, 1, 1, 1, 0], mask=[1, 0, 0, 0, 0, 1])
1281: (8)           assert_(isinstance(test, MaskedArray))
1282: (8)           assert_equal(test.filled(0), control.filled(0))
1283: (8)           assert_equal(test.mask, control.mask)
1284: (4) def test_intersect1d(self):
1285: (8)           x = array([1, 3, 3, 3], mask=[0, 0, 0, 1])
1286: (8)           y = array([3, 1, 1, 1], mask=[0, 0, 0, 1])
1287: (8)           test = intersect1d(x, y)
1288: (8)           control = array([1, 3, -1], mask=[0, 0, 1])
1289: (8)           assert_equal(test, control)
1290: (4) def test_setxor1d(self):
1291: (8)           a = array([1, 2, 5, 7, -1], mask=[0, 0, 0, 0, 1])
1292: (8)           b = array([1, 2, 3, 4, 5, -1], mask=[0, 0, 0, 0, 0, 1])
1293: (8)           test = setxor1d(a, b)
1294: (8)           assert_equal(test, array([3, 4, 7]))
1295: (8)           a = array([1, 2, 5, 7, -1], mask=[0, 0, 0, 0, 1])
1296: (8)           b = [1, 2, 3, 4, 5]

```

```

1297: (8)             test = setxor1d(a, b)
1298: (8)             assert_equal(test, array([3, 4, 7, -1], mask=[0, 0, 0, 1]))
1299: (8)             a = array([1, 2, 3])
1300: (8)             b = array([6, 5, 4])
1301: (8)             test = setxor1d(a, b)
1302: (8)             assert_(isinstance(test, MaskedArray))
1303: (8)             assert_equal(test, [1, 2, 3, 4, 5, 6])
1304: (8)             a = array([1, 8, 2, 3], mask=[0, 1, 0, 0])
1305: (8)             b = array([6, 5, 4, 8], mask=[0, 0, 0, 1])
1306: (8)             test = setxor1d(a, b)
1307: (8)             assert_(isinstance(test, MaskedArray))
1308: (8)             assert_equal(test, [1, 2, 3, 4, 5, 6])
1309: (8)             assert_array_equal([], setxor1d([], []))
1310: (4) def test_isin(self):
1311: (8)     a = np.arange(24).reshape([2, 3, 4])
1312: (8)     mask = np.zeros([2, 3, 4])
1313: (8)     mask[1, 2, 0] = 1
1314: (8)     a = array(a, mask=mask)
1315: (8)     b = array(data=[0, 10, 20, 30, 1, 3, 11, 22, 33],
1316: (18)                 mask=[0, 1, 0, 1, 0, 1, 0, 1, 0])
1317: (8)     ec = zeros((2, 3, 4), dtype=bool)
1318: (8)     ec[0, 0, 0] = True
1319: (8)     ec[0, 0, 1] = True
1320: (8)     ec[0, 2, 3] = True
1321: (8)     c = isin(a, b)
1322: (8)     assert_(isinstance(c, MaskedArray))
1323: (8)     assert_array_equal(c, ec)
1324: (8)     d = np.isin(a, b[~b.mask]) & ~a.mask
1325: (8)     assert_array_equal(c, d)
1326: (4) def test_in1d(self):
1327: (8)     a = array([1, 2, 5, 7, -1], mask=[0, 0, 0, 0, 1])
1328: (8)     b = array([1, 2, 3, 4, 5, -1], mask=[0, 0, 0, 0, 0, 1])
1329: (8)     test = in1d(a, b)
1330: (8)     assert_equal(test, [True, True, True, False, True])
1331: (8)     a = array([5, 5, 2, 1, -1], mask=[0, 0, 0, 0, 1])
1332: (8)     b = array([1, 5, -1], mask=[0, 0, 1])
1333: (8)     test = in1d(a, b)
1334: (8)     assert_equal(test, [True, True, False, True, True])
1335: (8)     assert_array_equal([], in1d([], []))
1336: (4) def test_in1d_invert(self):
1337: (8)     a = array([1, 2, 5, 7, -1], mask=[0, 0, 0, 0, 1])
1338: (8)     b = array([1, 2, 3, 4, 5, -1], mask=[0, 0, 0, 0, 0, 1])
1339: (8)     assert_equal(np.invert(in1d(a, b)), in1d(a, b, invert=True))
1340: (8)     a = array([5, 5, 2, 1, -1], mask=[0, 0, 0, 0, 1])
1341: (8)     b = array([1, 5, -1], mask=[0, 0, 1])
1342: (8)     assert_equal(np.invert(in1d(a, b)), in1d(a, b, invert=True))
1343: (8)     assert_array_equal([], in1d([], [], invert=True))
1344: (4) def test_union1d(self):
1345: (8)     a = array([1, 2, 5, 7, 5, -1], mask=[0, 0, 0, 0, 0, 1])
1346: (8)     b = array([1, 2, 3, 4, 5, -1], mask=[0, 0, 0, 0, 0, 1])
1347: (8)     test = union1d(a, b)
1348: (8)     control = array([1, 2, 3, 4, 5, 7, -1], mask=[0, 0, 0, 0, 0, 0, 1])
1349: (8)     assert_equal(test, control)
1350: (8)     x = array([[0, 1, 2], [3, 4, 5]], mask=[[0, 0, 0], [0, 0, 1]])
1351: (8)     y = array([0, 1, 2, 3, 4], mask=[0, 0, 0, 0, 1])
1352: (8)     ez = array([0, 1, 2, 3, 4, 5], mask=[0, 0, 0, 0, 0, 1])
1353: (8)     z = union1d(x, y)
1354: (8)     assert_equal(z, ez)
1355: (8)     assert_array_equal([], union1d([], []))
1356: (4) def test_setdiff1d(self):
1357: (8)     a = array([6, 5, 4, 7, 7, 1, 2, 1], mask=[0, 0, 0, 0, 0, 0, 0, 1])
1358: (8)     b = array([2, 4, 3, 3, 2, 1, 5])
1359: (8)     test = setdiff1d(a, b)
1360: (8)     assert_equal(test, array([6, 7, -1], mask=[0, 0, 1]))
1361: (8)     a = arange(10)
1362: (8)     b = arange(8)
1363: (8)     assert_equal(setdiff1d(a, b), array([8, 9]))
1364: (8)     a = array([], np.uint32, mask[])
1365: (8)     assert_equal(setdiff1d(a, []).dtype, np.uint32)

```

```

1366: (4)           def test_setdiff1d_char_array(self):
1367: (8)             a = np.array(['a', 'b', 'c'])
1368: (8)             b = np.array(['a', 'b', 's'])
1369: (8)             assert_array_equal(setdiff1d(a, b), np.array(['c']))
1370: (0)           class TestShapeBase:
1371: (4)             def test_atleast_2d(self):
1372: (8)               a = masked_array([0, 1, 2], mask=[0, 1, 0])
1373: (8)               b = atleast_2d(a)
1374: (8)               assert_equal(b.shape, (1, 3))
1375: (8)               assert_equal(b.mask.shape, b.data.shape)
1376: (8)               assert_equal(a.shape, (3,))
1377: (8)               assert_equal(a.mask.shape, a.data.shape)
1378: (8)               assert_equal(b.mask.shape, b.data.shape)
1379: (4)             def test_shape_scalar(self):
1380: (8)               b = atleast_1d(1.0)
1381: (8)               assert_equal(b.shape, (1,))
1382: (8)               assert_equal(b.mask.shape, b.shape)
1383: (8)               assert_equal(b.data.shape, b.shape)
1384: (8)               b = atleast_1d(1.0, 2.0)
1385: (8)               for a in b:
1386: (12)                 assert_equal(a.shape, (1,))
1387: (12)                 assert_equal(a.mask.shape, a.shape)
1388: (12)                 assert_equal(a.data.shape, a.shape)
1389: (8)               b = atleast_2d(1.0)
1390: (8)               assert_equal(b.shape, (1, 1))
1391: (8)               assert_equal(b.mask.shape, b.shape)
1392: (8)               assert_equal(b.data.shape, b.shape)
1393: (8)               b = atleast_2d(1.0, 2.0)
1394: (8)               for a in b:
1395: (12)                 assert_equal(a.shape, (1, 1))
1396: (12)                 assert_equal(a.mask.shape, a.shape)
1397: (12)                 assert_equal(a.data.shape, a.shape)
1398: (8)               b = atleast_3d(1.0)
1399: (8)               assert_equal(b.shape, (1, 1, 1))
1400: (8)               assert_equal(b.mask.shape, b.shape)
1401: (8)               assert_equal(b.data.shape, b.shape)
1402: (8)               b = atleast_3d(1.0, 2.0)
1403: (8)               for a in b:
1404: (12)                 assert_equal(a.shape, (1, 1, 1))
1405: (12)                 assert_equal(a.mask.shape, a.shape)
1406: (12)                 assert_equal(a.data.shape, a.shape)
1407: (8)               b = diagflat(1.0)
1408: (8)               assert_equal(b.shape, (1, 1))
1409: (8)               assert_equal(b.mask.shape, b.data.shape)
1410: (0)           class TestNDEnumerate:
1411: (4)             def test_ndenumerate_nomasked(self):
1412: (8)               ordinary = np.arange(6.).reshape((1, 3, 2))
1413: (8)               empty_mask = np.zeros_like(ordinary, dtype=bool)
1414: (8)               with_mask = masked_array(ordinary, mask=empty_mask)
1415: (8)               assert_equal(list(ndenumerate(ordinary)),
1416: (21)                 list(ndenumerate(ordinary)))
1417: (8)               assert_equal(list(ndenumerate(ordinary)),
1418: (21)                 list(ndenumerate(with_mask)))
1419: (8)               assert_equal(list(ndenumerate(with_mask)),
1420: (21)                 list(ndenumerate(with_mask, compressed=False)))
1421: (4)             def test_ndenumerate_allmasked(self):
1422: (8)               a = masked_all(())
1423: (8)               b = masked_all((100,))
1424: (8)               c = masked_all((2, 3, 4))
1425: (8)               assert_equal(list(ndenumerate(a)), [])
1426: (8)               assert_equal(list(ndenumerate(b)), [])
1427: (8)               assert_equal(list(ndenumerate(b, compressed=False)),
1428: (21)                 list(zip(np.ndindex((100,)), 100 * [masked])))
1429: (8)               assert_equal(list(ndenumerate(c)), [])
1430: (8)               assert_equal(list(ndenumerate(c, compressed=False)),
1431: (21)                 list(zip(np.ndindex((2, 3, 4)), 2 * 3 * 4 * [masked])))
1432: (4)             def test_ndenumerate_mixedmasked(self):
1433: (8)               a = masked_array(np.arange(12).reshape((3, 4)),
1434: (25)                 mask=[[1, 1, 1, 1],

```

```

1435: (31) [1, 1, 0, 1],
1436: (31) [0, 0, 0, 0])
1437: (8) items = [((1, 2), 6),
1438: (17) ((2, 0), 8), ((2, 1), 9), ((2, 2), 10), ((2, 3), 11)]
1439: (8) assert_equal(list(ndenumerate(a)), items)
1440: (8) assert_equal(len(list(ndenumerate(a, compressed=False))), a.size)
1441: (8) for coordinate, value in ndenumerate(a, compressed=False):
1442: (12)     assert_equal(a[coordinate], value)
1443: (0) class TestStack:
1444: (4)     def test_stack_1d(self):
1445: (8)         a = masked_array([0, 1, 2], mask=[0, 1, 0])
1446: (8)         b = masked_array([9, 8, 7], mask=[1, 0, 0])
1447: (8)         c = stack([a, b], axis=0)
1448: (8)         assert_equal(c.shape, (2, 3))
1449: (8)         assert_array_equal(a.mask, c[0].mask)
1450: (8)         assert_array_equal(b.mask, c[1].mask)
1451: (8)         d = vstack([a, b])
1452: (8)         assert_array_equal(c.data, d.data)
1453: (8)         assert_array_equal(c.mask, d.mask)
1454: (8)         c = stack([a, b], axis=1)
1455: (8)         assert_equal(c.shape, (3, 2))
1456: (8)         assert_array_equal(a.mask, c[:, 0].mask)
1457: (8)         assert_array_equal(b.mask, c[:, 1].mask)
1458: (4)     def test_stack_masks(self):
1459: (8)         a = masked_array([0, 1, 2], mask=True)
1460: (8)         b = masked_array([9, 8, 7], mask=False)
1461: (8)         c = stack([a, b], axis=0)
1462: (8)         assert_equal(c.shape, (2, 3))
1463: (8)         assert_array_equal(a.mask, c[0].mask)
1464: (8)         assert_array_equal(b.mask, c[1].mask)
1465: (8)         d = vstack([a, b])
1466: (8)         assert_array_equal(c.data, d.data)
1467: (8)         assert_array_equal(c.mask, d.mask)
1468: (8)         c = stack([a, b], axis=1)
1469: (8)         assert_equal(c.shape, (3, 2))
1470: (8)         assert_array_equal(a.mask, c[:, 0].mask)
1471: (8)         assert_array_equal(b.mask, c[:, 1].mask)
1472: (4)     def test_stack_nd(self):
1473: (8)         shp = (3, 2)
1474: (8)         d1 = np.random.randint(0, 10, shp)
1475: (8)         d2 = np.random.randint(0, 10, shp)
1476: (8)         m1 = np.random.randint(0, 2, shp).astype(bool)
1477: (8)         m2 = np.random.randint(0, 2, shp).astype(bool)
1478: (8)         a1 = masked_array(d1, mask=m1)
1479: (8)         a2 = masked_array(d2, mask=m2)
1480: (8)         c = stack([a1, a2], axis=0)
1481: (8)         c_shp = (2,) + shp
1482: (8)         assert_equal(c.shape, c_shp)
1483: (8)         assert_array_equal(a1.mask, c[0].mask)
1484: (8)         assert_array_equal(a2.mask, c[1].mask)
1485: (8)         c = stack([a1, a2], axis=-1)
1486: (8)         c_shp = shp + (2,)
1487: (8)         assert_equal(c.shape, c_shp)
1488: (8)         assert_array_equal(a1.mask, c[..., 0].mask)
1489: (8)         assert_array_equal(a2.mask, c[..., 1].mask)
1490: (8)         shp = (3, 2, 4, 5,)
1491: (8)         d1 = np.random.randint(0, 10, shp)
1492: (8)         d2 = np.random.randint(0, 10, shp)
1493: (8)         m1 = np.random.randint(0, 2, shp).astype(bool)
1494: (8)         m2 = np.random.randint(0, 2, shp).astype(bool)
1495: (8)         a1 = masked_array(d1, mask=m1)
1496: (8)         a2 = masked_array(d2, mask=m2)
1497: (8)         c = stack([a1, a2], axis=0)
1498: (8)         c_shp = (2,) + shp
1499: (8)         assert_equal(c.shape, c_shp)
1500: (8)         assert_array_equal(a1.mask, c[0].mask)
1501: (8)         assert_array_equal(a2.mask, c[1].mask)
1502: (8)         c = stack([a1, a2], axis=-1)
1503: (8)         c_shp = shp + (2,)

```

```

1504: (8)             assert_equal(c.shape, c_shp)
1505: (8)             assert_array_equal(a1.mask, c[..., 0].mask)
1506: (8)             assert_array_equal(a2.mask, c[..., 1].mask)
-----
```

## File 270 - test\_deprecations.py:

```

1: (0)             """Test deprecation and future warnings.
2: (0)             """
3: (0)             import pytest
4: (0)             import numpy as np
5: (0)             from numpy.testing import assert_warns
6: (0)             from numpy.ma.testutils import assert_equal
7: (0)             from numpy.ma.core import MaskedArrayFutureWarning
8: (0)             import io
9: (0)             import textwrap
10: (0)            class TestArgsort:
11: (4)              """ gh-8701 """
12: (4)              def _test_base(self, argsort, cls):
13: (8)                  arr_0d = np.array(1).view(cls)
14: (8)                  argsort(arr_0d)
15: (8)                  arr_1d = np.array([1, 2, 3]).view(cls)
16: (8)                  argsort(arr_1d)
17: (8)                  arr_2d = np.array([[1, 2], [3, 4]]).view(cls)
18: (8)                  result = assert_warns(
19: (12)                      np.ma.core.MaskedArrayFutureWarning, argsort, arr_2d)
20: (8)                  assert_equal(result, argsort(arr_2d, axis=None))
21: (8)                  argsort(arr_2d, axis=None)
22: (8)                  argsort(arr_2d, axis=-1)
23: (4)                  def test_function_ndarray(self):
24: (8)                      return self._test_base(np.ma.argsort, np.ndarray)
25: (4)                  def test_function_maskedarray(self):
26: (8)                      return self._test_base(np.ma.argsort, np.ma.MaskedArray)
27: (4)                  def test_method(self):
28: (8)                      return self._test_base(np.ma.MaskedArray.argsort, np.ma.MaskedArray)
29: (0)            class TestMinimumMaximum:
30: (4)              def test_axis_default(self):
31: (8)                  data1d = np.ma.arange(6)
32: (8)                  data2d = data1d.reshape(2, 3)
33: (8)                  ma_min = np.ma.minimum.reduce
34: (8)                  ma_max = np.ma.maximum.reduce
35: (8)                  result = assert_warns(MaskedArrayFutureWarning, ma_max, data2d)
36: (8)                  assert_equal(result, ma_max(data2d, axis=None))
37: (8)                  result = assert_warns(MaskedArrayFutureWarning, ma_min, data2d)
38: (8)                  assert_equal(result, ma_min(data2d, axis=None))
39: (8)                  result = ma_min(data1d)
40: (8)                  assert_equal(result, ma_min(data1d, axis=None))
41: (8)                  assert_equal(result, ma_min(data1d, axis=0))
42: (8)                  result = ma_max(data1d)
43: (8)                  assert_equal(result, ma_max(data1d, axis=None))
44: (8)                  assert_equal(result, ma_max(data1d, axis=0))
45: (0)            class TestFromtextfile:
46: (4)              def test_fromtextfile_delimiter(self):
47: (8)                  textfile = io.StringIO(textwrap.dedent(
48: (12)                    """
49: (12)                    A,B,C,D
50: (12)                    'string 1';1;1.0;'mixed column'
51: (12)                    'string 2';2;2.0;
52: (12)                    'string 3';3;3.0;123
53: (12)                    'string 4';4;4.0;3.14
54: (12)                    """
55: (8)                ))
56: (8)                with pytest.warns(DeprecationWarning):
57: (12)                    result = np.ma.mrecords.fromtextfile(textfile, delimiter=';')
-----
```

## File 271 - test\_mrecords.py:

```

1: (0)     """Tests suite for mrecords.
2: (0)     :author: Pierre Gerard-Marchant
3: (0)     :contact: pierregm_at_uga_dot_edu
4: (0)
5: (0)     import numpy as np
6: (0)     import numpy.ma as ma
7: (0)     from numpy import recarray
8: (0)     from numpy.ma import masked, nomask
9: (0)     from numpy.testing import temppath
10: (0)    from numpy.core.records import (
11: (4)        fromrecords as recfromrecords, fromarrays as recfromarrays
12: (4)    )
13: (0)    from numpy.ma.mrecords import (
14: (4)        MaskedRecords, mrecarray, fromarrays, fromtextfile, fromrecords,
15: (4)        addfield
16: (4)    )
17: (0)    from numpy.ma.testutils import (
18: (4)        assert_, assert_equal,
19: (4)        assert_equal_records,
20: (4)    )
21: (0)    from numpy.compat import pickle
22: (0)    class TestMRecords:
23: (4)        ilist = [1, 2, 3, 4, 5]
24: (4)        flist = [1.1, 2.2, 3.3, 4.4, 5.5]
25: (4)        slist = [b'one', b'two', b'three', b'four', b'five']
26: (4)        dtype = [(b'a', int), (b'b', float), (b'c', '|S8')]
27: (4)        mask = [0, 1, 0, 0, 1]
28: (4)        base = ma.array(list(zip(ilist, flist, slist)), mask=mask, dtype=dtype)
29: (4)    def test_byview(self):
30: (8)        base = self.base
31: (8)        mbase = base.view(mrecarray)
32: (8)        assert_equal(mbase.recordmask, base.recordmask)
33: (8)        assert_equal_records(mbase._mask, base._mask)
34: (8)        assert_(isinstance(mbase._data, recarray))
35: (8)        assert_equal_records(mbase._data, base._data.view(recarray))
36: (8)        for field in ('a', 'b', 'c'):
37: (12)            assert_equal(base[field], mbase[field])
38: (8)        assert_equal_records(mbase.view(mrecarray), mbase)
39: (4)    def test_get(self):
40: (8)        base = self.base.copy()
41: (8)        mbase = base.view(mrecarray)
42: (8)        for field in ('a', 'b', 'c'):
43: (12)            assert_equal(getattr(mbase, field), mbase[field])
44: (12)            assert_equal(base[field], mbase[field])
45: (8)        mbase_first = mbase[0]
46: (8)        assert_(isinstance(mbase_first, mrecarray))
47: (8)        assert_equal(mbase_first.dtype, mbase.dtype)
48: (8)        assert_equal(mbase_first.tolist(), (1, 1.1, b'one'))
49: (8)        assert_equal(mbase_first.recordmask, nomask)
50: (8)        assert_equal(mbase_first._mask.item(), (False, False, False))
51: (8)        assert_equal(mbase_first['a'], mbase['a'][0])
52: (8)        mbase_last = mbase[-1]
53: (8)        assert_(isinstance(mbase_last, mrecarray))
54: (8)        assert_equal(mbase_last.dtype, mbase.dtype)
55: (8)        assert_equal(mbase_last.tolist(), (None, None, None))
56: (8)        assert_equal(mbase_last.recordmask, True)
57: (8)        assert_equal(mbase_last._mask.item(), (True, True, True))
58: (8)        assert_equal(mbase_last['a'], mbase['a'][-1])
59: (8)        assert_((mbase_last['a'] is masked))
60: (8)        mbase_sl = mbase[:2]
61: (8)        assert_(isinstance(mbase_sl, mrecarray))
62: (8)        assert_equal(mbase_sl.dtype, mbase.dtype)
63: (8)        assert_equal(mbase_sl.recordmask, [0, 1])
64: (8)        assert_equal_records(mbase_sl.mask,
65: (29)                            np.array([(False, False, False),
66: (39)                                (True, True, True)],
67: (38)                                dtype=mbase._mask.dtype))
68: (8)        assert_equal_records(mbase_sl, base[:2].view(mrecarray))

```

```

69: (8)             for field in ('a', 'b', 'c'):
70: (12)             assert_equal(getattr(mbase_sl, field), base[:2][field])
71: (4)         def test_set_fields(self):
72: (8)             base = self.base.copy()
73: (8)             mbase = base.view(mrecarray)
74: (8)             mbase = mbase.copy()
75: (8)             mbase.fill_value = (999999, 1e20, 'N/A')
76: (8)             mbase.a._data[:] = 5
77: (8)             assert_equal(mbase['a']._data, [5, 5, 5, 5, 5])
78: (8)             assert_equal(mbase['a']._mask, [0, 1, 0, 0, 1])
79: (8)             mbase.a = 1
80: (8)             assert_equal(mbase['a']._data, [1]*5)
81: (8)             assert_equal(ma.getmaskarray(mbase['a']), [0]*5)
82: (8)             assert_equal(mbase.recordmask, [False]*5)
83: (8)             assert_equal(mbase._mask.tolist(),
84: (21)                 np.array([(0, 0, 0),
85: (31)                     (0, 1, 1),
86: (31)                     (0, 0, 0),
87: (31)                     (0, 0, 0),
88: (31)                     (0, 1, 1)],
89: (30)                         dtype=bool))
90: (8)             mbase.c = masked
91: (8)             assert_equal(mbase.c.mask, [1]*5)
92: (8)             assert_equal(mbase.c.recordmask, [1]*5)
93: (8)             assert_equal(ma.getmaskarray(mbase['c']), [1]*5)
94: (8)             assert_equal(ma.getdata(mbase['c']), [b'N/A']*5)
95: (8)             assert_equal(mbase._mask.tolist(),
96: (21)                 np.array([(0, 0, 1),
97: (31)                     (0, 1, 1),
98: (31)                     (0, 0, 1),
99: (31)                     (0, 0, 1),
100: (31)                     (0, 1, 1)],
101: (30)                         dtype=bool))
102: (8)             mbase = base.view(mrecarray).copy()
103: (8)             mbase.a[3:] = 5
104: (8)             assert_equal(mbase.a, [1, 2, 3, 5, 5])
105: (8)             assert_equal(mbase.a._mask, [0, 1, 0, 0, 0])
106: (8)             mbase.b[3:] = masked
107: (8)             assert_equal(mbase.b, base['b'])
108: (8)             assert_equal(mbase.b._mask, [0, 1, 0, 1, 1])
109: (8)             ndtype = [('alpha', '|S1'), ('num', int)]
110: (8)             data = ma.array([('a', 1), ('b', 2), ('c', 3)], dtype=ndtype)
111: (8)             rdata = data.view(MaskedRecords)
112: (8)             val = ma.array([10, 20, 30], mask=[1, 0, 0])
113: (8)             rdata['num'] = val
114: (8)             assert_equal(rdata.num, val)
115: (8)             assert_equal(rdata.num.mask, [1, 0, 0])
116: (4)         def test_set_fields_mask(self):
117: (8)             base = self.base.copy()
118: (8)             mbase = base.view(mrecarray)
119: (8)             mbase['a'][-2] = masked
120: (8)             assert_equal(mbase.a, [1, 2, 3, 4, 5])
121: (8)             assert_equal(mbase.a._mask, [0, 1, 0, 1, 1])
122: (8)             mbase = fromarrays([np.arange(5), np.random.rand(5)],
123: (27)                             dtype=[('a', int), ('b', float)])
124: (8)             mbase['a'][-2] = masked
125: (8)             assert_equal(mbase.a, [0, 1, 2, 3, 4])
126: (8)             assert_equal(mbase.a._mask, [0, 0, 0, 1, 0])
127: (4)         def test_set_mask(self):
128: (8)             base = self.base.copy()
129: (8)             mbase = base.view(mrecarray)
130: (8)             mbase.mask = masked
131: (8)             assert_equal(ma.getmaskarray(mbase['b']), [1]*5)
132: (8)             assert_equal(mbase['a']._mask, mbase['b']._mask)
133: (8)             assert_equal(mbase['a']._mask, mbase['c']._mask)
134: (8)             assert_equal(mbase._mask.tolist(),
135: (21)                 np.array([(1, 1, 1)]*5, dtype=bool))
136: (8)             mbase.mask = nomask
137: (8)             assert_equal(ma.getmaskarray(mbase['c']), [0]*5)

```

```

138: (8)
139: (21)
140: (4)
141: (8)
142: (8)
143: (8)
144: (8)
145: (8)
146: (8)
147: (8)
148: (8)
149: (8)
150: (8)
151: (4)
152: (8)
153: (8)
154: (12)
155: (12)
156: (8)
157: (8)
158: (8)
159: (8)
160: (8)
161: (8)
162: (8)
163: (8)
164: (8)
165: (4)
166: (8)
167: (8)
168: (8)
169: (8)
170: (12)
171: (12)
172: (21)
173: (8)
174: (8)
175: (8)
176: (8)
177: (8)
178: (8)
179: (8)
180: (8)
181: (21)
182: (8)
183: (8)
184: (8)
185: (8)
186: (8)
187: (8)
188: (8)
189: (8)
190: (21)
191: (8)
192: (4)
193: (8)
194: (8)
195: (8)
196: (8)
197: (12)
198: (12)
199: (12)
200: (12)
201: (25)
202: (12)
203: (12)
204: (12)
205: (8)
206: (12)

        assert_equal(mbase._mask.tolist(),
                      np.array([(0, 0, 0)]*5, dtype=bool))
    def test_set_mask_fromarray(self):
        base = self.base.copy()
        mbase = base.view(mrecarray)
        mbase.mask = [1, 0, 0, 0, 1]
        assert_equal(mbase.a.mask, [1, 0, 0, 0, 1])
        assert_equal(mbase.b.mask, [1, 0, 0, 0, 1])
        assert_equal(mbase.c.mask, [1, 0, 0, 0, 1])
        mbase.mask = [0, 0, 0, 0, 1]
        assert_equal(mbase.a.mask, [0, 0, 0, 0, 1])
        assert_equal(mbase.b.mask, [0, 0, 0, 0, 1])
        assert_equal(mbase.c.mask, [0, 0, 0, 0, 1])
    def test_set_mask_fromfields(self):
        mbase = self.base.copy().view(mrecarray)
        nmask = np.array(
            [(0, 1, 0), (0, 1, 0), (1, 0, 1), (1, 0, 1), (0, 0, 0)],
            dtype=[('a', bool), ('b', bool), ('c', bool)])
        mbase.mask = nmask
        assert_equal(mbase.a.mask, [0, 0, 1, 1, 0])
        assert_equal(mbase.b.mask, [1, 1, 0, 0, 0])
        assert_equal(mbase.c.mask, [0, 0, 1, 1, 0])
        mbase.mask = False
        mbase.fieldmask = nmask
        assert_equal(mbase.a.mask, [0, 0, 1, 1, 0])
        assert_equal(mbase.b.mask, [1, 1, 0, 0, 0])
        assert_equal(mbase.c.mask, [0, 0, 1, 1, 0])
    def test_set_elements(self):
        base = self.base.copy()
        mbase = base.view(mrecarray).copy()
        mbase[-2] = masked
        assert_equal(
            mbase._mask.tolist(),
            np.array([(0, 0, 0), (1, 1, 1), (0, 0, 0), (1, 1, 1), (1, 1, 1)],

dtype=bool))
        assert_equal(mbase.recordmask, [0, 1, 0, 1, 1])
        mbase = base.view(mrecarray).copy()
        mbase[:2] = (5, 5, 5)
        assert_equal(mbase.a._data, [5, 5, 3, 4, 5])
        assert_equal(mbase.a._mask, [0, 0, 0, 0, 1])
        assert_equal(mbase.b._data, [5., 5., 3.3, 4.4, 5.5])
        assert_equal(mbase.b._mask, [0, 0, 0, 0, 1])
        assert_equal(mbase.c._data,
                     [b'5', b'5', b'three', b'four', b'five'])
        assert_equal(mbase.b._mask, [0, 0, 0, 0, 1])
        mbase = base.view(mrecarray).copy()
        mbase[:2] = masked
        assert_equal(mbase.a._data, [1, 2, 3, 4, 5])
        assert_equal(mbase.a._mask, [1, 1, 0, 0, 1])
        assert_equal(mbase.b._data, [1.1, 2.2, 3.3, 4.4, 5.5])
        assert_equal(mbase.b._mask, [1, 1, 0, 0, 1])
        assert_equal(mbase.c._data,
                     [b'one', b'two', b'three', b'four', b'five'])
        assert_equal(mbase.b._mask, [1, 1, 0, 0, 1])
    def test_setslices_hardmask(self):
        base = self.base.copy()
        mbase = base.view(mrecarray)
        mbase.harden_mask()
        try:
            mbase[-2:] = (5, 5, 5)
            assert_equal(mbase.a._data, [1, 2, 3, 5, 5])
            assert_equal(mbase.b._data, [1.1, 2.2, 3.3, 5, 5.5])
            assert_equal(mbase.c._data,
                         [b'one', b'two', b'three', b'5', b'five'])
            assert_equal(mbase.a._mask, [0, 1, 0, 0, 1])
            assert_equal(mbase.b._mask, mbase.a._mask)
            assert_equal(mbase.b._mask, mbase.c._mask)
        except NotImplementedError:
            pass

```

```

207: (8)
208: (12)
209: (8)
210: (12)
211: (8)
212: (12)
213: (8)
214: (12)
215: (8)
216: (12)
217: (4)
218: (8)
219: (8)
220: (8)
221: (8)
222: (8)
223: (8)
224: (8)
225: (8)
226: (8)
227: (8)
228: (29)
229: (8)
230: (8)
231: (4)
232: (8)
233: (8)
234: (8)
235: (12)
236: (12)
237: (12)
238: (12)
239: (12)
240: (12)
241: (4)
242: (8)
243: (8)
244: (8)
245: (8)
246: (8)
247: (26)
248: (8)
249: (8)
250: (8)
251: (47)
252: (8)
253: (47)
254: (4)
255: (8)
256: (8)
257: (8)
258: (8)
259: (8)
260: (26)
261: (8)
262: (21)
263: (22)
264: (4)
265: (8)
266: (8)
267: (8)
268: (4)
269: (8)
270: (8)
271: (8)
272: (8)
273: (8)
274: (8)
275: (21)

        except AssertionError:
            raise
        else:
            raise Exception("Flexible hard masks should be supported !")
    try:
        mbase[-2:] = 3
    except (NotImplementedError, TypeError):
        pass
    else:
        raise TypeError("Should have expected a readable buffer object!")
def test_hardmask(self):
    base = self.base.copy()
    mbase = base.view(mrecarray)
    mbase.harden_mask()
    assert_(mbase._hardmask)
    mbase.mask = nomask
    assert_equal_records(mbase._mask, base._mask)
    mbase.soften_mask()
    assert_(not mbase._hardmask)
    mbase.mask = nomask
    assert_equal_records(mbase._mask,
                         ma.make_mask_none(base.shape, base.dtype))
    assert_(ma.make_mask(mbase['b']._mask) is nomask)
    assert_equal(mbase['a']._mask, mbase['b']._mask)
def test_pickling(self):
    base = self.base.copy()
    mrec = base.view(mrecarray)
    for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
        _ = pickle.dumps(mrec, protocol=proto)
        mrec_ = pickle.loads(_)
        assert_equal(mrec_.dtype, mrec.dtype)
        assert_equal_records(mrec_.data, mrec.data)
        assert_equal(mrec_.mask, mrec.mask)
        assert_equal_records(mrec_.mask, mrec.mask)
def test_filled(self):
    _a = ma.array([1, 2, 3], mask=[0, 0, 1], dtype=int)
    _b = ma.array([1.1, 2.2, 3.3], mask=[0, 0, 1], dtype=float)
    _c = ma.array(['one', 'two', 'three'], mask=[0, 0, 1], dtype='|S8')
    ddtype = [('a', int), ('b', float), ('c', '|S8')]
    mrec = fromarrays([_a, _b, _c], dtype=ddtype,
                      fill_value=(99999, 99999., 'N/A'))
    mrecfilled = mrec.filled()
    assert_equal(mrecfilled['a'], np.array((1, 2, 99999), dtype=int))
    assert_equal(mrecfilled['b'], np.array((1.1, 2.2, 99999.), dtype=float))
    assert_equal(mrecfilled['c'], np.array(('one', 'two', 'N/A'), dtype='|S8'))
def test_tolist(self):
    _a = ma.array([1, 2, 3], mask=[0, 0, 1], dtype=int)
    _b = ma.array([1.1, 2.2, 3.3], mask=[0, 0, 1], dtype=float)
    _c = ma.array(['one', 'two', 'three'], mask=[1, 0, 0], dtype='|S8')
    ddtype = [('a', int), ('b', float), ('c', '|S8')]
    mrec = fromarrays([_a, _b, _c], dtype=ddtype,
                      fill_value=(99999, 99999., 'N/A'))
    assert_equal(mrec.tolist(),
                 [(1, 1.1, None), (2, 2.2, b'two'),
                  (None, None, b'three')])
def test_withnames(self):
    x = mrecarray(1, formats=float, names='base')
    x[0]['base'] = 10
    assert_equal(x['base'][0], 10)
def test_exotic_formats(self):
    easy = mrecarray(1, dtype=[('i', int), ('s', '|S8'), ('f', float)])
    easy[0] = masked
    assert_equal(easy.filled(1).item(), (1, b'1', 1.))
    solo = mrecarray(1, dtype=[('f0', '<f8', (2, 2))])
    solo[0] = masked
    assert_equal(solo.filled(1).item(),
                np.array((1,), dtype=solo.dtype).item())

```

```

276: (8)             mult = mrecarray(2, dtype="i4, (2,3)float, float")
277: (8)             mult[0] = masked
278: (8)             mult[1] = (1, 1, 1)
279: (8)             mult.filled(0)
280: (8)             assert_equal_records(mult.filled(0),
281: (29)                         np.array([(0, 0, 0), (1, 1, 1)],
282: (38)                         dtype=mult.dtype))
283: (0)             class TestView:
284: (4)                 def setup_method(self):
285: (8)                     (a, b) = (np.arange(10), np.random.rand(10))
286: (8)                     ndtype = [('a', float), ('b', float)]
287: (8)                     arr = np.array(list(zip(a, b)), dtype=ndtype)
288: (8)                     mrec = fromarrays([a, b], dtype=ndtype, fill_value=(-9., -99.))
289: (8)                     mrec.mask[3] = (False, True)
290: (8)                     self.data = (mrec, a, b, arr)
291: (4)                 def test_view_by_itself(self):
292: (8)                     (mrec, a, b, arr) = self.data
293: (8)                     test = mrec.view()
294: (8)                     assert_(isinstance(test, MaskedRecords))
295: (8)                     assert_equal_records(test, mrec)
296: (8)                     assert_equal_records(test._mask, mrec._mask)
297: (4)                 def test_view_simple_dtype(self):
298: (8)                     (mrec, a, b, arr) = self.data
299: (8)                     ntype = (float, 2)
300: (8)                     test = mrec.view(ntype)
301: (8)                     assert_(isinstance(test, ma.MaskedArray))
302: (8)                     assert_equal(test, np.array(list(zip(a, b)), dtype=float))
303: (8)                     assert_(test[3, 1] is ma.masked)
304: (4)                 def test_view_flexible_type(self):
305: (8)                     (mrec, a, b, arr) = self.data
306: (8)                     alttype = [('A', float), ('B', float)]
307: (8)                     test = mrec.view(alttype)
308: (8)                     assert_(isinstance(test, MaskedRecords))
309: (8)                     assert_equal_records(test, arr.view(alttype))
310: (8)                     assert_(test['B'][3] is masked)
311: (8)                     assert_equal(test.dtype, np.dtype(alttype))
312: (8)                     assert_(test._fill_value is None)
313: (0)             class TestMRecordsImport:
314: (4)                 _a = ma.array([1, 2, 3], mask=[0, 0, 1], dtype=int)
315: (4)                 _b = ma.array([1.1, 2.2, 3.3], mask=[0, 0, 1], dtype=float)
316: (4)                 _c = ma.array(['one', 'two', 'three'],
317: (18)                               mask=[0, 0, 1], dtype='|S8')
318: (4)                 ddtype = [('a', int), ('b', float), ('c', '|S8')]
319: (4)                 mrec = fromarrays([_a, _b, _c], dtype=ddtype,
320: (22)                               fill_value=(b'99999', b'99999.',
321: (34)                               b'N/A'))
322: (4)                 nrec = recfromarrays((_a._data, _b._data, _c._data), dtype=ddtype)
323: (4)                 data = (mrec, nrec, ddtype)
324: (4)                 def test_fromarrays(self):
325: (8)                     _a = ma.array([1, 2, 3], mask=[0, 0, 1], dtype=int)
326: (8)                     _b = ma.array([1.1, 2.2, 3.3], mask=[0, 0, 1], dtype=float)
327: (8)                     _c = ma.array(['one', 'two', 'three'], mask=[0, 0, 1], dtype='|S8')
328: (8)                     (mrec, nrec, _) = self.data
329: (8)                     for (f, l) in zip(('a', 'b', 'c'), (_a, _b, _c)):
330: (12)                         assert_equal(getattr(mrec, f).mask, l._mask)
331: (8)                         _x = ma.array([1, 1.1, 'one'], mask=[1, 0, 0], dtype=object)
332: (8)                         assert_equal_records(fromarrays(_x, dtype=mrec.dtype), mrec[0])
333: (4)                 def test_fromrecords(self):
334: (8)                     (mrec, nrec, ddtype) = self.data
335: (8)                     palist = [(1, 'abc', 3.7000002861022949, 0),
336: (18)                           (2, 'xy', 6.6999998092651367, 1),
337: (18)                           (0, ' ', 0.40000000596046448, 0)]
338: (8)                     pa = recfromrecords(palist, names='c1, c2, c3, c4')
339: (8)                     mpa = fromrecords(palist, names='c1, c2, c3, c4')
340: (8)                     assert_equal_records(pa, mpa)
341: (8)                     _mrec = fromrecords(nrec)
342: (8)                     assert_equal(_mrec.dtype, mrec.dtype)
343: (8)                     for field in _mrec.dtype.names:
344: (12)                         assert_equal(getattr(_mrec, field), getattr(mrec._data, field))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

345: (8)             _mrec = fromrecords(nrec.tolist(), names='c1,c2,c3')
346: (8)             assert_equal(_mrec.dtype, [('c1', int), ('c2', float), ('c3', '|S5')])
347: (8)             for (f, n) in zip(('c1', 'c2', 'c3'), ('a', 'b', 'c')):
348: (12)                 assert_equal(getattr(_mrec, f), getattr(mrec._data, n))
349: (8)             _mrec = fromrecords(mrec)
350: (8)             assert_equal(_mrec.dtype, mrec.dtype)
351: (8)             assert_equal_records(_mrec._data, mrec.filled())
352: (8)             assert_equal_records(_mrec._mask, mrec._mask)
353: (4)             def test_fromrecords_wmask(self):
354: (8)                 (mrec, nrec, dtype) = self.data
355: (8)                 _mrec = fromrecords(nrec.tolist(), dtype=dtype, mask=[0, 1, 0,])
356: (8)                 assert_equal_records(_mrec._data, mrec._data)
357: (8)                 assert_equal(_mrec._mask.tolist(), [(0, 0, 0), (1, 1, 1), (0, 0, 0)])
358: (8)                 _mrec = fromrecords(nrec.tolist(), dtype=dtype, mask=True)
359: (8)                 assert_equal_records(_mrec._data, mrec._data)
360: (8)                 assert_equal(_mrec._mask.tolist(), [(1, 1, 1), (1, 1, 1), (1, 1, 1)])
361: (8)                 _mrec = fromrecords(nrec.tolist(), dtype=dtype, mask=mrec._mask)
362: (8)                 assert_equal_records(_mrec._data, mrec._data)
363: (8)                 assert_equal(_mrec._mask.tolist(), mrec._mask.tolist())
364: (8)                 _mrec = fromrecords(nrec.tolist(), dtype=dtype,
365: (28)                         mask=mrec._mask.tolist())
366: (8)                 assert_equal_records(_mrec._data, mrec._data)
367: (8)                 assert_equal(_mrec._mask.tolist(), mrec._mask.tolist())
368: (4)             def test_fromtextfile(self):
369: (8)                 fcontent = (
370: (0)                 """#
371: (0)                 'One (S)', 'Two (I)', 'Three (F)', 'Four (M)', 'Five (-)', 'Six (C)'
372: (0)                 'strings', 1, 1.0, 'mixed column', , 1
373: (0)                 'with embedded "double quotes"', 2, 2.0, 1.0, , 1
374: (0)                 'strings', 3, 3.0E5, 3, , 1
375: (0)                 'strings', 4, -1e-10, , , 1
376: (0)                 """
377: (8)                 with temppath() as path:
378: (12)                     with open(path, 'w') as f:
379: (16)                         f.write(fcontent)
380: (12)                     mrectxt = fromtextfile(path, delimiter=',', varnames='ABCDEFG')
381: (8)                     assert_(isinstance(mrectxt, MaskedRecords))
382: (8)                     assert_equal(mrectxt.F, [1, 1, 1, 1])
383: (8)                     assert_equal(mrectxt.E._mask, [1, 1, 1, 1])
384: (8)                     assert_equal(mrectxt.C, [1, 2, 3.e+5, -1e-10])
385: (4)             def test_addfield(self):
386: (8)                 (mrec, nrec, dtype) = self.data
387: (8)                 (d, m) = ([100, 200, 300], [1, 0, 0])
388: (8)                 mrec = addfield(mrec, ma.array(d, mask=m))
389: (8)                 assert_equal(mrec.f3, d)
390: (8)                 assert_equal(mrec.f3._mask, m)
391: (0)             def test_record_array_with_object_field():
392: (4)                 y = ma.masked_array(
393: (8)                     [(1, '2'), (3, '4')],
394: (8)                     mask=[(0, 0), (0, 1)],
395: (8)                     dtype=[('a', int), ('b', object)])
396: (4)                 y[1]

```

---

## File 272 - test\_old\_ma.py:

```

1: (0)             from functools import reduce
2: (0)             import pytest
3: (0)             import numpy as np
4: (0)             import numpy.core.umath as umath
5: (0)             import numpy.core.fromnumeric as fromnumeric
6: (0)             from numpy.testing import (
7: (4)                 assert_, assert_raises, assert_equal,
8: (4)             )
9: (0)             from numpy.ma import (
10: (4)                 MaskType, MaskedArray, absolute, add, all, allclose, allequal, alltrue,
11: (4)                 arange, arccos, arcsin, arctan, arctan2, array, average, choose,
12: (4)                 concatenate, conjugate, cos, cosh, count, divide, equal, exp, filled,

```

```

13: (4)         getmask, greater, greater_equal, inner, isMaskedArray, less,
14: (4)         less_equal, log, log10, make_mask, masked, masked_array, masked_equal,
15: (4)         masked_greater, masked_greater_equal, masked_inside, masked_less,
16: (4)         masked_less_equal, masked_not_equal, masked_outside,
17: (4)         masked_print_option, masked_values, masked_where, maximum, minimum,
18: (4)         multiply, nomask, nonzero, not_equal, ones, outer, product, put, ravel,
19: (4)         repeat, resize, shape, sin, sinh, somettrue, sort, sqrt, subtract, sum,
20: (4)         take, tan, tanh, transpose, where, zeros,
21: (4)     )
22: (0)     from numpy.compat import pickle
23: (0)     pi = np.pi
24: (0)     def eq(v, w, msg=''):
25: (4)         result = allclose(v, w)
26: (4)         if not result:
27: (8)             print(f'Not eq:{msg}\n{v}\n----{w}')
28: (4)         return result
29: (0)     class TestMa:
30: (4)         def setup_method(self):
31: (8)             x = np.array([1., 1., 1., -2., pi/2.0, 4., 5., -10., 10., 1., 2., 3.])
32: (8)             y = np.array([5., 0., 3., 2., -1., -4., 0., -10., 10., 1., 0., 3.])
33: (8)             a10 = 10.
34: (8)             m1 = [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
35: (8)             m2 = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1]
36: (8)             xm = array(x, mask=m1)
37: (8)             ym = array(y, mask=m2)
38: (8)             z = np.array([-0.5, 0., 0.5, 0.8])
39: (8)             zm = array(z, mask=[0, 1, 0, 0])
40: (8)             xf = np.where(m1, 1e+20, x)
41: (8)             s = x.shape
42: (8)             xm.set_fill_value(1e+20)
43: (8)             self.d = (x, y, a10, m1, m2, xm, ym, z, zm, xf, s)
44: (4)         def test_testBasic1d(self):
45: (8)             (x, y, a10, m1, m2, xm, ym, z, zm, xf, s) = self.d
46: (8)             assert_(not isMaskedArray(x))
47: (8)             assert_(isMaskedArray(xm))
48: (8)             assert_equal(shape(xm), s)
49: (8)             assert_equal(xm.shape, s)
50: (8)             assert_equal(xm.dtype, x.dtype)
51: (8)             assert_equal(xm.size, reduce(lambda x, y: x * y, s))
52: (8)             assert_equal(count(xm), len(m1) - reduce(lambda x, y: x + y, m1))
53: (8)             assert_(eq(xm, xf))
54: (8)             assert_(eq(filled(xm, 1.e20), xf))
55: (8)             assert_(eq(x, xm))
56: (4)         @pytest.mark.parametrize("s", [(4, 3), (6, 2)])
57: (4)         def test_testBasic2d(self, s):
58: (8)             (x, y, a10, m1, m2, xm, ym, z, zm, xf, s) = self.d
59: (8)             x.shape = s
60: (8)             y.shape = s
61: (8)             xm.shape = s
62: (8)             ym.shape = s
63: (8)             xf.shape = s
64: (8)             assert_(not isMaskedArray(x))
65: (8)             assert_(isMaskedArray(xm))
66: (8)             assert_equal(shape(xm), s)
67: (8)             assert_equal(xm.shape, s)
68: (8)             assert_equal(xm.size, reduce(lambda x, y: x * y, s))
69: (8)             assert_equal(count(xm), len(m1) - reduce(lambda x, y: x + y, m1))
70: (8)             assert_(eq(xm, xf))
71: (8)             assert_(eq(filled(xm, 1.e20), xf))
72: (8)             assert_(eq(x, xm))
73: (4)         def test_testArithmetic(self):
74: (8)             (x, y, a10, m1, m2, xm, ym, z, zm, xf, s) = self.d
75: (8)             a2d = array([[1, 2], [0, 4]])
76: (8)             a2dm = masked_array(a2d, [[0, 0], [1, 0]])
77: (8)             assert_(eq(a2d * a2d, a2d * a2dm))
78: (8)             assert_(eq(a2d + a2d, a2d + a2dm))
79: (8)             assert_(eq(a2d - a2d, a2d - a2dm))
80: (8)             for s in [(12,), (4, 3), (2, 6)]:
81: (12)                 x = x.reshape(s)

```

```

82: (12)             y = y.reshape(s)
83: (12)             xm = xm.reshape(s)
84: (12)             ym = ym.reshape(s)
85: (12)             xf = xf.reshape(s)
86: (12)             assert_(eq(-x, -xm))
87: (12)             assert_(eq(x + y, xm + ym))
88: (12)             assert_(eq(x - y, xm - ym))
89: (12)             assert_(eq(x * y, xm * ym))
90: (12)             with np.errstate(divide='ignore', invalid='ignore'):
91: (16)                 assert_(eq(x / y, xm / ym))
92: (12)                 assert_(eq(a10 + y, a10 + ym))
93: (12)                 assert_(eq(a10 - y, a10 - ym))
94: (12)                 assert_(eq(a10 * y, a10 * ym))
95: (12)                 with np.errstate(divide='ignore', invalid='ignore'):
96: (16)                     assert_(eq(a10 / y, a10 / ym))
97: (12)                     assert_(eq(x + a10, xm + a10))
98: (12)                     assert_(eq(x - a10, xm - a10))
99: (12)                     assert_(eq(x * a10, xm * a10))
100: (12)                    assert_(eq(x / a10, xm / a10))
101: (12)                    assert_(eq(x ** 2, xm ** 2))
102: (12)                    assert_(eq(abs(x) ** 2.5, abs(xm) ** 2.5))
103: (12)                    assert_(eq(x ** y, xm ** ym))
104: (12)                    assert_(eq(np.add(x, y), add(xm, ym)))
105: (12)                    assert_(eq(np.subtract(x, y), subtract(xm, ym)))
106: (12)                    assert_(eq(np.multiply(x, y), multiply(xm, ym)))
107: (12)                    with np.errstate(divide='ignore', invalid='ignore'):
108: (16)                        assert_(eq(np.divide(x, y), divide(xm, ym)))
109: (4) def test_testMixedArithmetric(self):
110: (8)     na = np.array([1])
111: (8)     ma = array([1])
112: (8)     assert_(isinstance(na + ma, MaskedArray))
113: (8)     assert_(isinstance(ma + na, MaskedArray))
114: (4) def test_testUfuncs1(self):
115: (8)     (x, y, a10, m1, m2, xm, ym, z, zm, xf, s) = self.d
116: (8)     assert_(eq(np.cos(x), cos(xm)))
117: (8)     assert_(eq(np.cosh(x), cosh(xm)))
118: (8)     assert_(eq(np.sin(x), sin(xm)))
119: (8)     assert_(eq(np.sinh(x), sinh(xm)))
120: (8)     assert_(eq(np.tan(x), tan(xm)))
121: (8)     assert_(eq(np.tanh(x), tanh(xm)))
122: (8)     with np.errstate(divide='ignore', invalid='ignore'):
123: (12)         assert_(eq(np.sqrt(abs(x)), sqrt(xm)))
124: (12)         assert_(eq(np.log(abs(x)), log(xm)))
125: (12)         assert_(eq(np.log10(abs(x)), log10(xm)))
126: (8)     assert_(eq(np.exp(x), exp(xm)))
127: (8)     assert_(eq(np.arcsin(z), arcsin(zm)))
128: (8)     assert_(eq(np.arccos(z), arccos(zm)))
129: (8)     assert_(eq(np.arctan(z), arctan(zm)))
130: (8)     assert_(eq(np.arctan2(x, y), arctan2(xm, ym)))
131: (8)     assert_(eq(np.absolute(x), absolute(xm)))
132: (8)     assert_(eq(np.equal(x, y), equal(xm, ym)))
133: (8)     assert_(eq(np.not_equal(x, y), not_equal(xm, ym)))
134: (8)     assert_(eq(np.less(x, y), less(xm, ym)))
135: (8)     assert_(eq(np.greater(x, y), greater(xm, ym)))
136: (8)     assert_(eq(np.less_equal(x, y), less_equal(xm, ym)))
137: (8)     assert_(eq(np.greater_equal(x, y), greater_equal(xm, ym)))
138: (8)     assert_(eq(np.conjugate(x), conjugate(xm)))
139: (8)     assert_(eq(np.concatenate((x, y)), concatenate((xm, ym))))
140: (8)     assert_(eq(np.concatenate((x, y)), concatenate((x, y))))
141: (8)     assert_(eq(np.concatenate((x, y)), concatenate((xm, y))))
142: (8)     assert_(eq(np.concatenate((x, y, x)), concatenate((x, ym, x))))
143: (4) def test_xtestCount(self):
144: (8)     ott = array([0., 1., 2., 3.], mask=[1, 0, 0, 0])
145: (8)     assert_(count(ott).dtype.type is np.intp)
146: (8)     assert_equal(3, count(ott))
147: (8)     assert_equal(1, count(1))
148: (8)     assert_(eq(0, array(1, mask=[1])))
149: (8)     ott = ott.reshape((2, 2))
150: (8)     assert_(count(ott).dtype.type is np.intp)

```

```

151: (8) assert_(isinstance(count(ott, 0), np.ndarray))
152: (8) assert_(count(ott).dtype.type is np.intp)
153: (8) assert_(eq(3, count(ott)))
154: (8) assert_(getmask(count(ott, 0)) is nomask)
155: (8) assert_(eq([1, 2], count(ott, 0)))
156: (4) def test_testMinMax(self):
157: (8)     (x, y, a10, m1, m2, xm, ym, z, zm, xf, s) = self.d
158: (8)     xr = np.ravel(x) # max doesn't work if shaped
159: (8)     xmr = ravel(xm)
160: (8)     assert_(eq(max(xr), maximum.reduce(xmr)))
161: (8)     assert_(eq(min(xr), minimum.reduce(xmr)))
162: (4) def test_testAddSumProd(self):
163: (8)     (x, y, a10, m1, m2, xm, ym, z, zm, xf, s) = self.d
164: (8)     assert_(eq(np.add.reduce(x), add.reduce(x)))
165: (8)     assert_(eq(np.add.accumulate(x), add.accumulate(x)))
166: (8)     assert_(eq(4, sum(array(4), axis=0)))
167: (8)     assert_(eq(4, sum(array(4), axis=0)))
168: (8)     assert_(eq(np.sum(x, axis=0), sum(x, axis=0)))
169: (8)     assert_(eq(np.sum(filled(xm, 0), axis=0), sum(xm, axis=0)))
170: (8)     assert_(eq(np.sum(x, 0), sum(x, 0)))
171: (8)     assert_(eq(np.prod(x, axis=0), product(x, axis=0)))
172: (8)     assert_(eq(np.prod(x, 0), product(x, 0)))
173: (8)     assert_(eq(np.prod(filled(xm, 1), axis=0),
174: (27)                     product(xm, axis=0)))
175: (8) if len(s) > 1:
176: (12)     assert_(eq(np.concatenate((x, y), 1),
177: (31)                     concatenate((xm, ym), 1)))
178: (12)     assert_(eq(np.add.reduce(x, 1), add.reduce(x, 1)))
179: (12)     assert_(eq(np.sum(x, 1), sum(x, 1)))
180: (12)     assert_(eq(np.prod(x, 1), product(x, 1)))
181: (4) def test_testCI(self):
182: (8)     x1 = np.array([1, 2, 4, 3])
183: (8)     x2 = array(x1, mask=[1, 0, 0, 0])
184: (8)     x3 = array(x1, mask=[0, 1, 0, 1])
185: (8)     x4 = array(x1)
186: (8)     str(x2) # raises?
187: (8)     repr(x2) # raises?
188: (8)     assert_(eq(np.sort(x1), sort(x2, fill_value=0)))
189: (8)     assert_(type(x2[1]) is type(x1[1]))
190: (8)     assert_(x1[1] == x2[1])
191: (8)     assert_(x2[0] is masked)
192: (8)     assert_(eq(x1[2], x2[2]))
193: (8)     assert_(eq(x1[2:5], x2[2:5]))
194: (8)     assert_(eq(x1[:, :], x2[:, :]))
195: (8)     assert_(eq(x1[1:], x3[1:]))
196: (8)     x1[2] = 9
197: (8)     x2[2] = 9
198: (8)     assert_(eq(x1, x2))
199: (8)     x1[1:3] = 99
200: (8)     x2[1:3] = 99
201: (8)     assert_(eq(x1, x2))
202: (8)     x2[1] = masked
203: (8)     assert_(eq(x1, x2))
204: (8)     x2[1:3] = masked
205: (8)     assert_(eq(x1, x2))
206: (8)     x2[:] = x1
207: (8)     x2[1] = masked
208: (8)     assert_(allequal(getmask(x2), array([0, 1, 0, 0])))
209: (8)     x3[:] = masked_array([1, 2, 3, 4], [0, 1, 1, 0])
210: (8)     assert_(allequal(getmask(x3), array([0, 1, 1, 0])))
211: (8)     x4[:] = masked_array([1, 2, 3, 4], [0, 1, 1, 0])
212: (8)     assert_(allequal(getmask(x4), array([0, 1, 1, 0])))
213: (8)     assert_(allequal(x4, array([1, 2, 3, 4])))
214: (8)     x1 = np.arange(5) * 1.0
215: (8)     x2 = masked_values(x1, 3.0)
216: (8)     assert_(eq(x1, x2))
217: (8)     assert_(allequal(array([0, 0, 0, 1, 0], MaskType), x2.mask))
218: (8)     assert_(eq(3.0, x2.fill_value))
219: (8)     x1 = array([1, 'hello', 2, 3], object)

```

```

220: (8)           x2 = np.array([1, 'hello', 2, 3], object)
221: (8)           s1 = x1[1]
222: (8)           s2 = x2[1]
223: (8)           assert_equal(type(s2), str)
224: (8)           assert_equal(type(s1), str)
225: (8)           assert_equal(s1, s2)
226: (8)           assert_(x1[1:1].shape == (0,))
227: (4) def test_testCopySize(self):
228: (8)     n = [0, 0, 1, 0, 0]
229: (8)     m = make_mask(n)
230: (8)     m2 = make_mask(m)
231: (8)     assert_(m is m2)
232: (8)     m3 = make_mask(m, copy=True)
233: (8)     assert_(m is not m3)
234: (8)     x1 = np.arange(5)
235: (8)     y1 = array(x1, mask=m)
236: (8)     assert_(y1._data is not x1)
237: (8)     assert_(allequal(x1, y1._data))
238: (8)     assert_(y1._mask is m)
239: (8)     y1a = array(y1, copy=0)
240: (8)     assert_(y1a._mask.__array_interface__ ==
241: (16)                 y1._mask.__array_interface__)
242: (8)     y2 = array(x1, mask=m3, copy=0)
243: (8)     assert_(y2._mask is m3)
244: (8)     assert_(y2[2] is masked)
245: (8)     y2[2] = 9
246: (8)     assert_(y2[2] is not masked)
247: (8)     assert_(y2._mask is m3)
248: (8)     assert_(allequal(y2.mask, 0))
249: (8)     y2a = array(x1, mask=m, copy=1)
250: (8)     assert_(y2a._mask is not m)
251: (8)     assert_(y2a[2] is masked)
252: (8)     y2a[2] = 9
253: (8)     assert_(y2a[2] is not masked)
254: (8)     assert_(y2a._mask is not m)
255: (8)     assert_(allequal(y2a.mask, 0))
256: (8)     y3 = array(x1 * 1.0, mask=m)
257: (8)     assert_(filled(y3).dtype is (x1 * 1.0).dtype)
258: (8)     x4 = arange(4)
259: (8)     x4[2] = masked
260: (8)     y4 = resize(x4, (8,))
261: (8)     assert_(eq(concatenate([x4, x4]), y4))
262: (8)     assert_(eq(getmask(y4), [0, 0, 1, 0, 0, 0, 1, 0]))
263: (8)     y5 = repeat(x4, (2, 2, 2, 2), axis=0)
264: (8)     assert_(eq(y5, [0, 0, 1, 1, 2, 2, 3, 3]))
265: (8)     y6 = repeat(x4, 2, axis=0)
266: (8)     assert_(eq(y5, y6))
267: (4) def test_testPut(self):
268: (8)     d = arange(5)
269: (8)     n = [0, 0, 0, 1, 1]
270: (8)     m = make_mask(n)
271: (8)     m2 = m.copy()
272: (8)     x = array(d, mask=m)
273: (8)     assert_(x[3] is masked)
274: (8)     assert_(x[4] is masked)
275: (8)     x[[1, 4]] = [10, 40]
276: (8)     assert_(x._mask is m)
277: (8)     assert_(x[3] is masked)
278: (8)     assert_(x[4] is not masked)
279: (8)     assert_(eq(x, [0, 10, 2, -1, 40]))
280: (8)     x = array(d, mask=m2, copy=True)
281: (8)     x.put([0, 1, 2], [-1, 100, 200])
282: (8)     assert_(x._mask is not m2)
283: (8)     assert_(x[3] is masked)
284: (8)     assert_(x[4] is masked)
285: (8)     assert_(eq(x, [-1, 100, 200, 0, 0]))
286: (4) def test_testPut2(self):
287: (8)     d = arange(5)
288: (8)     x = array(d, mask=[0, 0, 0, 0, 0])

```

```

289: (8)             z = array([10, 40], mask=[1, 0])
290: (8)             assert_(x[2] is not masked)
291: (8)             assert_(x[3] is not masked)
292: (8)             x[2:4] = z
293: (8)             assert_(x[2] is masked)
294: (8)             assert_(x[3] is not masked)
295: (8)             assert_(eq(x, [0, 1, 10, 40, 4]))
296: (8)             d = arange(5)
297: (8)             x = array(d, mask=[0, 0, 0, 0, 0])
298: (8)             y = x[2:4]
299: (8)             z = array([10, 40], mask=[1, 0])
300: (8)             assert_(x[2] is not masked)
301: (8)             assert_(x[3] is not masked)
302: (8)             y[:] = z
303: (8)             assert_(y[0] is masked)
304: (8)             assert_(y[1] is not masked)
305: (8)             assert_(eq(y, [10, 40]))
306: (8)             assert_(x[2] is masked)
307: (8)             assert_(x[3] is not masked)
308: (8)             assert_(eq(x, [0, 1, 10, 40, 4]))
309: (4)             def test_testMaPut(self):
310: (8)                 (x, y, a10, m1, m2, xm, ym, zm, xf, s) = self.d
311: (8)                 m = [1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1]
312: (8)                 i = np.nonzero(m)[0]
313: (8)                 put(ym, i, zm)
314: (8)                 assert_(all(take(ym, i, axis=0) == zm))
315: (4)             def test_testOddFeatures(self):
316: (8)                 x = arange(20)
317: (8)                 x = x.reshape(4, 5)
318: (8)                 x.flat[5] = 12
319: (8)                 assert_(x[1, 0] == 12)
320: (8)                 z = x + 10j * x
321: (8)                 assert_(eq(z.real, x))
322: (8)                 assert_(eq(z.imag, 10 * x))
323: (8)                 assert_(eq((z * conjugate(z)).real, 101 * x * x))
324: (8)                 z.imag[...] = 0.0
325: (8)                 x = arange(10)
326: (8)                 x[3] = masked
327: (8)                 assert_(str(x[3]) == str(masked))
328: (8)                 c = x >= 8
329: (8)                 assert_(count(where(c, masked, masked)) == 0)
330: (8)                 assert_(shape(where(c, masked, masked)) == c.shape)
331: (8)                 z = where(c, x, masked)
332: (8)                 assert_(z.dtype is x.dtype)
333: (8)                 assert_(z[3] is masked)
334: (8)                 assert_(z[4] is masked)
335: (8)                 assert_(z[7] is masked)
336: (8)                 assert_(z[8] is not masked)
337: (8)                 assert_(z[9] is not masked)
338: (8)                 assert_(eq(x, z))
339: (8)                 z = where(c, masked, x)
340: (8)                 assert_(z.dtype is x.dtype)
341: (8)                 assert_(z[3] is masked)
342: (8)                 assert_(z[4] is not masked)
343: (8)                 assert_(z[7] is not masked)
344: (8)                 assert_(z[8] is masked)
345: (8)                 assert_(z[9] is masked)
346: (8)                 z = masked_where(c, x)
347: (8)                 assert_(z.dtype is x.dtype)
348: (8)                 assert_(z[3] is masked)
349: (8)                 assert_(z[4] is not masked)
350: (8)                 assert_(z[7] is not masked)
351: (8)                 assert_(z[8] is masked)
352: (8)                 assert_(z[9] is masked)
353: (8)                 assert_(eq(x, z))
354: (8)                 x = array([1., 2., 3., 4., 5.])
355: (8)                 c = array([1, 1, 1, 0, 0])
356: (8)                 x[2] = masked
357: (8)                 z = where(c, x, -x)

```

```

358: (8)             assert_(eq(z, [1., 2., 0., -4., -5]))
359: (8)             c[0] = masked
360: (8)             z = where(c, x, -x)
361: (8)             assert_(eq(z, [1., 2., 0., -4., -5]))
362: (8)             assert_(z[0] is masked)
363: (8)             assert_(z[1] is not masked)
364: (8)             assert_(z[2] is masked)
365: (8)             assert_(eq(masked_where(greater(x, 2), x), masked_greater(x, 2)))
366: (8)             assert_(eq(masked_where(greater_equal(x, 2), x),
367: (19)                 masked_greater_equal(x, 2)))
368: (8)             assert_(eq(masked_where(less(x, 2), x), masked_less(x, 2)))
369: (8)             assert_(eq(masked_where(less_equal(x, 2), x), masked_less_equal(x,
2)))
370: (8)             assert_(eq(masked_where(not_equal(x, 2), x), masked_not_equal(x, 2)))
371: (8)             assert_(eq(masked_where(equal(x, 2), x), masked_equal(x, 2)))
372: (8)             assert_(eq(masked_where(not_equal(x, 2), x), masked_not_equal(x, 2)))
373: (8)             assert_(eq(masked_inside(list(range(5)), 1, 3), [0, 199, 199, 199,
4]))
374: (8)             assert_(eq(masked_outside(list(range(5)), 1, 3), [199, 1, 2, 3, 199]))
375: (8)             assert_(eq(masked_inside(array(list(range(5))),
376: (39)                 mask=[1, 0, 0, 0, 0]), 1, 3).mask,
377: (19)                 [1, 1, 1, 1, 0]))
378: (8)             assert_(eq(masked_outside(array(list(range(5))),
379: (40)                 mask=[0, 1, 0, 0, 0]), 1, 3).mask,
380: (19)                 [1, 1, 0, 0, 1]))
381: (8)             assert_(eq(masked_equal(array(list(range(5))),
382: (38)                 mask=[1, 0, 0, 0, 0]), 2).mask,
383: (19)                 [1, 0, 1, 0, 0]))
384: (8)             assert_(eq(masked_not_equal(array([2, 2, 1, 2, 1]),
385: (42)                 mask=[1, 0, 0, 0, 0]), 2).mask,
386: (19)                 [1, 0, 1, 0, 1)))
387: (8)             assert_(eq(masked_where([1, 1, 0, 0, 0], [1, 2, 3, 4, 5]),
388: (19)                 [99, 99, 3, 4, 5]))
389: (8)             atest = ones((10, 10, 10), dtype=np.float32)
390: (8)             btest = zeros(atest.shape, MaskType)
391: (8)             ctest = masked_where(btest, atest)
392: (8)             assert_(eq(atest, ctest))
393: (8)             z = choose(c, (-x, x))
394: (8)             assert_(eq(z, [1., 2., 0., -4., -5]))
395: (8)             assert_(z[0] is masked)
396: (8)             assert_(z[1] is not masked)
397: (8)             assert_(z[2] is masked)
398: (8)             x = arange(6)
399: (8)             x[5] = masked
400: (8)             y = arange(6) * 10
401: (8)             y[2] = masked
402: (8)             c = array([1, 1, 1, 0, 0, 0], mask=[1, 0, 0, 0, 0, 0])
403: (8)             cm = c.filled(1)
404: (8)             z = where(c, x, y)
405: (8)             zm = where(cm, x, y)
406: (8)             assert_(eq(z, zm))
407: (8)             assert_(getmask(zm) is nomask)
408: (8)             assert_(eq(zm, [0, 1, 2, 30, 40, 50]))
409: (8)             z = where(c, masked, 1)
410: (8)             assert_(eq(z, [99, 99, 99, 1, 1, 1]))
411: (8)             z = where(c, 1, masked)
412: (8)             assert_(eq(z, [99, 1, 1, 99, 99, 99]))
413: (4)             def test_testMinMax2(self):
414: (8)                 assert_(eq(minimum([1, 2, 3], [4, 0, 9]), [1, 0, 3]))
415: (8)                 assert_(eq(maximum([1, 2, 3], [4, 0, 9]), [4, 2, 9]))
416: (8)                 x = arange(5)
417: (8)                 y = arange(5) - 2
418: (8)                 x[3] = masked
419: (8)                 y[0] = masked
420: (8)                 assert_(eq(minimum(x, y), where(less(x, y), x, y)))
421: (8)                 assert_(eq(maximum(x, y), where(greater(x, y), x, y)))
422: (8)                 assert_(minimum.reduce(x) == 0)
423: (8)                 assert_(maximum.reduce(x) == 4)
424: (4)             def test_testTakeTransposeInnerOuter(self):

```

```

425: (8)          x = arange(24)
426: (8)          y = np.arange(24)
427: (8)          x[5:6] = masked
428: (8)          x = x.reshape(2, 3, 4)
429: (8)          y = y.reshape(2, 3, 4)
430: (8)          assert_(eq(np.transpose(y, (2, 0, 1)), transpose(x, (2, 0, 1))))
431: (8)          assert_(eq(np.take(y, (2, 0, 1), 1), take(x, (2, 0, 1), 1)))
432: (8)          assert_(eq(np.inner(filled(x, 0), filled(y, 0)),
433: (19)                  inner(x, y)))
434: (8)          assert_(eq(np.outer(filled(x, 0), filled(y, 0)),
435: (19)                  outer(x, y)))
436: (8)          y = array(['abc', 1, 'def', 2, 3], object)
437: (8)          y[2] = masked
438: (8)          t = take(y, [0, 3, 4])
439: (8)          assert_(t[0] == 'abc')
440: (8)          assert_(t[1] == 2)
441: (8)          assert_(t[2] == 3)
442: (4)          def test_testInplace(self):
443: (8)          y = arange(10)
444: (8)          x = arange(10)
445: (8)          xm = arange(10)
446: (8)          xm[2] = masked
447: (8)          x += 1
448: (8)          assert_(eq(x, y + 1))
449: (8)          xm += 1
450: (8)          assert_(eq(x, y + 1))
451: (8)          x = arange(10)
452: (8)          xm = arange(10)
453: (8)          xm[2] = masked
454: (8)          x -= 1
455: (8)          assert_(eq(x, y - 1))
456: (8)          xm -= 1
457: (8)          assert_(eq(xm, y - 1))
458: (8)          x = arange(10) * 1.0
459: (8)          xm = arange(10) * 1.0
460: (8)          xm[2] = masked
461: (8)          x *= 2.0
462: (8)          assert_(eq(x, y * 2))
463: (8)          xm *= 2.0
464: (8)          assert_(eq(xm, y * 2))
465: (8)          x = arange(10) * 2
466: (8)          xm = arange(10)
467: (8)          xm[2] = masked
468: (8)          x // 2
469: (8)          assert_(eq(x, y))
470: (8)          xm // 2
471: (8)          assert_(eq(x, y))
472: (8)          x = arange(10) * 1.0
473: (8)          xm = arange(10) * 1.0
474: (8)          xm[2] = masked
475: (8)          x /= 2.0
476: (8)          assert_(eq(x, y / 2.0))
477: (8)          xm /= arange(10)
478: (8)          assert_(eq(xm, ones((10,))))
479: (8)          x = arange(10).astype(np.float32)
480: (8)          xm = arange(10)
481: (8)          xm[2] = masked
482: (8)          x += 1.
483: (8)          assert_(eq(x, y + 1.))
484: (4)          def test_testPickle(self):
485: (8)          x = arange(12)
486: (8)          x[4:10:2] = masked
487: (8)          x = x.reshape(4, 3)
488: (8)          for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
489: (12)              s = pickle.dumps(x, protocol=proto)
490: (12)              y = pickle.loads(s)
491: (12)              assert_(eq(x, y))
492: (4)          def test_testMasked(self):
493: (8)              xx = arange(6)

```

```

494: (8)             xx[1] = masked
495: (8)             assert_(str(masked) == '--')
496: (8)             assert_(xx[1] is masked)
497: (8)             assert_equal(filled(xx[1], 0), 0)
498: (4)             def test_testAverage1(self):
499: (8)                 ott = array([0., 1., 2., 3.], mask=[1, 0, 0, 0])
500: (8)                 assert_(eq(2.0, average(ott, axis=0)))
501: (8)                 assert_(eq(2.0, average(ott, weights=[1., 1., 2., 1.])))
502: (8)                 result, wts = average(ott, weights=[1., 1., 2., 1.], returned=True)
503: (8)                 assert_(eq(2.0, result))
504: (8)                 assert_(wts == 4.0)
505: (8)                 ott[:] = masked
506: (8)                 assert_(average(ott, axis=0) is masked)
507: (8)                 ott = array([0., 1., 2., 3.], mask=[1, 0, 0, 0])
508: (8)                 ott = ott.reshape(2, 2)
509: (8)                 ott[:, 1] = masked
510: (8)                 assert_(eq(average(ott, axis=0), [2.0, 0.0]))
511: (8)                 assert_(average(ott, axis=1)[0] is masked)
512: (8)                 assert_(eq([2., 0.], average(ott, axis=0)))
513: (8)                 result, wts = average(ott, axis=0, returned=True)
514: (8)                 assert_(eq(wts, [1., 0.]))
515: (4)             def test_testAverage2(self):
516: (8)                 w1 = [0, 1, 1, 1, 1, 0]
517: (8)                 w2 = [[0, 1, 1, 1, 1, 0], [1, 0, 0, 0, 0, 1]]
518: (8)                 x = arange(6)
519: (8)                 assert_(allclose(average(x, axis=0), 2.5))
520: (8)                 assert_(allclose(average(x, axis=0, weights=w1), 2.5))
521: (8)                 y = array([arange(6), 2.0 * arange(6)])
522: (8)                 assert_(allclose(average(y, None),
523: (33)                               np.add.reduce(np.arange(6)) * 3. / 12.))
524: (8)                 assert_(allclose(average(y, axis=0), np.arange(6) * 3. / 2.))
525: (8)                 assert_(allclose(average(y, axis=1),
526: (33)                               [average(x, axis=0), average(x,
527: (8)                               axis=0)*2.0]))
528: (8)                 assert_(allclose(average(y, None, weights=w2), 20. / 6.))
529: (33)                 assert_(allclose(average(y, axis=0, weights=w2),
530: (8)                               [0., 1., 2., 3., 4., 10.]))
531: (33)                 assert_(allclose(average(y, axis=1),
532: (8)                               [average(x, axis=0), average(x,
533: (8)                               m1 = zeros(6)
534: (8)                               m2 = [0, 0, 1, 1, 0, 0]
535: (8)                               m3 = [[0, 0, 1, 1, 0, 0], [0, 1, 1, 1, 1, 0]]
536: (8)                               m4 = ones(6)
537: (8)                               m5 = [0, 1, 1, 1, 1, 1]
538: (8)                               assert_(allclose(average(masked_array(x, m1), axis=0), 2.5))
539: (8)                               assert_(allclose(average(masked_array(x, m2), axis=0), 2.5))
540: (8)                               assert_(average(masked_array(x, m4), axis=0) is masked)
541: (8)                               assert_equal(average(masked_array(x, m5), axis=0), 0.0)
542: (8)                               assert_equal(count(average(masked_array(x, m4), axis=0)), 0)
543: (8)                               z = masked_array(y, m3)
544: (8)                               assert_(allclose(average(z, None), 20. / 6.))
545: (33)                 assert_(allclose(average(z, axis=0),
546: (8)                               [0., 1., 99., 99., 4.0, 7.5]))
547: (8)                 assert_(allclose(average(z, axis=1), [2.5, 5.0]))
548: (33)                 assert_(allclose(average(z, axis=0, weights=w2),
549: (8)                               [0., 1., 99., 99., 4.0, 10.0]))
550: (8)                 a = arange(6)
551: (8)                 b = arange(6) * 3
552: (8)                 r1, w1 = average([[a, b], [b, a]], axis=1, returned=True)
553: (8)                 assert_equal(shape(r1), shape(w1))
554: (8)                 assert_equal(r1.shape, w1.shape)
555: (8)                 returned=True)
556: (8)                 r2, w2 = average(ones((2, 2, 3)), axis=0, weights=[3, 1],
557: (8)                               assert_equal(shape(w2), shape(r2))
558: (8)                               r2, w2 = average(ones((2, 2, 3)), returned=True)
559: (8)                               assert_equal(shape(w2), shape(r2))
560: (8)                               r2, w2 = average(ones((2, 2, 3)), weights=ones((2, 2, 3)),
561: (8)                               returned=True)

```

```

559: (8)             assert_(shape(w2) == shape(r2))
560: (8)             a2d = array([[1, 2], [0, 4]], float)
561: (8)             a2dm = masked_array(a2d, [[0, 0], [1, 0]])
562: (8)             a2da = average(a2d, axis=0)
563: (8)             assert_(eq(a2da, [0.5, 3.0]))
564: (8)             a2dma = average(a2dm, axis=0)
565: (8)             assert_(eq(a2dma, [1.0, 3.0]))
566: (8)             a2dma = average(a2dm, axis=None)
567: (8)             assert_(eq(a2dma, 7. / 3.))
568: (8)             a2dma = average(a2dm, axis=1)
569: (8)             assert_(eq(a2dma, [1.5, 4.0]))
570: (4)             def test_toPython(self):
571: (8)                 assert_equal(1, int(array(1)))
572: (8)                 assert_equal(1.0, float(array(1)))
573: (8)                 assert_equal(1, int(array([[1]])))
574: (8)                 assert_equal(1.0, float(array([[1]])))
575: (8)                 assert_raises(TypeError, float, array([1, 1]))
576: (8)                 assert_raises(ValueError, bool, array([0, 1]))
577: (8)                 assert_raises(ValueError, bool, array([0, 0], mask=[0, 1]))
578: (4)             def test_scalarArithmetic(self):
579: (8)                 xm = array(0, mask=1)
580: (8)                 with np.errstate(divide='ignore'):
581: (12)                     assert_((1 / array(0)).mask)
582: (8)                     assert_((1 + xm).mask)
583: (8)                     assert_((-xm).mask)
584: (8)                     assert_((-xm).mask)
585: (8)                     assert_(maximum(xm, xm).mask)
586: (8)                     assert_(minimum(xm, xm).mask)
587: (8)                     assert_(xm.filled().dtype is xm._data.dtype)
588: (8)                     x = array(0, mask=0)
589: (8)                     assert_(x.filled() == x._data)
590: (8)                     assert_equal(str(xm), str(masked_print_option))
591: (4)             def test_arrayMethods(self):
592: (8)                 a = array([1, 3, 2])
593: (8)                 assert_(eq(a.any(), a._data.any()))
594: (8)                 assert_(eq(a.all(), a._data.all()))
595: (8)                 assert_(eq(a.argmax(), a._data.argmax()))
596: (8)                 assert_(eq(a.argmin(), a._data.argmin()))
597: (8)                 assert_(eq(a.choose(0, 1, 2, 3, 4),
598: (27)                               a._data.choose(0, 1, 2, 3, 4)))
599: (8)                 assert_(eq(a.compress([1, 0, 1]), a._data.compress([1, 0, 1])))
600: (8)                 assert_(eq(a.conj(), a._data.conj()))
601: (8)                 assert_(eq(a.conjugate(), a._data.conjugate()))
602: (8)                 m = array([[1, 2], [3, 4]])
603: (8)                 assert_(eq(m.diagonal(), m._data.diagonal()))
604: (8)                 assert_(eq(a.sum(), a._data.sum()))
605: (8)                 assert_(eq(a.take([1, 2]), a._data.take([1, 2])))
606: (8)                 assert_(eq(m.transpose(), m._data.transpose()))
607: (4)             def test_arrayAttributes(self):
608: (8)                 a = array([1, 3, 2])
609: (8)                 assert_equal(a.ndim, 1)
610: (4)             def test_API(self):
611: (8)                 assert_(not [m for m in dir(np.ndarray)
612: (21)                               if m not in dir(MaskedArray) and
613: (21)                               not m.startswith('_')])
614: (4)             def test_singleElementSubscript(self):
615: (8)                 a = array([1, 3, 2])
616: (8)                 b = array([1, 3, 2], mask=[1, 0, 1])
617: (8)                 assert_equal(a[0].shape, ())
618: (8)                 assert_equal(b[0].shape, ())
619: (8)                 assert_equal(b[1].shape, ())
620: (4)             def test_assignment_by_condition(self):
621: (8)                 a = array([1, 2, 3, 4], mask=[1, 0, 1, 0])
622: (8)                 c = a >= 3
623: (8)                 a[c] = 5
624: (8)                 assert_(a[2] is masked)
625: (4)             def test_assignment_by_condition_2(self):
626: (8)                 a = masked_array([0, 1], mask=[False, False])
627: (8)                 b = masked_array([0, 1], mask=[True, True])

```

```

628: (8)             mask = a < 1
629: (8)             b[mask] = a[mask]
630: (8)             expected_mask = [False, True]
631: (8)             assert_equal(b.mask, expected_mask)
632: (0)             class TestUfuncs:
633: (4)             def setup_method(self):
634: (8)                 self.d = (array([1.0, 0, -1, pi / 2] * 2, mask=[0, 1] + [0] * 6),
635: (18)                         array([1.0, 0, -1, pi / 2] * 2, mask=[1, 0] + [0] * 6),)
636: (4)             def test_testUfuncRegression(self):
637: (8)                 f_invalid_ignore = [
638: (12)                     'sqrt', 'arctanh', 'arcsin', 'arccos',
639: (12)                     'arccosh', 'arctanh', 'log', 'log10', 'divide',
640: (12)                     'true_divide', 'floor_divide', 'remainder', 'fmod']
641: (8)                 for f in ['sqrt', 'log', 'log10', 'exp', 'conjugate',
642: (18)                         'sin', 'cos', 'tan',
643: (18)                         'arcsin', 'arccos', 'arctan',
644: (18)                         'sinh', 'cosh', 'tanh',
645: (18)                         'arcsinh',
646: (18)                         'arccosh',
647: (18)                         'arctanh',
648: (18)                         'absolute', 'fabs', 'negative',
649: (18)                         'floor', 'ceil',
650: (18)                         'logical_not',
651: (18)                         'add', 'subtract', 'multiply',
652: (18)                         'divide', 'true_divide', 'floor_divide',
653: (18)                         'remainder', 'fmod', 'hypot', 'arctan2',
654: (18)                         'equal', 'not_equal', 'less_equal', 'greater_equal',
655: (18)                         'less', 'greater',
656: (18)                         'logical_and', 'logical_or', 'logical_xor']:
657: (12)                 try:
658: (16)                     uf = getattr(umath, f)
659: (12)                 except AttributeError:
660: (16)                     uf = getattr(fromnumeric, f)
661: (12)                     mf = getattr(np.ma, f)
662: (12)                     args = self.d[:uf.nin]
663: (12)                     with np.errstate():
664: (16)                         if f in f_invalid_ignore:
665: (20)                             np.seterr(invalid='ignore')
666: (16)                         if f in ['arctanh', 'log', 'log10']:
667: (20)                             np.seterr(divide='ignore')
668: (16)                         ur = uf(*args)
669: (16)                         mr = mf(*args)
670: (12)                         assert_(eq(ur.filled(0), mr.filled(0), f))
671: (12)                         assert_(eqmask(ur.mask, mr.mask))
672: (4)             def test_reduce(self):
673: (8)                 a = self.d[0]
674: (8)                 assert_(not alltrue(a, axis=0))
675: (8)                 assert_(sometrue(a, axis=0))
676: (8)                 assert_equal(sum(a[:3], axis=0), 0)
677: (8)                 assert_equal(product(a, axis=0), 0)
678: (4)             def test_minmax(self):
679: (8)                 a = arange(1, 13).reshape(3, 4)
680: (8)                 amask = masked_where(a < 5, a)
681: (8)                 assert_equal(amask.max(), a.max())
682: (8)                 assert_equal(amask.min(), 5)
683: (8)                 assert_((amask.max(0) == a.max(0)).all())
684: (8)                 assert_((amask.min(0) == [5, 6, 7, 8]).all())
685: (8)                 assert_(amask.max(1)[0].mask)
686: (8)                 assert_(amask.min(1)[0].mask)
687: (4)             def test_nonzero(self):
688: (8)                 for t in "?bhilqpBHILQPfdgFDGO":
689: (12)                     x = array([1, 0, 2, 0], mask=[0, 0, 1, 1])
690: (12)                     assert_(eq(nonzero(x), [0]))
691: (0)             class TestArrayMethods:
692: (4)             def setup_method(self):
693: (8)                 x = np.array([8.375, 7.545, 8.828, 8.5, 1.757, 5.928,
694: (22)                         8.43, 7.78, 9.865, 5.878, 8.979, 4.732,
695: (22)                         3.012, 6.022, 5.095, 3.116, 5.238, 3.957,
696: (22)                         6.04, 9.63, 7.712, 3.382, 4.489, 6.479,

```

```

697: (22)                                     7.189, 9.645, 5.395, 4.961, 9.894, 2.893,
698: (22)                                     7.357, 9.828, 6.272, 3.758, 6.693, 0.993])
699: (8)
700: (8)
701: (8)
702: (22)
703: (22)
704: (22)
705: (22)
706: (22)
707: (8)
708: (8)
709: (8)
710: (8)
711: (4)
712: (8)
713: (8)
714: (8)
715: (8)
716: (27)
717: (43)
718: (4)
719: (8)
720: (8)
721: (8)
722: (8)
723: (8)
724: (4)
725: (8)
726: (8)
727: (8)
728: (8)
729: (8)
730: (8)
731: (12)
732: (8)
733: (12)
734: (8)
735: (8)
736: (4)
737: (8)
738: (8)
739: (8)
740: (8)
741: (8)
742: (4)
743: (8)
744: (8)
745: (8)
746: (8)
747: (8)
748: (4)
749: (8)
750: (8)
751: (8)
752: (8)
753: (8)
754: (4)
755: (8)
756: (8)
757: (8)
758: (8)
759: (8)
760: (8)
761: (8)
762: (12)
763: (12)
764: (12)
765: (31)

         X = x.reshape(6, 6)
         XX = x.reshape(3, 2, 2, 3)
         m = np.array([0, 1, 0, 1, 0, 0,
                       1, 0, 1, 1, 0, 1,
                       0, 0, 0, 1, 0, 1,
                       0, 0, 0, 1, 1, 1,
                       1, 0, 0, 1, 0, 0,
                       0, 0, 1, 0, 1, 0])
         mx = array(data=x, mask=m)
         mX = array(data=X, mask=m.reshape(X.shape))
         mXX = array(data=XX, mask=m.reshape(XX.shape))
         self.d = (x, X, XX, m, mx, mX, mXX)

def test_trace(self):
    (x, X, XX, m, mx, mX, mXX,) = self.d
    mXdiag = mX.diagonal()
    assert_equal(mX.trace(), mX.diagonal().compressed().sum())
    assert_(eq(mX.trace(),
              X.trace() - sum(mXdiag.mask * X.diagonal(),
                               axis=0)))

def test_clip(self):
    (x, X, XX, m, mx, mX, mXX,) = self.d
    clipped = mx.clip(2, 8)
    assert_(eq(clipped.mask, mx.mask))
    assert_(eq(clipped._data, x.clip(2, 8)))
    assert_(eq(clipped._data, mx._data.clip(2, 8)))

def test_ptp(self):
    (x, X, XX, m, mx, mX, mXX,) = self.d
    (n, m) = X.shape
    assert_equal(mx.ptp(), mx.compressed().ptp())
    rows = np.zeros(n, np.float_)
    cols = np.zeros(m, np.float_)
    for k in range(m):
        cols[k] = mX[:, k].compressed().ptp()
    for k in range(n):
        rows[k] = mX[k].compressed().ptp()
    assert_(eq(mX.ptp(0), cols))
    assert_(eq(mX.ptp(1), rows))

def test_swapaxes(self):
    (x, X, XX, m, mx, mX, mXX,) = self.d
    mXswapped = mX.swapaxes(0, 1)
    assert_(eq(mXswapped[-1], mX[:, -1]))
    mXXswapped = mXX.swapaxes(0, 2)
    assert_equal(mXXswapped.shape, (2, 2, 3, 3))

def test_cumprod(self):
    (x, X, XX, m, mx, mX, mXX,) = self.d
    mXcp = mX.cumprod(0)
    assert_(eq(mXcp._data, mX.filled(1).cumprod(0)))
    mXcp = mX.cumprod(1)
    assert_(eq(mXcp._data, mX.filled(1).cumprod(1)))

def test_cumsum(self):
    (x, X, XX, m, mx, mX, mXX,) = self.d
    mXcp = mX.cumsum(0)
    assert_(eq(mXcp._data, mX.filled(0).cumsum(0)))
    mXcp = mX.cumsum(1)
    assert_(eq(mXcp._data, mX.filled(0).cumsum(1)))

def test_varstd(self):
    (x, X, XX, m, mx, mX, mXX,) = self.d
    assert_(eq(mX.var(axis=None), mX.compressed().var()))
    assert_(eq(mX.std(axis=None), mX.compressed().std()))
    assert_(eq(mXX.var(axis=3).shape, XX.var(axis=3).shape))
    assert_(eq(mX.var().shape, X.var().shape))
    (mXvar0, mXvar1) = (mX.var(axis=0), mX.var(axis=1))
    for k in range(6):
        assert_(eq(mXvar1[k], mX[k].compressed().var()))
        assert_(eq(mXvar0[k], mX[:, k].compressed().var()))
        assert_(eq(np.sqrt(mXvar0[k]),
                  mX[:, k].compressed().std()))

```

```

766: (0)         def eqmask(m1, m2):
767: (4)             if m1 is nomask:
768: (8)                 return m2 is nomask
769: (4)             if m2 is nomask:
770: (8)                 return m1 is nomask
771: (4)             return (m1 == m2).all()

```

---

File 273 - test\_regression.py:

```

1: (0)             import numpy as np
2: (0)             from numpy.testing import (
3: (4)                 assert_, assert_array_equal, assert_allclose, suppress_warnings
4: (4)             )
5: (0)             class TestRegression:
6: (4)                 def test_masked_array_create(self):
7: (8)                     x = np.ma.masked_array([0, 1, 2, 3, 0, 4, 5, 6],
8: (31)                         mask=[0, 0, 0, 1, 1, 1, 0, 0])
9: (8)                     assert_array_equal(np.ma.nonzero(x), [[1, 2, 6, 7]])
10: (4)                def test_masked_array(self):
11: (8)                    np.ma.array(1, mask=[1])
12: (4)                def test_mem_masked_where(self):
13: (8)                    from numpy.ma import masked_where, MaskType
14: (8)                    a = np.zeros((1, 1))
15: (8)                    b = np.zeros(a.shape, MaskType)
16: (8)                    c = masked_where(b, a)
17: (8)                    a-c
18: (4)                def test_masked_array_multiply(self):
19: (8)                    a = np.ma.zeros((4, 1))
20: (8)                    a[2, 0] = np.ma.masked
21: (8)                    b = np.zeros((4, 2))
22: (8)                    a*b
23: (8)                    b*a
24: (4)                def test_masked_array_repeat(self):
25: (8)                    np.ma.array([1], mask=False).repeat(10)
26: (4)                def test_masked_array_repr_unicode(self):
27: (8)                    repr(np.ma.array("Unicode"))
28: (4)                def test_atleast_2d(self):
29: (8)                    a = np.ma.masked_array([0.0, 1.2, 3.5], mask=[False, True, False])
30: (8)                    b = np.atleast_2d(a)
31: (8)                    assert_(a.mask.ndim == 1)
32: (8)                    assert_(b.mask.ndim == 2)
33: (4)                def test_set_fill_value_unicode_py3(self):
34: (8)                    a = np.ma.masked_array(['a', 'b', 'c'], mask=[1, 0, 0])
35: (8)                    a.fill_value = 'X'
36: (8)                    assert_(a.fill_value == 'X')
37: (4)                def test_var_sets_maskedarray_scalar(self):
38: (8)                    a = np.ma.array(np.arange(5), mask=True)
39: (8)                    mout = np.ma.array(-1, dtype=float)
40: (8)                    a.var(out=mout)
41: (8)                    assert_(mout._data == 0)
42: (4)                def test_ddof_corrcoef(self):
43: (8)                    x = np.ma.masked_equal([1, 2, 3, 4, 5], 4)
44: (8)                    y = np.array([2, 2.5, 3.1, 3, 5])
45: (8)                    with suppress_warnings() as sup:
46: (12)                        sup.filter(DeprecationWarning, "bias and ddof have no effect")
47: (12)                        r0 = np.ma.corrcoef(x, y, ddof=0)
48: (12)                        r1 = np.ma.corrcoef(x, y, ddof=1)
49: (12)                        assert_allclose(r0.data, r1.data)
50: (4)                def test_mask_not_backmangled(self):
51: (8)                    a = np.ma.MaskedArray([1., 2.], mask=[False, False])
52: (8)                    assert_(a.mask.shape == (2,))
53: (8)                    b = np.tile(a, (2, 1))
54: (8)                    assert_(a.mask.shape == (2,))
55: (8)                    assert_(b.shape == (2, 2))
56: (8)                    assert_(b.mask.shape == (2, 2))
57: (4)                def test_empty_list_on_structured(self):
58: (8)                    ma = np.ma.MaskedArray([(1, 1.), (2, 2.), (3, 3.)], dtype='i4,f4')

```

```

59: (8)             assert_array_equal(ma[[]], ma[:0])
60: (4)         def test_masked_array_tobytes_fortran(self):
61: (8)             ma = np.ma.arange(4).reshape((2,2))
62: (8)             assert_array_equal(ma.tobytes(order='F'), ma.T.tobytes())
63: (4)         def test_structured_array(self):
64: (8)             np.ma.array((1, (b'', b'')), 
65: (20)                 dtype=[("x", np.int_), 
66: (26)                     ("y", [(("i", np void), ("j", np void))])])
-----
```

File 274 - \_\_init\_\_.py:

```
1: (0)
```

File 275 - test\_subclassing.py:

```

1: (0)         """Tests suite for MaskedArray & subclassing.
2: (0)         :author: Pierre Gerard-Marchant
3: (0)         :contact: pierregm_at_uga_dot_edu
4: (0)         :version: $Id: test_subclassing.py 3473 2007-10-29 15:18:13Z jarrod.millman $
5: (0)
6: (0)         import numpy as np
7: (0)         from numpy.lib.mixins import NDArrayOperatorsMixin
8: (0)         from numpy.testing import assert_, assert_raises
9: (0)         from numpy.ma.testutils import assert_equal
10: (0)        from numpy.ma.core import (
11: (4)            array, arange, masked, MaskedArray, masked_array, log, add, hypot,
12: (4)            divide, asarray, asanyarray, nomask
13: (4)        )
14: (0)        def assert_startswith(a, b):
15: (4)            assert_equal(a[:len(b)], b)
16: (0)        class SubArray(np.ndarray):
17: (4)            def __new__(cls, arr, info={}):
18: (8)                x = np.asanyarray(arr).view(cls)
19: (8)                x.info = info.copy()
20: (8)                return x
21: (4)            def __array_finalize__(self, obj):
22: (8)                super().__array_finalize__(obj)
23: (8)                self.info = getattr(obj, 'info', {}).copy()
24: (8)                return
25: (4)            def __add__(self, other):
26: (8)                result = super().__add__(other)
27: (8)                result.info['added'] = result.info.get('added', 0) + 1
28: (8)                return result
29: (4)            def __iadd__(self, other):
30: (8)                result = super().__iadd__(other)
31: (8)                result.info['iadded'] = result.info.get('iadded', 0) + 1
32: (8)                return result
33: (0)        subarray = SubArray
34: (0)        class SubMaskedArray(MaskedArray):
35: (4)            """Pure subclass of MaskedArray, keeping some info on subclass."""
36: (4)            def __new__(cls, info=None, **kwargs):
37: (8)                obj = super().__new__(cls, **kwargs)
38: (8)                obj._optinfo['info'] = info
39: (8)                return obj
40: (0)        class MSubArray(SubArray, MaskedArray):
41: (4)            def __new__(cls, data, info={}, mask=nomask):
42: (8)                subarr = SubArray(data, info)
43: (8)                _data = MaskedArray.__new__(cls, data=subarr, mask=mask)
44: (8)                _data.info = subarr.info
45: (8)                return _data
46: (4)            @property
47: (4)            def _series(self):
48: (8)                _view = self.view(MaskedArray)
49: (8)                _view._sharedmask = False
50: (8)                return _view
-----
```

```

51: (0)
52: (0)
53: (4)
54: (4)
55: (4)
56: (4)
57: (4)
58: (4)
59: (4)
60: (8)
61: (8)
62: (4)
63: (8)
64: (4)
65: (8)
66: (8)
67: (12)
68: (8)
69: (8)
70: (4)
71: (8)
72: (4)
73: (8)
74: (0)
75: (4)
76: (8)
77: (4)
78: (8)
79: (4)
80: (8)
81: (12)
82: (8)
83: (4)
84: (8)
85: (4)
86: (8)
87: (8)
88: (12)
89: (8)
90: (4)
91: (4)
92: (8)
93: (4)
94: (4)
95: (8)
96: (8)
97: (4)
98: (8)
99: (8)
100: (12)
101: (8)
102: (0)
103: (4)
104: (4)
105: (4)
106: (4)
107: (4)
108: (4)
109: (4)
110: (4)
111: (8)
112: (8)
113: (4)
114: (8)
115: (4)
116: (8)
117: (4)
118: (8)
119: (12)

msubarray = MSubArray
class CSAIterator:
    """
        Flat iterator object that uses its own setter/getter
        (works around ndarray.flat not propagating subclass setters/getters
        see https://github.com/numpy/numpy/issues/4564)
        roughly following MaskedIterator
    """
    def __init__(self, a):
        self._original = a
        self._dataiter = a.view(np.ndarray).flat
    def __iter__(self):
        return self
    def __getitem__(self, indx):
        out = self._dataiter.__getitem__(indx)
        if not isinstance(out, np.ndarray):
            out = out.__array__()
        out = out.view(type(self._original))
        return out
    def __setitem__(self, index, value):
        self._dataiter[index] = self._original._validate_input(value)
    def __next__(self):
        return next(self._dataiter).__array__().view(type(self._original))

class ComplicatedSubArray(SubArray):
    def __str__(self):
        return f'myprefix {self.view(SubArray)} mypostfix'
    def __repr__(self):
        return f'{self.__class__.__name__} {self}'
    def _validate_input(self, value):
        if not isinstance(value, ComplicatedSubArray):
            raise ValueError("Can only set to MySubArray values")
        return value
    def __setitem__(self, item, value):
        super().__setitem__(item, self._validate_input(value))
    def __getitem__(self, item):
        value = super().__getitem__(item)
        if not isinstance(value, np.ndarray): # scalar
            value = value.__array__().view(ComplicatedSubArray)
        return value
    @property
    def flat(self):
        return CSAIterator(self)
    @flat.setter
    def flat(self, value):
        y = self.ravel()
        y[:] = value
    def __array_wrap__(self, obj, context=None):
        obj = super().__array_wrap__(obj, context)
        if context is not None and context[0] is np.multiply:
            obj.info['multiplied'] = obj.info.get('multiplied', 0) + 1
        return obj

class WrappedArray(NDArrayOperatorsMixin):
    """
        Wrapping a MaskedArray rather than subclassing to test that
        ufunc deferrals are commutative.
        See: https://github.com/numpy/numpy/issues/15200
    """
    __slots__ = ('_array', 'attrs')
    __array_priority__ = 20
    def __init__(self, array, **attrs):
        self._array = array
        self.attrs = attrs
    def __repr__(self):
        return f'{self.__class__.__name__}(\n{self._array}\n{self.attrs}\n)'
    def __array__(self):
        return np.asarray(self._array)
    def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
        if method == '__call__':
            inputs = [arg._array if isinstance(arg, self.__class__) else arg

```

```

120: (22)                         for arg in inputs]
121: (12)                         return self.__class__(ufunc(*inputs, **kwargs), **self.attrs)
122: (8)                          else:
123: (12)                          return NotImplemented
124: (0)                           class TestSubclassing:
125: (4)                           def setup_method(self):
126: (8)                             x = np.arange(5, dtype='float')
127: (8)                             mx = msubarray(x, mask=[0, 1, 0, 0, 0])
128: (8)                             self.data = (x, mx)
129: (4)                           def test_data_subclassing(self):
130: (8)                             x = np.arange(5)
131: (8)                             m = [0, 0, 1, 0, 0]
132: (8)                             xsub = SubArray(x)
133: (8)                             xmsub = masked_array(xsub, mask=m)
134: (8)                             assert_(isinstance(xmsub, MaskedArray))
135: (8)                             assert_equal(xmsub._data, xsub)
136: (8)                             assert_(isinstance(xmsub._data, SubArray))
137: (4)                           def test_maskedarray_subclassing(self):
138: (8)                             (x, mx) = self.data
139: (8)                             assert_(isinstance(mx._data, subarray))
140: (4)                           def test_masked Unary_operations(self):
141: (8)                             (x, mx) = self.data
142: (8)                             with np.errstate(divide='ignore'):
143: (12)                               assert_(isinstance(log(mx), msubarray))
144: (12)                               assert_equal(log(x), np.log(x))
145: (4)                           def test_masked_binary_operations(self):
146: (8)                             (x, mx) = self.data
147: (8)                             assert_(isinstance(add(mx, mx), msubarray))
148: (8)                             assert_(isinstance(add(mx, x), msubarray))
149: (8)                             assert_equal(add(mx, x), mx+x)
150: (8)                             assert_(isinstance(add(mx, mx)._data, subarray))
151: (8)                             assert_(isinstance(add.outer(mx, mx), msubarray))
152: (8)                             assert_(isinstance(hypot(mx, mx), msubarray))
153: (8)                             assert_(isinstance(hypot(mx, x), msubarray))
154: (4)                           def test_masked_binary_operations2(self):
155: (8)                             (x, mx) = self.data
156: (8)                             xmx = masked_array(mx.data.__array__(), mask=mx.mask)
157: (8)                             assert_(isinstance(divide(mx, mx), msubarray))
158: (8)                             assert_(isinstance(divide(mx, x), msubarray))
159: (8)                             assert_equal(divide(mx, mx), divide(xmx, xmx))
160: (4)                           def test_attributepropagation(self):
161: (8)                             x = array(arange(5), mask=[0]+[1]*4)
162: (8)                             my = masked_array(subarray(x))
163: (8)                             ym = msubarray(x)
164: (8)                             z = (my+1)
165: (8)                             assert_(isinstance(z, MaskedArray))
166: (8)                             assert_(not isinstance(z, MSubArray))
167: (8)                             assert_(isinstance(z._data, SubArray))
168: (8)                             assert_equal(z._data.info, {})
169: (8)                             z = (ym+1)
170: (8)                             assert_(isinstance(z, MaskedArray))
171: (8)                             assert_(isinstance(z, MSubArray))
172: (8)                             assert_(isinstance(z._data, SubArray))
173: (8)                             assert_(z._data.info['added'] > 0)
174: (8)                             ym += 1
175: (8)                             assert_(isinstance(ym, MaskedArray))
176: (8)                             assert_(isinstance(ym, MSubArray))
177: (8)                             assert_(isinstance(ym._data, SubArray))
178: (8)                             assert_(ym._data.info['iadded'] > 0)
179: (8)                             ym._set_mask([1, 0, 0, 0, 1])
180: (8)                             assert_equal(ym._mask, [1, 0, 0, 0, 1])
181: (8)                             ym._series._set_mask([0, 0, 0, 0, 1])
182: (8)                             assert_equal(ym._mask, [0, 0, 0, 0, 1])
183: (8)                             xsub = subarray(x, info={'name': 'x'})
184: (8)                             mxsub = masked_array(xsub)
185: (8)                             assert_(hasattr(mxsub, 'info'))
186: (8)                             assert_equal(mxsub.info, xsub.info)
187: (4)                           def test_subclasspreservation(self):
188: (8)                             x = np.arange(5)

```

```

189: (8)             m = [0, 0, 1, 0, 0]
190: (8)             xinfo = [(i, j) for (i, j) in zip(x, m)]
191: (8)             xsub = MSubArray(x, mask=m, info={'xsub':xinfo})
192: (8)             mxsub = masked_array(xsub, subok=False)
193: (8)             assert_(not isinstance(mxsub, MSubArray))
194: (8)             assert_(isinstance(mxsub, MaskedArray))
195: (8)             assert_equal(mxsub._mask, m)
196: (8)             mxsub = asarray(xsub)
197: (8)             assert_(not isinstance(mxsub, MSubArray))
198: (8)             assert_(isinstance(mxsub, MaskedArray))
199: (8)             assert_equal(mxsub._mask, m)
200: (8)             mxsub = masked_array(xsub, subok=True)
201: (8)             assert_(isinstance(mxsub, MSubArray))
202: (8)             assert_equal(mxsub.info, xsub.info)
203: (8)             assert_equal(mxsub._mask, xsub._mask)
204: (8)             mxsub = asanyarray(xsub)
205: (8)             assert_(isinstance(mxsub, MSubArray))
206: (8)             assert_equal(mxsub.info, xsub.info)
207: (8)             assert_equal(mxsub._mask, m)
208: (4)             def test_subclass_items(self):
209: (8)                 """test that getter and setter go via baseclass"""
210: (8)                 x = np.arange(5)
211: (8)                 xcsub = ComplicatedSubArray(x)
212: (8)                 mxcsub = masked_array(xcsub, mask=[True, False, True, False, False])
213: (8)                 assert_(isinstance(xcsub[1], ComplicatedSubArray))
214: (8)                 assert_(isinstance(xcsub[1,...], ComplicatedSubArray))
215: (8)                 assert_(isinstance(xcsub[1:4], ComplicatedSubArray))
216: (8)                 assert_(isinstance(mxcsub[1], ComplicatedSubArray))
217: (8)                 assert_(isinstance(mxcsub[1,...].data, ComplicatedSubArray))
218: (8)                 assert_(mxcsub[0] is masked)
219: (8)                 assert_(isinstance(mxcsub[0,...].data, ComplicatedSubArray))
220: (8)                 assert_(isinstance(mxcsub[1:4].data, ComplicatedSubArray))
221: (8)                 assert_(isinstance(mxcsub.flat[1].data, ComplicatedSubArray))
222: (8)                 assert_(mxcsub.flat[0] is masked)
223: (8)                 assert_(isinstance(mxcsub.flat[1:4].base, ComplicatedSubArray))
224: (8)                 assert_raises(ValueError, xcsub.__setitem__, 1, x[4])
225: (8)                 assert_raises(ValueError, mxcsub.__setitem__, 1, x[4])
226: (8)                 assert_raises(ValueError, mxcsub.__setitem__, slice(1, 4), x[1:4])
227: (8)                 mxcsub[1] = xcsub[4]
228: (8)                 mxcsub[1:4] = xcsub[1:4]
229: (8)                 assert_raises(ValueError, mxcsub.flat.__setitem__, 1, x[4])
230: (8)                 assert_raises(ValueError, mxcsub.flat.__setitem__, slice(1, 4),
231: (8)                               x[1:4])
232: (8)                 mxcsub.flat[1] = xcsub[4]
233: (4)                 mxcsub.flat[1:4] = xcsub[1:4]
234: (4)             def test_subclass_nomask_items(self):
235: (8)                 x = np.arange(5)
236: (8)                 xcsub = ComplicatedSubArray(x)
237: (8)                 mxcsub_nomask = masked_array(xcsub)
238: (8)                 assert_(isinstance(mxcsub_nomask[1,...].data, ComplicatedSubArray))
239: (8)                 assert_(isinstance(mxcsub_nomask[0,...].data, ComplicatedSubArray))
240: (8)                 assert_(isinstance(mxcsub_nomask[1], ComplicatedSubArray))
241: (8)                 assert_(isinstance(mxcsub_nomask[0], ComplicatedSubArray))
242: (4)             def test_subclass_repr(self):
243: (8)                 """test that repr uses the name of the subclass
244: (8)                 and 'array' for np.ndarray"""
245: (8)                 x = np.arange(5)
246: (8)                 mx = masked_array(x, mask=[True, False, True, False, False])
247: (8)                 assert_startswith(repr(mx), 'masked_array')
248: (8)                 xsub = SubArray(x)
249: (8)                 mxsub = masked_array(xsub, mask=[True, False, True, False, False])
250: (12)                 assert_startswith(repr(mxsub),
251: (4)                               f'masked_{SubArray.__name__}(data=[--, 1, --, 3, 4])')
252: (4)             def test_subclass_str(self):
253: (8)                 """test str with subclass that has overridden str, setitem"""
254: (8)                 x = np.arange(5)
255: (8)                 xsub = SubArray(x)
256: (8)                 mxsub = masked_array(xsub, mask=[True, False, True, False, False])
257: (8)                 assert_equal(str(mxsub), '[-- 1 -- 3 4]')

```

```

257: (8)           xcsub = ComplicatedSubArray(x)
258: (8)           assert_raises(ValueError, xcsub.__setitem__, 0,
259: (22)                     np.ma.core.masked_print_option)
260: (8)           mxcsub = masked_array(xcsub, mask=[True, False, True, False, False])
261: (8)           assert_equal(str(mxcsub), 'myprefix [-- 1 -- 3 4] mypostfix')
262: (4)           def test_pure_subclass_info_preservation(self):
263: (8)             arr1 = SubMaskedArray('test', data=[1,2,3,4,5,6])
264: (8)             arr2 = SubMaskedArray(data=[0,1,2,3,4,5])
265: (8)             diff1 = np.subtract(arr1, arr2)
266: (8)             assert_('info' in diff1._optinfo)
267: (8)             assert_(diff1._optinfo['info'] == 'test')
268: (8)             diff2 = arr1 - arr2
269: (8)             assert_('info' in diff2._optinfo)
270: (8)             assert_(diff2._optinfo['info'] == 'test')
271: (0)           class ArrayNoInheritance:
272: (4)             """Quantity-like class that does not inherit from ndarray"""
273: (4)             def __init__(self, data, units):
274: (8)               self.magnitude = data
275: (8)               self.units = units
276: (4)             def __getattr__(self, attr):
277: (8)               return getattr(self.magnitude, attr)
278: (0)           def test_array_no_inheritance():
279: (4)             data_masked = np.ma.array([1, 2, 3], mask=[True, False, True])
280: (4)             data_masked_units = ArrayNoInheritance(data_masked, 'meters')
281: (4)             new_array = np.ma.array(data_masked_units)
282: (4)             assert_equal(data_masked.data, new_array.data)
283: (4)             assert_equal(data_masked.mask, new_array.mask)
284: (4)             data_masked.mask = [True, False, False]
285: (4)             assert_equal(data_masked.mask, new_array.mask)
286: (4)             assert_(new_array.sharedmask)
287: (4)             new_array = np.ma.array(data_masked_units, copy=True)
288: (4)             assert_equal(data_masked.data, new_array.data)
289: (4)             assert_equal(data_masked.mask, new_array.mask)
290: (4)             data_masked.mask = [True, False, True]
291: (4)             assert_equal([True, False, False], new_array.mask)
292: (4)             assert_(not new_array.sharedmask)
293: (4)             new_array = np.ma.array(data_masked_units, keep_mask=False)
294: (4)             assert_equal(data_masked.data, new_array.data)
295: (4)             assert_equal(data_masked.mask, [True, False, True])
296: (4)             assert_(not new_array.mask)
297: (4)             assert_(not new_array.sharedmask)
298: (0)           class TestClassWrapping:
299: (4)             def setup_method(self):
300: (8)               m = np.ma.masked_array([1, 3, 5], mask=[False, True, False])
301: (8)               wm = WrappedArray(m)
302: (8)               self.data = (m, wm)
303: (4)             def test_masked Unary_operations(self):
304: (8)               (m, wm) = self.data
305: (8)               with np.errstate(divide='ignore'):
306: (12)                 assert_(isinstance(np.log(wm), WrappedArray))
307: (4)             def test_masked_binary_operations(self):
308: (8)               (m, wm) = self.data
309: (8)               assert_(isinstance(np.add(wm, wm), WrappedArray))
310: (8)               assert_(isinstance(np.add(m, wm), WrappedArray))
311: (8)               assert_(isinstance(np.add(wm, m), WrappedArray))
312: (8)               assert_equal(np.add(m, wm), m + wm)
313: (8)               assert_(isinstance(np.hypot(m, wm), WrappedArray))
314: (8)               assert_(isinstance(np.hypot(wm, m), WrappedArray))
315: (8)               assert_(isinstance(np.divide(wm, m), WrappedArray))
316: (8)               assert_(isinstance(np.divide(m, wm), WrappedArray))
317: (8)               assert_equal(np.divide(wm, m) * m, np.divide(m, m) * wm)
318: (8)               m2 = np.stack([m, m])
319: (8)               assert_(isinstance(np.divide(wm, m2), WrappedArray))
320: (8)               assert_(isinstance(np.divide(m2, wm), WrappedArray))
321: (8)               assert_equal(np.divide(m2, wm), np.divide(wm, m2))
322: (4)             def test_mixins_have_slots(self):
323: (8)               mixin = NDArryOperatorsMixin()
324: (8)               assert_raises(AttributeError, mixin.__setattr__, "not_a_real_attr", 1)
325: (8)               m = np.ma.masked_array([1, 3, 5], mask=[False, True, False])

```

```
326: (8)             wm = WrappedArray(m)
327: (8)             assert_raises(AttributeError, wm.__setattr__, "not_an_attr", 2)
```

---

## File 276 - defmatrix.py:

```
1: (0)             __all__ = ['matrix', 'bmat', 'mat', 'asmatrix']
2: (0)             import sys
3: (0)             import warnings
4: (0)             import ast
5: (0)             from .._utils import set_module
6: (0)             import numpy.core.numeric as N
7: (0)             from numpy.core.numeric import concatenate, isscalar
8: (0)             from numpy.linalg import matrix_power
9: (0)             def _convert_from_string(data):
10: (4)             for char in []:
11: (8)                 data = data.replace(char, '')
12: (4)             rows = data.split(';')
13: (4)             newdata = []
14: (4)             count = 0
15: (4)             for row in rows:
16: (8)                 trow = row.split(',')
17: (8)                 newrow = []
18: (8)                 for col in trow:
19: (12)                     temp = col.split()
20: (12)                     newrow.extend(map(ast.literal_eval, temp))
21: (8)                     if count == 0:
22: (12)                         Ncols = len(newrow)
23: (8)                     elif len(newrow) != Ncols:
24: (12)                         raise ValueError("Rows not the same size.")
25: (8)                     count += 1
26: (8)                     newdata.append(newrow)
27: (4)             return newdata
28: (0)             @set_module('numpy')
29: (0)             def asmatrix(data, dtype=None):
30: (4)                 """
31: (4)                 Interpret the input as a matrix.
32: (4)                 Unlike `matrix`, `asmatrix` does not make a copy if the input is already
33: (4)                 a matrix or an ndarray. Equivalent to ``matrix(data, copy=False)``.
34: (4)                 Parameters
35: (4)                 -----
36: (4)                 data : array_like
37: (8)                     Input data.
38: (4)                 dtype : data-type
39: (7)                     Data-type of the output matrix.
40: (4)                 Returns
41: (4)                 -----
42: (4)                 mat : matrix
43: (8)                     `data` interpreted as a matrix.
44: (4)                 Examples
45: (4)                 -----
46: (4)                 >>> x = np.array([[1, 2], [3, 4]])
47: (4)                 >>> m = np.asmatrix(x)
48: (4)                 >>> x[0,0] = 5
49: (4)                 >>> m
50: (4)                 matrix([[5, 2],
51: (12)                     [3, 4]])
52: (4)                 """
53: (4)                 return matrix(data, dtype=dtype, copy=False)
54: (0)             @set_module('numpy')
55: (0)             class matrix(N.ndarray):
56: (4)                 """
57: (4)                 matrix(data, dtype=None, copy=True)
58: (4)                 .. note:: It is no longer recommended to use this class, even for linear
59: (14)                     algebra. Instead use regular arrays. The class may be removed
60: (14)                     in the future.
61: (4)                 Returns a matrix from an array-like object, or from a string of data.
62: (4)                 A matrix is a specialized 2-D array that retains its 2-D nature
```

```

63: (4)           through operations. It has certain special operators, such as ``*```
64: (4)           (matrix multiplication) and ``**`` (matrix power).
65: (4)           Parameters
66: (4)           -----
67: (4)           data : array_like or string
68: (7)             If `data` is a string, it is interpreted as a matrix with commas
69: (7)             or spaces separating columns, and semicolons separating rows.
70: (4)           dtype : data-type
71: (7)             Data-type of the output matrix.
72: (4)           copy : bool
73: (7)             If `data` is already an `ndarray`, then this flag determines
74: (7)             whether the data is copied (the default), or whether a view is
75: (7)             constructed.
76: (4)           See Also
77: (4)           -----
78: (4)           array
79: (4)           Examples
80: (4)           -----
81: (4)           >>> a = np.matrix('1 2; 3 4')
82: (4)
83: (4)           matrix([[1, 2],
84: (12)             [3, 4]])
85: (4)
86: (4)           >>> np.matrix([[1, 2], [3, 4]])
87: (12)             matrix([[1, 2],
88: (4)               [3, 4]])
89: (4)             """
90: (4)             __array_priority__ = 10.0
91: (8)             def __new__(subtype, data, dtype=None, copy=True):
92: (22)               warnings.warn('the matrix subclass is not the recommended way to '
93: (22)                 'represent matrices or deal with linear algebra (see '
94: (22)                   'https://docs.scipy.org/doc/numpy/user/'
95: (22)                     'numpy-for-matlab-users.html). '
96: (22)                     'Please adjust your code to use regular ndarray.',
97: (8)                     PendingDeprecationWarning, stacklevel=2)
98: (12)
99: (12)
100: (16)
101: (12)
102: (16)
103: (12)
104: (8)
105: (12)
106: (16)
107: (12)
108: (16)
109: (12)
110: (12)
111: (16)
112: (12)
113: (12)
114: (8)
115: (12)
116: (8)
117: (8)
118: (8)
119: (8)
120: (12)
121: (8)
122: (12)
123: (8)
124: (12)
125: (8)
126: (8)
127: (12)
128: (8)
129: (12)
130: (8)
131: (32)

```

through operations. It has certain special operators, such as ``\*`` (matrix multiplication) and ``\*\*`` (matrix power).

Parameters

-----

data : array\_like or string

If `data` is a string, it is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows.

dtype : data-type

Data-type of the output matrix.

copy : bool

If `data` is already an `ndarray`, then this flag determines whether the data is copied (the default), or whether a view is constructed.

See Also

-----

array

Examples

-----

```
>>> a = np.matrix('1 2; 3 4')
>>> a
matrix([[1, 2],
       [3, 4]])
>>> np.matrix([[1, 2], [3, 4]])
matrix([[1, 2],
       [3, 4]])
"""
__array_priority__ = 10.0
def __new__(subtype, data, dtype=None, copy=True):
    warnings.warn('the matrix subclass is not the recommended way to '
                  'represent matrices or deal with linear algebra (see '
                  'https://docs.scipy.org/doc/numpy/user/'
                  'numpy-for-matlab-users.html). '
                  'Please adjust your code to use regular ndarray.',
                  PendingDeprecationWarning, stacklevel=2)
    if isinstance(data, matrix):
        dtype2 = data.dtype
        if (dtype is None):
            dtype = dtype2
        if (dtype2 == dtype) and (not copy):
            return data
        return data.astype(dtype)
    if isinstance(data, N.ndarray):
        if dtype is None:
            intype = data.dtype
        else:
            intype = N.dtype(dtype)
        new = data.view(subtype)
        if intype != data.dtype:
            return new.astype(intype)
        if copy: return new.copy()
        else: return new
    if isinstance(data, str):
        data = _convert_from_string(data)
    arr = N.array(data, dtype=dtype, copy=copy)
    ndim = arr.ndim
    shape = arr.shape
    if (ndim > 2):
        raise ValueError("matrix must be 2-dimensional")
    elif ndim == 0:
        shape = (1, 1)
    elif ndim == 1:
        shape = (1, shape[0])
    order = 'C'
    if (ndim == 2) and arr.flags.fortran:
        order = 'F'
    if not (order or arr.flags.contiguous):
        arr = arr.copy()
    ret = N.ndarray.__new__(subtype, shape, arr.dtype,
                           buffer=arr,
```

```

132: (32)                                order=order)
133: (8)        return ret
134: (4)        def __array_finalize__(self, obj):
135: (8)            self._getitem = False
136: (8)            if (isinstance(obj, matrix) and obj._getitem): return
137: (8)            ndim = self.ndim
138: (8)            if (ndim == 2):
139: (12)                return
140: (8)            if (ndim > 2):
141: (12)                newshape = tuple([x for x in self.shape if x > 1])
142: (12)                ndim = len(newshape)
143: (12)                if ndim == 2:
144: (16)                    self.shape = newshape
145: (16)                    return
146: (12)                elif (ndim > 2):
147: (16)                    raise ValueError("shape too large to be a matrix.")
148: (8)            else:
149: (12)                newshape = self.shape
150: (8)            if ndim == 0:
151: (12)                self.shape = (1, 1)
152: (8)            elif ndim == 1:
153: (12)                self.shape = (1, newshape[0])
154: (8)            return
155: (4)        def __getitem__(self, index):
156: (8)            self._getitem = True
157: (8)            try:
158: (12)                out = N.ndarray.__getitem__(self, index)
159: (8)            finally:
160: (12)                self._getitem = False
161: (8)            if not isinstance(out, N.ndarray):
162: (12)                return out
163: (8)            if out.ndim == 0:
164: (12)                return out[()]
165: (8)            if out.ndim == 1:
166: (12)                sh = out.shape[0]
167: (12)                try:
168: (16)                    n = len(index)
169: (12)                except Exception:
170: (16)                    n = 0
171: (12)                if n > 1 and isscalar(index[1]):
172: (16)                    out.shape = (sh, 1)
173: (12)                else:
174: (16)                    out.shape = (1, sh)
175: (8)            return out
176: (4)        def __mul__(self, other):
177: (8)            if isinstance(other, (N.ndarray, list, tuple)) :
178: (12)                return N.dot(self, asmatrix(other))
179: (8)            if isscalar(other) or not hasattr(other, '__rmul__') :
180: (12)                return N.dot(self, other)
181: (8)            return NotImplemented
182: (4)        def __rmul__(self, other):
183: (8)            return N.dot(other, self)
184: (4)        def __imul__(self, other):
185: (8)            self[:] = self * other
186: (8)            return self
187: (4)        def __pow__(self, other):
188: (8)            return matrix_power(self, other)
189: (4)        def __ipow__(self, other):
190: (8)            self[:] = self ** other
191: (8)            return self
192: (4)        def __rpow__(self, other):
193: (8)            return NotImplemented
194: (4)        def __align__(self, axis):
195: (8)            """A convenience function for operations that need to preserve axis
196: (8)            orientation.
197: (8)            """
198: (8)            if axis is None:
199: (12)                return self[0, 0]
200: (8)            elif axis==0:

```

```

201: (12)                      return self
202: (8)                       elif axis==1:
203: (12)                         return self.transpose()
204: (8)                       else:
205: (12)                         raise ValueError("unsupported axis")
206: (4) def __collapse(self, axis):
207: (8)   """A convenience function for operations that want to collapse
208: (8)   to a scalar like _align, but are using keepdims=True
209: (8)   """
210: (8)   if axis is None:
211: (12)     return self[0, 0]
212: (8)   else:
213: (12)     return self
214: (4) def tolist(self):
215: (8)   """
216: (8)   Return the matrix as a (possibly nested) list.
217: (8)   See `ndarray.tolist` for full documentation.
218: (8)   See Also
219: (8)   -----
220: (8)   ndarray.tolist
221: (8)   Examples
222: (8)   -----
223: (8)   >>> x = np.matrix(np.arange(12).reshape((3,4))); x
224: (8)   matrix([[ 0,  1,  2,  3],
225: (16)     [ 4,  5,  6,  7],
226: (16)     [ 8,  9, 10, 11]])
227: (8)   >>> x.tolist()
228: (8)   [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
229: (8)   """
230: (8)   return self.__array__().tolist()
231: (4) def sum(self, axis=None, dtype=None, out=None):
232: (8)   """
233: (8)   Returns the sum of the matrix elements, along the given axis.
234: (8)   Refer to `numpy.sum` for full documentation.
235: (8)   See Also
236: (8)   -----
237: (8)   numpy.sum
238: (8)   Notes
239: (8)   -----
240: (8)   This is the same as `ndarray.sum`, except that where an `ndarray`
would
241: (8)   be returned, a `matrix` object is returned instead.
242: (8)   Examples
243: (8)   -----
244: (8)   >>> x = np.matrix([[1, 2], [4, 3]])
245: (8)   >>> x.sum()
246: (8)   10
247: (8)   >>> x.sum(axis=1)
248: (8)   matrix([[3,
249: (16)     [7]])
250: (8)   >>> x.sum(axis=1, dtype='float')
251: (8)   matrix([[3.],
252: (16)     [7.]])
253: (8)   >>> out = np.zeros((2, 1), dtype='float')
254: (8)   >>> x.sum(axis=1, dtype='float', out=np.asmatrix(out))
255: (8)   matrix([[3.],
256: (16)     [7.]])
257: (8)   """
258: (8)   return N.ndarray.sum(self, axis, dtype, out,
keepdims=True).__collapse__(axis)
259: (4) def squeeze(self, axis=None):
260: (8)   """
261: (8)   Return a possibly reshaped matrix.
262: (8)   Refer to `numpy.squeeze` for more documentation.
263: (8)   Parameters
264: (8)   -----
265: (8)   axis : None or int or tuple of ints, optional
266: (12)     Selects a subset of the axes of length one in the shape.
267: (12)     If an axis is selected with shape entry greater than one,
```

```

268: (12)                                an error is raised.
269: (8)                                 Returns
270: (8)                                 -----
271: (8)                                 squeezed : matrix
272: (12)                                The matrix, but as a (1, N) matrix if it had shape (N, 1).
273: (8)                                 See Also
274: (8)                                 -----
275: (8)                                 numpy.squeeze : related function
276: (8)                                 Notes
277: (8)                                 -----
278: (8)                                If `m` has a single column then that column is returned
279: (8)                                as the single row of a matrix. Otherwise `m` is returned.
280: (8)                                The returned matrix is always either `m` itself or a view into `m`.
281: (8)                                Supplying an axis keyword argument will not affect the returned matrix
282: (8)                                but it may cause an error to be raised.
283: (8)                                 Examples
284: (8)                                 -----
285: (8)                                >>> c = np.matrix([[1], [2]])
286: (8)                                >>> c
287: (8)                                matrix([[1],
288: (16)                                  [2]])
289: (8)                                >>> c.squeeze()
290: (8)                                matrix([[1, 2]])
291: (8)                                >>> r = c.T
292: (8)                                >>> r
293: (8)                                matrix([[1, 2]])
294: (8)                                >>> r.squeeze()
295: (8)                                matrix([[1, 2]])
296: (8)                                >>> m = np.matrix([[1, 2], [3, 4]])
297: (8)                                >>> m.squeeze()
298: (8)                                matrix([[1, 2],
299: (16)                                  [3, 4]])
300: (8)                                """
301: (8)                                return N.ndarray.squeeze(self, axis=axis)
302: (4)                                 def flatten(self, order='C'):
303: (8)                                """
304: (8)                                Return a flattened copy of the matrix.
305: (8)                                All `N` elements of the matrix are placed into a single row.
306: (8)                                 Parameters
307: (8)                                 -----
308: (8)                                 order : {'C', 'F', 'A', 'K'}, optional
309: (12)                                'C' means to flatten in row-major (C-style) order. 'F' means to
310: (12)                                flatten in column-major (Fortran-style) order. 'A' means to
311: (12)                                flatten in column-major order if `m` is Fortran *contiguous* in
312: (12)                                memory, row-major order otherwise. 'K' means to flatten `m` in
313: (12)                                the order the elements occur in memory. The default is 'C'.
314: (8)                                 Returns
315: (8)                                 -----
316: (8)                                 y : matrix
317: (12)                                A copy of the matrix, flattened to a `(1, N)` matrix where `N`
318: (12)                                is the number of elements in the original matrix.
319: (8)                                 See Also
320: (8)                                 -----
321: (8)                                 ravel : Return a flattened array.
322: (8)                                 flat : A 1-D flat iterator over the matrix.
323: (8)                                 Examples
324: (8)                                 -----
325: (8)                                >>> m = np.matrix([[1,2], [3,4]])
326: (8)                                >>> m.flatten()
327: (8)                                matrix([[1, 2, 3, 4]])
328: (8)                                >>> m.flatten('F')
329: (8)                                matrix([[1, 3, 2, 4]])
330: (8)                                """
331: (8)                                return N.ndarray.flatten(self, order=order)
332: (4)                                 def mean(self, axis=None, dtype=None, out=None):
333: (8)                                """
334: (8)                                Returns the average of the matrix elements along the given axis.
335: (8)                                Refer to `numpy.mean` for full documentation.
336: (8)                                See Also

```

```

337: (8)          -----
338: (8)          numpy.mean
339: (8)          Notes
340: (8)          -----
341: (8)          Same as `ndarray.mean` except that, where that returns an `ndarray`,
342: (8)          this returns a `matrix` object.
343: (8)          Examples
344: (8)          -----
345: (8)          >>> x = np.matrix(np.arange(12).reshape((3, 4)))
346: (8)          >>> x
347: (8)          matrix([[ 0,  1,  2,  3],
348: (16)           [ 4,  5,  6,  7],
349: (16)           [ 8,  9, 10, 11]])
350: (8)          >>> x.mean()
351: (8)          5.5
352: (8)          >>> x.mean(0)
353: (8)          matrix([[4., 5., 6., 7.]])
354: (8)          >>> x.mean(1)
355: (8)          matrix([[ 1.5],
356: (16)           [ 5.5],
357: (16)           [ 9.5]])
358: (8)          """
359: (8)          return N.ndarray.mean(self, axis, dtype, out,
keepdims=True)._collapse(axis)
360: (4)          def std(self, axis=None, dtype=None, out=None, ddof=0):
361: (8)          """
362: (8)          Return the standard deviation of the array elements along the given
axis.
363: (8)          Refer to `numpy.std` for full documentation.
364: (8)          See Also
365: (8)          -----
366: (8)          numpy.std
367: (8)          Notes
368: (8)          -----
369: (8)          This is the same as `ndarray.std`, except that where an `ndarray`
would
370: (8)          be returned, a `matrix` object is returned instead.
371: (8)          Examples
372: (8)          -----
373: (8)          >>> x = np.matrix(np.arange(12).reshape((3, 4)))
374: (8)          >>> x
375: (8)          matrix([[ 0,  1,  2,  3],
376: (16)           [ 4,  5,  6,  7],
377: (16)           [ 8,  9, 10, 11]])
378: (8)          >>> x.std()
379: (8)          3.4520525295346629 # may vary
380: (8)          >>> x.std(0)
381: (8)          matrix([[ 3.26598632,  3.26598632,  3.26598632,  3.26598632]]) # may
vary
382: (8)          >>> x.std(1)
383: (8)          matrix([[ 1.11803399],
384: (16)           [ 1.11803399],
385: (16)           [ 1.11803399]])
386: (8)          """
387: (8)          return N.ndarray.std(self, axis, dtype, out, ddof,
keepdims=True)._collapse(axis)
388: (4)          def var(self, axis=None, dtype=None, out=None, ddof=0):
389: (8)          """
390: (8)          Returns the variance of the matrix elements, along the given axis.
391: (8)          Refer to `numpy.var` for full documentation.
392: (8)          See Also
393: (8)          -----
394: (8)          numpy.var
395: (8)          Notes
396: (8)          -----
397: (8)          This is the same as `ndarray.var`, except that where an `ndarray`
would
398: (8)          be returned, a `matrix` object is returned instead.
399: (8)          Examples

```

```

400: (8)          -----
401: (8)          >>> x = np.matrix(np.arange(12).reshape((3, 4)))
402: (8)          >>> x
403: (8)          matrix([[ 0,  1,  2,  3],
404: (16)           [ 4,  5,  6,  7],
405: (16)           [ 8,  9, 10, 11]])
406: (8)          >>> x.var()
407: (8)          11.916666666666666
408: (8)          >>> x.var(0)
409: (8)          matrix([[ 10.66666667, 10.66666667, 10.66666667, 10.66666667]]) #
may vary
410: (8)          >>> x.var(1)
411: (8)          matrix([[1.25],
412: (16)           [1.25],
413: (16)           [1.25]])
414: (8)          """
415: (8)          return N.ndarray.var(self, axis, dtype, out, ddof,
keepdims=True)._collapse(axis)
416: (4)          def prod(self, axis=None, dtype=None):
417: (8)          """
418: (8)          Return the product of the array elements over the given axis.
419: (8)          Refer to `prod` for full documentation.
420: (8)          See Also
421: (8)          -----
422: (8)          prod, ndarray.prod
423: (8)          Notes
424: (8)          -----
425: (8)          Same as `ndarray.prod`, except, where that returns an `ndarray`, this
426: (8)          returns a `matrix` object instead.
427: (8)          Examples
428: (8)          -----
429: (8)          >>> x = np.matrix(np.arange(12).reshape((3,4))); x
430: (8)          matrix([[ 0,  1,  2,  3],
431: (16)           [ 4,  5,  6,  7],
432: (16)           [ 8,  9, 10, 11]])
433: (8)          >>> x.prod()
434: (8)          0
435: (8)          >>> x.prod(0)
436: (8)          matrix([[ 0, 45, 120, 231]])
437: (8)          >>> x.prod(1)
438: (8)          matrix([[ 0,
439: (16)           [ 840,
440: (16)           [7920]])
441: (8)          """
442: (8)          return N.ndarray.prod(self, axis, dtype, out,
keepdims=True)._collapse(axis)
443: (4)          def any(self, axis=None, out=None):
444: (8)          """
445: (8)          Test whether any array element along a given axis evaluates to True.
446: (8)          Refer to `numpy.any` for full documentation.
447: (8)          Parameters
448: (8)          -----
449: (8)          axis : int, optional
450: (12)           Axis along which logical OR is performed
451: (8)          out : ndarray, optional
452: (12)           Output to existing array instead of creating new one, must have
453: (12)           same shape as expected output
454: (8)          Returns
455: (8)          -----
456: (12)          any : bool, ndarray
457: (16)           Returns a single bool if `axis` is ``None``; otherwise,
458: (16)           returns `ndarray`
459: (8)          """
460: (8)          return N.ndarray.any(self, axis, out, keepdims=True)._collapse(axis)
461: (4)          def all(self, axis=None, out=None):
462: (8)          """
463: (8)          Test whether all matrix elements along a given axis evaluate to True.
464: (8)          Parameters
465: (8)          -----

```

```

466: (8)             See `numpy.all` for complete descriptions
467: (8)             See Also
468: (8)
469: (8)             -----
470: (8)             numpy.all
471: (8)             Notes
472: (8)             -----
473: (8)             This is the same as `ndarray.all`, but it returns a `matrix` object.
474: (8)             Examples
475: (8)             -----
476: (8)             >>> x = np.matrix(np.arange(12).reshape((3,4))); x
477: (16)            matrix([[ 0,  1,  2,  3],
478: (16)                      [ 4,  5,  6,  7],
479: (8)                      [ 8,  9, 10, 11]])
480: (8)             >>> y = x[0]; y
481: (8)             matrix([[0, 1, 2, 3]])
482: (8)             >>> (x == y)
483: (16)            matrix([[ True,  True,  True,  True],
484: (16)                      [False, False, False, False],
485: (8)                      [False, False, False, False]])
486: (8)             >>> (x == y).all()
487: (8)             False
488: (8)             >>> (x == y).all(0)
489: (8)             matrix([[False, False, False, False]])
490: (8)             >>> (x == y).all(1)
491: (16)            matrix([[ True],
492: (16)                      [False],
493: (8)                      [False]])
494: (8)             """
495: (4)             return N.ndarray.all(self, axis, out, keepdims=True)._collapse(axis)
496: (8)             def max(self, axis=None, out=None):
497: (8)                 """
498: (8)                 Return the maximum value along an axis.
499: (8)                 Parameters
500: (8)                 -----
501: (8)                 See `amax` for complete descriptions
502: (8)                 See Also
503: (8)
504: (8)                 amax, ndarray.max
505: (8)
506: (8)                 Notes
507: (8)
508: (8)                 -----
509: (8)                 This is the same as `ndarray.max`, but returns a `matrix` object
510: (8)                 where `ndarray.max` would return an ndarray.
511: (8)                 Examples
512: (8)                 -----
513: (16)                >>> x = np.matrix(np.arange(12).reshape((3,4))); x
514: (16)                matrix([[ 0,  1,  2,  3],
515: (8)                      [ 4,  5,  6,  7],
516: (8)                      [ 8,  9, 10, 11]])
517: (8)                >>> x.max()
518: (8)                11
519: (8)                >>> x.max(0)
520: (16)                matrix([[ 8,  9, 10, 11]])
521: (16)                >>> x.max(1)
522: (8)                matrix([[ 3],
523: (8)                      [ 7],
524: (8)                      [11]])
525: (8)                """
526: (8)                return N.ndarray.max(self, axis, out, keepdims=True)._collapse(axis)
527: (8)                def argmax(self, axis=None, out=None):
528: (8)                    """
529: (8)                    Indexes of the maximum values along an axis.
530: (8)                    Return the indexes of the first occurrences of the maximum values
531: (8)                    along the specified axis. If axis is None, the index is for the
532: (8)                    flattened matrix.
533: (8)                    Parameters
534: (8)                    -----
535: (8)                    See `numpy.argmax` for complete descriptions
536: (8)                    See Also
537: (8)                    -----
```

```
535: (8)             numpy.argmax
536: (8)             Notes
537: (8)
538: (8)             -----
539: (8)             This is the same as `ndarray.argmax` , but returns a `matrix` object
540: (8)             where `ndarray.argmax` would return an `ndarray` .
541: (8)             Examples
542: (8)             -----
543: (8)             >>> x = np.matrix(np.arange(12).reshape((3,4))); x
544: (16)            matrix([[ 0,  1,  2,  3],
545: (16)                      [ 4,  5,  6,  7],
546: (8)                      [ 8,  9, 10, 11]])
547: (8)             >>> x.argmax()
548: (8)             11
549: (8)             >>> x.argmax(0)
550: (8)             matrix([[2, 2, 2, 2]])
551: (8)             >>> x.argmax(1)
552: (16)            matrix([[3],
553: (16)                      [3],
554: (8)                      [3]])
555: (8)             """
556: (4)             return N.ndarray.argmax(self, axis, out)._align(axis)
557: (8)             def min(self, axis=None, out=None):
558: (8)                 """
559: (8)                 Return the minimum value along an axis.
560: (8)                 Parameters
561: (8)                 -----
562: (8)                 See `amin` for complete descriptions.
563: (8)                 See Also
564: (8)                 -----
565: (8)                 amin, ndarray.min
566: (8)                 Notes
567: (8)                 -----
568: (8)                 This is the same as `ndarray.min` , but returns a `matrix` object
569: (8)                 where `ndarray.min` would return an ndarray.
570: (8)                 Examples
571: (8)                 -----
572: (8)                 >>> x = -np.matrix(np.arange(12).reshape((3,4))); x
573: (16)                matrix([[ 0, -1, -2, -3],
574: (16)                      [ -4, -5, -6, -7],
575: (8)                      [ -8, -9, -10, -11]])
576: (8)                 >>> x.min()
577: (8)                 -11
578: (8)                 >>> x.min(0)
579: (8)                 matrix([[ -8, -9, -10, -11]])
580: (8)                 >>> x.min(1)
581: (16)                matrix([[ -3],
582: (16)                      [ -7],
583: (8)                      [-11]])
584: (8)                 """
585: (4)                 return N.ndarray.min(self, axis, out, keepdims=True)._collapse(axis)
586: (8)                 def argmin(self, axis=None, out=None):
587: (8)                     """
588: (8)                     Indexes of the minimum values along an axis.
589: (8)                     Return the indexes of the first occurrences of the minimum values
590: (8)                     along the specified axis. If axis is None, the index is for the
591: (8)                     flattened matrix.
592: (8)                     Parameters
593: (8)                     -----
594: (8)                     See `numpy.argmin` for complete descriptions.
595: (8)                     See Also
596: (8)                     -----
597: (8)                     numpy.argmin
598: (8)                     Notes
599: (8)                     -----
600: (8)                     This is the same as `ndarray.argmin` , but returns a `matrix` object
601: (8)                     where `ndarray.argmin` would return an `ndarray` .
602: (8)                     Examples
603: (8)                     -----
604: (8)                     >>> x = -np.matrix(np.arange(12).reshape((3,4))); x
```

```

604: (8)           matrix([[ 0, -1, -2, -3],
605: (16)          [ -4, -5, -6, -7],
606: (16)          [ -8, -9, -10, -11]])
607: (8)          >>> x.argmin()
608: (8)          11
609: (8)          >>> x.argmin(0)
610: (8)          matrix([[2, 2, 2, 2]])
611: (8)          >>> x.argmin(1)
612: (8)          matrix([[3],
613: (16)          [3],
614: (16)          [3]])
615: (8)          """
616: (8)          return N.ndarray.argmin(self, axis, out)._align(axis)
617: (4)          def ptp(self, axis=None, out=None):
618: (8)          """
619: (8)          Peak-to-peak (maximum - minimum) value along the given axis.
620: (8)          Refer to `numpy.ptp` for full documentation.
621: (8)          See Also
622: (8)          -----
623: (8)          numpy.ptp
624: (8)          Notes
625: (8)          -----
626: (8)          Same as `ndarray.ptp`, except, where that would return an `ndarray`
object,
627: (8)          this returns a `matrix` object.
628: (8)          Examples
629: (8)          -----
630: (8)          >>> x = np.matrix(np.arange(12).reshape((3,4))); x
631: (8)          matrix([[ 0,  1,  2,  3],
632: (16)          [ 4,  5,  6,  7],
633: (16)          [ 8,  9, 10, 11]])
634: (8)          >>> x.ptp()
635: (8)          11
636: (8)          >>> x.ptp(0)
637: (8)          matrix([[8, 8, 8, 8]])
638: (8)          >>> x.ptp(1)
639: (8)          matrix([[3],
640: (16)          [3],
641: (16)          [3]])
642: (8)          """
643: (8)          return N.ndarray.ptp(self, axis, out)._align(axis)
644: (4)          @property
645: (4)          def I(self):
646: (8)          """
647: (8)          Returns the (multiplicative) inverse of invertible `self`.
648: (8)          Parameters
649: (8)          -----
650: (8)          None
651: (8)          Returns
652: (8)          -----
653: (8)          ret : matrix object
654: (12)          If `self` is non-singular, `ret` is such that ``ret * self`` ==
655: (12)          ``self * ret`` == ``np.matrix(np.eye(self[0,:].size))`` all return
656: (12)          ``True``.
657: (8)          Raises
658: (8)          -----
659: (8)          numpy.linalg.LinAlgError: Singular matrix
660: (12)          If `self` is singular.
661: (8)          See Also
662: (8)          -----
663: (8)          linalg.inv
664: (8)          Examples
665: (8)          -----
666: (8)          >>> m = np.matrix('1, 2; 3, 4'); m
667: (8)          matrix([[1, 2],
668: (16)          [3, 4]])
669: (8)          >>> m.getI()
670: (8)          matrix([[-2.,  1.],
671: (16)          [ 1.5, -0.5]])

```

```

672: (8)             >>> m.getI() * m
673: (8)             matrix([[ 1.,  0.], # may vary
674: (16)             [ 0.,  1.]])
675: (8)
676: (8)             """
677: (8)             M, N = self.shape
678: (12)            if M == N:
679: (8)                 from numpy.linalg import inv as func
680: (12)            else:
681: (8)                 from numpy.linalg import pinv as func
682: (4)             return asmatrix(func(self))
683: (4)
684: (8)             @property
685: (8)             def A(self):
686: (8)                 """
687: (8)                 Return `self` as an `ndarray` object.
688: (8)                 Equivalent to ``np.asarray(self)``.
689: (8)                 Parameters
690: (8)                 -----
691: (8)                 None
692: (8)                 Returns
693: (8)                 -----
694: (8)                 ret : ndarray
695: (8)                 `self` as an `ndarray`
696: (8)             Examples
697: (8)             -----
698: (16)            >>> x = np.matrix(np.arange(12).reshape((3,4))); x
699: (16)            matrix([[ 0,  1,  2,  3],
700: (16)              [ 4,  5,  6,  7],
701: (16)              [ 8,  9, 10, 11]])
702: (8)            >>> x.getA()
703: (15)            array([[ 0,  1,  2,  3],
704: (15)              [ 4,  5,  6,  7],
705: (8)              [ 8,  9, 10, 11]])
706: (4)
707: (4)             return self.__array__()
708: (8)             @property
709: (8)             def A1(self):
710: (8)                 """
711: (8)                 Return `self` as a flattened `ndarray`.
712: (8)                 Equivalent to ``np.asarray(x).ravel()``.
713: (8)                 Parameters
714: (8)                 -----
715: (8)                 None
716: (8)                 Returns
717: (8)                 -----
718: (8)                 ret : ndarray
719: (8)                 `self` , 1-D, as an `ndarray`
720: (8)             Examples
721: (8)             -----
722: (16)            >>> x = np.matrix(np.arange(12).reshape((3,4))); x
723: (16)            matrix([[ 0,  1,  2,  3],
724: (16)              [ 4,  5,  6,  7],
725: (16)              [ 8,  9, 10, 11]])
726: (8)            >>> x.getA1()
727: (8)            array([ 0,  1,  2, ...,  9, 10, 11])
728: (4)
729: (8)             return self.__array__().ravel()
730: (8)             def ravel(self, order='C'):
731: (8)                 """
732: (8)                 Return a flattened matrix.
733: (8)                 Refer to `numpy.ravel` for more documentation.
734: (8)                 Parameters
735: (8)                 -----
736: (8)                 order : {'C', 'F', 'A', 'K'}, optional
737: (12)                The elements of `m` are read using this index order. 'C' means to
738: (12)                index the elements in C-like order, with the last axis index
739: (12)                changing fastest, back to the first axis index changing slowest.
740: (12)                'F' means to index the elements in Fortran-like index order, with
    
```

```

741: (12)           memory layout of the underlying array, and only refer to the order
742: (12)           of axis indexing. 'A' means to read the elements in Fortran-like
743: (12)           index order if `m` is Fortran *contiguous* in memory, C-like order
744: (12)           otherwise. 'K' means to read the elements in the order they occur
745: (12)           in memory, except for reversing the data when strides are
negative.

746: (12)           By default, 'C' index order is used.

747: (8)           Returns
748: (8)
749: (8)
750: (12)           -----
751: (12)           ret : matrix
752: (12)           Return the matrix flattened to shape `(1, N)` where `N`
753: (12)           is the number of elements in the original matrix.
754: (12)           A copy is made only if necessary.

755: (8)           See Also
756: (8)
757: (8)
758: (8)
759: (8)
760: (4)           -----
761: (4)           @property
762: (8)           def T(self):
763: (8)           """
764: (8)           Returns the transpose of the matrix.
765: (8)           Does *not* conjugate! For the complex conjugate transpose, use
766: (8)           ``.H``.

767: (8)           Parameters
768: (8)
769: (8)
770: (8)
771: (12)           -----
772: (8)           ret : matrix object
773: (8)           The (non-conjugated) transpose of the matrix.

774: (8)           See Also
775: (8)
776: (8)
777: (8)           -----
778: (8)           transpose, getH
779: (8)           Examples
780: (16)
781: (8)
782: (8)
783: (16)
784: (8)
785: (8)
786: (4)           -----
787: (4)           @property
788: (8)           def H(self):
789: (8)           """
790: (8)           Returns the (complex) conjugate transpose of `self`.
791: (8)           Equivalent to ``np.transpose(self).conjugate()`` if `self` is real-valued.

792: (8)           Parameters
793: (8)
794: (8)
795: (8)
796: (8)
797: (12)           -----
798: (8)           ret : matrix object
799: (8)           complex conjugate transpose of `self`

800: (8)           Examples
801: (8)
802: (8)
803: (16)
804: (16)
805: (8)
806: (8)
807: (16)

```

```

808: (16)           [ 2. +2.j,  6. +6.j, 10.+10.j],
809: (16)           [ 3. +3.j,  7. +7.j, 11.+11.j]])
810: (8)
811: (8)
812: (12)           if issubclass(self.dtype.type, N.complexfloating):
813: (8)               return self.transpose().conjugate()
814: (12)           else:
815: (4)               return self.transpose()
816: (4)           getT = T.fget
817: (4)           getA = A.fget
818: (4)           getA1 = A1.fget
819: (4)           getH = H.fget
820: (0)           getI = I.fget
821: (4)           def _from_string(str, gdict, ldict):
822: (4)               rows = str.split(';')
823: (4)               rowtup = []
824: (8)               for row in rows:
825: (8)                   trow = row.split(',')
826: (8)                   newrow = []
827: (12)                   for x in trow:
828: (8)                       newrow.extend(x.split())
829: (8)                   trow = newrow
830: (8)                   coltup = []
831: (12)                   for col in trow:
832: (12)                       col = col.strip()
833: (16)                       try:
834: (12)                           thismat = ldict[col]
835: (16)                       except KeyError:
836: (20)                           try:
837: (16)                               thismat = gdict[col]
838: (20)                           except KeyError as e:
839: (12)                               raise NameError(f"name {col!r} is not defined") from None
840: (8)                               coltup.append(thismat)
841: (4)                               rowtup.append(concatenate(coltup, axis=-1))
842: (0)               return concatenate(rowtup, axis=0)
843: (0)           @set_module('numpy')
844: (4)           def bmat(obj, ldict=None, gdict=None):
845: (4)               """
846: (4)               Build a matrix object from a string, nested sequence, or array.
847: (4)               Parameters
848: (4)               -----
849: (8)                   obj : str or array_like
850: (8)                   Input data. If a string, variables in the current scope may be
851: (8)                   referenced by name.
852: (4)                   ldict : dict, optional
853: (8)                   A dictionary that replaces local operands in current frame.
854: (8)                   Ignored if `obj` is not a string or `gdict` is None.
855: (4)                   gdict : dict, optional
856: (8)                   A dictionary that replaces global operands in current frame.
857: (8)                   Ignored if `obj` is not a string.
858: (4)               Returns
859: (4)               -----
860: (8)                   out : matrix
861: (8)                   Returns a matrix object, which is a specialized 2-D array.
862: (4)               See Also
863: (4)               -----
864: (8)                   block :
865: (8)                   A generalization of this function for N-d arrays, that returns normal
866: (4)                   ndarrays.
867: (4)               Examples
868: (4)               -----
869: (4)                   >>> A = np.mat('1 1; 1 1')
870: (4)                   >>> B = np.mat('2 2; 2 2')
871: (4)                   >>> C = np.mat('3 4; 5 6')
872: (4)                   >>> D = np.mat('7 8; 9 0')
873: (4)               All the following expressions construct the same block matrix:
874: (4)                   >>> np.bmat([[A, B], [C, D]])
875: (4)                   matrix([[1, 1, 2, 2],
876: (12)                           [1, 1, 2, 2],
876: (12)                           [3, 4, 7, 8],

```

```

877: (12)          [5, 6, 9, 0]])
878: (4)          >>> np.bmat(np.r_[np.c_[A, B], np.c_[C, D]])
879: (4)          matrix([[1, 1, 2, 2],
880: (12)             [1, 1, 2, 2],
881: (12)             [3, 4, 7, 8],
882: (12)             [5, 6, 9, 0]])
883: (4)          >>> np.bmat('A,B; C,D')
884: (4)          matrix([[1, 1, 2, 2],
885: (12)             [1, 1, 2, 2],
886: (12)             [3, 4, 7, 8],
887: (12)             [5, 6, 9, 0]])
888: (4)          """
889: (4)          if isinstance(obj, str):
890: (8)              if gdict is None:
891: (12)                  frame = sys._getframe().f_back
892: (12)                  glob_dict = frame.f_globals
893: (12)                  loc_dict = frame.f_locals
894: (8)              else:
895: (12)                  glob_dict = gdict
896: (12)                  loc_dict = ldict
897: (8)              return matrix(_from_string(obj, glob_dict, loc_dict))
898: (4)          if isinstance(obj, (tuple, list)):
899: (8)              arr_rows = []
900: (8)              for row in obj:
901: (12)                  if isinstance(row, N.ndarray): # not 2-d
902: (16)                      return matrix(concatenate(obj, axis=-1))
903: (12)                  else:
904: (16)                      arr_rows.append(concatenate(row, axis=-1))
905: (8)              return matrix(concatenate(arr_rows, axis=0))
906: (4)          if isinstance(obj, N.ndarray):
907: (8)              return matrix(obj)
908: (0)          mat = asmatrix

```

-----

## File 277 - setup.py:

```

1: (0)          def configuration(parent_package='', top_path=None):
2: (4)              from numpy.distutils.misc_util import Configuration
3: (4)              config = Configuration('matrixlib', parent_package, top_path)
4: (4)              config.add_subpackage('tests')
5: (4)              config.add_data_files('*/*.pyi')
6: (4)              return config
7: (0)          if __name__ == "__main__":
8: (4)              from numpy.distutils.core import setup
9: (4)              config = configuration(top_path='').todict()
10: (4)              setup(**config)

```

-----

## File 278 - \_\_init\_\_.py:

```

1: (0)          """Sub-package containing the matrix class and related functions.
2: (0)          """
3: (0)          from . import defmatrix
4: (0)          from .defmatrix import *
5: (0)          __all__ = defmatrix.__all__
6: (0)          from numpy._pytesttester import PytestTester
7: (0)          test = PytestTester(__name__)
8: (0)          del PytestTester

```

-----

## File 279 - test\_defmatrix.py:

```

1: (0)          import collections.abc
2: (0)          import numpy as np
3: (0)          from numpy import matrix, asmatrix, bmat
4: (0)          from numpy.testing import (

```

```

5: (4)             assert_, assert_equal, assert_almost_equal, assert_array_equal,
6: (4)             assert_array_almost_equal, assert_raises
7: (4)
8: (0)             from numpy.linalg import matrix_power
9: (0)             from numpy.matrixlib import mat
10: (0)            class TestCtor:
11: (4)              def test_basic(self):
12: (8)                A = np.array([[1, 2], [3, 4]])
13: (8)                mA = matrix(A)
14: (8)                assert_(np.all(mA.A == A))
15: (8)                B = bmat("A,A;A,A")
16: (8)                C = bmat([[A, A], [A, A]])
17: (8)                D = np.array([[1, 2, 1, 2],
18: (22)                  [3, 4, 3, 4],
19: (22)                  [1, 2, 1, 2],
20: (22)                  [3, 4, 3, 4]])
21: (8)                assert_(np.all(B.A == D))
22: (8)                assert_(np.all(C.A == D))
23: (8)                E = np.array([[5, 6], [7, 8]])
24: (8)                AResult = matrix([[1, 2, 5, 6], [3, 4, 7, 8]])
25: (8)                assert_(np.all(bmat([A, E]) == AResult))
26: (8)                vec = np.arange(5)
27: (8)                mvec = matrix(vec)
28: (8)                assert_(mvec.shape == (1, 5))
29: (4)              def test_exceptions(self):
30: (8)                assert_raises(ValueError, matrix, "invalid")
31: (4)              def test_bmat_nondefault_str(self):
32: (8)                A = np.array([[1, 2], [3, 4]])
33: (8)                B = np.array([[5, 6], [7, 8]])
34: (8)                Aresult = np.array([[1, 2, 1, 2],
35: (28)                  [3, 4, 3, 4],
36: (28)                  [1, 2, 1, 2],
37: (28)                  [3, 4, 3, 4]])
38: (8)                mixresult = np.array([[1, 2, 5, 6],
39: (30)                  [3, 4, 7, 8],
40: (30)                  [5, 6, 1, 2],
41: (30)                  [7, 8, 3, 4]])
42: (8)                assert_(np.all(bmat("A,A;A,A") == Aresult))
43: (8)                assert_(np.all(bmat("A,A;A,A", ldict={'A':B}) == Aresult))
44: (8)                assert_raises(TypeError, bmat, "A,A;A,A", gdict={'A':B})
45: (8)                assert_
46: (12)                  np.all(bmat("A,A;A,A", ldict={'A':A}, gdict={'A':B}) == Aresult))
47: (8)                b2 = bmat("A,B;C,D", ldict={'A':A, 'B':B}, gdict={'C':B, 'D':A})
48: (8)                assert_(np.all(b2 == mixresult))
49: (0)            class TestProperties:
50: (4)              def test_sum(self):
51: (8)                """Test whether matrix.sum(axis=1) preserves orientation.
52: (8)                Fails in NumPy <= 0.9.6.2127.
53: (8)
54: (8)                M = matrix([[1, 2, 0, 0],
55: (19)                  [3, 4, 0, 0],
56: (19)                  [1, 2, 1, 2],
57: (19)                  [3, 4, 3, 4]])
58: (8)                sum0 = matrix([8, 12, 4, 6])
59: (8)                sum1 = matrix([3, 7, 6, 14]).T
60: (8)                sumall = 30
61: (8)                assert_array_equal(sum0, M.sum(axis=0))
62: (8)                assert_array_equal(sum1, M.sum(axis=1))
63: (8)                assert_equal(sumall, M.sum())
64: (8)                assert_array_equal(sum0, np.sum(M, axis=0))
65: (8)                assert_array_equal(sum1, np.sum(M, axis=1))
66: (8)                assert_equal(sumall, np.sum(M))
67: (4)              def test_prod(self):
68: (8)                x = matrix([[1, 2, 3], [4, 5, 6]])
69: (8)                assert_equal(x.prod(), 720)
70: (8)                assert_equal(x.prod(0), matrix([[4, 10, 18]]))
71: (8)                assert_equal(x.prod(1), matrix([[6], [120]]))
72: (8)                assert_equal(np.prod(x), 720)
73: (8)                assert_equal(np.prod(x, axis=0), matrix([[4, 10, 18]]))

```

```

74: (8)             assert_equal(np.prod(x, axis=1), matrix([[6], [120]]))
75: (8)             y = matrix([0, 1, 3])
76: (8)             assert_(y.prod() == 0)
77: (4)             def test_max(self):
78: (8)                 x = matrix([[1, 2, 3], [4, 5, 6]])
79: (8)                 assert_equal(x.max(), 6)
80: (8)                 assert_equal(x.max(0), matrix([[4, 5, 6]]))
81: (8)                 assert_equal(x.max(1), matrix([[3], [6]]))
82: (8)                 assert_equal(np.max(x), 6)
83: (8)                 assert_equal(np.max(x, axis=0), matrix([[4, 5, 6]]))
84: (8)                 assert_equal(np.max(x, axis=1), matrix([[3], [6]]))
85: (4)             def test_min(self):
86: (8)                 x = matrix([[1, 2, 3], [4, 5, 6]])
87: (8)                 assert_equal(x.min(), 1)
88: (8)                 assert_equal(x.min(0), matrix([[1, 2, 3]]))
89: (8)                 assert_equal(x.min(1), matrix([[1], [4]]))
90: (8)                 assert_equal(np.min(x), 1)
91: (8)                 assert_equal(np.min(x, axis=0), matrix([[1, 2, 3]]))
92: (8)                 assert_equal(np.min(x, axis=1), matrix([[1], [4]]))
93: (4)             def test_ptp(self):
94: (8)                 x = np.arange(4).reshape((2, 2))
95: (8)                 assert_(x.ptp() == 3)
96: (8)                 assert_(np.all(x.ptp(0) == np.array([2, 2])))
97: (8)                 assert_(np.all(x.ptp(1) == np.array([1, 1])))
98: (4)             def test_var(self):
99: (8)                 x = np.arange(9).reshape((3, 3))
100: (8)                mx = x.view(np.matrix)
101: (8)                assert_equal(x.var(ddof=0), mx.var(ddof=0))
102: (8)                assert_equal(x.var(ddof=1), mx.var(ddof=1))
103: (4)             def test_basic(self):
104: (8)                 import numpy.linalg as linalg
105: (8)                 A = np.array([[1., 2.],
106: (22)                           [3., 4.]])
107: (8)                 mA = matrix(A)
108: (8)                 assert_(np.allclose(linalg.inv(A), mA.I))
109: (8)                 assert_(np.all(np.array(np.transpose(A)) == mA.T)))
110: (8)                 assert_(np.all(np.array(np.transpose(A)) == mA.H)))
111: (8)                 assert_(np.all(A == mA.A))
112: (8)                 B = A + 2j*A
113: (8)                 mB = matrix(B)
114: (8)                 assert_(np.allclose(linalg.inv(B), mB.I))
115: (8)                 assert_(np.all(np.array(np.transpose(B)) == mB.T)))
116: (8)                 assert_(np.all(np.array(np.transpose(B).conj()) == mB.H)))
117: (4)             def test_pinv(self):
118: (8)                 x = matrix(np.arange(6).reshape(2, 3))
119: (8)                 xpinv = matrix([[-0.77777778,  0.27777778],
120: (24)                               [-0.11111111,  0.11111111],
121: (24)                               [ 0.55555556, -0.05555556]])
122: (8)                 assert_almost_equal(x.I, xpinv)
123: (4)             def test_comparisons(self):
124: (8)                 A = np.arange(100).reshape(10, 10)
125: (8)                 mA = matrix(A)
126: (8)                 mB = matrix(A) + 0.1
127: (8)                 assert_(np.all(mB == A+0.1))
128: (8)                 assert_(np.all(mB == matrix(A+0.1)))
129: (8)                 assert_(not np.any(mB == matrix(A-0.1)))
130: (8)                 assert_(np.all(mA < mB))
131: (8)                 assert_(np.all(mA <= mB))
132: (8)                 assert_(np.all(mA <= mA))
133: (8)                 assert_(not np.any(mA < mA))
134: (8)                 assert_(not np.any(mB < mA))
135: (8)                 assert_(np.all(mB >= mA))
136: (8)                 assert_(np.all(mB >= mB))
137: (8)                 assert_(not np.any(mB > mB))
138: (8)                 assert_(np.all(mA == mA))
139: (8)                 assert_(not np.any(mA == mB))
140: (8)                 assert_(np.all(mB != mA))
141: (8)                 assert_(not np.all(abs(mA) > 0))
142: (8)                 assert_(np.all(abs(mB) > 0)))

```

```

143: (4)             def test_asmatrix(self):
144: (8)                 A = np.arange(100).reshape(10, 10)
145: (8)                 mA = asmatrix(A)
146: (8)                 A[0, 0] = -10
147: (8)                 assert_(A[0, 0] == mA[0, 0])
148: (4)             def test_noaxis(self):
149: (8)                 A = matrix([[1, 0], [0, 1]])
150: (8)                 assert_(A.sum() == matrix(2))
151: (8)                 assert_(A.mean() == matrix(0.5))
152: (4)             def test_repr(self):
153: (8)                 A = matrix([[1, 0], [0, 1]])
154: (8)                 assert_(repr(A) == "matrix([[1, 0],\n                           [0, 1]]))")
155: (4)             def test_make_bool_matrix_from_str(self):
156: (8)                 A = matrix('True; True; False')
157: (8)                 B = matrix([[True], [True], [False]])
158: (8)                 assert_array_equal(A, B)
159: (0)         class TestCasting:
160: (4)             def test_basic(self):
161: (8)                 A = np.arange(100).reshape(10, 10)
162: (8)                 mA = matrix(A)
163: (8)                 mB = mA.copy()
164: (8)                 O = np.ones((10, 10), np.float64) * 0.1
165: (8)                 mB = mB + O
166: (8)                 assert_(mB.dtype.type == np.float64)
167: (8)                 assert_(np.all(mA != mB))
168: (8)                 assert_(np.all(mB == mA+0.1))
169: (8)                 mC = mA.copy()
170: (8)                 O = np.ones((10, 10), np.complex128)
171: (8)                 mC = mC * O
172: (8)                 assert_(mC.dtype.type == np.complex128)
173: (8)                 assert_(np.all(mA != mB))
174: (0)         class TestAlgebra:
175: (4)             def test_basic(self):
176: (8)                 import numpy.linalg as linalg
177: (8)                 A = np.array([[1., 2.], [3., 4.]])
178: (8)                 mA = matrix(A)
179: (8)                 B = np.identity(2)
180: (8)                 for i in range(6):
181: (12)                     assert_(np.allclose((mA ** i).A, B))
182: (12)                     B = np.dot(B, A)
183: (8)                 Ainv = linalg.inv(A)
184: (8)                 B = np.identity(2)
185: (8)                 for i in range(6):
186: (12)                     assert_(np.allclose((mA ** -i).A, B))
187: (12)                     B = np.dot(B, Ainv)
188: (8)                 assert_(np.allclose((mA * mA).A, np.dot(A, A)))
189: (8)                 assert_(np.allclose((mA + mA).A, (A + A)))
190: (8)                 assert_(np.allclose((3*mA).A, (3*A)))
191: (8)                 mA2 = matrix(A)
192: (8)                 mA2 *= 3
193: (8)                 assert_(np.allclose(mA2.A, 3*A))
194: (4)             def test_pow(self):
195: (8)                 """Test raising a matrix to an integer power works as expected."""
196: (8)                 m = matrix("1. 2.; 3. 4.")
197: (8)                 m2 = m.copy()
198: (8)                 m2 **= 2
199: (8)                 mi = m.copy()
200: (8)                 mi **= -1
201: (8)                 m4 = m2.copy()
202: (8)                 m4 **= 2
203: (8)                 assert_array_almost_equal(m2, m**2)
204: (8)                 assert_array_almost_equal(m4, np.dot(m2, m2))
205: (8)                 assert_array_almost_equal(np.dot(mi, m), np.eye(2))
206: (4)             def test_scalar_type_pow(self):
207: (8)                 m = matrix([[1, 2], [3, 4]])
208: (8)                 for scalar_t in [np.int8, np.uint8]:
209: (12)                     two = scalar_t(2)
210: (12)                     assert_array_almost_equal(m ** 2, m ** two)
211: (4)             def test_notimplemented(self):

```

```

212: (8)             '''Check that 'not implemented' operations produce a failure.'''
213: (8)             A = matrix([[1., 2.],
214: (20)                 [3., 4.]])
215: (8)             with assert_raises(TypeError):
216: (12)                 1.0**A
217: (8)             with assert_raises(TypeError):
218: (12)                 A*object()
219: (0)             class TestMatrixReturn:
220: (4)                 def test_instance_methods(self):
221: (8)                     a = matrix([1.0], dtype='f8')
222: (8)                     methodargs = {
223: (12)                         'astype': ('intc',),
224: (12)                         'clip': (0.0, 1.0),
225: (12)                         'compress': ([1],),
226: (12)                         'repeat': (1,),
227: (12)                         'reshape': (1,),
228: (12)                         'swapaxes': (0, 0),
229: (12)                         'dot': np.array([1.0]),
230: (12)                     }
231: (8)                     excluded_methods = [
232: (12)                         'argmin', 'choose', 'dump', 'dumps', 'fill', 'getfield',
233: (12)                         'getA', 'getA1', 'item', 'nonzero', 'put', 'putmask', 'resize',
234: (12)                         'searchsorted', 'setflags', 'setfield', 'sort',
235: (12)                         'partition', 'argpartition',
236: (12)                         'take', 'tofile', 'tolist', 'tostring', 'tobytes', 'all', 'any',
237: (12)                         'sum', 'argmax', 'argmin', 'min', 'max', 'mean', 'var', 'ptp',
238: (12)                         'prod', 'std', 'ctypes', 'itemset',
239: (12)                     ]
240: (8)                     for attrib in dir(a):
241: (12)                         if attrib.startswith('_') or attrib in excluded_methods:
242: (16)                             continue
243: (12)                         f = getattr(a, attrib)
244: (12)                         if isinstance(f, collections.abc.Callable):
245: (16)                             a.astype('f8')
246: (16)                             a.fill(1.0)
247: (16)                             if attrib in methodargs:
248: (20)                                 args = methodargs[attrib]
249: (16)                             else:
250: (20)                                 args = ()
251: (16)                             b = f(*args)
252: (16)                             assert_(type(b) is matrix, "%s" % attrib)
253: (8)                             assert_(type(a.real) is matrix)
254: (8)                             assert_(type(a.imag) is matrix)
255: (8)                             c, d = matrix([0.0]).nonzero()
256: (8)                             assert_(type(c) is np.ndarray)
257: (8)                             assert_(type(d) is np.ndarray)
258: (0)             class TestIndexing:
259: (4)                 def test_basic(self):
260: (8)                     x = asmatrix(np.zeros((3, 2), float))
261: (8)                     y = np.zeros((3, 1), float)
262: (8)                     y[:, 0] = [0.8, 0.2, 0.3]
263: (8)                     x[:, 1] = y > 0.5
264: (8)                     assert_equal(x, [[0, 1], [0, 0], [0, 0]])
265: (0)             class TestNewScalarIndexing:
266: (4)                 a = matrix([[1, 2], [3, 4]])
267: (4)                 def test_dimesions(self):
268: (8)                     a = self.a
269: (8)                     x = a[0]
270: (8)                     assert_equal(x.ndim, 2)
271: (4)                     def test_array_from_matrix_list(self):
272: (8)                         a = self.a
273: (8)                         x = np.array([a, a])
274: (8)                         assert_equal(x.shape, [2, 2, 2])
275: (4)                     def test_array_to_list(self):
276: (8)                         a = self.a
277: (8)                         assert_equal(a.tolist(), [[1, 2], [3, 4]])
278: (4)                     def test_fancy_indexing(self):
279: (8)                         a = self.a
280: (8)                         x = a[1, [0, 1, 0]]

```

```

281: (8)             assert_(isinstance(x, matrix))
282: (8)             assert_equal(x, matrix([[3, 4, 3]]))
283: (8)             x = a[[1, 0]]
284: (8)             assert_(isinstance(x, matrix))
285: (8)             assert_equal(x, matrix([[3, 4], [1, 2]]))
286: (8)             x = a[[[1], [0]], [[1, 0], [0, 1]]]
287: (8)             assert_(isinstance(x, matrix))
288: (8)             assert_equal(x, matrix([[4, 3], [1, 2]]))
289: (4)             def test_matrix_element(self):
290: (8)                 x = matrix([[1, 2, 3], [4, 5, 6]])
291: (8)                 assert_equal(x[0][0], matrix([[1, 2, 3]]))
292: (8)                 assert_equal(x[0][0].shape, (1, 3))
293: (8)                 assert_equal(x[0].shape, (1, 3))
294: (8)                 assert_equal(x[:, 0].shape, (2, 1))
295: (8)                 x = matrix(0)
296: (8)                 assert_equal(x[0, 0], 0)
297: (8)                 assert_equal(x[0], 0)
298: (8)                 assert_equal(x[:, 0].shape, x.shape)
299: (4)             def test_scalar_indexing(self):
300: (8)                 x = asmatrix(np.zeros((3, 2), float))
301: (8)                 assert_equal(x[0, 0], x[0][0])
302: (4)             def test_row_column_indexing(self):
303: (8)                 x = asmatrix(np.eye(2))
304: (8)                 assert_array_equal(x[0,:], [[1, 0]])
305: (8)                 assert_array_equal(x[1,:], [[0, 1]])
306: (8)                 assert_array_equal(x[:, 0], [[1], [0]])
307: (8)                 assert_array_equal(x[:, 1], [[0], [1]])
308: (4)             def test_boolean_indexing(self):
309: (8)                 A = np.arange(6)
310: (8)                 A.shape = (3, 2)
311: (8)                 x = asmatrix(A)
312: (8)                 assert_array_equal(x[:, np.array([True, False])], x[:, 0])
313: (8)                 assert_array_equal(x[np.array([True, False, False]),:], x[0,:])
314: (4)             def test_list_indexing(self):
315: (8)                 A = np.arange(6)
316: (8)                 A.shape = (3, 2)
317: (8)                 x = asmatrix(A)
318: (8)                 assert_array_equal(x[:, [1, 0]], x[:, ::-1])
319: (8)                 assert_array_equal(x[[2, 1, 0],:], x[::-1,:])
320: (0)             class TestPower:
321: (4)                 def test_returntype(self):
322: (8)                     a = np.array([[0, 1], [0, 0]])
323: (8)                     assert_(type(matrix_power(a, 2)) is np.ndarray)
324: (8)                     a = mat(a)
325: (8)                     assert_(type(matrix_power(a, 2)) is matrix)
326: (4)                 def test_list(self):
327: (8)                     assert_array_equal(matrix_power([[0, 1], [0, 0]], 2), [[0, 0], [0, 0]])
328: (0)             class TestShape:
329: (4)                 a = np.array([[1], [2]])
330: (4)                 m = matrix([[1], [2]])
331: (4)                 def test_shape(self):
332: (8)                     assert_equal(self.a.shape, (2, 1))
333: (8)                     assert_equal(self.m.shape, (2, 1))
334: (4)                 def test_numpy_ravel(self):
335: (8)                     assert_equal(np.ravel(self.a).shape, (2,))
336: (8)                     assert_equal(np.ravel(self.m).shape, (2,))
337: (4)                 def test_member_ravel(self):
338: (8)                     assert_equal(self.a.ravel().shape, (2,))
339: (8)                     assert_equal(self.m.ravel().shape, (1, 2))
340: (4)                 def test_member_flatten(self):
341: (8)                     assert_equal(self.a.flatten().shape, (2,))
342: (8)                     assert_equal(self.m.flatten().shape, (1, 2))
343: (4)                 def test_numpy_ravel_order(self):
344: (8)                     x = np.array([[1, 2, 3], [4, 5, 6]])
345: (8)                     assert_equal(np.ravel(x), [1, 2, 3, 4, 5, 6])
346: (8)                     assert_equal(np.ravel(x, order='F'), [1, 4, 2, 5, 3, 6])
347: (8)                     assert_equal(np.ravel(x.T), [1, 4, 2, 5, 3, 6])
348: (8)                     assert_equal(np.ravel(x.T, order='A'), [1, 2, 3, 4, 5, 6])

```

```

349: (8)          x = matrix([[1, 2, 3], [4, 5, 6]])
350: (8)          assert_equal(np.ravel(x), [1, 2, 3, 4, 5, 6])
351: (8)          assert_equal(np.ravel(x, order='F'), [1, 4, 2, 5, 3, 6])
352: (8)          assert_equal(np.ravel(x.T), [1, 4, 2, 5, 3, 6])
353: (8)          assert_equal(np.ravel(x.T, order='A'), [1, 2, 3, 4, 5, 6])
354: (4)          def test_matrix_ravel_order(self):
355: (8)              x = matrix([[1, 2, 3], [4, 5, 6]])
356: (8)              assert_equal(x.ravel(), [[1, 2, 3, 4, 5, 6]])
357: (8)              assert_equal(x.ravel(order='F'), [[1, 4, 2, 5, 3, 6]])
358: (8)              assert_equal(x.T.ravel(), [[1, 4, 2, 5, 3, 6]])
359: (8)              assert_equal(x.T.ravel(order='A'), [[1, 2, 3, 4, 5, 6]])
360: (4)          def test_array_memory_sharing(self):
361: (8)              assert_(np.may_share_memory(self.a, self.a.ravel()))
362: (8)              assert_(not np.may_share_memory(self.a, self.a.flatten()))
363: (4)          def test_matrix_memory_sharing(self):
364: (8)              assert_(np.may_share_memory(self.m, self.m.ravel()))
365: (8)              assert_(not np.may_share_memory(self.m, self.m.flatten()))
366: (4)          def test_expand_dims_matrix(self):
367: (8)              a = np.arange(10).reshape((2, 5)).view(np.matrix)
368: (8)              expanded = np.expand_dims(a, axis=1)
369: (8)              assert_equal(expanded.ndim, 3)
370: (8)              assert_(not isinstance(expanded, np.matrix))

```

---

## File 280 - test\_interaction.py:

```

1: (0)          """Tests of interaction of matrix with other parts of numpy.
2: (0)          Note that tests with MaskedArray and linalg are done in separate files.
3: (0)          """
4: (0)          import pytest
5: (0)          import textwrap
6: (0)          import warnings
7: (0)          import numpy as np
8: (0)          from numpy.testing import (assert_, assert_equal, assert_raises,
9: (27)                  assert_raises_regex, assert_array_equal,
10: (27)                 assert_almost_equal, assert_array_almost_equal)
11: (0)          def test_fancy_indexing():
12: (4)              m = np.matrix([[1, 2], [3, 4]])
13: (4)              assert_(isinstance(m[[0, 1, 0], :], np.matrix))
14: (4)              x = np.asmatrix(np.arange(50).reshape(5, 10))
15: (4)              assert_equal(x[:2, np.array(-1)], x[:2, -1].T)
16: (0)          def test_polynomial_mapdomain():
17: (4)              dom1 = [0, 4]
18: (4)              dom2 = [1, 3]
19: (4)              x = np.matrix([dom1, dom1])
20: (4)              res = np.polynomial.polyutils.mapdomain(x, dom1, dom2)
21: (4)              assert_(isinstance(res, np.matrix))
22: (0)          def test_sort_matrix_none():
23: (4)              a = np.matrix([[2, 1, 0]])
24: (4)              actual = np.sort(a, axis=None)
25: (4)              expected = np.matrix([[0, 1, 2]])
26: (4)              assert_equal(actual, expected)
27: (4)              assert_(type(expected) is np.matrix)
28: (0)          def test_partition_matrix_none():
29: (4)              a = np.matrix([[2, 1, 0]])
30: (4)              actual = np.partition(a, 1, axis=None)
31: (4)              expected = np.matrix([[0, 1, 2]])
32: (4)              assert_equal(actual, expected)
33: (4)              assert_(type(expected) is np.matrix)
34: (0)          def test_dot_scalar_and_matrix_of_objects():
35: (4)              arr = np.matrix([1, 2], dtype=object)
36: (4)              desired = np.matrix([[3, 6]], dtype=object)
37: (4)              assert_equal(np.dot(arr, 3), desired)
38: (4)              assert_equal(np.dot(3, arr), desired)
39: (0)          def test_inner_scalar_and_matrix():
40: (4)              for dt in np.typecodes['AllInteger'] + np.typecodes['AllFloat'] + '?':
41: (8)                  sca = np.array(3, dtype=dt)[()]
42: (8)                  arr = np.matrix([[1, 2], [3, 4]], dtype=dt)

```

```

43: (8)             desired = np.matrix([[3, 6], [9, 12]], dtype=dt)
44: (8)             assert_equal(np.inner(arr, sca), desired)
45: (8)             assert_equal(np.inner(sca, arr), desired)
46: (0)             def test_inner_scalar_and_matrix_of_objects():
47: (4)                 arr = np.matrix([1, 2], dtype=object)
48: (4)                 desired = np.matrix([[3, 6]], dtype=object)
49: (4)                 assert_equal(np.inner(arr, 3), desired)
50: (4)                 assert_equal(np.inner(3, arr), desired)
51: (0)             def test_iter_allocate_output_subtype():
52: (4)                 a = np.matrix([[1, 2], [3, 4]])
53: (4)                 b = np.arange(4).reshape(2, 2).T
54: (4)                 i = np.nditer([a, b, None], [],
55: (18)                   [['readonly'], ['readonly'], ['writeonly', 'allocate']])
56: (4)                 assert_(type(i.operands[2]) is np.matrix)
57: (4)                 assert_(type(i.operands[2]) is not np.ndarray)
58: (4)                 assert_equal(i.operands[2].shape, (2, 2))
59: (4)                 b = np.arange(4).reshape(1, 2, 2)
60: (4)                 assert_raises(RuntimeError, np.nditer, [a, b, None], [],
61: (18)                   [['readonly'], ['readonly'], ['writeonly', 'allocate']])
62: (4)                 i = np.nditer([a, b, None], [],
63: (18)                   [['readonly'], ['readonly'],
64: (19)                     ['writeonly', 'allocate', 'no_subtype']])
65: (4)                 assert_(type(i.operands[2]) is np.ndarray)
66: (4)                 assert_(type(i.operands[2]) is not np.matrix)
67: (4)                 assert_equal(i.operands[2].shape, (1, 2, 2))
68: (0)             def like_function():
69: (4)                 a = np.matrix([[1, 2], [3, 4]])
70: (4)                 for like_function in np.zeros_like, np.ones_like, np.empty_like:
71: (8)                     b = like_function(a)
72: (8)                     assert_(type(b) is np.matrix)
73: (8)                     c = like_function(a, subok=False)
74: (8)                     assert_(type(c) is not np.matrix)
75: (0)             def test_array_astype():
76: (4)                 a = np.matrix([[0, 1, 2], [3, 4, 5]], dtype='f4')
77: (4)                 b = a.astype('f4', subok=True, copy=False)
78: (4)                 assert_(a is b)
79: (4)                 b = a.astype('i4', copy=False)
80: (4)                 assert_equal(a, b)
81: (4)                 assert_equal(type(b), np.matrix)
82: (4)                 b = a.astype('f4', subok=False, copy=False)
83: (4)                 assert_equal(a, b)
84: (4)                 assert_(not (a is b))
85: (4)                 assert_(type(b) is not np.matrix)
86: (0)             def test_stack():
87: (4)                 m = np.matrix([[1, 2], [3, 4]])
88: (4)                 assert_raises_regex(ValueError, 'shape too large to be a matrix',
89: (24)                   np.stack, [m, m])
90: (0)             def test_object_scalar_multiply():
91: (4)                 arr = np.matrix([1, 2], dtype=object)
92: (4)                 desired = np.matrix([[3, 6]], dtype=object)
93: (4)                 assert_equal(np.multiply(arr, 3), desired)
94: (4)                 assert_equal(np.multiply(3, arr), desired)
95: (0)             def test_nanfunctions_matrices():
96: (4)                 mat = np.matrix(np.eye(3))
97: (4)                 for f in [np.nanmin, np.nanmax]:
98: (8)                     res = f(mat, axis=0)
99: (8)                     assert_(isinstance(res, np.matrix))
100: (8)                     assert_(res.shape == (1, 3))
101: (8)                     res = f(mat, axis=1)
102: (8)                     assert_(isinstance(res, np.matrix))
103: (8)                     assert_(res.shape == (3, 1))
104: (8)                     res = f(mat)
105: (8)                     assert_(np.isscalar(res))
106: (4)                 mat[1] = np.nan
107: (4)                 for f in [np.nanmin, np.nanmax]:
108: (8)                     with warnings.catch_warnings(record=True) as w:
109: (12)                         warnings.simplefilter('always')
110: (12)                         res = f(mat, axis=0)
111: (12)                         assert_(isinstance(res, np.matrix))

```

```

112: (12)                                assert_(not np.any(np.isnan(res)))
113: (12)                                assert_(len(w) == 0)
114: (8)                                 with warnings.catch_warnings(record=True) as w:
115: (12)                                     warnings.simplefilter('always')
116: (12)                                     res = f(mat, axis=1)
117: (12)                                     assert_(isinstance(res, np.matrix))
118: (12)                                     assert_(np.isnan(res[1, 0]) and not np.isnan(res[0, 0])
119: (20)                                         and not np.isnan(res[2, 0]))
120: (12)                                     assert_(len(w) == 1, 'no warning raised')
121: (12)                                     assert_(issubclass(w[0].category, RuntimeWarning))
122: (8)                                 with warnings.catch_warnings(record=True) as w:
123: (12)                                     warnings.simplefilter('always')
124: (12)                                     res = f(mat)
125: (12)                                     assert_(np.isscalar(res))
126: (12)                                     assert_(res != np.nan)
127: (12)                                     assert_(len(w) == 0)
128: (0)  def test_nanfunctions_matrices_general():
129: (4)      mat = np.matrix(np.eye(3))
130: (4)      for f in (np.nanargmin, np.nanargmax, np.nansum, np.nanprod,
131: (14)          np.nanmean, np.nanvar, np.nanstd):
132: (8)          res = f(mat, axis=0)
133: (8)          assert_(isinstance(res, np.matrix))
134: (8)          assert_(res.shape == (1, 3))
135: (8)          res = f(mat, axis=1)
136: (8)          assert_(isinstance(res, np.matrix))
137: (8)          assert_(res.shape == (3, 1))
138: (8)          res = f(mat)
139: (8)          assert_(np.isscalar(res))
140: (4)      for f in np.nancumsum, np.nancumprod:
141: (8)          res = f(mat, axis=0)
142: (8)          assert_(isinstance(res, np.matrix))
143: (8)          assert_(res.shape == (3, 3))
144: (8)          res = f(mat, axis=1)
145: (8)          assert_(isinstance(res, np.matrix))
146: (8)          assert_(res.shape == (3, 3))
147: (8)          res = f(mat)
148: (8)          assert_(isinstance(res, np.matrix))
149: (8)          assert_(res.shape == (1, 3*3))
150: (0)  def test_average_matrix():
151: (4)      y = np.matrix(np.random.rand(5, 5))
152: (4)      assert_array_equal(y.mean(0), np.average(y, 0))
153: (4)      a = np.matrix([[1, 2], [3, 4]])
154: (4)      w = np.matrix([[1, 2], [3, 4]])
155: (4)      r = np.average(a, axis=0, weights=w)
156: (4)      assert_equal(type(r), np.matrix)
157: (4)      assert_equal(r, [[2.5, 10.0/3]])
158: (0)  def test_trapz_matrix():
159: (4)      x = np.linspace(0, 5)
160: (4)      y = x * x
161: (4)      r = np.trapz(y, x)
162: (4)      mx = np.matrix(x)
163: (4)      my = np.matrix(y)
164: (4)      mr = np.trapz(my, mx)
165: (4)      assert_almost_equal(mr, r)
166: (0)  def test_ediff1d_matrix():
167: (4)      assert(isinstance(np.ediff1d(np.matrix(1)), np.matrix))
168: (4)      assert(isinstance(np.ediff1d(np.matrix(1), to_begin=1), np.matrix))
169: (0)  def test_apply_along_axis_matrix():
170: (4)      def double(row):
171: (8)          return row * 2
172: (4)      m = np.matrix([[0, 1], [2, 3]])
173: (4)      expected = np.matrix([[0, 2], [4, 6]])
174: (4)      result = np.apply_along_axis(double, 0, m)
175: (4)      assert_(isinstance(result, np.matrix))
176: (4)      assert_array_equal(result, expected)
177: (4)      result = np.apply_along_axis(double, 1, m)
178: (4)      assert_(isinstance(result, np.matrix))
179: (4)      assert_array_equal(result, expected)
180: (0)  def test_kron_matrix():

```

```

181: (4)             a = np.ones([2, 2])
182: (4)             m = np.asmatrix(a)
183: (4)             assert_equal(type(np.kron(a, a)), np.ndarray)
184: (4)             assert_equal(type(np.kron(m, m)), np.matrix)
185: (4)             assert_equal(type(np.kron(a, m)), np.matrix)
186: (4)             assert_equal(type(np.kron(m, a)), np.matrix)
187: (0)             class TestConcatenatorMatrix:
188: (4)                 def test_matrix(self):
189: (8)                     a = [1, 2]
190: (8)                     b = [3, 4]
191: (8)                     ab_r = np.r_['r', a, b]
192: (8)                     ab_c = np.r_['c', a, b]
193: (8)                     assert_equal(type(ab_r), np.matrix)
194: (8)                     assert_equal(type(ab_c), np.matrix)
195: (8)                     assert_equal(np.array(ab_r), [[1, 2, 3, 4]])
196: (8)                     assert_equal(np.array(ab_c), [[1], [2], [3], [4]])
197: (8)                     assert_raises(ValueError, lambda: np.r_['rc', a, b])
198: (4)                 def test_matrix_scalar(self):
199: (8)                     r = np.r_['r', [1, 2], 3]
200: (8)                     assert_equal(type(r), np.matrix)
201: (8)                     assert_equal(np.array(r), [[1, 2, 3]])
202: (4)                 def test_matrix_builder(self):
203: (8)                     a = np.array([1])
204: (8)                     b = np.array([2])
205: (8)                     c = np.array([3])
206: (8)                     d = np.array([4])
207: (8)                     actual = np.r_['a, b; c, d']
208: (8)                     expected = np.bmat([[a, b], [c, d]])
209: (8)                     assert_equal(actual, expected)
210: (8)                     assert_equal(type(actual), type(expected))
211: (0)                 def test_array_equal_error_message_matrix():
212: (4)                     with pytest.raises(AssertionError) as exc_info:
213: (8)                         assert_equal(np.array([1, 2]), np.matrix([1, 2]))
214: (4)                         msg = str(exc_info.value)
215: (4)                         msg_reference = textwrap.dedent("""\
216: (4)                         Arrays are not equal
217: (4)                         (shapes (2,), (1, 2) mismatch)
218: (5)                         x: array([1, 2])
219: (5)                         y: matrix([[1, 2]]""")
220: (4)                         assert_equal(msg, msg_reference)
221: (0)                 def test_array_almost_equal_matrix():
222: (4)                     m1 = np.matrix([[1., 2.]])
223: (4)                     m2 = np.matrix([[1., np.nan]])
224: (4)                     m3 = np.matrix([[1., -np.inf]])
225: (4)                     m4 = np.matrix([[np.nan, np.inf]])
226: (4)                     m5 = np.matrix([[1., 2.], [np.nan, np.inf]])
227: (4)                     for assert_func in assert_array_almost_equal, assert_almost_equal:
228: (8)                         for m in m1, m2, m3, m4, m5:
229: (12)                             assert_func(m, m)
230: (12)                             a = np.array(m)
231: (12)                             assert_func(a, m)
232: (12)                             assert_func(m, a)

```

-----

## File 281 - test\_masked\_matrix.py:

```

1: (0)             import numpy as np
2: (0)             from numpy.testing import assert_warnings
3: (0)             from numpy.ma.testutils import (assert_, assert_equal, assert_raises,
4: (32)                           assert_array_equal)
5: (0)             from numpy.ma.core import (masked_array, masked_values, masked, allequal,
6: (27)                           MaskType, getmask, MaskedArray, nomask,
7: (27)                           log, add, hypot, divide)
8: (0)             from numpy.ma.extras import mr_
9: (0)             from numpy.compat import pickle
10: (0)             class MMatrix(MaskedArray, np.matrix,):
11: (4)                 def __new__(cls, data, mask=nomask):
12: (8)                     mat = np.matrix(data)

```

```

13: (8)             _data = MaskedArray.__new__(cls, data=mat, mask=mask)
14: (8)             return _data
15: (4)             def __array_finalize__(self, obj):
16: (8)                 np.matrix.__array_finalize__(self, obj)
17: (8)                 MaskedArray.__array_finalize__(self, obj)
18: (8)                 return
19: (4)             @property
20: (4)             def _series(self):
21: (8)                 _view = self.view(MaskedArray)
22: (8)                 _view._sharedmask = False
23: (8)                 return _view
24: (0)             class TestMaskedMatrix:
25: (4)                 def test_matrix_indexing(self):
26: (8)                     x1 = np.matrix([[1, 2, 3], [4, 3, 2]])
27: (8)                     x2 = masked_array(x1, mask=[[1, 0, 0], [0, 1, 0]])
28: (8)                     x3 = masked_array(x1, mask=[[0, 1, 0], [1, 0, 0]])
29: (8)                     x4 = masked_array(x1)
30: (8)                     str(x2) # raises?
31: (8)                     repr(x2) # raises?
32: (8)                     assert_(type(x2[1, 0]) is type(x1[1, 0]))
33: (8)                     assert_(x1[1, 0] == x2[1, 0])
34: (8)                     assert_(x2[1, 1] is masked)
35: (8)                     assert_equal(x1[0, 2], x2[0, 2])
36: (8)                     assert_equal(x1[0, 1:], x2[0, 1:])
37: (8)                     assert_equal(x1[:, 2], x2[:, 2])
38: (8)                     assert_equal(x1[:, :], x2[:, :])
39: (8)                     assert_equal(x1[1:], x3[1:])
40: (8)                     x1[0, 2] = 9
41: (8)                     x2[0, 2] = 9
42: (8)                     assert_equal(x1, x2)
43: (8)                     x1[0, 1:] = 99
44: (8)                     x2[0, 1:] = 99
45: (8)                     assert_equal(x1, x2)
46: (8)                     x2[0, 1] = masked
47: (8)                     assert_equal(x1, x2)
48: (8)                     x2[0, 1:] = masked
49: (8)                     assert_equal(x1, x2)
50: (8)                     x2[0, :] = x1[0, :]
51: (8)                     x2[0, 1] = masked
52: (8)                     assert_(allequal(getmask(x2), np.array([[0, 1, 0], [0, 1, 0]])))
53: (8)                     x3[1, :] = masked_array([1, 2, 3], [1, 1, 0])
54: (8)                     assert_(allequal(getmask(x3[1]), masked_array([1, 1, 0])))
55: (8)                     assert_(allequal(getmask(x3[1]), masked_array([1, 1, 0])))
56: (8)                     x4[1, :] = masked_array([1, 2, 3], [1, 1, 0])
57: (8)                     assert_(allequal(getmask(x4[1]), masked_array([1, 1, 0])))
58: (8)                     assert_(allequal(x4[1], masked_array([1, 2, 3])))
59: (8)                     x1 = np.matrix(np.arange(5) * 1.0)
60: (8)                     x2 = masked_values(x1, 3.0)
61: (8)                     assert_equal(x1, x2)
62: (8)                     assert_(allequal(masked_array([0, 0, 0, 1, 0], dtype=MaskType),
63: (25)                               x2.mask))
64: (8)                     assert_equal(3.0, x2.fill_value)
65: (4)             def test_pickling_subbaseclass(self):
66: (8)                 a = masked_array(np.matrix(list(range(10))), mask=[1, 0, 1, 0, 0] * 2)
67: (8)                 for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
68: (12)                     a_pickled = pickle.loads(pickle.dumps(a, protocol=proto))
69: (12)                     assert_equal(a_pickled._mask, a._mask)
70: (12)                     assert_equal(a_pickled, a)
71: (12)                     assert_(isinstance(a_pickled._data, np.matrix))
72: (4)             def test_count_mean_with_matrix(self):
73: (8)                 m = masked_array(np.matrix([[1, 2], [3, 4]]), mask=np.zeros((2, 2)))
74: (8)                 assert_equal(m.count(axis=0).shape, (1, 2))
75: (8)                 assert_equal(m.count(axis=1).shape, (2, 1))
76: (8)                 assert_equal(m.mean(axis=0), [[2., 3.]])
77: (8)                 assert_equal(m.mean(axis=1), [[1.5], [3.5]])
78: (4)             def test_flat(self):
79: (8)                 test = masked_array(np.matrix([[1, 2, 3]]), mask=[0, 0, 1])
80: (8)                 assert_equal(test.flat[1], 2)
81: (8)                 assert_equal(test.flat[2], masked)

```

```

82: (8)             assert_(np.all(test.flat[0:2] == test[0, 0:2]))
83: (8)             test = masked_array(np.matrix([[1, 2, 3]]), mask=[0, 0, 1])
84: (8)             test.flat = masked_array([3, 2, 1], mask=[1, 0, 0])
85: (8)             control = masked_array(np.matrix([[3, 2, 1]]), mask=[1, 0, 0])
86: (8)             assert_equal(test, control)
87: (8)             test = masked_array(np.matrix([[1, 2, 3]]), mask=[0, 0, 1])
88: (8)             testflat = test.flat
89: (8)             testflat[:] = testflat[[2, 1, 0]]
90: (8)             assert_equal(test, control)
91: (8)             testflat[0] = 9
92: (8)             a = masked_array(np.matrix(np.eye(2)), mask=0)
93: (8)             b = a.flat
94: (8)             b01 = b[:2]
95: (8)             assert_equal(b01.data, np.array([[1., 0.]]))
96: (8)             assert_equal(b01.mask, np.array([[False, False]]))
97: (4)             def test_allany_onmatrices(self):
98: (8)                 x = np.array([[0.13, 0.26, 0.90],
99: (22)                               [0.28, 0.33, 0.63],
100: (22)                              [0.31, 0.87, 0.70]])
101: (8)                 X = np.matrix(x)
102: (8)                 m = np.array([[True, False, False],
103: (22)                               [False, False, False],
104: (22)                               [True, True, False]], dtype=np.bool_)
105: (8)                 mX = masked_array(X, mask=m)
106: (8)                 mXbig = (mX > 0.5)
107: (8)                 mXsmall = (mX < 0.5)
108: (8)                 assert_(not mXbig.all())
109: (8)                 assert_(mXbig.any())
110: (8)                 assert_equal(mXbig.all(0), np.matrix([False, False, True]))
111: (8)                 assert_equal(mXbig.all(1), np.matrix([False, False, True]).T)
112: (8)                 assert_equal(mXbig.any(0), np.matrix([False, False, True]))
113: (8)                 assert_equal(mXbig.any(1), np.matrix([True, True, True]).T)
114: (8)                 assert_(not mXsmall.all())
115: (8)                 assert_(mXsmall.any())
116: (8)                 assert_equal(mXsmall.all(0), np.matrix([True, True, False]))
117: (8)                 assert_equal(mXsmall.all(1), np.matrix([False, False, False]).T)
118: (8)                 assert_equal(mXsmall.any(0), np.matrix([True, True, False]))
119: (8)                 assert_equal(mXsmall.any(1), np.matrix([True, True, False]).T)
120: (4)             def test_compressed(self):
121: (8)                 a = masked_array(np.matrix([1, 2, 3, 4]), mask=[0, 0, 0, 0])
122: (8)                 b = a.compressed()
123: (8)                 assert_equal(b, a)
124: (8)                 assert_(isinstance(b, np.matrix))
125: (8)                 a[0, 0] = masked
126: (8)                 b = a.compressed()
127: (8)                 assert_equal(b, [[2, 3, 4]])
128: (4)             def test_ravel(self):
129: (8)                 a = masked_array(np.matrix([1, 2, 3, 4, 5]), mask=[[0, 1, 0, 0, 0]])
130: (8)                 aravel = a.ravel()
131: (8)                 assert_equal(aravel.shape, (1, 5))
132: (8)                 assert_equal(aravel._mask.shape, a.shape)
133: (4)             def test_view(self):
134: (8)                 iterator = list(zip(np.arange(10), np.random.rand(10)))
135: (8)                 data = np.array(iterator)
136: (8)                 a = masked_array(iterator, dtype=[('a', float), ('b', float)])
137: (8)                 a.mask[0] = (1, 0)
138: (8)                 test = a.view((float, 2), np.matrix)
139: (8)                 assert_equal(test, data)
140: (8)                 assert_(isinstance(test, np.matrix))
141: (8)                 assert_(not isinstance(test, MaskedArray))
142: (0)             class TestSubclassing:
143: (4)                 def setup_method(self):
144: (8)                     x = np.arange(5, dtype='float')
145: (8)                     mx = MMMatrix(x, mask=[0, 1, 0, 0, 0])
146: (8)                     self.data = (x, mx)
147: (4)                 def test_maskedarray_subclassing(self):
148: (8)                     (x, mx) = self.data
149: (8)                     assert_(isinstance(mx._data, np.matrix))
150: (4)                 def test_masked_unary_operations(self):

```

```

151: (8)             (x, mx) = self.data
152: (8)             with np.errstate(divide='ignore'):
153: (12)             assert_(isinstance(log(mx), MMatrix))
154: (12)             assert_equal(log(x), np.log(x))
155: (4)              def test_masked_binary_operations(self):
156: (8)                (x, mx) = self.data
157: (8)                assert_(isinstance(add(mx, mx), MMatrix))
158: (8)                assert_(isinstance(add(mx, x), MMatrix))
159: (8)                assert_equal(add(mx, x), mx+mx)
160: (8)                assert_(isinstance(add(mx, mx)._data, np.matrix))
161: (8)                with assert_warns(DeprecationWarning):
162: (12)                  assert_(isinstance(add.outer(mx, mx), MMatrix))
163: (8)                  assert_(isinstance(hypot(mx, mx), MMatrix))
164: (8)                  assert_(isinstance(hypot(mx, x), MMatrix))
165: (4)              def test_masked_binary_operations2(self):
166: (8)                (x, mx) = self.data
167: (8)                xmx = masked_array(mx.data.__array__(), mask=mx.mask)
168: (8)                assert_(isinstance(divide(mx, mx), MMatrix))
169: (8)                assert_(isinstance(divide(mx, x), MMatrix))
170: (8)                assert_equal(divide(mx, mx), divide(xmx, xmx))
171: (0)              class TestConcatenator:
172: (4)                def test_matrix_builder(self):
173: (8)                  assert_raises(np.ma.MAError, lambda: mr_['1, 2; 3, 4'])
174: (4)                def test_matrix(self):
175: (8)                  actual = mr_['r', 1, 2, 3]
176: (8)                  expected = np.ma.array(np.r_['r', 1, 2, 3])
177: (8)                  assert_array_equal(actual, expected)
178: (8)                  assert_equal(type(actual), type(expected))
179: (8)                  assert_equal(type(actual.data), type(expected.data))

```

---

## File 282 - test\_matrix\_linalg.py:

```

1: (0)      """ Test functions for linalg module using the matrix class."""
2: (0)      import numpy as np
3: (0)      from numpy.linalg.tests.test_linalg import (
4: (4)        LinalgCase, apply_tag, TestQR as _TestQR, LinalgTestCase,
5: (4)        _TestNorm2D, _TestNormDoubleBase, _TestNormSingleBase, _TestNormInt64Base,
6: (4)        SolveCases, InvCases, EigvalsCases, EigCases, SVDCases, CondCases,
7: (4)        PinvCases, DetCases, LstsqCases)
8: (0)      CASES = []
9: (0)      CASES += apply_tag('square', [
10: (4)        LinalgCase("0x0_matrix",
11: (15)          np.empty((0, 0), dtype=np.double).view(np.matrix),
12: (15)          np.empty((0, 1), dtype=np.double).view(np.matrix),
13: (15)          tags={'size-0'}),
14: (4)        LinalgCase("matrix_b_only",
15: (15)          np.array([[1., 2.], [3., 4.]]),
16: (15)          np.matrix([2., 1.]).T),
17: (4)        LinalgCase("matrix_a_and_b",
18: (15)          np.matrix([[1., 2.], [3., 4.]]),
19: (15)          np.matrix([2., 1.]).T),
20: (0)      ])
21: (0)      CASES += apply_tag('hermitian', [
22: (4)        LinalgCase("hmatrix_a_and_b",
23: (15)          np.matrix([[1., 2.], [2., 1.]]),
24: (15)          None),
25: (0)      ])
26: (0)      class MatrixTestCase(LinalgTestCase):
27: (4)        TEST_CASES = CASES
28: (0)      class TestSolveMatrix(SolveCases, MatrixTestCase):
29: (4)        pass
30: (0)      class TestInvMatrix(InvCases, MatrixTestCase):
31: (4)        pass
32: (0)      class TestEigvalsMatrix(EigvalsCases, MatrixTestCase):
33: (4)        pass
34: (0)      class TestEigMatrix(EigCases, MatrixTestCase):
35: (4)        pass

```

12/16/24, 6:27 PM

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```
36: (0)         class TestSVDMatrix(SVDCases, MatrixTestCase):
37: (4)             pass
38: (0)         class TestCondMatrix(CondCases, MatrixTestCase):
39: (4)             pass
40: (0)         class TestPinvMatrix(PinvCases, MatrixTestCase):
41: (4)             pass
42: (0)         class TestDetMatrix(DetCases, MatrixTestCase):
43: (4)             pass
44: (0)         class TestLstsqMatrix(LstsqCases, MatrixTestCase):
45: (4)             pass
46: (0)         class _TestNorm2DMatrix(_TestNorm2D):
47: (4)             array = np.matrix
48: (0)         class TestNormDoubleMatrix(_TestNorm2DMatrix, _TestNormDoubleBase):
49: (4)             pass
50: (0)         class TestNormSingleMatrix(_TestNorm2DMatrix, _TestNormSingleBase):
51: (4)             pass
52: (0)         class TestNormInt64Matrix(_TestNorm2DMatrix, _TestNormInt64Base):
53: (4)             pass
54: (0)         class TestQRMatrix(_TestQR):
55: (4)             array = np.matrix
```

---

File 283 - test\_multiarray.py:

```
1: (0)         import numpy as np
2: (0)         from numpy.testing import assert_, assert_equal, assert_array_equal
3: (0)         class TestView:
4: (4)             def test_type(self):
5: (8)                 x = np.array([1, 2, 3])
6: (8)                 assert_(isinstance(x.view(np.matrix), np.matrix))
7: (4)             def test_keywords(self):
8: (8)                 x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
9: (8)                 y = x.view(dtype='<i2', type=np.matrix)
10: (8)                 assert_array_equal(y, [[513]])
11: (8)                 assert_(isinstance(y, np.matrix))
12: (8)                 assert_equal(y.dtype, np.dtype('<i2'))
```

---

File 284 - \_\_init\_\_.py:

```
1: (0)
```

---

File 285 - test\_numeric.py:

```
1: (0)         import numpy as np
2: (0)         from numpy.testing import assert_equal
3: (0)         class TestDot:
4: (4)             def test_matscalar(self):
5: (8)                 b1 = np.matrix(np.ones((3, 3), dtype=complex))
6: (8)                 assert_equal(b1*1.0, b1)
7: (0)             def test_diagonal():
8: (4)                 b1 = np.matrix([[1,2],[3,4]])
9: (4)                 diag_b1 = np.matrix([[1, 4]])
10: (4)                 array_b1 = np.array([1, 4])
11: (4)                 assert_equal(b1.diagonal(), diag_b1)
12: (4)                 assert_equal(np.diagonal(b1), array_b1)
13: (4)                 assert_equal(np.diag(b1), array_b1)
```

---

File 286 - test\_regression.py:

```
1: (0)         import numpy as np
2: (0)         from numpy.testing import assert_, assert_equal, assert_raises
3: (0)         class TestRegression:
```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

4: (4)         def test_kron_matrix(self):
5: (8)             x = np.matrix('[[1 0; 1 0]]')
6: (8)             assert_equal(type(np.kron(x, x)), type(x))
7: (4)         def test_matrix_properties(self):
8: (8)             a = np.matrix([1.0], dtype=float)
9: (8)             assert_(type(a.real) is np.matrix)
10: (8)            assert_(type(a.imag) is np.matrix)
11: (8)            c, d = np.matrix([0.0]).nonzero()
12: (8)            assert_(type(c) is np.ndarray)
13: (8)            assert_(type(d) is np.ndarray)
14: (4)         def test_matrix_multiply_by_1d_vector(self):
15: (8)             def mul():
16: (12)                 np.mat(np.eye(2))*np.ones(2)
17: (8)                 assert_raises(ValueError, mul)
18: (4)         def test_matrix_std_argmax(self):
19: (8)             x = np.asmatrix(np.random.uniform(0, 1, (3, 3)))
20: (8)             assert_equal(x.std().shape, ())
21: (8)             assert_equal(x.argmax().shape, ())

-----
```

## File 287 - chebyshev.py:

```
1: (0)         """
2: (0) =====
3: (0) Chebyshev Series (:mod:`numpy.polynomial.chebyshev`)
4: (0) =====
5: (0) This module provides a number of objects (mostly functions) useful for
6: (0) dealing with Chebyshev series, including a `Chebyshev` class that
7: (0) encapsulates the usual arithmetic operations. (General information
8: (0) on how this module represents and works with such polynomials is in the
9: (0) docstring for its "parent" sub-package, `numpy.polynomial`).
10: (0) Classes
11: (0) -----
12: (0) .. autosummary::
13: (3)     :toctree: generated/
14: (3)     Chebyshev
15: (0) Constants
16: (0) -----
17: (0) .. autosummary::
18: (3)     :toctree: generated/
19: (3)     chebdomain
20: (3)     chebzero
21: (3)     chebone
22: (3)     chebx
23: (0) Arithmetic
24: (0) -----
25: (0) .. autosummary::
26: (3)     :toctree: generated/
27: (3)     chebadd
28: (3)     chebsub
29: (3)     chebmulp
30: (3)     chebmul
31: (3)     chebdiv
32: (3)     chebpow
33: (3)     chebval
34: (3)     chebval2d
35: (3)     chebval3d
36: (3)     chebgrid2d
37: (3)     chebgrid3d
38: (0) Calculus
39: (0) -----
40: (0) .. autosummary::
41: (3)     :toctree: generated/
42: (3)     chebder
43: (3)     chebint
44: (0) Misc Functions
45: (0) -----
46: (0) .. autosummary::
```

```

47: (3) :toctree: generated/
48: (3) chebfromroots
49: (3) chebroots
50: (3) chebvander
51: (3) chebvander2d
52: (3) chebvander3d
53: (3) chebgauss
54: (3) chebweight
55: (3) chebcompanion
56: (3) chebfat
57: (3) chebpts1
58: (3) chebpts2
59: (3) chebtrim
60: (3) chebline
61: (3) cheb2poly
62: (3) poly2cheb
63: (3) chebinterpolate
64: (0) See also -----
65: (0)
66: (0) `numpy.polynomial`_
67: (0) Notes -----
68: (0)
69: (0) The implementations of multiplication, division, integration, and
70: (0) differentiation use the algebraic identities [1]_:
71: (0) .. math::
72: (4)    $T_n(x) = \frac{z^n + z^{-n}}{2}$ 
73: (4)    $z \frac{dx}{dz} = \frac{z - z^{-1}}{2}$ .
74: (0) where
75: (0)   .. math:: x = \frac{z + z^{-1}}{2}.
76: (0) These identities allow a Chebyshev series to be expressed as a finite,
77: (0) symmetric Laurent series. In this module, this sort of Laurent series
78: (0) is referred to as a "z-series."
79: (0) References -----
80: (0)
81: (0)   .. [1] A. T. Benjamin, et al., "Combinatorial Trigonometry with Chebyshev
82: (2)   Polynomials," *Journal of Statistical Planning and Inference 14*, 2008
83: (2)

(https://web.archive.org/web/20080221202153/https://www.math.hmc.edu/~benjamin/papers/CombTrig.pdf
, pg. 4)
84: (0) """
85: (0) import numpy as np
86: (0) import numpy.linalg as la
87: (0) from numpy.core.multiarray import normalize_axis_index
88: (0) from . import polyutils as pu
89: (0) from ._polybase import ABCPolyBase
90: (0) __all__ = [
91: (4)   'chebzero', 'chebone', 'chebx', 'chebdomain', 'chebline', 'chebadd',
92: (4)   'chebsub', 'chebmulp', 'chebmul', 'chebdiv', 'chebpow', 'chebval',
93: (4)   'chebder', 'chebint', 'cheb2poly', 'poly2cheb', 'chebfromroots',
94: (4)   'chebvander', 'chebfat', 'chebtrim', 'chebroots', 'chebpts1',
95: (4)   'chebpts2', 'Chebyshev', 'chebval2d', 'chebval3d', 'chebgrid2d',
96: (4)   'chebgrid3d', 'chebvander2d', 'chebvander3d', 'chebcompanion',
97: (4)   'chebgauss', 'chebweight', 'chebinterpolate']
98: (0) chebtrim = pu.trimcoef
99: (0) def _cseries_to_zseries(c):
100: (4)     """Convert Chebyshev series to z-series.
101: (4)     Convert a Chebyshev series to the equivalent z-series. The result is
102: (4)     never an empty array. The dtype of the return is the same as that of
103: (4)     the input. No checks are run on the arguments as this routine is for
104: (4)     internal use.
105: (4)     Parameters
106: (4)     -----
107: (4)     c : 1-D ndarray
108: (8)         Chebyshev coefficients, ordered from low to high
109: (4)     Returns
109: (4)     -----
111: (4)     zs : 1-D ndarray
112: (8)         Odd length symmetric z-series, ordered from low to high.
113: (4) """

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

114: (4) n = c.size
115: (4) zs = np.zeros(2*n-1, dtype=c.dtype)
116: (4) zs[n-1:] = c/2
117: (4) return zs + zs[::-1]
118: (0) def _zseries_to_cseries(zs):
119: (4) """Convert z-series to a Chebyshev series.
120: (4) Convert a z series to the equivalent Chebyshev series. The result is
121: (4) never an empty array. The dtype of the return is the same as that of
122: (4) the input. No checks are run on the arguments as this routine is for
123: (4) internal use.
124: (4) Parameters
125: (4) -----
126: (4) zs : 1-D ndarray
127: (8)      Odd length symmetric z-series, ordered from low to high.
128: (4) Returns
129: (4) -----
130: (4) c : 1-D ndarray
131: (8)      Chebyshev coefficients, ordered from low to high.
132: (4) """
133: (4) n = (zs.size + 1)//2
134: (4) c = zs[n-1:].copy()
135: (4) c[1:n] *= 2
136: (4) return c
137: (0) def _zseries_mul(z1, z2):
138: (4) """Multiply two z-series.
139: (4) Multiply two z-series to produce a z-series.
140: (4) Parameters
141: (4) -----
142: (4) z1, z2 : 1-D ndarray
143: (8)      The arrays must be 1-D but this is not checked.
144: (4) Returns
145: (4) -----
146: (4) product : 1-D ndarray
147: (8)      The product z-series.
148: (4) Notes
149: (4) -----
150: (4) This is simply convolution. If symmetric/anti-symmetric z-series are
151: (4) denoted by S/A then the following rules apply:
152: (4) S*S, A*A -> S
153: (4) S*A, A*S -> A
154: (4) """
155: (4) return np.convolve(z1, z2)
156: (0) def _zseries_div(z1, z2):
157: (4) """Divide the first z-series by the second.
158: (4) Divide `z1` by `z2` and return the quotient and remainder as z-series.
159: (4) Warning: this implementation only applies when both z1 and z2 have the
160: (4) same symmetry, which is sufficient for present purposes.
161: (4) Parameters
162: (4) -----
163: (4) z1, z2 : 1-D ndarray
164: (8)      The arrays must be 1-D and have the same symmetry, but this is not
165: (8)      checked.
166: (4) Returns
167: (4) -----
168: (4) (quotient, remainder) : 1-D ndarrays
169: (8)      Quotient and remainder as z-series.
170: (4) Notes
171: (4) -----
172: (4) This is not the same as polynomial division on account of the desired form
173: (4) of the remainder. If symmetric/anti-symmetric z-series are denoted by S/A
174: (4) then the following rules apply:
175: (4) S/S -> S,S
176: (4) A/A -> S,A
177: (4) The restriction to types of the same symmetry could be fixed but seems
like
178: (4) unneeded generality. There is no natural form for the remainder in the
case
179: (4) where there is no symmetry.
180: (4) """

```

```

181: (4)             z1 = z1.copy()
182: (4)             z2 = z2.copy()
183: (4)             lc1 = len(z1)
184: (4)             lc2 = len(z2)
185: (4)             if lc2 == 1:
186: (8)                 z1 /= z2
187: (8)                 return z1, z1[:1]*0
188: (4)             elif lc1 < lc2:
189: (8)                 return z1[:1]*0, z1
190: (4)             else:
191: (8)                 dlen = lc1 - lc2
192: (8)                 scl = z2[0]
193: (8)                 z2 /= scl
194: (8)                 quo = np.empty(dlen + 1, dtype=z1.dtype)
195: (8)                 i = 0
196: (8)                 j = dlen
197: (8)                 while i < j:
198: (12)                     r = z1[i]
199: (12)                     quo[i] = z1[i]
200: (12)                     quo[dlen - i] = r
201: (12)                     tmp = r*z2
202: (12)                     z1[i:i+lc2] -= tmp
203: (12)                     z1[j:j+lc2] -= tmp
204: (12)                     i += 1
205: (12)                     j -= 1
206: (8)                     r = z1[i]
207: (8)                     quo[i] = r
208: (8)                     tmp = r*z2
209: (8)                     z1[i:i+lc2] -= tmp
210: (8)                     quo /= scl
211: (8)                     rem = z1[i+1:i-1+lc2].copy()
212: (8)                     return quo, rem
213: (0)             def _zseries_der(zs):
214: (4)                 """Differentiate a z-series.
215: (4)                 The derivative is with respect to x, not z. This is achieved using the
216: (4)                 chain rule and the value of dx/dz given in the module notes.
217: (4)                 Parameters
218: (4)                 -----
219: (4)                 zs : z-series
220: (8)                     The z-series to differentiate.
221: (4)                 Returns
222: (4)                 -----
223: (4)                 derivative : z-series
224: (8)                     The derivative
225: (4)                 Notes
226: (4)                 -----
227: (4)                 The zseries for x (ns) has been multiplied by two in order to avoid
228: (4)                 using floats that are incompatible with Decimal and likely other
229: (4)                 specialized scalar types. This scaling has been compensated by
230: (4)                 multiplying the value of zs by two also so that the two cancels in the
231: (4)                 division.
232: (4)                 """
233: (4)                 n = len(zs)//2
234: (4)                 ns = np.array([-1, 0, 1], dtype=zs.dtype)
235: (4)                 zs *= np.arange(-n, n+1)*2
236: (4)                 d, r = _zseries_div(zs, ns)
237: (4)                 return d
238: (0)             def _zseries_int(zs):
239: (4)                 """Integrate a z-series.
240: (4)                 The integral is with respect to x, not z. This is achieved by a change
241: (4)                 of variable using dx/dz given in the module notes.
242: (4)                 Parameters
243: (4)                 -----
244: (4)                 zs : z-series
245: (8)                     The z-series to integrate
246: (4)                 Returns
247: (4)                 -----
248: (4)                 integral : z-series
249: (8)                     The indefinite integral

```

```

250: (4)          Notes
251: (4)
252: (4)          -----
253: (4)          The zseries for x (ns) has been multiplied by two in order to avoid
254: (4)          using floats that are incompatible with Decimal and likely other
255: (4)          specialized scalar types. This scaling has been compensated by
256: (4)          dividing the resulting zs by two.
257: (4)          """
258: (4)          n = 1 + len(zs)//2
259: (4)          ns = np.array([-1, 0, 1], dtype=zs.dtype)
260: (4)          zs = _zseries_mul(zs, ns)
261: (4)          div = np.arange(-n, n+1)*2
262: (4)          zs[:n] /= div[:n]
263: (4)          zs[n+1:] /= div[n+1:]
264: (4)          zs[n] = 0
265: (4)          return zs
266: (4) def poly2cheb(pol):
267: (4)          """
268: (4)          Convert a polynomial to a Chebyshev series.
269: (4)          Convert an array representing the coefficients of a polynomial (relative
270: (4)          to the "standard" basis) ordered from lowest degree to highest, to an
271: (4)          array of the coefficients of the equivalent Chebyshev series, ordered
272: (4)          from lowest to highest degree.
273: (4)          Parameters
274: (4)          -----
275: (8)          pol : array_like
276: (4)          1-D array containing the polynomial coefficients
277: (4)          Returns
278: (4)          -----
279: (8)          c : ndarray
280: (8)          1-D array containing the coefficients of the equivalent Chebyshev
281: (4)          series.
282: (4)          See Also
283: (4)          -----
284: (4)          cheb2poly
285: (4)          Notes
286: (4)          -----
287: (4)          The easy way to do conversions between polynomial basis sets
288: (4)          is to use the convert method of a class instance.
289: (4)          Examples
290: (4)          -----
291: (4)          >>> from numpy import polynomial as P
292: (4)          >>> p = P.Polynomial(range(4))
293: (4)          >>> p
294: (4)          Polynomial([0., 1., 2., 3.], domain=[-1, 1], window=[-1, 1])
295: (4)          >>> c = p.convert(kind=P.Chebyshev)
296: (4)          >>> c
297: (4)          Chebyshev([1. , 3.25, 1. , 0.75], domain=[-1., 1.], window=[-1., 1.])
298: (4)          >>> P.chebyshev.poly2cheb(range(4))
299: (4)          array([1. , 3.25, 1. , 0.75])
299: (4)          """
300: (4)          [pol] = pu.as_series([pol])
301: (4)          deg = len(pol) - 1
302: (4)          res = 0
303: (4)          for i in range(deg, -1, -1):
304: (8)              res = chebadd(chebmulp(res), pol[i])
305: (4)          return res
306: (0) def cheb2poly(c):
307: (4)          """
308: (4)          Convert a Chebyshev series to a polynomial.
309: (4)          Convert an array representing the coefficients of a Chebyshev series,
310: (4)          ordered from lowest degree to highest, to an array of the coefficients
311: (4)          of the equivalent polynomial (relative to the "standard" basis) ordered
312: (4)          from lowest to highest degree.
313: (4)          Parameters
314: (4)          -----
315: (4)          c : array_like
316: (8)          1-D array containing the Chebyshev series coefficients, ordered
317: (8)          from lowest order term to highest.
318: (4)          Returns

```

```

319: (4)      -----
320: (4)      pol : ndarray
321: (8)      1-D array containing the coefficients of the equivalent polynomial
322: (8)      (relative to the "standard" basis) ordered from lowest order term
323: (8)      to highest.
324: (4)      See Also
325: (4)      -----
326: (4)      poly2cheb
327: (4)      Notes
328: (4)      -----
329: (4)      The easy way to do conversions between polynomial basis sets
330: (4)      is to use the convert method of a class instance.
331: (4)      Examples
332: (4)      -----
333: (4)      >>> from numpy import polynomial as P
334: (4)      >>> c = P.Chebyshev(range(4))
335: (4)      >>> c
336: (4)      Chebyshev([0., 1., 2., 3.], domain=[-1, 1], window=[-1, 1])
337: (4)      >>> p = c.convert(kind=P.Polynomial)
338: (4)      >>> p
339: (4)      Polynomial([-2., -8., 4., 12.], domain=[-1., 1.], window=[-1., 1.])
340: (4)      >>> P.chebyshev.cheb2poly(range(4))
341: (4)      array([-2., -8., 4., 12.])
342: (4)      """
343: (4)      from .polynomial import polyadd, polysub, polymulx
344: (4)      [c] = pu.as_series([c])
345: (4)      n = len(c)
346: (4)      if n < 3:
347: (8)          return c
348: (4)      else:
349: (8)          c0 = c[-2]
350: (8)          c1 = c[-1]
351: (8)          for i in range(n - 1, 1, -1):
352: (12)              tmp = c0
353: (12)              c0 = polysub(c[i - 2], c1)
354: (12)              c1 = polyadd(tmp, polymulx(c1)*2)
355: (8)          return polyadd(c0, polymulx(c1))
356: (0)      chebdomain = np.array([-1, 1])
357: (0)      chebzero = np.array([0])
358: (0)      chebone = np.array([1])
359: (0)      chebx = np.array([0, 1])
360: (0)      def chebline(off, scl):
361: (4)          """
362: (4)          Chebyshev series whose graph is a straight line.
363: (4)          Parameters
364: (4)          -----
365: (4)          off, scl : scalars
366: (8)          The specified line is given by ``off + scl*x``.
367: (4)          Returns
368: (4)          -----
369: (4)          y : ndarray
370: (8)          This module's representation of the Chebyshev series for
371: (8)          ``off + scl*x``.
372: (4)          See Also
373: (4)          -----
374: (4)          numpy.polynomial.polynomial.polyline
375: (4)          numpy.polynomial.legendre.legline
376: (4)          numpy.polynomial.laguerre.lagline
377: (4)          numpy.polynomial.hermite.hermeline
378: (4)          numpy.polynomial.hermite_e.hermeline
379: (4)          Examples
380: (4)          -----
381: (4)          >>> import numpy.polynomial.chebyshev as C
382: (4)          >>> C.chebline(3,2)
383: (4)          array([3, 2])
384: (4)          >>> C.chebval(-3, C.chebline(3,2)) # should be -3
385: (4)          -3.0
386: (4)          """
387: (4)          if scl != 0:

```

```

388: (8)             return np.array([off, scl])
389: (4)         else:
390: (8)             return np.array([off])
391: (0)     def chebfromroots(roots):
392: (4)         """
393: (4)             Generate a Chebyshev series with given roots.
394: (4)             The function returns the coefficients of the polynomial
395: (4)             .. math:: p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),
396: (4)             in Chebyshev form, where the `r_n` are the roots specified in `roots`.
397: (4)             If a zero has multiplicity n, then it must appear in `roots` n times.
398: (4)             For instance, if 2 is a root of multiplicity three and 3 is a root of
399: (4)             multiplicity 2, then `roots` looks something like [2, 2, 2, 3, 3]. The
400: (4)             roots can appear in any order.
401: (4)             If the returned coefficients are `c`, then
402: (4)             .. math:: p(x) = c_0 + c_1 * T_1(x) + ... + c_n * T_n(x)
403: (4)             The coefficient of the last term is not generally 1 for monic
404: (4)             polynomials in Chebyshev form.
405: (4)             Parameters
406: (4)             -----
407: (4)             roots : array_like
408: (8)                 Sequence containing the roots.
409: (4)             Returns
410: (4)             -----
411: (4)             out : ndarray
412: (8)                 1-D array of coefficients. If all roots are real then `out` is a
413: (8)                 real array, if some of the roots are complex, then `out` is complex
414: (8)                 even if all the coefficients in the result are real (see Examples
415: (8)                 below).
416: (4)             See Also
417: (4)             -----
418: (4)             numpy.polynomial.polynomial.polyfromroots
419: (4)             numpy.polynomial.legendre.legfromroots
420: (4)             numpy.polynomial.laguerre.lagfromroots
421: (4)             numpy.polynomial.hermite.hermfromroots
422: (4)             numpy.polynomial.hermite_e.hermefromroots
423: (4)             Examples
424: (4)             -----
425: (4)             >>> import numpy.polynomial.chebyshev as C
426: (4)             >>> C.chebfromroots((-1,0,1)) # x^3 - x relative to the standard basis
427: (4)             array([ 0. , -0.25,  0. ,  0.25])
428: (4)             >>> j = complex(0,1)
429: (4)             >>> C.chebfromroots((-j,j)) # x^2 + 1 relative to the standard basis
430: (4)             array([1.5+0.j,  0. +0.j,  0.5+0.j])
431: (4)             """
432: (4)             return pu._fromroots(chebline, chebmul, roots)
433: (0)     def chebadd(c1, c2):
434: (4)         """
435: (4)             Add one Chebyshev series to another.
436: (4)             Returns the sum of two Chebyshev series `c1` + `c2`. The arguments
437: (4)             are sequences of coefficients ordered from lowest order term to
438: (4)             highest, i.e., [1,2,3] represents the series ``T_0 + 2*T_1 + 3*T_2``.
439: (4)             Parameters
440: (4)             -----
441: (4)             c1, c2 : array_like
442: (8)                 1-D arrays of Chebyshev series coefficients ordered from low to
443: (8)                 high.
444: (4)             Returns
445: (4)             -----
446: (4)             out : ndarray
447: (8)                 Array representing the Chebyshev series of their sum.
448: (4)             See Also
449: (4)             -----
450: (4)             chebsub, chebmulx, chebmul, chebdiv, chebpow
451: (4)             Notes
452: (4)             -----
453: (4)             Unlike multiplication, division, etc., the sum of two Chebyshev series
454: (4)             is a Chebyshev series (without having to "reproject" the result onto
455: (4)             the basis set) so addition, just like that of "standard" polynomials,
456: (4)             is simply "component-wise."

```

```

457: (4)          Examples
458: (4)
459: (4)          -----
460: (4)          >>> from numpy.polynomial import chebyshev as C
461: (4)          >>> c1 = (1,2,3)
462: (4)          >>> c2 = (3,2,1)
463: (4)          >>> C.chebadd(c1,c2)
464: (4)          array([4., 4., 4.])
465: (4)          """
466: (0)          return pu._add(c1, c2)
467: (4)
468: (4)          def chebsub(c1, c2):
469: (4)          """
470: (4)          Subtract one Chebyshev series from another.
471: (4)          Returns the difference of two Chebyshev series `c1` - `c2`. The
472: (4)          sequences of coefficients are from lowest order term to highest, i.e.,
473: (4)          [1,2,3] represents the series `` $T_0 + 2*T_1 + 3*T_2```.
474: (4)          Parameters
475: (4)          -----
476: (8)          c1, c2 : array_like
477: (8)          1-D arrays of Chebyshev series coefficients ordered from low to
478: (4)          high.
479: (4)          Returns
480: (8)          -----
481: (4)          out : ndarray
482: (8)          Of Chebyshev series coefficients representing their difference.
483: (4)          See Also
484: (4)          -----
485: (4)          chebadd, chebmulp, chebmul, chebdiv, chebpow
486: (4)          Notes
487: (4)          -----
488: (4)          Unlike multiplication, division, etc., the difference of two Chebyshev
489: (4)          series is a Chebyshev series (without having to "reproject" the result
490: (4)          onto the basis set) so subtraction, just like that of "standard"
491: (4)          polynomials, is simply "component-wise."
492: (4)          Examples
493: (4)
494: (4)          -----
495: (4)          >>> from numpy.polynomial import chebyshev as C
496: (4)          >>> c1 = (1,2,3)
497: (4)          >>> c2 = (3,2,1)
498: (4)          >>> C.chebsub(c1,c2)
499: (4)          array([-2., 0., 2.])
500: (4)          >>> C.chebsub(c2,c1) # -C.chebsub(c1,c2)
501: (4)          array([ 2., 0., -2.])
502: (0)          """
503: (4)          return pu._sub(c1, c2)
504: (4)
505: (4)          def chebmulp(x):
506: (4)          """
507: (4)          Multiply a Chebyshev series by x.
508: (4)          Multiply the polynomial `c` by x, where x is the independent
509: (4)          variable.
510: (4)          Parameters
511: (4)          -----
512: (4)          c : array_like
513: (8)          1-D array of Chebyshev series coefficients ordered from low to
514: (4)          high.
515: (4)          Returns
516: (4)          -----
517: (4)          out : ndarray
518: (4)          Array representing the result of the multiplication.
519: (4)          Notes
520: (4)          -----
521: (4)          .. versionadded:: 1.5.0
522: (4)
523: (4)          Examples
524: (4)
525: (8)          -----
526: (4)          >>> from numpy.polynomial import chebyshev as C
527: (4)          >>> C.chebmulp([1,2,3])
528: (4)          array([1., 2.5, 1., 1.5])
529: (4)          """
530: (4)          [c] = pu.as_series([c])
531: (4)          if len(c) == 1 and c[0] == 0:
532: (4)              return c$ 
```

```

526: (4)             prd = np.empty(len(c) + 1, dtype=c.dtype)
527: (4)             prd[0] = c[0]*0
528: (4)             prd[1] = c[0]
529: (4)             if len(c) > 1:
530: (8)                 tmp = c[1:]/2
531: (8)                 prd[2:] = tmp
532: (8)                 prd[0:-2] += tmp
533: (4)             return prd
534: (0)         def chebmul(c1, c2):
535: (4)             """
536: (4)             Multiply one Chebyshev series by another.
537: (4)             Returns the product of two Chebyshev series `c1` * `c2`. The arguments
538: (4)             are sequences of coefficients, from lowest order "term" to highest,
539: (4)             e.g., [1,2,3] represents the series `` $T_0 + 2*T_1 + 3*T_2```.
540: (4)             Parameters
541: (4)             -----
542: (4)             c1, c2 : array_like
543: (8)                 1-D arrays of Chebyshev series coefficients ordered from low to
544: (8)                 high.
545: (4)             Returns
546: (4)             -----
547: (4)             out : ndarray
548: (8)                 Of Chebyshev series coefficients representing their product.
549: (4)             See Also
550: (4)             -----
551: (4)             chebadd, chebsub, chebmulp, chebdiv, chebpow
552: (4)             Notes
553: (4)             -----
554: (4)             In general, the (polynomial) product of two C-series results in terms
555: (4)             that are not in the Chebyshev polynomial basis set. Thus, to express
556: (4)             the product as a C-series, it is typically necessary to "reproject"
557: (4)             the product onto said basis set, which typically produces
558: (4)             "unintuitive live" (but correct) results; see Examples section below.
559: (4)             Examples
560: (4)             -----
561: (4)             >>> from numpy.polynomial import chebyshev as C
562: (4)             >>> c1 = (1,2,3)
563: (4)             >>> c2 = (3,2,1)
564: (4)             >>> C.chebmul(c1,c2) # multiplication requires "reprojection"
565: (4)             array([ 6.5, 12. , 12. , 4. , 1.5])
566: (4)             """
567: (4)             [c1, c2] = pu.as_series([c1, c2])
568: (4)             z1 = _cseries_to_zseries(c1)
569: (4)             z2 = _cseries_to_zseries(c2)
570: (4)             prd = _zseries_mul(z1, z2)
571: (4)             ret = _zseries_to_cseries(prd)
572: (4)             return pu.trimseq(ret)
573: (0)         def chebdiv(c1, c2):
574: (4)             """
575: (4)             Divide one Chebyshev series by another.
576: (4)             Returns the quotient-with-remainder of two Chebyshev series
577: (4)             `c1` / `c2`. The arguments are sequences of coefficients from lowest
578: (4)             order "term" to highest, e.g., [1,2,3] represents the series
579: (4)             `` $T_0 + 2*T_1 + 3*T_2```.
580: (4)             Parameters
581: (4)             -----
582: (4)             c1, c2 : array_like
583: (8)                 1-D arrays of Chebyshev series coefficients ordered from low to
584: (8)                 high.
585: (4)             Returns
586: (4)             -----
587: (4)             [quo, rem] : ndarrays
588: (8)                 Of Chebyshev series coefficients representing the quotient and
589: (8)                 remainder.
590: (4)             See Also
591: (4)             -----
592: (4)             chebadd, chebsub, chebmulp, chebmul, chebpow
593: (4)             Notes
594: (4)             -----$$ 
```

```

595: (4) In general, the (polynomial) division of one C-series by another
596: (4) results in quotient and remainder terms that are not in the Chebyshev
597: (4) polynomial basis set. Thus, to express these results as C-series, it
598: (4) is typically necessary to "reproject" the results onto said basis
599: (4) set, which typically produces "unintuitive" (but correct) results;
600: (4) see Examples section below.
601: (4) Examples
602: (4) -----
603: (4) >>> from numpy.polynomial import chebyshev as C
604: (4)
605: (4) >>> c1 = (1,2,3)
606: (4) >>> c2 = (3,2,1)
607: (4) >>> C.chebdiv(c1,c2) # quotient "intuitive," remainder not
608: (4) (array([3.]), array([-8., -4.]))
609: (4) >>> c2 = (0,1,2,3)
610: (4) >>> C.chebdiv(c2,c1) # neither "intuitive"
611: (4) (array([0., 2.]), array([-2., -4.]))
612: (4) """
613: (4) [c1, c2] = pu.as_series([c1, c2])
614: (8) if c2[-1] == 0:
615: (4)     raise ZeroDivisionError()
616: (4) lc1 = len(c1)
617: (4) lc2 = len(c2)
618: (8) if lc1 < lc2:
619: (4)     return c1[:1]*0, c1
620: (8) elif lc2 == 1:
621: (4)     return c1/c2[-1], c1[:1]*0
622: (4) else:
623: (8)     z1 = _cseries_to_zseries(c1)
624: (8)     z2 = _cseries_to_zseries(c2)
625: (8)     quo, rem = _zseries_div(z1, z2)
626: (8)     quo = pu.trimseq(_zseries_to_cseries(quo))
627: (8)     rem = pu.trimseq(_zseries_to_cseries(rem))
628: (0)     return quo, rem
629: (4) def chebpow(c, pow, maxpower=16):
630: (4)     """Raise a Chebyshev series to a power.
631: (4)     Returns the Chebyshev series `c` raised to the power `pow`. The
632: (4)     argument `c` is a sequence of coefficients ordered from low to high.
633: (4)     i.e., [1,2,3] is the series ``T_0 + 2*T_1 + 3*T_2.``"
634: (4)     Parameters
635: (4)     -----
636: (8)     c : array_like
637: (8)         1-D array of Chebyshev series coefficients ordered from low to
638: (4)         high.
639: (8)     pow : integer
639: (8)         Power to which the series will be raised
640: (4)     maxpower : integer, optional
641: (8)         Maximum power allowed. This is mainly to limit growth of the series
642: (8)         to unmanageable size. Default is 16
643: (4)     Returns
644: (4)     -----
645: (4)     coef : ndarray
646: (8)         Chebyshev series of power.
647: (4)     See Also
648: (4)     -----
649: (4)     chebadd, chebsub, chebmulp, chebmul, chebdiv
650: (4)     Examples
651: (4)     -----
652: (4)     >>> from numpy.polynomial import chebyshev as C
653: (4)
654: (4)     >>> C.chebpow([1, 2, 3, 4], 2)
655: (4)     array([15.5, 22. , 16. , ..., 12.5, 12. , 8. ])
656: (4) """
656: (4)     [c] = pu.as_series([c])
657: (4)     power = int(pow)
658: (4)     if power != pow or power < 0:
659: (8)         raise ValueError("Power must be a non-negative integer.")
660: (4)     elif maxpower is not None and power > maxpower:
661: (8)         raise ValueError("Power is too large")
662: (4)     elif power == 0:
663: (8)         return np.array([1], dtype=c.dtype)

```

```

664: (4)             elif power == 1:
665: (8)                 return c
666: (4)             else:
667: (8)                 zs = _cseries_to_zseries(c)
668: (8)                 prd = zs
669: (8)                 for i in range(2, power + 1):
670: (12)                     prd = np.convolve(prd, zs)
671: (8)                 return _zseries_to_cseries(prd)
672: (0)             def chebder(c, m=1, scl=1, axis=0):
673: (4)                 """
674: (4)                 Differentiate a Chebyshev series.
675: (4)                 Returns the Chebyshev series coefficients `c` differentiated `m` times
676: (4)                 along `axis`. At each iteration the result is multiplied by `scl` (the
677: (4)                 scaling factor is for use in a linear change of variable). The argument
678: (4)                 `c` is an array of coefficients from low to high degree along each
679: (4)                 axis, e.g., [1,2,3] represents the series `` $1*T_0 + 2*T_1 + 3*T_2$ ``.
680: (4)                 while [[1,2],[1,2]] represents `` $1*T_0(x)*T_0(y) + 1*T_1(x)*T_0(y) +$ 
681: (4)                  $2*T_0(x)*T_1(y) + 2*T_1(x)*T_1(y)$ `` if axis=0 is ``x`` and axis=1 is
682: (4)                 ``y``.
683: (4)                 Parameters
684: (4)                 -----
685: (4)                 c : array_like
686: (8)                     Array of Chebyshev series coefficients. If c is multidimensional
687: (8)                     the different axis correspond to different variables with the
688: (8)                     degree in each axis given by the corresponding index.
689: (4)                 m : int, optional
690: (8)                     Number of derivatives taken, must be non-negative. (Default: 1)
691: (4)                 scl : scalar, optional
692: (8)                     Each differentiation is multiplied by `scl`. The end result is
693: (8)                     multiplication by ``scl**m``. This is for use in a linear change of
694: (8)                     variable. (Default: 1)
695: (4)                 axis : int, optional
696: (8)                     Axis over which the derivative is taken. (Default: 0).
697: (8)                     .. versionadded:: 1.7.0
698: (4)             Returns
699: (4)                 -----
700: (4)                 der : ndarray
701: (8)                     Chebyshev series of the derivative.
702: (4)             See Also
703: (4)                 -----
704: (4)                 chebint
705: (4)                 Notes
706: (4)                 -----
707: (4)                 In general, the result of differentiating a C-series needs to be
708: (4)                 "reprojected" onto the C-series basis set. Thus, typically, the
709: (4)                 result of this function is "unintuitive," albeit correct; see Examples
710: (4)                 section below.
711: (4)             Examples
712: (4)                 -----
713: (4)                 >>> from numpy.polynomial import chebyshev as C
714: (4)                 >>> c = (1,2,3,4)
715: (4)                 >>> C.chebder(c)
716: (4)                 array([14., 12., 24.])
717: (4)                 >>> C.chebder(c,3)
718: (4)                 array([96.])
719: (4)                 >>> C.chebder(c,scl=-1)
720: (4)                 array([-14., -12., -24.])
721: (4)                 >>> C.chebder(c,2,-1)
722: (4)                 array([12., 96.])
723: (4)                 """
724: (4)                 c = np.array(c, ndmin=1, copy=True)
725: (4)                 if c.dtype.char in '?bBhHiIlLqQpP':
726: (8)                     c = c.astype(np.double)
727: (4)                 cnt = pu._deprecate_as_int(m, "the order of derivation")
728: (4)                 iaxis = pu._deprecate_as_int(axis, "the axis")
729: (4)                 if cnt < 0:
730: (8)                     raise ValueError("The order of derivation must be non-negative")
731: (4)                 iaxis = normalize_axis_index(iaxis, c.ndim)
732: (4)                 if cnt == 0:

```

```

733: (8)             return c
734: (4)             c = np.moveaxis(c, iaxis, 0)
735: (4)             n = len(c)
736: (4)             if cnt >= n:
737: (8)                 c = c[:1]*0
738: (4)             else:
739: (8)                 for i in range(cnt):
740: (12)                     n = n - 1
741: (12)                     c *= scl
742: (12)                     der = np.empty((n,) + c.shape[1:], dtype=c.dtype)
743: (12)                     for j in range(n, 2, -1):
744: (16)                         der[j - 1] = (2*j)*c[j]
745: (16)                         c[j - 2] += (j*c[j])/(j - 2)
746: (12)                     if n > 1:
747: (16)                         der[1] = 4*c[2]
748: (12)                     der[0] = c[1]
749: (12)                     c = der
750: (4)             c = np.moveaxis(c, 0, iaxis)
751: (4)             return c
752: (0)             def chebint(c, m=1, k=[], lbnd=0, scl=1, axis=0):
753: (4)                 """
754: (4)                 Integrate a Chebyshev series.
755: (4)                 Returns the Chebyshev series coefficients `c` integrated `m` times from
756: (4)                 `lbnd` along `axis`. At each iteration the resulting series is
757: (4)                 **multiplied** by `scl` and an integration constant, `k`, is added.
758: (4)                 The scaling factor is for use in a linear change of variable. ("Buyer
759: (4)                 beware": note that, depending on what one is doing, one may want `scl`
760: (4)                 to be the reciprocal of what one might expect; for more information,
761: (4)                 see the Notes section below.) The argument `c` is an array of
762: (4)                 coefficients from low to high degree along each axis, e.g., [1,2,3]
763: (4)                 represents the series `` $T_0 + 2T_1 + 3T_2$ `` while [[1,2],[1,2]]
764: (4)                 represents `` $1*T_0(x)*T_0(y) + 1*T_1(x)*T_0(y) + 2*T_0(x)*T_1(y) +$ 
765: (4)                  $2*T_1(x)*T_1(y)$ `` if axis=0 is ``x`` and axis=1 is ``y``.
766: (4)                 Parameters
767: (4)                 -----
768: (4)                 c : array_like
769: (8)                     Array of Chebyshev series coefficients. If c is multidimensional
770: (8)                     the different axis correspond to different variables with the
771: (8)                     degree in each axis given by the corresponding index.
772: (4)                 m : int, optional
773: (8)                     Order of integration, must be positive. (Default: 1)
774: (4)                 k : [], list, scalar, optional
775: (8)                     Integration constant(s). The value of the first integral at zero
776: (8)                     is the first value in the list, the value of the second integral
777: (8)                     at zero is the second value, etc. If ``k == []`` (the default),
778: (8)                     all constants are set to zero. If ``m == 1``, a single scalar can
779: (8)                     be given instead of a list.
780: (4)                 lbnd : scalar, optional
781: (8)                     The lower bound of the integral. (Default: 0)
782: (4)                 scl : scalar, optional
783: (8)                     Following each integration the result is *multiplied* by `scl`
784: (8)                     before the integration constant is added. (Default: 1)
785: (4)                 axis : int, optional
786: (8)                     Axis over which the integral is taken. (Default: 0).
787: (8)                     .. versionadded:: 1.7.0
788: (4)             Returns
789: (4)             -----
790: (4)             S : ndarray
791: (8)                 C-series coefficients of the integral.
792: (4)             Raises
793: (4)             -----
794: (4)             ValueError
795: (8)                 If ``m < 1``, ``len(k) > m``, ``np.ndim(lbnd) != 0``, or
796: (8)                 ``np.ndim(scl) != 0``.
797: (4)             See Also
798: (4)             -----
799: (4)             chebder
800: (4)             Notes
801: (4)             -----

```

```

802: (4) Note that the result of each integration is *multiplied* by `scl`.
803: (4) Why is this important to note? Say one is making a linear change of
804: (4) variable :math:`u = ax + b` in an integral relative to `x`. Then
805: (4) :math:`dx = du/a`, so one will need to set `scl` equal to
806: (4) :math:`1/a` - perhaps not what one would have first thought.
807: (4) Also note that, in general, the result of integrating a C-series needs
808: (4) to be "reprojected" onto the C-series basis set. Thus, typically,
809: (4) the result of this function is "unintuitive," albeit correct; see
810: (4) Examples section below.
811: (4) Examples
812: (4) -----
813: (4) >>> from numpy.polynomial import chebyshev as C
814: (4) >>> c = (1,2,3)
815: (4) >>> C.chebint(c)
816: (4) array([ 0.5, -0.5,  0.5,  0.5])
817: (4) >>> C.chebint(c,3)
818: (4) array([ 0.03125   , -0.1875   ,  0.04166667, -0.05208333,  0.01041667, #
may vary
819: (8)      0.00625   ])
820: (4) >>> C.chebint(c, k=3)
821: (4) array([ 3.5, -0.5,  0.5,  0.5])
822: (4) >>> C.chebint(c,lbnd=-2)
823: (4) array([ 8.5, -0.5,  0.5,  0.5])
824: (4) >>> C.chebint(c,scl=-2)
825: (4) array([-1.,  1., -1., -1.])
826: (4) """
827: (4) c = np.array(c, ndmin=1, copy=True)
828: (4) if c.dtype.char in '?bBhHiIlLqQpP':
829: (8)     c = c.astype(np.double)
830: (4) if not np.iterable(k):
831: (8)     k = [k]
832: (4) cnt = pu._deprecate_as_int(m, "the order of integration")
833: (4) iaxis = pu._deprecate_as_int(axis, "the axis")
834: (4) if cnt < 0:
835: (8)     raise ValueError("The order of integration must be non-negative")
836: (4) if len(k) > cnt:
837: (8)     raise ValueError("Too many integration constants")
838: (4) if np.ndim(lbnd) != 0:
839: (8)     raise ValueError("lbnd must be a scalar.")
840: (4) if np.ndim(scl) != 0:
841: (8)     raise ValueError("scl must be a scalar.")
842: (4) iaxis = normalize_axis_index(iaxis, c.ndim)
843: (4) if cnt == 0:
844: (8)     return c
845: (4) c = np.moveaxis(c, iaxis, 0)
846: (4) k = list(k) + [0]*(cnt - len(k))
847: (4) for i in range(cnt):
848: (8)     n = len(c)
849: (8)     c *= scl
850: (8)     if n == 1 and np.all(c[0] == 0):
851: (12)         c[0] += k[i]
852: (8)     else:
853: (12)         tmp = np.empty((n + 1,) + c.shape[1:], dtype=c.dtype)
854: (12)         tmp[0] = c[0]*0
855: (12)         tmp[1] = c[0]
856: (12)         if n > 1:
857: (16)             tmp[2] = c[1]/4
858: (12)             for j in range(2, n):
859: (16)                 tmp[j + 1] = c[j]/(2*(j + 1))
860: (16)                 tmp[j - 1] -= c[j]/(2*(j - 1))
861: (12)             tmp[0] += k[i] - chebval(lbnd, tmp)
862: (12)             c = tmp
863: (4)         c = np.moveaxis(c, 0, iaxis)
864: (4)     return c
865: (0) def chebval(x, c, tensor=True):
866: (4) """
867: (4) Evaluate a Chebyshev series at points x.
868: (4) If `c` is of length `n + 1`, this function returns the value:
.. math:: p(x) = c_0 * T_0(x) + c_1 * T_1(x) + \dots + c_n * T_n(x)

```

```

870: (4) The parameter `x` is converted to an array only if it is a tuple or a
871: (4) list, otherwise it is treated as a scalar. In either case, either `x`
872: (4) or its elements must support multiplication and addition both with
873: (4) themselves and with the elements of `c`.
874: (4) If `c` is a 1-D array, then `p(x)` will have the same shape as `x`. If
875: (4) `c` is multidimensional, then the shape of the result depends on the
876: (4) value of `tensor`. If `tensor` is true the shape will be c.shape[1:] +
877: (4) x.shape. If `tensor` is false the shape will be c.shape[1:].
878: (4) Note that scalars have shape (,).
879: (4) Trailing zeros in the coefficients will be used in the evaluation, so
880: (4) they should be avoided if efficiency is a concern.
881: (4) Parameters
882: (4) -----
883: (4) x : array_like, compatible object
884: (8) If `x` is a list or tuple, it is converted to an ndarray, otherwise
885: (8) it is left unchanged and treated as a scalar. In either case, `x`
886: (8) or its elements must support addition and multiplication with
887: (8) themselves and with the elements of `c`.
888: (4) c : array_like
889: (8) Array of coefficients ordered so that the coefficients for terms of
890: (8) degree n are contained in c[n]. If `c` is multidimensional the
891: (8) remaining indices enumerate multiple polynomials. In the two
892: (8) dimensional case the coefficients may be thought of as stored in
893: (8) the columns of `c`.
894: (4) tensor : boolean, optional
895: (8) If True, the shape of the coefficient array is extended with ones
896: (8) on the right, one for each dimension of `x`. Scalars have dimension 0
897: (8) for this action. The result is that every column of coefficients in
898: (8) `c` is evaluated for every element of `x`. If False, `x` is broadcast
899: (8) over the columns of `c` for the evaluation. This keyword is useful
900: (8) when `c` is multidimensional. The default value is True.
901: (8) .. versionadded:: 1.7.0
902: (4) Returns
903: (4) -----
904: (4) values : ndarray, algebra_like
905: (8) The shape of the return value is described above.
906: (4) See Also
907: (4) -----
908: (4) chebval2d, chebgrid2d, chebval3d, chebgrid3d
909: (4) Notes
910: (4) -----
911: (4) The evaluation uses Clenshaw recursion, aka synthetic division.
912: (4) """
913: (4) c = np.array(c, ndmin=1, copy=True)
914: (4) if c.dtype.char in '?bBhHiIlLqQpP':
915: (8)     c = c.astype(np.double)
916: (4) if isinstance(x, (tuple, list)):
917: (8)     x = np.asarray(x)
918: (4) if isinstance(x, np.ndarray) and tensor:
919: (8)     c = c.reshape(c.shape + (1,) * x.ndim)
920: (4) if len(c) == 1:
921: (8)     c0 = c[0]
922: (8)     c1 = 0
923: (4) elif len(c) == 2:
924: (8)     c0 = c[0]
925: (8)     c1 = c[1]
926: (4) else:
927: (8)     x2 = 2*x
928: (8)     c0 = c[-2]
929: (8)     c1 = c[-1]
930: (8)     for i in range(3, len(c) + 1):
931: (12)         tmp = c0
932: (12)         c0 = c[-i] - c1
933: (12)         c1 = tmp + c1*x2
934: (4)     return c0 + c1*x
935: (0) def chebval2d(x, y, c):
936: (4) """
937: (4) Evaluate a 2-D Chebyshev series at points (x, y).
938: (4) This function returns the values:

```

```

939: (4) .. math:: p(x,y) = \sum_{i,j} c_{i,j} * T_i(x) * T_j(y)
940: (4) The parameters `x` and `y` are converted to arrays only if they are
941: (4) tuples or a lists, otherwise they are treated as a scalars and they
942: (4) must have the same shape after conversion. In either case, either `x`
943: (4) and `y` or their elements must support multiplication and addition both
944: (4) with themselves and with the elements of `c`.
945: (4) If `c` is a 1-D array a one is implicitly appended to its shape to make
946: (4) it 2-D. The shape of the result will be c.shape[2:] + x.shape.
947: (4) Parameters
948: (4) -----
949: (4) x, y : array_like, compatible objects
950: (8) The two dimensional series is evaluated at the points `(x, y)`, 
951: (8) where `x` and `y` must have the same shape. If `x` or `y` is a list
952: (8) or tuple, it is first converted to an ndarray, otherwise it is left
953: (8) unchanged and if it isn't an ndarray it is treated as a scalar.
954: (4) c : array_like
955: (8) Array of coefficients ordered so that the coefficient of the term
956: (8) of multi-degree i,j is contained in ``c[i,j]``. If `c` has
957: (8) dimension greater than 2 the remaining indices enumerate multiple
958: (8) sets of coefficients.
959: (4) Returns
960: (4) -----
961: (4) values : ndarray, compatible object
962: (8) The values of the two dimensional Chebyshev series at points formed
963: (8) from pairs of corresponding values from `x` and `y`.
964: (4) See Also
965: (4) -----
966: (4) chebval, chebgrid2d, chebval3d, chebgrid3d
967: (4) Notes
968: (4) -----
969: (4) .. versionadded:: 1.7.0
970: (4) """
971: (4)     return pu._valnd(chebval, c, x, y)
972: (0) def chebgrid2d(x, y, c):
973: (4) """
974: (4) Evaluate a 2-D Chebyshev series on the Cartesian product of x and y.
975: (4) This function returns the values:
976: (4) .. math:: p(a,b) = \sum_{i,j} c_{i,j} * T_i(a) * T_j(b),
977: (4) where the points `(a, b)` consist of all pairs formed by taking
978: (4) `a` from `x` and `b` from `y`. The resulting points form a grid with
979: (4) `x` in the first dimension and `y` in the second.
980: (4) The parameters `x` and `y` are converted to arrays only if they are
981: (4) tuples or a lists, otherwise they are treated as a scalars. In either
982: (4) case, either `x` and `y` or their elements must support multiplication
983: (4) and addition both with themselves and with the elements of `c`.
984: (4) If `c` has fewer than two dimensions, ones are implicitly appended to
985: (4) its shape to make it 2-D. The shape of the result will be c.shape[2:] +
986: (4) x.shape + y.shape.
987: (4) Parameters
988: (4) -----
989: (4) x, y : array_like, compatible objects
990: (8) The two dimensional series is evaluated at the points in the
991: (8) Cartesian product of `x` and `y`. If `x` or `y` is a list or
992: (8) tuple, it is first converted to an ndarray, otherwise it is left
993: (8) unchanged and, if it isn't an ndarray, it is treated as a scalar.
994: (4) c : array_like
995: (8) Array of coefficients ordered so that the coefficient of the term of
996: (8) multi-degree i,j is contained in `c[i,j]`. If `c` has dimension
997: (8) greater than two the remaining indices enumerate multiple sets of
998: (8) coefficients.
999: (4) Returns
1000: (4) -----
1001: (4) values : ndarray, compatible object
1002: (8) The values of the two dimensional Chebyshev series at points in the
1003: (8) Cartesian product of `x` and `y`.
1004: (4) See Also
1005: (4) -----
1006: (4) chebval, chebval2d, chebval3d, chebgrid3d
1007: (4) Notes

```

```

1008: (4)
1009: (4)
1010: (4)
1011: (4)
1012: (0)
1013: (4)
1014: (4)
1015: (4)
1016: (4)
1017: (4)
1018: (4)
1019: (4)
1020: (4)
1021: (4)
1022: (4)
1023: (4)
1024: (4)
1025: (4)
1026: (4)
1027: (4)
1028: (8)
1029: (8)
1030: (8)
1031: (8)
1032: (8)
1033: (4)
1034: (8)
1035: (8)
1036: (8)
1037: (8)
1038: (4)
1039: (4)
1040: (4)
1041: (8)
1042: (8)
1043: (4)
1044: (4)
1045: (4)
1046: (4)
1047: (4)
1048: (4)
1049: (4)
1050: (4)
1051: (0)
1052: (4)
1053: (4)
1054: (4)
1055: (4)
1056: (4)
1057: (4)
1058: (4)
1059: (4)
1060: (4)
1061: (4)
1062: (4)
1063: (4)
1064: (4)
1065: (4)
1066: (4)
1067: (4)
1068: (4)
1069: (4)
1070: (4)
1071: (8)
1072: (8)
1073: (8)
1074: (8)
1075: (8)
1076: (4)

-----  

.. versionadded:: 1.7.0  

"""  

    return pu._gridnd(chebval, c, x, y)  

def chebval3d(x, y, z, c):  

    """  

        Evaluate a 3-D Chebyshev series at points (x, y, z).  

        This function returns the values:  

        .. math:: p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * T_i(x) * T_j(y) * T_k(z)  

        The parameters `x`, `y`, and `z` are converted to arrays only if  

        they are tuples or a lists, otherwise they are treated as a scalars and  

        they must have the same shape after conversion. In either case, either  

        `x`, `y`, and `z` or their elements must support multiplication and  

        addition both with themselves and with the elements of `c`.  

        If `c` has fewer than 3 dimensions, ones are implicitly appended to its  

        shape to make it 3-D. The shape of the result will be c.shape[3:] +  

        x.shape.  

    Parameters  

    -----  

    x, y, z : array_like, compatible object  

        The three dimensional series is evaluated at the points  

        `(x, y, z)`, where `x`, `y`, and `z` must have the same shape. If  

        any of `x`, `y`, or `z` is a list or tuple, it is first converted  

        to an ndarray, otherwise it is left unchanged and if it isn't an  

        ndarray it is treated as a scalar.  

    c : array_like  

        Array of coefficients ordered so that the coefficient of the term of  

        multi-degree i,j,k is contained in `c[i,j,k]`. If `c` has dimension  

        greater than 3 the remaining indices enumerate multiple sets of  

        coefficients.  

    Returns  

    -----  

    values : ndarray, compatible object  

        The values of the multidimensional polynomial on points formed with  

        triples of corresponding values from `x`, `y`, and `z`.  

    See Also  

    -----  

    chebval, chebval2d, chebgrid2d, chebgrid3d  

    Notes  

    -----  

.. versionadded:: 1.7.0  

"""  

    return pu._valnd(chebval, c, x, y, z)  

def chebgrid3d(x, y, z, c):  

    """  

        Evaluate a 3-D Chebyshev series on the Cartesian product of x, y, and z.  

        This function returns the values:  

        .. math:: p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * T_i(a) * T_j(b) * T_k(c)  

        where the points `(a, b, c)` consist of all triples formed by taking  

        `a` from `x`, `b` from `y`, and `c` from `z`. The resulting points form  

        a grid with `x` in the first dimension, `y` in the second, and `z` in  

        the third.  

        The parameters `x`, `y`, and `z` are converted to arrays only if they  

        are tuples or a lists, otherwise they are treated as a scalars. In  

        either case, either `x`, `y`, and `z` or their elements must support  

        multiplication and addition both with themselves and with the elements  

        of `c`.  

        If `c` has fewer than three dimensions, ones are implicitly appended to  

        its shape to make it 3-D. The shape of the result will be c.shape[3:] +  

        x.shape + y.shape + z.shape.  

    Parameters  

    -----  

    x, y, z : array_like, compatible objects  

        The three dimensional series is evaluated at the points in the  

        Cartesian product of `x`, `y`, and `z`. If `x`, `y`, or `z` is a  

        list or tuple, it is first converted to an ndarray, otherwise it is  

        left unchanged and, if it isn't an ndarray, it is treated as a  

        scalar.  

    c : array_like

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1077: (8)           Array of coefficients ordered so that the coefficients for terms of
1078: (8)           degree i,j are contained in ``c[i,j]``. If `c` has dimension
1079: (8)           greater than two the remaining indices enumerate multiple sets of
1080: (8)           coefficients.
1081: (4)           Returns
1082: (4)           -----
1083: (4)           values : ndarray, compatible object
1084: (8)           The values of the two dimensional polynomial at points in the
Cartesian
1085: (8)           product of `x` and `y`.
1086: (4)           See Also
1087: (4)           -----
1088: (4)           chebval, chebval2d, chebgrid2d, chebval3d
1089: (4)           Notes
1090: (4)           -----
1091: (4)           .. versionadded:: 1.7.0
1092: (4)           """
1093: (4)           return pu._gridnd(chebval, c, x, y, z)
def chebvander(x, deg):
    """Pseudo-Vandermonde matrix of given degree.
    Returns the pseudo-Vandermonde matrix of degree `deg` and sample points
    `x`. The pseudo-Vandermonde matrix is defined by
    .. math:: V[..., i] = T_i(x),
    where `0 <= i <= deg`. The leading indices of `V` index the elements of
    `x` and the last index is the degree of the Chebyshev polynomial.
    If `c` is a 1-D array of coefficients of length `n + 1` and `V` is the
    matrix ``V = chebvander(x, n)``, then ``np.dot(V, c)`` and
    ``chebval(x, c)`` are the same up to roundoff. This equivalence is
    useful both for least squares fitting and for the evaluation of a large
    number of Chebyshev series of the same degree and sample points.
    Parameters
    -----
    x : array_like
        Array of points. The dtype is converted to float64 or complex128
        depending on whether any of the elements are complex. If `x` is
        scalar it is converted to a 1-D array.
    deg : int
        Degree of the resulting matrix.
    Returns
    -----
    vander : ndarray
        The pseudo Vandermonde matrix. The shape of the returned matrix is
        ``x.shape + (deg + 1,)``, where The last index is the degree of the
        corresponding Chebyshev polynomial. The dtype will be the same as
        the converted `x`.
    """
    ideg = pu._deprecate_as_int(deg, "deg")
    if ideg < 0:
        raise ValueError("deg must be non-negative")
    x = np.array(x, copy=False, ndmin=1) + 0.0
    dims = (ideg + 1,) + x.shape
    dtyp = x.dtype
    v = np.empty(dims, dtype=dtyp)
    v[0] = x**0 + 1
    if ideg > 0:
        x2 = 2*x
        v[1] = x
        for i in range(2, ideg + 1):
            v[i] = v[i-1]*x2 - v[i-2]
    return np.moveaxis(v, 0, -1)
def chebvander2d(x, y, deg):
    """Pseudo-Vandermonde matrix of given degrees.
    Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
    points `(x, y)`. The pseudo-Vandermonde matrix is defined by
    .. math:: V[..., (deg[1] + 1)*i + j] = T_i(x) * T_j(y),
    where `0 <= i <= deg[0]` and `0 <= j <= deg[1]`. The leading indices of
    `V` index the points `(x, y)` and the last index encodes the degrees of
    the Chebyshev polynomials.
    If ``V = chebvander2d(x, y, [xdeg, ydeg])``, then the columns of `V`

```

```

1145: (4) correspond to the elements of a 2-D coefficient array `c` of shape
1146: (4) (xdeg + 1, ydeg + 1) in the order
1147: (4) .. math:: c_{00}, c_{01}, c_{02} \dots , c_{10}, c_{11}, c_{12} \dots
1148: (4) and ``np.dot(V, c.flat)`` and ``chebval2d(x, y, c)`` will be the same
1149: (4) up to roundoff. This equivalence is useful both for least squares
1150: (4) fitting and for the evaluation of a large number of 2-D Chebyshev
1151: (4) series of the same degrees and sample points.
1152: (4) Parameters
1153: (4) -----
1154: (4) x, y : array_like
1155: (8) Arrays of point coordinates, all of the same shape. The dtypes
1156: (8) will be converted to either float64 or complex128 depending on
1157: (8) whether any of the elements are complex. Scalars are converted to
1158: (8) 1-D arrays.
1159: (4) deg : list of ints
1160: (8) List of maximum degrees of the form [x_deg, y_deg].
1161: (4) Returns
1162: (4) -----
1163: (4) vander2d : ndarray
1164: (8) The shape of the returned matrix is ``x.shape + (order,)``, where
1165: (8) :math:`\text{order} = (\text{deg}[0]+1)*(\text{deg}[1]+1)` . The dtype will be the same
1166: (8) as the converted `x` and `y` .
1167: (4) See Also
1168: (4) -----
1169: (4) chebvander, chebvander3d, chebval2d, chebval3d
1170: (4) Notes
1171: (4) -----
1172: (4) .. versionadded:: 1.7.0
1173: (4) """
1174: (4) return pu._vander_nd_flat((chebvander, chebvander), (x, y), deg)
1175: (0) def chebvander3d(x, y, z, deg):
1176: (4) """Pseudo-Vandermonde matrix of given degrees.
1177: (4) Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
1178: (4) points `(x, y, z)` . If `l, m, n` are the given degrees in `x, y, z` ,
1179: (4) then The pseudo-Vandermonde matrix is defined by
1180: (4) .. math:: V[..., (m+1)(n+1)i + (n+1)j + k] = T_i(x)*T_j(y)*T_k(z),
1181: (4) where `0 <= i <= l` , `0 <= j <= m` , and `0 <= j <= n` . The leading
1182: (4) indices of `V` index the points `(x, y, z)` and the last index encodes
1183: (4) the degrees of the Chebyshev polynomials.
1184: (4) If ``V = chebvander3d(x, y, z, [xdeg, ydeg, zdeg])`` , then the columns
1185: (4) of `V` correspond to the elements of a 3-D coefficient array `c` of
1186: (4) shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order
1187: (4) .. math:: c_{000}, c_{001}, c_{002}, \dots , c_{010}, c_{011}, c_{012}, \dots
1188: (4) and ``np.dot(V, c.flat)`` and ``chebval3d(x, y, z, c)`` will be the
1189: (4) same up to roundoff. This equivalence is useful both for least squares
1190: (4) fitting and for the evaluation of a large number of 3-D Chebyshev
1191: (4) series of the same degrees and sample points.
1192: (4) Parameters
1193: (4) -----
1194: (4) x, y, z : array_like
1195: (8) Arrays of point coordinates, all of the same shape. The dtypes will
1196: (8) be converted to either float64 or complex128 depending on whether
1197: (8) any of the elements are complex. Scalars are converted to 1-D
1198: (8) arrays.
1199: (4) deg : list of ints
1200: (8) List of maximum degrees of the form [x_deg, y_deg, z_deg].
1201: (4) Returns
1202: (4) -----
1203: (4) vander3d : ndarray
1204: (8) The shape of the returned matrix is ``x.shape + (order,)`` , where
1205: (8) :math:`\text{order} = (\text{deg}[0]+1)*(\text{deg}[1]+1)*(\text{deg}[2]+1)` . The dtype will
1206: (8) be the same as the converted `x` , `y` , and `z` .
1207: (4) See Also
1208: (4) -----
1209: (4) chebvander, chebvander3d, chebval2d, chebval3d
1210: (4) Notes
1211: (4) -----
1212: (4) .. versionadded:: 1.7.0
1213: (4) """

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1214: (4)
1215: (0)    return pu._vander_nd_flat((chebvander, chebvander, chebvander), (x, y, z),
1216: (4)
1217: (4)        deg)
1218: (4)    def chebfit(x, y, deg, rcond=None, full=False, w=None):
1219: (4)        """
1220: (4)            Least squares fit of Chebyshev series to data.
1221: (4)            Return the coefficients of a Chebyshev series of degree `deg` that is the
1222: (4)            least squares fit to the data values `y` given at points `x`. If `y` is
1223: (4)            1-D the returned coefficients will also be 1-D. If `y` is 2-D multiple
1224: (4)            fits are done, one for each column of `y`, and the resulting
1225: (4)            coefficients are stored in the corresponding columns of a 2-D return.
1226: (4)            The fitted polynomial(s) are in the form
1227: (4)                .. math:: p(x) = c_0 + c_1 * T_1(x) + \dots + c_n * T_n(x),
1228: (4)            where `n` is `deg`.
1229: (8)        Parameters
1230: (4)        -----
1231: (4)        x : array_like, shape (M,)
1232: (8)            x-coordinates of the M sample points ``(x[i], y[i])``.
1233: (4)        y : array_like, shape (M,) or (M, K)
1234: (8)            y-coordinates of the sample points. Several data sets of sample
1235: (8)            points sharing the same x-coordinates can be fitted at once by
1236: (8)            passing in a 2D-array that contains one dataset per column.
1237: (8)        deg : int or 1-D array_like
1238: (8)            Degree(s) of the fitting polynomials. If `deg` is a single integer,
1239: (4)            all terms up to and including the `deg`'th term are included in the
1240: (8)            fit. For NumPy versions >= 1.11.0 a list of integers specifying the
1241: (8)            degrees of the terms to include may be used instead.
1242: (8)        rcond : float, optional
1243: (8)            Relative condition number of the fit. Singular values smaller than
1244: (8)            this relative to the largest singular value will be ignored. The
1245: (8)            default value is len(x)*eps, where eps is the relative precision of
1246: (8)            the float type, about 2e-16 in most cases.
1247: (8)        full : bool, optional
1248: (8)            Switch determining nature of return value. When it is False (the
1249: (8)            default) just the coefficients are returned, when True diagnostic
1250: (8)            information from the singular value decomposition is also returned.
1251: (8)        w : array_like, shape (M,), optional
1252: (8)            Weights. If not None, the weight ``w[i]`` applies to the unsquared
1253: (8)            residual ``y[i] - y_hat[i]`` at ``x[i]``. Ideally the weights are
1254: (8)            chosen so that the errors of the products ``w[i]*y[i]`` all have the
1255: (8)            same variance. When using inverse-variance weighting, use
1256: (8)                ``w[i] = 1/sigma(y[i])``. The default value is None.
1257: (8)            .. versionadded:: 1.5.0
1258: (4)        Returns
1259: (4)        -----
1260: (4)        coef : ndarray, shape (M,) or (M, K)
1261: (8)            Chebyshev coefficients ordered from low to high. If `y` was 2-D,
1262: (8)            the coefficients for the data in column k of `y` are in column
1263: (8)                `k`.
1264: (8)        [residuals, rank, singular_values, rcond] : list
1265: (8)            These values are only returned if ``full == True``
1266: (8)            - residuals -- sum of squared residuals of the least squares fit
1267: (8)            - rank -- the numerical rank of the scaled Vandermonde matrix
1268: (8)            - singular_values -- singular values of the scaled Vandermonde matrix
1269: (8)            - rcond -- value of `rcond`.
1270: (8)            For more details, see `numpy.linalg.lstsq`.
1271: (4)        Warns
1272: (4)        -----
1273: (4)        RankWarning
1274: (8)            The rank of the coefficient matrix in the least-squares fit is
1275: (8)            deficient. The warning is only raised if ``full == False``. The
1276: (8)            warnings can be turned off by
1277: (8)                >>> import warnings
1278: (8)                >>> warnings.simplefilter('ignore', np.RankWarning)
1279: (4)        See Also
1280: (4)        -----
1281: (4)        numpy.polynomial.polynomial.polyfit
1282: (4)        numpy.polynomial.legendre.legfit
1283: (4)        numpy.polynomial.laguerre.lagfit
1284: (4)        numpy.polynomial.hermite.hermfit

```

```

1282: (4) numpy.polynomial.hermite_e.hermefit
1283: (4) chebval : Evaluates a Chebyshev series.
1284: (4) chebvander : Vandermonde matrix of Chebyshev series.
1285: (4) chebweight : Chebyshev weight function.
1286: (4) numpy.linalg.lstsq : Computes a least-squares fit from the matrix.
1287: (4) scipy.interpolate.UnivariateSpline : Computes spline fits.
1288: (4) Notes
1289: (4)
1290: (4) -----
1291: (4) The solution is the coefficients of the Chebyshev series `p` that
1292: (4) minimizes the sum of the weighted squared errors
1293: (4) .. math:: E = \sum_j w_j^2 * |y_j - p(x_j)|^2,
1294: (4) where :math:`w_j` are the weights. This problem is solved by setting up
1295: (4) as the (typically) overdetermined matrix equation
1296: (4) .. math:: V(x) * c = w * y,
1297: (4) where `V` is the weighted pseudo Vandermonde matrix of `x`, `c` are the
1298: (4) coefficients to be solved for, `w` are the weights, and `y` are the
1299: (4) observed values. This equation is then solved using the singular value
1300: (4) decomposition of `V`.
1301: (4) If some of the singular values of `V` are so small that they are
1302: (4) neglected, then a `RankWarning` will be issued. This means that the
1303: (4) coefficient values may be poorly determined. Using a lower order fit
1304: (4) will usually get rid of the warning. The `rcond` parameter can also be
1305: (4) set to a value smaller than its default, but the resulting fit may be
1306: (4) spurious and have large contributions from roundoff error.
1307: (4) Fits using Chebyshev series are usually better conditioned than fits
1308: (4) using power series, but much can depend on the distribution of the
1309: (4) sample points and the smoothness of the data. If the quality of the fit
1310: (4) is inadequate splines may be a good alternative.
1311: (4) References
1312: (4) -----
1313: (11) .. [1] Wikipedia, "Curve fitting",
1314: (4) https://en.wikipedia.org/wiki/Curve_fitting
1315: (4) Examples
1316: (4) -----
1317: (4) """
1318: (0) def chebcompanion(c):
1319: (4) """Return the scaled companion matrix of c.
1320: (4) The basis polynomials are scaled so that the companion matrix is
1321: (4) symmetric when `c` is a Chebyshev basis polynomial. This provides
1322: (4) better eigenvalue estimates than the unscaled case and for basis
1323: (4) polynomials the eigenvalues are guaranteed to be real if
1324: (4) `numpy.linalg.eigvalsh` is used to obtain them.
1325: (4) Parameters
1326: (4) -----
1327: (4) c : array_like
1328: (8)     1-D array of Chebyshev series coefficients ordered from low to high
1329: (8)     degree.
1330: (4) Returns
1331: (4) -----
1332: (4) mat : ndarray
1333: (8)     Scaled companion matrix of dimensions (deg, deg).
1334: (4) Notes
1335: (4) -----
1336: (4) .. versionadded:: 1.7.0
1337: (4) """
1338: (4) [c] = pu.as_series([c])
1339: (4) if len(c) < 2:
1340: (8)     raise ValueError('Series must have maximum degree of at least 1.')
1341: (4) if len(c) == 2:
1342: (8)     return np.array([-c[0]/c[1]])
1343: (4) n = len(c) - 1
1344: (4) mat = np.zeros((n, n), dtype=c.dtype)
1345: (4) scl = np.array([1.] + [np.sqrt(.5)]*(n-1))
1346: (4) top = mat.reshape(-1)[1::n+1]
1347: (4) bot = mat.reshape(-1)[n::n+1]
1348: (4) top[0] = np.sqrt(.5)
1349: (4) top[1:] = 1/2
1350: (4) bot[...] = top

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1351: (4) mat[:, -1] -= (c[:-1]/c[-1])*(scl/scl[-1]).*.5
1352: (4) return mat
1353: (0) def chebroots(c):
1354: (4) """
1355: (4)     Compute the roots of a Chebyshev series.
1356: (4)     Return the roots (a.k.a. "zeros") of the polynomial
1357: (4)     .. math:: p(x) = \sum_i c[i] * T_i(x).
1358: (4)     Parameters
1359: (4)     -----
1360: (4)     c : 1-D array_like
1361: (8)         1-D array of coefficients.
1362: (4)     Returns
1363: (4)     -----
1364: (4)     out : ndarray
1365: (8)         Array of the roots of the series. If all the roots are real,
1366: (8)             then `out` is also real, otherwise it is complex.
1367: (4)     See Also
1368: (4)     -----
1369: (4)     numpy.polynomial.polynomial.polyroots
1370: (4)     numpy.polynomial.legendre.legroots
1371: (4)     numpy.polynomial.laguerre.lagroots
1372: (4)     numpy.polynomial.hermite.hermroots
1373: (4)     numpy.polynomial.hermite_e.hermroots
1374: (4)     Notes
1375: (4)     -----
1376: (4)     The root estimates are obtained as the eigenvalues of the companion
1377: (4)     matrix, Roots far from the origin of the complex plane may have large
1378: (4)     errors due to the numerical instability of the series for such
1379: (4)     values. Roots with multiplicity greater than 1 will also show larger
1380: (4)     errors as the value of the series near such points is relatively
1381: (4)     insensitive to errors in the roots. Isolated roots near the origin can
1382: (4)     be improved by a few iterations of Newton's method.
1383: (4)     The Chebyshev series basis polynomials aren't powers of `x` so the
1384: (4)     results of this function may seem unintuitive.
1385: (4)     Examples
1386: (4)     -----
1387: (4)     >>> import numpy.polynomial.chebyshev as cheb
1388: (4)     >>> cheb.chebroots((-1, 1,-1, 1)) # T3 - T2 + T1 - T0 has real roots
1389: (4)     array([-5.00000000e-01, 2.60860684e-17, 1.00000000e+00]) # may vary
1390: (4)
1391: (4)     [c] = pu.as_series([c])
1392: (4)     if len(c) < 2:
1393: (8)         return np.array([], dtype=c.dtype)
1394: (4)     if len(c) == 2:
1395: (8)         return np.array([-c[0]/c[1]])
1396: (4)     m = chebcompanion(c)[::-1,::-1]
1397: (4)     r = la.eigvals(m)
1398: (4)     r.sort()
1399: (4)     return r
1400: (0) def chebinterpolate(func, deg, args=()):
1401: (4)     """
1402: (4)         Interpolate a function at the Chebyshev points of the first kind.
1403: (4)         Returns the Chebyshev series that interpolates `func` at the Chebyshev
1404: (4)         points of the first kind in the interval [-1, 1]. The interpolating
1405: (4)         series tends to a minmax approximation to `func` with increasing `deg`
1406: (4)         if the function is continuous in the interval.
1407: (4)         .. versionadded:: 1.14.0
1408: (4)         Parameters
1409: (4)         -----
1410: (8)         func : function
1411: (8)             The function to be approximated. It must be a function of a single
1412: (8)             variable of the form ``f(x, a, b, c...)``, where ``a, b, c...`` are
1413: (4)             extra arguments passed in the `args` parameter.
1414: (8)         deg : int
1415: (4)             Degree of the interpolating polynomial
1416: (8)         args : tuple, optional
1417: (8)             Extra arguments to be used in the function call. Default is no extra
1418: (4)             arguments.
1419: (4)         Returns
1420: (4)         -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1420: (4) coef : ndarray, shape (deg + 1,) Chebyshev coefficients of the interpolating series ordered from low to
1421: (8) high.
1422: (8)
1423: (4)
1424: (4)
1425: (4)
1426: (4)
1427: (4)
1428: (12)
1429: (13)
1430: (4)
1431: (4)
1432: (4) Examples
1433: (4)
1434: (4)
1435: (4)
1436: (4)
1437: (4)
1438: (4)
1439: (4)
1440: (4)
1441: (4)
1442: (8)
1443: (4)
1444: (8)
1445: (4)
1446: (4)
1447: (4)
1448: (4)
1449: (4)
1450: (4)
1451: (4)
1452: (4)
1453: (0)
1454: (4)
1455: (4)
1456: (4)
1457: (4)
1458: (4)
1459: (4)
1460: (4)
1461: (4)
1462: (4)
1463: (8)
1464: (4)
1465: (4)
1466: (4)
1467: (8)
1468: (4)
1469: (8)
1470: (4)
1471: (4)
1472: (4)
1473: (4)
1474: (4)
1475: (4)
1476: (4)
1477: (4)
1478: (4)
1479: (4)
1480: (4)
1481: (8)
1482: (4)
1483: (4)
1484: (4)
1485: (0)
1486: (4)
1487: (4)
1488: (4)

coef : ndarray, shape (deg + 1,) Chebyshev coefficients of the interpolating series ordered from low to
high.
Examples
-----
>>> import numpy.polynomial.chebyshev as C
>>> C.chebfromfunction(lambda x: np.tanh(x) + 0.5, 8)
array([ 5.00000000e-01,  8.11675684e-01, -9.86864911e-17,
       -5.42457905e-02, -2.71387850e-16,  4.51658839e-03,
       2.46716228e-17, -3.79694221e-04, -3.26899002e-16])

Notes
-----
The Chebyshev polynomials used in the interpolation are orthogonal when
sampled at the Chebyshev points of the first kind. If it is desired to
constrain some of the coefficients they can simply be set to the desired
value after the interpolation, no new interpolation or fit is needed. This
is especially useful if it is known apriori that some of coefficients are
zero. For instance, if the function is even then the coefficients of the
terms of odd degree in the result can be set to zero.
"""
deg = np.asarray(deg)
if deg.ndim > 0 or deg.dtype.kind not in 'iu' or deg.size == 0:
    raise TypeError("deg must be an int")
if deg < 0:
    raise ValueError("expected deg >= 0")
order = deg + 1
xcheb = chebpts1(order)
yfunc = func(xcheb, *args)
m = chebvander(xcheb, deg)
c = np.dot(m.T, yfunc)
c[0] /= order
c[1:] /= 0.5*order
return c

def chebgauss(deg):
"""
Gauss-Chebyshev quadrature.
Computes the sample points and weights for Gauss-Chebyshev quadrature.
These sample points and weights will correctly integrate polynomials of
degree :math:`2*deg - 1` or less over the interval :math:`[-1, 1]` with
the weight function :math:`f(x) = 1/\sqrt{1 - x^2}`.
Parameters
-----
deg : int
    Number of sample points and weights. It must be >= 1.
Returns
-----
x : ndarray
    1-D ndarray containing the sample points.
y : ndarray
    1-D ndarray containing the weights.
Notes
-----
.. versionadded:: 1.7.0
The results have only been tested up to degree 100, higher degrees may
be problematic. For Gauss-Chebyshev there are closed form solutions for
the sample points and weights. If n = `deg`, then
.. math:: x_i = \cos(\pi (2 i - 1) / (2 n))
.. math:: w_i = \pi / n
"""
ideg = pu._deprecate_as_int(deg, "deg")
if ideg <= 0:
    raise ValueError("deg must be a positive integer")
x = np.cos(np.pi * np.arange(1, 2*ideg, 2) / (2.0*ideg))
w = np.ones(ideg)*(np.pi/ideg)
return x, w

def chebweight(x):
"""
The weight function of the Chebyshev polynomials.
The weight function is :math:`1/\sqrt{1 - x^2}` and the interval of

```

```

1489: (4)             integration is :math:`[-1, 1]`. The Chebyshev polynomials are
1490: (4)             orthogonal, but not normalized, with respect to this weight function.
1491: (4)             Parameters
1492: (4)             -----
1493: (4)             x : array_like
1494: (7)                 Values at which the weight function will be computed.
1495: (4)             Returns
1496: (4)             -----
1497: (4)             w : ndarray
1498: (7)                 The weight function at `x`.
1499: (4)             Notes
1500: (4)             -----
1501: (4)             .. versionadded:: 1.7.0
1502: (4)             """
1503: (4)             w = 1. / (np.sqrt(1. + x) * np.sqrt(1. - x))
1504: (4)             return w
1505: (0)             def chebpts1(npts):
1506: (4)             """
1507: (4)                 Chebyshev points of the first kind.
1508: (4)                 The Chebyshev points of the first kind are the points ``cos(x)``,
1509: (4)                 where ``x = [pi*(k + .5)/npts for k in range(npts)]``.
1510: (4)                 Parameters
1511: (4)                 -----
1512: (4)                 npts : int
1513: (8)                     Number of sample points desired.
1514: (4)             Returns
1515: (4)             -----
1516: (4)             pts : ndarray
1517: (8)                     The Chebyshev points of the first kind.
1518: (4)             See Also
1519: (4)             -----
1520: (4)             chebpts2
1521: (4)             Notes
1522: (4)             -----
1523: (4)             .. versionadded:: 1.5.0
1524: (4)             """
1525: (4)             _npts = int(npts)
1526: (4)             if _npts != npts:
1527: (8)                 raise ValueError("npts must be integer")
1528: (4)             if _npts < 1:
1529: (8)                 raise ValueError("npts must be >= 1")
1530: (4)             x = 0.5 * np.pi / _npts * np.arange(-_npts+1, _npts+1, 2)
1531: (4)             return np.sin(x)
1532: (0)             def chebpts2(npts):
1533: (4)             """
1534: (4)                 Chebyshev points of the second kind.
1535: (4)                 The Chebyshev points of the second kind are the points ``cos(x)``,
1536: (4)                 where ``x = [pi*k/(npts - 1) for k in range(npts)]`` sorted in ascending
1537: (4)                 order.
1538: (4)                 Parameters
1539: (4)                 -----
1540: (4)                 npts : int
1541: (8)                     Number of sample points desired.
1542: (4)             Returns
1543: (4)             -----
1544: (4)             pts : ndarray
1545: (8)                     The Chebyshev points of the second kind.
1546: (4)             Notes
1547: (4)             -----
1548: (4)             .. versionadded:: 1.5.0
1549: (4)             """
1550: (4)             _npts = int(npts)
1551: (4)             if _npts != npts:
1552: (8)                 raise ValueError("npts must be integer")
1553: (4)             if _npts < 2:
1554: (8)                 raise ValueError("npts must be >= 2")
1555: (4)             x = np.linspace(-np.pi, 0, _npts)
1556: (4)             return np.cos(x)
1557: (0)             class Chebyshev(APCPolyBase):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1558: (4)      """A Chebyshev series class.
1559: (4)      The Chebyshev class provides the standard Python numerical methods
1560: (4)      '+', '-', '*', '//', '%', 'divmod', '**', and '(' as well as the
1561: (4)      methods listed below.
1562: (4)      Parameters
1563: (4)      -----
1564: (4)      coef : array_like
1565: (8)          Chebyshev coefficients in order of increasing degree, i.e.,
1566: (8)          ``(1, 2, 3)`` gives ``1*T_0(x) + 2*T_1(x) + 3*T_2(x)``.
1567: (4)      domain : (2,) array_like, optional
1568: (8)          Domain to use. The interval ``[domain[0], domain[1]]`` is mapped
1569: (8)          to the interval ``[window[0], window[1]]`` by shifting and scaling.
1570: (8)          The default value is [-1, 1].
1571: (4)      window : (2,) array_like, optional
1572: (8)          Window, see `domain` for its use. The default value is [-1, 1].
1573: (8)          .. versionadded:: 1.6.0
1574: (4)      symbol : str, optional
1575: (8)          Symbol used to represent the independent variable in string
1576: (8)          representations of the polynomial expression, e.g. for printing.
1577: (8)          The symbol must be a valid Python identifier. Default value is 'x'.
1578: (8)          .. versionadded:: 1.24
1579: (4)      """
1580: (4)      _add = staticmethod(chebadd)
1581: (4)      _sub = staticmethod(chebsub)
1582: (4)      _mul = staticmethod(chebmul)
1583: (4)      _div = staticmethod(chebdiv)
1584: (4)      _pow = staticmethod(chebpow)
1585: (4)      _val = staticmethod(chebval)
1586: (4)      _int = staticmethod(chebint)
1587: (4)      _der = staticmethod(chebder)
1588: (4)      _fit = staticmethod(chebfit)
1589: (4)      _line = staticmethod(chebline)
1590: (4)      _roots = staticmethod(chebroots)
1591: (4)      _fromroots = staticmethod(chebfromroots)
1592: (4)      @classmethod
1593: (4)      def interpolate(cls, func, deg, domain=None, args=()):
1594: (8)          """Interpolate a function at the Chebyshev points of the first kind.
1595: (8)          Returns the series that interpolates `func` at the Chebyshev points of
1596: (8)          the first kind scaled and shifted to the `domain`. The resulting
series
1597: (8)          tends to a minmax approximation of `func` when the function is
1598: (8)          continuous in the domain.
1599: (8)          .. versionadded:: 1.14.0
1600: (8)          Parameters
1601: (8)          -----
1602: (8)          func : function
1603: (12)          The function to be interpolated. It must be a function of a single
1604: (12)          variable of the form ``f(x, a, b, c...)``, where ``a, b, c...``
are
1605: (12)          extra arguments passed in the `args` parameter.
1606: (8)          deg : int
1607: (12)          Degree of the interpolating polynomial.
1608: (8)          domain : {None, [beg, end]}, optional
1609: (12)          Domain over which `func` is interpolated. The default is None, in
1610: (12)          which case the domain is [-1, 1].
1611: (8)          args : tuple, optional
1612: (12)          Extra arguments to be used in the function call. Default is no
1613: (12)          extra arguments.
1614: (8)          Returns
1615: (8)          -----
1616: (8)          polynomial : Chebyshev instance
1617: (12)          Interpolating Chebyshev instance.
1618: (8)          Notes
1619: (8)          -----
1620: (8)          See `numpy.polynomial.chebfromfunction` for more details.
1621: (8)          """
1622: (8)          if domain is None:
1623: (12)              domain = cls.domain
1624: (8)          xfunc = lambda x: func(pu.mapdomain(x, cls.window, domain), *args)

```

```

1625: (8)           coef = chebinterpolate(xfunc, deg)
1626: (8)           return cls(coef, domain=domain)
1627: (4)           domain = np.array(chebdomain)
1628: (4)           window = np.array(chebdomain)
1629: (4)           basis_name = 'T'
-----
```

File 288 - hermite.py:

```

1: (0)           """
2: (0)           =====
3: (0)           Hermite Series, "Physicists" (:mod:`numpy.polynomial.hermite`)
4: (0)           =====
5: (0)           This module provides a number of objects (mostly functions) useful for
6: (0)           dealing with Hermite series, including a `Hermite` class that
7: (0)           encapsulates the usual arithmetic operations. (General information
8: (0)           on how this module represents and works with such polynomials is in the
9: (0)           docstring for its "parent" sub-package, `numpy.polynomial`).
10: (0)          Classes
11: (0)          -----
12: (0)          .. autosummary::
13: (3)              :toctree: generated/
14: (3)                  Hermite
15: (0)          Constants
16: (0)          -----
17: (0)          .. autosummary::
18: (3)              :toctree: generated/
19: (3)                  hermdomain
20: (3)                  hermzero
21: (3)                  hermone
22: (3)                  hermx
23: (0)          Arithmetic
24: (0)          -----
25: (0)          .. autosummary::
26: (3)              :toctree: generated/
27: (3)                  hermadd
28: (3)                  hermsub
29: (3)                  hermmulx
30: (3)                  hermmul
31: (3)                  hermdiv
32: (3)                  hermpow
33: (3)                  hermval
34: (3)                  hermval2d
35: (3)                  hermval3d
36: (3)                  hermgrid2d
37: (3)                  hermgrid3d
38: (0)          Calculus
39: (0)          -----
40: (0)          .. autosummary::
41: (3)              :toctree: generated/
42: (3)                  hermder
43: (3)                  hermint
44: (0)          Misc Functions
45: (0)          -----
46: (0)          .. autosummary::
47: (3)              :toctree: generated/
48: (3)                  hermfromroots
49: (3)                  hermroots
50: (3)                  hermvander
51: (3)                  hermvander2d
52: (3)                  hermvander3d
53: (3)                  hermgauss
54: (3)                  hermweight
55: (3)                  hermcompanion
56: (3)                  hermfit
57: (3)                  hermtrim
58: (3)                  hermline
59: (3)                  herm2poly
```

```

60: (3)          poly2herm
61: (0)          See also
62: (0)          -----
63: (0)          `numpy.polynomial`_
64: (0)          """
65: (0)          import numpy as np
66: (0)          import numpy.linalg as la
67: (0)          from numpy.core.multiarray import normalize_axis_index
68: (0)          from . import polyutils as pu
69: (0)          from ._polybase import ABCPolyBase
70: (0)          __all__ = [
71: (4)          'hermzero', 'hermone', 'hermx', 'hermdomain', 'hermline', 'hermadd',
72: (4)          'hermsub', 'hermmulx', 'hermmul', 'hermdiv', 'hermpow', 'hermval',
73: (4)          'hermder', 'hermint', 'herm2poly', 'poly2herm', 'hermfromroots',
74: (4)          'hermvander', 'hermfit', 'hermtrim', 'hermroots', 'Hermite',
75: (4)          'hermval2d', 'hermval3d', 'hermgrid2d', 'hermgrid3d', 'hermvander2d',
76: (4)          'hermvander3d', 'hermcompanion', 'hermgauss', 'hermweight']
77: (0)          hermtrim = pu.trimcoef
78: (0)          def poly2herm(pol):
79: (4)          """
80: (4)          poly2herm(pol)
81: (4)          Convert a polynomial to a Hermite series.
82: (4)          Convert an array representing the coefficients of a polynomial (relative
83: (4)          to the "standard" basis) ordered from lowest degree to highest, to an
84: (4)          array of the coefficients of the equivalent Hermite series, ordered
85: (4)          from lowest to highest degree.
86: (4)          Parameters
87: (4)          -----
88: (4)          pol : array_like
89: (8)          1-D array containing the polynomial coefficients
90: (4)          Returns
91: (4)          -----
92: (4)          c : ndarray
93: (8)          1-D array containing the coefficients of the equivalent Hermite
94: (8)          series.
95: (4)          See Also
96: (4)          -----
97: (4)          herm2poly
98: (4)          Notes
99: (4)          -----
100: (4)          The easy way to do conversions between polynomial basis sets
101: (4)          is to use the convert method of a class instance.
102: (4)          Examples
103: (4)          -----
104: (4)          >>> from numpy.polynomial.hermite import poly2herm
105: (4)          >>> poly2herm(np.arange(4))
106: (4)          array([1. ,  2.75 ,  0.5 ,  0.375])
107: (4)          """
108: (4)          [pol] = pu.as_series([pol])
109: (4)          deg = len(pol) - 1
110: (4)          res = 0
111: (4)          for i in range(deg, -1, -1):
112: (8)              res = hermadd(hermmulx(res), pol[i])
113: (4)          return res
114: (0)          def herm2poly(c):
115: (4)          """
116: (4)          Convert a Hermite series to a polynomial.
117: (4)          Convert an array representing the coefficients of a Hermite series,
118: (4)          ordered from lowest degree to highest, to an array of the coefficients
119: (4)          of the equivalent polynomial (relative to the "standard" basis) ordered
120: (4)          from lowest to highest degree.
121: (4)          Parameters
122: (4)          -----
123: (4)          c : array_like
124: (8)          1-D array containing the Hermite series coefficients, ordered
125: (8)          from lowest order term to highest.
126: (4)          Returns
127: (4)          -----
128: (4)          pol : ndarray

```

```

129: (8)           1-D array containing the coefficients of the equivalent polynomial
130: (8)           (relative to the "standard" basis) ordered from lowest order term
131: (8)           to highest.
132: (4)           See Also
133: (4)           -----
134: (4)           poly2herm
135: (4)           Notes
136: (4)           -----
137: (4)           The easy way to do conversions between polynomial basis sets
138: (4)           is to use the convert method of a class instance.
139: (4)           Examples
140: (4)           -----
141: (4)           >>> from numpy.polynomial.hermite import herm2poly
142: (4)           >>> herm2poly([ 1. , 2.75 , 0.5 , 0.375])
143: (4)           array([0., 1., 2., 3.])
144: (4)           """
145: (4)           from .polynomial import polyadd, polysub, polymulx
146: (4)           [c] = pu.as_series([c])
147: (4)           n = len(c)
148: (4)           if n == 1:
149: (8)             return c
150: (4)           if n == 2:
151: (8)             c[1] *= 2
152: (8)             return c
153: (4)           else:
154: (8)             c0 = c[-2]
155: (8)             c1 = c[-1]
156: (8)             for i in range(n - 1, 1, -1):
157: (12)               tmp = c0
158: (12)               c0 = polysub(c[i - 2], c1*(2*(i - 1)))
159: (12)               c1 = polyadd(tmp, polymulx(c1)*2)
160: (8)             return polyadd(c0, polymulx(c1)*2)
161: (0)           hermdomain = np.array([-1, 1])
162: (0)           hermzero = np.array([0])
163: (0)           hermone = np.array([1])
164: (0)           hermx = np.array([0, 1/2])
165: (0)           def hermline(off, scl):
166: (4)             """
167: (4)             Hermite series whose graph is a straight line.
168: (4)             Parameters
169: (4)             -----
170: (4)             off, scl : scalars
171: (8)               The specified line is given by ``off + scl*x``.
172: (4)             Returns
173: (4)             -----
174: (4)             y : ndarray
175: (8)               This module's representation of the Hermite series for
176: (8)               ``off + scl*x``.
177: (4)             See Also
178: (4)             -----
179: (4)             numpy.polynomial.polynomial.polyline
180: (4)             numpy.polynomial.chebyshev.chebline
181: (4)             numpy.polynomial.legendre.legline
182: (4)             numpy.polynomial.laguerre.lagline
183: (4)             numpy.polynomial.hermite_e.hermeline
184: (4)             Examples
185: (4)             -----
186: (4)             >>> from numpy.polynomial.hermite import hermline, hermval
187: (4)             >>> hermval(0,hermline(3, 2))
188: (4)             3.0
189: (4)             >>> hermval(1,hermline(3, 2))
190: (4)             5.0
191: (4)             """
192: (4)             if scl != 0:
193: (8)               return np.array([off, scl/2])
194: (4)             else:
195: (8)               return np.array([off])
196: (0)             def hermfromroots(roots):
197: (4)               """

```

```

198: (4)                                     Generate a Hermite series with given roots.
199: (4)                                     The function returns the coefficients of the polynomial
200: (4)                                     .. math:: p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),
201: (4)                                     in Hermite form, where the `r_n` are the roots specified in `roots`.
202: (4)                                     If a zero has multiplicity n, then it must appear in `roots` n times.
203: (4)                                     For instance, if 2 is a root of multiplicity three and 3 is a root of
204: (4)                                     multiplicity 2, then `roots` looks something like [2, 2, 2, 3, 3]. The
205: (4)                                     roots can appear in any order.
206: (4)                                     If the returned coefficients are `c`, then
207: (4)                                     .. math:: p(x) = c_0 + c_1 * H_1(x) + ... + c_n * H_n(x)
208: (4)                                     The coefficient of the last term is not generally 1 for monic
209: (4)                                     polynomials in Hermite form.
210: (4)                                     Parameters
211: (4)                                     -----
212: (4)                                     roots : array_like
213: (8)                                     Sequence containing the roots.
214: (4)                                     Returns
215: (4)                                     -----
216: (4)                                     out : ndarray
217: (8)                                     1-D array of coefficients. If all roots are real then `out` is a
218: (8)                                     real array, if some of the roots are complex, then `out` is complex
219: (8)                                     even if all the coefficients in the result are real (see Examples
220: (8)                                     below).
221: (4)                                     See Also
222: (4)                                     -----
223: (4)                                     numpy.polynomial.polynomial.polyfromroots
224: (4)                                     numpy.polynomial.legendre.legfromroots
225: (4)                                     numpy.polynomial.laguerre.lagfromroots
226: (4)                                     numpy.polynomial.chebyshev.chebfromroots
227: (4)                                     numpy.polynomial.hermite_e.hermefromroots
228: (4)                                     Examples
229: (4)                                     -----
230: (4)                                     >>> from numpy.polynomial.hermite import hermfromroots, hermval
231: (4)                                     >>> coef = hermfromroots((-1, 0, 1))
232: (4)                                     >>> hermval((-1, 0, 1), coef)
233: (4)                                     array([0., 0., 0.])
234: (4)                                     >>> coef = hermfromroots((-1j, 1j))
235: (4)                                     >>> hermval((-1j, 1j), coef)
236: (4)                                     array([0.+0.j, 0.+0.j])
237: (4)                                     """
238: (4)                                     return pu._fromroots(hermline, hermmul, roots)
239: (0)                                     def hermadd(c1, c2):
240: (4)                                     """
241: (4)                                     Add one Hermite series to another.
242: (4)                                     Returns the sum of two Hermite series `c1` + `c2`. The arguments
243: (4)                                     are sequences of coefficients ordered from lowest order term to
244: (4)                                     highest, i.e., [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
245: (4)                                     Parameters
246: (4)                                     -----
247: (4)                                     c1, c2 : array_like
248: (8)                                     1-D arrays of Hermite series coefficients ordered from low to
249: (8)                                     high.
250: (4)                                     Returns
251: (4)                                     -----
252: (4)                                     out : ndarray
253: (8)                                     Array representing the Hermite series of their sum.
254: (4)                                     See Also
255: (4)                                     -----
256: (4)                                     hermsub, hermmulx, hermmul, hermdiv, hermpow
257: (4)                                     Notes
258: (4)                                     -----
259: (4)                                     Unlike multiplication, division, etc., the sum of two Hermite series
260: (4)                                     is a Hermite series (without having to "reproject" the result onto
261: (4)                                     the basis set) so addition, just like that of "standard" polynomials,
262: (4)                                     is simply "component-wise."
263: (4)                                     Examples
264: (4)                                     -----
265: (4)                                     >>> from numpy.polynomial.hermite import hermadd
266: (4)                                     >>> hermadd([1, 2, 3], [1, 2, 3, 4])

```

```

267: (4)             array([2., 4., 6., 4.])
268: (4)             """
269: (4)             return pu._add(c1, c2)
270: (0)             def hermsub(c1, c2):
271: (4)             """
272: (4)             Subtract one Hermite series from another.
273: (4)             Returns the difference of two Hermite series `c1` - `c2`. The
274: (4)             sequences of coefficients are from lowest order term to highest, i.e.,
275: (4)             [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
276: (4)             Parameters
277: (4)             -----
278: (4)             c1, c2 : array_like
279: (8)                 1-D arrays of Hermite series coefficients ordered from low to
280: (8)                 high.
281: (4)             Returns
282: (4)             -----
283: (4)             out : ndarray
284: (8)                 Of Hermite series coefficients representing their difference.
285: (4)             See Also
286: (4)             -----
287: (4)             hermadd, hermmulx, hermmul, hermdiv, hermpow
288: (4)             Notes
289: (4)             -----
290: (4)             Unlike multiplication, division, etc., the difference of two Hermite
291: (4)             series is a Hermite series (without having to "reproject" the result
292: (4)             onto the basis set) so subtraction, just like that of "standard"
293: (4)             polynomials, is simply "component-wise."
294: (4)             Examples
295: (4)             -----
296: (4)             >>> from numpy.polynomial.hermite import hermsub
297: (4)             >>> hermsub([1, 2, 3, 4], [1, 2, 3])
298: (4)             array([0., 0., 0., 4.])
299: (4)             """
300: (4)             return pu._sub(c1, c2)
301: (0)             def hermmulx(c):
302: (4)             """Multiply a Hermite series by x.
303: (4)             Multiply the Hermite series `c` by x, where x is the independent
304: (4)             variable.
305: (4)             Parameters
306: (4)             -----
307: (4)             c : array_like
308: (8)                 1-D array of Hermite series coefficients ordered from low to
309: (8)                 high.
310: (4)             Returns
311: (4)             -----
312: (4)             out : ndarray
313: (8)                 Array representing the result of the multiplication.
314: (4)             See Also
315: (4)             -----
316: (4)             hermadd, hermsub, hermmul, hermdiv, hermpow
317: (4)             Notes
318: (4)             -----
319: (4)             The multiplication uses the recursion relationship for Hermite
320: (4)             polynomials in the form
321: (4)             .. math::
322: (8)                 xP_i(x) = (P_{i + 1}(x)/2 + i*P_{i - 1}(x))
323: (4)             Examples
324: (4)             -----
325: (4)             >>> from numpy.polynomial.hermite import hermmulx
326: (4)             >>> hermmulx([1, 2, 3])
327: (4)             array([2., 6.5, 1., 1.5])
328: (4)             """
329: (4)             [c] = pu.as_series([c])
330: (4)             if len(c) == 1 and c[0] == 0:
331: (8)                 return c
332: (4)             prd = np.empty(len(c) + 1, dtype=c.dtype)
333: (4)             prd[0] = c[0]*0
334: (4)             prd[1] = c[0]/2
335: (4)             for i in range(1, len(c)):
```

```

336: (8)             prd[i + 1] = c[i]/2
337: (8)             prd[i - 1] += c[i]*i
338: (4)             return prd
339: (0)             def hermmul(c1, c2):
340: (4)             """
341: (4)             Multiply one Hermite series by another.
342: (4)             Returns the product of two Hermite series `c1` * `c2`. The arguments
343: (4)             are sequences of coefficients, from lowest order "term" to highest,
344: (4)             e.g., [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
345: (4)             Parameters
346: (4)             -----
347: (4)             c1, c2 : array_like
348: (8)             1-D arrays of Hermite series coefficients ordered from low to
349: (8)             high.
350: (4)             Returns
351: (4)             -----
352: (4)             out : ndarray
353: (8)             Of Hermite series coefficients representing their product.
354: (4)             See Also
355: (4)             -----
356: (4)             hermadd, hermsub, hermmulx, hermdiv, hermpow
357: (4)             Notes
358: (4)             -----
359: (4)             In general, the (polynomial) product of two C-series results in terms
360: (4)             that are not in the Hermite polynomial basis set. Thus, to express
361: (4)             the product as a Hermite series, it is necessary to "reproject" the
362: (4)             product onto said basis set, which may produce "unintuitive" (but
363: (4)             correct) results; see Examples section below.
364: (4)             Examples
365: (4)             -----
366: (4)             >>> from numpy.polynomial.hermite import hermmul
367: (4)             >>> hermmul([1, 2, 3], [0, 1, 2])
368: (4)             array([52.,  29.,  52.,   7.,   6.])
369: (4)             """
370: (4)             [c1, c2] = pu.as_series([c1, c2])
371: (4)             if len(c1) > len(c2):
372: (8)                 c = c2
373: (8)                 xs = c1
374: (4)             else:
375: (8)                 c = c1
376: (8)                 xs = c2
377: (4)             if len(c) == 1:
378: (8)                 c0 = c[0]*xs
379: (8)                 c1 = 0
380: (4)             elif len(c) == 2:
381: (8)                 c0 = c[0]*xs
382: (8)                 c1 = c[1]*xs
383: (4)             else:
384: (8)                 nd = len(c)
385: (8)                 c0 = c[-2]*xs
386: (8)                 c1 = c[-1]*xs
387: (8)                 for i in range(3, len(c) + 1):
388: (12)                     tmp = c0
389: (12)                     nd = nd - 1
390: (12)                     c0 = hermsub(c[-i]*xs, c1*(2*(nd - 1)))
391: (12)                     c1 = hermadd(tmp, hermmulx(c1)*2)
392: (4)             return hermadd(c0, hermmulx(c1)*2)
393: (0)             def hermdiv(c1, c2):
394: (4)             """
395: (4)             Divide one Hermite series by another.
396: (4)             Returns the quotient-with-remainder of two Hermite series
397: (4)             `c1` / `c2`. The arguments are sequences of coefficients from lowest
398: (4)             order "term" to highest, e.g., [1,2,3] represents the series
399: (4)             ``P_0 + 2*P_1 + 3*P_2``.
400: (4)             Parameters
401: (4)             -----
402: (4)             c1, c2 : array_like
403: (8)             1-D arrays of Hermite series coefficients ordered from low to
404: (8)             high.

```

```

405: (4)          Returns
406: (4)          -----
407: (4)          [quo, rem] : ndarrays
408: (8)          Of Hermite series coefficients representing the quotient and
409: (8)          remainder.
410: (4)          See Also
411: (4)          -----
412: (4)          hermadd, hermsub, hermmulx, hermmul, hermpow
413: (4)          Notes
414: (4)          -----
415: (4)          In general, the (polynomial) division of one Hermite series by another
416: (4)          results in quotient and remainder terms that are not in the Hermite
417: (4)          polynomial basis set. Thus, to express these results as a Hermite
418: (4)          series, it is necessary to "reproject" the results onto the Hermite
419: (4)          basis set, which may produce "unintuitive" (but correct) results; see
420: (4)          Examples section below.
421: (4)          Examples
422: (4)          -----
423: (4)          >>> from numpy.polynomial.hermite import hermdiv
424: (4)          >>> hermdiv([ 52.,  29.,  52.,  7.,  6.], [0, 1, 2])
425: (4)          (array([1., 2., 3.]), array([0.]))
426: (4)          >>> hermdiv([ 54.,  31.,  52.,  7.,  6.], [0, 1, 2])
427: (4)          (array([1., 2., 3.]), array([2., 2.]))
428: (4)          >>> hermdiv([ 53.,  30.,  52.,  7.,  6.], [0, 1, 2])
429: (4)          (array([1., 2., 3.]), array([1., 1.]))
430: (4)          """
431: (4)          return pu._div(hermmul, c1, c2)
432: (0)          def hermpow(c, pow, maxpower=16):
433: (4)          """Raise a Hermite series to a power.
434: (4)          Returns the Hermite series `c` raised to the power `pow`. The
435: (4)          argument `c` is a sequence of coefficients ordered from low to high.
436: (4)          i.e., [1,2,3] is the series ``P_0 + 2*P_1 + 3*P_2.``
437: (4)          Parameters
438: (4)          -----
439: (4)          c : array_like
440: (8)          1-D array of Hermite series coefficients ordered from low to
441: (8)          high.
442: (4)          pow : integer
443: (8)          Power to which the series will be raised
444: (4)          maxpower : integer, optional
445: (8)          Maximum power allowed. This is mainly to limit growth of the series
446: (8)          to unmanageable size. Default is 16
447: (4)          Returns
448: (4)          -----
449: (4)          coef : ndarray
450: (8)          Hermite series of power.
451: (4)          See Also
452: (4)          -----
453: (4)          hermadd, hermsub, hermmulx, hermmul, hermdiv
454: (4)          Examples
455: (4)          -----
456: (4)          >>> from numpy.polynomial.hermite import hermpow
457: (4)          >>> hermpow([1, 2, 3], 2)
458: (4)          array([81.,  52.,  82.,  12.,   9.])
459: (4)          """
460: (4)          return pu._pow(hermmul, c, pow, maxpower)
461: (0)          def hermder(c, m=1, scl=1, axis=0):
462: (4)          """
463: (4)          Differentiate a Hermite series.
464: (4)          Returns the Hermite series coefficients `c` differentiated `m` times
465: (4)          along `axis`. At each iteration the result is multiplied by `scl` (the
466: (4)          scaling factor is for use in a linear change of variable). The argument
467: (4)          `c` is an array of coefficients from low to high degree along each
468: (4)          axis, e.g., [1,2,3] represents the series ``1*H_0 + 2*H_1 + 3*H_2``
469: (4)          while [[1,2],[1,2]] represents ``1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) +
470: (4)          2*H_0(x)*H_1(y) + 2*H_1(x)*H_1(y)`` if axis=0 is ``x`` and axis=1 is
471: (4)          ``y``.
472: (4)          Parameters
473: (4)          -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

474: (4) c : array_like
475: (8)     Array of Hermite series coefficients. If `c` is multidimensional the
476: (8)     different axis correspond to different variables with the degree in
477: (8)     each axis given by the corresponding index.
478: (4) m : int, optional
479: (8)     Number of derivatives taken, must be non-negative. (Default: 1)
480: (4) scl : scalar, optional
481: (8)     Each differentiation is multiplied by `scl`. The end result is
482: (8)     multiplication by ``scl**m``. This is for use in a linear change of
483: (8)     variable. (Default: 1)
484: (4) axis : int, optional
485: (8)     Axis over which the derivative is taken. (Default: 0).
486: (8)     .. versionadded:: 1.7.0
487: (4) Returns
488: (4) -----
489: (4) der : ndarray
490: (8)     Hermite series of the derivative.
491: (4) See Also
492: (4) -----
493: (4) hermint
494: (4) Notes
495: (4) -----
496: (4) In general, the result of differentiating a Hermite series does not
497: (4) resemble the same operation on a power series. Thus the result of this
498: (4) function may be "unintuitive," albeit correct; see Examples section
499: (4) below.
500: (4) Examples
501: (4) -----
502: (4) >>> from numpy.polynomial.hermite import hermder
503: (4) >>> hermder([ 1. , 0.5, 0.5, 0.5])
504: (4) array([1., 2., 3.])
505: (4) >>> hermder([-0.5, 1./2., 1./8., 1./12., 1./16.], m=2)
506: (4) array([1., 2., 3.])
507: (4) """
508: (4) c = np.array(c, ndmin=1, copy=True)
509: (4) if c.dtype.char in '?bBhHiIlLqQpP':
510: (8)     c = c.astype(np.double)
511: (4) cnt = pu._deprecate_as_int(m, "the order of derivation")
512: (4) iaxis = pu._deprecate_as_int(axis, "the axis")
513: (4) if cnt < 0:
514: (8)     raise ValueError("The order of derivation must be non-negative")
515: (4) iaxis = normalize_axis_index(iaxis, c.ndim)
516: (4) if cnt == 0:
517: (8)     return c
518: (4) c = np.moveaxis(c, iaxis, 0)
519: (4) n = len(c)
520: (4) if cnt >= n:
521: (8)     c = c[:1]*0
522: (4) else:
523: (8)     for i in range(cnt):
524: (12)         n = n - 1
525: (12)         c *= scl
526: (12)         der = np.empty((n,) + c.shape[1:], dtype=c.dtype)
527: (12)         for j in range(n, 0, -1):
528: (16)             der[j - 1] = (2*j)*c[j]
529: (12)         c = der
530: (4)     c = np.moveaxis(c, 0, iaxis)
531: (4)     return c
532: (0) def hermint(c, m=1, k=[], lbnd=0, scl=1, axis=0):
533: (4) """
534: (4)     Integrate a Hermite series.
535: (4)     Returns the Hermite series coefficients `c` integrated `m` times from
536: (4)     `lbnd` along `axis`. At each iteration the resulting series is
537: (4)     **multiplied** by `scl` and an integration constant, `k`, is added.
538: (4)     The scaling factor is for use in a linear change of variable. ("Buyer
539: (4)     beware": note that, depending on what one is doing, one may want `scl`
540: (4)     to be the reciprocal of what one might expect; for more information,
541: (4)     see the Notes section below.) The argument `c` is an array of
542: (4)     coefficients from low to high degree along each axis, e.g., [1,2,3]
```

```

543: (4) represents the series ``H_0 + 2*H_1 + 3*H_2`` while [[1,2],[1,2]]
544: (4) represents ``1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) + 2*H_0(x)*H_1(y) +
545: (4) 2*H_1(x)*H_1(y)`` if axis=0 is ``x`` and axis=1 is ``y``.
546: (4) Parameters
547: (4)
548: (4) c : array_like
549: (8)     Array of Hermite series coefficients. If c is multidimensional the
550: (8)     different axis correspond to different variables with the degree in
551: (8)     each axis given by the corresponding index.
552: (4) m : int, optional
553: (8)     Order of integration, must be positive. (Default: 1)
554: (4) k : [], list, scalar}, optional
555: (8)     Integration constant(s). The value of the first integral at
556: (8)     ``lbnd`` is the first value in the list, the value of the second
557: (8)     integral at ``lbnd`` is the second value, etc. If ``k == []`` (the
558: (8)     default), all constants are set to zero. If ``m == 1``, a single
559: (8)     scalar can be given instead of a list.
560: (4) lbnd : scalar, optional
561: (8)     The lower bound of the integral. (Default: 0)
562: (4) scl : scalar, optional
563: (8)     Following each integration the result is *multiplied* by `scl`
564: (8)     before the integration constant is added. (Default: 1)
565: (4) axis : int, optional
566: (8)     Axis over which the integral is taken. (Default: 0).
567: (8)     .. versionadded:: 1.7.0
568: (4) Returns
569: (4)
570: (4) S : ndarray
571: (8)     Hermite series coefficients of the integral.
572: (4) Raises
573: (4)
574: (4) ValueError
575: (8)     If ``m < 0``, ``len(k) > m``, ``np.ndim(lbnd) != 0``, or
576: (8)     ``np.ndim(scl) != 0``.
577: (4) See Also
578: (4)
579: (4) hermder
580: (4) Notes
581: (4)
582: (4) Note that the result of each integration is *multiplied* by `scl`.
583: (4) Why is this important to note? Say one is making a linear change of
584: (4) variable :math:`u = ax + b` in an integral relative to `x`. Then
585: (4) :math:`dx = du/a`, so one will need to set `scl` equal to
586: (4) :math:`1/a` - perhaps not what one would have first thought.
587: (4) Also note that, in general, the result of integrating a C-series needs
588: (4) to be "reprojected" onto the C-series basis set. Thus, typically,
589: (4) the result of this function is "unintuitive," albeit correct; see
590: (4) Examples section below.
591: (4) Examples
592: (4)
593: (4) >>> from numpy.polynomial.hermite import hermint
594: (4) >>> hermint([1,2,3]) # integrate once, value 0 at 0.
595: (4) array([1. , 0.5, 0.5, 0.5])
596: (4) >>> hermint([1,2,3], m=2) # integrate twice, value & deriv 0 at 0
597: (4) array([-0.5         ,  0.5         ,  0.125       ,  0.08333333,  0.0625      ]) #
may vary
598: (4) >>> hermint([1,2,3], k=1) # integrate once, value 1 at 0.
599: (4) array([2. , 0.5, 0.5, 0.5])
600: (4) >>> hermint([1,2,3], lbnd=-1) # integrate once, value 0 at -1
601: (4) array([-2. ,  0.5,  0.5,  0.5])
602: (4) >>> hermint([1,2,3], m=2, k=[1,2], lbnd=-1)
603: (4) array([ 1.66666667, -0.5         ,  0.125       ,  0.08333333,  0.0625      ]) #
may vary
604: (4)
605: (4) """
606: (4) c = np.array(c, ndmin=1, copy=True)
607: (8) if c.dtype.char in '?bBhHiIlLqQpP':
608: (4)     c = c.astype(np.double)
609: (4) if not np.iterable(k):
609: (8)     k = [k]

```

```

610: (4)             cnt = pu._deprecate_as_int(m, "the order of integration")
611: (4)             iaxis = pu._deprecate_as_int(axis, "the axis")
612: (4)             if cnt < 0:
613: (8)                 raise ValueError("The order of integration must be non-negative")
614: (4)             if len(k) > cnt:
615: (8)                 raise ValueError("Too many integration constants")
616: (4)             if np.ndim(lbnd) != 0:
617: (8)                 raise ValueError("lbnd must be a scalar.")
618: (4)             if np.ndim(scl) != 0:
619: (8)                 raise ValueError("scl must be a scalar.")
620: (4)             iaxis = normalize_axis_index(iaxis, c.ndim)
621: (4)             if cnt == 0:
622: (8)                 return c
623: (4)             c = np.moveaxis(c, iaxis, 0)
624: (4)             k = list(k) + [0] * (cnt - len(k))
625: (4)             for i in range(cnt):
626: (8)                 n = len(c)
627: (8)                 c *= scl
628: (8)                 if n == 1 and np.all(c[0] == 0):
629: (12)                     c[0] += k[i]
630: (8)                 else:
631: (12)                     tmp = np.empty((n + 1,) + c.shape[1:], dtype=c.dtype)
632: (12)                     tmp[0] = c[0]*0
633: (12)                     tmp[1] = c[0]/2
634: (12)                     for j in range(1, n):
635: (16)                         tmp[j + 1] = c[j]/(2*(j + 1))
636: (12)                     tmp[0] += k[i] - hermval(lbnd, tmp)
637: (12)                     c = tmp
638: (4)             c = np.moveaxis(c, 0, iaxis)
639: (4)             return c
640: (0)             def hermval(x, c, tensor=True):
641: (4)                 """
642: (4)                 Evaluate an Hermite series at points x.
643: (4)                 If `c` is of length `n + 1`, this function returns the value:
644: (4)                 .. math:: p(x) = c_0 * H_0(x) + c_1 * H_1(x) + \dots + c_n * H_n(x)
645: (4)                 The parameter `x` is converted to an array only if it is a tuple or a
646: (4)                 list, otherwise it is treated as a scalar. In either case, either `x`
647: (4)                 or its elements must support multiplication and addition both with
648: (4)                 themselves and with the elements of `c`.
649: (4)                 If `c` is a 1-D array, then `p(x)` will have the same shape as `x`. If
650: (4)                 `c` is multidimensional, then the shape of the result depends on the
651: (4)                 value of `tensor`. If `tensor` is true the shape will be c.shape[1:] +
652: (4)                 x.shape. If `tensor` is false the shape will be c.shape[1:].
653: (4)                 Note that scalars have shape ().
654: (4)                 Trailing zeros in the coefficients will be used in the evaluation, so
655: (4)                 they should be avoided if efficiency is a concern.
656: (4)                 Parameters
657: (4)                 -----
658: (4)                 x : array_like, compatible object
659: (8)                     If `x` is a list or tuple, it is converted to an ndarray, otherwise
660: (8)                     it is left unchanged and treated as a scalar. In either case, `x`
661: (8)                     or its elements must support addition and multiplication with
662: (8)                     themselves and with the elements of `c`.
663: (4)                 c : array_like
664: (8)                     Array of coefficients ordered so that the coefficients for terms of
665: (8)                     degree n are contained in c[n]. If `c` is multidimensional the
666: (8)                     remaining indices enumerate multiple polynomials. In the two
667: (8)                     dimensional case the coefficients may be thought of as stored in
668: (8)                     the columns of `c`.
669: (4)                 tensor : boolean, optional
670: (8)                     If True, the shape of the coefficient array is extended with ones
671: (8)                     on the right, one for each dimension of `x`. Scalars have dimension 0
672: (8)                     for this action. The result is that every column of coefficients in
673: (8)                     `c` is evaluated for every element of `x`. If False, `x` is broadcast
674: (8)                     over the columns of `c` for the evaluation. This keyword is useful
675: (8)                     when `c` is multidimensional. The default value is True.
676: (8)                     .. versionadded:: 1.7.0
677: (4)                 Returns
678: (4)                 -----

```

```

679: (4)             values : ndarray, algebra_like
680: (8)             The shape of the return value is described above.
681: (4)             See Also
682: (4)
683: (4)             -----
684: (4)             hermval2d, hermgrid2d, hermval3d, hermgrid3d
685: (4)             Notes
686: (4)             -----
687: (4)             The evaluation uses Clenshaw recursion, aka synthetic division.
688: (4)             Examples
689: (4)             -----
690: (4)             >>> from numpy.polynomial.hermite import hermval
691: (4)             >>> coef = [1,2,3]
692: (4)             >>> hermval(1, coef)
693: (4)             11.0
694: (4)             >>> hermval([[1,2],[3,4]], coef)
695: (11)            array([[ 11.,   51.],
696: (4)                         [115.,  203.]])
697: (4)             """
698: (4)             c = np.array(c, ndmin=1, copy=False)
699: (8)             if c.dtype.char in '?bBhHiIlLqQpP':
700: (4)                 c = c.astype(np.double)
701: (8)             if isinstance(x, (tuple, list)):
702: (4)                 x = np.asarray(x)
703: (8)             if isinstance(x, np.ndarray) and tensor:
704: (4)                 c = c.reshape(c.shape + (1,)*x.ndim)
705: (4)             x2 = x*x
706: (8)             if len(c) == 1:
707: (8)                 c0 = c[0]
708: (4)                 c1 = 0
709: (8)             elif len(c) == 2:
710: (8)                 c0 = c[0]
711: (4)                 c1 = c[1]
712: (8)             else:
713: (8)                 nd = len(c)
714: (8)                 c0 = c[-2]
715: (8)                 c1 = c[-1]
716: (12)                for i in range(3, len(c) + 1):
717: (12)                    tmp = c0
718: (12)                    nd = nd - 1
719: (12)                    c0 = c[-i] - c1*(2*(nd - 1))
720: (4)                    c1 = tmp + c1*x2
721: (0)                return c0 + c1*x2
722: (4)            def hermval2d(x, y, c):
723: (4)                """
724: (4)                Evaluate a 2-D Hermite series at points (x, y).
725: (4)                This function returns the values:
726: (4)                .. math:: p(x,y) = \sum_{i,j} c_{i,j} * H_i(x) * H_j(y)
727: (4)                The parameters `x` and `y` are converted to arrays only if they are
728: (4)                tuples or a lists, otherwise they are treated as a scalars and they
729: (4)                must have the same shape after conversion. In either case, either `x`
730: (4)                and `y` or their elements must support multiplication and addition both
731: (4)                with themselves and with the elements of `c`.
732: (4)                If `c` is a 1-D array a one is implicitly appended to its shape to make
733: (4)                it 2-D. The shape of the result will be c.shape[2:] + x.shape.
734: (4)                Parameters
735: (4)                -----
736: (8)                x, y : array_like, compatible objects
737: (8)                The two dimensional series is evaluated at the points `(x, y)`,
738: (8)                where `x` and `y` must have the same shape. If `x` or `y` is a list
739: (8)                or tuple, it is first converted to an ndarray, otherwise it is left
740: (4)                unchanged and if it isn't an ndarray it is treated as a scalar.
741: (8)                c : array_like
742: (8)                Array of coefficients ordered so that the coefficient of the term
743: (8)                of multi-degree  $i, j$  is contained in `c[i,j]`. If `c` has
744: (8)                dimension greater than two the remaining indices enumerate multiple
745: (4)                sets of coefficients.
746: (4)                Returns
747: (4)                -----
748: (4)                values : ndarray, compatible object

```

```

748: (8)           The values of the two dimensional polynomial at points formed with
749: (8)           pairs of corresponding values from `x` and `y`.
750: (4)           See Also
751: (4)
752: (4)           -----
753: (4)           hermval, hermgrid2d, hermval3d, hermgrid3d
754: (4)           Notes
755: (4)           -----
756: (4)           .. versionadded:: 1.7.0
757: (4)           """
758: (0)           return pu._valnd(hermval, c, x, y)
759: (4)           def hermgrid2d(x, y, c):
760: (4)           """
761: (4)           Evaluate a 2-D Hermite series on the Cartesian product of x and y.
762: (4)           This function returns the values:
763: (4)           .. math:: p(a,b) = \sum_{i,j} c_{i,j} * H_i(a) * H_j(b)
764: (4)           where the points `(a, b)` consist of all pairs formed by taking
765: (4)           `a` from `x` and `b` from `y`. The resulting points form a grid with
766: (4)           `x` in the first dimension and `y` in the second.
767: (4)           The parameters `x` and `y` are converted to arrays only if they are
768: (4)           tuples or a lists, otherwise they are treated as a scalars. In either
769: (4)           case, either `x` and `y` or their elements must support multiplication
770: (4)           and addition both with themselves and with the elements of `c`.
771: (4)           If `c` has fewer than two dimensions, ones are implicitly appended to
772: (4)           its shape to make it 2-D. The shape of the result will be c.shape[2:] +
773: (4)           x.shape.
774: (4)           Parameters
775: (4)           -----
776: (4)           x, y : array_like, compatible objects
777: (8)           The two dimensional series is evaluated at the points in the
778: (8)           Cartesian product of `x` and `y`. If `x` or `y` is a list or
779: (8)           tuple, it is first converted to an ndarray, otherwise it is left
780: (8)           unchanged and, if it isn't an ndarray, it is treated as a scalar.
781: (4)           c : array_like
782: (8)           Array of coefficients ordered so that the coefficients for terms of
783: (8)           degree i,j are contained in ``c[i,j]``. If `c` has dimension
784: (8)           greater than two the remaining indices enumerate multiple sets of
785: (8)           coefficients.
786: (4)           Returns
787: (4)           -----
788: (8)           values : ndarray, compatible object
789: (8)           The values of the two dimensional polynomial at points in the
790: (8)           product of `x` and `y`.
791: (4)           See Also
792: (4)           -----
793: (4)           hermval, hermval2d, hermval3d, hermgrid3d
794: (4)           Notes
795: (4)           -----
796: (4)           .. versionadded:: 1.7.0
797: (4)           """
798: (0)           return pu._gridnd(hermval, c, x, y)
799: (4)           def hermval3d(x, y, z, c):
800: (4)           """
801: (4)           Evaluate a 3-D Hermite series at points (x, y, z).
802: (4)           This function returns the values:
803: (4)           .. math:: p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * H_i(x) * H_j(y) * H_k(z)
804: (4)           The parameters `x`, `y`, and `z` are converted to arrays only if
805: (4)           they are tuples or a lists, otherwise they are treated as a scalars and
806: (4)           they must have the same shape after conversion. In either case, either
807: (4)           `x`, `y`, and `z` or their elements must support multiplication and
808: (4)           addition both with themselves and with the elements of `c`.
809: (4)           If `c` has fewer than 3 dimensions, ones are implicitly appended to its
810: (4)           shape to make it 3-D. The shape of the result will be c.shape[3:] +
811: (4)           x.shape.
812: (4)           Parameters
813: (4)           -----
814: (8)           x, y, z : array_like, compatible object
815: (8)           The three dimensional series is evaluated at the points
816: (8)           `(x, y, z)`, where `x`, `y`, and `z` must have the same shape. If

```

```

816: (8)             any of `x`, `y`, or `z` is a list or tuple, it is first converted
817: (8)             to an ndarray, otherwise it is left unchanged and if it isn't an
818: (8)             ndarray it is treated as a scalar.
819: (4)             c : array_like
820: (8)                 Array of coefficients ordered so that the coefficient of the term of
821: (8)                 multi-degree i,j,k is contained in ``c[i,j,k]``. If `c` has dimension
822: (8)                 greater than 3 the remaining indices enumerate multiple sets of
823: (8)                 coefficients.
824: (4)             Returns
825: (4)
826: (4)             -----
827: (8)                 values : ndarray, compatible object
828: (8)                     The values of the multidimensional polynomial on points formed with
829: (8)                     triples of corresponding values from `x`, `y`, and `z`.
830: (4)             See Also
831: (4)             -----
832: (4)                 hermval, hermval2d, hermgrid2d, hermgrid3d
833: (4)             Notes
834: (4)             -----
835: (4)                 .. versionadded:: 1.7.0
836: (4)                 """
837: (0)             def hermgrid3d(x, y, z, c):
838: (4)                 """
839: (4)                 Evaluate a 3-D Hermite series on the Cartesian product of x, y, and z.
840: (4)                 This function returns the values:
841: (4)                 .. math:: p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * H_i(a) * H_j(b) * H_k(c)
842: (4)                 where the points `(a, b, c)` consist of all triples formed by taking
843: (4)                 `a` from `x`, `b` from `y`, and `c` from `z`. The resulting points form
844: (4)                 a grid with `x` in the first dimension, `y` in the second, and `z` in
845: (4)                 the third.
846: (4)                 The parameters `x`, `y`, and `z` are converted to arrays only if they
847: (4)                 are tuples or a lists, otherwise they are treated as a scalars. In
848: (4)                 either case, either `x`, `y`, and `z` or their elements must support
849: (4)                 multiplication and addition both with themselves and with the elements
850: (4)                 of `c`.
851: (4)                 If `c` has fewer than three dimensions, ones are implicitly appended to
852: (4)                 its shape to make it 3-D. The shape of the result will be c.shape[3:] +
853: (4)                 x.shape + y.shape + z.shape.
854: (4)             Parameters
855: (4)
856: (4)             x, y, z : array_like, compatible objects
857: (8)                 The three dimensional series is evaluated at the points in the
858: (8)                 Cartesian product of `x`, `y`, and `z`. If `x`, `y`, or `z` is a
859: (8)                 list or tuple, it is first converted to an ndarray, otherwise it is
860: (8)                 left unchanged and, if it isn't an ndarray, it is treated as a
861: (8)                 scalar.
862: (4)             c : array_like
863: (8)                 Array of coefficients ordered so that the coefficients for terms of
864: (8)                 degree i,j are contained in ``c[i,j]``. If `c` has dimension
865: (8)                 greater than two the remaining indices enumerate multiple sets of
866: (8)                 coefficients.
867: (4)             Returns
868: (4)
869: (4)             values : ndarray, compatible object
870: (8)                 The values of the two dimensional polynomial at points in the
871: (8)                     product of `x` and `y`.
872: (4)             See Also
873: (4)
874: (4)             -----
875: (4)                 hermval, hermval2d, hermgrid2d, hermval3d
876: (4)             Notes
877: (4)             -----
878: (4)                 .. versionadded:: 1.7.0
879: (4)                 """
880: (0)             def hermvander(x, deg):
881: (4)                 """
882: (4)                 Pseudo-Vandermonde matrix of given degree.
883: (4)                 Returns the pseudo-Vandermonde matrix of degree `deg` and sample points

```

```

884: (4) .. math:: V[..., i] = H_i(x),
885: (4) where `0 <= i <= deg`. The leading indices of `V` index the elements of
886: (4) `x` and the last index is the degree of the Hermite polynomial.
887: (4) If `c` is a 1-D array of coefficients of length `n + 1` and `V` is the
888: (4) array ``V = hermvander(x, n)``, then ``np.dot(V, c)`` and
889: (4) ``hermval(x, c)`` are the same up to roundoff. This equivalence is
890: (4) useful both for least squares fitting and for the evaluation of a large
891: (4) number of Hermite series of the same degree and sample points.
892: (4) Parameters
893: (4) -----
894: (4) x : array_like
895: (8)     Array of points. The dtype is converted to float64 or complex128
896: (8)     depending on whether any of the elements are complex. If `x` is
897: (8)     scalar it is converted to a 1-D array.
898: (4) deg : int
899: (8)     Degree of the resulting matrix.
900: (4) Returns
901: (4) -----
902: (4) vander : ndarray
903: (8)     The pseudo-Vandermonde matrix. The shape of the returned matrix is
904: (8)     ``x.shape + (deg + 1,)``, where The last index is the degree of the
905: (8)     corresponding Hermite polynomial. The dtype will be the same as
906: (8)     the converted `x`.
907: (4) Examples
908: (4) -----
909: (4) >>> from numpy.polynomial.hermite import hermvander
910: (4) >>> x = np.array([-1, 0, 1])
911: (4) >>> hermvander(x, 3)
912: (4) array([[ 1., -2.,  2.,  4.],
913: (11)      [ 1.,  0., -2., -0.],
914: (11)      [ 1.,  2.,  2., -4.]])
915: (4) """
916: (4) ideg = pu._deprecate_as_int(deg, "deg")
917: (4) if ideg < 0:
918: (8)     raise ValueError("deg must be non-negative")
919: (4) x = np.array(x, copy=False, ndmin=1) + 0.0
920: (4) dims = (ideg + 1,) + x.shape
921: (4) dtyp = x.dtype
922: (4) v = np.empty(dims, dtype=dtyp)
923: (4) v[0] = x*0 + 1
924: (4) if ideg > 0:
925: (8)     x2 = x*x
926: (8)     v[1] = x2
927: (8)     for i in range(2, ideg + 1):
928: (12)         v[i] = (v[i-1]*x2 - v[i-2]*(2*(i - 1)))
929: (4) return np.moveaxis(v, 0, -1)
930: (0) def hermvander2d(x, y, deg):
931: (4) """Pseudo-Vandermonde matrix of given degrees.
932: (4) Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
933: (4) points `(x, y)`. The pseudo-Vandermonde matrix is defined by
934: (4) .. math:: V[..., (deg[1] + 1)*i + j] = H_i(x) * H_j(y),
935: (4) where `0 <= i <= deg[0]` and `0 <= j <= deg[1]`. The leading indices of
936: (4) `V` index the points `(x, y)` and the last index encodes the degrees of
937: (4) the Hermite polynomials.
938: (4) If ``V = hermvander2d(x, y, [xdeg, ydeg])``, then the columns of `V`
939: (4) correspond to the elements of a 2-D coefficient array `c` of shape
940: (4) (xdeg + 1, ydeg + 1) in the order
941: (4) .. math:: c_{\{00\}}, c_{\{01\}}, c_{\{02\}} \dots, c_{\{10\}}, c_{\{11\}}, c_{\{12\}} \dots
942: (4) and ``np.dot(V, c.flat)`` and ``hermval2d(x, y, c)`` will be the same
943: (4) up to roundoff. This equivalence is useful both for least squares
944: (4) fitting and for the evaluation of a large number of 2-D Hermite
945: (4) series of the same degrees and sample points.
946: (4) Parameters
947: (4) -----
948: (4) x, y : array_like
949: (8)     Arrays of point coordinates, all of the same shape. The dtypes
950: (8)     will be converted to either float64 or complex128 depending on
951: (8)     whether any of the elements are complex. Scalars are converted to 1-D
952: (8)     arrays.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

953: (4)             deg : list of ints
954: (8)             List of maximum degrees of the form [x_deg, y_deg].
955: (4)             Returns
956: (4)
957: (4)             vander2d : ndarray
958: (8)             The shape of the returned matrix is ``x.shape + (order,)``, where
959: (8)             :math:`\text{order} = (\deg[0]+1)*(deg[1]+1)` . The dtype will be the same
960: (8)             as the converted `x` and `y` .
961: (4)             See Also
962: (4)
963: (4)             hermvander, hermvander3d, hermval2d, hermval3d
964: (4)             Notes
965: (4)
966: (4)             .. versionadded:: 1.7.0
967: (4)
968: (4)             return pu._vander_nd_flat((hermvander, hermvander), (x, y), deg)
969: (0)             def hermvander3d(x, y, z, deg):
970: (4)               """Pseudo-Vandermonde matrix of given degrees.
971: (4)               Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
972: (4)               points `(x, y, z)` . If `l, m, n` are the given degrees in `x, y, z` ,
973: (4)               then The pseudo-Vandermonde matrix is defined by
974: (4)               .. math:: V[..., (m+1)(n+1)i + (n+1)j + k] = H_i(x)*H_j(y)*H_k(z),
975: (4)               where `0 <= i <= l` , `0 <= j <= m` , and `0 <= j <= n` . The leading
976: (4)               indices of `V` index the points `(x, y, z)` and the last index encodes
977: (4)               the degrees of the Hermite polynomials.
978: (4)               If ``V = hermvander3d(x, y, z, [xdeg, ydeg, zdeg])`` , then the columns
979: (4)               of `V` correspond to the elements of a 3-D coefficient array `c` of
980: (4)               shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order
981: (4)               .. math:: c_{000}, c_{001}, c_{002}, \dots, c_{010}, c_{011}, c_{012}, \dots
982: (4)               and ``np.dot(V, c.flat)`` and ``hermval3d(x, y, z, c)`` will be the
983: (4)               same up to roundoff. This equivalence is useful both for least squares
984: (4)               fitting and for the evaluation of a large number of 3-D Hermite
985: (4)               series of the same degrees and sample points.
986: (4)             Parameters
987: (4)
988: (4)             x, y, z : array_like
989: (8)             Arrays of point coordinates, all of the same shape. The dtypes will
990: (8)             be converted to either float64 or complex128 depending on whether
991: (8)             any of the elements are complex. Scalars are converted to 1-D
992: (8)             arrays.
993: (4)             deg : list of ints
994: (8)             List of maximum degrees of the form [x_deg, y_deg, z_deg].
995: (4)             Returns
996: (4)
997: (4)             vander3d : ndarray
998: (8)             The shape of the returned matrix is ``x.shape + (order,)``, where
999: (8)             :math:`\text{order} = (\deg[0]+1)*(deg[1]+1)*(deg[2]+1)` . The dtype will
1000: (8)             be the same as the converted `x` , `y` , and `z` .
1001: (4)             See Also
1002: (4)
1003: (4)             hermvander, hermvander3d, hermval2d, hermval3d
1004: (4)             Notes
1005: (4)
1006: (4)             .. versionadded:: 1.7.0
1007: (4)
1008: (4)             return pu._vander_nd_flat((hermvander, hermvander, hermvander), (x, y, z),
1009: (0)             deg)
1010: (4)
1011: (4)             def hermfit(x, y, deg, rcond=None, full=False, w=None):
1012: (4)               """
1013: (4)               Least squares fit of Hermite series to data.
1014: (4)               Return the coefficients of a Hermite series of degree `deg` that is the
1015: (4)               least squares fit to the data values `y` given at points `x` . If `y` is
1016: (4)               1-D the returned coefficients will also be 1-D. If `y` is 2-D multiple
1017: (4)               fits are done, one for each column of `y` , and the resulting
1018: (4)               coefficients are stored in the corresponding columns of a 2-D return.
1019: (4)               The fitted polynomial(s) are in the form
1020: (4)               .. math:: p(x) = c_0 + c_1 * H_1(x) + \dots + c_n * H_n(x),
1020: (4)               where `n` is `deg` .
1020: (4)             Parameters

```

```

1021: (4)
1022: (4)
1023: (8)
1024: (4)
1025: (8)
1026: (8)
1027: (8)
1028: (4)
1029: (8)
1030: (8)
1031: (8)
1032: (8)
1033: (4)
1034: (8)
1035: (8)
1036: (8)
1037: (8)
1038: (4)
1039: (8)
1040: (8)
1041: (8)
1042: (4)
1043: (8)
1044: (8)
1045: (8)
1046: (8)
1047: (8)
1048: (4)
1049: (4)
1050: (4)
1051: (8)
1052: (8)
1053: (8)
1054: (4)
1055: (8)
1056: (8)
1057: (8)
1058: (8)
1059: (8)
1060: (8)
1061: (4)
1062: (4)
1063: (4)
1064: (8)
1065: (8)
1066: (8)
1067: (8)
1068: (8)
1069: (4)
1070: (4)
1071: (4)
1072: (4)
1073: (4)
1074: (4)
1075: (4)
1076: (4)
1077: (4)
1078: (4)
1079: (4)
1080: (4)
1081: (4)
1082: (4)
1083: (4)
1084: (4)
1085: (4)
1086: (4)
1087: (4)
1088: (4)
1089: (4)

-----  

x : array_like, shape (M,)  

    x-coordinates of the M sample points ``(x[i], y[i])``.  

y : array_like, shape (M,) or (M, K)  

    y-coordinates of the sample points. Several data sets of sample  

    points sharing the same x-coordinates can be fitted at once by  

    passing in a 2D-array that contains one dataset per column.  

deg : int or 1-D array_like  

    Degree(s) of the fitting polynomials. If `deg` is a single integer  

    all terms up to and including the `deg`'th term are included in the  

    fit. For NumPy versions >= 1.11.0 a list of integers specifying the  

    degrees of the terms to include may be used instead.  

rcond : float, optional  

    Relative condition number of the fit. Singular values smaller than  

    this relative to the largest singular value will be ignored. The  

    default value is len(x)*eps, where eps is the relative precision of  

    the float type, about 2e-16 in most cases.  

full : bool, optional  

    Switch determining nature of return value. When it is False (the  

    default) just the coefficients are returned, when True diagnostic  

    information from the singular value decomposition is also returned.  

w : array_like, shape (`M`), optional  

    Weights. If not None, the weight ``w[i]`` applies to the unsquared  

    residual ``y[i] - y_hat[i]`` at ``x[i]``. Ideally the weights are  

    chosen so that the errors of the products ``w[i]*y[i]`` all have the  

    same variance. When using inverse-variance weighting, use  

    ``w[i] = 1/sigma(y[i])``. The default value is None.  

Returns  

-----  

coef : ndarray, shape (M,) or (M, K)  

    Hermite coefficients ordered from low to high. If `y` was 2-D,  

    the coefficients for the data in column k of `y` are in column  

    `k`.  

[residuals, rank, singular_values, rcond] : list  

    These values are only returned if ``full == True``  

    - residuals -- sum of squared residuals of the least squares fit  

    - rank -- the numerical rank of the scaled Vandermonde matrix  

    - singular_values -- singular values of the scaled Vandermonde matrix  

    - rcond -- value of `rcond`.  

    For more details, see `numpy.linalg.lstsq`.  

Warns  

-----  

RankWarning  

    The rank of the coefficient matrix in the least-squares fit is  

    deficient. The warning is only raised if ``full == False``. The  

    warnings can be turned off by  

    >>> import warnings  

    >>> warnings.simplefilter('ignore', np.RankWarning)  

See Also  

-----  

numpy.polynomial.chebyshev.chebfit  

numpy.polynomial.legendre.legfit  

numpy.polynomial.laguerre.lagfit  

numpy.polynomial.polynomial.polyfit  

numpy.polynomial.hermite_e.hermefit  

hermval : Evaluates a Hermite series.  

hermvander : Vandermonde matrix of Hermite series.  

hermweight : Hermite weight function  

numpy.linalg.lstsq : Computes a least-squares fit from the matrix.  

scipy.interpolate.UnivariateSpline : Computes spline fits.  

Notes  

-----  

The solution is the coefficients of the Hermite series `p` that  

minimizes the sum of the weighted squared errors  

.. math:: E = \sum_j w_j^2 * |y_j - p(x_j)|^2,  

where the :math:`w_j` are the weights. This problem is solved by  

setting up the (typically) overdetermined matrix equation  

.. math:: V(x) * c = w * y,  

where `V` is the weighted pseudo Vandermonde matrix of `x`, `c` are the

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1090: (4) coefficients to be solved for, `w` are the weights, `y` are the
1091: (4) observed values. This equation is then solved using the singular value
1092: (4) decomposition of `V`.
1093: (4) If some of the singular values of `V` are so small that they are
1094: (4) neglected, then a `RankWarning` will be issued. This means that the
1095: (4) coefficient values may be poorly determined. Using a lower order fit
1096: (4) will usually get rid of the warning. The `rcond` parameter can also be
1097: (4) set to a value smaller than its default, but the resulting fit may be
1098: (4) spurious and have large contributions from roundoff error.
1099: (4) Fits using Hermite series are probably most useful when the data can be
1100: (4) approximated by ``sqrt(w(x)) * p(x)``, where `w(x)` is the Hermite
1101: (4) weight. In that case the weight ``sqrt(w(x[i]))`` should be used
1102: (4) together with data values ``y[i]/sqrt(w(x[i]))``. The weight function is
1103: (4) available as `hermweight`.
1104: (4) References
1105: (4) -----
1106: (4) .. [1] Wikipedia, "Curve fitting",
1107: (11) https://en.wikipedia.org/wiki/Curve_fitting
1108: (4) Examples
1109: (4) -----
1110: (4) >>> from numpy.polynomial.hermite import hermfit, hermval
1111: (4) >>> x = np.linspace(-10, 10)
1112: (4) >>> err = np.random.randn(len(x))/10
1113: (4) >>> y = hermval(x, [1, 2, 3]) + err
1114: (4) >>> hermfit(x, y, 2)
1115: (4) array([1.0218, 1.9986, 2.9999]) # may vary
1116: (4) """
1117: (4) return pu._fit(hermvander, x, y, deg, rcond, full, w)
def hermcompanion(c):
    """Return the scaled companion matrix of c.
    The basis polynomials are scaled so that the companion matrix is
    symmetric when `c` is an Hermite basis polynomial. This provides
    better eigenvalue estimates than the unscaled case and for basis
    polynomials the eigenvalues are guaranteed to be real if
    `numpy.linalg.eigvalsh` is used to obtain them.
    Parameters
    -----
    c : array_like
        1-D array of Hermite series coefficients ordered from low to high
        degree.
    Returns
    -----
    mat : ndarray
        Scaled companion matrix of dimensions (deg, deg).
    Notes
    -----
    .. versionadded:: 1.7.0
    """
    [c] = pu.as_series([c])
    if len(c) < 2:
        raise ValueError('Series must have maximum degree of at least 1.')
    if len(c) == 2:
        return np.array([[-.5*c[0]/c[1]]])
    n = len(c) - 1
    mat = np.zeros((n, n), dtype=c.dtype)
    scl = np.hstack((1., 1./np.sqrt(2.*np.arange(n - 1, 0, -1))))
    scl = np.multiply.accumulate(scl)[::-1]
    top = mat.reshape(-1)[1::n+1]
    bot = mat.reshape(-1)[n::n+1]
    top[...] = np.sqrt(.5*np.arange(1, n))
    bot[...] = top
    mat[:, -1] -= scl*c[:-1]/(2.0*c[-1])
    return mat
def hermroots(c):
    """
    Compute the roots of a Hermite series.
    Return the roots (a.k.a. "zeros") of the polynomial
    .. math:: p(x) = \sum_i c[i] * H_i(x).
    Parameters

```

```

1159: (4)      -----
1160: (4)      c : 1-D array_like
1161: (8)      1-D array of coefficients.
1162: (4)      Returns
1163: (4)      -----
1164: (4)      out : ndarray
1165: (8)      Array of the roots of the series. If all the roots are real,
1166: (8)      then `out` is also real, otherwise it is complex.
1167: (4)      See Also
1168: (4)      -----
1169: (4)      numpy.polynomial.polynomial.polyroots
1170: (4)      numpy.polynomial.legendre.legroots
1171: (4)      numpy.polynomial.laguerre.lagroots
1172: (4)      numpy.polynomial.chebyshev.chebroots
1173: (4)      numpy.polynomial.hermite_e.hermroots
1174: (4)      Notes
1175: (4)      -----
1176: (4)      The root estimates are obtained as the eigenvalues of the companion
1177: (4)      matrix, Roots far from the origin of the complex plane may have large
1178: (4)      errors due to the numerical instability of the series for such
1179: (4)      values. Roots with multiplicity greater than 1 will also show larger
1180: (4)      errors as the value of the series near such points is relatively
1181: (4)      insensitive to errors in the roots. Isolated roots near the origin can
1182: (4)      be improved by a few iterations of Newton's method.
1183: (4)      The Hermite series basis polynomials aren't powers of `x` so the
1184: (4)      results of this function may seem unintuitive.
1185: (4)      Examples
1186: (4)      -----
1187: (4)      >>> from numpy.polynomial.hermite import hermroots, hermfromroots
1188: (4)      >>> coef = hermfromroots([-1, 0, 1])
1189: (4)      >>> coef
1190: (4)      array([0. ,  0.25,  0. ,  0.125])
1191: (4)      >>> hermroots(coef)
1192: (4)      array([-1.00000000e+00, -1.38777878e-17,  1.00000000e+00])
1193: (4)      """
1194: (4)      [c] = pu.as_series([c])
1195: (4)      if len(c) <= 1:
1196: (8)          return np.array([], dtype=c.dtype)
1197: (4)      if len(c) == 2:
1198: (8)          return np.array([-0.5*c[0]/c[1]])
1199: (4)      m = hermcompanion(c)[::-1,::-1]
1200: (4)      r = la.eigvals(m)
1201: (4)      r.sort()
1202: (4)      return r
1203: (0)      def _normed_hermite_n(x, n):
1204: (4)      """
1205: (4)          Evaluate a normalized Hermite polynomial.
1206: (4)          Compute the value of the normalized Hermite polynomial of degree ``n``
1207: (4)          at the points ``x``.
1208: (4)          Parameters
1209: (4)          -----
1210: (4)          x : ndarray of double.
1211: (8)              Points at which to evaluate the function
1212: (4)          n : int
1213: (8)              Degree of the normalized Hermite function to be evaluated.
1214: (4)          Returns
1215: (4)          -----
1216: (4)          values : ndarray
1217: (8)              The shape of the return value is described above.
1218: (4)          Notes
1219: (4)          -----
1220: (4)          .. versionadded:: 1.10.0
1221: (4)          This function is needed for finding the Gauss points and integration
1222: (4)          weights for high degrees. The values of the standard Hermite functions
1223: (4)          overflow when n >= 207.
1224: (4)          """
1225: (4)          if n == 0:
1226: (8)              return np.full(x.shape, 1/np.sqrt(np.sqrt(np.pi)))
1227: (4)          c0 = 0.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1228: (4)         c1 = 1./np.sqrt(np.sqrt(np.pi))
1229: (4)         nd = float(n)
1230: (4)         for i in range(n - 1):
1231: (8)             tmp = c0
1232: (8)             c0 = -c1*np.sqrt((nd - 1.)/nd)
1233: (8)             c1 = tmp + c1*x*np.sqrt(2./nd)
1234: (8)             nd = nd - 1.0
1235: (4)         return c0 + c1*x*np.sqrt(2)
1236: (0)     def hermgauss(deg):
1237: (4)         """
1238: (4)             Gauss-Hermite quadrature.
1239: (4)             Computes the sample points and weights for Gauss-Hermite quadrature.
1240: (4)             These sample points and weights will correctly integrate polynomials of
1241: (4)             degree :math:`2*deg - 1` or less over the interval :math:`[-\infty, \infty]`
1242: (4)             with the weight function :math:`f(x) = \exp(-x^2)`.

1243: (4)             Parameters
1244: (4)             -----
1245: (4)             deg : int
1246: (8)                 Number of sample points and weights. It must be >= 1.
1247: (4)             Returns
1248: (4)             -----
1249: (4)             x : ndarray
1250: (8)                 1-D ndarray containing the sample points.
1251: (4)             y : ndarray
1252: (8)                 1-D ndarray containing the weights.
1253: (4)             Notes
1254: (4)             -----
1255: (4)             .. versionadded:: 1.7.0
1256: (4)             The results have only been tested up to degree 100, higher degrees may
1257: (4)             be problematic. The weights are determined by using the fact that
1258: (4)             .. math:: w_k = c / (H'_n(x_k) * H_{n-1}(x_k))
1259: (4)             where :math:`c` is a constant independent of :math:`k` and :math:`x_k` is the k'th root of :math:`H_n`, and then scaling the results to get
1260: (4)             the right value when integrating 1.
1261: (4)
1262: (4)
1263: (4)             ideg = pu._deprecate_as_int(deg, "deg")
1264: (4)             if ideg <= 0:
1265: (8)                 raise ValueError("deg must be a positive integer")
1266: (4)             c = np.array([0]*deg + [1], dtype=np.float64)
1267: (4)             m = hermcompanion(c)
1268: (4)             x = la.eigvalsh(m)
1269: (4)             dy = _normed_hermite_n(x, ideg)
1270: (4)             df = _normed_hermite_n(x, ideg - 1) * np.sqrt(2*ideg)
1271: (4)             x -= dy/df
1272: (4)             fm = _normed_hermite_n(x, ideg - 1)
1273: (4)             fm /= np.abs(fm).max()
1274: (4)             w = 1/(fm * fm)
1275: (4)             w = (w + w[::-1])/2
1276: (4)             x = (x - x[::-1])/2
1277: (4)             w *= np.sqrt(np.pi) / w.sum()
1278: (4)             return x, w
1279: (0)     def hermweight(x):
1280: (4)         """
1281: (4)             Weight function of the Hermite polynomials.
1282: (4)             The weight function is :math:`\exp(-x^2)` and the interval of
1283: (4)             integration is :math:`[-\infty, \infty]`. the Hermite polynomials are
1284: (4)             orthogonal, but not normalized, with respect to this weight function.
1285: (4)             Parameters
1286: (4)             -----
1287: (4)             x : array_like
1288: (7)                 Values at which the weight function will be computed.
1289: (4)             Returns
1290: (4)             -----
1291: (4)             w : ndarray
1292: (7)                 The weight function at `x`.
1293: (4)             Notes
1294: (4)             -----
1295: (4)             .. versionadded:: 1.7.0
1296: (4)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1297: (4)             w = np.exp(-x**2)
1298: (4)             return w
1299: (0)             class Hermite(ABCPolyBase):
1300: (4)                 """An Hermite series class.
1301: (4)                 The Hermite class provides the standard Python numerical methods
1302: (4)                 '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the
1303: (4)                 attributes and methods listed in the `ABCPolyBase` documentation.
1304: (4)                 Parameters
1305: (4)                 -----
1306: (4)                 coef : array_like
1307: (8)                   Hermite coefficients in order of increasing degree, i.e,
1308: (8)                   ``(1, 2, 3)`` gives ``1*H_0(x) + 2*H_1(X) + 3*H_2(x)``.
1309: (4)                 domain : (2,) array_like, optional
1310: (8)                   Domain to use. The interval ``[domain[0], domain[1]]`` is mapped
1311: (8)                   to the interval ``[window[0], window[1]]`` by shifting and scaling.
1312: (8)                   The default value is [-1, 1].
1313: (4)                 window : (2,) array_like, optional
1314: (8)                   Window, see `domain` for its use. The default value is [-1, 1].
1315: (8)                   .. versionadded:: 1.6.0
1316: (4)                 symbol : str, optional
1317: (8)                   Symbol used to represent the independent variable in string
1318: (8)                   representations of the polynomial expression, e.g. for printing.
1319: (8)                   The symbol must be a valid Python identifier. Default value is 'x'.
1320: (8)                   .. versionadded:: 1.24
1321: (4)                 """
1322: (4)                 _add = staticmethod(hermadd)
1323: (4)                 _sub = staticmethod(hermsub)
1324: (4)                 _mul = staticmethod(hermmul)
1325: (4)                 _div = staticmethod(hermdiv)
1326: (4)                 _pow = staticmethod(hermpow)
1327: (4)                 _val = staticmethod(hermval)
1328: (4)                 _int = staticmethod(hermint)
1329: (4)                 _der = staticmethod(hermder)
1330: (4)                 _fit = staticmethod(hermfit)
1331: (4)                 _line = staticmethod(hermline)
1332: (4)                 _roots = staticmethod(hermroots)
1333: (4)                 _fromroots = staticmethod(hermfromroots)
1334: (4)                 domain = np.array(hermdomain)
1335: (4)                 window = np.array(hermdomain)
1336: (4)                 basis_name = 'H'

```

---

## File 289 - hermite\_e.py:

```

1: (0)             """
2: (0)             =====
3: (0)             HermiteE Series, "Probabilists" (:mod:`numpy.polynomial.hermite_e`)
4: (0)             =====
5: (0)             This module provides a number of objects (mostly functions) useful for
6: (0)             dealing with Hermite_e series, including a `HermiteE` class that
7: (0)             encapsulates the usual arithmetic operations. (General information
8: (0)             on how this module represents and works with such polynomials is in the
9: (0)             docstring for its "parent" sub-package, `numpy.polynomial`).
10: (0)             Classes
11: (0)             -----
12: (0)             .. autosummary::
13: (3)                 :toctree: generated/
14: (3)                     HermiteE
15: (0)             Constants
16: (0)             -----
17: (0)             .. autosummary::
18: (3)                 :toctree: generated/
19: (3)                     hermedomain
20: (3)                     hermezzero
21: (3)                     hermeone
22: (3)                     hermex
23: (0)             Arithmetic
24: (0)             -----

```

```

25: (0) .. autosummary::
26: (3)   :toctree: generated/
27: (3)     hermeadd
28: (3)     hermesub
29: (3)     hermemulx
30: (3)     hermemul
31: (3)     hermediv
32: (3)     hermepow
33: (3)     hermeval
34: (3)     hermeval2d
35: (3)     hermeval3d
36: (3)     hermegrid2d
37: (3)       hermegrid3d
38: (0) Calculus
39: (0) -----
40: (0) .. autosummary::
41: (3)   :toctree: generated/
42: (3)     hermeder
43: (3)     hermeint
44: (0) Misc Functions
45: (0) -----
46: (0) .. autosummary::
47: (3)   :toctree: generated/
48: (3)     hermefromroots
49: (3)     hermroots
50: (3)     hermevander
51: (3)     hermevander2d
52: (3)     hermevander3d
53: (3)     hermegauss
54: (3)     hermeweight
55: (3)     hermecompanion
56: (3)     hermefit
57: (3)     hermetrim
58: (3)     hermeline
59: (3)     herme2poly
60: (3)       poly2herme
61: (0) See also
62: (0) -----
63: (0) `numpy.polynomial`_
64: (0) """
65: (0) import numpy as np
66: (0) import numpy.linalg as la
67: (0) from numpy.core.multiarray import normalize_axis_index
68: (0) from . import polyutils as pu
69: (0) from ._polybase import ABCPolyBase
70: (0) __all__ = [
71: (4)   'hermezzero', 'hermeone', 'hermex', 'hermedomain', 'hermeline',
72: (4)   'hermeadd', 'hermesub', 'hermemulx', 'hermemul', 'hermediv',
73: (4)   'hermepow', 'hermeval', 'hermeder', 'hermeint', 'herme2poly',
74: (4)   'poly2herme', 'hermefromroots', 'hermevander', 'hermefit', 'hermetrim',
75: (4)   'hermroots', 'HermiteE', 'hermeval2d', 'hermeval3d', 'hermegrid2d',
76: (4)   'hermegrid3d', 'hermevander2d', 'hermevander3d', 'hermecompanion',
77: (4)   'hermegauss', 'hermeweight']
78: (0) hermetrim = pu.trimcoef
79: (0) def poly2herme(pol):
80: (4)   """
81: (4)     poly2herme(pol)
82: (4)     Convert a polynomial to a Hermite series.
83: (4)     Convert an array representing the coefficients of a polynomial (relative
84: (4)       to the "standard" basis) ordered from lowest degree to highest, to an
85: (4)       array of the coefficients of the equivalent Hermite series, ordered
86: (4)       from lowest to highest degree.
87: (4)     Parameters
88: (4)     -----
89: (4)     pol : array_like
90: (8)       1-D array containing the polynomial coefficients
91: (4)     Returns
92: (4)     -----
93: (4)     c : ndarray

```

```

94: (8)           1-D array containing the coefficients of the equivalent Hermite
95: (8)           series.
96: (4)           See Also
97: (4)           -----
98: (4)           herme2poly
99: (4)           Notes
100: (4)          -----
101: (4)          The easy way to do conversions between polynomial basis sets
102: (4)          is to use the convert method of a class instance.
103: (4)          Examples
104: (4)          -----
105: (4)          >>> from numpy.polynomial.hermite_e import poly2herme
106: (4)          >>> poly2herme(np.arange(4))
107: (4)          array([ 2., 10., 2., 3.])
108: (4)          """
109: (4)          [pol] = pu.as_series([pol])
110: (4)          deg = len(pol) - 1
111: (4)          res = 0
112: (4)          for i in range(deg, -1, -1):
113: (8)              res = hermeadd(hermemulx(res), pol[i])
114: (4)          return res
115: (0)          def herme2poly(c):
116: (4)          """
117: (4)          Convert a Hermite series to a polynomial.
118: (4)          Convert an array representing the coefficients of a Hermite series,
119: (4)          ordered from lowest degree to highest, to an array of the coefficients
120: (4)          of the equivalent polynomial (relative to the "standard" basis) ordered
121: (4)          from lowest to highest degree.
122: (4)          Parameters
123: (4)          -----
124: (4)          c : array_like
125: (8)              1-D array containing the Hermite series coefficients, ordered
126: (8)              from lowest order term to highest.
127: (4)          Returns
128: (4)          -----
129: (4)          pol : ndarray
130: (8)              1-D array containing the coefficients of the equivalent polynomial
131: (8)              (relative to the "standard" basis) ordered from lowest order term
132: (8)              to highest.
133: (4)          See Also
134: (4)          -----
135: (4)          poly2herme
136: (4)          Notes
137: (4)          -----
138: (4)          The easy way to do conversions between polynomial basis sets
139: (4)          is to use the convert method of a class instance.
140: (4)          Examples
141: (4)          -----
142: (4)          >>> from numpy.polynomial.hermite_e import herme2poly
143: (4)          >>> herme2poly([ 2., 10., 2., 3.])
144: (4)          array([0., 1., 2., 3.])
145: (4)          """
146: (4)          from .polynomial import polyadd, polysub, polymulx
147: (4)          [c] = pu.as_series([c])
148: (4)          n = len(c)
149: (4)          if n == 1:
150: (8)              return c
151: (4)          if n == 2:
152: (8)              return c
153: (4)          else:
154: (8)              c0 = c[-2]
155: (8)              c1 = c[-1]
156: (8)              for i in range(n - 1, 1, -1):
157: (12)                  tmp = c0
158: (12)                  c0 = polysub(c[i - 2], c1*(i - 1))
159: (12)                  c1 = polyadd(tmp, polymulx(c1))
160: (8)                  return polyadd(c0, polymulx(c1))
161: (0)          hermedomain = np.array([-1, 1])
162: (0)          hermezzero = np.array([0])

```

```

163: (0) hermeone = np.array([1])
164: (0) hermex = np.array([0, 1])
165: (0) def hermeline(off, scl):
166: (4)     """
167: (4)         Hermite series whose graph is a straight line.
168: (4)         Parameters
169: (4)         -----
170: (4)         off, scl : scalars
171: (8)             The specified line is given by ``off + scl*x``.
172: (4)         Returns
173: (4)         -----
174: (4)         y : ndarray
175: (8)             This module's representation of the Hermite series for
176: (8)             ``off + scl*x``.
177: (4)         See Also
178: (4)         -----
179: (4)         numpy.polynomial.polynomial.polyline
180: (4)         numpy.polynomial.chebyshev.chebline
181: (4)         numpy.polynomial.legendre.legline
182: (4)         numpy.polynomial.laguerre.lagline
183: (4)         numpy.polynomial.hermite.hermeline
184: (4)         Examples
185: (4)         -----
186: (4)         >>> from numpy.polynomial.hermite_e import hermeline
187: (4)         >>> from numpy.polynomial.hermite_e import hermeline, hermeval
188: (4)         >>> hermeline(0,hermeline(3, 2))
189: (4)         3.0
190: (4)         >>> hermeval(1,hermeline(3, 2))
191: (4)         5.0
192: (4)         """
193: (4)         if scl != 0:
194: (8)             return np.array([off, scl])
195: (4)         else:
196: (8)             return np.array([off])
197: (0) def hermefromroots(roots):
198: (4)     """
199: (4)         Generate a HermiteE series with given roots.
200: (4)         The function returns the coefficients of the polynomial
201: (4)         .. math:: p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),
202: (4)         in HermiteE form, where the `r_n` are the roots specified in `roots`.
203: (4)         If a zero has multiplicity n, then it must appear in `roots` n times.
204: (4)         For instance, if 2 is a root of multiplicity three and 3 is a root of
205: (4)         multiplicity 2, then `roots` looks something like [2, 2, 2, 3, 3]. The
206: (4)         roots can appear in any order.
207: (4)         If the returned coefficients are `c`, then
208: (4)         .. math:: p(x) = c_0 + c_1 * He_1(x) + ... + c_n * He_n(x)
209: (4)         The coefficient of the last term is not generally 1 for monic
210: (4)         polynomials in HermiteE form.
211: (4)         Parameters
212: (4)         -----
213: (4)         roots : array_like
214: (8)             Sequence containing the roots.
215: (4)         Returns
216: (4)         -----
217: (4)         out : ndarray
218: (8)             1-D array of coefficients. If all roots are real then `out` is a
219: (8)             real array, if some of the roots are complex, then `out` is complex
220: (8)             even if all the coefficients in the result are real (see Examples
221: (8)             below).
222: (4)         See Also
223: (4)         -----
224: (4)         numpy.polynomial.polynomial.polyfromroots
225: (4)         numpy.polynomial.legendre.legfromroots
226: (4)         numpy.polynomial.laguerre.lagfromroots
227: (4)         numpy.polynomial.hermite.hermfromroots
228: (4)         numpy.polynomial.chebyshev.chebfromroots
229: (4)         Examples
229: (4)         -----
230: (4)         >>> from numpy.polynomial.hermite_e import hermefromroots, hermeval

```

```

232: (4)
233: (4)
234: (4)
235: (4)
236: (4)
237: (4)
238: (4)
239: (4)
240: (0)
241: (4)
242: (4)
243: (4)
244: (4)
245: (4)
246: (4)
247: (4)
248: (4)
249: (8)
250: (8)
251: (4)
252: (4)
253: (4)
254: (8)
255: (4)
256: (4)
257: (4)
258: (4)
259: (4)
260: (4)
261: (4)
262: (4)
263: (4)
264: (4)
265: (4)
266: (4)
267: (4)
268: (4)
269: (4)
270: (4)
271: (0)
272: (4)
273: (4)
274: (4)
275: (4)
276: (4)
277: (4)
278: (4)
279: (4)
280: (8)
281: (8)
282: (4)
283: (4)
284: (4)
285: (8)
286: (4)
287: (4)
288: (4)
289: (4)
290: (4)
291: (4)
292: (4)
293: (4)
294: (4)
295: (4)
296: (4)
297: (4)
298: (4)
299: (4)
300: (4)

        >>> coef = hermefromroots((-1, 0, 1))
        >>> hermeval((-1, 0, 1), coef)
        array([0., 0., 0.])
        >>> coef = hermefromroots((-1j, 1j))
        >>> hermeval((-1j, 1j), coef)
        array([0.+0.j, 0.+0.j])
        """
        return pu._fromroots(hermeline, hermemul, roots)
def hermeadd(c1, c2):
    """
    Add one Hermite series to another.
    Returns the sum of two Hermite series `c1` + `c2`. The arguments
    are sequences of coefficients ordered from lowest order term to
    highest, i.e., [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
    Parameters
    -----
    c1, c2 : array_like
        1-D arrays of Hermite series coefficients ordered from low to
        high.
    Returns
    -----
    out : ndarray
        Array representing the Hermite series of their sum.
    See Also
    -----
    hermesub, hermemulx, hermemul, hermediv, hermepow
    Notes
    -----
    Unlike multiplication, division, etc., the sum of two Hermite series
    is a Hermite series (without having to "reproject" the result onto
    the basis set) so addition, just like that of "standard" polynomials,
    is simply "component-wise."
    Examples
    -----
    >>> from numpy.polynomial.hermite_e import hermeadd
    >>> hermeadd([1, 2, 3], [1, 2, 3, 4])
    array([2., 4., 6., 4.])
    """
    return pu._add(c1, c2)
def hermesub(c1, c2):
    """
    Subtract one Hermite series from another.
    Returns the difference of two Hermite series `c1` - `c2`. The
    sequences of coefficients are from lowest order term to highest, i.e.,
    [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
    Parameters
    -----
    c1, c2 : array_like
        1-D arrays of Hermite series coefficients ordered from low to
        high.
    Returns
    -----
    out : ndarray
        Of Hermite series coefficients representing their difference.
    See Also
    -----
    hermeadd, hermemulx, hermemul, hermediv, hermepow
    Notes
    -----
    Unlike multiplication, division, etc., the difference of two Hermite
    series is a Hermite series (without having to "reproject" the result
    onto the basis set) so subtraction, just like that of "standard"
    polynomials, is simply "component-wise."
    Examples
    -----
    >>> from numpy.polynomial.hermite_e import hermesub
    >>> hermesub([1, 2, 3, 4], [1, 2, 3])
    array([0., 0., 0., 4.])
    """

```

```

301: (4)             return pu._sub(c1, c2)
302: (0)             def hermemulx(c):
303: (4)                 """Multiply a Hermite series by x.
304: (4)                 Multiply the Hermite series `c` by x, where x is the independent
305: (4)                 variable.
306: (4)                 Parameters
307: (4)                 -----
308: (4)                 c : array_like
309: (8)                     1-D array of Hermite series coefficients ordered from low to
310: (8)                     high.
311: (4)                 Returns
312: (4)                 -----
313: (4)                 out : ndarray
314: (8)                     Array representing the result of the multiplication.
315: (4)                 Notes
316: (4)                 -----
317: (4)                 The multiplication uses the recursion relationship for Hermite
318: (4)                 polynomials in the form
319: (4)                 .. math::
320: (8)                     xP_i(x) = (P_{i + 1}(x) + iP_{i - 1}(x)))
321: (4)                 Examples
322: (4)                 -----
323: (4)                 >>> from numpy.polynomial.hermite_e import hermemulx
324: (4)                 >>> hermemulx([1, 2, 3])
325: (4)                 array([2.,  7.,  2.,  3.])
326: (4)                 """
327: (4)                 [c] = pu.as_series([c])
328: (4)                 if len(c) == 1 and c[0] == 0:
329: (8)                     return c
330: (4)                 prd = np.empty(len(c) + 1, dtype=c.dtype)
331: (4)                 prd[0] = c[0]*0
332: (4)                 prd[1] = c[0]
333: (4)                 for i in range(1, len(c)):
334: (8)                     prd[i + 1] = c[i]
335: (8)                     prd[i - 1] += c[i]*i
336: (4)                 return prd
337: (0)             def hermemul(c1, c2):
338: (4)                 """
339: (4)                 Multiply one Hermite series by another.
340: (4)                 Returns the product of two Hermite series `c1` * `c2`. The arguments
341: (4)                 are sequences of coefficients, from lowest order "term" to highest,
342: (4)                 e.g., [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
343: (4)                 Parameters
344: (4)                 -----
345: (4)                 c1, c2 : array_like
346: (8)                     1-D arrays of Hermite series coefficients ordered from low to
347: (8)                     high.
348: (4)                 Returns
349: (4)                 -----
350: (4)                 out : ndarray
351: (8)                     Of Hermite series coefficients representing their product.
352: (4)                 See Also
353: (4)                 -----
354: (4)                 hermeadd, hermesub, hermemulx, hermediv, hermepow
355: (4)                 Notes
356: (4)                 -----
357: (4)                 In general, the (polynomial) product of two C-series results in terms
358: (4)                 that are not in the Hermite polynomial basis set. Thus, to express
359: (4)                 the product as a Hermite series, it is necessary to "reproject" the
360: (4)                 product onto said basis set, which may produce "unintuitive" (but
361: (4)                 correct) results; see Examples section below.
362: (4)                 Examples
363: (4)                 -----
364: (4)                 >>> from numpy.polynomial.hermite_e import hermemul
365: (4)                 >>> hermemul([1, 2, 3], [0, 1, 2])
366: (4)                 array([14.,  15.,  28.,   7.,   6.])
367: (4)                 """
368: (4)                 [c1, c2] = pu.as_series([c1, c2])
369: (4)                 if len(c1) > len(c2):

```

```

370: (8)           c = c2
371: (8)           xs = c1
372: (4)           else:
373: (8)             c = c1
374: (8)             xs = c2
375: (4)           if len(c) == 1:
376: (8)             c0 = c[0]*xs
377: (8)             c1 = 0
378: (4)           elif len(c) == 2:
379: (8)             c0 = c[0]*xs
380: (8)             c1 = c[1]*xs
381: (4)           else:
382: (8)             nd = len(c)
383: (8)             c0 = c[-2]*xs
384: (8)             c1 = c[-1]*xs
385: (8)             for i in range(3, len(c) + 1):
386: (12)               tmp = c0
387: (12)               nd = nd - 1
388: (12)               c0 = hermesub(c[-i]*xs, c1*(nd - 1))
389: (12)               c1 = hermeadd(tmp, hermemulx(c1))
390: (4)             return hermeadd(c0, hermemulx(c1))
391: (0)           def hermediv(c1, c2):
392: (4)             """
393: (4)               Divide one Hermite series by another.
394: (4)               Returns the quotient-with-remainder of two Hermite series
395: (4)               `c1` / `c2`. The arguments are sequences of coefficients from lowest
396: (4)               order "term" to highest, e.g., [1,2,3] represents the series
397: (4)               ``P_0 + 2*P_1 + 3*P_2``.
398: (4)               Parameters
399: (4)               -----
400: (4)               c1, c2 : array_like
401: (8)                 1-D arrays of Hermite series coefficients ordered from low to
402: (8)                 high.
403: (4)               Returns
404: (4)               -----
405: (4)               [quo, rem] : ndarray
406: (8)                 Of Hermite series coefficients representing the quotient and
407: (8)                 remainder.
408: (4)               See Also
409: (4)               -----
410: (4)               hermeadd, hermesub, hermemulx, hermemul, hermepow
411: (4)               Notes
412: (4)               -----
413: (4)               In general, the (polynomial) division of one Hermite series by another
414: (4)               results in quotient and remainder terms that are not in the Hermite
415: (4)               polynomial basis set. Thus, to express these results as a Hermite
416: (4)               series, it is necessary to "reproject" the results onto the Hermite
417: (4)               basis set, which may produce "unintuitive" (but correct) results; see
418: (4)               Examples section below.
419: (4)               Examples
420: (4)               -----
421: (4)               >>> from numpy.polynomial.hermite_e import hermediv
422: (4)               >>> hermediv([ 14.,  15.,  28.,   7.,   6.], [0, 1, 2])
423: (4)               (array([1., 2., 3.]), array([0.]))
424: (4)               >>> hermediv([ 15.,  17.,  28.,   7.,   6.], [0, 1, 2])
425: (4)               (array([1., 2., 3.]), array([1., 2.]))
426: (4)               """
427: (4)               return pu._div(hermemul, c1, c2)
428: (0)           def hermepow(c, pow, maxpower=16):
429: (4)             """
430: (4)               Raise a Hermite series to a power.
431: (4)               Returns the Hermite series `c` raised to the power `pow`. The
432: (4)               argument `c` is a sequence of coefficients ordered from low to high.
433: (4)               i.e., [1,2,3] is the series ``P_0 + 2*P_1 + 3*P_2``.
434: (4)               Parameters
435: (4)               -----
436: (4)               c : array_like
437: (8)                 1-D array of Hermite series coefficients ordered from low to
438: (8)                 high.
439: (4)               pow : integer

```

```

439: (8)          Power to which the series will be raised
440: (4)          maxpower : integer, optional
441: (8)          Maximum power allowed. This is mainly to limit growth of the series
442: (8)          to unmanageable size. Default is 16
443: (4)          Returns
444: (4)          -----
445: (4)          coef : ndarray
446: (8)          Hermite series of power.
447: (4)          See Also
448: (4)          -----
449: (4)          hermeadd, hermesub, hermemulx, hermemul, hermediv
450: (4)          Examples
451: (4)          -----
452: (4)          >>> from numpy.polynomial.hermite_e import hermepow
453: (4)          >>> hermepow([1, 2, 3], 2)
454: (4)          array([23.,  28.,  46.,  12.,   9.])
455: (4)          """
456: (4)          return pu._pow(hermemul, c, pow, maxpower)
457: (0)          def hermeder(c, m=1, scl=1, axis=0):
458: (4)          """
459: (4)          Differentiate a Hermite_e series.
460: (4)          Returns the series coefficients `c` differentiated `m` times along
461: (4)          `axis`. At each iteration the result is multiplied by `scl` (the
462: (4)          scaling factor is for use in a linear change of variable). The argument
463: (4)          `c` is an array of coefficients from low to high degree along each
464: (4)          axis, e.g., [1,2,3] represents the series ``1*He_0 + 2*He_1 + 3*He_2``
465: (4)          while [[1,2],[1,2]] represents ``1*He_0(x)*He_0(y) + 1*He_1(x)*He_0(y)``
466: (4)          + 2*He_0(x)*He_1(y) + 2*He_1(x)*He_1(y)`` if axis=0 is ``x`` and axis=1
467: (4)          is ``y``.
468: (4)          Parameters
469: (4)          -----
470: (4)          c : array_like
471: (8)          Array of Hermite_e series coefficients. If `c` is multidimensional
472: (8)          the different axis correspond to different variables with the
473: (8)          degree in each axis given by the corresponding index.
474: (4)          m : int, optional
475: (8)          Number of derivatives taken, must be non-negative. (Default: 1)
476: (4)          scl : scalar, optional
477: (8)          Each differentiation is multiplied by `scl`. The end result is
478: (8)          multiplication by ``scl**m``. This is for use in a linear change of
479: (8)          variable. (Default: 1)
480: (4)          axis : int, optional
481: (8)          Axis over which the derivative is taken. (Default: 0).
482: (8)          .. versionadded:: 1.7.0
483: (4)          Returns
484: (4)          -----
485: (4)          der : ndarray
486: (8)          Hermite series of the derivative.
487: (4)          See Also
488: (4)          -----
489: (4)          hermeint
490: (4)          Notes
491: (4)          -----
492: (4)          In general, the result of differentiating a Hermite series does not
493: (4)          resemble the same operation on a power series. Thus the result of this
494: (4)          function may be "unintuitive," albeit correct; see Examples section
495: (4)          below.
496: (4)          Examples
497: (4)          -----
498: (4)          >>> from numpy.polynomial.hermite_e import hermeder
499: (4)          >>> hermeder([ 1.,  1.,  1.,  1.])
500: (4)          array([1.,  2.,  3.])
501: (4)          >>> hermeder([-0.25,  1.,  1./2.,  1./3.,  1./4 ], m=2)
502: (4)          array([1.,  2.,  3.])
503: (4)          """
504: (4)          c = np.array(c, ndmin=1, copy=True)
505: (4)          if c.dtype.char in '?bBhHiIlLqQpP':
506: (8)              c = c.astype(np.double)
507: (4)          cnt = pu._deprecate_as_int(m, "the order of derivation")

```

```

508: (4)             iaxis = pu._deprecate_as_int(axis, "the axis")
509: (4)             if cnt < 0:
510: (8)                 raise ValueError("The order of derivation must be non-negative")
511: (4)             iaxis = normalize_axis_index(iaxis, c.ndim)
512: (4)             if cnt == 0:
513: (8)                 return c
514: (4)             c = np.moveaxis(c, iaxis, 0)
515: (4)             n = len(c)
516: (4)             if cnt >= n:
517: (8)                 return c[:]*0
518: (4)             else:
519: (8)                 for i in range(cnt):
520: (12)                     n = n - 1
521: (12)                     c *= scl
522: (12)                     der = np.empty((n,) + c.shape[1:], dtype=c.dtype)
523: (12)                     for j in range(n, 0, -1):
524: (16)                         der[j - 1] = j*c[j]
525: (12)                     c = der
526: (4)             c = np.moveaxis(c, 0, iaxis)
527: (4)             return c
528: (0)             def hermeint(c, m=1, k=[], lbnd=0, scl=1, axis=0):
529: (4)                 """
530: (4)                 Integrate a Hermite_e series.
531: (4)                 Returns the Hermite_e series coefficients `c` integrated `m` times from
532: (4)                 `lbnd` along `axis`. At each iteration the resulting series is
533: (4)                 **multiplied** by `scl` and an integration constant, `k`, is added.
534: (4)                 The scaling factor is for use in a linear change of variable. ("Buyer
535: (4)                 beware": note that, depending on what one is doing, one may want `scl`
536: (4)                 to be the reciprocal of what one might expect; for more information,
537: (4)                 see the Notes section below.) The argument `c` is an array of
538: (4)                 coefficients from low to high degree along each axis, e.g., [1,2,3]
539: (4)                 represents the series ``H_0 + 2*H_1 + 3*H_2`` while [[1,2],[1,2]]
540: (4)                 represents ``1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) + 2*H_0(x)*H_1(y) +
541: (4)                 2*H_1(x)*H_1(y)`` if axis=0 is ``x`` and axis=1 is ``y``.
542: (4)             Parameters
543: (4)             -----
544: (4)             c : array_like
545: (8)                 Array of Hermite_e series coefficients. If c is multidimensional
546: (8)                 the different axis correspond to different variables with the
547: (8)                 degree in each axis given by the corresponding index.
548: (4)             m : int, optional
549: (8)                 Order of integration, must be positive. (Default: 1)
550: (4)             k : {}, list, scalar, optional
551: (8)                 Integration constant(s). The value of the first integral at
552: (8)                 ``lbnd`` is the first value in the list, the value of the second
553: (8)                 integral at ``lbnd`` is the second value, etc. If ``k == []`` (the
554: (8)                 default), all constants are set to zero. If ``m == 1``, a single
555: (8)                 scalar can be given instead of a list.
556: (4)             lbnd : scalar, optional
557: (8)                 The lower bound of the integral. (Default: 0)
558: (4)             scl : scalar, optional
559: (8)                 Following each integration the result is *multiplied* by `scl`
560: (8)                 before the integration constant is added. (Default: 1)
561: (4)             axis : int, optional
562: (8)                 Axis over which the integral is taken. (Default: 0).
563: (8)                 .. versionadded:: 1.7.0
564: (4)             Returns
565: (4)             -----
566: (4)             S : ndarray
567: (8)                 Hermite_e series coefficients of the integral.
568: (4)             Raises
569: (4)             -----
570: (4)             ValueError
571: (8)                 If ``m < 0``, ``len(k) > m``, ``np.ndim(lbnd) != 0``, or
572: (8)                 ``np.ndim(scl) != 0``.
573: (4)             See Also
574: (4)             -----
575: (4)             hermeder
576: (4)             Notes

```

```

577: (4)      -----
578: (4)      Note that the result of each integration is *multiplied* by `scl`.
579: (4)      Why is this important to note? Say one is making a linear change of
580: (4)      variable :math:`u = ax + b` in an integral relative to `x`. Then
581: (4)      :math:`dx = du/a`, so one will need to set `scl` equal to
582: (4)      :math:`1/a` - perhaps not what one would have first thought.
583: (4)      Also note that, in general, the result of integrating a C-series needs
584: (4)      to be "reprojected" onto the C-series basis set. Thus, typically,
585: (4)      the result of this function is "unintuitive," albeit correct; see
586: (4)      Examples section below.
587: (4)      Examples
588: (4)      -----
589: (4)      >>> from numpy.polynomial.hermite_e import hermeint
590: (4)      >>> hermeint([1, 2, 3]) # integrate once, value 0 at 0.
591: (4)      array([1., 1., 1., 1.])
592: (4)      >>> hermeint([1, 2, 3], m=2) # integrate twice, value & deriv 0 at 0
593: (4)      array([-0.25      ,  1.          ,  0.5          ,  0.33333333,  0.25      ]) #
may vary
594: (4)      >>> hermeint([1, 2, 3], k=1) # integrate once, value 1 at 0.
595: (4)      array([2., 1., 1., 1.])
596: (4)      >>> hermeint([1, 2, 3], lbnd=-1) # integrate once, value 0 at -1
597: (4)      array([-1.,  1.,  1.,  1.])
598: (4)      >>> hermeint([1, 2, 3], m=2, k=[1, 2], lbnd=-1)
599: (4)      array([ 1.83333333,  0.          ,  0.5          ,  0.33333333,  0.25      ]) #
may vary
600: (4)      """
601: (4)      c = np.array(c, ndmin=1, copy=True)
602: (4)      if c.dtype.char in '?bBhHiIlLqQpP':
603: (8)          c = c.astype(np.double)
604: (4)      if not np.iterable(k):
605: (8)          k = [k]
606: (4)      cnt = pu._deprecate_as_int(m, "the order of integration")
607: (4)      iaxis = pu._deprecate_as_int(axis, "the axis")
608: (4)      if cnt < 0:
609: (8)          raise ValueError("The order of integration must be non-negative")
610: (4)      if len(k) > cnt:
611: (8)          raise ValueError("Too many integration constants")
612: (4)      if np.ndim(lbnd) != 0:
613: (8)          raise ValueError("lbnd must be a scalar.")
614: (4)      if np.ndim(scl) != 0:
615: (8)          raise ValueError("scl must be a scalar.")
616: (4)      iaxis = normalize_axis_index(iaxis, c.ndim)
617: (4)      if cnt == 0:
618: (8)          return c
619: (4)      c = np.moveaxis(c, iaxis, 0)
620: (4)      k = list(k) + [0]*(cnt - len(k))
621: (4)      for i in range(cnt):
622: (8)          n = len(c)
623: (8)          c *= scl
624: (8)          if n == 1 and np.all(c[0] == 0):
625: (12)              c[0] += k[i]
626: (8)          else:
627: (12)              tmp = np.empty((n + 1,) + c.shape[1:], dtype=c.dtype)
628: (12)              tmp[0] = c[0]*0
629: (12)              tmp[1] = c[0]
630: (12)              for j in range(1, n):
631: (16)                  tmp[j + 1] = c[j]/(j + 1)
632: (12)                  tmp[0] += k[i] - hermeval(lbnd, tmp)
633: (12)                  c = tmp
634: (4)      c = np.moveaxis(c, 0, iaxis)
635: (4)      return c
636: (0)      def hermeval(x, c, tensor=True):
637: (4)          """
638: (4)          Evaluate an HermiteE series at points x.
639: (4)          If `c` is of length `n + 1`, this function returns the value:
640: (4)          .. math:: p(x) = c_0 * He_0(x) + c_1 * He_1(x) + \dots + c_n * He_n(x)
641: (4)          The parameter `x` is converted to an array only if it is a tuple or a
642: (4)          list, otherwise it is treated as a scalar. In either case, either `x`
643: (4)          or its elements must support multiplication and addition both with

```

```

644: (4)           themselves and with the elements of `c`.
645: (4)           If `c` is a 1-D array, then `p(x)` will have the same shape as `x`. If
646: (4)           `c` is multidimensional, then the shape of the result depends on the
647: (4)           value of `tensor`. If `tensor` is true the shape will be c.shape[1:] +
648: (4)           x.shape. If `tensor` is false the shape will be c.shape[1:].
649: (4)           Note that scalars have shape ().
650: (4)           Trailing zeros in the coefficients will be used in the evaluation, so
651: (4)           they should be avoided if efficiency is a concern.
652: (4)           Parameters
653: (4)           -----
654: (4)           x : array_like, compatible object
655: (8)             If `x` is a list or tuple, it is converted to an ndarray, otherwise
656: (8)             it is left unchanged and treated as a scalar. In either case, `x`
657: (8)             or its elements must support addition and multiplication with
658: (8)             with themselves and with the elements of `c`.
659: (4)           c : array_like
660: (8)             Array of coefficients ordered so that the coefficients for terms of
661: (8)             degree n are contained in c[n]. If `c` is multidimensional the
662: (8)             remaining indices enumerate multiple polynomials. In the two
663: (8)             dimensional case the coefficients may be thought of as stored in
664: (8)             the columns of `c`.
665: (4)           tensor : boolean, optional
666: (8)             If True, the shape of the coefficient array is extended with ones
667: (8)             on the right, one for each dimension of `x`. Scalars have dimension 0
668: (8)             for this action. The result is that every column of coefficients in
669: (8)             `c` is evaluated for every element of `x`. If False, `x` is broadcast
670: (8)             over the columns of `c` for the evaluation. This keyword is useful
671: (8)             when `c` is multidimensional. The default value is True.
672: (8)             .. versionadded:: 1.7.0
673: (4)           Returns
674: (4)           -----
675: (4)           values : ndarray, algebra_like
676: (8)             The shape of the return value is described above.
677: (4)           See Also
678: (4)           -----
679: (4)           hermeval2d, hermegrid2d, hermeval3d, hermegrid3d
680: (4)           Notes
681: (4)           -----
682: (4)           The evaluation uses Clenshaw recursion, aka synthetic division.
683: (4)           Examples
684: (4)           -----
685: (4)           >>> from numpy.polynomial.hermite_e import hermeval
686: (4)           >>> coef = [1,2,3]
687: (4)           >>> hermeval(1, coef)
688: (4)           3.0
689: (4)           >>> hermeval([[1,2],[3,4]], coef)
690: (4)           array([[ 3., 14.],
691: (11)             [31., 54.]])
692: (4)           """
693: (4)           c = np.array(c, ndmin=1, copy=False)
694: (4)           if c.dtype.char in '?bBhHiIlLqQpP':
695: (8)             c = c.astype(np.double)
696: (4)           if isinstance(x, (tuple, list)):
697: (8)             x = np.asarray(x)
698: (4)           if isinstance(x, np.ndarray) and tensor:
699: (8)             c = c.reshape(c.shape + (1,)*x.ndim)
700: (4)           if len(c) == 1:
701: (8)             c0 = c[0]
702: (8)             c1 = 0
703: (4)           elif len(c) == 2:
704: (8)             c0 = c[0]
705: (8)             c1 = c[1]
706: (4)           else:
707: (8)             nd = len(c)
708: (8)             c0 = c[-2]
709: (8)             c1 = c[-1]
710: (8)             for i in range(3, len(c) + 1):
711: (12)               tmp = c0
712: (12)               nd = nd - 1

```

```

713: (12)          c0 = c[-i] - c1*(nd - 1)
714: (12)          c1 = tmp + c1*x
715: (4)          return c0 + c1*x
716: (0)          def hermeval2d(x, y, c):
717: (4)          """
718: (4)          Evaluate a 2-D HermiteE series at points (x, y).
719: (4)          This function returns the values:
720: (4)          .. math:: p(x,y) = \sum_{i,j} c_{i,j} * He_i(x) * He_j(y)
721: (4)          The parameters `x` and `y` are converted to arrays only if they are
722: (4)          tuples or a lists, otherwise they are treated as a scalars and they
723: (4)          must have the same shape after conversion. In either case, either `x`
724: (4)          and `y` or their elements must support multiplication and addition both
725: (4)          with themselves and with the elements of `c`.
726: (4)          If `c` is a 1-D array a one is implicitly appended to its shape to make
727: (4)          it 2-D. The shape of the result will be c.shape[2:] + x.shape.
728: (4)          Parameters
729: (4)          -----
730: (4)          x, y : array_like, compatible objects
731: (8)          The two dimensional series is evaluated at the points `(x, y)`,
732: (8)          where `x` and `y` must have the same shape. If `x` or `y` is a list
733: (8)          or tuple, it is first converted to an ndarray, otherwise it is left
734: (8)          unchanged and if it isn't an ndarray it is treated as a scalar.
735: (4)          c : array_like
736: (8)          Array of coefficients ordered so that the coefficient of the term
737: (8)          of multi-degree i,j is contained in ``c[i,j]``. If `c` has
738: (8)          dimension greater than two the remaining indices enumerate multiple
739: (8)          sets of coefficients.
740: (4)          Returns
741: (4)          -----
742: (4)          values : ndarray, compatible object
743: (8)          The values of the two dimensional polynomial at points formed with
744: (8)          pairs of corresponding values from `x` and `y`.
745: (4)          See Also
746: (4)          -----
747: (4)          hermeval, hermgrid2d, hermeval3d, hermgrid3d
748: (4)          Notes
749: (4)          -----
750: (4)          .. versionadded:: 1.7.0
751: (4)          """
752: (4)          return pu._valnd(hermeval, c, x, y)
753: (0)          def hermgrid2d(x, y, c):
754: (4)          """
755: (4)          Evaluate a 2-D HermiteE series on the Cartesian product of x and y.
756: (4)          This function returns the values:
757: (4)          .. math:: p(a,b) = \sum_{i,j} c_{i,j} * H_i(a) * H_j(b)
758: (4)          where the points `(a, b)` consist of all pairs formed by taking
759: (4)          `a` from `x` and `b` from `y`. The resulting points form a grid with
760: (4)          `x` in the first dimension and `y` in the second.
761: (4)          The parameters `x` and `y` are converted to arrays only if they are
762: (4)          tuples or a lists, otherwise they are treated as a scalars. In either
763: (4)          case, either `x` and `y` or their elements must support multiplication
764: (4)          and addition both with themselves and with the elements of `c`.
765: (4)          If `c` has fewer than two dimensions, ones are implicitly appended to
766: (4)          its shape to make it 2-D. The shape of the result will be c.shape[2:] +
767: (4)          x.shape.
768: (4)          Parameters
769: (4)          -----
770: (4)          x, y : array_like, compatible objects
771: (8)          The two dimensional series is evaluated at the points in the
772: (8)          Cartesian product of `x` and `y`. If `x` or `y` is a list or
773: (8)          tuple, it is first converted to an ndarray, otherwise it is left
774: (8)          unchanged and, if it isn't an ndarray, it is treated as a scalar.
775: (4)          c : array_like
776: (8)          Array of coefficients ordered so that the coefficients for terms of
777: (8)          degree i,j are contained in ``c[i,j]``. If `c` has dimension
778: (8)          greater than two the remaining indices enumerate multiple sets of
779: (8)          coefficients.
780: (4)          Returns
781: (4)          -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

782: (4)           values : ndarray, compatible object
783: (8)             The values of the two dimensional polynomial at points in the
Cartesian
784: (8)               product of `x` and `y`.
785: (4)             See Also
786: (4)               -----
787: (4)               hermeval, hermeval2d, hermeval3d, hermagrid3d
788: (4)             Notes
789: (4)               -----
790: (4)               .. versionadded:: 1.7.0
791: (4)               """
792: (4)               return pu._gridnd(hermeval, c, x, y)
def hermeval3d(x, y, z, c):
    """
    Evaluate a 3-D Hermite_e series at points (x, y, z).
    This function returns the values:
    .. math:: p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * He_i(x) * He_j(y) * He_k(z)
    The parameters `x`, `y`, and `z` are converted to arrays only if
    they are tuples or a lists, otherwise they are treated as a scalars and
    they must have the same shape after conversion. In either case, either
    `x`, `y`, and `z` or their elements must support multiplication and
    addition both with themselves and with the elements of `c`.
    If `c` has fewer than 3 dimensions, ones are implicitly appended to its
    shape to make it 3-D. The shape of the result will be c.shape[3:] +
    x.shape.
    Parameters
    -----
    x, y, z : array_like, compatible object
        The three dimensional series is evaluated at the points
        `(x, y, z)`, where `x`, `y`, and `z` must have the same shape. If
        any of `x`, `y`, or `z` is a list or tuple, it is first converted
        to an ndarray, otherwise it is left unchanged and if it isn't an
        ndarray it is treated as a scalar.
    c : array_like
        Array of coefficients ordered so that the coefficient of the term of
        multi-degree i,j,k is contained in ``c[i,j,k]``. If `c` has dimension
        greater than 3 the remaining indices enumerate multiple sets of
        coefficients.
    Returns
    -----
    values : ndarray, compatible object
        The values of the multidimensional polynomial on points formed with
        triples of corresponding values from `x`, `y`, and `z`.
    See Also
    -----
    hermeval, hermeval2d, hermagrid2d, hermagrid3d
    Notes
    -----
    .. versionadded:: 1.7.0
    """
    return pu._valnd(hermeval, c, x, y, z)
def hermagrid3d(x, y, z, c):
    """
    Evaluate a 3-D HermiteE series on the Cartesian product of x, y, and z.
    This function returns the values:
    .. math:: p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * He_i(a) * He_j(b) * He_k(c)
    where the points `(a, b, c)` consist of all triples formed by taking
    `a` from `x`, `b` from `y`, and `c` from `z`. The resulting points form
    a grid with `x` in the first dimension, `y` in the second, and `z` in
    the third.
    The parameters `x`, `y`, and `z` are converted to arrays only if they
    are tuples or a lists, otherwise they are treated as a scalars. In
    either case, either `x`, `y`, and `z` or their elements must support
    multiplication and addition both with themselves and with the elements
    of `c`.
    If `c` has fewer than three dimensions, ones are implicitly appended to
    its shape to make it 3-D. The shape of the result will be c.shape[3:] +
    x.shape + y.shape + z.shape.
    Parameters

```

```

850: (4)
851: (4)
852: (8)
853: (8)
854: (8)
855: (8)
856: (8)
857: (4)
858: (8)
859: (8)
860: (8)
861: (8)
862: (4)
863: (4)
864: (4)
865: (8)
Cartesian
866: (8)
867: (4)
868: (4)
869: (4)
870: (4)
871: (4)
872: (4)
873: (4)
874: (4)
875: (0)
def hermevander(x, deg):
    """Pseudo-Vandermonde matrix of given degree.
    Returns the pseudo-Vandermonde matrix of degree `deg` and sample points
    `x`. The pseudo-Vandermonde matrix is defined by
    .. math:: V[..., i] = He_i(x),
    where `0 <= i <= deg`. The leading indices of `V` index the elements of
    `x` and the last index is the degree of the HermiteE polynomial.
    If `c` is a 1-D array of coefficients of length `n + 1` and `V` is the
    array ``V = hermevander(x, n)``, then ``np.dot(V, c)`` and
    ``hermeval(x, c)`` are the same up to roundoff. This equivalence is
    useful both for least squares fitting and for the evaluation of a large
    number of HermiteE series of the same degree and sample points.
    Parameters
    -----
    x : array_like
        Array of points. The dtype is converted to float64 or complex128
        depending on whether any of the elements are complex. If `x` is
        scalar it is converted to a 1-D array.
    deg : int
        Degree of the resulting matrix.
    Returns
    -----
    vander : ndarray
        The pseudo-Vandermonde matrix. The shape of the returned matrix is
        ``x.shape + (deg + 1,)``, where The last index is the degree of the
        corresponding HermiteE polynomial. The dtype will be the same as
        the converted `x`.
    Examples
    -----
    >>> from numpy.polynomial.hermite_e import hermevander
    >>> x = np.array([-1, 0, 1])
    >>> hermevander(x, 3)
    array([[ 1., -1.,  0.,  2.],
           [ 1.,  0., -1., -0.],
           [ 1.,  1.,  0., -2.]])
    """
    ideg = pu._deprecate_as_int(deg, "deg")
    if ideg < 0:
        raise ValueError("deg must be non-negative")
    x = np.array(x, copy=False, ndmin=1) + 0.0
    dims = (ideg + 1,) + x.shape
    dtyp = x.dtype
    v = np.empty(dims, dtype=dtyp)

```

```

918: (4)           v[0] = x*0 + 1
919: (4)           if ideg > 0:
920: (8)             v[1] = x
921: (8)             for i in range(2, ideg + 1):
922: (12)               v[i] = (v[i-1]*x - v[i-2]*(i - 1))
923: (4)           return np.moveaxis(v, 0, -1)
924: (0)           def hermievander2d(x, y, deg):
925: (4)             """Pseudo-Vandermonde matrix of given degrees.
926: (4)             Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
927: (4)             points `(x, y)`. The pseudo-Vandermonde matrix is defined by
928: (4)               .. math:: V[..., (deg[1] + 1)*i + j] = He_i(x) * He_j(y),
929: (4)             where `0 <= i <= deg[0]` and `0 <= j <= deg[1]`. The leading indices of
930: (4)             `V` index the points `(x, y)` and the last index encodes the degrees of
931: (4)             the HermiteE polynomials.
932: (4)             If ``V = hermievander2d(x, y, [xdeg, ydeg])``, then the columns of `V`
933: (4)             correspond to the elements of a 2-D coefficient array `c` of shape
934: (4)             (xdeg + 1, ydeg + 1) in the order
935: (4)               .. math:: c_{\{00\}}, c_{\{01\}}, c_{\{02\}} \dots , c_{\{10\}}, c_{\{11\}}, c_{\{12\}} \dots
936: (4)             and ``np.dot(V, c.flat)`` and ``hermeval2d(x, y, c)`` will be the same
937: (4)             up to roundoff. This equivalence is useful both for least squares
938: (4)             fitting and for the evaluation of a large number of 2-D HermiteE
939: (4)             series of the same degrees and sample points.
940: (4)           Parameters
941: (4)           -----
942: (4)           x, y : array_like
943: (8)             Arrays of point coordinates, all of the same shape. The dtypes
944: (8)             will be converted to either float64 or complex128 depending on
945: (8)             whether any of the elements are complex. Scalars are converted to
946: (8)             1-D arrays.
947: (4)           deg : list of ints
948: (8)             List of maximum degrees of the form [x_deg, y_deg].
949: (4)           Returns
950: (4)           -----
951: (4)           vander2d : ndarray
952: (8)             The shape of the returned matrix is ``x.shape + (order,)``, where
953: (8)               :math:`order = (deg[0]+1)*(deg[1]+1)` . The dtype will be the same
954: (8)             as the converted `x` and `y` .
955: (4)           See Also
956: (4)           -----
957: (4)           hermievander, hermievander3d, hermeval2d, hermeval3d
958: (4)           Notes
959: (4)           -----
960: (4)           .. versionadded:: 1.7.0
961: (4)           """
962: (4)           return pu._vander_nd_flat((hermievander, hermievander), (x, y), deg)
963: (0)           def hermievander3d(x, y, z, deg):
964: (4)             """Pseudo-Vandermonde matrix of given degrees.
965: (4)             Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
966: (4)             points `(x, y, z)`. If `l, m, n` are the given degrees in `x, y, z`,
967: (4)             then Hehe pseudo-Vandermonde matrix is defined by
968: (4)               .. math:: V[..., (m+1)(n+1)i + (n+1)j + k] = He_i(x)*He_j(y)*He_k(z),
969: (4)             where `0 <= i <= l`, `0 <= j <= m`, and `0 <= j <= n` . The leading
970: (4)             indices of `V` index the points `(x, y, z)` and the last index encodes
971: (4)             the degrees of the HermiteE polynomials.
972: (4)             If ``V = hermievander3d(x, y, z, [xdeg, ydeg, zdeg])``, then the columns
973: (4)             of `V` correspond to the elements of a 3-D coefficient array `c` of
974: (4)             shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order
975: (4)               .. math:: c_{\{000\}}, c_{\{001\}}, c_{\{002\}}, \dots , c_{\{010\}}, c_{\{011\}}, c_{\{012\}}, \dots
976: (4)             and ``np.dot(V, c.flat)`` and ``hermeval3d(x, y, z, c)`` will be the
977: (4)             same up to roundoff. This equivalence is useful both for least squares
978: (4)             fitting and for the evaluation of a large number of 3-D HermiteE
979: (4)             series of the same degrees and sample points.
980: (4)           Parameters
981: (4)           -----
982: (4)           x, y, z : array_like
983: (8)             Arrays of point coordinates, all of the same shape. The dtypes will
984: (8)             be converted to either float64 or complex128 depending on whether
985: (8)             any of the elements are complex. Scalars are converted to 1-D
986: (8)             arrays.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

987: (4)             deg : list of ints
988: (8)             List of maximum degrees of the form [x_deg, y_deg, z_deg].
989: (4)             Returns
990: (4)
991: (4)             vander3d : ndarray
992: (8)             The shape of the returned matrix is ``x.shape + (order,)``, where
993: (8)             :math:`order = (deg[0]+1)*(deg[1]+1)*(deg[2]+1)` . The dtype will
994: (8)             be the same as the converted `x`, `y`, and `z`.
995: (4)             See Also
996: (4)
997: (4)             hermевандер, herмевандер3д, herмевал2д, herмевал3д
998: (4)             Notes
999: (4)
1000: (4)            .. versionadded:: 1.7.0
1001: (4)
1002: (4)            return pu._vander_nd_flat((hermевандер, herмевандер, herмевандер), (x, y,
z), deg)
1003: (0)             def hermefit(x, y, deg, rcond=None, full=False, w=None):
1004: (4)             """
1005: (4)             Least squares fit of Hermite series to data.
1006: (4)             Return the coefficients of a HermiteE series of degree `deg` that is
1007: (4)             the least squares fit to the data values `y` given at points `x`. If
1008: (4)             `y` is 1-D the returned coefficients will also be 1-D. If `y` is 2-D
1009: (4)             multiple fits are done, one for each column of `y`, and the resulting
1010: (4)             coefficients are stored in the corresponding columns of a 2-D return.
1011: (4)             The fitted polynomial(s) are in the form
1012: (4)             .. math:: p(x) = c_0 + c_1 * He_1(x) + ... + c_n * He_n(x),
1013: (4)             where `n` is `deg`.
1014: (4)             Parameters
1015: (4)
1016: (4)             x : array_like, shape (M,)
1017: (8)             x-coordinates of the M sample points ``(x[i], y[i])``.
1018: (4)             y : array_like, shape (M,) or (M, K)
1019: (8)             y-coordinates of the sample points. Several data sets of sample
1020: (8)             points sharing the same x-coordinates can be fitted at once by
1021: (8)             passing in a 2D-array that contains one dataset per column.
1022: (4)             deg : int or 1-D array_like
1023: (8)             Degree(s) of the fitting polynomials. If `deg` is a single integer
1024: (8)             all terms up to and including the `deg`'th term are included in the
1025: (8)             fit. For NumPy versions >= 1.11.0 a list of integers specifying the
1026: (8)             degrees of the terms to include may be used instead.
1027: (4)             rcond : float, optional
1028: (8)             Relative condition number of the fit. Singular values smaller than
1029: (8)             this relative to the largest singular value will be ignored. The
1030: (8)             default value is len(x)*eps, where eps is the relative precision of
1031: (8)             the float type, about 2e-16 in most cases.
1032: (4)             full : bool, optional
1033: (8)             Switch determining nature of return value. When it is False (the
1034: (8)             default) just the coefficients are returned, when True diagnostic
1035: (8)             information from the singular value decomposition is also returned.
1036: (4)             w : array_like, shape (M,), optional
1037: (8)             Weights. If not None, the weight ``w[i]`` applies to the unsquared
1038: (8)             residual ``y[i] - y_hat[i]`` at ``x[i]``. Ideally the weights are
1039: (8)             chosen so that the errors of the products ``w[i]*y[i]`` all have the
1040: (8)             same variance. When using inverse-variance weighting, use
1041: (8)             ``w[i] = 1/sigma(y[i])``. The default value is None.
1042: (4)             Returns
1043: (4)
1044: (4)             coef : ndarray, shape (M,) or (M, K)
1045: (8)             Hermite coefficients ordered from low to high. If `y` was 2-D,
1046: (8)             the coefficients for the data in column k of `y` are in column
1047: (8)             `k`.
1048: (4)             [residuals, rank, singular_values, rcond] : list
1049: (8)             These values are only returned if ``full == True``
1050: (8)             - residuals -- sum of squared residuals of the least squares fit
1051: (8)             - rank -- the numerical rank of the scaled Vandermonde matrix
1052: (8)             - singular_values -- singular values of the scaled Vandermonde matrix
1053: (8)             - rcond -- value of `rcond`.
1054: (8)             For more details, see `numpy.linalg.lstsq` .

```

```

1055: (4)          Warns
1056: (4)
1057: (4)
1058: (8)          -----
1059: (8)          RankWarning
1060: (8)          The rank of the coefficient matrix in the least-squares fit is
1061: (8)          deficient. The warning is only raised if ``full = False``. The
1062: (8)          warnings can be turned off by
1063: (4)          >>> import warnings
1064: (4)          >>> warnings.simplefilter('ignore', np.RankWarning)
1065: (4)          See Also
1066: (4)          -----
1067: (4)          numpy.polynomial.chebyshev.chebfit
1068: (4)          numpy.polynomial.legendre.legfit
1069: (4)          numpy.polynomial.polynomial.polyfit
1070: (4)          numpy.polynomial.hermite.hermfit
1071: (4)          numpy.polynomial.laguerre.lagfit
1072: (4)          hermeval : Evaluates a Hermite series.
1073: (4)          hermevander : pseudo Vandermonde matrix of Hermite series.
1074: (4)          hermeweight : HermiteE weight function.
1075: (4)          numpy.linalg.lstsq : Computes a least-squares fit from the matrix.
1076: (4)          scipy.interpolate.UnivariateSpline : Computes spline fits.
1077: (4)          Notes
1078: (4)          -----
1079: (4)          The solution is the coefficients of the HermiteE series `p` that
1080: (4)          minimizes the sum of the weighted squared errors
1081: (4)          .. math:: E = \sum_j w_j^2 * |y_j - p(x_j)|^2,
1082: (4)          where the :math:`w_j` are the weights. This problem is solved by
1083: (4)          setting up the (typically) overdetermined matrix equation
1084: (4)          .. math:: V(x) * c = w * y,
1085: (4)          where `V` is the pseudo Vandermonde matrix of `x`, the elements of `c`
1086: (4)          are the coefficients to be solved for, and the elements of `y` are the
1087: (4)          observed values. This equation is then solved using the singular value
1088: (4)          decomposition of `V`.
1089: (4)          If some of the singular values of `V` are so small that they are
1090: (4)          neglected, then a `RankWarning` will be issued. This means that the
1091: (4)          coefficient values may be poorly determined. Using a lower order fit
1092: (4)          will usually get rid of the warning. The `rcond` parameter can also be
1093: (4)          set to a value smaller than its default, but the resulting fit may be
1094: (4)          spurious and have large contributions from roundoff error.
1095: (4)          Fits using HermiteE series are probably most useful when the data can
1096: (4)          be approximated by ``sqrt(w(x)) * p(x)``, where `w(x)` is the HermiteE
1097: (4)          weight. In that case the weight ``sqrt(w(x[i]))`` should be used
1098: (4)          together with data values ``y[i]/sqrt(w(x[i]))``. The weight function is
1099: (4)          available as `hermeweight`.
1100: (4)          References
1101: (11)          -----
1102: (4)          .. [1] Wikipedia, "Curve fitting",
1103: (4)          https://en.wikipedia.org/wiki/Curve_fitting
1104: (4)          Examples
1105: (4)          -----
1106: (4)          >>> from numpy.polynomial.hermite_e import hermefit, hermeval
1107: (4)          >>> x = np.linspace(-10, 10)
1108: (4)          >>> np.random.seed(123)
1109: (4)          >>> err = np.random.randn(len(x))/10
1110: (4)          >>> y = hermeval(x, [1, 2, 3]) + err
1111: (4)          >>> hermefit(x, y, 2)
1112: (4)          array([ 1.01690445,  1.99951418,  2.99948696]) # may vary
1113: (0)          """
1114: (4)          return pu._fit(hermevander, x, y, deg, rcond, full, w)
def hermecompanion(c):
    """
    Return the scaled companion matrix of c.
    The basis polynomials are scaled so that the companion matrix is
    symmetric when `c` is an HermiteE basis polynomial. This provides
    better eigenvalue estimates than the unscaled case and for basis
    polynomials the eigenvalues are guaranteed to be real if
    `numpy.linalg.eigvalsh` is used to obtain them.
    Parameters
    -----
    c : array_like

```

```

1124: (8)           1-D array of HermiteE series coefficients ordered from low to high
1125: (8)           degree.
1126: (4)           Returns
1127: (4)
1128: (4)
1129: (8)           mat : ndarray
1130: (4)           Scaled companion matrix of dimensions (deg, deg).
1131: (4)           Notes
1132: (4)
1133: (4)
1134: (4)           .. versionadded:: 1.7.0
1135: (4)
1136: (8)           """
1137: (4)           [c] = pu.as_series([c])
1138: (8)           if len(c) < 2:
1139: (4)               raise ValueError('Series must have maximum degree of at least 1.')
1140: (4)           if len(c) == 2:
1141: (4)               return np.array([[[-c[0]/c[1]]]])
1142: (4)           n = len(c) - 1
1143: (4)           mat = np.zeros((n, n), dtype=c.dtype)
1144: (4)           scl = np.hstack((1., 1./np.sqrt(np.arange(n - 1, 0, -1))))
1145: (4)           scl = np.multiply.accumulate(scl)[::-1]
1146: (4)           top = mat.reshape(-1)[1::n+1]
1147: (4)           bot = mat.reshape(-1)[n::n+1]
1148: (4)           top[...] = np.sqrt(np.arange(1, n))
1149: (0)           bot[...] = top
1150: (4)           mat[:, -1] -= scl*c[:-1]/c[-1]
1151: (4)           return mat
1152: (4)
1153: (4)           def hermroots(c):
1154: (4)               """
1155: (4)               Compute the roots of a HermiteE series.
1156: (4)               Return the roots (a.k.a. "zeros") of the polynomial
1157: (4)               .. math:: p(x) = \sum_i c[i] * He_i(x).
1158: (4)               Parameters
1159: (4)
1160: (4)               c : 1-D array_like
1161: (8)               1-D array of coefficients.
1162: (4)           Returns
1163: (4)
1164: (4)           out : ndarray
1165: (4)               Array of the roots of the series. If all the roots are real,
1166: (4)               then `out` is also real, otherwise it is complex.
1167: (4)
1168: (4)
1169: (4)
1170: (4)
1171: (4)
1172: (4)
1173: (4)
1174: (4)
1175: (4)
1176: (4)
1177: (4)
1178: (4)
1179: (4)
1180: (4)
1181: (4)
1182: (4)
1183: (4)
1184: (4)
1185: (4)
1186: (4)
1187: (4)
1188: (4)
1189: (4)
1190: (4)
1191: (4)
1192: (8)           See Also
1193: (4)
1194: (4)           numpy.polynomial.polynomial.polyroots
1195: (4)           numpy.polynomial.legendre.legroots
1196: (4)           numpy.polynomial.laguerre.lagroots
1197: (4)           numpy.polynomial.hermite.hermroots
1198: (4)           numpy.polynomial.chebyshev.chebroots
1199: (4)
1200: (4)           Notes
1201: (4)
1202: (4)           The root estimates are obtained as the eigenvalues of the companion
1203: (4)           matrix, Roots far from the origin of the complex plane may have large
1204: (4)           errors due to the numerical instability of the series for such
1205: (4)           values. Roots with multiplicity greater than 1 will also show larger
1206: (4)           errors as the value of the series near such points is relatively
1207: (4)           insensitive to errors in the roots. Isolated roots near the origin can
1208: (4)           be improved by a few iterations of Newton's method.
1209: (4)           The HermiteE series basis polynomials aren't powers of `x` so the
1210: (4)           results of this function may seem unintuitive.
1211: (4)
1212: (4)           Examples
1213: (4)
1214: (4)           >>> from numpy.polynomial.hermite_e import hermroots, hermefromroots
1215: (4)           >>> coef = hermefromroots([-1, 0, 1])
1216: (4)           >>> coef
1217: (4)           array([0., 2., 0., 1.])
1218: (4)           >>> hermroots(coef)
1219: (4)           array([-1.,  0.,  1.]) # may vary
1220: (4)
1221: (4)           """
1222: (4)           [c] = pu.as_series([c])
1223: (4)           if len(c) <= 1:
1224: (4)               return np.array([], dtype=c.dtype)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1193: (4)             if len(c) == 2:
1194: (8)                 return np.array([-c[0]/c[1]])
1195: (4)                 m = hermecompanion(c)[::-1, ::-1]
1196: (4)                 r = la.eigvals(m)
1197: (4)                 r.sort()
1198: (4)                 return r
1199: (0)             def _normed_hermite_e_n(x, n):
1200: (4)                 """
1201: (4)                     Evaluate a normalized HermiteE polynomial.
1202: (4)                     Compute the value of the normalized HermiteE polynomial of degree ``n``
1203: (4)                     at the points ``x``.
1204: (4)                     Parameters
1205: (4)                         -----
1206: (4)                         x : ndarray of double.
1207: (8)                             Points at which to evaluate the function
1208: (4)                         n : int
1209: (8)                             Degree of the normalized HermiteE function to be evaluated.
1210: (4)             Returns
1211: (4)                         -----
1212: (4)                         values : ndarray
1213: (8)                             The shape of the return value is described above.
1214: (4)             Notes
1215: (4)                         -----
1216: (4)                         .. versionadded:: 1.10.0
1217: (4)                         This function is needed for finding the Gauss points and integration
1218: (4)                         weights for high degrees. The values of the standard HermiteE functions
1219: (4)                         overflow when n >= 207.
1220: (4)                         """
1221: (4)             if n == 0:
1222: (8)                 return np.full(x.shape, 1/np.sqrt(np.sqrt(2*np.pi)))
1223: (4)             c0 = 0.
1224: (4)             c1 = 1./np.sqrt(np.sqrt(2*np.pi))
1225: (4)             nd = float(n)
1226: (4)             for i in range(n - 1):
1227: (8)                 tmp = c0
1228: (8)                 c0 = -c1*np.sqrt((nd - 1.)/nd)
1229: (8)                 c1 = tmp + c1*x*np.sqrt(1./nd)
1230: (8)                 nd = nd - 1.0
1231: (4)             return c0 + c1*x
1232: (0)             def hermegauss(deg):
1233: (4)                 """
1234: (4)                     Gauss-HermiteE quadrature.
1235: (4)                     Computes the sample points and weights for Gauss-HermiteE quadrature.
1236: (4)                     These sample points and weights will correctly integrate polynomials of
1237: (4)                     degree :math:`2*deg - 1` or less over the interval :math:`[-\infty, \infty]`
1238: (4)                     with the weight function :math:`f(x) = \exp(-x^2/2)`.

1239: (4)                     Parameters
1240: (4)                         -----
1241: (4)                         deg : int
1242: (8)                             Number of sample points and weights. It must be >= 1.
1243: (4)             Returns
1244: (4)                         -----
1245: (4)                         x : ndarray
1246: (8)                             1-D ndarray containing the sample points.
1247: (4)                         y : ndarray
1248: (8)                             1-D ndarray containing the weights.
1249: (4)             Notes
1250: (4)                         -----
1251: (4)                         .. versionadded:: 1.7.0
1252: (4)                         The results have only been tested up to degree 100, higher degrees may
1253: (4)                         be problematic. The weights are determined by using the fact that
1254: (4)                         .. math:: w_k = c / (He'_n(x_k) * He_{n-1}(x_k))
1255: (4)                         where :math:`c` is a constant independent of :math:`k` and :math:`x_k`
1256: (4)                         is the k'th root of :math:`He_n`, and then scaling the results to get
1257: (4)                         the right value when integrating 1.
1258: (4)                         """
1259: (4)                         ideg = pu._deprecate_as_int(deg, "deg")
1260: (4)                         if ideg <= 0:
1261: (8)                             raise ValueError("deg must be a positive integer")

```

```

1262: (4)         c = np.array([0]*deg + [1])
1263: (4)         m = hermecompanion(c)
1264: (4)         x = la.eigvalsh(m)
1265: (4)         dy = _normed_hermite_e_n(x, ideg)
1266: (4)         df = _normed_hermite_e_n(x, ideg - 1) * np.sqrt(ideg)
1267: (4)         x -= dy/df
1268: (4)         fm = _normed_hermite_e_n(x, ideg - 1)
1269: (4)         fm /= np.abs(fm).max()
1270: (4)         w = 1/(fm * fm)
1271: (4)         w = (w + w[::-1])/2
1272: (4)         x = (x - x[::-1])/2
1273: (4)         w *= np.sqrt(2*np.pi) / w.sum()
1274: (4)         return x, w
1275: (0) def hermeweight(x):
1276: (4)     """Weight function of the Hermite_e polynomials.
1277: (4)     The weight function is :math:`\exp(-x^2/2)` and the interval of
1278: (4)     integration is :math:`[-\infty, \infty]`. the HermiteE polynomials are
1279: (4)     orthogonal, but not normalized, with respect to this weight function.
1280: (4)     Parameters
1281: (4)     -----
1282: (4)     x : array_like
1283: (7)         Values at which the weight function will be computed.
1284: (4)     Returns
1285: (4)     -----
1286: (4)     w : ndarray
1287: (7)         The weight function at `x`.
1288: (4)     Notes
1289: (4)     -----
1290: (4)     .. versionadded:: 1.7.0
1291: (4)     """
1292: (4)     w = np.exp(-.5*x**2)
1293: (4)     return w
1294: (0) class HermiteE(ABCPolyBase):
1295: (4)     """An HermiteE series class.
1296: (4)     The HermiteE class provides the standard Python numerical methods
1297: (4)     '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the
1298: (4)     attributes and methods listed in the `ABCPolyBase` documentation.
1299: (4)     Parameters
1300: (4)     -----
1301: (4)     coef : array_like
1302: (8)         HermiteE coefficients in order of increasing degree, i.e.,
1303: (8)         ``(1, 2, 3)`` gives ``1*He_0(x) + 2*He_1(X) + 3*He_2(x)``.
1304: (4)     domain : (2,) array_like, optional
1305: (8)         Domain to use. The interval ``[domain[0], domain[1]]`` is mapped
1306: (8)         to the interval ``[window[0], window[1]]`` by shifting and scaling.
1307: (8)         The default value is [-1, 1].
1308: (4)     window : (2,) array_like, optional
1309: (8)         Window, see `domain` for its use. The default value is [-1, 1].
1310: (8)         .. versionadded:: 1.6.0
1311: (4)     symbol : str, optional
1312: (8)         Symbol used to represent the independent variable in string
1313: (8)         representations of the polynomial expression, e.g. for printing.
1314: (8)         The symbol must be a valid Python identifier. Default value is 'x'.
1315: (8)         .. versionadded:: 1.24
1316: (4)     """
1317: (4)     _add = staticmethod(hermeadd)
1318: (4)     _sub = staticmethod(hermesub)
1319: (4)     _mul = staticmethod(hermemul)
1320: (4)     _div = staticmethod(hermediv)
1321: (4)     _pow = staticmethod(hermepow)
1322: (4)     _val = staticmethod(hermeval)
1323: (4)     _int = staticmethod(hermeint)
1324: (4)     _der = staticmethod(hermeder)
1325: (4)     _fit = staticmethod(hermefit)
1326: (4)     _line = staticmethod(hermeline)
1327: (4)     _roots = staticmethod(hermeroots)
1328: (4)     _fromroots = staticmethod(hermefromroots)
1329: (4)     domain = np.array(hermedomain)
1330: (4)     window = np.array(hermedomain)

```

1331: (4)

basis\_name = 'He'

-----  
File 290 - laguerre.py:

```
1: (0)      """
2: (0)      =====
3: (0)      Laguerre Series (:mod:`numpy.polynomial.laguerre`)
4: (0)      =====
5: (0)      This module provides a number of objects (mostly functions) useful for
6: (0)      dealing with Laguerre series, including a `Laguerre` class that
7: (0)      encapsulates the usual arithmetic operations. (General information
8: (0)      on how this module represents and works with such polynomials is in the
9: (0)      docstring for its "parent" sub-package, `numpy.polynomial`).
10: (0)      Classes
11: (0)      -----
12: (0)      .. autosummary::
13: (3)          :toctree: generated/
14: (3)          Laguerre
15: (0)      Constants
16: (0)      -----
17: (0)      .. autosummary::
18: (3)          :toctree: generated/
19: (3)          lagdomain
20: (3)          lagzero
21: (3)          lagone
22: (3)          lagx
23: (0)      Arithmetic
24: (0)      -----
25: (0)      .. autosummary::
26: (3)          :toctree: generated/
27: (3)          lagadd
28: (3)          lagsub
29: (3)          lagmulx
30: (3)          lagmul
31: (3)          lagdiv
32: (3)          lagpow
33: (3)          lagval
34: (3)          lagval2d
35: (3)          lagval3d
36: (3)          laggrid2d
37: (3)          laggrid3d
38: (0)      Calculus
39: (0)      -----
40: (0)      .. autosummary::
41: (3)          :toctree: generated/
42: (3)          lagder
43: (3)          lagint
44: (0)      Misc Functions
45: (0)      -----
46: (0)      .. autosummary::
47: (3)          :toctree: generated/
48: (3)          lagfromroots
49: (3)          lagroots
50: (3)          lagvander
51: (3)          lagvander2d
52: (3)          lagvander3d
53: (3)          laggauss
54: (3)          lagweight
55: (3)          lagcompanion
56: (3)          lagfit
57: (3)          lagtrim
58: (3)          lagline
59: (3)          lag2poly
60: (3)          poly2lag
61: (0)      See also
62: (0)      -----
63: (0)      `numpy.polynomial`
```

```

64: (0)
65: (0)
66: (0)
67: (0)
68: (0)
69: (0)
70: (0)
71: (4)     """
72: (4)     import numpy as np
73: (4)     import numpy.linalg as la
74: (4)     from numpy.core.multiarray import normalize_axis_index
75: (4)     from . import polyutils as pu
76: (4)     from ._polybase import ABCPolyBase
77: (4)     __all__ = [
78: (4)         'lagzero', 'lagone', 'lagx', 'lagdomain', 'lagline', 'lagadd',
79: (4)         'lagsub', 'lagmulx', 'lagmul', 'lagdiv', 'lagpow', 'lagval', 'lagder',
80: (4)         'lagint', 'lag2poly', 'poly2lag', 'lagfromroots', 'lagvander',
81: (4)         'lagfit', 'lagtrim', 'lagroots', 'Laguerre', 'lagval2d', 'lagval3d',
82: (4)         'laggrid2d', 'laggrid3d', 'lagvander2d', 'lagvander3d', 'lagcompanion',
83: (4)         'laggauss', 'lagweight']
84: (0)     lagtrim = pu.trimcoef
85: (0)     def poly2lag(pol):
86: (4)         """
87: (4)             poly2lag(pol)
88: (4)             Convert a polynomial to a Laguerre series.
89: (4)             Convert an array representing the coefficients of a polynomial (relative
90: (4)                 to the "standard" basis) ordered from lowest degree to highest, to an
91: (4)                 array of the coefficients of the equivalent Laguerre series, ordered
92: (4)                 from lowest to highest degree.
93: (4)             Parameters
94: (4)             -----
95: (4)             pol : array_like
96: (4)                 1-D array containing the polynomial coefficients
97: (4)             Returns
98: (4)             -----
99: (4)             c : ndarray
100: (8)                 1-D array containing the coefficients of the equivalent Laguerre
101: (8)                 series.
102: (4)             See Also
103: (4)             -----
104: (4)             lag2poly
105: (4)             Notes
106: (4)             -----
107: (4)             The easy way to do conversions between polynomial basis sets
108: (4)                 is to use the convert method of a class instance.
109: (4)             Examples
110: (4)             -----
111: (4)             >>> from numpy.polynomial.laguerre import poly2lag
112: (4)             >>> poly2lag(np.arange(4))
113: (4)             array([ 23., -63.,  58., -18.])
114: (4)             """
115: (0)             [pol] = pu.as_series([pol])
116: (0)             res = 0
117: (0)             for p in pol[::-1]:
118: (8)                 res = lagadd(lagmulx(res), p)
119: (4)             return res
120: (0)             def lag2poly(c):
121: (4)                 """
122: (4)                     Convert a Laguerre series to a polynomial.
123: (4)                     Convert an array representing the coefficients of a Laguerre series,
124: (4)                         ordered from lowest degree to highest, to an array of the coefficients
125: (4)                         of the equivalent polynomial (relative to the "standard" basis) ordered
126: (4)                         from lowest to highest degree.
127: (4)                     Parameters
128: (4)                     -----
129: (4)                     c : array_like
130: (8)                         1-D array containing the Laguerre series coefficients, ordered
131: (8)                             from lowest order term to highest.
132: (4)                     Returns
133: (4)                     -----
134: (4)                     pol : ndarray
135: (8)                         1-D array containing the coefficients of the equivalent polynomial
136: (8)                             (relative to the "standard" basis) ordered from lowest order term
137: (8)                             to highest.
138: (4)                     See Also
139: (4)                     -----

```

```

133: (4)          poly2lag
134: (4)          Notes
135: (4)
136: (4)          -----
137: (4)          The easy way to do conversions between polynomial basis sets
138: (4)          is to use the convert method of a class instance.
139: (4)          Examples
140: (4)          -----
141: (4)          >>> from numpy.polynomial.laguerre import lag2poly
142: (4)          >>> lag2poly([ 23., -63., 58., -18.])
143: (4)          array([0., 1., 2., 3.])
144: (4)          """
145: (4)          from .polynomial import polyadd, polysub, polymulx
146: (4)          [c] = pu.as_series([c])
147: (4)          n = len(c)
148: (8)          if n == 1:
149: (4)              return c
150: (8)          else:
151: (8)              c0 = c[-2]
152: (8)              c1 = c[-1]
153: (12)             for i in range(n - 1, 1, -1):
154: (12)                 tmp = c0
155: (12)                 c0 = polysub(c[i - 2], (c1*(i - 1))/i)
156: (8)                 c1 = polyadd(tmp, polysub((2*i - 1)*c1, polymulx(c1))/i)
157: (0)                 return polyadd(c0, polysub(c1, polymulx(c1)))
158: (0)             lagdomain = np.array([0, 1])
159: (0)             lagzero = np.array([0])
160: (0)             lagone = np.array([1])
161: (0)             lagx = np.array([1, -1])
162: (0)             def lagline(off, scl):
163: (4)                 """
164: (4)                 Laguerre series whose graph is a straight line.
165: (4)                 Parameters
166: (4)                 -----
167: (8)                 off, scl : scalars
168: (4)                 The specified line is given by ``off + scl*x``.
169: (4)                 Returns
170: (4)                 -----
171: (8)                 y : ndarray
172: (8)                 This module's representation of the Laguerre series for
173: (4)                 ``off + scl*x``.
174: (4)                 See Also
175: (4)                 -----
176: (4)                 numpy.polynomial.polynomial.polyline
177: (4)                 numpy.polynomial.chebyshev.chebline
178: (4)                 numpy.polynomial.legendre.legline
179: (4)                 numpy.polynomial.hermite.hermeline
180: (4)                 numpy.polynomial.hermite_e.hermeline
181: (4)                 Examples
182: (4)                 -----
183: (4)                 >>> from numpy.polynomial.laguerre import lagline, lagval
184: (4)                 >>> lagval(0,lagline(3, 2))
185: (4)                 3.0
186: (4)                 >>> lagval(1,lagline(3, 2))
187: (4)                 5.0
188: (4)                 """
189: (8)                 if scl != 0:
190: (4)                     return np.array([off + scl, -scl])
191: (8)                 else:
192: (0)                     return np.array([off])
193: (4)             def lagfromroots(roots):
194: (4)                 """
195: (4)                 Generate a Laguerre series with given roots.
196: (4)                 The function returns the coefficients of the polynomial
197: (4)                 .. math:: p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),
198: (4)                 in Laguerre form, where the `r_n` are the roots specified in `roots`.
199: (4)                 If a zero has multiplicity n, then it must appear in `roots` n times.
200: (4)                 For instance, if 2 is a root of multiplicity three and 3 is a root of
201: (4)                 multiplicity 2, then `roots` looks something like [2, 2, 2, 3, 3]. The
201: (4)                 roots can appear in any order.

```

```

202: (4)           If the returned coefficients are `c`, then
203: (4)           .. math:: p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x)
204: (4)           The coefficient of the last term is not generally 1 for monic
205: (4)           polynomials in Laguerre form.
206: (4)           Parameters
207: (4)           -----
208: (4)           roots : array_like
209: (8)             Sequence containing the roots.
210: (4)           Returns
211: (4)           -----
212: (4)           out : ndarray
213: (8)             1-D array of coefficients. If all roots are real then `out` is a
214: (8)             real array, if some of the roots are complex, then `out` is complex
215: (8)             even if all the coefficients in the result are real (see Examples
216: (8)             below).
217: (4)           See Also
218: (4)           -----
219: (4)           numpy.polynomial.polynomial.polyfromroots
220: (4)           numpy.polynomial.legendre.legfromroots
221: (4)           numpy.polynomial.chebyshev.chebfromroots
222: (4)           numpy.polynomial.hermite.hermfromroots
223: (4)           numpy.polynomial.hermite_e.hermefromroots
224: (4)           Examples
225: (4)           -----
226: (4)           >>> from numpy.polynomial.laguerre import lagfromroots, lagval
227: (4)           >>> coef = lagfromroots((-1, 0, 1))
228: (4)           >>> lagval((-1, 0, 1), coef)
229: (4)           array([0., 0., 0.])
230: (4)           >>> coef = lagfromroots((-1j, 1j))
231: (4)           >>> lagval((-1j, 1j), coef)
232: (4)           array([0.+0.j, 0.+0.j])
233: (4)           """
234: (4)           return pu._fromroots(lagline, lagmul, roots)
235: (0)           def lagadd(c1, c2):
236: (4)             """
237: (4)             Add one Laguerre series to another.
238: (4)             Returns the sum of two Laguerre series `c1` + `c2`. The arguments
239: (4)             are sequences of coefficients ordered from lowest order term to
240: (4)             highest, i.e., [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
241: (4)             Parameters
242: (4)             -----
243: (4)             c1, c2 : array_like
244: (8)               1-D arrays of Laguerre series coefficients ordered from low to
245: (8)               high.
246: (4)             Returns
247: (4)             -----
248: (4)             out : ndarray
249: (8)               Array representing the Laguerre series of their sum.
250: (4)             See Also
251: (4)             -----
252: (4)             lagsub, lagmulx, lagmul, lagdiv, lagpow
253: (4)             Notes
254: (4)             -----
255: (4)             Unlike multiplication, division, etc., the sum of two Laguerre series
256: (4)             is a Laguerre series (without having to "reproject" the result onto
257: (4)             the basis set) so addition, just like that of "standard" polynomials,
258: (4)             is simply "component-wise."
259: (4)             Examples
260: (4)             -----
261: (4)             >>> from numpy.polynomial.laguerre import lagadd
262: (4)             >>> lagadd([1, 2, 3], [1, 2, 3, 4])
263: (4)             array([2., 4., 6., 4.])
264: (4)             """
265: (4)             return pu._add(c1, c2)
266: (0)           def lagsub(c1, c2):
267: (4)             """
268: (4)             Subtract one Laguerre series from another.
269: (4)             Returns the difference of two Laguerre series `c1` - `c2`. The
270: (4)             sequences of coefficients are from lowest order term to highest, i.e.,

```

```

271: (4) [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
272: (4) Parameters
273: (4)
274: (4) c1, c2 : array_like
275: (8) 1-D arrays of Laguerre series coefficients ordered from low to
276: (8) high.
277: (4) Returns
278: (4)
279: (4) out : ndarray
280: (8) Of Laguerre series coefficients representing their difference.
281: (4) See Also
282: (4)
283: (4) lagadd, lagmulx, lagmul, lagdiv, lagpow
284: (4) Notes
285: (4)
286: (4) Unlike multiplication, division, etc., the difference of two Laguerre
287: (4) series is a Laguerre series (without having to "reproject" the result
288: (4) onto the basis set) so subtraction, just like that of "standard"
289: (4) polynomials, is simply "component-wise."
290: (4) Examples
291: (4)
292: (4) >>> from numpy.polynomial.laguerre import lagsub
293: (4) >>> lagsub([1, 2, 3, 4], [1, 2, 3])
294: (4) array([0.,  0.,  0.,  4.])
295: (4) """
296: (4)     return pu._sub(c1, c2)
297: (0) def lagmulx(c):
298: (4) """Multiply a Laguerre series by x.
299: (4) Multiply the Laguerre series `c` by x, where x is the independent
300: (4) variable.
301: (4) Parameters
302: (4)
303: (4) c : array_like
304: (8) 1-D array of Laguerre series coefficients ordered from low to
305: (8) high.
306: (4) Returns
307: (4)
308: (4) out : ndarray
309: (8) Array representing the result of the multiplication.
310: (4) See Also
311: (4)
312: (4) lagadd, lagsub, lagmul, lagdiv, lagpow
313: (4) Notes
314: (4)
315: (4) The multiplication uses the recursion relationship for Laguerre
316: (4) polynomials in the form
317: (4) .. math::
318: (8) xP_i(x) = (-(i + 1)*P_{i + 1}(x) + (2i + 1)P_{i}(x) - iP_{i - 1}(x))
319: (4) Examples
320: (4)
321: (4) >>> from numpy.polynomial.laguerre import lagmulx
322: (4) >>> lagmulx([1, 2, 3])
323: (4) array([-1., -1., 11., -9.])
324: (4) """
325: (4) [c] = pu.as_series([c])
326: (4) if len(c) == 1 and c[0] == 0:
327: (8)     return c
328: (4) prd = np.empty(len(c) + 1, dtype=c.dtype)
329: (4) prd[0] = c[0]
330: (4) prd[1] = -c[0]
331: (4) for i in range(1, len(c)):
332: (8)     prd[i + 1] = -c[i]*(i + 1)
333: (8)     prd[i] += c[i]*(2*i + 1)
334: (8)     prd[i - 1] -= c[i]*i
335: (4) return prd
336: (0) def lagmul(c1, c2):
337: (4) """
338: (4) Multiply one Laguerre series by another.
339: (4) Returns the product of two Laguerre series `c1` * `c2`. The arguments

```

```

340: (4)             are sequences of coefficients, from lowest order "term" to highest,
341: (4)             e.g., [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
342: (4)             Parameters
343: (4)
344: (4)             c1, c2 : array_like
345: (8)             1-D arrays of Laguerre series coefficients ordered from low to
346: (8)             high.
347: (4)             Returns
348: (4)
349: (4)             out : ndarray
350: (8)             Of Laguerre series coefficients representing their product.
351: (4)             See Also
352: (4)
353: (4)             lagadd, lagsub, lagmulx, lagdiv, lagpow
354: (4)
355: (4)
356: (4)             In general, the (polynomial) product of two C-series results in terms
357: (4)             that are not in the Laguerre polynomial basis set. Thus, to express
358: (4)             the product as a Laguerre series, it is necessary to "reproject" the
359: (4)             product onto said basis set, which may produce "unintuitive" (but
360: (4)             correct) results; see Examples section below.
361: (4)             Examples
362: (4)
363: (4)             >>> from numpy.polynomial.laguerre import lagmul
364: (4)             >>> lagmul([1, 2, 3], [0, 1, 2])
365: (4)             array([ 8., -13., 38., -51., 36.])
366: (4)
367: (4)             [c1, c2] = pu.as_series([c1, c2])
368: (4)             if len(c1) > len(c2):
369: (8)                 c = c2
370: (8)                 xs = c1
371: (4)             else:
372: (8)                 c = c1
373: (8)                 xs = c2
374: (4)             if len(c) == 1:
375: (8)                 c0 = c[0]*xs
376: (8)                 c1 = 0
377: (4)             elif len(c) == 2:
378: (8)                 c0 = c[0]*xs
379: (8)                 c1 = c[1]*xs
380: (4)             else:
381: (8)                 nd = len(c)
382: (8)                 c0 = c[-2]*xs
383: (8)                 c1 = c[-1]*xs
384: (8)                 for i in range(3, len(c) + 1):
385: (12)                     tmp = c0
386: (12)                     nd = nd - 1
387: (12)                     c0 = lagsub(c[-i]*xs, (c1*(nd - 1))/nd)
388: (12)                     c1 = lagadd(tmp, lagsub((2*nd - 1)*c1, lagmulx(c1))/nd)
389: (4)             return lagadd(c0, lagsub(c1, lagmulx(c1)))
390: (0)             def lagdiv(c1, c2):
391: (4)                 """
392: (4)                 Divide one Laguerre series by another.
393: (4)                 Returns the quotient-with-remainder of two Laguerre series
394: (4)                 `c1` / `c2`. The arguments are sequences of coefficients from lowest
395: (4)                 order "term" to highest, e.g., [1,2,3] represents the series
396: (4)                 ``P_0 + 2*P_1 + 3*P_2``.
397: (4)                 Parameters
398: (4)
399: (4)                 c1, c2 : array_like
400: (8)                 1-D arrays of Laguerre series coefficients ordered from low to
401: (8)                 high.
402: (4)                 Returns
403: (4)
404: (4)                 [quo, rem] : ndarrays
405: (8)                 Of Laguerre series coefficients representing the quotient and
406: (8)                 remainder.
407: (4)                 See Also
408: (4)

```

```

409: (4)                                     lagadd, lagsub, lagmulx, lagmul, lagpow
410: (4)                                     Notes
411: (4)                                     -----
412: (4)                                     In general, the (polynomial) division of one Laguerre series by another
413: (4)                                     results in quotient and remainder terms that are not in the Laguerre
414: (4)                                     polynomial basis set. Thus, to express these results as a Laguerre
415: (4)                                     series, it is necessary to "reproject" the results onto the Laguerre
416: (4)                                     basis set, which may produce "unintuitive" (but correct) results; see
417: (4)                                     Examples section below.
418: (4)                                     Examples
419: (4)                                     -----
420: (4)                                     >>> from numpy.polynomial.laguerre import lagdiv
421: (4)                                     >>> lagdiv([ 8., -13., 38., -51., 36.], [0, 1, 2])
422: (4)                                     (array([1., 2., 3.]), array([0.]))
423: (4)                                     >>> lagdiv([ 9., -12., 38., -51., 36.], [0, 1, 2])
424: (4)                                     (array([1., 2., 3.]), array([1., 1.]))
425: (4)                                     """
426: (4)                                     return pu._div(lagmul, c1, c2)
427: (0) def lagpow(c, pow, maxpower=16):
428: (4)                                     """Raise a Laguerre series to a power.
429: (4)                                     Returns the Laguerre series `c` raised to the power `pow`. The
430: (4)                                     argument `c` is a sequence of coefficients ordered from low to high.
431: (4)                                     i.e., [1,2,3] is the series ``P_0 + 2*P_1 + 3*P_2.```
432: (4)                                     Parameters
433: (4)                                     -----
434: (4)                                     c : array_like
435: (8)                                     1-D array of Laguerre series coefficients ordered from low to
436: (8)                                     high.
437: (4)                                     pow : integer
438: (8)                                     Power to which the series will be raised
439: (4)                                     maxpower : integer, optional
440: (8)                                     Maximum power allowed. This is mainly to limit growth of the series
441: (8)                                     to unmanageable size. Default is 16
442: (4)                                     Returns
443: (4)                                     -----
444: (4)                                     coef : ndarray
445: (8)                                     Laguerre series of power.
446: (4)                                     See Also
447: (4)                                     -----
448: (4)                                     lagadd, lagsub, lagmulx, lagmul, lagdiv
449: (4)                                     Examples
450: (4)                                     -----
451: (4)                                     >>> from numpy.polynomial.laguerre import lagpow
452: (4)                                     >>> lagpow([1, 2, 3], 2)
453: (4)                                     array([ 14., -16., 56., -72., 54.])
454: (4)                                     """
455: (4)                                     return pu._pow(lagmul, c, pow, maxpower)
456: (0) def lagder(c, m=1, scl=1, axis=0):
457: (4)                                     """
458: (4)                                     Differentiate a Laguerre series.
459: (4)                                     Returns the Laguerre series coefficients `c` differentiated `m` times
460: (4)                                     along `axis`. At each iteration the result is multiplied by `scl` (the
461: (4)                                     scaling factor is for use in a linear change of variable). The argument
462: (4)                                     `c` is an array of coefficients from low to high degree along each
463: (4)                                     axis, e.g., [1,2,3] represents the series ``1*L_0 + 2*L_1 + 3*L_2``
464: (4)                                     while [[1,2],[1,2]] represents ``1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) +
465: (4)                                     2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)`` if axis=0 is ``x`` and axis=1 is
466: (4)                                     ``y``.
467: (4)                                     Parameters
468: (4)                                     -----
469: (4)                                     c : array_like
470: (8)                                     Array of Laguerre series coefficients. If `c` is multidimensional
471: (8)                                     the different axis correspond to different variables with the
472: (8)                                     degree in each axis given by the corresponding index.
473: (4)                                     m : int, optional
474: (8)                                     Number of derivatives taken, must be non-negative. (Default: 1)
475: (4)                                     scl : scalar, optional
476: (8)                                     Each differentiation is multiplied by `scl`. The end result is
477: (8)                                     multiplication by ``scl**m``. This is for use in a linear change of

```

```

478: (8)                         variable. (Default: 1)
479: (4)                         axis : int, optional
480: (8)                         Axis over which the derivative is taken. (Default: 0).
481: (8)                         .. versionadded:: 1.7.0
482: (4)                         Returns
483: (4)                         -----
484: (4)                         der : ndarray
485: (8)                         Laguerre series of the derivative.
486: (4)                         See Also
487: (4)                         -----
488: (4)                         lagint
489: (4)                         Notes
490: (4)                         -----
491: (4)                         In general, the result of differentiating a Laguerre series does not
492: (4)                         resemble the same operation on a power series. Thus the result of this
493: (4)                         function may be "unintuitive," albeit correct; see Examples section
494: (4)                         below.
495: (4)                         Examples
496: (4)                         -----
497: (4)                         >>> from numpy.polynomial.laguerre import lagder
498: (4)                         >>> lagder([ 1.,  1.,  1., -3.])
499: (4)                         array([1.,  2.,  3.])
500: (4)                         >>> lagder([ 1.,  0.,  0., -4.,  3.], m=2)
501: (4)                         array([1.,  2.,  3.])
502: (4)                         """
503: (4)                         c = np.array(c, ndmin=1, copy=True)
504: (4)                         if c.dtype.char in '?bBhHiIlLqQpP':
505: (8)                           c = c.astype(np.double)
506: (4)                         cnt = pu._deprecate_as_int(m, "the order of derivation")
507: (4)                         iaxis = pu._deprecate_as_int(axis, "the axis")
508: (4)                         if cnt < 0:
509: (8)                           raise ValueError("The order of derivation must be non-negative")
510: (4)                         iaxis = normalize_axis_index(iaxis, c.ndim)
511: (4)                         if cnt == 0:
512: (8)                           return c
513: (4)                         c = np.moveaxis(c, iaxis, 0)
514: (4)                         n = len(c)
515: (4)                         if cnt >= n:
516: (8)                           c = c[:1]*0
517: (4)                         else:
518: (8)                           for i in range(cnt):
519: (12)                             n = n - 1
520: (12)                             c *= scl
521: (12)                             der = np.empty((n,) + c.shape[1:], dtype=c.dtype)
522: (12)                             for j in range(n, 1, -1):
523: (16)                               der[j - 1] = -c[j]
524: (16)                               c[j - 1] += c[j]
525: (12)                               der[0] = -c[1]
526: (12)                               c = der
527: (4)                         c = np.moveaxis(c, 0, iaxis)
528: (4)                         return c
529: (0)                         def lagint(c, m=1, k=[], lbnd=0, scl=1, axis=0):
530: (4)                         """
531: (4)                         Integrate a Laguerre series.
532: (4)                         Returns the Laguerre series coefficients `c` integrated `m` times from
533: (4)                         `lbnd` along `axis`. At each iteration the resulting series is
534: (4)                         **multiplied** by `scl` and an integration constant, `k`, is added.
535: (4)                         The scaling factor is for use in a linear change of variable. ("Buyer
536: (4)                         beware": note that, depending on what one is doing, one may want `scl`
537: (4)                         to be the reciprocal of what one might expect; for more information,
538: (4)                         see the Notes section below.) The argument `c` is an array of
539: (4)                         coefficients from low to high degree along each axis, e.g., [1,2,3]
540: (4)                         represents the series `` $L_0 + 2*L_1 + 3*L_2`` while [[1,2],[1,2]]
541: (4)                         represents `` $1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) +$ 
542: (4)                          $2*L_1(x)*L_1(y)$ `` if axis=0 is ``x`` and axis=1 is ``y``.
543: (4)                         Parameters
544: (4)                         -----
545: (4)                         c : array_like
546: (8)                           Array of Laguerre series coefficients. If `c` is multidimensional$ 
```

```

547: (8)          the different axis correspond to different variables with the
548: (8)          degree in each axis given by the corresponding index.
549: (4)          m : int, optional
550: (8)          Order of integration, must be positive. (Default: 1)
551: (4)          k : {}, list, scalar}, optional
552: (8)          Integration constant(s). The value of the first integral at
553: (8)          ``lbnd`` is the first value in the list, the value of the second
554: (8)          integral at ``lbnd`` is the second value, etc. If ``k == []`` (the
555: (8)          default), all constants are set to zero. If ``m == 1``, a single
556: (8)          scalar can be given instead of a list.
557: (4)          lbnd : scalar, optional
558: (8)          The lower bound of the integral. (Default: 0)
559: (4)          scl : scalar, optional
560: (8)          Following each integration the result is *multiplied* by `scl`
561: (8)          before the integration constant is added. (Default: 1)
562: (4)          axis : int, optional
563: (8)          Axis over which the integral is taken. (Default: 0).
564: (8)          .. versionadded:: 1.7.0
565: (4)          Returns
566: (4)          -----
567: (4)          S : ndarray
568: (8)          Laguerre series coefficients of the integral.
569: (4)          Raises
570: (4)          -----
571: (4)          ValueError
572: (8)          If ``m < 0``, ``len(k) > m``, ``np.ndim(lbnd) != 0``, or
573: (8)          ``np.ndim(scl) != 0``.
574: (4)          See Also
575: (4)          -----
576: (4)          lagder
577: (4)          Notes
578: (4)          -----
579: (4)          Note that the result of each integration is *multiplied* by `scl`.
580: (4)          Why is this important to note? Say one is making a linear change of
581: (4)          variable :math:`u = ax + b` in an integral relative to `x`. Then
582: (4)          :math:`dx = du/a`, so one will need to set `scl` equal to
583: (4)          :math:`1/a` - perhaps not what one would have first thought.
584: (4)          Also note that, in general, the result of integrating a C-series needs
585: (4)          to be "reprojected" onto the C-series basis set. Thus, typically,
586: (4)          the result of this function is "unintuitive," albeit correct; see
587: (4)          Examples section below.
588: (4)          Examples
589: (4)          -----
590: (4)          >>> from numpy.polynomial.laguerre import lagint
591: (4)          >>> lagint([1,2,3])
592: (4)          array([ 1.,  1.,  1., -3.])
593: (4)          >>> lagint([1,2,3], m=2)
594: (4)          array([ 1.,  0.,  0., -4.,  3.])
595: (4)          >>> lagint([1,2,3], k=1)
596: (4)          array([ 2.,  1.,  1., -3.])
597: (4)          >>> lagint([1,2,3], lbnd=-1)
598: (4)          array([11.5,  1. ,  1. , -3. ])
599: (4)          >>> lagint([1,2], m=2, k=[1,2], lbnd=-1)
600: (4)          array([ 11.16666667, -5.           , -3.           ,  2.           ]) # may vary
601: (4)          """
602: (4)          c = np.array(c, ndmin=1, copy=True)
603: (4)          if c.dtype.char in '?bBhHiIlLqQpP':
604: (8)              c = c.astype(np.double)
605: (4)          if not np.iterable(k):
606: (8)              k = [k]
607: (4)          cnt = pu._deprecate_as_int(m, "the order of integration")
608: (4)          iaxis = pu._deprecate_as_int(axis, "the axis")
609: (4)          if cnt < 0:
610: (8)              raise ValueError("The order of integration must be non-negative")
611: (4)          if len(k) > cnt:
612: (8)              raise ValueError("Too many integration constants")
613: (4)          if np.ndim(lbnd) != 0:
614: (8)              raise ValueError("lbnd must be a scalar.")
615: (4)          if np.ndim(scl) != 0:

```

```

616: (8)             raise ValueError("scl must be a scalar.")
617: (4)             iaxis = normalize_axis_index(iaxis, c.ndim)
618: (4)             if cnt == 0:
619: (8)                 return c
620: (4)             c = np.moveaxis(c, iaxis, 0)
621: (4)             k = list(k) + [0]*(cnt - len(k))
622: (4)             for i in range(cnt):
623: (8)                 n = len(c)
624: (8)                 c *= scl
625: (8)                 if n == 1 and np.all(c[0] == 0):
626: (12)                     c[0] += k[i]
627: (8)                 else:
628: (12)                     tmp = np.empty((n + 1,) + c.shape[1:], dtype=c.dtype)
629: (12)                     tmp[0] = c[0]
630: (12)                     tmp[1] = -c[0]
631: (12)                     for j in range(1, n):
632: (16)                         tmp[j] += c[j]
633: (16)                         tmp[j + 1] = -c[j]
634: (12)                     tmp[0] += k[i] - lagval(lbnd, tmp)
635: (12)                     c = tmp
636: (4)             c = np.moveaxis(c, 0, iaxis)
637: (4)             return c
638: (0)             def lagval(x, c, tensor=True):
639: (4)                 """
640: (4)                 Evaluate a Laguerre series at points x.
641: (4)                 If `c` is of length `n + 1`, this function returns the value:
642: (4)                 .. math:: p(x) = c_0 * L_0(x) + c_1 * L_1(x) + \dots + c_n * L_n(x)
643: (4)                 The parameter `x` is converted to an array only if it is a tuple or a
644: (4)                 list, otherwise it is treated as a scalar. In either case, either `x`
645: (4)                 or its elements must support multiplication and addition both with
646: (4)                 themselves and with the elements of `c`.
647: (4)                 If `c` is a 1-D array, then `p(x)` will have the same shape as `x`. If
648: (4)                 `c` is multidimensional, then the shape of the result depends on the
649: (4)                 value of `tensor`. If `tensor` is true the shape will be `c.shape[1:] + x.shape`. If `tensor` is false the shape will be `c.shape[1:]`. Note that
650: (4)                 scalars have shape ().
651: (4)                 Trailing zeros in the coefficients will be used in the evaluation, so
652: (4)                 they should be avoided if efficiency is a concern.
653: (4)             Parameters
654: (4)             -----
655: (4)             x : array_like, compatible object
656: (4)                 If `x` is a list or tuple, it is converted to an ndarray, otherwise
657: (8)                 it is left unchanged and treated as a scalar. In either case, `x`
658: (8)                 or its elements must support addition and multiplication with
659: (8)                 themselves and with the elements of `c`.
660: (8)             c : array_like
661: (4)                 Array of coefficients ordered so that the coefficients for terms of
662: (8)                 degree n are contained in c[n]. If `c` is multidimensional the
663: (8)                 remaining indices enumerate multiple polynomials. In the two
664: (8)                 dimensional case the coefficients may be thought of as stored in
665: (8)                 the columns of `c`.
666: (8)             tensor : boolean, optional
667: (4)                 If True, the shape of the coefficient array is extended with ones
668: (8)                 on the right, one for each dimension of `x`. Scalars have dimension 0
669: (8)                 for this action. The result is that every column of coefficients in
670: (8)                 `c` is evaluated for every element of `x`. If False, `x` is broadcast
671: (8)                 over the columns of `c` for the evaluation. This keyword is useful
672: (8)                 when `c` is multidimensional. The default value is True.
673: (8)                 .. versionadded:: 1.7.0
674: (8)             Returns
675: (4)             -----
676: (4)             values : ndarray, algebra_like
677: (4)                 The shape of the return value is described above.
678: (8)             See Also
679: (4)             -----
680: (4)             lagval2d, laggrid2d, lagval3d, laggrid3d
681: (4)             Notes
682: (4)             -----
683: (4)             The evaluation uses Clenshaw recursion, aka synthetic division.
684: (4)

```

```

685: (4) Examples
686: (4) -----
687: (4) >>> from numpy.polynomial.laguerre import lagval
688: (4) >>> coef = [1,2,3]
689: (4) >>> lagval(1, coef)
690: (4) -0.5
691: (4) >>> lagval([[1,2],[3,4]], coef)
692: (4) array([[-0.5, -4. ],
693: (11)      [-4.5, -2. ]])
694: (4) """
695: (4) c = np.array(c, ndmin=1, copy=False)
696: (4) if c.dtype.char in '?bBhHiIlLqQpP':
697: (8)     c = c.astype(np.double)
698: (4) if isinstance(x, (tuple, list)):
699: (8)     x = np.asarray(x)
700: (4) if isinstance(x, np.ndarray) and tensor:
701: (8)     c = c.reshape(c.shape + (1,)*x.ndim)
702: (4) if len(c) == 1:
703: (8)     c0 = c[0]
704: (8)     c1 = 0
705: (4) elif len(c) == 2:
706: (8)     c0 = c[0]
707: (8)     c1 = c[1]
708: (4) else:
709: (8)     nd = len(c)
710: (8)     c0 = c[-2]
711: (8)     c1 = c[-1]
712: (8)     for i in range(3, len(c) + 1):
713: (12)         tmp = c0
714: (12)         nd = nd - 1
715: (12)         c0 = c[-i] - (c1*(nd - 1))/nd
716: (12)         c1 = tmp + (c1*((2*nd - 1) - x))/nd
717: (4)     return c0 + c1*(1 - x)
718: (0) def lagval2d(x, y, c):
719: (4) """
720: (4) Evaluate a 2-D Laguerre series at points (x, y).
721: (4) This function returns the values:
722: (4) .. math:: p(x,y) = \sum_{i,j} c_{i,j} * L_i(x) * L_j(y)
723: (4) The parameters `x` and `y` are converted to arrays only if they are
724: (4) tuples or a lists, otherwise they are treated as a scalars and they
725: (4) must have the same shape after conversion. In either case, either `x`
726: (4) and `y` or their elements must support multiplication and addition both
727: (4) with themselves and with the elements of `c`.
728: (4) If `c` is a 1-D array a one is implicitly appended to its shape to make
729: (4) it 2-D. The shape of the result will be c.shape[2:] + x.shape.
730: (4) Parameters
731: (4) -----
732: (4) x, y : array_like, compatible objects
733: (8) The two dimensional series is evaluated at the points `(x, y)`,
734: (8) where `x` and `y` must have the same shape. If `x` or `y` is a list
735: (8) or tuple, it is first converted to an ndarray, otherwise it is left
736: (8) unchanged and if it isn't an ndarray it is treated as a scalar.
737: (4) c : array_like
738: (8) Array of coefficients ordered so that the coefficient of the term
739: (8) of multi-degree  $i, j$  is contained in ``c[i,j]``. If `c` has
740: (8) dimension greater than two the remaining indices enumerate multiple
741: (8) sets of coefficients.
742: (4) Returns
743: (4) -----
744: (4) values : ndarray, compatible object
745: (8) The values of the two dimensional polynomial at points formed with
746: (8) pairs of corresponding values from `x` and `y`.
747: (4) See Also
748: (4) -----
749: (4) lagval, laggrid2d, lagval3d, laggrid3d
750: (4) Notes
751: (4) -----
752: (4) .. versionadded:: 1.7.0
753: (4) """

```

```

754: (4)             return pu._valnd(lagval, c, x, y)
755: (0)             def laggrid2d(x, y, c):
756: (4)                 """
757: (4)                     Evaluate a 2-D Laguerre series on the Cartesian product of x and y.
758: (4)                     This function returns the values:
759: (4)                     .. math:: p(a,b) = \sum_{i,j} c_{i,j} * L_i(a) * L_j(b)
760: (4)                     where the points `(a, b)` consist of all pairs formed by taking
761: (4)                     `a` from `x` and `b` from `y`. The resulting points form a grid with
762: (4)                     `x` in the first dimension and `y` in the second.
763: (4)                     The parameters `x` and `y` are converted to arrays only if they are
764: (4)                     tuples or a lists, otherwise they are treated as a scalars. In either
765: (4)                     case, either `x` and `y` or their elements must support multiplication
766: (4)                     and addition both with themselves and with the elements of `c`.
767: (4)                     If `c` has fewer than two dimensions, ones are implicitly appended to
768: (4)                     its shape to make it 2-D. The shape of the result will be c.shape[2:] +
769: (4)                     x.shape + y.shape.
770: (4)             Parameters
771: (4)                 -----
772: (4)                 x, y : array_like, compatible objects
773: (8)                     The two dimensional series is evaluated at the points in the
774: (8)                     Cartesian product of `x` and `y`. If `x` or `y` is a list or
775: (8)                     tuple, it is first converted to an ndarray, otherwise it is left
776: (8)                     unchanged and, if it isn't an ndarray, it is treated as a scalar.
777: (4)                 c : array_like
778: (8)                     Array of coefficients ordered so that the coefficient of the term of
779: (8)                     multi-degree i,j is contained in `c[i,j]`. If `c` has dimension
780: (8)                     greater than two the remaining indices enumerate multiple sets of
781: (8)                     coefficients.
782: (4)             Returns
783: (4)                 -----
784: (4)                 values : ndarray, compatible object
785: (8)                     The values of the two dimensional Chebyshev series at points in the
786: (8)                     Cartesian product of `x` and `y`.
787: (4)             See Also
788: (4)                 -----
789: (4)                 lagval, lagval2d, lagval3d, laggrid3d
790: (4)             Notes
791: (4)                 -----
792: (4)                 .. versionadded:: 1.7.0
793: (4)                 """
794: (4)             return pu._gridnd(lagval, c, x, y)
795: (0)             def lagval3d(x, y, z, c):
796: (4)                 """
797: (4)                     Evaluate a 3-D Laguerre series at points (x, y, z).
798: (4)                     This function returns the values:
799: (4)                     .. math:: p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * L_i(x) * L_j(y) * L_k(z)
800: (4)                     The parameters `x`, `y`, and `z` are converted to arrays only if
801: (4)                     they are tuples or a lists, otherwise they are treated as a scalars and
802: (4)                     they must have the same shape after conversion. In either case, either
803: (4)                     `x`, `y`, and `z` or their elements must support multiplication and
804: (4)                     addition both with themselves and with the elements of `c`.
805: (4)                     If `c` has fewer than 3 dimensions, ones are implicitly appended to its
806: (4)                     shape to make it 3-D. The shape of the result will be c.shape[3:] +
807: (4)                     x.shape.
808: (4)             Parameters
809: (4)                 -----
810: (4)                 x, y, z : array_like, compatible object
811: (8)                     The three dimensional series is evaluated at the points
812: (8)                     `(x, y, z)`, where `x`, `y`, and `z` must have the same shape. If
813: (8)                     any of `x`, `y`, or `z` is a list or tuple, it is first converted
814: (8)                     to an ndarray, otherwise it is left unchanged and if it isn't an
815: (8)                     ndarray it is treated as a scalar.
816: (4)                 c : array_like
817: (8)                     Array of coefficients ordered so that the coefficient of the term of
818: (8)                     multi-degree i,j,k is contained in `c[i,j,k]`. If `c` has dimension
819: (8)                     greater than 3 the remaining indices enumerate multiple sets of
820: (8)                     coefficients.
821: (4)             Returns
822: (4)                 -----

```

```

823: (4)             values : ndarray, compatible object
824: (8)             The values of the multidimensional polynomial on points formed with
825: (8)             triples of corresponding values from `x`, `y`, and `z`.
826: (4)             See Also
827: (4)             -----
828: (4)             lagval, lagval2d, laggrid2d, laggrid3d
829: (4)             Notes
830: (4)             -----
831: (4)             .. versionadded:: 1.7.0
832: (4)             """
833: (4)             return pu._valnd(lagval, c, x, y, z)
834: (0) def laggrid3d(x, y, z, c):
835: (4)             """
836: (4)             Evaluate a 3-D Laguerre series on the Cartesian product of x, y, and z.
837: (4)             This function returns the values:
838: (4)             .. math:: p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * L_i(a) * L_j(b) * L_k(c)
839: (4)             where the points `(a, b, c)` consist of all triples formed by taking
840: (4)             `a` from `x`, `b` from `y`, and `c` from `z`. The resulting points form
841: (4)             a grid with `x` in the first dimension, `y` in the second, and `z` in
842: (4)             the third.
843: (4)             The parameters `x`, `y`, and `z` are converted to arrays only if they
844: (4)             are tuples or a lists, otherwise they are treated as a scalars. In
845: (4)             either case, either `x`, `y`, and `z` or their elements must support
846: (4)             multiplication and addition both with themselves and with the elements
847: (4)             of `c`.
848: (4)             If `c` has fewer than three dimensions, ones are implicitly appended to
849: (4)             its shape to make it 3-D. The shape of the result will be c.shape[3:] +
850: (4)             x.shape + y.shape + z.shape.
851: (4)             Parameters
852: (4)             -----
853: (4)             x, y, z : array_like, compatible objects
854: (8)             The three dimensional series is evaluated at the points in the
855: (8)             Cartesian product of `x`, `y`, and `z`. If `x`, `y`, or `z` is a
856: (8)             list or tuple, it is first converted to an ndarray, otherwise it is
857: (8)             left unchanged and, if it isn't an ndarray, it is treated as a
858: (8)             scalar.
859: (4)             c : array_like
860: (8)             Array of coefficients ordered so that the coefficients for terms of
861: (8)             degree i,j are contained in ``c[i,j]``. If `c` has dimension
862: (8)             greater than two the remaining indices enumerate multiple sets of
863: (8)             coefficients.
864: (4)             Returns
865: (4)             -----
866: (4)             values : ndarray, compatible object
867: (8)             The values of the two dimensional polynomial at points in the
868: (8)             product of `x` and `y`.
869: (4)             See Also
870: (4)             -----
871: (4)             lagval, lagval2d, laggrid2d, lagval3d
872: (4)             Notes
873: (4)             -----
874: (4)             .. versionadded:: 1.7.0
875: (4)             """
876: (4)             return pu._gridnd(lagval, c, x, y, z)
877: (0) def lagvander(x, deg):
878: (4)             """Pseudo-Vandermonde matrix of given degree.
879: (4)             Returns the pseudo-Vandermonde matrix of degree `deg` and sample points
880: (4)             `x`. The pseudo-Vandermonde matrix is defined by
881: (4)             .. math:: V[..., i] = L_i(x)
882: (4)             where `0 <= i <= deg`. The leading indices of `V` index the elements of
883: (4)             `x` and the last index is the degree of the Laguerre polynomial.
884: (4)             If `c` is a 1-D array of coefficients of length `n + 1` and `V` is the
885: (4)             array ``V = lagvander(x, n)``, then ``np.dot(V, c)`` and
886: (4)             ``lagval(x, c)`` are the same up to roundoff. This equivalence is
887: (4)             useful both for least squares fitting and for the evaluation of a large
888: (4)             number of Laguerre series of the same degree and sample points.
889: (4)             Parameters
890: (4)             -----

```

```

891: (4)           x : array_like
892: (8)             Array of points. The dtype is converted to float64 or complex128
893: (8)             depending on whether any of the elements are complex. If `x` is
894: (8)             scalar it is converted to a 1-D array.
895: (4)           deg : int
896: (8)             Degree of the resulting matrix.
897: (4)           Returns
898: (4)           -----
899: (4)           vander : ndarray
900: (8)             The pseudo-Vandermonde matrix. The shape of the returned matrix is
901: (8)             ``x.shape + (deg + 1,)`` , where The last index is the degree of the
902: (8)             corresponding Laguerre polynomial. The dtype will be the same as
903: (8)             the converted `x` .
904: (4)           Examples
905: (4)           -----
906: (4)           >>> from numpy.polynomial.laguerre import lagvander
907: (4)           >>> x = np.array([0, 1, 2])
908: (4)           >>> lagvander(x, 3)
909: (4)           array([[ 1.          ,  1.          ,  1.          ,  1.          ],
910: (11)            [ 1.          ,  0.          , -0.5         , -0.66666667],
911: (11)            [ 1.          , -1.          , -1.          , -0.33333333]])
912: (4)           """
913: (4)           ideg = pu._deprecate_as_int(deg, "deg")
914: (4)           if ideg < 0:
915: (8)             raise ValueError("deg must be non-negative")
916: (4)           x = np.array(x, copy=False, ndmin=1) + 0.0
917: (4)           dims = (ideg + 1,) + x.shape
918: (4)           dtype = x.dtype
919: (4)           v = np.empty(dims, dtype=dtype)
920: (4)           v[0] = x**0 + 1
921: (4)           if ideg > 0:
922: (8)             v[1] = 1 - x
923: (8)             for i in range(2, ideg + 1):
924: (12)               v[i] = (v[i-1]*(2*i - 1 - x) - v[i-2]*(i - 1))/i
925: (4)           return np.moveaxis(v, 0, -1)
926: (0)           def lagvander2d(x, y, deg):
927: (4)             """Pseudo-Vandermonde matrix of given degrees.
928: (4)             Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
929: (4)             points `(x, y)` . The pseudo-Vandermonde matrix is defined by
930: (4)             .. math:: V[..., (deg[1] + 1)*i + j] = L_i(x) * L_j(y),
931: (4)             where `0 <= i <= deg[0]` and `0 <= j <= deg[1]` . The leading indices of
932: (4)             `V` index the points `(x, y)` and the last index encodes the degrees of
933: (4)             the Laguerre polynomials.
934: (4)             If ``V = lagvander2d(x, y, [xdeg, ydeg])`` , then the columns of `V`
935: (4)             correspond to the elements of a 2-D coefficient array `c` of shape
936: (4)             (xdeg + 1, ydeg + 1) in the order
937: (4)             .. math:: c_{\{00\}}, c_{\{01\}}, c_{\{02\}} \dots , c_{\{10\}}, c_{\{11\}}, c_{\{12\}} \dots
938: (4)             and ``np.dot(V, c.flat)`` and ``lagval2d(x, y, c)`` will be the same
939: (4)             up to roundoff. This equivalence is useful both for least squares
940: (4)             fitting and for the evaluation of a large number of 2-D Laguerre
941: (4)             series of the same degrees and sample points.
942: (4)           Parameters
943: (4)           -----
944: (4)           x, y : array_like
945: (8)             Arrays of point coordinates, all of the same shape. The dtypes
946: (8)             will be converted to either float64 or complex128 depending on
947: (8)             whether any of the elements are complex. Scalars are converted to
948: (8)             1-D arrays.
949: (4)           deg : list of ints
950: (8)             List of maximum degrees of the form [x_deg, y_deg].
951: (4)           Returns
952: (4)           -----
953: (4)           vander2d : ndarray
954: (8)             The shape of the returned matrix is ``x.shape + (order,)`` , where
955: (8)             :math:`order = (deg[0]+1)*(deg[1]+1)` . The dtype will be the same
956: (8)             as the converted `x` and `y` .
957: (4)           See Also
958: (4)           -----
959: (4)           lagvander, lagvander3d, lagval2d, lagval3d

```

```

960: (4)          Notes
961: (4)          -----
962: (4)          .. versionadded:: 1.7.0
963: (4)          """
964: (4)          return pu._vander_nd_flat((lagvander, lagvander), (x, y), deg)
965: (0)          def lagvander3d(x, y, z, deg):
966: (4)          """Pseudo-Vandermonde matrix of given degrees.
967: (4)          Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
968: (4)          points `(x, y, z)`. If `l, m, n` are the given degrees in `x, y, z`,
969: (4)          then The pseudo-Vandermonde matrix is defined by
970: (4)          .. math:: V[..., (m+1)(n+1)i + (n+1)j + k] = L_i(x)*L_j(y)*L_k(z),
971: (4)          where `0 <= i <= l`, `0 <= j <= m`, and `0 <= k <= n`. The leading
972: (4)          indices of `V` index the points `(x, y, z)` and the last index encodes
973: (4)          the degrees of the Laguerre polynomials.
974: (4)          If ``V = lagvander3d(x, y, z, [xdeg, ydeg, zdeg])``, then the columns
975: (4)          of `V` correspond to the elements of a 3-D coefficient array `c` of
976: (4)          shape `(xdeg + 1, ydeg + 1, zdeg + 1)` in the order
977: (4)          .. math:: c_{\{000\}}, c_{\{001\}}, c_{\{002\}}, \dots, c_{\{010\}}, c_{\{011\}}, c_{\{012\}}, \dots
978: (4)          and ``np.dot(V, c.flat)`` and ``lagval3d(x, y, z, c)`` will be the
979: (4)          same up to roundoff. This equivalence is useful both for least squares
980: (4)          fitting and for the evaluation of a large number of 3-D Laguerre
981: (4)          series of the same degrees and sample points.
982: (4)          Parameters
983: (4)          -----
984: (4)          x, y, z : array_like
985: (8)          Arrays of point coordinates, all of the same shape. The dtypes will
986: (8)          be converted to either float64 or complex128 depending on whether
987: (8)          any of the elements are complex. Scalars are converted to 1-D
988: (8)          arrays.
989: (4)          deg : list of ints
990: (8)          List of maximum degrees of the form [x_deg, y_deg, z_deg].
991: (4)          Returns
992: (4)          -----
993: (4)          vander3d : ndarray
994: (8)          The shape of the returned matrix is ``x.shape + (order,)``, where
995: (8)          :math:`order = (deg[0]+1)*(deg[1]+1)*(deg[2]+1)`. The dtype will
996: (8)          be the same as the converted `x`, `y`, and `z`.
997: (4)          See Also
998: (4)          -----
999: (4)          lagvander, lagvander3d, lagval2d, lagval3d
1000: (4)          Notes
1001: (4)          -----
1002: (4)          .. versionadded:: 1.7.0
1003: (4)          """
1004: (4)          return pu._vander_nd_flat((lagvander, lagvander, lagvander), (x, y, z),
1005: (0)          deg)
1006: (4)          def lagfit(x, y, deg, rcond=None, full=False, w=None):
1007: (4)          """
1008: (4)          Least squares fit of Laguerre series to data.
1009: (4)          Return the coefficients of a Laguerre series of degree `deg` that is the
1010: (4)          least squares fit to the data values `y` given at points `x`. If `y` is
1011: (4)          1-D the returned coefficients will also be 1-D. If `y` is 2-D multiple
1012: (4)          fits are done, one for each column of `y`, and the resulting
1013: (4)          coefficients are stored in the corresponding columns of a 2-D return.
1014: (4)          The fitted polynomial(s) are in the form
1015: (4)          .. math:: p(x) = c_0 + c_1 * L_1(x) + \dots + c_n * L_n(x),
1016: (4)          where ``n`` is `deg`.
1017: (4)          Parameters
1018: (4)          -----
1019: (8)          x : array_like, shape (M,)
1020: (4)          x-coordinates of the M sample points ``x[i], y[i]``.
1021: (8)          y : array_like, shape (M,) or (M, K)
1022: (8)          y-coordinates of the sample points. Several data sets of sample
1023: (8)          points sharing the same x-coordinates can be fitted at once by
1024: (4)          passing in a 2D-array that contains one dataset per column.
1025: (8)          deg : int or 1-D array_like
1026: (8)          Degree(s) of the fitting polynomials. If `deg` is a single integer
1027: (8)          all terms up to and including the `deg`'th term are included in the
               fit. For NumPy versions >= 1.11.0 a list of integers specifying the

```

```

1028: (8)           degrees of the terms to include may be used instead.
1029: (4)           rcond : float, optional
1030: (8)             Relative condition number of the fit. Singular values smaller than
1031: (8)             this relative to the largest singular value will be ignored. The
1032: (8)             default value is len(x)*eps, where eps is the relative precision of
1033: (8)             the float type, about 2e-16 in most cases.
1034: (4)           full : bool, optional
1035: (8)             Switch determining nature of return value. When it is False (the
1036: (8)             default) just the coefficients are returned, when True diagnostic
1037: (8)             information from the singular value decomposition is also returned.
1038: (4)           w : array_like, shape (`M`,), optional
1039: (8)             Weights. If not None, the weight ``w[i]`` applies to the unsquared
1040: (8)             residual ``y[i] - y_hat[i]`` at ``x[i]``. Ideally the weights are
1041: (8)             chosen so that the errors of the products ``w[i]*y[i]`` all have the
1042: (8)             same variance. When using inverse-variance weighting, use
1043: (8)             ``w[i] = 1/sigma(y[i])``. The default value is None.
1044: (4)           Returns
1045: (4)           -----
1046: (4)           coef : ndarray, shape (M,) or (M, K)
1047: (8)             Laguerre coefficients ordered from low to high. If `y` was 2-D,
1048: (8)             the coefficients for the data in column *k* of `y` are in column
1049: (8)             *k*.
1050: (4)           [residuals, rank, singular_values, rcond] : list
1051: (8)             These values are only returned if ``full == True``
1052: (8)             - residuals -- sum of squared residuals of the least squares fit
1053: (8)             - rank -- the numerical rank of the scaled Vandermonde matrix
1054: (8)             - singular_values -- singular values of the scaled Vandermonde matrix
1055: (8)             - rcond -- value of `rcond`.
1056: (8)             For more details, see `numpy.linalg.lstsq`.
1057: (4)           Warns
1058: (4)           -----
1059: (4)           RankWarning
1060: (8)             The rank of the coefficient matrix in the least-squares fit is
1061: (8)             deficient. The warning is only raised if ``full == False``. The
1062: (8)             warnings can be turned off by
1063: (8)               >>> import warnings
1064: (8)               >>> warnings.simplefilter('ignore', np.RankWarning)
1065: (4)           See Also
1066: (4)           -----
1067: (4)           numpy.polynomial.polynomial.polyfit
1068: (4)           numpy.polynomial.legendre.legfit
1069: (4)           numpy.polynomial.chebyshev.chebfit
1070: (4)           numpy.polynomial.hermite.hermfit
1071: (4)           numpy.polynomial.hermite_e.hermefit
1072: (4)           lagval : Evaluates a Laguerre series.
1073: (4)           lagvander : pseudo Vandermonde matrix of Laguerre series.
1074: (4)           lagweight : Laguerre weight function.
1075: (4)           numpy.linalg.lstsq : Computes a least-squares fit from the matrix.
1076: (4)           scipy.interpolate.UnivariateSpline : Computes spline fits.
1077: (4)           Notes
1078: (4)           -----
1079: (4)           The solution is the coefficients of the Laguerre series ``p`` that
1080: (4)           minimizes the sum of the weighted squared errors
1081: (4)           .. math:: E = \sum_j w_j^2 * |y_j - p(x_j)|^2,
1082: (4)           where the :math:`w_j` are the weights. This problem is solved by
1083: (4)           setting up as the (typically) overdetermined matrix equation
1084: (4)           .. math:: V(x) * c = w * y,
1085: (4)           where ``V`` is the weighted pseudo Vandermonde matrix of `x`, ``c`` are
the
1086: (4)           coefficients to be solved for, `w` are the weights, and `y` are the
1087: (4)           observed values. This equation is then solved using the singular value
1088: (4)           decomposition of ``V``.
1089: (4)           If some of the singular values of `V` are so small that they are
1090: (4)           neglected, then a `RankWarning` will be issued. This means that the
1091: (4)           coefficient values may be poorly determined. Using a lower order fit
1092: (4)           will usually get rid of the warning. The `rcond` parameter can also be
1093: (4)           set to a value smaller than its default, but the resulting fit may be
1094: (4)           spurious and have large contributions from roundoff error.
1095: (4)           Fits using Laguerre series are probably most useful when the data can

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1096: (4)      be approximated by ``sqrt(w(x)) * p(x)`` , where ``w(x)`` is the Laguerre
1097: (4)      weight. In that case the weight ``sqrt(w(x[i]))`` should be used
1098: (4)      together with data values ``y[i]/sqrt(w(x[i]))`` . The weight function is
1099: (4)      available as `lagweight`.
1100: (4)      References
1101: (4)      -----
1102: (4)          .. [1] Wikipedia, "Curve fitting",
1103: (11)             https://en.wikipedia.org/wiki/Curve_fitting
1104: (4)      Examples
1105: (4)      -----
1106: (4)          >>> from numpy.polynomial.laguerre import lagfit, lagval
1107: (4)          >>> x = np.linspace(0, 10)
1108: (4)          >>> err = np.random.randn(len(x))/10
1109: (4)          >>> y = lagval(x, [1, 2, 3]) + err
1110: (4)          >>> lagfit(x, y, 2)
1111: (4)          array([ 0.96971004,  2.00193749,  3.00288744]) # may vary
1112: (4)          """
1113: (4)          return pu._fit(lagvander, x, y, deg, rcond, full, w)
1114: (0)      def lagcompanion(c):
1115: (4)          """
1116: (4)          Return the companion matrix of c.
1117: (4)          The usual companion matrix of the Laguerre polynomials is already
1118: (4)          symmetric when `c` is a basis Laguerre polynomial, so no scaling is
1119: (4)          applied.
1120: (4)          Parameters
1121: (4)          -----
1122: (4)          c : array_like
1123: (8)              1-D array of Laguerre series coefficients ordered from low to high
1124: (8)              degree.
1125: (4)          Returns
1126: (4)          -----
1127: (4)          mat : ndarray
1128: (8)              Companion matrix of dimensions (deg, deg).
1129: (4)          Notes
1130: (4)          -----
1131: (4)          .. versionadded:: 1.7.0
1132: (4)          """
1133: (4)          [c] = pu.as_series([c])
1134: (4)          if len(c) < 2:
1135: (8)              raise ValueError('Series must have maximum degree of at least 1.')
1136: (4)          if len(c) == 2:
1137: (8)              return np.array([[1 + c[0]/c[1]]])
1138: (4)          n = len(c) - 1
1139: (4)          mat = np.zeros((n, n), dtype=c.dtype)
1140: (4)          top = mat.reshape(-1)[1:n+1]
1141: (4)          mid = mat.reshape(-1)[0:n+1]
1142: (4)          bot = mat.reshape(-1)[n:-1]
1143: (4)          top[...] = -np.arange(1, n)
1144: (4)          mid[...] = 2.*np.arange(n) + 1.
1145: (4)          bot[...] = top
1146: (4)          mat[:, -1] += (c[:-1]/c[-1])*n
1147: (4)          return mat
1148: (0)      def lagroots(c):
1149: (4)          """
1150: (4)          Compute the roots of a Laguerre series.
1151: (4)          Return the roots (a.k.a. "zeros") of the polynomial
1152: (4)          .. math:: p(x) = \sum_i c[i] * L_i(x).
1153: (4)          Parameters
1154: (4)          -----
1155: (4)          c : 1-D array_like
1156: (8)              1-D array of coefficients.
1157: (4)          Returns
1158: (4)          -----
1159: (4)          out : ndarray
1160: (8)              Array of the roots of the series. If all the roots are real,
1161: (8)                  then `out` is also real, otherwise it is complex.
1162: (4)          See Also
1163: (4)          -----
1164: (4)          numpy.polynomial.polynomial.polyroots

```

```

1165: (4) numpy.polynomial.legendre.legroots
1166: (4) numpy.polynomial.chebyshev.chebroots
1167: (4) numpy.polynomial.hermite.hermroots
1168: (4) numpy.polynomial.hermite_e.hermroots
1169: (4) Notes
1170: (4) -----
1171: (4) The root estimates are obtained as the eigenvalues of the companion
1172: (4) matrix, Roots far from the origin of the complex plane may have large
1173: (4) errors due to the numerical instability of the series for such
1174: (4) values. Roots with multiplicity greater than 1 will also show larger
1175: (4) errors as the value of the series near such points is relatively
1176: (4) insensitive to errors in the roots. Isolated roots near the origin can
1177: (4) be improved by a few iterations of Newton's method.
1178: (4) The Laguerre series basis polynomials aren't powers of `x` so the
1179: (4) results of this function may seem unintuitive.
1180: (4) Examples
1181: (4) -----
1182: (4) >>> from numpy.polynomial.laguerre import lagroots, lagfromroots
1183: (4) >>> coef = lagfromroots([0, 1, 2])
1184: (4) >>> coef
1185: (4) array([ 2., -8., 12., -6.])
1186: (4) >>> lagroots(coef)
1187: (4) array([-4.4408921e-16, 1.0000000e+00, 2.0000000e+00])
1188: (4) """
1189: (4) [c] = pu.as_series([c])
1190: (4) if len(c) <= 1:
1191: (8)     return np.array([], dtype=c.dtype)
1192: (4) if len(c) == 2:
1193: (8)     return np.array([1 + c[0]/c[1]])
1194: (4) m = lagcompanion(c)[::-1, ::-1]
1195: (4) r = la.eigvals(m)
1196: (4) r.sort()
1197: (4) return r
1198: (0) def laggauss(deg):
1199: (4) """
1200: (4) Gauss-Laguerre quadrature.
1201: (4) Computes the sample points and weights for Gauss-Laguerre quadrature.
1202: (4) These sample points and weights will correctly integrate polynomials of
1203: (4) degree :math:`2*deg - 1` or less over the interval :math:`[0, \infty]`
1204: (4) with the weight function :math:`f(x) = \exp(-x)`.

Parameters
-----
deg : int
    Number of sample points and weights. It must be >= 1.

Returns
-----
x : ndarray
    1-D ndarray containing the sample points.
y : ndarray
    1-D ndarray containing the weights.

Notes
-----
.. versionadded:: 1.7.0
The results have only been tested up to degree 100 higher degrees may
be problematic. The weights are determined by using the fact that
.. math:: w_k = c / (L'_n(x_k) * L_{n-1}(x_k))
where :math:`c` is a constant independent of :math:`k` and :math:`x_k` is the k'th root of :math:`L_n`, and then scaling the results to get
the right value when integrating 1.

"""
1225: (4) ideg = pu._deprecate_as_int(deg, "deg")
1226: (4) if ideg <= 0:
1227: (8)     raise ValueError("deg must be a positive integer")
1228: (4) c = np.array([0]*deg + [1])
1229: (4) m = lagcompanion(c)
1230: (4) x = la.eigvalsh(m)
1231: (4) dy = lagval(x, c)
1232: (4) df = lagval(x, lagder(c))
1233: (4) x -= dy/df

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1234: (4)             fm = lagval(x, c[1:])
1235: (4)             fm /= np.abs(fm).max()
1236: (4)             df /= np.abs(df).max()
1237: (4)             w = 1/(fm * df)
1238: (4)             w /= w.sum()
1239: (4)             return x, w
1240: (0)             def lagweight(x):
1241: (4)                 """Weight function of the Laguerre polynomials.
1242: (4)                 The weight function is :math:`\exp(-x)` and the interval of integration
1243: (4)                 is :math:`[0, \infty]`. The Laguerre polynomials are orthogonal, but not
1244: (4)                 normalized, with respect to this weight function.
1245: (4)                 Parameters
1246: (4)                 -----
1247: (4)                 x : array_like
1248: (7)                   Values at which the weight function will be computed.
1249: (4)                 Returns
1250: (4)                 -----
1251: (4)                 w : ndarray
1252: (7)                   The weight function at `x`.
1253: (4)                 Notes
1254: (4)                 -----
1255: (4)                 .. versionadded:: 1.7.0
1256: (4)                 """
1257: (4)                 w = np.exp(-x)
1258: (4)                 return w
1259: (0)             class Laguerre(ABCPolyBase):
1260: (4)                 """A Laguerre series class.
1261: (4)                 The Laguerre class provides the standard Python numerical methods
1262: (4)                 '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the
1263: (4)                 attributes and methods listed in the `ABCPolyBase` documentation.
1264: (4)                 Parameters
1265: (4)                 -----
1266: (4)                 coef : array_like
1267: (8)                   Laguerre coefficients in order of increasing degree, i.e,
1268: (8)                   ````(1, 2, 3)```` gives ````1*L_0(x) + 2*L_1(X) + 3*L_2(x)````.
1269: (4)                 domain : (2,) array_like, optional
1270: (8)                   Domain to use. The interval ``[domain[0], domain[1]]`` is mapped
1271: (8)                   to the interval ``[window[0], window[1]]`` by shifting and scaling.
1272: (8)                   The default value is [0, 1].
1273: (4)                 window : (2,) array_like, optional
1274: (8)                   Window, see `domain` for its use. The default value is [0, 1].
1275: (8)                   .. versionadded:: 1.6.0
1276: (4)                 symbol : str, optional
1277: (8)                   Symbol used to represent the independent variable in string
1278: (8)                   representations of the polynomial expression, e.g. for printing.
1279: (8)                   The symbol must be a valid Python identifier. Default value is 'x'.
1280: (8)                   .. versionadded:: 1.24
1281: (4)                   """
1282: (4)                 _add = staticmethod(lagadd)
1283: (4)                 _sub = staticmethod(lagsub)
1284: (4)                 _mul = staticmethod(lagmul)
1285: (4)                 _div = staticmethod(lagdiv)
1286: (4)                 _pow = staticmethod(lagpow)
1287: (4)                 _val = staticmethod(lagval)
1288: (4)                 _int = staticmethod(lagint)
1289: (4)                 _der = staticmethod(lagder)
1290: (4)                 _fit = staticmethod(lagfit)
1291: (4)                 _line = staticmethod(lagline)
1292: (4)                 _roots = staticmethod(lagroots)
1293: (4)                 _fromroots = staticmethod(lagfromroots)
1294: (4)                 domain = np.array(lagdomain)
1295: (4)                 window = np.array(lagdomain)
1296: (4)                 basis_name = 'L'

```

-----  
File 291 - legendre.py:

1: (0) """

```
2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (0)
8: (0)
9: (0)
10: (0)
11: (0)
12: (0)
13: (3)
14: (4)
15: (0)
16: (0)
17: (0)
18: (3)
19: (3)
20: (3)
21: (3)
22: (3)
23: (0)
24: (0)
25: (0)
26: (3)
27: (3)
28: (3)
29: (3)
30: (3)
31: (3)
32: (3)
33: (3)
34: (3)
35: (3)
36: (3)
37: (3)
38: (0)
39: (0)
40: (0)
41: (3)
42: (3)
43: (3)
44: (0)
45: (0)
46: (0)
47: (3)
48: (3)
49: (3)
50: (3)
51: (3)
52: (3)
53: (3)
54: (3)
55: (3)
56: (3)
57: (3)
58: (3)
59: (3)
60: (3)
61: (0)
62: (0)
63: (0)
64: (0)
65: (0)
66: (0)
67: (0)
68: (0)
69: (0)
70: (0)

=====
Legendre Series (:mod:`numpy.polynomial.legendre`)
=====

This module provides a number of objects (mostly functions) useful for dealing with Legendre series, including a `Legendre` class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

Classes
-----
.. autosummary::
   :toctree: generated/
   Legendre

Constants
-----
.. autosummary::
   :toctree: generated/
   legdomain
   legzero
   legone
   legx

Arithmetic
-----
.. autosummary::
   :toctree: generated/
   legadd
   legsub
   legmulx
   legmul
   legdiv
   legpow
   legval
   legval2d
   legval3d
   leggrid2d
   leggrid3d

Calculus
-----
.. autosummary::
   :toctree: generated/
   legder
   legint

Misc Functions
-----
.. autosummary::
   :toctree: generated/
   legfromroots
   legroots
   legvander
   legvander2d
   legvander3d
   leggauss
   legweight
   legcompanion
   legfit
   legtrim
   legline
   leg2poly
   poly2leg

See also
-----
numpy.polynomial
"""

import numpy as np
import numpy.linalg as la
from numpy.core.multiarray import normalize_axis_index
from . import polyutils as pu
from ._polybase import ABCPolyBase
__all__ = [
```

```

71: (4)          'legzero', 'legone', 'legx', 'legdomain', 'legline', 'legadd',
72: (4)          'legsub', 'legmulx', 'legmul', 'legdiv', 'legpow', 'legval', 'legder',
73: (4)          'legint', 'leg2poly', 'poly2leg', 'legfromroots', 'legvander',
74: (4)          'legfit', 'legtrim', 'legroots', 'Legendre', 'legval2d', 'legval3d',
75: (4)          'leggrid2d', 'leggrid3d', 'legvander2d', 'legvander3d', 'legcompanion',
76: (4)          'leggauss', 'legweight']
77: (0)          legtrim = pu.trimcoef
78: (0)          def poly2leg(pol):
79: (4)            """
80: (4)              Convert a polynomial to a Legendre series.
81: (4)              Convert an array representing the coefficients of a polynomial (relative
82: (4)                  to the "standard" basis) ordered from lowest degree to highest, to an
83: (4)                  array of the coefficients of the equivalent Legendre series, ordered
84: (4)                  from lowest to highest degree.
85: (4)          Parameters
86: (4)          -----
87: (4)          pol : array_like
88: (8)            1-D array containing the polynomial coefficients
89: (4)          Returns
89: (4)          -----
91: (4)          c : ndarray
92: (8)            1-D array containing the coefficients of the equivalent Legendre
93: (8)            series.
94: (4)          See Also
95: (4)          -----
96: (4)          leg2poly
97: (4)          Notes
98: (4)          -----
99: (4)          The easy way to do conversions between polynomial basis sets
100: (4)         is to use the convert method of a class instance.
101: (4)          Examples
102: (4)          -----
103: (4)          >>> from numpy import polynomial as P
104: (4)          >>> p = P.Polynomial(np.arange(4))
105: (4)          >>> p
106: (4)          Polynomial([0., 1., 2., 3.], domain=[-1, 1], window=[-1, 1])
107: (4)          >>> c = P.Legendre(P.legendre.poly2leg(p.coef))
108: (4)          >>> c
109: (4)          Legendre([ 1. ,  3.25,  1. ,  0.75], domain=[-1, 1], window=[-1, 1]) #
may vary
110: (4)          """
111: (4)          [pol] = pu.as_series([pol])
112: (4)          deg = len(pol) - 1
113: (4)          res = 0
114: (4)          for i in range(deg, -1, -1):
115: (8)            res = legadd(legmulx(res), pol[i])
116: (4)          return res
117: (0)          def leg2poly(c):
118: (4)            """
119: (4)              Convert a Legendre series to a polynomial.
120: (4)              Convert an array representing the coefficients of a Legendre series,
121: (4)                  ordered from lowest degree to highest, to an array of the coefficients
122: (4)                  of the equivalent polynomial (relative to the "standard" basis) ordered
123: (4)                  from lowest to highest degree.
124: (4)          Parameters
125: (4)          -----
126: (4)          c : array_like
127: (8)            1-D array containing the Legendre series coefficients, ordered
128: (8)            from lowest order term to highest.
129: (4)          Returns
129: (4)          -----
131: (4)          pol : ndarray
132: (8)            1-D array containing the coefficients of the equivalent polynomial
133: (8)            (relative to the "standard" basis) ordered from lowest order term
134: (8)            to highest.
135: (4)          See Also
136: (4)          -----
137: (4)          poly2leg
138: (4)          Notes

```

```

139: (4)      -----
140: (4)      The easy way to do conversions between polynomial basis sets
141: (4)      is to use the convert method of a class instance.
142: (4)      Examples
143: (4)      -----
144: (4)      >>> from numpy import polynomial as P
145: (4)      >>> c = P.Legendre(range(4))
146: (4)      >>> c
147: (4)      Legendre([0., 1., 2., 3.], domain=[-1, 1], window=[-1, 1])
148: (4)      >>> p = c.convert(kind=P.Polynomial)
149: (4)      >>> p
150: (4)      Polynomial([-1. , -3.5,  3. ,  7.5], domain=[-1.,  1.], window=[-1.,  1.])
151: (4)      >>> P.legendre.leg2poly(range(4))
152: (4)      array([-1. , -3.5,  3. ,  7.5])
153: (4)      """
154: (4)      from .polynomial import polyadd, polysub, polymulx
155: (4)      [c] = pu.as_series([c])
156: (4)      n = len(c)
157: (4)      if n < 3:
158: (8)          return c
159: (4)      else:
160: (8)          c0 = c[-2]
161: (8)          c1 = c[-1]
162: (8)          for i in range(n - 1, 1, -1):
163: (12)              tmp = c0
164: (12)              c0 = polysub(c[i - 2], (c1*(i - 1))/i)
165: (12)              c1 = polyadd(tmp, (polymulx(c1)*(2*i - 1))/i)
166: (8)          return polyadd(c0, polymulx(c1))
167: (0)          legdomain = np.array([-1, 1])
168: (0)          legzero = np.array([0])
169: (0)          legone = np.array([1])
170: (0)          legx = np.array([0, 1])
171: (0)          def legline(off, scl):
172: (4)              """
173: (4)              Legendre series whose graph is a straight line.
174: (4)              Parameters
175: (4)              -----
176: (4)              off, scl : scalars
177: (8)                  The specified line is given by ``off + scl*x``.
178: (4)              Returns
179: (4)              -----
180: (4)              y : ndarray
181: (8)                  This module's representation of the Legendre series for
182: (8)                  ``off + scl*x``.
183: (4)              See Also
184: (4)              -----
185: (4)              numpy.polynomial.polynomial.polyline
186: (4)              numpy.polynomial.chebyshev.chebline
187: (4)              numpy.polynomial.laguerre.lagline
188: (4)              numpy.polynomial.hermite.hermeline
189: (4)              numpy.polynomial.hermite_e.hermeline
190: (4)              Examples
191: (4)              -----
192: (4)              >>> import numpy.polynomial.legendre as L
193: (4)              >>> L.legline(3,2)
194: (4)              array([3, 2])
195: (4)              >>> L.legval(-3, L.legline(3,2)) # should be -3
196: (4)              -3.0
197: (4)              """
198: (4)              if scl != 0:
199: (8)                  return np.array([off, scl])
200: (4)              else:
201: (8)                  return np.array([off])
202: (0)          def legfromroots(roots):
203: (4)              """
204: (4)              Generate a Legendre series with given roots.
205: (4)              The function returns the coefficients of the polynomial
206: (4)              .. math:: p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),
207: (4)              in Legendre form, where the `r_n` are the roots specified in `roots`.

```

```

208: (4) If a zero has multiplicity n, then it must appear in `roots` n times.
209: (4) For instance, if 2 is a root of multiplicity three and 3 is a root of
210: (4) multiplicity 2, then `roots` looks something like [2, 2, 2, 3, 3]. The
211: (4) roots can appear in any order.
212: (4) If the returned coefficients are `c`, then
213: (4) .. math:: p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x)
214: (4) The coefficient of the last term is not generally 1 for monic
215: (4) polynomials in Legendre form.
216: (4) Parameters
217: (4) -----
218: (4) roots : array_like
219: (8) Sequence containing the roots.
220: (4) Returns
221: (4) -----
222: (4) out : ndarray
223: (8) 1-D array of coefficients. If all roots are real then `out` is a
224: (8) real array, if some of the roots are complex, then `out` is complex
225: (8) even if all the coefficients in the result are real (see Examples
226: (8) below).
227: (4) See Also
228: (4) -----
229: (4) numpy.polynomial.polynomial.polyfromroots
230: (4) numpy.polynomial.chebyshev.chebfromroots
231: (4) numpy.polynomial.laguerre.lagfromroots
232: (4) numpy.polynomial.hermite.hermfromroots
233: (4) numpy.polynomial.hermite_e.hermefromroots
234: (4) Examples
235: (4) -----
236: (4) >>> import numpy.polynomial.legendre as L
237: (4) >>> L.legfromroots((-1,0,1)) # x^3 - x relative to the standard basis
238: (4) array([ 0. , -0.4,  0. ,  0.4])
239: (4) >>> j = complex(0,1)
240: (4) >>> L.legfromroots((-j,j)) # x^2 + 1 relative to the standard basis
241: (4) array([ 1.33333333+0.j,  0.00000000+0.j,  0.66666667+0.j]) # may vary
242: (4) """
243: (4)     return pu._fromroots(legline, legmul, roots)
244: (0) def legadd(c1, c2):
245: (4) """
246: (4)     Add one Legendre series to another.
247: (4)     Returns the sum of two Legendre series `c1` + `c2`. The arguments
248: (4)     are sequences of coefficients ordered from lowest order term to
249: (4)     highest, i.e., [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
250: (4)     Parameters
251: (4) -----
252: (4)     c1, c2 : array_like
253: (8)     1-D arrays of Legendre series coefficients ordered from low to
254: (8)     high.
255: (4)     Returns
256: (4) -----
257: (4)     out : ndarray
258: (8)     Array representing the Legendre series of their sum.
259: (4) See Also
260: (4) -----
261: (4)     legsub, legmulx, legmul, legdiv, legpow
262: (4) Notes
263: (4) -----
264: (4)     Unlike multiplication, division, etc., the sum of two Legendre series
265: (4)     is a Legendre series (without having to "reproject" the result onto
266: (4)     the basis set) so addition, just like that of "standard" polynomials,
267: (4)     is simply "component-wise."
268: (4) Examples
269: (4) -----
270: (4) >>> from numpy.polynomial import legendre as L
271: (4) >>> c1 = (1,2,3)
272: (4) >>> c2 = (3,2,1)
273: (4) >>> L.legadd(c1,c2)
274: (4) array([4.,  4.,  4.])
275: (4) """
276: (4)     return pu._add(c1, c2)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

277: (0)
278: (4)
279: (4)
280: (4)
281: (4)
282: (4)
283: (4)
284: (4)
285: (4)
286: (8)
287: (8)
288: (4)
289: (4)
290: (4)
291: (8)
292: (4)
293: (4)
294: (4)
295: (4)
296: (4)
297: (4)
298: (4)
299: (4)
300: (4)
301: (4)
302: (4)
303: (4)
304: (4)
305: (4)
306: (4)
307: (4)
308: (4)
309: (4)
310: (4)
311: (4)
312: (0)
def legsub(c1, c2):
    """
    Subtract one Legendre series from another.
    Returns the difference of two Legendre series `c1` - `c2`. The
    sequences of coefficients are from lowest order term to highest, i.e.,
    [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
    Parameters
    -----
    c1, c2 : array_like
        1-D arrays of Legendre series coefficients ordered from low to
        high.
    Returns
    -----
    out : ndarray
        Of Legendre series coefficients representing their difference.
    See Also
    -----
    legadd, legmulx, legmul, legdiv, legpow
    Notes
    -----
    Unlike multiplication, division, etc., the difference of two Legendre
    series is a Legendre series (without having to "reproject" the result
    onto the basis set) so subtraction, just like that of "standard"
    polynomials, is simply "component-wise."
    Examples
    -----
    >>> from numpy.polynomial import legendre as L
    >>> c1 = (1,2,3)
    >>> c2 = (3,2,1)
    >>> L.legsub(c1,c2)
    array([-2.,  0.,  2.])
    >>> L.legsub(c2,c1) # -C.legsub(c1,c2)
    array([ 2.,  0., -2.])
    """
    return pu._sub(c1, c2)

def legmulx(c):
    """
    Multiply a Legendre series by x.
    Multiply the Legendre series `c` by x, where x is the independent
    variable.
    Parameters
    -----
    c : array_like
        1-D array of Legendre series coefficients ordered from low to
        high.
    Returns
    -----
    out : ndarray
        Array representing the result of the multiplication.
    See Also
    -----
    legadd, legmul, legdiv, legpow
    Notes
    -----
    The multiplication uses the recursion relationship for Legendre
    polynomials in the form
    .. math::
        xP_i(x) = ((i + 1)*P_{i + 1}(x) + i*P_{i - 1}(x))/(2i + 1)
    Examples
    -----
    >>> from numpy.polynomial import legendre as L
    >>> L.legmulx([1,2,3])
    array([ 0.66666667,  2.2,  1.33333333,  1.8]) # may vary
    """
    [c] = pu.as_series([c])
    if len(c) == 1 and c[0] == 0:
        return c
    prd = np.empty(len(c) + 1, dtype=c.dtype)
    prd[0] = c[0]*0
    prd[1] = c[0]

```

```

346: (4)             for i in range(1, len(c)):
347: (8)                 j = i + 1
348: (8)                 k = i - 1
349: (8)                 s = i + j
350: (8)                 prd[j] = (c[i]*j)/s
351: (8)                 prd[k] += (c[i]*i)/s
352: (4)             return prd
353: (0)         def legmul(c1, c2):
354: (4)             """
355: (4)                 Multiply one Legendre series by another.
356: (4)                 Returns the product of two Legendre series `c1` * `c2`. The arguments
357: (4)                 are sequences of coefficients, from lowest order "term" to highest,
358: (4)                 e.g., [1,2,3] represents the series ``P_0 + 2*P_1 + 3*P_2``.
359: (4)             Parameters
360: (4)             -----
361: (4)             c1, c2 : array_like
362: (8)                 1-D arrays of Legendre series coefficients ordered from low to
363: (8)                 high.
364: (4)             Returns
365: (4)             -----
366: (4)             out : ndarray
367: (8)                 Of Legendre series coefficients representing their product.
368: (4)             See Also
369: (4)             -----
370: (4)             legadd, legsub, legmulx, legdiv, legpow
371: (4)             Notes
372: (4)             -----
373: (4)             In general, the (polynomial) product of two C-series results in terms
374: (4)             that are not in the Legendre polynomial basis set. Thus, to express
375: (4)             the product as a Legendre series, it is necessary to "reproject" the
376: (4)             product onto said basis set, which may produce "unintuitive" (but
377: (4)             correct) results; see Examples section below.
378: (4)             Examples
379: (4)             -----
380: (4)             >>> from numpy.polynomial import legendre as L
381: (4)             >>> c1 = (1,2,3)
382: (4)             >>> c2 = (3,2)
383: (4)             >>> L.legmul(c1,c2) # multiplication requires "reprojection"
384: (4)             array([ 4.33333333,  10.4        ,  11.66666667,   3.6        ]) # may vary
385: (4)             """
386: (4)             [c1, c2] = pu.as_series([c1, c2])
387: (4)             if len(c1) > len(c2):
388: (8)                 c = c2
389: (8)                 xs = c1
390: (4)             else:
391: (8)                 c = c1
392: (8)                 xs = c2
393: (4)             if len(c) == 1:
394: (8)                 c0 = c[0]*xs
395: (8)                 c1 = 0
396: (4)             elif len(c) == 2:
397: (8)                 c0 = c[0]*xs
398: (8)                 c1 = c[1]*xs
399: (4)             else:
400: (8)                 nd = len(c)
401: (8)                 c0 = c[-2]*xs
402: (8)                 c1 = c[-1]*xs
403: (8)                 for i in range(3, len(c) + 1):
404: (12)                     tmp = c0
405: (12)                     nd = nd - 1
406: (12)                     c0 = legsub(c[-i]*xs, (c1*(nd - 1))/nd)
407: (12)                     c1 = legadd(tmp, (legmulx(c1)*(2*nd - 1))/nd)
408: (4)             return legadd(c0, legmulx(c1))
409: (0)         def legdiv(c1, c2):
410: (4)             """
411: (4)                 Divide one Legendre series by another.
412: (4)                 Returns the quotient-with-remainder of two Legendre series
413: (4)                 `c1` / `c2`. The arguments are sequences of coefficients from lowest
414: (4)                 order "term" to highest, e.g., [1,2,3] represents the series

```

```

415: (4)           ``P_0 + 2*P_1 + 3*P_2``.
416: (4)           Parameters
417: (4)           -----
418: (4)           c1, c2 : array_like
419: (8)           1-D arrays of Legendre series coefficients ordered from low to
420: (8)           high.
421: (4)           Returns
422: (4)           -----
423: (4)           quo, rem : ndarrays
424: (8)           Of Legendre series coefficients representing the quotient and
425: (8)           remainder.
426: (4)           See Also
427: (4)           -----
428: (4)           legadd, legsub, legmulx, legmul, legpow
429: (4)           Notes
430: (4)           -----
431: (4)           In general, the (polynomial) division of one Legendre series by another
432: (4)           results in quotient and remainder terms that are not in the Legendre
433: (4)           polynomial basis set. Thus, to express these results as a Legendre
434: (4)           series, it is necessary to "reproject" the results onto the Legendre
435: (4)           basis set, which may produce "unintuitive" (but correct) results; see
436: (4)           Examples section below.
437: (4)           Examples
438: (4)           -----
439: (4)           >>> from numpy.polynomial import legendre as L
440: (4)           >>> c1 = (1,2,3)
441: (4)           >>> c2 = (3,2,1)
442: (4)           >>> L.legdiv(c1,c2) # quotient "intuitive," remainder not
443: (4)           (array([3.]), array([-8., -4.]))
444: (4)           >>> c2 = (0,1,2,3)
445: (4)           >>> L.legdiv(c2,c1) # neither "intuitive"
446: (4)           (array([-0.07407407,  1.66666667]), array([-1.03703704, -2.51851852])) #
may vary
447: (4)           """
448: (4)           return pu._div(legmul, c1, c2)
449: (0)           def legpow(c, pow, maxpower=16):
450: (4)           """Raise a Legendre series to a power.
451: (4)           Returns the Legendre series `c` raised to the power `pow`. The
452: (4)           argument `c` is a sequence of coefficients ordered from low to high.
453: (4)           i.e., [1,2,3] is the series ``P_0 + 2*P_1 + 3*P_2``.
454: (4)           Parameters
455: (4)           -----
456: (4)           c : array_like
457: (8)           1-D array of Legendre series coefficients ordered from low to
458: (8)           high.
459: (4)           pow : integer
460: (8)           Power to which the series will be raised
461: (4)           maxpower : integer, optional
462: (8)           Maximum power allowed. This is mainly to limit growth of the series
463: (8)           to unmanageable size. Default is 16
464: (4)           Returns
465: (4)           -----
466: (4)           coef : ndarray
467: (8)           Legendre series of power.
468: (4)           See Also
469: (4)           -----
470: (4)           legadd, legsub, legmulx, legmul, legdiv
471: (4)           """
472: (4)           return pu._pow(legmul, c, pow, maxpower)
473: (0)           def ledger(c, m=1, scl=1, axis=0):
474: (4)           """
475: (4)           Differentiate a Legendre series.
476: (4)           Returns the Legendre series coefficients `c` differentiated `m` times
477: (4)           along `axis`. At each iteration the result is multiplied by `scl` (the
478: (4)           scaling factor is for use in a linear change of variable). The argument
479: (4)           `c` is an array of coefficients from low to high degree along each
480: (4)           axis, e.g., [1,2,3] represents the series ``1*L_0 + 2*L_1 + 3*L_2``
481: (4)           while [[1,2],[1,2]] represents ``1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) +
482: (4)           2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)`` if axis=0 is ``x`` and axis=1 is

```

```

483: (4)          ``y``.
484: (4)          Parameters
485: (4)          -----
486: (4)          c : array_like
487: (8)          Array of Legendre series coefficients. If c is multidimensional the
488: (8)          different axis correspond to different variables with the degree in
489: (8)          each axis given by the corresponding index.
490: (4)          m : int, optional
491: (8)          Number of derivatives taken, must be non-negative. (Default: 1)
492: (4)          scl : scalar, optional
493: (8)          Each differentiation is multiplied by `scl`. The end result is
494: (8)          multiplication by ``scl**m``. This is for use in a linear change of
495: (8)          variable. (Default: 1)
496: (4)          axis : int, optional
497: (8)          Axis over which the derivative is taken. (Default: 0).
498: (8)          .. versionadded:: 1.7.0
499: (4)          Returns
500: (4)          -----
501: (4)          der : ndarray
502: (8)          Legendre series of the derivative.
503: (4)          See Also
504: (4)          -----
505: (4)          legint
506: (4)          Notes
507: (4)          -----
508: (4)          In general, the result of differentiating a Legendre series does not
509: (4)          resemble the same operation on a power series. Thus the result of this
510: (4)          function may be "unintuitive," albeit correct; see Examples section
511: (4)          below.
512: (4)          Examples
513: (4)          -----
514: (4)          >>> from numpy.polynomial import legendre as L
515: (4)          >>> c = (1,2,3,4)
516: (4)          >>> L.legder(c)
517: (4)          array([ 6.,  9., 20.])
518: (4)          >>> L.legder(c, 3)
519: (4)          array([60.])
520: (4)          >>> L.legder(c, scl=-1)
521: (4)          array([-6., -9., -20.])
522: (4)          >>> L.legder(c, 2,-1)
523: (4)          array([ 9., 60.])
524: (4)          """
525: (4)          c = np.array(c, ndmin=1, copy=True)
526: (4)          if c.dtype.char in '?bBhHiIlLqQpP':
527: (8)              c = c.astype(np.double)
528: (4)          cnt = pu._deprecate_as_int(m, "the order of derivation")
529: (4)          iaxis = pu._deprecate_as_int(axis, "the axis")
530: (4)          if cnt < 0:
531: (8)              raise ValueError("The order of derivation must be non-negative")
532: (4)          iaxis = normalize_axis_index(iaxis, c.ndim)
533: (4)          if cnt == 0:
534: (8)              return c
535: (4)          c = np.moveaxis(c, iaxis, 0)
536: (4)          n = len(c)
537: (4)          if cnt >= n:
538: (8)              c = c[:1]*0
539: (4)          else:
540: (8)              for i in range(cnt):
541: (12)                  n = n - 1
542: (12)                  c *= scl
543: (12)                  der = np.empty((n,) + c.shape[1:], dtype=c.dtype)
544: (12)                  for j in range(n, 2, -1):
545: (16)                      der[j - 1] = (2*j - 1)*c[j]
546: (16)                      c[j - 2] += c[j]
547: (12)                      if n > 1:
548: (16)                          der[1] = 3*c[2]
549: (12)                          der[0] = c[1]
550: (12)                          c = der
551: (4)          c = np.moveaxis(c, 0, iaxis)

```

```

552: (4)             return c
553: (0)             def legint(c, m=1, k=[], lbnd=0, scl=1, axis=0):
554: (4)                 """
555: (4)                     Integrate a Legendre series.
556: (4)                     Returns the Legendre series coefficients `c` integrated `m` times from
557: (4)                     `lbnd` along `axis`. At each iteration the resulting series is
558: (4)                     **multiplied** by `scl` and an integration constant, `k`, is added.
559: (4)                     The scaling factor is for use in a linear change of variable. ("Buyer
560: (4)                     beware": note that, depending on what one is doing, one may want `scl`
561: (4)                     to be the reciprocal of what one might expect; for more information,
562: (4)                     see the Notes section below.) The argument `c` is an array of
563: (4)                     coefficients from low to high degree along each axis, e.g., [1,2,3]
564: (4)                     represents the series `` $L_0 + 2L_1 + 3L_2$ `` while [[1,2],[1,2]]
565: (4)                     represents `` $1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) +$ 
566: (4)                      $2*L_1(x)*L_1(y)$ `` if axis=0 is ``x`` and axis=1 is ``y``.
567: (4)             Parameters
568: (4)                 -----
569: (4)                 c : array_like
570: (8)                     Array of Legendre series coefficients. If c is multidimensional the
571: (8)                     different axis correspond to different variables with the degree in
572: (8)                     each axis given by the corresponding index.
573: (4)                 m : int, optional
574: (8)                     Order of integration, must be positive. (Default: 1)
575: (4)                 k : [], list, scalar, optional
576: (8)                     Integration constant(s). The value of the first integral at
577: (8)                     ``lbnd`` is the first value in the list, the value of the second
578: (8)                     integral at ``lbnd`` is the second value, etc. If ``k == []`` (the
579: (8)                     default), all constants are set to zero. If ``m == 1``, a single
580: (8)                     scalar can be given instead of a list.
581: (4)                 lbnd : scalar, optional
582: (8)                     The lower bound of the integral. (Default: 0)
583: (4)                 scl : scalar, optional
584: (8)                     Following each integration the result is *multiplied* by `scl`
585: (8)                     before the integration constant is added. (Default: 1)
586: (4)                 axis : int, optional
587: (8)                     Axis over which the integral is taken. (Default: 0).
588: (8)                     .. versionadded:: 1.7.0
589: (4)             Returns
590: (4)                 -----
591: (4)                 S : ndarray
592: (8)                     Legendre series coefficient array of the integral.
593: (4)             Raises
594: (4)                 -----
595: (4)                 ValueError
596: (8)                     If ``m < 0``, ``len(k) > m``, ``np.ndim(lbnd) != 0``, or
597: (8)                     ``np.ndim(scl) != 0``.
598: (4)             See Also
599: (4)                 -----
600: (4)                 legder
601: (4)                 Notes
602: (4)                 -----
603: (4)                 Note that the result of each integration is *multiplied* by `scl`.
604: (4)                 Why is this important to note? Say one is making a linear change of
605: (4)                 variable :math:`u = ax + b` in an integral relative to `x`. Then
606: (4)                 :math:`dx = du/a`, so one will need to set `scl` equal to
607: (4)                 :math:`1/a` - perhaps not what one would have first thought.
608: (4)                 Also note that, in general, the result of integrating a C-series needs
609: (4)                 to be "reprojected" onto the C-series basis set. Thus, typically,
610: (4)                 the result of this function is "unintuitive," albeit correct; see
611: (4)                 Examples section below.
612: (4)             Examples
613: (4)                 -----
614: (4)                 >>> from numpy.polynomial import legendre as L
615: (4)                 >>> c = (1,2,3)
616: (4)                 >>> L.legint(c)
617: (4)                 array([ 0.33333333,  0.4           ,  0.66666667,  0.6           ]) # may vary
618: (4)                 >>> L.legint(c, 3)
619: (4)                 array([ 1.66666667e-02, -1.78571429e-02,  4.76190476e-02, # may vary
620: (13)                               -1.73472348e-18,   1.90476190e-02,   9.52380952e-03])

```

```

621: (4)             >>> L.legint(c, k=3)
622: (5)             array([ 3.33333333,  0.4        ,  0.66666667,  0.6        ]) # may vary
623: (4)             >>> L.legint(c, lbnd=-2)
624: (4)             array([ 7.33333333,  0.4        ,  0.66666667,  0.6        ]) # may vary
625: (4)             >>> L.legint(c, scl=2)
626: (4)             array([ 0.66666667,  0.8        ,  1.33333333,  1.2        ]) # may vary
627: (4)             """
628: (4)             c = np.array(c, ndmin=1, copy=True)
629: (4)             if c.dtype.char in '?bBhHiIlqQpP':
630: (8)                 c = c.astype(np.double)
631: (4)             if not np.iterable(k):
632: (8)                 k = [k]
633: (4)             cnt = pu._deprecate_as_int(m, "the order of integration")
634: (4)             iaxis = pu._deprecate_as_int(axis, "the axis")
635: (4)             if cnt < 0:
636: (8)                 raise ValueError("The order of integration must be non-negative")
637: (4)             if len(k) > cnt:
638: (8)                 raise ValueError("Too many integration constants")
639: (4)             if np.ndim(lbnd) != 0:
640: (8)                 raise ValueError("lbnd must be a scalar.")
641: (4)             if np.ndim(scl) != 0:
642: (8)                 raise ValueError("scl must be a scalar.")
643: (4)             iaxis = normalize_axis_index(iaxis, c.ndim)
644: (4)             if cnt == 0:
645: (8)                 return c
646: (4)             c = np.moveaxis(c, iaxis, 0)
647: (4)             k = list(k) + [0]*(cnt - len(k))
648: (4)             for i in range(cnt):
649: (8)                 n = len(c)
650: (8)                 c *= scl
651: (8)                 if n == 1 and np.all(c[0] == 0):
652: (12)                     c[0] += k[i]
653: (8)                 else:
654: (12)                     tmp = np.empty((n + 1,) + c.shape[1:], dtype=c.dtype)
655: (12)                     tmp[0] = c[0]*0
656: (12)                     tmp[1] = c[0]
657: (12)                     if n > 1:
658: (16)                         tmp[2] = c[1]/3
659: (12)                         for j in range(2, n):
660: (16)                             t = c[j]/(2*j + 1)
661: (16)                             tmp[j + 1] = t
662: (16)                             tmp[j - 1] -= t
663: (12)                         tmp[0] += k[i] - legval(lbnd, tmp)
664: (12)                     c = tmp
665: (4)                     c = np.moveaxis(c, 0, iaxis)
666: (4)             return c
667: (0)             def legval(x, c, tensor=True):
668: (4)                 """
669: (4)                     Evaluate a Legendre series at points x.
670: (4)                     If `c` is of length `n + 1`, this function returns the value:
671: (4)                     .. math:: p(x) = c_0 * L_0(x) + c_1 * L_1(x) + \dots + c_n * L_n(x)
672: (4)                     The parameter `x` is converted to an array only if it is a tuple or a
673: (4)                     list, otherwise it is treated as a scalar. In either case, either `x`
674: (4)                     or its elements must support multiplication and addition both with
675: (4)                     themselves and with the elements of `c`.
676: (4)                     If `c` is a 1-D array, then `p(x)` will have the same shape as `x`. If
677: (4)                     `c` is multidimensional, then the shape of the result depends on the
678: (4)                     value of `tensor`. If `tensor` is true the shape will be c.shape[1:] +
679: (4)                     x.shape. If `tensor` is false the shape will be c.shape[1:].
680: (4)                     Note that scalars have shape ().
681: (4)                     Trailing zeros in the coefficients will be used in the evaluation, so
682: (4)                     they should be avoided if efficiency is a concern.
683: (4)                     Parameters
684: (4)                     -----
685: (4)                     x : array_like, compatible object
686: (8)                     If `x` is a list or tuple, it is converted to an ndarray, otherwise
687: (8)                     it is left unchanged and treated as a scalar. In either case, `x`
688: (8)                     or its elements must support addition and multiplication with
689: (8)                     themselves and with the elements of `c`.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

690: (4)           c : array_like
691: (8)           Array of coefficients ordered so that the coefficients for terms of
692: (8)           degree n are contained in c[n]. If `c` is multidimensional the
693: (8)           remaining indices enumerate multiple polynomials. In the two
694: (8)           dimensional case the coefficients may be thought of as stored in
695: (8)           the columns of `c`.
696: (4)           tensor : boolean, optional
697: (8)           If True, the shape of the coefficient array is extended with ones
698: (8)           on the right, one for each dimension of `x`. Scalars have dimension 0
699: (8)           for this action. The result is that every column of coefficients in
700: (8)           `c` is evaluated for every element of `x`. If False, `x` is broadcast
701: (8)           over the columns of `c` for the evaluation. This keyword is useful
702: (8)           when `c` is multidimensional. The default value is True.
703: (8)           .. versionadded:: 1.7.0
704: (4)           Returns
705: (4)           -----
706: (4)           values : ndarray, algebra_like
707: (8)           The shape of the return value is described above.
708: (4)           See Also
709: (4)           -----
710: (4)           legval2d, leggrid2d, legval3d, leggrid3d
711: (4)           Notes
712: (4)           -----
713: (4)           The evaluation uses Clenshaw recursion, aka synthetic division.
714: (4)
715: (4)           """
716: (4)           c = np.array(c, ndmin=1, copy=False)
717: (8)           if c.dtype.char in '?bBhHiIlLqQpP':
718: (8)               c = c.astype(np.double)
719: (8)           if isinstance(x, (tuple, list)):
720: (8)               x = np.asarray(x)
721: (8)           if isinstance(x, np.ndarray) and tensor:
722: (8)               c = c.reshape(c.shape + (1,)*x.ndim)
723: (8)           if len(c) == 1:
724: (8)               c0 = c[0]
725: (8)               c1 = 0
726: (8)           elif len(c) == 2:
727: (8)               c0 = c[0]
728: (8)               c1 = c[1]
729: (8)           else:
730: (8)               nd = len(c)
731: (8)               c0 = c[-2]
732: (8)               c1 = c[-1]
733: (12)              for i in range(3, len(c) + 1):
734: (12)                  tmp = c0
735: (12)                  nd = nd - 1
736: (12)                  c0 = c[-i] - (c1*(nd - 1))/nd
737: (12)                  c1 = tmp + (c1*x*(2*nd - 1))/nd
738: (0)           return c0 + c1*x
739: (4)           def legval2d(x, y, c):
740: (4)           """
741: (4)           Evaluate a 2-D Legendre series at points (x, y).
742: (4)           This function returns the values:
743: (4)           .. math:: p(x,y) = \sum_{i,j} c_{i,j} * L_i(x) * L_j(y)
744: (4)           The parameters `x` and `y` are converted to arrays only if they are
745: (4)           tuples or a lists, otherwise they are treated as a scalars and they
746: (4)           must have the same shape after conversion. In either case, either `x`
747: (4)           and `y` or their elements must support multiplication and addition both
748: (4)           with themselves and with the elements of `c` .
749: (4)           If `c` is a 1-D array a one is implicitly appended to its shape to make
750: (4)           it 2-D. The shape of the result will be c.shape[2:] + x.shape.
751: (4)           Parameters
752: (4)           -----
753: (4)           x, y : array_like, compatible objects
754: (8)           The two dimensional series is evaluated at the points `(x, y)`,
755: (8)           where `x` and `y` must have the same shape. If `x` or `y` is a list
756: (8)           or tuple, it is first converted to an ndarray, otherwise it is left
757: (8)           unchanged and if it isn't an ndarray it is treated as a scalar.
758: (8)           c : array_like
759: (8)           Array of coefficients ordered so that the coefficient of the term

```

```

759: (8)          of multi-degree i,j is contained in ``c[i,j]``. If `c` has
760: (8)          dimension greater than two the remaining indices enumerate multiple
761: (8)          sets of coefficients.
762: (4)          Returns
763: (4)
764: (4)          values : ndarray, compatible object
765: (8)          The values of the two dimensional Legendre series at points formed
766: (8)          from pairs of corresponding values from `x` and `y`.
767: (4)          See Also
768: (4)
769: (4)          legval, leggrid2d, legval3d, leggrid3d
770: (4)          Notes
771: (4)
772: (4)          .. versionadded:: 1.7.0
773: (4)
774: (4)          return pu._valnd(legval, c, x, y)
775: (0)          def leggrid2d(x, y, c):
776: (4)          """
777: (4)          Evaluate a 2-D Legendre series on the Cartesian product of x and y.
778: (4)          This function returns the values:
779: (4)          .. math:: p(a,b) = \sum_{i,j} c_{i,j} * L_i(a) * L_j(b)
780: (4)          where the points `(a, b)` consist of all pairs formed by taking
781: (4)          `a` from `x` and `b` from `y`. The resulting points form a grid with
782: (4)          `x` in the first dimension and `y` in the second.
783: (4)          The parameters `x` and `y` are converted to arrays only if they are
784: (4)          tuples or a lists, otherwise they are treated as a scalars. In either
785: (4)          case, either `x` and `y` or their elements must support multiplication
786: (4)          and addition both with themselves and with the elements of `c`.
787: (4)          If `c` has fewer than two dimensions, ones are implicitly appended to
788: (4)          its shape to make it 2-D. The shape of the result will be c.shape[2:] +
789: (4)          x.shape + y.shape.
790: (4)          Parameters
791: (4)
792: (4)          x, y : array_like, compatible objects
793: (8)          The two dimensional series is evaluated at the points in the
794: (8)          Cartesian product of `x` and `y`. If `x` or `y` is a list or
795: (8)          tuple, it is first converted to an ndarray, otherwise it is left
796: (8)          unchanged and, if it isn't an ndarray, it is treated as a scalar.
797: (4)          c : array_like
798: (8)          Array of coefficients ordered so that the coefficient of the term of
799: (8)          multi-degree i,j is contained in `c[i,j]`. If `c` has dimension
800: (8)          greater than two the remaining indices enumerate multiple sets of
801: (8)          coefficients.
802: (4)          Returns
803: (4)
804: (4)          values : ndarray, compatible object
805: (8)          The values of the two dimensional Chebyshev series at points in the
806: (8)          Cartesian product of `x` and `y`.
807: (4)          See Also
808: (4)
809: (4)          legval, legval2d, legval3d, leggrid3d
810: (4)          Notes
811: (4)
812: (4)          .. versionadded:: 1.7.0
813: (4)
814: (4)          return pu._gridnd(legval, c, x, y)
815: (0)          def legval3d(x, y, z, c):
816: (4)          """
817: (4)          Evaluate a 3-D Legendre series at points (x, y, z).
818: (4)          This function returns the values:
819: (4)          .. math:: p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * L_i(x) * L_j(y) * L_k(z)
820: (4)          The parameters `x`, `y`, and `z` are converted to arrays only if
821: (4)          they are tuples or a lists, otherwise they are treated as a scalars and
822: (4)          they must have the same shape after conversion. In either case, either
823: (4)          `x`, `y`, and `z` or their elements must support multiplication and
824: (4)          addition both with themselves and with the elements of `c`.
825: (4)          If `c` has fewer than 3 dimensions, ones are implicitly appended to its
826: (4)          shape to make it 3-D. The shape of the result will be c.shape[3:] +
827: (4)          x.shape.

```

```

828: (4)                               Parameters
829: (4)-----x, y, z : array_like, compatible object
830: (4)                               The three dimensional series is evaluated at the points
831: (8)                               `(x, y, z)`, where `x`, `y`, and `z` must have the same shape. If
832: (8)                               any of `x`, `y`, or `z` is a list or tuple, it is first converted
833: (8)                               to an ndarray, otherwise it is left unchanged and if it isn't an
834: (8)                               ndarray it is treated as a scalar.
835: (8)c : array_like
836: (4)                               Array of coefficients ordered so that the coefficient of the term of
837: (8)                               multi-degree i,j,k is contained in ``c[i,j,k]``. If `c` has dimension
838: (8)                               greater than 3 the remaining indices enumerate multiple sets of
839: (8)                               coefficients.
840: (8)-----Returnsthe values : ndarray, compatible object
841: (4)-----values : ndarray, compatible object
842: (4)-----The values of the multidimensional polynomial on points formed with
843: (4)-----triples of corresponding values from `x`, `y`, and `z`.
844: (8)-----See Also
845: (8)-----legval, legval2d, leggrid2d, leggrid3d
846: (4)-----Notes
847: (4)-----.. versionadded:: 1.7.0
848: (4)-----"""
849: (4)-----def leggrid3d(x, y, z, c):
850: (4)-----"""
851: (4)-----Evaluate a 3-D Legendre series on the Cartesian product of x, y, and z.
852: (4)-----This function returns the values:
853: (4)-----.. math:: p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * L_i(a) * L_j(b) * L_k(c)
854: (4)-----where the points `(a, b, c)` consist of all triples formed by taking
855: (4)-----`a` from `x`, `b` from `y`, and `c` from `z`. The resulting points form
856: (4)-----a grid with `x` in the first dimension, `y` in the second, and `z` in
857: (4)-----the third.
858: (4)-----The parameters `x`, `y`, and `z` are converted to arrays only if they
859: (4)-----are tuples or a lists, otherwise they are treated as a scalars. In
860: (4)-----either case, either `x`, `y`, and `z` or their elements must support
861: (4)-----multiplication and addition both with themselves and with the elements
862: (4)-----of `c`.
863: (4)-----If `c` has fewer than three dimensions, ones are implicitly appended to
864: (4)-----its shape to make it 3-D. The shape of the result will be c.shape[3:] +
865: (4)-----x.shape + y.shape + z.shape.
866: (4)-----Parameters
867: (4)-----x, y, z : array_like, compatible objects
868: (8)-----The three dimensional series is evaluated at the points in the
869: (8)-----Cartesian product of `x`, `y`, and `z`. If `x`, `y`, or `z` is a
870: (8)-----list or tuple, it is first converted to an ndarray, otherwise it is
871: (8)-----left unchanged and, if it isn't an ndarray, it is treated as a
872: (8)-----scalar.
873: (4)-----c : array_like
874: (8)-----Array of coefficients ordered so that the coefficients for terms of
875: (8)-----degree i,j are contained in ``c[i,j]``. If `c` has dimension
876: (8)-----greater than two the remaining indices enumerate multiple sets of
877: (8)-----coefficients.
878: (4)-----Returns
879: (4)-----values : ndarray, compatible object
880: (8)-----The values of the two dimensional polynomial at points in the
881: (8)-----product of `x` and `y`.
882: (8)-----See Also
883: (4)-----legval, legval2d, leggrid2d, legval3d
884: (4)-----Notes
885: (4)-----.. versionadded:: 1.7.0
886: (4)-----"""
887: (4)-----Cartesian
888: (8)-----product of `x` and `y`.
889: (4)-----See Also
890: (4)-----legval, legval2d, leggrid2d, legval3d
891: (4)-----Notes
892: (4)-----.. versionadded:: 1.7.0
893: (4)-----"""
894: (4)-----"""
895: (4)-----"""

```

```

896: (4)             return pu._gridnd(legval, c, x, y, z)
897: (0)             def legvander(x, deg):
898: (4)                 """Pseudo-Vandermonde matrix of given degree.
899: (4)                 Returns the pseudo-Vandermonde matrix of degree `deg` and sample points
900: (4)                 `x`. The pseudo-Vandermonde matrix is defined by
901: (4)                 .. math:: V[..., i] = L_i(x)
902: (4)                 where `0 <= i <= deg`. The leading indices of `V` index the elements of
903: (4)                 `x` and the last index is the degree of the Legendre polynomial.
904: (4)                 If `c` is a 1-D array of coefficients of length `n + 1` and `V` is the
905: (4)                 array ``V = legvander(x, n)``, then ``np.dot(V, c)`` and
906: (4)                 ``legval(x, c)`` are the same up to roundoff. This equivalence is
907: (4)                 useful both for least squares fitting and for the evaluation of a large
908: (4)                 number of Legendre series of the same degree and sample points.
909: (4)             Parameters
910: (4)             -----
911: (4)             x : array_like
912: (8)                 Array of points. The dtype is converted to float64 or complex128
913: (8)                 depending on whether any of the elements are complex. If `x` is
914: (8)                 scalar it is converted to a 1-D array.
915: (4)             deg : int
916: (8)                 Degree of the resulting matrix.
917: (4)             Returns
918: (4)             -----
919: (4)             vander : ndarray
920: (8)                 The pseudo-Vandermonde matrix. The shape of the returned matrix is
921: (8)                 ``x.shape + (deg + 1,)``, where The last index is the degree of the
922: (8)                 corresponding Legendre polynomial. The dtype will be the same as
923: (8)                 the converted `x`.
924: (4)             """
925: (4)             ideg = pu._deprecate_as_int(deg, "deg")
926: (4)             if ideg < 0:
927: (8)                 raise ValueError("deg must be non-negative")
928: (4)             x = np.array(x, copy=False, ndmin=1) + 0.0
929: (4)             dims = (ideg + 1,) + x.shape
930: (4)             dtyp = x.dtype
931: (4)             v = np.empty(dims, dtype=dtyp)
932: (4)             v[0] = x*0 + 1
933: (4)             if ideg > 0:
934: (8)                 v[1] = x
935: (8)                 for i in range(2, ideg + 1):
936: (12)                     v[i] = (v[i-1]*x*(2*i - 1) - v[i-2]*(i - 1))/i
937: (4)             return np.moveaxis(v, 0, -1)
938: (0)             def legvander2d(x, y, deg):
939: (4)                 """Pseudo-Vandermonde matrix of given degrees.
940: (4)                 Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
941: (4)                 points `(x, y)`. The pseudo-Vandermonde matrix is defined by
942: (4)                 .. math:: V[..., (deg[1] + 1)*i + j] = L_i(x) * L_j(y),
943: (4)                 where `0 <= i <= deg[0]` and `0 <= j <= deg[1]`. The leading indices of
944: (4)                 `V` index the points `(x, y)` and the last index encodes the degrees of
945: (4)                 the Legendre polynomials.
946: (4)                 If ``V = legvander2d(x, y, [xdeg, ydeg])``, then the columns of `V`
947: (4)                 correspond to the elements of a 2-D coefficient array `c` of shape
948: (4)                 `(xdeg + 1, ydeg + 1)` in the order
949: (4)                 .. math:: c_{\{0\}}, c_{\{01\}}, c_{\{02\}} \dots, c_{\{10\}}, c_{\{11\}}, c_{\{12\}} \dots
950: (4)                 and ``np.dot(V, c.flat)`` and ``legval2d(x, y, c)`` will be the same
951: (4)                 up to roundoff. This equivalence is useful both for least squares
952: (4)                 fitting and for the evaluation of a large number of 2-D Legendre
953: (4)                 series of the same degrees and sample points.
954: (4)             Parameters
955: (4)             -----
956: (4)             x, y : array_like
957: (8)                 Arrays of point coordinates, all of the same shape. The dtypes
958: (8)                 will be converted to either float64 or complex128 depending on
959: (8)                 whether any of the elements are complex. Scalars are converted to
960: (8)                 1-D arrays.
961: (4)             deg : list of ints
962: (8)                 List of maximum degrees of the form [x_deg, y_deg].
963: (4)             Returns
964: (4)             -----

```

```

965: (4)           vander2d : ndarray
966: (8)             The shape of the returned matrix is ``x.shape + (order,)`` , where
967: (8)             :math:`order = (deg[0]+1)*(deg[1]+1)` . The dtype will be the same
968: (8)             as the converted `x` and `y` .
969: (4)           See Also
970: (4)             -----
971: (4)             legvander, legvander3d, legval2d, legval3d
972: (4)           Notes
973: (4)             -----
974: (4)             .. versionadded:: 1.7.0
975: (4)             """
976: (4)             return pu._vander_nd_flat((legvander, legvander), (x, y), deg)
977: (0)           def legvander3d(x, y, z, deg):
978: (4)             """Pseudo-Vandermonde matrix of given degrees.
979: (4)             Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
980: (4)             points `(x, y, z)` . If `l, m, n` are the given degrees in `x, y, z` ,
981: (4)             then The pseudo-Vandermonde matrix is defined by
982: (4)             .. math:: V[..., (m+1)(n+1)i + (n+1)j + k] = L_i(x)*L_j(y)*L_k(z),
983: (4)             where `0 <= i <= l` , `0 <= j <= m` , and `0 <= j <= n` . The leading
984: (4)             indices of `V` index the points `(x, y, z)` and the last index encodes
985: (4)             the degrees of the Legendre polynomials.
986: (4)             If ``V = legvander3d(x, y, z, [xdeg, ydeg, zdeg])`` , then the columns
987: (4)             of `V` correspond to the elements of a 3-D coefficient array `c` of
988: (4)             shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order
989: (4)             .. math:: c_{000}, c_{001}, c_{002}, \dots, c_{010}, c_{011}, c_{012}, \dots
990: (4)             and ``np.dot(V, c.flat)`` and ``legval3d(x, y, z, c)`` will be the
991: (4)             same up to roundoff. This equivalence is useful both for least squares
992: (4)             fitting and for the evaluation of a large number of 3-D Legendre
993: (4)             series of the same degrees and sample points.
994: (4)           Parameters
995: (4)             -----
996: (4)             x, y, z : array_like
997: (8)               Arrays of point coordinates, all of the same shape. The dtypes will
998: (8)               be converted to either float64 or complex128 depending on whether
999: (8)               any of the elements are complex. Scalars are converted to 1-D
1000: (8)              arrays.
1001: (4)             deg : list of ints
1002: (8)               List of maximum degrees of the form [x_deg, y_deg, z_deg].
1003: (4)           Returns
1004: (4)             -----
1005: (4)             vander3d : ndarray
1006: (8)               The shape of the returned matrix is ``x.shape + (order,)`` , where
1007: (8)               :math:`order = (deg[0]+1)*(deg[1]+1)*(deg[2]+1)` . The dtype will
1008: (8)               be the same as the converted `x` , `y` , and `z` .
1009: (4)           See Also
1010: (4)             -----
1011: (4)             legvander, legvander3d, legval2d, legval3d
1012: (4)           Notes
1013: (4)             -----
1014: (4)             .. versionadded:: 1.7.0
1015: (4)             """
1016: (4)             return pu._vander_nd_flat((legvander, legvander, legvander), (x, y, z),
1017: (0)           deg)
1018: (4)           def legfit(x, y, deg, rcond=None, full=False, w=None):
1019: (4)             """
1020: (4)               Least squares fit of Legendre series to data.
1021: (4)               Return the coefficients of a Legendre series of degree `deg` that is the
1022: (4)               least squares fit to the data values `y` given at points `x` . If `y` is
1023: (4)               1-D the returned coefficients will also be 1-D. If `y` is 2-D multiple
1024: (4)               fits are done, one for each column of `y` , and the resulting
1025: (4)               coefficients are stored in the corresponding columns of a 2-D return.
1026: (4)               The fitted polynomial(s) are in the form
1027: (4)               .. math:: p(x) = c_0 + c_1 * L_1(x) + \dots + c_n * L_n(x),
1028: (4)               where `n` is `deg` .
1029: (4)           Parameters
1030: (4)             -----
1031: (8)             x : array_like, shape (M,)
1032: (4)               x-coordinates of the M sample points ``x[i], y[i]`` .
1031: (8)             y : array_like, shape (M,) or (M, K)

```

```

1033: (8) y-coordinates of the sample points. Several data sets of sample
1034: (8) points sharing the same x-coordinates can be fitted at once by
1035: (8) passing in a 2D-array that contains one dataset per column.
1036: (4) deg : int or 1-D array_like
1037: (8) Degree(s) of the fitting polynomials. If `deg` is a single integer
1038: (8) all terms up to and including the `deg`'th term are included in the
1039: (8) fit. For NumPy versions >= 1.11.0 a list of integers specifying the
1040: (8) degrees of the terms to include may be used instead.
1041: (4) rcond : float, optional
1042: (8) Relative condition number of the fit. Singular values smaller than
1043: (8) this relative to the largest singular value will be ignored. The
1044: (8) default value is  $\text{len}(x)*\text{eps}$ , where  $\text{eps}$  is the relative precision of
1045: (8) the float type, about  $2e-16$  in most cases.
1046: (4) full : bool, optional
1047: (8) Switch determining nature of return value. When it is False (the
1048: (8) default) just the coefficients are returned, when True diagnostic
1049: (8) information from the singular value decomposition is also returned.
1050: (4) w : array_like, shape (M,), optional
1051: (8) Weights. If not None, the weight ``w[i]`` applies to the unsquared
1052: (8) residual ``y[i] - y_hat[i]`` at ``x[i]``. Ideally the weights are
1053: (8) chosen so that the errors of the products ``w[i]*y[i]`` all have the
1054: (8) same variance. When using inverse-variance weighting, use
1055: (8) ``w[i] = 1/sigma(y[i])``. The default value is None.
1056: (8) .. versionadded:: 1.5.0
1057: (4) Returns
1058: (4) -----
1059: (4) coef : ndarray, shape (M,) or (M, K)
1060: (8) Legendre coefficients ordered from low to high. If `y` was
1061: (8) 2-D, the coefficients for the data in column k of `y` are in
1062: (8) column `k`. If `deg` is specified as a list, coefficients for
1063: (8) terms not included in the fit are set equal to zero in the
1064: (8) returned `coef`.
1065: (4) [residuals, rank, singular_values, rcond] : list
1066: (8) These values are only returned if ``full == True``
1067: (8) - residuals -- sum of squared residuals of the least squares fit
1068: (8) - rank -- the numerical rank of the scaled Vandermonde matrix
1069: (8) - singular_values -- singular values of the scaled Vandermonde matrix
1070: (8) - rcond -- value of `rcond`.
1071: (8) For more details, see `numpy.linalg.lstsq`.
1072: (4) Warns
1073: (4) -----
1074: (4) RankWarning
1075: (8) The rank of the coefficient matrix in the least-squares fit is
1076: (8) deficient. The warning is only raised if ``full == False``. The
1077: (8) warnings can be turned off by
1078: (8) >>> import warnings
1079: (8) >>> warnings.simplefilter('ignore', np.RankWarning)
1080: (4) See Also
1081: (4) -----
1082: (4) numpy.polynomial.polynomial.polyfit
1083: (4) numpy.polynomial.chebyshev.chebfit
1084: (4) numpy.polynomial.laguerre.lagfit
1085: (4) numpy.polynomial.hermite.hermfit
1086: (4) numpy.polynomial.hermite_e.hermefit
1087: (4) legval : Evaluates a Legendre series.
1088: (4) legvander : Vandermonde matrix of Legendre series.
1089: (4) legweight : Legendre weight function (= 1).
1090: (4) numpy.linalg.lstsq : Computes a least-squares fit from the matrix.
1091: (4) scipy.interpolate.UnivariateSpline : Computes spline fits.
1092: (4) Notes
1093: (4) -----
1094: (4) The solution is the coefficients of the Legendre series `p` that
1095: (4) minimizes the sum of the weighted squared errors
1096: (4) .. math:: E = \sum_j w_j^2 * |y_j - p(x_j)|^2,
1097: (4) where :math:`w_j` are the weights. This problem is solved by setting up
1098: (4) as the (typically) overdetermined matrix equation
1099: (4) .. math:: V(x) * c = w * y,
1100: (4) where `V` is the weighted pseudo Vandermonde matrix of `x`, `c` are the
1101: (4) coefficients to be solved for, `w` are the weights, and `y` are the

```

```

1102: (4) observed values. This equation is then solved using the singular value
1103: (4) decomposition of `V`.
1104: (4) If some of the singular values of `V` are so small that they are
1105: (4) neglected, then a `RankWarning` will be issued. This means that the
1106: (4) coefficient values may be poorly determined. Using a lower order fit
1107: (4) will usually get rid of the warning. The `rcond` parameter can also be
1108: (4) set to a value smaller than its default, but the resulting fit may be
1109: (4) spurious and have large contributions from roundoff error.
1110: (4) Fits using Legendre series are usually better conditioned than fits
1111: (4) using power series, but much can depend on the distribution of the
1112: (4) sample points and the smoothness of the data. If the quality of the fit
1113: (4) is inadequate splines may be a good alternative.
1114: (4) References
1115: (4) -----
1116: (4) .. [1] Wikipedia, "Curve fitting",
1117: (11) https://en.wikipedia.org/wiki/Curve_fitting
1118: (4) Examples
1119: (4) -----
1120: (4) """
1121: (4)     return pu._fit(legvander, x, y, deg, rcond, full, w)
1122: (0) def legcompanion(c):
1123: (4) """Return the scaled companion matrix of c.
1124: (4) The basis polynomials are scaled so that the companion matrix is
1125: (4) symmetric when `c` is an Legendre basis polynomial. This provides
1126: (4) better eigenvalue estimates than the unscaled case and for basis
1127: (4) polynomials the eigenvalues are guaranteed to be real if
1128: (4) `numpy.linalg.eigvalsh` is used to obtain them.
1129: (4) Parameters
1130: (4) -----
1131: (4)     c : array_like
1132: (8)         1-D array of Legendre series coefficients ordered from low to high
1133: (8)         degree.
1134: (4) Returns
1135: (4) -----
1136: (4)     mat : ndarray
1137: (8)         Scaled companion matrix of dimensions (deg, deg).
1138: (4) Notes
1139: (4) -----
1140: (4) .. versionadded:: 1.7.0
1141: (4) """
1142: (4)     [c] = pu.as_series([c])
1143: (4)     if len(c) < 2:
1144: (8)         raise ValueError('Series must have maximum degree of at least 1.')
1145: (4)     if len(c) == 2:
1146: (8)         return np.array([[[-c[0]/c[1]]]])
1147: (4)     n = len(c) - 1
1148: (4)     mat = np.zeros((n, n), dtype=c.dtype)
1149: (4)     scl = 1./np.sqrt(2*np.arange(n) + 1)
1150: (4)     top = mat.reshape(-1)[1::n+1]
1151: (4)     bot = mat.reshape(-1)[n::n+1]
1152: (4)     top[...] = np.arange(1, n)*scl[:n-1]*scl[1:n]
1153: (4)     bot[...] = top
1154: (4)     mat[:, -1] -= (c[:-1]/c[-1])*(scl/scl[-1])*(n/(2*n - 1))
1155: (4)     return mat
1156: (0) def legroots(c):
1157: (4) """
1158: (4) Compute the roots of a Legendre series.
1159: (4) Return the roots (a.k.a. "zeros") of the polynomial
1160: (4) .. math:: p(x) = \sum_i c[i] * L_i(x).
1161: (4) Parameters
1162: (4) -----
1163: (4)     c : 1-D array_like
1164: (8)         1-D array of coefficients.
1165: (4) Returns
1166: (4) -----
1167: (4)     out : ndarray
1168: (8)         Array of the roots of the series. If all the roots are real,
1169: (8)         then `out` is also real, otherwise it is complex.
1170: (4) See Also

```

```

1171: (4) -----
1172: (4) numpy.polynomial.polynomial.polyroots
1173: (4) numpy.polynomial.chebyshev.chebroots
1174: (4) numpy.polynomial.laguerre.lagroots
1175: (4) numpy.polynomial.hermite.hermroots
1176: (4) numpy.polynomial.hermite_e.hermroots
1177: (4) Notes
1178: (4) -----
1179: (4) The root estimates are obtained as the eigenvalues of the companion
1180: (4) matrix, Roots far from the origin of the complex plane may have large
1181: (4) errors due to the numerical instability of the series for such values.
1182: (4) Roots with multiplicity greater than 1 will also show larger errors as
1183: (4) the value of the series near such points is relatively insensitive to
1184: (4) errors in the roots. Isolated roots near the origin can be improved by
1185: (4) a few iterations of Newton's method.
1186: (4) The Legendre series basis polynomials aren't powers of ``x`` so the
1187: (4) results of this function may seem unintuitive.
1188: (4) Examples
1189: (4) -----
1190: (4) >>> import numpy.polynomial.legendre as leg
1191: (4) >>> leg.legroots((1, 2, 3, 4)) # 4L_3 + 3L_2 + 2L_1 + 1L_0, all real roots
1192: (4) array([-0.85099543, -0.11407192,  0.51506735]) # may vary
1193: (4) """
1194: (4) [c] = pu.as_series([c])
1195: (4) if len(c) < 2:
1196: (8)     return np.array([], dtype=c.dtype)
1197: (4) if len(c) == 2:
1198: (8)     return np.array([-c[0]/c[1]])
1199: (4) m = legcompanion(c)[::-1, ::-1]
1200: (4) r = la.eigvals(m)
1201: (4) r.sort()
1202: (4) return r
1203: (0) def leggauss(deg):
1204: (4) """
1205: (4) Gauss-Legendre quadrature.
1206: (4) Computes the sample points and weights for Gauss-Legendre quadrature.
1207: (4) These sample points and weights will correctly integrate polynomials of
1208: (4) degree :math:`2*deg - 1` or less over the interval :math:`[-1, 1]` with
1209: (4) the weight function :math:`f(x) = 1`.
1210: (4) Parameters
1211: (4) -----
1212: (4) deg : int
1213: (8)     Number of sample points and weights. It must be >= 1.
1214: (4) Returns
1215: (4) -----
1216: (4) x : ndarray
1217: (8)     1-D ndarray containing the sample points.
1218: (4) y : ndarray
1219: (8)     1-D ndarray containing the weights.
1220: (4) Notes
1221: (4) -----
1222: (4) .. versionadded:: 1.7.0
1223: (4) The results have only been tested up to degree 100, higher degrees may
1224: (4) be problematic. The weights are determined by using the fact that
1225: (4) .. math:: w_k = c / (L'_n(x_k) * L_{n-1}(x_k))
1226: (4) where :math:`c` is a constant independent of :math:`k` and :math:`x_k`
1227: (4) is the k'th root of :math:`L_n`, and then scaling the results to get
1228: (4) the right value when integrating 1.
1229: (4) """
1230: (4) ideg = pu._deprecate_as_int(deg, "deg")
1231: (4) if ideg <= 0:
1232: (8)     raise ValueError("deg must be a positive integer")
1233: (4) c = np.array([0]*deg + [1])
1234: (4) m = legcompanion(c)
1235: (4) x = la.eigvalsh(m)
1236: (4) dy = legval(x, c)
1237: (4) df = legval(x, legder(c))
1238: (4) x -= dy/df
1239: (4) fm = legval(x, c[1:])

```

```

1240: (4)         fm /= np.abs(fm).max()
1241: (4)         df /= np.abs(df).max()
1242: (4)         w = 1/(fm * df)
1243: (4)         w = (w + w[::-1])/2
1244: (4)         x = (x - x[::-1])/2
1245: (4)         w *= 2. / w.sum()
1246: (4)         return x, w
1247: (0)     def legweight(x):
1248: (4)         """
1249: (4)             Weight function of the Legendre polynomials.
1250: (4)             The weight function is :math:`1` and the interval of integration is
1251: (4)             :math:`[-1, 1]`. The Legendre polynomials are orthogonal, but not
1252: (4)             normalized, with respect to this weight function.
1253: (4)             Parameters
1254: (4)             -----
1255: (4)             x : array_like
1256: (7)                 Values at which the weight function will be computed.
1257: (4)             Returns
1258: (4)             -----
1259: (4)             w : ndarray
1260: (7)                 The weight function at `x`.
1261: (4)             Notes
1262: (4)             -----
1263: (4)             .. versionadded:: 1.7.0
1264: (4)             """
1265: (4)             w = x*0.0 + 1.0
1266: (4)             return w
1267: (0)     class Legendre(ABCPolyBase):
1268: (4)         """A Legendre series class.
1269: (4)             The Legendre class provides the standard Python numerical methods
1270: (4)             '+', '-', '*', '//', '%', 'divmod', '**', and '(' as well as the
1271: (4)             attributes and methods listed in the `ABCPolyBase` documentation.
1272: (4)             Parameters
1273: (4)             -----
1274: (4)             coef : array_like
1275: (8)                 Legendre coefficients in order of increasing degree, i.e.,
1276: (8)                 ``(1, 2, 3)`` gives ``1*P_0(x) + 2*P_1(x) + 3*P_2(x)``.
1277: (4)             domain : (2,) array_like, optional
1278: (8)                 Domain to use. The interval ``[domain[0], domain[1]]`` is mapped
1279: (8)                 to the interval ``[window[0], window[1]]`` by shifting and scaling.
1280: (8)                 The default value is [-1, 1].
1281: (4)             window : (2,) array_like, optional
1282: (8)                 Window, see `domain` for its use. The default value is [-1, 1].
1283: (8)                 .. versionadded:: 1.6.0
1284: (4)             symbol : str, optional
1285: (8)                 Symbol used to represent the independent variable in string
1286: (8)                 representations of the polynomial expression, e.g. for printing.
1287: (8)                 The symbol must be a valid Python identifier. Default value is 'x'.
1288: (8)                 .. versionadded:: 1.24
1289: (4)             """
1290: (4)             _add = staticmethod(legadd)
1291: (4)             _sub = staticmethod(legsub)
1292: (4)             _mul = staticmethod(legmul)
1293: (4)             _div = staticmethod(legdiv)
1294: (4)             _pow = staticmethod(legpow)
1295: (4)             _val = staticmethod(legval)
1296: (4)             _int = staticmethod(legint)
1297: (4)             _der = staticmethod(legder)
1298: (4)             _fit = staticmethod(legfit)
1299: (4)             _line = staticmethod(legline)
1300: (4)             _roots = staticmethod(legroots)
1301: (4)             _fromroots = staticmethod(legfromroots)
1302: (4)             domain = np.array(legdomain)
1303: (4)             window = np.array(legdomain)
1304: (4)             basis_name = 'P'

```

```
1: (0)
2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (0)
8: (0)
9: (0)
10: (0)
11: (0)
12: (0)
13: (3)
14: (3)
15: (0)
16: (0)
17: (0)
18: (3)
19: (3)
20: (3)
21: (3)
22: (3)
23: (0)
24: (0)
25: (0)
26: (3)
27: (3)
28: (3)
29: (3)
30: (3)
31: (3)
32: (3)
33: (3)
34: (3)
35: (3)
36: (3)
37: (3)
38: (0)
39: (0)
40: (0)
41: (3)
42: (3)
43: (3)
44: (0)
45: (0)
46: (0)
47: (3)
48: (3)
49: (3)
50: (3)
51: (3)
52: (3)
53: (3)
54: (3)
55: (3)
56: (3)
57: (3)
58: (0)
59: (0)
60: (0)
61: (0)
62: (0)
63: (4)
64: (4)
65: (4)
66: (4)
'polyval3d',
67: (4)      """
```

Power Series (:mod:`numpy.polynomial.polynomial`)

This module provides a number of objects (mostly functions) useful for dealing with polynomials, including a `Polynomial` class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with polynomial objects is in the docstring for its "parent" sub-package, `numpy.polynomial`).

Classes

-----

.. autosummary::  
 :toctree: generated/  
 Polynomial

Constants

-----

.. autosummary::  
 :toctree: generated/  
 polydomain  
 polyzero  
 polyone  
 polyx

Arithmetic

-----

.. autosummary::  
 :toctree: generated/  
 polyadd  
 polysub  
 polymulx  
 polymul  
 polydiv  
 polypow  
 polyval  
 polyval2d  
 polyval3d  
 polygrid2d  
 polygrid3d

Calculus

-----

.. autosummary::  
 :toctree: generated/  
 polyder  
 polyint

Misc Functions

-----

.. autosummary::  
 :toctree: generated/  
 polyfromroots  
 polyroots  
 polyvalfromroots  
 polyvander  
 polyvander2d  
 polyvander3d  
 polycompanion  
 polyfit  
 polytrim  
 polyline

See Also

-----

`numpy.polynomial`

"""

`_all__ = [  
 'polyzero', 'polyone', 'polyx', 'polydomain', 'polyline', 'polyadd',  
 'polysub', 'polymulx', 'polymul', 'polydiv', 'polypow', 'polyval',  
 'polyvalfromroots', 'polyder', 'polyint', 'polyfromroots', 'polyvander',  
 'polyfit', 'polytrim', 'polyroots', 'Polynomial', 'polyval2d',  
 'polyval3d', 'polygrid2d', 'polygrid3d', 'polyvander2d', 'polyvander3d']`

```

68: (0) import numpy as np
69: (0) import numpy.linalg as la
70: (0) from numpy.core.multiarray import normalize_axis_index
71: (0) from . import polyutils as pu
72: (0) from ._polybase import ABCPolyBase
73: (0) polytrim = pu.trimcoef
74: (0) polydomain = np.array([-1, 1])
75: (0) polyzero = np.array([0])
76: (0) polyone = np.array([1])
77: (0) polyx = np.array([0, 1])
78: (0) def polyline(off, scl):
79: (4) """
80: (4)     Returns an array representing a linear polynomial.
81: (4) Parameters
82: (4) -----
83: (4) off, scl : scalars
84: (8)     The "y-intercept" and "slope" of the line, respectively.
85: (4) Returns
86: (4) -----
87: (4) y : ndarray
88: (8)     This module's representation of the linear polynomial ``off +
89: (8)     scl*x``.
90: (4) See Also
91: (4) -----
92: (4)     numpy.polynomial.chebyshev.chebline
93: (4)     numpy.polynomial.legendre.legline
94: (4)     numpy.polynomial.laguerre.lagline
95: (4)     numpy.polynomial.hermite.hermeline
96: (4)     numpy.polynomial.hermite_e.hermeline
97: (4) Examples
98: (4) -----
99: (4) >>> from numpy.polynomial import polynomial as P
100: (4) >>> P.polyline(1,-1)
101: (4) array([ 1, -1])
102: (4) >>> P.polyval(1, P.polyline(1,-1)) # should be 0
103: (4) 0.0
104: (4) """
105: (4) if scl != 0:
106: (8)     return np.array([off, scl])
107: (4) else:
108: (8)     return np.array([off])
109: (0) def polyfromroots(roots):
110: (4) """
111: (4) Generate a monic polynomial with given roots.
112: (4) Return the coefficients of the polynomial
113: (4) .. math:: p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),
114: (4) where the ``r_n`` are the roots specified in `roots`. If a zero has
115: (4) multiplicity n, then it must appear in `roots` n times. For instance,
116: (4) if 2 is a root of multiplicity three and 3 is a root of multiplicity 2,
117: (4) then `roots` looks something like [2, 2, 2, 3, 3]. The roots can appear
118: (4) in any order.
119: (4) If the returned coefficients are `c`, then
120: (4) .. math:: p(x) = c_0 + c_1 * x + ... + x^n
121: (4) The coefficient of the last term is 1 for monic polynomials in this
122: (4) form.
123: (4) Parameters
124: (4) -----
125: (4) roots : array_like
126: (8)     Sequence containing the roots.
127: (4) Returns
128: (4) -----
129: (4) out : ndarray
130: (8)     1-D array of the polynomial's coefficients If all the roots are
131: (8)     real, then `out` is also real, otherwise it is complex. (see
132: (8)     Examples below).
133: (4) See Also
134: (4) -----
135: (4)     numpy.polynomial.chebyshev.chebfromroots
136: (4)     numpy.polynomial.legendre.legfromroots

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

137: (4) numpy.polynomial.laguerre.lagfromroots
138: (4) numpy.polynomial.hermite.hermfromroots
139: (4) numpy.polynomial.hermite_e.hermefromroots
140: (4) Notes
141: (4) -----
142: (4) The coefficients are determined by multiplying together linear factors
143: (4) of the form `` $(x - r_i)$ ``, i.e.
144: (4) .. math:: p(x) = (x - r_0) (x - r_1) \dots (x - r_n)
145: (4) where ``n == len(roots) - 1``; note that this implies that ``1`` is always
146: (4) returned for :math:`a_n`.
147: (4) Examples
148: (4) -----
149: (4) >>> from numpy.polynomial import polynomial as P
150: (4) >>> P.polyfromroots((-1,0,1)) #  $x(x - 1)(x + 1) = x^3 - x$ 
151: (4) array([ 0., -1.,  0.,  1.])
152: (4) >>> j = complex(0,1)
153: (4) >>> P.polyfromroots((-j,j)) # complex returned, though values are real
154: (4) array([1.+0.j,  0.+0.j,  1.+0.j])
155: (4) """
156: (4)     return pu._fromroots(polyline, polymul, roots)
157: (0) def polyadd(c1, c2):
158: (4) """
159: (4) Add one polynomial to another.
160: (4) Returns the sum of two polynomials `c1` + `c2`. The arguments are
161: (4) sequences of coefficients from lowest order term to highest, i.e.,
162: (4) [1,2,3] represents the polynomial `` $1 + 2x + 3x^2$ ``.
163: (4) Parameters
164: (4) -----
165: (4) c1, c2 : array_like
166: (8)     1-D arrays of polynomial coefficients ordered from low to high.
167: (4) Returns
168: (4) -----
169: (4) out : ndarray
170: (8)     The coefficient array representing their sum.
171: (4) See Also
172: (4) -----
173: (4) polysub, polymulx, polymul, polydiv, polypow
174: (4) Examples
175: (4) -----
176: (4) >>> from numpy.polynomial import polynomial as P
177: (4) >>> c1 = (1,2,3)
178: (4) >>> c2 = (3,2,1)
179: (4) >>> sum = P.polyadd(c1,c2); sum
180: (4) array([4.,  4.,  4.])
181: (4) >>> P.polyval(2, sum) #  $4 + 4(2) + 4(2^2)$ 
182: (4) 28.0
183: (4) """
184: (4)     return pu._add(c1, c2)
185: (0) def polysub(c1, c2):
186: (4) """
187: (4) Subtract one polynomial from another.
188: (4) Returns the difference of two polynomials `c1` - `c2`. The arguments
189: (4) are sequences of coefficients from lowest order term to highest, i.e.,
190: (4) [1,2,3] represents the polynomial `` $1 + 2x + 3x^2$ ``.
191: (4) Parameters
192: (4) -----
193: (4) c1, c2 : array_like
194: (8)     1-D arrays of polynomial coefficients ordered from low to
195: (8)     high.
196: (4) Returns
197: (4) -----
198: (4) out : ndarray
199: (8)     Of coefficients representing their difference.
200: (4) See Also
201: (4) -----
202: (4) polyadd, polymulx, polymul, polydiv, polypow
203: (4) Examples
204: (4) -----
205: (4) >>> from numpy.polynomial import polynomial as P

```

```

206: (4)             >>> c1 = (1,2,3)
207: (4)             >>> c2 = (3,2,1)
208: (4)             >>> P.polysub(c1,c2)
209: (4)             array([-2.,  0.,  2.])
210: (4)             >>> P.polysub(c2,c1) # -P.polysub(c1,c2)
211: (4)             array([ 2.,  0., -2.])
212: (4)             """
213: (4)             return pu._sub(c1, c2)
214: (0) def polymulx(c):
215: (4)             """Multiply a polynomial by x.
216: (4)             Multiply the polynomial `c` by x, where x is the independent
217: (4)             variable.
218: (4)             Parameters
219: (4)             -----
220: (4)             c : array_like
221: (8)             1-D array of polynomial coefficients ordered from low to
222: (8)             high.
223: (4)             Returns
224: (4)             -----
225: (4)             out : ndarray
226: (8)             Array representing the result of the multiplication.
227: (4)             See Also
228: (4)             -----
229: (4)             polyadd, polysub, polymul, polydiv, polypow
230: (4)             Notes
231: (4)             -----
232: (4)             .. versionadded:: 1.5.0
233: (4)             """
234: (4)             [c] = pu.as_series([c])
235: (4)             if len(c) == 1 and c[0] == 0:
236: (8)                 return c
237: (4)             prd = np.empty(len(c) + 1, dtype=c.dtype)
238: (4)             prd[0] = c[0]*0
239: (4)             prd[1:] = c
240: (4)             return prd
241: (0) def polymul(c1, c2):
242: (4)             """
243: (4)             Multiply one polynomial by another.
244: (4)             Returns the product of two polynomials `c1` * `c2`. The arguments are
245: (4)             sequences of coefficients, from lowest order term to highest, e.g.,
246: (4)             [1,2,3] represents the polynomial `` $1 + 2x + 3x^2$ ``.
247: (4)             Parameters
248: (4)             -----
249: (4)             c1, c2 : array_like
250: (8)             1-D arrays of coefficients representing a polynomial, relative to the
251: (8)             "standard" basis, and ordered from lowest order term to highest.
252: (4)             Returns
253: (4)             -----
254: (4)             out : ndarray
255: (8)             Of the coefficients of their product.
256: (4)             See Also
257: (4)             -----
258: (4)             polyadd, polysub, polymulx, polydiv, polypow
259: (4)             Examples
260: (4)             -----
261: (4)             >>> from numpy.polynomial import polynomial as P
262: (4)             >>> c1 = (1,2,3)
263: (4)             >>> c2 = (3,2,1)
264: (4)             >>> P.polymul(c1,c2)
265: (4)             array([ 3.,  8.,  14.,  8.,  3.])
266: (4)             """
267: (4)             [c1, c2] = pu.as_series([c1, c2])
268: (4)             ret = np.convolve(c1, c2)
269: (4)             return pu.trimseq(ret)
270: (0) def polydiv(c1, c2):
271: (4)             """
272: (4)             Divide one polynomial by another.
273: (4)             Returns the quotient-with-remainder of two polynomials `c1` / `c2`.
274: (4)             The arguments are sequences of coefficients, from lowest order term

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

275: (4)          to highest, e.g., [1,2,3] represents `` $1 + 2*x + 3*x**2$ ``.
276: (4)          Parameters
277: (4)
278: (4)          c1, c2 : array_like
279: (8)          1-D arrays of polynomial coefficients ordered from low to high.
280: (4)          Returns
281: (4)
282: (4)          [quo, rem] : ndarrays
283: (8)          Of coefficient series representing the quotient and remainder.
284: (4)          See Also
285: (4)
286: (4)          polyadd, polysub, polymulx, polymul, polypow
287: (4)          Examples
288: (4)
289: (4)          >>> from numpy.polynomial import polynomial as P
290: (4)          >>> c1 = (1,2,3)
291: (4)          >>> c2 = (3,2,1)
292: (4)          >>> P.polydiv(c1,c2)
293: (4)          (array([3.]), array([-8., -4.]))
294: (4)          >>> P.polydiv(c2,c1)
295: (4)          (array([ 0.33333333]), array([ 2.66666667,  1.33333333])) # may vary
296: (4)
297: (4)          """
298: (4)          [c1, c2] = pu.as_series([c1, c2])
299: (8)          if c2[-1] == 0:
300: (4)              raise ZeroDivisionError()
301: (4)          lc1 = len(c1)
302: (4)          lc2 = len(c2)
303: (8)          if lc1 < lc2:
304: (4)              return c1[:1]*0, c1
305: (8)          elif lc2 == 1:
306: (4)              return c1/c2[-1], c1[:1]*0
307: (4)          else:
308: (8)              dlen = lc1 - lc2
309: (8)              scl = c2[-1]
310: (8)              c2 = c2[:-1]/scl
311: (8)              i = dlen
312: (8)              j = lc1 - 1
313: (12)             while i >= 0:
314: (12)                 c1[i:j] -= c2*c1[j]
315: (12)                 i -= 1
316: (8)                 j -= 1
317: (0)             return c1[j+1:]/scl, pu.trimseq(c1[:j+1])
def polypow(c, pow, maxpower=None):
    """Raise a polynomial to a power.
    Returns the polynomial `c` raised to the power `pow`. The argument
    `c` is a sequence of coefficients ordered from low to high. i.e.,
    [1,2,3] is the series `` $1 + 2*x + 3*x**2$ ``.
    Parameters
    -----
    c : array_like
        1-D array of array of series coefficients ordered from low to
        high degree.
    pow : integer
        Power to which the series will be raised
    maxpower : integer, optional
        Maximum power allowed. This is mainly to limit growth of the series
        to unmanageable size. Default is 16
    Returns
    -----
    coef : ndarray
        Power series of power.
    See Also
    -----
    polyadd, polysub, polymulx, polymul, polydiv
    Examples
    -----
    >>> from numpy.polynomial import polynomial as P
    >>> P.polypow([1,2,3], 2)
    array([ 1.,  4., 10., 12.,  9.])

```

```

344: (4)
345: (4)
346: (0)
347: (4)
348: (4)
349: (4)
350: (4)
351: (4)
352: (4)
353: (4)
354: (4)
355: (4)
356: (4)
357: (4)
358: (4)
359: (8)
360: (8)
361: (8)
362: (4)
363: (8)
364: (4)
365: (8)
366: (8)
367: (8)
368: (4)
369: (8)
370: (8)
371: (4)
372: (4)
373: (4)
374: (8)
375: (4)
376: (4)
377: (4)
378: (4)
379: (4)
380: (4)
381: (4)
382: (4)
383: (4)
384: (4)
385: (4)
386: (4)
387: (4)
388: (4)
389: (4)
390: (4)
391: (4)
392: (4)
393: (8)
394: (4)
395: (4)
396: (4)
397: (4)
398: (8)
399: (4)
400: (4)
401: (8)
402: (4)
403: (4)
404: (4)
405: (8)
406: (4)
407: (8)
408: (12)
409: (12)
410: (12)
411: (12)
412: (16)

        """
        return pu._pow(np.convolve, c, pow, maxpower)
def polyder(c, m=1, scl=1, axis=0):
    """
    Differentiate a polynomial.
    Returns the polynomial coefficients `c` differentiated `m` times along
    `axis`. At each iteration the result is multiplied by `scl` (the
    scaling factor is for use in a linear change of variable). The
    argument `c` is an array of coefficients from low to high degree along
    each axis, e.g., [1,2,3] represents the polynomial ``1 + 2*x + 3*x**2``
    while [[1,2],[1,2]] represents ``1 + 1*x + 2*y + 2*x*y`` if axis=0 is
    ``x`` and axis=1 is ``y``.
    Parameters
    -----
    c : array_like
        Array of polynomial coefficients. If c is multidimensional the
        different axis correspond to different variables with the degree
        in each axis given by the corresponding index.
    m : int, optional
        Number of derivatives taken, must be non-negative. (Default: 1)
    scl : scalar, optional
        Each differentiation is multiplied by `scl`. The end result is
        multiplication by ``scl**m``. This is for use in a linear change
        of variable. (Default: 1)
    axis : int, optional
        Axis over which the derivative is taken. (Default: 0).
        .. versionadded:: 1.7.0
    Returns
    -----
    der : ndarray
        Polynomial coefficients of the derivative.
    See Also
    -----
    polyint
    Examples
    -----
    >>> from numpy.polynomial import polynomial as P
    >>> c = (1,2,3,4) # 1 + 2x + 3x**2 + 4x**3
    >>> P.polyder(c) # (d/dx)(c) = 2 + 6x + 12x**2
    array([ 2.,  6., 12.])
    >>> P.polyder(c,3) # (d**3/dx**3)(c) = 24
    array([24.])
    >>> P.polyder(c,scl=-1) # (d/d(-x))(c) = -2 - 6x - 12x**2
    array([-2., -6., -12.])
    >>> P.polyder(c,2,-1) # (d**2/d(-x)**2)(c) = 6 + 24x
    array([ 6., 24.])
    """
    c = np.array(c, ndmin=1, copy=True)
    if c.dtype.char in '?bBhHiIlLqOpP':
        c = c + 0.0
    cdt = c.dtype
    cnt = pu._deprecate_as_int(m, "the order of derivation")
    iaxis = pu._deprecate_as_int(axis, "the axis")
    if cnt < 0:
        raise ValueError("The order of derivation must be non-negative")
    iaxis = normalize_axis_index(iaxis, c.ndim)
    if cnt == 0:
        return c
    c = np.moveaxis(c, iaxis, 0)
    n = len(c)
    if cnt >= n:
        c = c[:1]*0
    else:
        for i in range(cnt):
            n = n - 1
            c *= scl
        der = np.empty((n,) + c.shape[1:], dtype=cdt)
        for j in range(n, 0, -1):
            der[j - 1] = j*c[j]

```

```

413: (12)          c = der
414: (4)          c = np.moveaxis(c, 0, iaxis)
415: (4)          return c
416: (0)          def polyint(c, m=1, k=[], lbnd=0, scl=1, axis=0):
417: (4)          """
418: (4)              Integrate a polynomial.
419: (4)          Returns the polynomial coefficients `c` integrated `m` times from
420: (4)          `lbnd` along `axis`. At each iteration the resulting series is
421: (4)          **multiplied** by `scl` and an integration constant, `k`, is added.
422: (4)          The scaling factor is for use in a linear change of variable. ("Buyer
423: (4)          beware": note that, depending on what one is doing, one may want `scl`
424: (4)          to be the reciprocal of what one might expect; for more information,
425: (4)          see the Notes section below.) The argument `c` is an array of
426: (4)          coefficients, from low to high degree along each axis, e.g., [1,2,3]
427: (4)          represents the polynomial `` $1 + 2x + 3x^2$ `` while [[1,2],[1,2]]
428: (4)          represents `` $1 + 1x + 2y + 2xy$ `` if axis=0 is `` $x$ `` and axis=1 is
429: (4)          `` $y$ ``.
430: (4)          Parameters
431: (4)          -----
432: (4)          c : array_like
433: (8)          1-D array of polynomial coefficients, ordered from low to high.
434: (4)          m : int, optional
435: (8)          Order of integration, must be positive. (Default: 1)
436: (4)          k : [], list, scalar}, optional
437: (8)          Integration constant(s). The value of the first integral at zero
438: (8)          is the first value in the list, the value of the second integral
439: (8)          at zero is the second value, etc. If ``k == []`` (the default),
440: (8)          all constants are set to zero. If ``m == 1``, a single scalar can
441: (8)          be given instead of a list.
442: (4)          lbnd : scalar, optional
443: (8)          The lower bound of the integral. (Default: 0)
444: (4)          scl : scalar, optional
445: (8)          Following each integration the result is *multiplied* by `scl`
446: (8)          before the integration constant is added. (Default: 1)
447: (4)          axis : int, optional
448: (8)          Axis over which the integral is taken. (Default: 0).
449: (8)          .. versionadded:: 1.7.0
450: (4)          Returns
451: (4)          -----
452: (4)          S : ndarray
453: (8)          Coefficient array of the integral.
454: (4)          Raises
455: (4)          -----
456: (4)          ValueError
457: (8)          If ``m < 1``, ``len(k) > m``, ``np.ndim(lbnd) != 0``, or
458: (8)          ``np.ndim(scl) != 0``.
459: (4)          See Also
460: (4)          -----
461: (4)          polyder
462: (4)          Notes
463: (4)          -----
464: (4)          Note that the result of each integration is *multiplied* by `scl`. Why
465: (4)          is this important to note? Say one is making a linear change of
466: (4)          variable :math:`u = ax + b` in an integral relative to `x`. Then
467: (4)          :math:`dx = du/a`, so one will need to set `scl` equal to
468: (4)          :math:`1/a` - perhaps not what one would have first thought.
469: (4)          Examples
470: (4)          -----
471: (4)          >>> from numpy.polynomial import polynomial as P
472: (4)          >>> c = (1,2,3)
473: (4)          >>> P.polyint(c) # should return array([0, 1, 1, 1])
474: (4)          array([0., 1., 1., 1.])
475: (4)          >>> P.polyint(c,3) # should return array([0, 0, 0, 1/6, 1/12, 1/20])
476: (5)          array([ 0.          ,  0.          ,  0.          ,  0.16666667,  0.08333333, #
may vary
477: (13)                  0.05        ])
478: (4)          >>> P.polyint(c,k=3) # should return array([3, 1, 1, 1])
479: (4)          array([3., 1., 1., 1.])
480: (4)          >>> P.polyint(c,lbnd=-2) # should return array([6, 1, 1, 1])

```

```

481: (4) array([6., 1., 1., 1.])
482: (4) >>> P.polyint(c,scl=-2) # should return array([0, -2, -2, -2])
483: (4) array([ 0., -2., -2., -2.])
484: (4)
485: (4) """
486: (4) c = np.array(c, ndmin=1, copy=True)
487: (8) if c.dtype.char in '?bBhHiIlLqQpP':
488: (4)     c = c + 0.0
489: (4) cdt = c.dtype
490: (8) if not np.iterable(k):
491: (4)     k = [k]
492: (4) cnt = pu._deprecate_as_int(m, "the order of integration")
493: (4) iaxis = pu._deprecate_as_int(axis, "the axis")
494: (8) if cnt < 0:
495: (4)     raise ValueError("The order of integration must be non-negative")
496: (8) if len(k) > cnt:
497: (4)     raise ValueError("Too many integration constants")
498: (8) if np.ndim(lbnd) != 0:
499: (4)     raise ValueError("lbnd must be a scalar.")
500: (8) if np.ndim(scl) != 0:
501: (4)     raise ValueError("scl must be a scalar.")
502: (4) iaxis = normalize_axis_index(iaxis, c.ndim)
503: (8) if cnt == 0:
504: (4)     return c
505: (4) k = list(k) + [0]*(cnt - len(k))
506: (4) c = np.moveaxis(c, iaxis, 0)
507: (8) for i in range(cnt):
508: (4)     n = len(c)
509: (8)     c *= scl
510: (12)     if n == 1 and np.all(c[0] == 0):
511: (8)         c[0] += k[i]
512: (12)     else:
513: (12)         tmp = np.empty((n + 1,) + c.shape[1:], dtype=cdt)
514: (12)         tmp[0] = c[0]*0
515: (12)         tmp[1] = c[0]
516: (16)         for j in range(1, n):
517: (12)             tmp[j + 1] = c[j]/(j + 1)
518: (12)         tmp[0] += k[i] - polyval(lbnd, tmp)
519: (4)         c = tmp
520: (4)     c = np.moveaxis(c, 0, iaxis)
521: (0)     return c
522: (4)
523: (4) def polyval(x, c, tensor=True):
524: (4) """
525: (4) Evaluate a polynomial at points x.
526: (4) If `c` is of length `n + 1`, this function returns the value
527: (4) .. math:: p(x) = c_0 + c_1 * x + \dots + c_n * x^n
528: (4) The parameter `x` is converted to an array only if it is a tuple or a
529: (4) list, otherwise it is treated as a scalar. In either case, either `x`
530: (4) or its elements must support multiplication and addition both with
531: (4) themselves and with the elements of `c`.
532: (4) If `c` is a 1-D array, then `p(x)` will have the same shape as `x`. If
533: (4) `c` is multidimensional, then the shape of the result depends on the
534: (4) value of `tensor`. If `tensor` is true the shape will be `c.shape[1:] + x.shape`. If `tensor` is false the shape will be `c.shape[1:]`. Note that
535: (4) scalars have shape (,).
536: (4) Trailing zeros in the coefficients will be used in the evaluation, so
537: (4) they should be avoided if efficiency is a concern.
538: (4) Parameters
539: (4) -----
540: (8) x : array_like, compatible object
541: (8)     If `x` is a list or tuple, it is converted to an ndarray, otherwise
542: (8)     it is left unchanged and treated as a scalar. In either case, `x`
543: (8)     or its elements must support addition and multiplication with
544: (4)     themselves and with the elements of `c`.
545: (8) c : array_like
546: (8)     Array of coefficients ordered so that the coefficients for terms of
547: (8)     degree n are contained in c[n]. If `c` is multidimensional the
548: (8)     remaining indices enumerate multiple polynomials. In the two
549: (8)     dimensional case the coefficients may be thought of as stored in

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

550: (4)          tensor : boolean, optional
551: (8)          If True, the shape of the coefficient array is extended with ones
552: (8)          on the right, one for each dimension of `x`. Scalars have dimension 0
553: (8)          for this action. The result is that every column of coefficients in
554: (8)          `c` is evaluated for every element of `x`. If False, `x` is broadcast
555: (8)          over the columns of `c` for the evaluation. This keyword is useful
556: (8)          when `c` is multidimensional. The default value is True.
557: (8)          .. versionadded:: 1.7.0
558: (4)          Returns
559: (4)          -----
560: (4)          values : ndarray, compatible object
561: (8)          The shape of the returned array is described above.
562: (4)          See Also
563: (4)          -----
564: (4)          polyval2d, polygrid2d, polyval3d, polygrid3d
565: (4)          Notes
566: (4)          -----
567: (4)          The evaluation uses Horner's method.
568: (4)          Examples
569: (4)          -----
570: (4)          >>> from numpy.polynomial.polynomial import polyval
571: (4)          >>> polyval(1, [1,2,3])
572: (4)          6.0
573: (4)          >>> a = np.arange(4).reshape(2,2)
574: (4)          >>> a
575: (4)          array([[0, 1],
576: (11)             [2, 3]])
577: (4)          >>> polyval(a, [1,2,3])
578: (4)          array([[ 1.,   6.],
579: (11)             [17.,  34.]])
580: (4)          >>> coef = np.arange(4).reshape(2,2) # multidimensional coefficients
581: (4)          >>> coef
582: (4)          array([[0, 1],
583: (11)             [2, 3]])
584: (4)          >>> polyval([1,2], coef, tensor=True)
585: (4)          array([[2.,  4.],
586: (11)             [4.,  7.]])
587: (4)          >>> polyval([1,2], coef, tensor=False)
588: (4)          array([2.,  7.])
589: (4)          """
590: (4)          c = np.array(c, ndmin=1, copy=False)
591: (4)          if c.dtype.char in '?bBhHiIlLqQpP':
592: (8)              c = c + 0.0
593: (4)          if isinstance(x, (tuple, list)):
594: (8)              x = np.asarray(x)
595: (4)          if isinstance(x, np.ndarray) and tensor:
596: (8)              c = c.reshape(c.shape + (1,)*x.ndim)
597: (4)              c0 = c[-1] + x*0
598: (4)              for i in range(2, len(c) + 1):
599: (8)                  c0 = c[-i] + c0*x
600: (4)              return c0
601: (0)          def polyvalfromroots(x, r, tensor=True):
602: (4)          """
603: (4)          Evaluate a polynomial specified by its roots at points x.
604: (4)          If `r` is of length `N`, this function returns the value
605: (4)          .. math:: p(x) = \prod_{n=1}^N (x - r_n)
606: (4)          The parameter `x` is converted to an array only if it is a tuple or a
607: (4)          list, otherwise it is treated as a scalar. In either case, either `x`
608: (4)          or its elements must support multiplication and addition both with
609: (4)          themselves and with the elements of `r`.
610: (4)          If `r` is a 1-D array, then `p(x)` will have the same shape as `x`. If
`r`
611: (4)          is multidimensional, then the shape of the result depends on the value of
`tensor`. If `tensor` is ``True`` the shape will be r.shape[1:] + x.shape;
612: (4)          that is, each polynomial is evaluated at every value of `x`. If `tensor`
613: (4)          is
614: (4)          ``False``, the shape will be r.shape[1:]; that is, each polynomial is
615: (4)          evaluated only for the corresponding broadcast value of `x`. Note that
616: (4)          scalars have shape (,).

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

617: (4)          .. versionadded:: 1.12
618: (4)          Parameters
619: (4)
620: (4)          -----
621: (8)          x : array_like, compatible object
622: (8)          If `x` is a list or tuple, it is converted to an ndarray, otherwise
623: (8)          it is left unchanged and treated as a scalar. In either case, `x`
624: (8)          or its elements must support addition and multiplication with
625: (8)          with themselves and with the elements of `r`.
626: (8)          r : array_like
627: (8)          Array of roots. If `r` is multidimensional the first index is the
628: (8)          root index, while the remaining indices enumerate multiple
629: (8)          polynomials. For instance, in the two dimensional case the roots
630: (4)          of each polynomial may be thought of as stored in the columns of `r`.
631: (8)          tensor : boolean, optional
632: (8)          If True, the shape of the roots array is extended with ones on the
633: (8)          right, one for each dimension of `x`. Scalars have dimension 0 for
634: (8)          this
635: (8)          action. The result is that every column of coefficients in `r` is
636: (8)          evaluated for every element of `x`. If False, `x` is broadcast over
637: (4)          the
638: (4)          columns of `r` for the evaluation. This keyword is useful when `r` is
639: (4)          multidimensional. The default value is True.
640: (4)          Returns
641: (4)
642: (4)          -----
643: (4)          values : ndarray, compatible object
644: (8)          The shape of the returned array is described above.
645: (4)          See Also
646: (4)          -----
647: (4)          polyroots, polyfromroots, polyval
648: (4)          Examples
649: (4)
650: (4)
651: (4)
652: (11)
653: (4)
654: (4)
655: (11)
656: (4)
657: (4)
658: (4)
659: (11)
660: (4)
661: (4)
662: (4)
663: (11)
664: (4)
665: (4)
666: (4)
667: (4)
668: (4)
669: (8)
670: (4)
671: (8)
672: (4)
673: (8)
674: (12)
675: (8)
676: (12)
677: (4)
678: (0)
679: (4)
680: (4)
681: (4)
682: (4)
683: (4)

    >>> from numpy.polynomial.polynomial import polyvalfromroots
    >>> polyvalfromroots(1, [1,2,3])
    0.0
    >>> a = np.arange(4).reshape(2,2)
    >>> a
    array([[0, 1],
           [2, 3]])
    >>> polyvalfromroots(a, [-1, 0, 1])
    array([[-0.,  0.],
           [6.,  24.]])
    >>> r = np.arange(-2, 2).reshape(2,2) # multidimensional coefficients
    >>> r # each column of r defines one polynomial
    array([[[-2, -1],
           [0, 1]]])
    >>> b = [-2, 1]
    >>> polyvalfromroots(b, r, tensor=True)
    array([[-0.,  3.],
           [3.,  0.]])
    >>> polyvalfromroots(b, r, tensor=False)
    array([-0.,  0.])
    """
    r = np.array(r, ndmin=1, copy=False)
    if r.dtype.char in '?bBhHiIlLqQpP':
        r = r.astype(np.double)
    if isinstance(x, (tuple, list)):
        x = np.asarray(x)
    if isinstance(x, np.ndarray):
        if tensor:
            r = r.reshape(r.shape + (1,)*x.ndim)
        elif x.ndim >= r.ndim:
            raise ValueError("x.ndim must be < r.ndim when tensor == False")
    return np.prod(x - r, axis=0)

def polyval2d(x, y, c):
    """
    Evaluate a 2-D polynomial at points (x, y).
    This function returns the value
    .. math:: p(x,y) = \sum_{i,j} c_{i,j} * x^i * y^j
    The parameters `x` and `y` are converted to arrays only if they are

```

```

684: (4) tuples or a lists, otherwise they are treated as a scalars and they
685: (4) must have the same shape after conversion. In either case, either `x`
686: (4) and `y` or their elements must support multiplication and addition both
687: (4) with themselves and with the elements of `c`.
688: (4) If `c` has fewer than two dimensions, ones are implicitly appended to
689: (4) its shape to make it 2-D. The shape of the result will be c.shape[2:] +
690: (4) x.shape.
691: (4) Parameters
692: (4) -----
693: (4) x, y : array_like, compatible objects
694: (8) The two dimensional series is evaluated at the points `(x, y)`, 
695: (8) where `x` and `y` must have the same shape. If `x` or `y` is a list
696: (8) or tuple, it is first converted to an ndarray, otherwise it is left
697: (8) unchanged and, if it isn't an ndarray, it is treated as a scalar.
698: (4) c : array_like
699: (8) Array of coefficients ordered so that the coefficient of the term
700: (8) of multi-degree i,j is contained in `c[i,j]`. If `c` has
701: (8) dimension greater than two the remaining indices enumerate multiple
702: (8) sets of coefficients.
703: (4) Returns
704: (4) -----
705: (4) values : ndarray, compatible object
706: (8) The values of the two dimensional polynomial at points formed with
707: (8) pairs of corresponding values from `x` and `y`.
708: (4) See Also
709: (4) -----
710: (4) polyval, polygrid2d, polyval3d, polygrid3d
711: (4) Notes
712: (4) -----
713: (4) .. versionadded:: 1.7.0
714: (4) """
715: (4)     return pu._valnd(polyval, c, x, y)
716: (0) def polygrid2d(x, y, c):
717: (4) """
718: (4) Evaluate a 2-D polynomial on the Cartesian product of x and y.
719: (4) This function returns the values:
720: (4) .. math:: p(a,b) = \sum_{i,j} c_{i,j} * a^i * b^j
721: (4) where the points `(a, b)` consist of all pairs formed by taking
722: (4) `a` from `x` and `b` from `y`. The resulting points form a grid with
723: (4) `x` in the first dimension and `y` in the second.
724: (4) The parameters `x` and `y` are converted to arrays only if they are
725: (4) tuples or a lists, otherwise they are treated as a scalars. In either
726: (4) case, either `x` and `y` or their elements must support multiplication
727: (4) and addition both with themselves and with the elements of `c`.
728: (4) If `c` has fewer than two dimensions, ones are implicitly appended to
729: (4) its shape to make it 2-D. The shape of the result will be c.shape[2:] +
730: (4) x.shape + y.shape.
731: (4) Parameters
732: (4) -----
733: (4) x, y : array_like, compatible objects
734: (8) The two dimensional series is evaluated at the points in the
735: (8) Cartesian product of `x` and `y`. If `x` or `y` is a list or
736: (8) tuple, it is first converted to an ndarray, otherwise it is left
737: (8) unchanged and, if it isn't an ndarray, it is treated as a scalar.
738: (4) c : array_like
739: (8) Array of coefficients ordered so that the coefficients for terms of
740: (8) degree i,j are contained in ``c[i,j]``. If `c` has dimension
741: (8) greater than two the remaining indices enumerate multiple sets of
742: (8) coefficients.
743: (4) Returns
744: (4) -----
745: (4) values : ndarray, compatible object
746: (8) The values of the two dimensional polynomial at points in the
747: (8) product of `x` and `y`.
748: (4) See Also
749: (4) -----
750: (4) polyval, polyval2d, polyval3d, polygrid3d
751: (4) Notes

```

```

752: (4)
753: (4)
754: (4)
755: (4)
756: (0)
757: (4)
758: (4)
759: (4)
760: (4)
761: (4)
762: (4)
763: (4)
764: (4)
765: (4)
766: (4)
767: (4)
768: (4)
769: (4)
770: (4)
771: (4)
772: (8)
773: (8)
774: (8)
775: (8)
776: (8)
777: (4)
778: (8)
779: (8)
780: (8)
781: (8)
782: (4)
783: (4)
784: (4)
785: (8)
786: (8)
787: (4)
788: (4)
789: (4)
790: (4)
791: (4)
792: (4)
793: (4)
794: (4)
795: (0)
796: (4)
797: (4)
798: (4)
799: (4)
800: (4)
801: (4)
802: (4)
803: (4)
804: (4)
805: (4)
806: (4)
807: (4)
808: (4)
809: (4)
810: (4)
811: (4)
812: (4)
813: (4)
814: (4)
815: (8)
816: (8)
817: (8)
818: (8)
819: (8)
820: (4)

-----  

.. versionadded:: 1.7.0  

"""  

def polyval3d(x, y, z, c):  

    """  

        Evaluate a 3-D polynomial at points (x, y, z).  

        This function returns the values:  

        .. math:: p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * x^i * y^j * z^k  

        The parameters `x`, `y`, and `z` are converted to arrays only if  

        they are tuples or a lists, otherwise they are treated as a scalars and  

        they must have the same shape after conversion. In either case, either  

        `x`, `y`, and `z` or their elements must support multiplication and  

        addition both with themselves and with the elements of `c`.  

        If `c` has fewer than 3 dimensions, ones are implicitly appended to its  

        shape to make it 3-D. The shape of the result will be c.shape[3:] +  

        x.shape.  

    Parameters  

    -----  

        x, y, z : array_like, compatible object  

            The three dimensional series is evaluated at the points  

            `(x, y, z)`, where `x`, `y`, and `z` must have the same shape. If  

            any of `x`, `y`, or `z` is a list or tuple, it is first converted  

            to an ndarray, otherwise it is left unchanged and if it isn't an  

            ndarray it is treated as a scalar.  

        c : array_like  

            Array of coefficients ordered so that the coefficient of the term of  

            multi-degree i,j,k is contained in `c[i,j,k]`. If `c` has dimension  

            greater than 3 the remaining indices enumerate multiple sets of  

            coefficients.  

    Returns  

    -----  

        values : ndarray, compatible object  

            The values of the multidimensional polynomial on points formed with  

            triples of corresponding values from `x`, `y`, and `z`.  

    See Also  

    -----  

        polyval, polyval2d, polygrid2d, polygrid3d  

    Notes  

    -----  

.. versionadded:: 1.7.0  

"""  

def polygrid3d(x, y, z, c):  

    """  

        Evaluate a 3-D polynomial on the Cartesian product of x, y and z.  

        This function returns the values:  

        .. math:: p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * a^i * b^j * c^k  

        where the points `(a, b, c)` consist of all triples formed by taking  

        `a` from `x`, `b` from `y`, and `c` from `z`. The resulting points form  

        a grid with `x` in the first dimension, `y` in the second, and `z` in  

        the third.  

        The parameters `x`, `y`, and `z` are converted to arrays only if they  

        are tuples or a lists, otherwise they are treated as a scalars. In  

        either case, either `x`, `y`, and `z` or their elements must support  

        multiplication and addition both with themselves and with the elements  

        of `c`.  

        If `c` has fewer than three dimensions, ones are implicitly appended to  

        its shape to make it 3-D. The shape of the result will be c.shape[3:] +  

        x.shape + y.shape + z.shape.  

    Parameters  

    -----  

        x, y, z : array_like, compatible objects  

            The three dimensional series is evaluated at the points in the  

            Cartesian product of `x`, `y`, and `z`. If `x`, `y`, or `z` is a  

            list or tuple, it is first converted to an ndarray, otherwise it is  

            left unchanged and, if it isn't an ndarray, it is treated as a  

            scalar.  

        c : array_like

```

```

821: (8)           Array of coefficients ordered so that the coefficients for terms of
822: (8)           degree i,j are contained in ``c[i,j]``. If `c` has dimension
823: (8)           greater than two the remaining indices enumerate multiple sets of
824: (8)           coefficients.
825: (4)           Returns
826: (4)
827: (4)           values : ndarray, compatible object
828: (8)           The values of the two dimensional polynomial at points in the
829: (8)           product of `x` and `y`.
830: (4)           See Also
831: (4)
832: (4)           polyval, polyval2d, polygrid2d, polyval3d
833: (4)           Notes
834: (4)
835: (4)           .. versionadded:: 1.7.0
836: (4)
837: (4)           return pu._gridnd(polyval, c, x, y, z)
838: (0)           def polyvander(x, deg):
839: (4)             """Vandermonde matrix of given degree.
840: (4)             Returns the Vandermonde matrix of degree `deg` and sample points
841: (4)             `x`. The Vandermonde matrix is defined by
842: (4)             .. math:: V[..., i] = x^i,
843: (4)             where `0 <= i <= deg`. The leading indices of `V` index the elements of
844: (4)             `x` and the last index is the power of `x`.
845: (4)             If `c` is a 1-D array of coefficients of length `n + 1` and `V` is the
846: (4)             matrix ``V = polyvander(x, n)``, then ``np.dot(V, c)`` and
847: (4)             ```polyval(x, c)` are the same up to roundoff. This equivalence is
848: (4)             useful both for least squares fitting and for the evaluation of a large
849: (4)             number of polynomials of the same degree and sample points.
850: (4)             Parameters
851: (4)
852: (4)             x : array_like
853: (8)               Array of points. The dtype is converted to float64 or complex128
854: (8)               depending on whether any of the elements are complex. If `x` is
855: (8)               scalar it is converted to a 1-D array.
856: (4)             deg : int
857: (8)               Degree of the resulting matrix.
858: (4)             Returns
859: (4)
860: (4)             vander : ndarray.
861: (8)               The Vandermonde matrix. The shape of the returned matrix is
862: (8)               ``x.shape + (deg + 1,)``, where the last index is the power of `x`.
863: (8)               The dtype will be the same as the converted `x`.
864: (4)             See Also
865: (4)
866: (4)             polyvander2d, polyvander3d
867: (4)
868: (4)             """
869: (4)             ideg = pu._deprecate_as_int(deg, "deg")
870: (8)             if ideg < 0:
871: (4)                 raise ValueError("deg must be non-negative")
872: (4)             x = np.array(x, copy=False, ndmin=1) + 0.0
873: (4)             dims = (ideg + 1,) + x.shape
874: (4)             dtyp = x.dtype
875: (4)             v = np.empty(dims, dtype=dtyp)
876: (4)             v[0] = x**0 + 1
877: (4)             if ideg > 0:
878: (8)                 v[1] = x
879: (8)                 for i in range(2, ideg + 1):
880: (12)                     v[i] = v[i-1]*x
881: (4)             return np.moveaxis(v, 0, -1)
882: (0)           def polyvander2d(x, y, deg):
883: (4)             """Pseudo-Vandermonde matrix of given degrees.
884: (4)             Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
885: (4)             points `(x, y)`. The pseudo-Vandermonde matrix is defined by
886: (4)             .. math:: V[..., (deg[1] + 1)*i + j] = x^i * y^j,
887: (4)             where `0 <= i <= deg[0]` and `0 <= j <= deg[1]`. The leading indices of
888: (4)             `V` index the points `(x, y)` and the last index encodes the powers of
889: (4)             `x` and `y`.

```

```

889: (4)     If ``V = polyvander2d(x, y, [xdeg, ydeg])``, then the columns of `V`
890: (4)     correspond to the elements of a 2-D coefficient array `c` of shape
891: (4)     (xdeg + 1, ydeg + 1) in the order
892: (4)     .. math:: c_{\{0\}}, c_{\{01\}}, c_{\{02\}} \dots , c_{\{10\}}, c_{\{11\}}, c_{\{12\}} \dots
893: (4)     and ``np.dot(V, c.flat)`` and ``polyval2d(x, y, c)`` will be the same
894: (4)     up to roundoff. This equivalence is useful both for least squares
895: (4)     fitting and for the evaluation of a large number of 2-D polynomials
896: (4)     of the same degrees and sample points.
897: (4)     Parameters
898: (4)     -----
899: (4)     x, y : array_like
900: (8)         Arrays of point coordinates, all of the same shape. The dtypes
901: (8)         will be converted to either float64 or complex128 depending on
902: (8)         whether any of the elements are complex. Scalars are converted to
903: (8)         1-D arrays.
904: (4)     deg : list of ints
905: (8)         List of maximum degrees of the form [x_deg, y_deg].
906: (4)     Returns
907: (4)     -----
908: (4)     vander2d : ndarray
909: (8)         The shape of the returned matrix is ``x.shape + (order,)``, where
910: (8)         :math:`order = (deg[0]+1)*(deg[1]+1)` . The dtype will be the same
911: (8)         as the converted `x` and `y` .
912: (4)     See Also
913: (4)     -----
914: (4)     polyvander, polyvander3d, polyval2d, polyval3d
915: (4)     """
916: (4)     return pu._vander_nd_flat((polyvander, polyvander), (x, y), deg)
917: (0)   def polyvander3d(x, y, z, deg):
918: (4)       """Pseudo-Vandermonde matrix of given degrees.
919: (4)       Returns the pseudo-Vandermonde matrix of degrees `deg` and sample
920: (4)       points `(x, y, z)` . If `l, m, n` are the given degrees in `x, y, z` ,
921: (4)       then The pseudo-Vandermonde matrix is defined by
922: (4)       .. math:: V[..., (m+1)(n+1)i + (n+1)j + k] = x^i * y^j * z^k,
923: (4)       where `0 <= i <= l`, `0 <= j <= m` , and `0 <= j <= n` . The leading
924: (4)       indices of `V` index the points `(x, y, z)` and the last index encodes
925: (4)       the powers of `x` , `y` , and `z` .
926: (4)       If ``V = polyvander3d(x, y, z, [xdeg, ydeg, zdeg])``, then the columns
927: (4)       of `V` correspond to the elements of a 3-D coefficient array `c` of
928: (4)       shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order
929: (4)       .. math:: c_{\{000\}}, c_{\{001\}}, c_{\{002\}}, \dots , c_{\{010\}}, c_{\{011\}}, c_{\{012\}}, \dots
930: (4)       and ``np.dot(V, c.flat)`` and ``polyval3d(x, y, z, c)`` will be the
931: (4)       same up to roundoff. This equivalence is useful both for least squares
932: (4)       fitting and for the evaluation of a large number of 3-D polynomials
933: (4)       of the same degrees and sample points.
934: (4)       Parameters
935: (4)       -----
936: (4)       x, y, z : array_like
937: (8)           Arrays of point coordinates, all of the same shape. The dtypes will
938: (8)           be converted to either float64 or complex128 depending on whether
939: (8)           any of the elements are complex. Scalars are converted to 1-D
940: (8)           arrays.
941: (4)       deg : list of ints
942: (8)           List of maximum degrees of the form [x_deg, y_deg, z_deg].
943: (4)       Returns
944: (4)       -----
945: (4)       vander3d : ndarray
946: (8)           The shape of the returned matrix is ``x.shape + (order,)``, where
947: (8)           :math:`order = (deg[0]+1)*(deg[1]+1)*(deg[2]+1)` . The dtype will
948: (8)           be the same as the converted `x` , `y` , and `z` .
949: (4)       See Also
950: (4)       -----
951: (4)       polyvander, polyvander3d, polyval2d, polyval3d
952: (4)       Notes
953: (4)       -----
954: (4)       .. versionadded:: 1.7.0
955: (4)       """
956: (4)       return pu._vander_nd_flat((polyvander, polyvander, polyvander), (x, y, z),
957: (4)       deg)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

957: (0)
958: (4)
959: (4)
960: (4)
961: (4)
962: (4)
963: (4)
964: (4)
965: (4)
966: (4)
967: (4)
968: (4)
969: (4)
970: (4)
971: (8)
972: (4)
973: (8)
974: (8)
975: (8)
976: (8)
977: (4)
978: (8)
979: (8)
980: (8)
981: (8)
982: (4)
983: (8)
984: (8)
985: (8)
986: (8)
987: (8)
988: (4)
989: (8)
990: (8)
991: (8)
992: (8)
993: (4)
994: (8)
995: (8)
996: (8)
997: (8)
998: (8)
999: (8)
1000: (4)
1001: (4)
1002: (4)
1003: (8)
1004: (8)
1005: (8)
1006: (4)
1007: (8)
1008: (8)
1009: (8)
1010: (8)
1011: (8)
1012: (8)
1013: (4)
1014: (4)
1015: (4)
1016: (8)
1017: (8)
1018: (8)
1019: (8)
1020: (8)
1021: (4)
1022: (4)
1023: (4)
1024: (4)
1025: (4)

def polyfit(x, y, deg, rcond=None, full=False, w=None):
    """
    Least-squares fit of a polynomial to data.
    Return the coefficients of a polynomial of degree `deg` that is the
    least squares fit to the data values `y` given at points `x`. If `y` is
    1-D the returned coefficients will also be 1-D. If `y` is 2-D multiple
    fits are done, one for each column of `y`, and the resulting
    coefficients are stored in the corresponding columns of a 2-D return.
    The fitted polynomial(s) are in the form
    .. math:: p(x) = c_0 + c_1 * x + \dots + c_n * x^n,
    where `n` is `deg`.
    Parameters
    -----
    x : array_like, shape (`M`,)
        x-coordinates of the `M` sample (data) points ``x[i], y[i]``.
    y : array_like, shape (`M`,) or (`M`, `K`)
        y-coordinates of the sample points. Several sets of sample points
        sharing the same x-coordinates can be (independently) fit with one
        call to `polyfit` by passing in for `y` a 2-D array that contains
        one data set per column.
    deg : int or 1-D array_like
        Degree(s) of the fitting polynomials. If `deg` is a single integer
        all terms up to and including the `deg`'th term are included in the
        fit. For NumPy versions >= 1.11.0 a list of integers specifying the
        degrees of the terms to include may be used instead.
    rcond : float, optional
        Relative condition number of the fit. Singular values smaller
        than `rcond`, relative to the largest singular value, will be
        ignored. The default value is ``len(x)*eps``, where `eps` is the
        relative precision of the platform's float type, about 2e-16 in
        most cases.
    full : bool, optional
        Switch determining the nature of the return value. When ``False``
        (the default) just the coefficients are returned; when ``True``,
        diagnostic information from the singular value decomposition (used
        to solve the fit's matrix equation) is also returned.
    w : array_like, shape (`M`,), optional
        Weights. If not None, the weight ``w[i]`` applies to the unsquared
        residual ``y[i] - y_hat[i]`` at ``x[i]``. Ideally the weights are
        chosen so that the errors of the products ``w[i]*y[i]`` all have the
        same variance. When using inverse-variance weighting, use
        ``w[i] = 1/sigma(y[i])``. The default value is None.
        .. versionadded:: 1.5.0
    Returns
    -----
    coef : ndarray, shape (`deg` + 1,) or (`deg` + 1, `K`)
        Polynomial coefficients ordered from low to high. If `y` was 2-D,
        the coefficients in column `k` of `coef` represent the polynomial
        fit to the data in `y`'s `k`-th column.
    [residuals, rank, singular_values, rcond] : list
        These values are only returned if ``full == True``
        - residuals -- sum of squared residuals of the least squares fit
        - rank -- the numerical rank of the scaled Vandermonde matrix
        - singular_values -- singular values of the scaled Vandermonde matrix
        - rcond -- value of `rcond`.
        For more details, see `numpy.linalg.lstsq`.
    Raises
    -----
    RankWarning
        Raised if the matrix in the least-squares fit is rank deficient.
        The warning is only raised if ``full == False``. The warnings can
        be turned off by:
        >>> import warnings
        >>> warnings.simplefilter('ignore', np.RankWarning)
    See Also
    -----
    numpy.polynomial.chebyshev.chebfit
    numpy.polynomial.legendre.legfit
    numpy.polynomial.laguerre.lagfit

```

```

1026: (4) numpy.polynomial.hermite.hermfit
1027: (4) numpy.polynomial.hermite_e.hermefit
1028: (4) polyval : Evaluates a polynomial.
1029: (4) polyvander : Vandermonde matrix for powers.
1030: (4) numpy.linalg.lstsq : Computes a least-squares fit from the matrix.
1031: (4) scipy.interpolate.UnivariateSpline : Computes spline fits.
1032: (4) Notes
1033: (4)
1034: (4) -----
1035: (4) The solution is the coefficients of the polynomial `p` that minimizes
1036: (4) the sum of the weighted squared errors
1037: (4) .. math:: E = \sum_j w_j^2 * |y_j - p(x_j)|^2,
1038: (4) where the :math:`w_j` are the weights. This problem is solved by
1039: (4) setting up the (typically) over-determined matrix equation:
1040: (4) .. math:: V(x) * c = w * y,
1041: (4) where `V` is the weighted pseudo Vandermonde matrix of `x`, `c` are the
1042: (4) coefficients to be solved for, `w` are the weights, and `y` are the
1043: (4) observed values. This equation is then solved using the singular value
1044: (4) decomposition of `V`.
1045: (4) If some of the singular values of `V` are so small that they are
1046: (4) neglected (and `full` == ``False``), a `RankWarning` will be raised.
1047: (4) This means that the coefficient values may be poorly determined.
1048: (4) Fitting to a lower order polynomial will usually get rid of the warning
1049: (4) (but may not be what you want, of course; if you have independent
1050: (4) reason(s) for choosing the degree which isn't working, you may have to:
1051: (4) a) reconsider those reasons, and/or b) reconsider the quality of your
1052: (4) data). The `rcond` parameter can also be set to a value smaller than
1053: (4) its default, but the resulting fit may be spurious and have large
1054: (4) contributions from roundoff error.
1055: (4) Polynomial fits using double precision tend to "fail" at about
1056: (4) (polynomial) degree 20. Fits using Chebyshev or Legendre series are
1057: (4) generally better conditioned, but much can still depend on the
1058: (4) distribution of the sample points and the smoothness of the data. If
1059: (4) the quality of the fit is inadequate, splines may be a good
1060: (4) alternative.
1061: (4) Examples
1062: (4)
1063: (4) -----
1064: (4) >>> np.random.seed(123)
1065: (4) >>> from numpy.polynomial import polynomial as P
1066: (4) >>> x = np.linspace(-1,1,51) # x "data": [-1, -0.96, ..., 0.96, 1]
1067: (4) >>> y = x**3 - x + np.random.randn(len(x)) # x^3 - x + Gaussian noise
1068: (4) >>> c, stats = P.polyfit(x,y,3,full=True)
1069: (4) >>> c # c[0], c[2] should be approx. 0, c[1] approx. -1, c[3] approx. 1
1070: (4) array([ 0.01909725, -1.30598256, -0.00577963,  1.02644286]) # may vary
1071: (5) >>> stats # note the large SSR, explaining the rather poor results
# may vary
1072: (14) [array([ 38.06116253]), 4, array([ 1.38446749,  1.32119158,  0.50443316,
1073: (4) 0.28853036]), 1.1324274851176597e-014]
1074: (4) Same thing without the added noise
1075: (4) >>> y = x**3 - x
1076: (4) >>> c, stats = P.polyfit(x,y,3,full=True)
1077: (4) >>> c # c[0], c[2] should be "very close to 0", c[1] ~= -1, c[3] ~= 1
1.00000000e+00]
1078: (4) array([-6.36925336e-18, -1.00000000e+00, -4.08053781e-16,
1079: (4) 1.00000000e+00])
1080: (15) >>> stats # note the minuscule SSR
1081: (4) [array([ 7.46346754e-31]), 4, array([ 1.38446749,  1.32119158, # may vary
1082: (4) 0.50443316,  0.28853036]), 1.1324274851176597e-014]
1083: (0) """
1084: (4) def polycompanion(c):
1085: (4) """
1086: (4) Return the companion matrix of c.
1087: (4) The companion matrix for power series cannot be made symmetric by
1088: (4) scaling the basis, so this function differs from those for the
1089: (4) orthogonal polynomials.
1090: (4) Parameters
1091: (4) -----
1092: (8) c : array_like
1-D array of polynomial coefficients ordered from low to high

```

```

1093: (8)             degree.
1094: (4)             Returns
1095: (4)
1096: (4)
1097: (8)             mat : ndarray
1098: (4)             Companion matrix of dimensions (deg, deg).
1099: (4)             Notes
1100: (4)
1101: (4)
1102: (4)             [c] = pu.as_series([c])
1103: (4)             if len(c) < 2:
1104: (8)                 raise ValueError('Series must have maximum degree of at least 1.')
1105: (4)             if len(c) == 2:
1106: (8)                 return np.array([-c[0]/c[1]])
1107: (4)             n = len(c) - 1
1108: (4)             mat = np.zeros((n, n), dtype=c.dtype)
1109: (4)             bot = mat.reshape(-1)[n::n+1]
1110: (4)             bot[...] = 1
1111: (4)             mat[:, -1] -= c[:-1]/c[-1]
1112: (4)             return mat
1113: (0)             def polyroots(c):
1114: (4)                 """
1115: (4)                 Compute the roots of a polynomial.
1116: (4)                 Return the roots (a.k.a. "zeros") of the polynomial
1117: (4)                 .. math:: p(x) = \sum_i c[i] * x^i.
1118: (4)                 Parameters
1119: (4)
1120: (4)                 c : 1-D array_like
1121: (8)                     1-D array of polynomial coefficients.
1122: (4)                 Returns
1123: (4)
1124: (4)                 out : ndarray
1125: (8)                     Array of the roots of the polynomial. If all the roots are real,
1126: (8)                     then `out` is also real, otherwise it is complex.
1127: (4)                 See Also
1128: (4)
1129: (4)                     numpy.polynomial.chebyshev.chebroots
1130: (4)                     numpy.polynomial.legendre.legroots
1131: (4)                     numpy.polynomial.laguerre.lagroots
1132: (4)                     numpy.polynomial.hermite.hermroots
1133: (4)                     numpy.polynomial.hermite_e.hermroots
1134: (4)                 Notes
1135: (4)
1136: (4)                     The root estimates are obtained as the eigenvalues of the companion
1137: (4)                     matrix, Roots far from the origin of the complex plane may have large
1138: (4)                     errors due to the numerical instability of the power series for such
1139: (4)                     values. Roots with multiplicity greater than 1 will also show larger
1140: (4)                     errors as the value of the series near such points is relatively
1141: (4)                     insensitive to errors in the roots. Isolated roots near the origin can
1142: (4)                     be improved by a few iterations of Newton's method.
1143: (4)                 Examples
1144: (4)
1145: (4)                     >>> import numpy.polynomial.polynomial as poly
1146: (4)                     >>> poly.polyroots(poly.polyfromroots((-1,0,1)))
1147: (4)                     array([-1.,  0.,  1.])
1148: (4)                     >>> poly.polyroots(poly.polyfromroots((-1,0,1))).dtype
1149: (4)                     dtype('float64')
1150: (4)                     >>> j = complex(0,1)
1151: (4)                     >>> poly.polyroots(poly.polyfromroots((-j,0,j)))
1152: (4)                     array([ 0.0000000e+00+0.j,  0.0000000e+00+1.j,  2.77555756e-17-1.j])
# may vary
1153: (4)
1154: (4)             """
1155: (4)             [c] = pu.as_series([c])
1156: (8)             if len(c) < 2:
1157: (4)                 return np.array([], dtype=c.dtype)
1158: (8)             if len(c) == 2:
1159: (4)                 return np.array([-c[0]/c[1]])
m = polycompanion(c)[:-1,:-1]
r = la.eigvals(m)

```

```

1161: (4)             r.sort()
1162: (4)             return r
1163: (0) class Polynomial(ABCPolyBase):
1164: (4)             """A power series class.
1165: (4)             The Polynomial class provides the standard Python numerical methods
1166: (4)             '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the
1167: (4)             attributes and methods listed in the `ABCPolyBase` documentation.
1168: (4)             Parameters
1169: (4)             -----
1170: (4)             coef : array_like
1171: (8)                 Polynomial coefficients in order of increasing degree, i.e.,
1172: (8)                 ``(1, 2, 3)`` give `` $1 + 2x + 3x^2$ ``.
1173: (4)             domain : (2,) array_like, optional
1174: (8)                 Domain to use. The interval ``[domain[0], domain[1]]`` is mapped
1175: (8)                 to the interval ``[window[0], window[1]]`` by shifting and scaling.
1176: (8)                 The default value is [-1, 1].
1177: (4)             window : (2,) array_like, optional
1178: (8)                 Window, see `domain` for its use. The default value is [-1, 1].
1179: (8)             .. versionadded:: 1.6.0
1180: (4)             symbol : str, optional
1181: (8)                 Symbol used to represent the independent variable in string
1182: (8)                 representations of the polynomial expression, e.g. for printing.
1183: (8)                 The symbol must be a valid Python identifier. Default value is 'x'.
1184: (8)             .. versionadded:: 1.24
1185: (4)             """
1186: (4)             _add = staticmethod(polyadd)
1187: (4)             _sub = staticmethod(polysub)
1188: (4)             _mul = staticmethod(polymul)
1189: (4)             _div = staticmethod(polydiv)
1190: (4)             _pow = staticmethod(polypow)
1191: (4)             _val = staticmethod(polyval)
1192: (4)             _int = staticmethod(polyint)
1193: (4)             _der = staticmethod(polyder)
1194: (4)             _fit = staticmethod(polyfit)
1195: (4)             _line = staticmethod(polyline)
1196: (4)             _roots = staticmethod(polyroots)
1197: (4)             _fromroots = staticmethod(polyfromroots)
1198: (4)             domain = np.array(polydomain)
1199: (4)             window = np.array(polydomain)
1200: (4)             basis_name = None
1201: (4)             @classmethod
1202: (4)             def __str__(cls, i, arg_str):
1203: (8)                 if i == '1':
1204: (12)                     return f"1 · {arg_str}"
1205: (8)                 else:
1206: (12)                     return f"1 · {arg_str} · {i.translate(cls._superscript_mapping)}"
1207: (4)             @staticmethod
1208: (4)             def __str__(i, arg_str):
1209: (8)                 if i == '1':
1210: (12)                     return f"1 · {arg_str}"
1211: (8)                 else:
1212: (12)                     return f"1 · {arg_str}^{i}"
1213: (4)             @staticmethod
1214: (4)             def __repr__(i, arg_str, needs_parens):
1215: (8)                 if needs_parens:
1216: (12)                     arg_str = rf"\left({arg_str}\right)"
1217: (8)                 if i == 0:
1218: (12)                     return '1'
1219: (8)                 elif i == 1:
1220: (12)                     return arg_str
1221: (8)                 else:
1222: (12)                     return f"1 · {arg_str}^{i}"
-----
```

File 293 - polyutils.py:

```

1: (0)             """
2: (0)             Utility classes and functions for the polynomial modules.
```

```

3: (0) This module provides: error and warning objects; a polynomial base class;
4: (0) and some routines used in both the `polynomial` and `chebyshev` modules.
5: (0) Warning objects
6: (0) -----
7: (0) .. autosummary::
8: (3) :toctree: generated/
9: (3) RankWarning raised in least-squares fit for rank-deficient matrix.
10: (0) Functions
11: (0) -----
12: (0) .. autosummary::
13: (3) :toctree: generated/
14: (3) as_series convert list of array_likes into 1-D arrays of common type.
15: (3) trimseq remove trailing zeros.
16: (3) trimcoef remove small trailing coefficients.
17: (3) getdomain return the domain appropriate for a given set of abscissae.
18: (3) mapdomain maps points between domains.
19: (3) mapparms parameters of the linear map between domains.
20: (0)
21: (0) import operator
22: (0) import functools
23: (0) import warnings
24: (0) import numpy as np
25: (0) from numpy.core.multiarray import dragon4_positional, dragon4_scientific
26: (0) from numpy.core.umath import absolute
27: (0) __all__ = [
28: (4)     'RankWarning', 'as_series', 'trimseq',
29: (4)     'trimcoef', 'getdomain', 'mapdomain', 'mapparms',
30: (4)     'format_float']
31: (0) class RankWarning(UserWarning):
32: (4)     """Issued by chebfit when the design matrix is rank deficient."""
33: (4)     pass
34: (0) def trimseq(seq):
35: (4)     """Remove small Poly series coefficients.
36: (4)     Parameters
37: (4)     -----
38: (4)     seq : sequence
39: (8)     Sequence of Poly series coefficients. This routine fails for
40: (8)     empty sequences.
41: (4)     Returns
42: (4)     -----
43: (4)     series : sequence
44: (8)     Subsequence with trailing zeros removed. If the resulting sequence
45: (8)     would be empty, return the first element. The returned sequence may
46: (8)     or may not be a view.
47: (4)     Notes
48: (4)     -----
49: (4)     Do not lose the type info if the sequence contains unknown objects.
50: (4)
51: (4)     if len(seq) == 0:
52: (8)         return seq
53: (4)     else:
54: (8)         for i in range(len(seq) - 1, -1, -1):
55: (12)             if seq[i] != 0:
56: (16)                 break
57: (8)             return seq[:i+1]
58: (0) def as_series(alist, trim=True):
59: (4)
60: (4)     """Return argument as a list of 1-d arrays.
61: (4)     The returned list contains array(s) of dtype double, complex double, or
62: (4)     object. A 1-d argument of shape ``(N,)`` is parsed into ``N`` arrays of
63: (4)     size one; a 2-d argument of shape ``(M,N)`` is parsed into ``M`` arrays
64: (4)     of size ``N`` (i.e., is "parsed by row"); and a higher dimensional array
65: (4)     raises a Value Error if it is not first reshaped into either a 1-d or 2-d
66: (4)     array.
67: (4)     Parameters
68: (4)     -----
69: (4)     alist : array_like
70: (8)         A 1- or 2-d array_like
71: (4)     trim : boolean, optional

```

```

72: (8)             When True, trailing zeros are removed from the inputs.
73: (8)             When False, the inputs are passed through intact.
74: (4)             Returns
75: (4)
76: (4)             [a1, a2,...] : list of 1-D arrays
77: (8)             A copy of the input data as a list of 1-d arrays.
78: (4)             Raises
79: (4)
80: (4)             ValueError
81: (8)             Raised when `as_series` cannot convert its input to 1-d arrays, or at
82: (8)             least one of the resulting arrays is empty.
83: (4)             Examples
84: (4)
85: (4)             >>> from numpy.polynomial import polyutils as pu
86: (4)             >>> a = np.arange(4)
87: (4)             >>> pu.as_series(a)
88: (4)             [array([0.]), array([1.]), array([2.]), array([3.])]
89: (4)             >>> b = np.arange(6).reshape((2,3))
90: (4)             >>> pu.as_series(b)
91: (4)             [array([0., 1., 2.]), array([3., 4., 5.])]
92: (4)             >>> pu.as_series((1, np.arange(3), np.arange(2, dtype=np.float16)))
93: (4)             [array([1.]), array([0., 1., 2.]), array([0., 1.])]
94: (4)             >>> pu.as_series([2, [1.1, 0.]])
95: (4)             [array([2.]), array([1.1])]
96: (4)             >>> pu.as_series([2, [1.1, 0.]], trim=False)
97: (4)             [array([2.]), array([1.1, 0. ])]
98: (4)
99: (4)             """
100: (4)             arrays = [np.array(a, ndmin=1, copy=False) for a in alist]
101: (8)             if min([a.size for a in arrays]) == 0:
102: (4)                 raise ValueError("Coefficient array is empty")
103: (8)             if any(a.ndim != 1 for a in arrays):
104: (4)                 raise ValueError("Coefficient array is not 1-d")
105: (8)             if trim:
106: (4)                 arrays = [trimseq(a) for a in arrays]
107: (8)             if any(a.dtype == np.dtype(object) for a in arrays):
108: (4)                 ret = []
109: (12)                for a in arrays:
110: (16)                    if a.dtype != np.dtype(object):
111: (16)                        tmp = np.empty(len(a), dtype=np.dtype(object))
112: (16)                        tmp[:] = a[:]
113: (12)                        ret.append(tmp)
114: (16)                    else:
115: (4)                        ret.append(a.copy())
116: (8)
117: (12)                else:
118: (8)                    try:
119: (12)                        dtype = np.common_type(*arrays)
120: (8)                    except Exception as e:
121: (4)                        raise ValueError("Coefficient arrays have no common type") from e
122: (0)                    ret = [np.array(a, copy=True, dtype=dtype) for a in arrays]
123: (4)                return ret
124: (4)            def trimcoef(c, tol=0):
125: (4)                """
126: (4)                Remove "small" "trailing" coefficients from a polynomial.
127: (4)                "Small" means "small in absolute value" and is controlled by the
128: (4)                parameter `tol`; "trailing" means highest order coefficient(s), e.g., in
129: (4)                ``[0, 1, 1, 0, 0]`` (which represents `` $0 + x + x^2 + 0*x^3 + 0*x^4$ ``)
130: (4)                both the 3-rd and 4-th order coefficients would be "trimmed."
131: (4)                Parameters
132: (4)                -----
133: (4)                c : array_like
134: (8)                1-d array of coefficients, ordered from lowest order to highest.
135: (4)                tol : number, optional
136: (8)                Trailing (i.e., highest order) elements with absolute value less
137: (4)                than or equal to `tol` (default value is zero) are removed.
138: (4)                Returns
139: (4)                -----
140: (4)                trimmed : ndarray
141: (8)                1-d array with trailing zeros removed. If the resulting series
142: (4)                would be empty, a series containing a single zero is returned.

```

```

141: (4)             Raises
142: (4)             -----
143: (4)             ValueError
144: (8)             If `tol` < 0
145: (4)             See Also
146: (4)             -----
147: (4)             trimseq
148: (4)             Examples
149: (4)             -----
150: (4)             >>> from numpy.polynomial import polyutils as pu
151: (4)             >>> pu.trimcoef((0,0,3,0,5,0,0))
152: (4)             array([0., 0., 3., 0., 5.])
153: (4)             >>> pu.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
154: (4)             array([0.])
155: (4)             >>> i = complex(0,1) # works for complex
156: (4)             >>> pu.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
157: (4)             array([0.0003+0.j, 0.001 -0.001j])
158: (4)             """
159: (4)             if tol < 0:
160: (8)                 raise ValueError("tol must be non-negative")
161: (4)             [c] = as_series([c])
162: (4)             [ind] = np.nonzero(np.abs(c) > tol)
163: (4)             if len(ind) == 0:
164: (8)                 return c[:1]*0
165: (4)             else:
166: (8)                 return c[:ind[-1] + 1].copy()
167: (0)             def getdomain(x):
168: (4)                 """
169: (4)                 Return a domain suitable for given abscissae.
170: (4)                 Find a domain suitable for a polynomial or Chebyshev series
171: (4)                 defined at the values supplied.
172: (4)                 Parameters
173: (4)                 -----
174: (4)                 x : array_like
175: (8)                 1-d array of abscissae whose domain will be determined.
176: (4)                 Returns
177: (4)                 -----
178: (4)                 domain : ndarray
179: (8)                 1-d array containing two values. If the inputs are complex, then
180: (8)                 the two returned points are the lower left and upper right corners
181: (8)                 of the smallest rectangle (aligned with the axes) in the complex
182: (8)                 plane containing the points `x`. If the inputs are real, then the
183: (8)                 two points are the ends of the smallest interval containing the
184: (8)                 points `x`.
185: (4)                 See Also
186: (4)                 -----
187: (4)                 mapparms, mapdomain
188: (4)                 Examples
189: (4)                 -----
190: (4)                 >>> from numpy.polynomial import polyutils as pu
191: (4)                 >>> points = np.arange(4)**2 - 5; points
192: (4)                 array([-5, -4, -1, 4])
193: (4)                 >>> pu.getdomain(points)
194: (4)                 array([-5., 4.])
195: (4)                 >>> c = np.exp(complex(0,1)*np.pi*np.arange(12)/6) # unit circle
196: (4)                 >>> pu.getdomain(c)
197: (4)                 array([-1.-1.j, 1.+1.j])
198: (4)                 """
199: (4)                 [x] = as_series([x], trim=False)
200: (4)                 if x.dtype.char in np.typecodes['Complex']:
201: (8)                     rmin, rmax = x.real.min(), x.real.max()
202: (8)                     imin, imax = x.imag.min(), x.imag.max()
203: (8)                     return np.array((complex(rmin, imin), complex(rmax, imax)))
204: (4)                 else:
205: (8)                     return np.array((x.min(), x.max()))
206: (0)             def mapparms(old, new):
207: (4)                 """
208: (4)                 Linear map parameters between domains.
209: (4)                 Return the parameters of the linear map ``offset + scale*x`` that maps

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

210: (4) `old` to `new` such that ``old[i] -> new[i]``, ``i = 0, 1``.
211: (4)
212: (4)
213: (4)
214: (8) Parameters
215: (8)
216: (4)
217: (4)
218: (4)
219: (8) -----
220: (8) old, new : array_like
221: (4) Domains. Each domain must (successfully) convert to a 1-d array
222: (4) containing precisely two values.
223: (4)
224: (4) Returns
225: (4)
226: (4)
227: (4) -----
228: (4) offset, scale : scalars
229: (4) The map ``L(x) = offset + scale*x`` maps the first domain to the
230: (4) second.
231: (4) See Also
232: (4)
233: (4) -----
234: (4) getdomain, mapdomain
235: (4) Notes
236: (4)
237: (4) -----
238: (4) Also works for complex numbers, and thus can be used to calculate the
239: (4) parameters required to map any line in the complex plane to any other
240: (4) line therein.
241: (4) Examples
242: (4)
243: (4)
244: (4)
245: (0) -----
246: (4) >>> from numpy.polynomial import polyutils as pu
247: (4) >>> pu.mapparms((-1,1),(-1,1))
248: (4) (0.0, 1.0)
249: (4) >>> pu.mapparms((1,-1),(-1,1))
250: (4) (-0.0, -1.0)
251: (4) >>> i = complex(0,1)
252: (4) >>> pu.mapparms((-i,-1),(1,i))
253: (4) ((1+1j), (1-0j))
254: (4) """
255: (4) oldlen = old[1] - old[0]
256: (4) newlen = new[1] - new[0]
257: (4) off = (old[1]*new[0] - old[0]*new[1])/oldlen
258: (4) scl = newlen/oldlen
259: (4) return off, scl
260: (4)
261: (4) def mapdomain(x, old, new):
262: (4) """
263: (4) Apply linear map to input points.
264: (4) The linear map ``offset + scale*x`` that maps the domain `old` to
265: (4) the domain `new` is applied to the points `x`.
266: (4) Parameters
267: (4)
268: (4) x : array_like
269: (4) Points to be mapped. If `x` is a subtype of ndarray the subtype
270: (4) will be preserved.
271: (4) old, new : array_like
272: (4) The two domains that determine the map. Each must (successfully)
273: (4) convert to 1-d arrays containing precisely two values.
274: (4) Returns
275: (4)
276: (4) x_out : ndarray
277: (4) Array of points of the same shape as `x`, after application of the
278: (4) linear map between the two domains.

```

See Also

```

265: (4) -----
266: (4) getdomain, mapparms
267: (4) Notes
268: (4) -----
269: (4) Effectively, this implements:
270: (4) .. math::
271: (4)     x\_out = new[0] + m(x - old[0])
272: (4) where
273: (4) .. math::
274: (4)     m = \frac{new[1]-new[0]}{old[1]-old[0]}

```

Examples

```

275: (4)
276: (4) >>> from numpy.polynomial import polyutils as pu
277: (4) >>> old_domain = (-1,1)
278: (4) >>> new_domain = (0,2*np.pi)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

279: (4)          >>> x = np.linspace(-1,1,6); x
280: (4)          array([-1. , -0.6, -0.2,  0.2,  0.6,  1. ])
281: (4)          >>> x_out = pu.mapdomain(x, old_domain, new_domain); x_out
282: (4)          array([ 0.           ,  1.25663706,  2.51327412,  3.76991118,  5.02654825, #
may vary
283: (12)          6.28318531])
284: (4)          >>> x - pu.mapdomain(x_out, new_domain, old_domain)
285: (4)          array([0., 0., 0., 0., 0., 0.])
286: (4)          Also works for complex numbers (and thus can be used to map any line in
287: (4)          the complex plane to any other line therein).
288: (4)          >>> i = complex(0,1)
289: (4)          >>> old = (-1 - i, 1 + i)
290: (4)          >>> new = (-1 + i, 1 - i)
291: (4)          >>> z = np.linspace(old[0], old[1], 6); z
292: (4)          array([-1. -1.j , -0.6-0.6j, -0.2-0.2j,  0.2+0.2j,  0.6+0.6j,  1. +1.j ])
293: (4)          >>> new_z = pu.mapdomain(z, old, new); new_z
294: (4)          array([-1.0+1.j , -0.6+0.6j, -0.2+0.2j,  0.2-0.2j,  0.6-0.6j,  1.0-1.j ])
# may vary
295: (4)          """
296: (4)          x = np.asanyarray(x)
297: (4)          off, scl = mapparms(old, new)
298: (4)          return off + scl*x
299: (0)          def _nth_slice(i, ndim):
300: (4)          s1 = [np.newaxis] * ndim
301: (4)          s1[i] = slice(None)
302: (4)          return tuple(s1)
303: (0)          def _vander_nd(vander_fs, points, degrees):
304: (4)          """
305: (4)          A generalization of the Vandermonde matrix for N dimensions
306: (4)          The result is built by combining the results of 1d Vandermonde matrices,
307: (4)          .. math::
308: (8)          W[i_0, \ldots, i_M, j_0, \ldots, j_N] = \prod_{k=0}^N V_k(x_k)[i_0,
\ldots, i_M, j_k]
309: (4)          where
310: (4)          .. math::
311: (8)          N \&= \text{len}(points) = \text{len}(degrees) =
\text{len}(vander\_fs) \\
312: (8)          M \&= \text{points}[k].ndim \\
313: (8)          V_k \&= \text{vander\_fs}[k] \\
314: (8)          x_k \&= \text{points}[k] \\
315: (8)          0 \leq j_k \leq \text{degrees}[k]
316: (4)          Expanding the one-dimensional :math:`V_k` functions gives:
317: (4)          .. math::
318: (8)          W[i_0, \ldots, i_M, j_0, \ldots, j_N] = \prod_{k=0}^N B_{k,j_k}
(x_k[i_0, \ldots, i_M])
319: (4)          where :math:`B_{k,m}` is the m'th basis of the polynomial construction
used along
320: (4)          dimension :math:`k`. For a regular polynomial, :math:`B_{k,m}(x) = P_m(x)`
= x^m`.
321: (4)          Parameters
322: (4)          -----
323: (4)          vander_fs : Sequence[function(array_like, int) -> ndarray]
324: (8)          The 1d vander function to use for each axis, such as ``polyvander``
325: (4)          points : Sequence[array_like]
326: (8)          Arrays of point coordinates, all of the same shape. The dtypes
327: (8)          will be converted to either float64 or complex128 depending on
328: (8)          whether any of the elements are complex. Scalars are converted to
329: (8)          1-D arrays.
330: (8)          This must be the same length as `vander_fs`.
331: (4)          degrees : Sequence[int]
332: (8)          The maximum degree (inclusive) to use for each axis.
333: (8)          This must be the same length as `vander_fs`.
334: (4)          Returns
335: (4)          -----
336: (4)          vander_nd : ndarray
337: (8)          An array of shape ``points[0].shape + tuple(d + 1 for d in degrees)``.
338: (4)          """
339: (4)          n_dims = len(vander_fs)
340: (4)          if n_dims != len(points):

```

```

341: (8)
342: (12)
{len(points)})")
343: (4)
344: (8)
345: (12)
346: (4)
347: (8)
given")
348: (4)
349: (4)
350: (8)
351: (8)
352: (4)
353: (4)
354: (0)
355: (4)
356: (4)
single axis
357: (4)
358: (4)
359: (4)
360: (4)
361: (0)
362: (4)
363: (4)
364: (4)
365: (4)
366: (4)
367: (8)
368: (4)
369: (8)
370: (4)
371: (8)
372: (4)
373: (4)
374: (8)
375: (4)
376: (8)
377: (8)
378: (8)
379: (8)
380: (8)
381: (12)
382: (12)
383: (12)
384: (16)
385: (12)
386: (12)
387: (8)
388: (0)
389: (4)
390: (4)
391: (4)
392: (4)
393: (4)
394: (8)
395: (4)
396: (8)
397: (4)
398: (4)
399: (4)
400: (4)
401: (8)
402: (12)
403: (8)
404: (12)
405: (8)
406: (12)

        raise ValueError(
                    f"Expected {n_dims} dimensions of sample points, got"
            if n_dims != len(degrees):
                raise ValueError(
                    f"Expected {n_dims} dimensions of degrees, got {len(degrees)}")
            if n_dims == 0:
                raise ValueError("Unable to guess a dtype or shape when no points are
given")
            points = tuple(np.array(tuple(points), copy=False) + 0.0)
            vander_arrays = (
                vander_fs[i](points[i], degrees[i])[(..., ) + _nth_slice(i, n_dims)]
                for i in range(n_dims)
            )
            return functools.reduce(operator.mul, vander_arrays)
def _vander_nd_flat(vander_fs, points, degrees):
    """
        Like `_vander_nd`, but flattens the last ``len(degrees)`` axes into a
    Used to implement the public ``<type>vander<n>d`` functions.
    """
    v = _vander_nd(vander_fs, points, degrees)
    return v.reshape(v.shape[:-len(degrees)] + (-1,))
def _fromroots(line_f, mul_f, roots):
    """
        Helper function used to implement the ``<type>fromroots`` functions.
    Parameters
    -----
        line_f : function(float, float) -> ndarray
            The ``<type>line`` function, such as ``polyline``
        mul_f : function(array_like, array_like) -> ndarray
            The ``<type>mul`` function, such as ``polymul``
        roots
            See the ``<type>fromroots`` functions for more detail
    """
    if len(roots) == 0:
        return np.ones(1)
    else:
        [roots] = as_series([roots], trim=False)
        roots.sort()
        p = [line_f(-r, 1) for r in roots]
        n = len(p)
        while n > 1:
            m, r = divmod(n, 2)
            tmp = [mul_f(p[i], p[i+m]) for i in range(m)]
            if r:
                tmp[0] = mul_f(tmp[0], p[-1])
            p = tmp
            n = m
        return p[0]
def _valnd(val_f, c, *args):
    """
        Helper function used to implement the ``<type>val<n>d`` functions.
    Parameters
    -----
        val_f : function(array_like, array_like, tensor: bool) -> array_like
            The ``<type>val`` function, such as ``polyval``
        c, args
            See the ``<type>val<n>d`` functions for more detail
    """
    args = [np.asarray(a) for a in args]
    shape0 = args[0].shape
    if not all((a.shape == shape0 for a in args[1:])):
        if len(args) == 3:
            raise ValueError('x, y, z are incompatible')
        elif len(args) == 2:
            raise ValueError('x, y are incompatible')
        else:
            raise ValueError('ordinates are incompatible')

```

```

407: (4)             it = iter(args)
408: (4)             x0 = next(it)
409: (4)             c = val_f(x0, c)
410: (4)             for xi in it:
411: (8)                 c = val_f(xi, c, tensor=False)
412: (4)             return c
413: (0)         def _gridnd(val_f, c, *args):
414: (4)             """
415: (4)             Helper function used to implement the ``<type>grid<n>d`` functions.
416: (4)             Parameters
417: (4)             -----
418: (4)             val_f : function(array_like, array_like, tensor: bool) -> array_like
419: (8)                 The ``<type>val`` function, such as ``polyval``
420: (4)                 c, args
421: (8)                 See the ``<type>grid<n>d`` functions for more detail
422: (4)             """
423: (4)             for xi in args:
424: (8)                 c = val_f(xi, c)
425: (4)             return c
426: (0)         def _div(mul_f, c1, c2):
427: (4)             """
428: (4)             Helper function used to implement the ``<type>div`` functions.
429: (4)             Implementation uses repeated subtraction of c2 multiplied by the nth
basis.
430: (4)             For some polynomial types, a more efficient approach may be possible.
431: (4)             Parameters
432: (4)             -----
433: (4)             mul_f : function(array_like, array_like) -> array_like
434: (8)                 The ``<type>mul`` function, such as ``polymul``
435: (4)                 c1, c2
436: (8)                 See the ``<type>div`` functions for more detail
437: (4)             """
438: (4)             [c1, c2] = as_series([c1, c2])
439: (4)             if c2[-1] == 0:
440: (8)                 raise ZeroDivisionError()
441: (4)             lc1 = len(c1)
442: (4)             lc2 = len(c2)
443: (4)             if lc1 < lc2:
444: (8)                 return c1[:1]*0, c1
445: (4)             elif lc2 == 1:
446: (8)                 return c1/c2[-1], c1[:1]*0
447: (4)             else:
448: (8)                 quo = np.empty(lc1 - lc2 + 1, dtype=c1.dtype)
449: (8)                 rem = c1
450: (8)                 for i in range(lc1 - lc2, -1, -1):
451: (12)                     p = mul_f([0]*i + [1], c2)
452: (12)                     q = rem[-1]/p[-1]
453: (12)                     rem = rem[:-1] - q*p[:-1]
454: (12)                     quo[i] = q
455: (8)                 return quo, trimseq(rem)
456: (0)         def _add(c1, c2):
457: (4)             """ Helper function used to implement the ``<type>add`` functions. """
458: (4)             [c1, c2] = as_series([c1, c2])
459: (4)             if len(c1) > len(c2):
460: (8)                 c1[:c2.size] += c2
461: (8)                 ret = c1
462: (4)             else:
463: (8)                 c2[:c1.size] += c1
464: (8)                 ret = c2
465: (4)             return trimseq(ret)
466: (0)         def _sub(c1, c2):
467: (4)             """ Helper function used to implement the ``<type>sub`` functions. """
468: (4)             [c1, c2] = as_series([c1, c2])
469: (4)             if len(c1) > len(c2):
470: (8)                 c1[:c2.size] -= c2
471: (8)                 ret = c1
472: (4)             else:
473: (8)                 c2 = -c2
474: (8)                 c2[:c1.size] += c1

```

```

475: (8)           ret = c2
476: (4)           return trimseq(ret)
477: (0)           def _fit(vander_f, x, y, deg, rcond=None, full=False, w=None):
478: (4)               """
479: (4)                   Helper function used to implement the ``<type>fit`` functions.
480: (4)                   Parameters
481: (4)                   -----
482: (4)                   vander_f : function(array_like, int) -> ndarray
483: (8)                       The 1d vander function, such as ``polyvander``
484: (4)                   c1, c2
485: (8)                       See the ``<type>fit`` functions for more detail
486: (4)               """
487: (4)               x = np.asarray(x) + 0.0
488: (4)               y = np.asarray(y) + 0.0
489: (4)               deg = np.asarray(deg)
490: (4)               if deg.ndim > 1 or deg.dtype.kind not in 'iu' or deg.size == 0:
491: (8)                   raise TypeError("deg must be an int or non-empty 1-D array of int")
492: (4)               if deg.min() < 0:
493: (8)                   raise ValueError("expected deg >= 0")
494: (4)               if x.ndim != 1:
495: (8)                   raise TypeError("expected 1D vector for x")
496: (4)               if x.size == 0:
497: (8)                   raise TypeError("expected non-empty vector for x")
498: (4)               if y.ndim < 1 or y.ndim > 2:
499: (8)                   raise TypeError("expected 1D or 2D array for y")
500: (4)               if len(x) != len(y):
501: (8)                   raise TypeError("expected x and y to have same length")
502: (4)               if deg.ndim == 0:
503: (8)                   lmax = deg
504: (8)                   order = lmax + 1
505: (8)                   van = vander_f(x, lmax)
506: (4)               else:
507: (8)                   deg = np.sort(deg)
508: (8)                   lmax = deg[-1]
509: (8)                   order = len(deg)
510: (8)                   van = vander_f(x, lmax)[:, deg]
511: (4)               lhs = van.T
512: (4)               rhs = y.T
513: (4)               if w is not None:
514: (8)                   w = np.asarray(w) + 0.0
515: (8)                   if w.ndim != 1:
516: (12)                       raise TypeError("expected 1D vector for w")
517: (8)                   if len(x) != len(w):
518: (12)                       raise TypeError("expected x and w to have same length")
519: (8)                   lhs = lhs * w
520: (8)                   rhs = rhs * w
521: (4)               if rcond is None:
522: (8)                   rcond = len(x)*np.finfo(x.dtype).eps
523: (4)               if issubclass(lhs.dtype.type, np.complexloating):
524: (8)                   scl = np.sqrt((np.square(lhs.real) + np.square(lhs.imag)).sum(1))
525: (4)               else:
526: (8)                   scl = np.sqrt(np.square(lhs).sum(1))
527: (4)                   scl[scl == 0] = 1
528: (4)               c, resids, rank, s = np.linalg.lstsq(lhs.T/scl, rhs.T, rcond)
529: (4)               c = (c.T/scl).T
530: (4)               if deg.ndim > 0:
531: (8)                   if c.ndim == 2:
532: (12)                       cc = np.zeros((lmax+1, c.shape[1]), dtype=c.dtype)
533: (8)                   else:
534: (12)                       cc = np.zeros(lmax+1, dtype=c.dtype)
535: (8)                       cc[deg] = c
536: (8)                       c = cc
537: (4)               if rank != order and not full:
538: (8)                   msg = "The fit may be poorly conditioned"
539: (8)                   warnings.warn(msg, RankWarning, stacklevel=2)
540: (4)               if full:
541: (8)                   return c, [resids, rank, s, rcond]
542: (4)               else:
543: (8)                   return c

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

544: (0)
545: (4)
546: (4)
547: (4)
548: (4)
549: (4)
550: (8)
551: (4)
552: (8)
553: (4)
554: (8)
555: (4)
556: (4)
557: (4)
558: (4)
559: (8)
560: (4)
561: (8)
562: (4)
563: (8)
564: (4)
565: (8)
566: (4)
567: (8)
568: (8)
569: (12)
570: (8)
571: (0)
572: (4)
573: (4)
574: (4)
575: (4)
576: (4)
577: (8)
578: (4)
579: (8)
580: (4)
581: (4)
582: (4)
583: (4)
584: (4)
585: (4)
586: (8)
587: (4)
588: (8)
589: (12)
590: (8)
591: (12)
592: (8)
593: (12)
594: (16)
595: (20)
596: (20)
597: (20)
598: (20)
599: (16)
600: (16)
601: (8)
602: (0)
603: (4)
604: (8)
605: (4)
606: (4)
607: (8)
608: (4)
609: (8)
610: (4)
611: (4)
612: (8)

def _pow(mul_f, c, pow, maxpower):
    """
        Helper function used to implement the ``<type>pow`` functions.
        Parameters
        -----
        mul_f : function(array_like, array_like) -> ndarray
            The ``<type>mul`` function, such as ``polymul``
        c : array_like
            1-D array of array of series coefficients
        pow, maxpower
            See the ``<type>pow`` functions for more detail
    """
    [c] = as_series([c])
    power = int(pow)
    if power != pow or power < 0:
        raise ValueError("Power must be a non-negative integer.")
    elif maxpower is not None and power > maxpower:
        raise ValueError("Power is too large")
    elif power == 0:
        return np.array([1], dtype=c.dtype)
    elif power == 1:
        return c
    else:
        prd = c
        for i in range(2, power + 1):
            prd = mul_f(prd, c)
        return prd

def _deprecate_as_int(x, desc):
    """
        Like `operator.index`, but emits a deprecation warning when passed a float
        Parameters
        -----
        x : int-like, or float with integral value
            Value to interpret as an integer
        desc : str
            description to include in any error message
        Raises
        -----
        TypeError : if x is a non-integral float or non-numeric
        DeprecationWarning : if x is an integral float
    """
    try:
        return operator.index(x)
    except TypeError as e:
        try:
            ix = int(x)
        except TypeError:
            pass
        else:
            if ix == x:
                warnings.warn(
                    f"In future, this will raise TypeError, as {desc} will "
                    "need to be an integer not just an integral float.",
                    DeprecationWarning,
                    stacklevel=3
                )
                return ix
    raise TypeError(f"{desc} must be an integer") from e

def format_float(x, parens=False):
    if not np.issubdtype(type(x), np.floating):
        return str(x)
    opts = np.get_printoptions()
    if np.isnan(x):
        return opts['nanstr']
    elif np.isinf(x):
        return opts['infstr']
    exp_format = False
    if x != 0:
        a = absolute(x)

```

```

613: (8)             if a >= 1.e8 or a < 10**min(0, -(opts['precision'])-1)//2):
614: (12)             exp_format = True
615: (4)             trim, unique = '0', True
616: (4)             if opts['floatmode'] == 'fixed':
617: (8)                 trim, unique = 'k', False
618: (4)             if exp_format:
619: (8)                 s = dragon4_scientific(x, precision=opts['precision'],
620: (31)                             unique=unique, trim=trim,
621: (31)                             sign=opts['sign'] == '+')
622: (8)             if parens:
623: (12)                 s = '(' + s + ')'
624: (4)             else:
625: (8)                 s = dragon4_positional(x, precision=opts['precision'],
626: (31)                             fractional=True,
627: (31)                             unique=unique, trim=trim,
628: (31)                             sign=opts['sign'] == '+')
629: (4)             return s
-----
```

## File 294 - setup.py:

```

1: (0)         def configuration(parent_package='', top_path=None):
2: (4)             from numpy.distutils.misc_util import Configuration
3: (4)             config = Configuration('polynomial', parent_package, top_path)
4: (4)             config.add_subpackage('tests')
5: (4)             config.add_data_files('*/*.pyi')
6: (4)             return config
7: (0)         if __name__ == '__main__':
8: (4)             from numpy.distutils.core import setup
9: (4)             setup(configuration=configuration)
-----
```

## File 295 - \_polybase.py:

```

1: (0)         """
2: (0)         Abstract base class for the various polynomial Classes.
3: (0)         The ABCPolyBase class provides the methods needed to implement the common API
4: (0)         for the various polynomial classes. It operates as a mixin, but uses the
5: (0)         abc module from the stdlib, hence it is only available for Python >= 2.6.
6: (0)         """
7: (0)         import os
8: (0)         import abc
9: (0)         import numbers
10: (0)        import numpy as np
11: (0)        from . import polyutils as pu
12: (0)        __all__ = ['ABCPolyBase']
13: (0)        class ABCPolyBase(abc.ABC):
14: (4)            """An abstract base class for immutable series classes.
15: (4)            ABCPolyBase provides the standard Python numerical methods
16: (4)            '+', '-', '*', '//', '%', 'divmod', '**', and '()' along with the
17: (4)            methods listed below.
18: (4)            .. versionadded:: 1.9.0
19: (4)            Parameters
20: (4)            -----
21: (4)            coef : array_like
22: (8)                Series coefficients in order of increasing degree, i.e.,
23: (8)                ``(1, 2, 3)`` gives ``1*P_0(x) + 2*P_1(x) + 3*P_2(x)``, where
24: (8)                ``P_i`` is the basis polynomials of degree ``i``.
25: (4)            domain : (2,) array_like, optional
26: (8)                Domain to use. The interval ``[domain[0], domain[1]]`` is mapped
27: (8)                to the interval ``[window[0], window[1]]`` by shifting and scaling.
28: (8)                The default value is the derived class domain.
29: (4)            window : (2,) array_like, optional
30: (8)                Window, see domain for its use. The default value is the
31: (8)                derived class window.
32: (4)            symbol : str, optional
33: (8)                Symbol used to represent the independent variable in string
-----
```

```

34: (8)                         representations of the polynomial expression, e.g. for printing.
35: (8)                         The symbol must be a valid Python identifier. Default value is 'x'.
36: (8)                         .. versionadded:: 1.24
37: (4)                         Attributes
38: (4)                         -----
39: (4)                         coef : (N,) ndarray
40: (8)                         Series coefficients in order of increasing degree.
41: (4)                         domain : (2,) ndarray
42: (8)                         Domain that is mapped to window.
43: (4)                         window : (2,) ndarray
44: (8)                         Window that domain is mapped to.
45: (4)                         symbol : str
46: (8)                         Symbol representing the independent variable.
47: (4)                         Class Attributes
48: (4)                         -----
49: (4)                         maxpower : int
50: (8)                         Maximum power allowed, i.e., the largest number ``n`` such that
51: (8)                         ```p(x)**n`` is allowed. This is to limit runaway polynomial size.
52: (4)                         domain : (2,) ndarray
53: (8)                         Default domain of the class.
54: (4)                         window : (2,) ndarray
55: (8)                         Default window of the class.
56: (4)
57: (4)                         """
58: (4)                         __hash__ = None
59: (4)                         __array_ufunc__ = None
60: (4)                         maxpower = 100
61: (4)                         _superscript_mapping = str.maketrans({
62: (8)                             "0": "\u00b2",
63: (8)                             "1": "\u00b3",
64: (8)                             "2": "\u00b4",
65: (8)                             "3": "\u00b5",
66: (8)                             "4": "\u00b9",
67: (8)                             "5": "\u00b8",
68: (8)                             "6": "\u00b9\u00b3",
69: (8)                             "7": "\u00b9\u00b2",
70: (8)                             "8": "\u00b9\u00b4",
71: (8)                             "9": "\u00b9\u00b5"
72: (4)                         })
73: (4)                         _subscript_mapping = str.maketrans({
74: (8)                             "0": "\u00b2,\u20ac",
75: (8)                             "1": "\u00b2,\u2074",
76: (8)                             "2": "\u00b2,,",
77: (8)                             "3": "\u00b2,f",
78: (8)                             "4": "\u00b2,,,",
79: (8)                             "5": "\u00b2,...",
80: (8)                             "6": "\u00b2,t",
81: (8)                             "7": "\u00b2,\u00b4",
82: (8)                             "8": "\u00b2,^",
83: (8)                             "9": "\u00b2,\u2044"
84: (4)                         _use_unicode = not os.name == 'nt'
85: (4)                         @property
86: (4)                         def symbol(self):
87: (8)                             return self._symbol
88: (4)                         @property
89: (4)                         @abc.abstractmethod
90: (4)                         def domain(self):
91: (8)                             pass
92: (4)                         @property
93: (4)                         @abc.abstractmethod
94: (4)                         def window(self):
95: (8)                             pass
96: (4)                         @property
97: (4)                         @abc.abstractmethod
98: (4)                         def basis_name(self):
99: (8)                             pass
100: (4)                        @staticmethod
101: (4)                        @abc.abstractmethod
102: (4)                        def _add(c1, c2):

```

```
103: (8)                                pass
104: (4)                                @staticmethod
105: (4)                                @abc.abstractmethod
106: (4)                                def _sub(c1, c2):
107: (8)                                pass
108: (4)                                @staticmethod
109: (4)                                @abc.abstractmethod
110: (4)                                def _mul(c1, c2):
111: (8)                                pass
112: (4)                                @staticmethod
113: (4)                                @abc.abstractmethod
114: (4)                                def _div(c1, c2):
115: (8)                                pass
116: (4)                                @staticmethod
117: (4)                                @abc.abstractmethod
118: (4)                                def _pow(c, pow, maxpower=None):
119: (8)                                pass
120: (4)                                @staticmethod
121: (4)                                @abc.abstractmethod
122: (4)                                def _val(x, c):
123: (8)                                pass
124: (4)                                @staticmethod
125: (4)                                @abc.abstractmethod
126: (4)                                def _int(c, m, k, lbnd, scl):
127: (8)                                pass
128: (4)                                @staticmethod
129: (4)                                @abc.abstractmethod
130: (4)                                def _der(c, m, scl):
131: (8)                                pass
132: (4)                                @staticmethod
133: (4)                                @abc.abstractmethod
134: (4)                                def _fit(x, y, deg, rcond, full):
135: (8)                                pass
136: (4)                                @staticmethod
137: (4)                                @abc.abstractmethod
138: (4)                                def _line(off, scl):
139: (8)                                pass
140: (4)                                @staticmethod
141: (4)                                @abc.abstractmethod
142: (4)                                def _roots(c):
143: (8)                                pass
144: (4)                                @staticmethod
145: (4)                                @abc.abstractmethod
146: (4)                                def _fromroots(r):
147: (8)                                pass
148: (4)                                def has_samecoef(self, other):
149: (8)                                    """Check if coefficients match.
150: (8)                                    .. versionadded:: 1.6.0
151: (8)                                    Parameters
152: (8)                                    -----
153: (8)                                    other : class instance
154: (12)                                    The other class must have the ``coef`` attribute.
155: (8)                                    Returns
156: (8)                                    -----
157: (8)                                    bool : boolean
158: (12)                                    True if the coefficients are the same, False otherwise.
159: (8)                                    """
160: (8)                                    if len(self.coef) != len(other.coef):
161: (12)                                    return False
162: (8)                                    elif not np.all(self.coef == other.coef):
163: (12)                                    return False
164: (8)                                    else:
165: (12)                                    return True
166: (4)                                def has_samedomain(self, other):
167: (8)                                    """Check if domains match.
168: (8)                                    .. versionadded:: 1.6.0
169: (8)                                    Parameters
170: (8)                                    -----
171: (8)                                    other : class instance
```

```

172: (12)                               The other class must have the ``domain`` attribute.
173: (8)                                Returns
174: (8)
175: (8)
176: (12)
177: (8)
178: (8)
179: (4)                                bool : boolean
180: (8)                                True if the domains are the same, False otherwise.
181: (8)
182: (8)
183: (8)
184: (8)
185: (12)                               The other class must have the ``window`` attribute.
186: (8)                                Returns
187: (8)
188: (8)
189: (12)                               bool : boolean
190: (8)                                True if the windows are the same, False otherwise.
191: (8)
192: (4)                                return np.all(self.domain == other.domain)
193: (8)                                def has_samewindow(self, other):
194: (8)                                  """Check if windows match.
195: (8)                                  .. versionadded:: 1.6.0
196: (8)                                Parameters
197: (8)                                  other : class instance
198: (12)                               The other class must have the ``window`` attribute.
199: (8)                                Returns
200: (8)
201: (8)
202: (12)                               bool : boolean
203: (8)                                True if other is same class as self
204: (8)
205: (4)                                return isinstance(other, self.__class__)
206: (8)                                def _get_coefficients(self, other):
207: (8)                                  """Interpret other as polynomial coefficients.
208: (8)                                  The `other` argument is checked to see if it is of the same
209: (8)                                  class as self with identical domain and window. If so,
210: (8)                                  return its coefficients, otherwise return `other`.
211: (8)
212: (8)
213: (8)
214: (12)                               other : object
215: (8)                                Class instance.
216: (8)
217: (8)
218: (12)                               Returns
219: (12)
220: (8)
221: (8)
222: (8)
223: (12)                               other : anything
224: (8)                                Object to be checked.
225: (8)
226: (12)
227: (16)
228: (12)
229: (16)
230: (12)
231: (16)
232: (12)
233: (16)
234: (12)
235: (8)
236: (4)                                if isinstance(other, ABCPolyBase):
237: (8)                                  if not isinstance(other, self.__class__):
238: (8)                                      raise TypeError("Polynomial types differ")
239: (8)                                  elif not np.all(self.domain == other.domain):
240: (8)                                      raise TypeError("Domains differ")

```

```

230: (12)                               elif not np.all(self.window == other.window):
231: (16)                                      raise TypeError("Windows differ")
232: (12)
233: (16)
234: (12)
235: (8)
236: (4)                                elif self.symbol != other.symbol:
237: (8)                                      raise ValueError("Polynomial symbols differ")
238: (8)
239: (8)
240: (12)                               return other.coef

```

```

def __init__(self, coef, domain=None, window=None, symbol='x'):
    [coef] = pu.as_series([coef], trim=False)
    self.coef = coef
    if domain is not None:
        [domain] = pu.as_series([domain], trim=False)

```

```

241: (12)             if len(domain) != 2:
242: (16)                 raise ValueError("Domain has wrong number of elements.")
243: (12)             self.domain = domain
244: (8)             if window is not None:
245: (12)                 [window] = pu.as_series([window], trim=False)
246: (12)                 if len(window) != 2:
247: (16)                     raise ValueError("Window has wrong number of elements.")
248: (12)                 self.window = window
249: (8)             try:
250: (12)                 if not symbol.isidentifier():
251: (16)                     raise ValueError(
252: (20)                         "Symbol string must be a valid Python identifier"
253: (16)                     )
254: (8)             except AttributeError:
255: (12)                 raise TypeError("Symbol must be a non-empty string")
256: (8)             self._symbol = symbol
257: (4)             def __repr__(self):
258: (8)                 coef = repr(self.coef)[6:-1]
259: (8)                 domain = repr(self.domain)[6:-1]
260: (8)                 window = repr(self.window)[6:-1]
261: (8)                 name = self.__class__.__name__
262: (8)                 return (f"{name}({{coef}}, domain={{domain}}, window={{window}}, "
263: (16)                     f"symbol='{{self.symbol}}')")
264: (4)             def __format__(self, fmt_str):
265: (8)                 if fmt_str == '':
266: (12)                     return self.__str__()
267: (8)                 if fmt_str not in ('ascii', 'unicode'):
268: (12)                     raise ValueError(
269: (16)                         f"Unsupported format string '{fmt_str}' passed to "
270: (16)                         f"{{self.__class__}}.__format__. Valid options are "
271: (16)                         f"'ascii' and 'unicode'"
272: (12)                     )
273: (8)                 if fmt_str == 'ascii':
274: (12)                     return self._generate_string(self._str_term_ascii)
275: (8)                 return self._generate_string(self._str_term_unicode)
276: (4)             def __str__(self):
277: (8)                 if self._use_unicode:
278: (12)                     return self._generate_string(self._str_term_unicode)
279: (8)                 return self._generate_string(self._str_term_ascii)
280: (4)             def _generate_string(self, term_method):
281: (8)                 """
282: (8)                 Generate the full string representation of the polynomial, using
283: (8)                 ``term_method`` to generate each polynomial term.
284: (8)                 """
285: (8)                 linewidth = np.get_printoptions().get('linewidth', 75)
286: (8)                 if linewidth < 1:
287: (12)                     linewidth = 1
288: (8)                 out = pu.format_float(self.coef[0])
289: (8)                 for i, coef in enumerate(self.coef[1:]):
290: (12)                     out += " "
291: (12)                     power = str(i + 1)
292: (12)                     try:
293: (16)                         if coef >= 0:
294: (20)                             next_term = f"+ " + pu.format_float(coef, parens=True)
295: (16)                         else:
296: (20)                             next_term = f"- " + pu.format_float(-coef, parens=True)
297: (12)                     except TypeError:
298: (16)                         next_term = f"+ {coef}"
299: (12)                     next_term += term_method(power, self.symbol)
300: (12)                     line_len = len(out.split('\n')[-1]) + len(next_term)
301: (12)                     if i < len(self.coef[1:]) - 1:
302: (16)                         line_len += 2
303: (12)                         if line_len >= linewidth:
304: (16)                             next_term = next_term.replace(" ", "\n", 1)
305: (12)                         out += next_term
306: (8)                     return out
307: (4)             @classmethod
308: (4)             def _str_term_unicode(cls, i, arg_str):
309: (8)                 """

```

```

310: (8)           String representation of single polynomial term using unicode
311: (8)           characters for superscripts and subscripts.
312: (8)
313: (8)
314: (12)          if cls.basis_name is None:
315: (16)            raise NotImplementedError(
316: (16)              "Subclasses must define either a basis_name, or override "
317: (12)                "_str_term_unicode(cls, i, arg_str)"
318: (8)            )
319: (16)          return f"Â·{cls.basis_name}{i.translate(cls._subscript_mapping)}"
320: (4)            f"({arg_str})"
321: (4) @classmethod
322: (8)          def _str_term_ascii(cls, i, arg_str):
323: (8)            """
324: (8)              String representation of a single polynomial term using ** and _ to
325: (8)              represent superscripts and subscripts, respectively.
326: (8)            """
327: (12)          if cls.basis_name is None:
328: (16)            raise NotImplementedError(
329: (16)              "Subclasses must define either a basis_name, or override "
330: (12)                "_str_term_ascii(cls, i, arg_str)"
331: (8)            )
332: (4)          return f" {cls.basis_name}_{i}({arg_str})"
333: (4) @classmethod
334: (8)          def _repr_latex_term(cls, i, arg_str, needs_parens):
335: (12)            if cls.basis_name is None:
336: (16)              raise NotImplementedError(
337: (16)                  "Subclasses must define either a basis name, or override "
338: (8)                    "_repr_latex_term(i, arg_str, needs_parens)"
339: (4)            )
340: (4) @staticmethod
341: (8)          def _repr_latex_scalar(x, parens=False):
342: (4)            return r'\text{{{}}}'.format(pu.format_float(x, parens=parens))
343: (8)          def _repr_latex_(self):
344: (8)            off, scale = self.mapparms()
345: (12)            if off == 0 and scale == 1:
346: (12)              term = self.symbol
347: (8)              needs_parens = False
348: (12)            elif scale == 1:
349: (12)              term = f"{self._repr_latex_scalar(off)} + {self.symbol}"
350: (8)              needs_parens = True
351: (12)            elif off == 0:
352: (12)              term = f"{self._repr_latex_scalar(scale)}{self.symbol}"
353: (8)              needs_parens = True
354: (12)            else:
355: (16)              term = (
356: (16)                  f"{self._repr_latex_scalar(off)} + "
357: (12)                    f"{self._repr_latex_scalar(scale)}{self.symbol}"
358: (12)              )
359: (8)              needs_parens = True
360: (8)            mute = r"\color{{LightGray}}{{{}}}".format
361: (8)            parts = []
362: (12)            for i, c in enumerate(self.coef):
363: (16)              if i == 0:
364: (12)                coef_str = f"{self._repr_latex_scalar(c)}"
365: (16)              elif not isinstance(c, numbers.Real):
366: (12)                coef_str = f" + {self._repr_latex_scalar(c)}"
367: (16)              elif not np.signbit(c):
368: (12)                coef_str = f" + {self._repr_latex_scalar(c, parens=True)}"
369: (16)              else:
370: (12)                coef_str = f" - {self._repr_latex_scalar(-c, parens=True)}"
371: (12)            term_str = self._repr_latex_term(i, term, needs_parens)
372: (16)            if term_str == '1':
373: (12)              part = coef_str
374: (16)            else:
375: (12)              part = rf"{{coef_str}}\,{term_str}"
376: (16)            if c == 0:
377: (12)              part = mute(part)
378: (8)            parts.append(part)

```

```

379: (12)                                body = ''.join(parts)
380: (8)                                 else:
381: (12)                                body = '0'
382: (8)                                 return rf"${self.symbol} \mapsto {body}$"
383: (4)                                 def __getstate__(self):
384: (8)                                    ret = self.__dict__.copy()
385: (8)                                    ret['coef'] = self.coef.copy()
386: (8)                                    ret['domain'] = self.domain.copy()
387: (8)                                    ret['window'] = self.window.copy()
388: (8)                                    ret['symbol'] = self.symbol
389: (8)                                    return ret
390: (4)                                 def __setstate__(self, dict):
391: (8)                                    self.__dict__ = dict
392: (4)                                 def __call__(self, arg):
393: (8)                                    off, scl = pu.mapparms(self.domain, self.window)
394: (8)                                    arg = off + scl*arg
395: (8)                                    return self._val(arg, self.coef)
396: (4)                                 def __iter__(self):
397: (8)                                    return iter(self.coef)
398: (4)                                 def __len__(self):
399: (8)                                    return len(self.coef)
400: (4)                                 def __neg__(self):
401: (8)                                    return self.__class__(
402: (12)                                       -self.coef, self.domain, self.window, self.symbol
403: (8)                                     )
404: (4)                                 def __pos__(self):
405: (8)                                    return self
406: (4)                                 def __add__(self, other):
407: (8)                                    othercoef = self._get_coefficients(other)
408: (8)                                    try:
409: (12)                                       coef = self._add(self.coef, othercoef)
410: (8)                                    except Exception:
411: (12)                                       return NotImplemented
412: (8)                                    return self.__class__(coef, self.domain, self.window, self.symbol)
413: (4)                                 def __sub__(self, other):
414: (8)                                    othercoef = self._get_coefficients(other)
415: (8)                                    try:
416: (12)                                       coef = self._sub(self.coef, othercoef)
417: (8)                                    except Exception:
418: (12)                                       return NotImplemented
419: (8)                                    return self.__class__(coef, self.domain, self.window, self.symbol)
420: (4)                                 def __mul__(self, other):
421: (8)                                    othercoef = self._get_coefficients(other)
422: (8)                                    try:
423: (12)                                       coef = self._mul(self.coef, othercoef)
424: (8)                                    except Exception:
425: (12)                                       return NotImplemented
426: (8)                                    return self.__class__(coef, self.domain, self.window, self.symbol)
427: (4)                                 def __truediv__(self, other):
428: (8)                                    if not isinstance(other, numbers.Number) or isinstance(other, bool):
429: (12)                                       raise TypeError(
430: (16)                                         f"unsupported types for true division: "
431: (16)                                         f"'{type(self)}', '{type(other)}'"
432: (12)                                       )
433: (8)                                    return self.__floordiv__(other)
434: (4)                                 def __floordiv__(self, other):
435: (8)                                    res = self.__divmod__(other)
436: (8)                                    if res is NotImplemented:
437: (12)                                       return res
438: (8)                                    return res[0]
439: (4)                                 def __mod__(self, other):
440: (8)                                    res = self.__divmod__(other)
441: (8)                                    if res is NotImplemented:
442: (12)                                       return res
443: (8)                                    return res[1]
444: (4)                                 def __divmod__(self, other):
445: (8)                                    othercoef = self._get_coefficients(other)
446: (8)                                    try:
447: (12)                                       quo, rem = self._div(self.coef, othercoef)

```

```

448: (8)
449: (12)
450: (8)
451: (12)
452: (8)
453: (8)
454: (8)
455: (4)
456: (8)
457: (8)
458: (8)
459: (4)
460: (8)
461: (12)
462: (8)
463: (12)
464: (8)
465: (4)
466: (8)
467: (12)
468: (8)
469: (12)
470: (8)
471: (4)
472: (8)
473: (12)
474: (8)
475: (12)
476: (8)
477: (4)
478: (8)
479: (4)
480: (8)
481: (4)
482: (8)
483: (8)
484: (12)
485: (8)
486: (4)
487: (8)
488: (8)
489: (12)
490: (8)
491: (4)
492: (8)
493: (12)
494: (8)
495: (12)
496: (8)
497: (12)
498: (8)
499: (8)
500: (8)
501: (4)
502: (8)
503: (15)
504: (15)
505: (15)
506: (15)
507: (15)
508: (8)
509: (4)
510: (8)
511: (4)
512: (8)
513: (8)
514: (8)
515: (8)
516: (12)

        except ZeroDivisionError:
            raise
        except Exception:
            return NotImplemented
        quo = self.__class__(quo, self.domain, self.window, self.symbol)
        rem = self.__class__(rem, self.domain, self.window, self.symbol)
        return quo, rem
    def __pow__(self, other):
        coef = self._pow(self.coef, other, maxpower=self.maxpower)
        res = self.__class__(coef, self.domain, self.window, self.symbol)
        return res
    def __radd__(self, other):
        try:
            coef = self._add(other, self.coef)
        except Exception:
            return NotImplemented
        return self.__class__(coef, self.domain, self.window, self.symbol)
    def __rsub__(self, other):
        try:
            coef = self._sub(other, self.coef)
        except Exception:
            return NotImplemented
        return self.__class__(coef, self.domain, self.window, self.symbol)
    def __rmul__(self, other):
        try:
            coef = self._mul(other, self.coef)
        except Exception:
            return NotImplemented
        return self.__class__(coef, self.domain, self.window, self.symbol)
    def __rdiv__(self, other):
        return self.__rfloordiv__(other)
    def __rtruediv__(self, other):
        return NotImplemented
    def __rfloordiv__(self, other):
        res = self.__rdivmod__(other)
        if res is NotImplemented:
            return res
        return res[0]
    def __rmod__(self, other):
        res = self.__rdivmod__(other)
        if res is NotImplemented:
            return res
        return res[1]
    def __rdivmod__(self, other):
        try:
            quo, rem = self._div(other, self.coef)
        except ZeroDivisionError:
            raise
        except Exception:
            return NotImplemented
        quo = self.__class__(quo, self.domain, self.window, self.symbol)
        rem = self.__class__(rem, self.domain, self.window, self.symbol)
        return quo, rem
    def __eq__(self, other):
        res = (isinstance(other, self.__class__) and
               np.all(self.domain == other.domain) and
               np.all(self.window == other.window) and
               (self.coef.shape == other.coef.shape) and
               np.all(self.coef == other.coef) and
               (self.symbol == other.symbol))
        return res
    def __ne__(self, other):
        return not self.__eq__(other)
    def copy(self):
        """Return a copy.
        Returns
        -----
        new_series : series
            Copy of self.

```

```

517: (8)
518: (8)
self.symbol)
519: (4)
520: (8)
521: (8)
522: (8)
523: (8)
524: (8)
525: (12)
526: (8)
527: (8)
528: (8)
529: (8)
530: (8)
531: (8)
532: (8)
533: (8)
534: (8)
535: (8)
536: (8)
537: (8)
538: (8)
539: (8)
540: (8)
541: (8)
542: (8)
543: (8)
544: (8)
545: (4)
546: (8)
547: (8)
548: (8)
549: (8)
550: (8)
551: (8)
552: (8)
553: (8)
554: (8)
555: (8)
556: (12)
557: (12)
558: (8)
559: (8)
560: (8)
561: (12)
562: (8)
563: (8)
564: (4)
565: (8)
566: (8)
567: (8)
568: (8)
569: (8)
570: (8)
571: (8)
572: (8)
573: (8)
574: (12)
575: (8)
576: (8)
577: (8)
578: (12)
579: (8)
580: (8)
581: (8)
582: (4)
583: (8)
584: (8)

        """
        return self.__class__(self.coef, self.domain, self.window,
def degree(self):
    """The degree of the series.
.. versionadded:: 1.5.0
Returns
-----
degree : int
    Degree of the series, one less than the number of coefficients.
Examples
-----
Create a polynomial object for ``1 + 7*x + 4*x**2`` :
>>> poly = np.polynomial.Polynomial([1, 7, 4])
>>> print(poly)
1.0 + 7.0*x + 4.0*x^2
>>> poly.degree()
2
Note that this method does not check for non-zero coefficients.
You must trim the polynomial to remove any trailing zeroes:
>>> poly = np.polynomial.Polynomial([1, 7, 0])
>>> print(poly)
1.0 + 7.0*x + 0.0*x^2
>>> poly.degree()
2
>>> poly.trim().degree()
1
"""
return len(self) - 1
def cutdeg(self, deg):
    """Truncate series to the given degree.
Reduce the degree of the series to `deg` by discarding the
high order terms. If `deg` is greater than the current degree a
copy of the current series is returned. This can be useful in least
squares where the coefficients of the high degree terms may be very
small.
.. versionadded:: 1.5.0
Parameters
-----
deg : non-negative int
    The series is reduced to degree `deg` by discarding the high
    order terms. The value of `deg` must be a non-negative integer.
Returns
-----
new_series : series
    New instance of series with reduced degree.
"""
return self.truncate(deg + 1)
def trim(self, tol=0):
    """Remove trailing coefficients
Remove trailing coefficients until a coefficient is reached whose
absolute value greater than `tol` or the beginning of the series is
reached. If all the coefficients would be removed the series is set
to ``[0]``. A new series instance is returned with the new
coefficients. The current instance remains unchanged.
Parameters
-----
tol : non-negative number.
    All trailing coefficients less than `tol` will be removed.
Returns
-----
new_series : series
    New instance of series with trimmed coefficients.
"""
coef = pu.trimcoef(self.coef, tol)
return self.__class__(coef, self.domain, self.window, self.symbol)
def truncate(self, size):
    """Truncate series to length `size`.
Reduce the series to length `size` by discarding the high

```

```

585: (8) degree terms. The value of `size` must be a positive integer. This
586: (8) can be useful in least squares where the coefficients of the
587: (8) high degree terms may be very small.
588: (8) Parameters
589: (8) -----
590: (8) size : positive int
591: (12) The series is reduced to length `size` by discarding the high
592: (12) degree terms. The value of `size` must be a positive integer.
593: (8) Returns
594: (8) -----
595: (8) new_series : series
596: (12) New instance of series with truncated coefficients.
597: (8) """
598: (8) isize = int(size)
599: (8) if isize != size or isize < 1:
600: (12)     raise ValueError("size must be a positive integer")
601: (8) if isize >= len(self.coef):
602: (12)     coef = self.coef
603: (8) else:
604: (12)     coef = self.coef[:isize]
605: (8) return self.__class__(coef, self.domain, self.window, self.symbol)
def convert(self, domain=None, kind=None, window=None):
    """Convert series to a different kind and/or domain and/or window.
Parameters
-----
domain : array_like, optional
    The domain of the converted series. If the value is None,
    the default domain of `kind` is used.
kind : class, optional
    The polynomial series type class to which the current instance
    should be converted. If kind is None, then the class of the
    current instance is used.
window : array_like, optional
    The window of the converted series. If the value is None,
    the default window of `kind` is used.
Returns
-----
new_series : series
    The returned class can be of different type than the current
    instance and/or have a different domain and/or different
    window.
Notes
-----
Conversion between domains and class types can result in
numerically ill defined series.
"""
if kind is None:
    kind = self.__class__
if domain is None:
    domain = kind.domain
if window is None:
    window = kind.window
return self(kind.identity(domain, window=window, symbol=self.symbol))
def mapparms(self):
    """Return the mapping parameters.
The returned values define a linear map ``off + scl*x`` that is
applied to the input arguments before the series is evaluated. The
map depends on the ``domain`` and ``window``; if the current
``domain`` is equal to the ``window`` the resulting map is the
identity. If the coefficients of the series instance are to be
used by themselves outside this class, then the linear function
must be substituted for the ``x`` in the standard representation of
the base polynomials.
Returns
-----
off, scl : float or complex
    The mapping function is defined by ``off + scl*x``.
Notes
-----

```

```

654: (8)             If the current domain is the interval ``[l1, r1]`` and the window
655: (8)             is ``[l2, r2]``, then the linear mapping function ``L`` is
656: (8)             defined by the equations::
657: (12)            L(l1) = l2
658: (12)            L(r1) = r2
659: (8)            """
660: (8)            return pu.mapparms(self.domain, self.window)
661: (4) def integ(self, m=1, k=[], lbnd=None):
662: (8)             """Integrate.
663: (8)             Return a series instance that is the definite integral of the
664: (8)             current series.
665: (8)             Parameters
666: (8)             -----
667: (8)             m : non-negative int
668: (12)             The number of integrations to perform.
669: (8)             k : array_like
670: (12)             Integration constants. The first constant is applied to the
671: (12)             first integration, the second to the second, and so on. The
672: (12)             list of values must less than or equal to `m` in length and any
673: (12)             missing values are set to zero.
674: (8)             lbnd : Scalar
675: (12)             The lower bound of the definite integral.
676: (8)             Returns
677: (8)             -----
678: (8)             new_series : series
679: (12)             A new series representing the integral. The domain is the same
680: (12)             as the domain of the integrated series.
681: (8)             """
682: (8)             off, scl = self.mapparms()
683: (8)             if lbnd is None:
684: (12)                 lbnd = 0
685: (8)             else:
686: (12)                 lbnd = off + scl*lbnd
687: (8)             coef = self._int(self.coef, m, k, lbnd, 1./scl)
688: (8)             return self.__class__(coef, self.domain, self.window, self.symbol)
689: (4) def deriv(self, m=1):
690: (8)             """Differentiate.
691: (8)             Return a series instance of that is the derivative of the current
692: (8)             series.
693: (8)             Parameters
694: (8)             -----
695: (8)             m : non-negative int
696: (12)             Find the derivative of order `m`.
697: (8)             Returns
698: (8)             -----
699: (8)             new_series : series
700: (12)             A new series representing the derivative. The domain is the same
701: (12)             as the domain of the differentiated series.
702: (8)             """
703: (8)             off, scl = self.mapparms()
704: (8)             coef = self._der(self.coef, m, scl)
705: (8)             return self.__class__(coef, self.domain, self.window, self.symbol)
706: (4) def roots(self):
707: (8)             """Return the roots of the series polynomial.
708: (8)             Compute the roots for the series. Note that the accuracy of the
709: (8)             roots decreases the further outside the `domain` they lie.
710: (8)             Returns
711: (8)             -----
712: (8)             roots : ndarray
713: (12)             Array containing the roots of the series.
714: (8)             """
715: (8)             roots = self._roots(self.coef)
716: (8)             return pu.mapdomain(roots, self.window, self.domain)
717: (4) def linspace(self, n=100, domain=None):
718: (8)             """Return x, y values at equally spaced points in domain.
719: (8)             Returns the x, y values at `n` linearly spaced points across the
720: (8)             domain. Here y is the value of the polynomial at the points x. By
721: (8)             default the domain is the same as that of the series instance.
722: (8)             This method is intended mostly as a plotting aid.

```

```

723: (8)          .. versionadded:: 1.5.0
724: (8)          Parameters
725: (8)
726: (8)
727: (12)         n : int, optional
728: (8)           Number of point pairs to return. The default value is 100.
729: (12)         domain : {None, array_like}, optional
730: (12)           If not None, the specified domain is used instead of that of
731: (12)           the calling instance. It should be of the form ``[beg,end]``.
732: (8)           The default is None which case the class domain is used.
733: (8)
734: (8)         Returns
735: (12)         -----
736: (12)         x, y : ndarray
737: (8)           x is equal to linspace(self.domain[0], self.domain[1], n) and
738: (8)           y is the series evaluated at element of x.
739: (12)         """
740: (8)           if domain is None:
741: (8)             domain = self.domain
742: (8)             x = np.linspace(domain[0], domain[1], n)
743: (8)             y = self(x)
744: (8)             return x, y
745: (4)         @classmethod
746: (8)         def fit(cls, x, y, deg, domain=None, rcond=None, full=False, w=None,
747: (8)             window=None, symbol='x'):
748: (8)             """Least squares fit to data.
749: (8)             Return a series instance that is the least squares fit to the data
750: (8)             `y` sampled at `x`. The domain of the returned instance can be
751: (8)             specified and this will often result in a superior fit with less
752: (8)             chance of ill conditioning.
753: (8)             Parameters
754: (12)             -----
755: (8)             x : array_like, shape (M,)
756: (12)               x-coordinates of the M sample points ``(x[i], y[i])``.
757: (8)             y : array_like, shape (M,)
758: (12)               y-coordinates of the M sample points ``(x[i], y[i])``.
759: (12)             deg : int or 1-D array_like
760: (12)               Degree(s) of the fitting polynomials. If `deg` is a single integer
761: (8)                 all terms up to and including the `deg`'th term are included in
762: (8)                 the
763: (12)                 fit. For NumPy versions >= 1.11.0 a list of integers specifying
764: (8)                 degrees of the terms to include may be used instead.
765: (12)             domain : {None, [beg, end], []}, optional
766: (12)               Domain to use for the returned series. If ``None``,
767: (12)               then a minimal domain that covers the points `x` is chosen. If
768: (12)               ``[]`` the class domain is used. The default value was the
769: (12)               class domain in NumPy 1.4 and ``None`` in later versions.
770: (12)               The ``[]`` option was added in numpy 1.5.0.
771: (12)             rcond : float, optional
772: (12)               Relative condition number of the fit. Singular values smaller
773: (12)               than this relative to the largest singular value will be
774: (8)                 ignored. The default value is len(x)*eps, where eps is the
775: (12)                 relative precision of the float type, about 2e-16 in most
776: (12)                 cases.
777: (12)             full : bool, optional
778: (12)               Switch determining nature of return value. When it is False
779: (8)                 (the default) just the coefficients are returned, when True
780: (12)                 diagnostic information from the singular value decomposition is
781: (12)                 also returned.
782: (12)             w : array_like, shape (M,), optional
783: (12)               Weights. If not None, the weight ``w[i]`` applies to the unsquared
784: (12)               residual ``y[i] - y_hat[i]`` at ``x[i]``. Ideally the weights are
785: (12)               chosen so that the errors of the products ``w[i]*y[i]`` all have
786: (8)                 the same variance. When using inverse-variance weighting, use
787: (12)                 ``w[i] = 1/sigma(y[i])``. The default value is None.
788: (12)               .. versionadded:: 1.5.0
789: (12)             window : {[beg, end]}, optional
789: (12)               Window to use for the returned series. The default
789: (12)               value is the default class domain
789: (12)               .. versionadded:: 1.6.0

```

```

790: (8)             symbol : str, optional
791: (12)            Symbol representing the independent variable. Default is 'x'.
792: (8)
793: (8)
794: (8)
795: (12)            Returns
796: (12)            -----
797: (12)            new_series : series
798: (12)            A series that represents the least squares fit to the data and
799: (12)            has the domain and window specified in the call. If the
800: (12)            coefficients for the unscaled and unshifted basis polynomials are
801: (12)            of interest, do ``new_series.convert().coef``.
802: (12)            [resid, rank, sv, rcond] : list
803: (12)            These values are only returned if ``full == True``
804: (12)            - resid -- sum of squared residuals of the least squares fit
805: (12)            - rank -- the numerical rank of the scaled Vandermonde matrix
806: (12)            - sv -- singular values of the scaled Vandermonde matrix
807: (12)            - rcond -- value of `rcond`.
808: (12)            For more details, see `linalg.lstsq`.
809: (8)            """
810: (8)            if domain is None:
811: (8)                domain = pu.getdomain(x)
812: (8)            elif type(domain) is list and len(domain) == 0:
813: (8)                domain = cls.domain
814: (8)            if window is None:
815: (8)                window = cls.window
816: (12)            xnew = pu.mapdomain(x, domain, window)
817: (12)            res = cls._fit(xnew, y, deg, w=w, rcond=rcond, full=full)
818: (16)            if full:
819: (12)                [coef, status] = res
820: (8)                return (
821: (12)                    cls(coef, domain=domain, window=window, symbol=symbol), status
822: (12)                )
823: (4)            else:
824: (4)                coef = res
825: (8)                return cls(coef, domain=domain, window=window, symbol=symbol)
826: (4)            @classmethod
827: (4)            def fromroots(cls, roots, domain=[], window=None, symbol='x'):
828: (8)                """Return series instance that has the specified roots.
829: (8)                Returns a series representing the product
830: (8)                `` $(x - r[0])(x - r[1])\dots(x - r[n-1])$ ``, where ``r`` is a
831: (8)                list of roots.
832: (8)                Parameters
833: (8)                -----
834: (12)                roots : array_like
835: (12)                List of roots.
836: (12)                domain : {[], None, array_like}, optional
837: (12)                Domain for the resulting series. If None the domain is the
838: (12)                interval from the smallest root to the largest. If [] the
839: (12)                domain is the class domain. The default is [].
840: (8)                window : {None, array_like}, optional
841: (12)                Window for the returned series. If None the class window is
842: (8)                used. The default is None.
843: (8)                symbol : str, optional
844: (8)                Symbol representing the independent variable. Default is 'x'.
845: (12)            Returns
846: (8)            -----
847: (8)            new_series : series
848: (8)            Series with the specified roots.
849: (8)            """
850: (12)            [roots] = pu.as_series([roots], trim=False)
851: (12)            if domain is None:
852: (12)                domain = pu.getdomain(roots)
853: (12)            elif type(domain) is list and len(domain) == 0:
854: (12)                domain = cls.domain
855: (12)            if window is None:
856: (12)                window = cls.window
857: (12)            deg = len(roots)
858: (12)            off, scl = pu.mapparms(domain, window)

```

```

859: (4) @classmethod
860: (4) def identity(cls, domain=None, window=None, symbol='x'):
861: (8)     """Identity function.
862: (8)     If ``p`` is the returned series, then ``p(x) == x`` for all
863: (8)     values of x.
864: (8)     Parameters
865: (8)     -----
866: (8)     domain : {None, array_like}, optional
867: (12)         If given, the array must be of the form ``[beg, end]``, where
868: (12)             ``beg`` and ``end`` are the endpoints of the domain. If None is
869: (12)             given then the class domain is used. The default is None.
870: (8)     window : {None, array_like}, optional
871: (12)         If given, the resulting array must be if the form
872: (12)             ``[beg, end]``, where ``beg`` and ``end`` are the endpoints of
873: (12)                 the window. If None is given then the class window is used. The
874: (12)                 default is None.
875: (8)     symbol : str, optional
876: (12)         Symbol representing the independent variable. Default is 'x'.
877: (8)     Returns
878: (8)     -----
879: (8)     new_series : series
880: (13)         Series of representing the identity.
881: (8)     """
882: (8)     if domain is None:
883: (12)         domain = cls.domain
884: (8)     if window is None:
885: (12)         window = cls.window
886: (8)     off, scl = pu.mapparms(window, domain)
887: (8)     coef = cls._line(off, scl)
888: (8)     return cls(coef, domain, window, symbol)
889: (4) @classmethod
890: (4) def basis(cls, deg, domain=None, window=None, symbol='x'):
891: (8)     """Series basis polynomial of degree `deg`.
892: (8)     Returns the series representing the basis polynomial of degree `deg`.
893: (8)     .. versionadded:: 1.7.0
894: (8)     Parameters
895: (8)     -----
896: (8)     deg : int
897: (12)         Degree of the basis polynomial for the series. Must be >= 0.
898: (8)     domain : {None, array_like}, optional
899: (12)         If given, the array must be of the form ``[beg, end]``, where
900: (12)             ``beg`` and ``end`` are the endpoints of the domain. If None is
901: (12)             given then the class domain is used. The default is None.
902: (8)     window : {None, array_like}, optional
903: (12)         If given, the resulting array must be if the form
904: (12)             ``[beg, end]``, where ``beg`` and ``end`` are the endpoints of
905: (12)                 the window. If None is given then the class window is used. The
906: (12)                 default is None.
907: (8)     symbol : str, optional
908: (12)         Symbol representing the independent variable. Default is 'x'.
909: (8)     Returns
910: (8)     -----
911: (8)     new_series : series
912: (12)         A series with the coefficient of the `deg` term set to one and
913: (12)             all others zero.
914: (8)     """
915: (8)     if domain is None:
916: (12)         domain = cls.domain
917: (8)     if window is None:
918: (12)         window = cls.window
919: (8)     ideg = int(deg)
920: (8)     if ideg != deg or ideg < 0:
921: (12)         raise ValueError("deg must be non-negative integer")
922: (8)     return cls([0]*ideg + [1], domain, window, symbol)
923: (4) @classmethod
924: (4) def cast(cls, series, domain=None, window=None):
925: (8)     """Convert series to series of this class.
926: (8)     The `series` is expected to be an instance of some polynomial
927: (8)         series of one of the types supported by by the numpy.polynomial

```

```

928: (8)             module, but could be some other class that supports the convert
929: (8)             method.
930: (8)             .. versionadded:: 1.7.0
931: (8)             Parameters
932: (8)             -----
933: (8)             series : series
934: (12)            The series instance to be converted.
935: (8)             domain : {None, array_like}, optional
936: (12)            If given, the array must be of the form ``[beg, end]``, where
937: (12)            ``beg`` and ``end`` are the endpoints of the domain. If None is
938: (12)            given then the class domain is used. The default is None.
939: (8)             window : {None, array_like}, optional
940: (12)            If given, the resulting array must be if the form
941: (12)            ``[beg, end]``, where ``beg`` and ``end`` are the endpoints of
942: (12)            the window. If None is given then the class window is used. The
943: (12)            default is None.
944: (8)             Returns
945: (8)             -----
946: (8)             new_series : series
947: (12)            A series of the same kind as the calling class and equal to
948: (12)            `series` when evaluated.
949: (8)             See Also
950: (8)             -----
951: (8)             convert : similar instance method
952: (8)             """
953: (8)             if domain is None:
954: (12)                 domain = cls.domain
955: (8)             if window is None:
956: (12)                 window = cls.window
957: (8)             return series.convert(domain, cls, window)
-----
```

#### File 296 - \_\_init\_\_.py:

```

1: (0)             """
2: (0)             A sub-package for efficiently dealing with polynomials.
3: (0)             Within the documentation for this sub-package, a "finite power series,"
4: (0)             i.e., a polynomial (also referred to simply as a "series") is represented
5: (0)             by a 1-D numpy array of the polynomial's coefficients, ordered from lowest
6: (0)             order term to highest. For example, array([1,2,3]) represents
7: (0)             ``P_0 + 2*P_1 + 3*P_2``, where P_n is the n-th order basis polynomial
8: (0)             applicable to the specific module in question, e.g., `polynomial` (which
9: (0)             "wraps" the "standard" basis) or `chebyshev`. For optimal performance,
10: (0)             all operations on polynomials, including evaluation at an argument, are
11: (0)             implemented as operations on the coefficients. Additional (module-specific)
12: (0)             information can be found in the docstring for the module of interest.
13: (0)             This package provides *convenience classes* for each of six different kinds
14: (0)             of polynomials:
15: (9)             =====      =====
16: (9)             **Name**      **Provides**
17: (9)             =====      =====
18: (9)             `~polynomial.Polynomial`      Power series
19: (9)             `~chebyshev.Chebyshev`      Chebyshev series
20: (9)             `~legendre.Legendre`      Legendre series
21: (9)             `~laguerre.Laguerre`      Laguerre series
22: (9)             `~hermite.Hermite`      Hermite series
23: (9)             `~hermite_e.HermiteE`      HermiteE series
24: (9)             =====      =====
25: (0)             These *convenience classes* provide a consistent interface for creating,
26: (0)             manipulating, and fitting data with polynomials of different bases.
27: (0)             The convenience classes are the preferred interface for the
`~numpy.polynomial`  

28: (0)             package, and are available from the ``numpy.polynomial`` namespace.
29: (0)             This eliminates the need to navigate to the corresponding submodules, e.g.
30: (0)             ``np.polynomial.Polynomial`` or ``np.polynomial.Chebyshev`` instead of
31: (0)             ``np.polynomial.polynomial.Polynomial`` or
32: (0)             ``np.polynomial.chebyshev.Chebyshev``, respectively.
33: (0)             The classes provide a more consistent and concise interface than the
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

34: (0) type-specific functions defined in the submodules for each type of polynomial.
35: (0) For example, to fit a Chebyshev polynomial with degree ``1`` to data given
36: (0) by arrays ``xdata`` and ``ydata``, the
37: (0) `~chebyshev.Chebyshev.fit` class method::
38: (4)     >>> from numpy.polynomial import Chebyshev
39: (4)     >>> c = Chebyshev.fit(xdata, ydata, deg=1)
40: (0) is preferred over the `chebyshev.chebfit` function from the
41: (0) ``np.polynomial.chebyshev`` module::
42: (4)     >>> from numpy.polynomial.chebyshev import chebfit
43: (4)     >>> c = chebfit(xdata, ydata, deg=1)
44: (0) See :doc:`routines.polynomials.classes` for more details.
45: (0) Convenience Classes
46: (0) =====
47: (0) The following lists the various constants and methods common to all of
48: (0) the classes representing the various kinds of polynomials. In the following,
49: (0) the term ``Poly`` represents any one of the convenience classes (e.g.
50: (0) `~polynomial.Polynomial`, `~chebyshev.Chebyshev`, `~hermite.Hermite`, etc.)
51: (0) while the lowercase ``p`` represents an **instance** of a polynomial class.
52: (0) Constants
53: (0) -----
54: (0) - ``Poly.domain`` -- Default domain
55: (0) - ``Poly.window`` -- Default window
56: (0) - ``Poly.basis_name`` -- String used to represent the basis
57: (0) - ``Poly.maxpower`` -- Maximum value ``n`` such that ``p**n`` is allowed
58: (0) - ``Poly.nickname`` -- String used in printing
59: (0) Creation
60: (0) -----
61: (0) Methods for creating polynomial instances.
62: (0) - ``Poly.basis(degree)`` -- Basis polynomial of given degree
63: (0) - ``Poly.identity()`` -- ``p`` where ``p(x) = x`` for all ``x``
64: (0) - ``Poly.fit(x, y, deg)`` -- ``p`` of degree ``deg`` with coefficients
65: (2) determined by the least-squares fit to the data ``x``, ``y``
66: (0) - ``Poly.fromroots(roots)`` -- ``p`` with specified roots
67: (0) - ``p.copy()`` -- Create a copy of ``p``
68: (0) Conversion
69: (0) -----
70: (0) Methods for converting a polynomial instance of one kind to another.
71: (0) - ``p.cast(Poly)`` -- Convert ``p`` to instance of kind ``Poly``
72: (0) - ``p.convert(Poly)`` -- Convert ``p`` to instance of kind ``Poly`` or map
73: (2) between ``domain`` and ``window``
74: (0) Calculus
75: (0) -----
76: (0) - ``p.deriv()`` -- Take the derivative of ``p``
77: (0) - ``p.integ()`` -- Integrate ``p``
78: (0) Validation
79: (0) -----
80: (0) - ``Poly.has_samecoef(p1, p2)`` -- Check if coefficients match
81: (0) - ``Poly.has_samedomain(p1, p2)`` -- Check if domains match
82: (0) - ``Poly.has_sametype(p1, p2)`` -- Check if types match
83: (0) - ``Poly.has_samewindow(p1, p2)`` -- Check if windows match
84: (0) Misc
85: (0) -----
86: (0) - ``p.linspace()`` -- Return ``x, p(x)`` at equally-spaced points in
``domain``
87: (0) - ``p.mapparms()`` -- Return the parameters for the linear mapping between
``domain`` and ``window``.
88: (2)
89: (0) - ``p.roots()`` -- Return the roots of `p`.
90: (0) - ``p.trim()`` -- Remove trailing coefficients.
91: (0) - ``p.cutdeg(degree)`` -- Truncate p to given degree
92: (0) - ``ptruncate(size)`` -- Truncate p to given size
93: (0) """
94: (0)     from .polynomial import Polynomial
95: (0)     from .chebyshev import Chebyshev
96: (0)     from .legendre import Legendre
97: (0)     from .hermite import Hermite
98: (0)     from .hermite_e import HermiteE
99: (0)     from .laguerre import Laguerre
100: (0)     __all__ = [
101: (4)         "set_default_printstyle",

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

102: (4)         "polynomial", "Polynomial",
103: (4)         "chebyshev", "Chebyshev",
104: (4)         "legendre", "Legendre",
105: (4)         "hermite", "Hermite",
106: (4)         "hermite_e", "HermiteE",
107: (4)         "laguerre", "Laguerre",
108: (0)
109: (0)     ]
110: (4)     def set_default_printstyle(style):
111: (4)         """
112: (4)             Set the default format for the string representation of polynomials.
113: (4)             Values for ``style`` must be valid inputs to ``__format__``, i.e. 'ascii'
114: (4)             or 'unicode'.
115: (4)             Parameters
116: (4)             -----
117: (4)             style : str
118: (8)                 Format string for default printing style. Must be either 'ascii' or
119: (4)                 'unicode'.
120: (4)             Notes
121: (4)             -----
122: (4)                 The default format depends on the platform: 'unicode' is used on
123: (4)                 Unix-based systems and 'ascii' on Windows. This determination is based on
124: (4)                 default font support for the unicode superscript and subscript ranges.
125: (4)             Examples
126: (4)             -----
127: (4)             >>> p = np.polynomial.Polynomial([1, 2, 3])
128: (4)             >>> c = np.polynomial.Chebyshev([1, 2, 3])
129: (4)             >>> np.polynomial.set_default_printstyle('unicode')
130: (4)             >>> print(p)
131: (4)                 1.0 + 2.0Â·x + 3.0Â·xÂ²
132: (4)             >>> print(c)
133: (4)                 1.0 + 2.0Â·Tâ,â(x) + 3.0Â·Tâ,,(x)
134: (4)             >>> np.polynomial.set_default_printstyle('ascii')
135: (4)             >>> print(p)
136: (4)                 1.0 + 2.0 x + 3.0 x**2
137: (4)             >>> print(c)
138: (4)                 1.0 + 2.0 T_1(x) + 3.0 T_2(x)
139: (4)             >>> # Formatting supersedes all class/package-level defaults
140: (4)             >>> print(f"{p:unicode}")
141: (4)                 1.0 + 2.0Â·x + 3.0Â·xÂ²
142: (4)             """
143: (8)             if style not in ('unicode', 'ascii'):
144: (12)                 raise ValueError(
145: (12)                     f"Unsupported format string '{style}'. Valid options are 'ascii' "
146: (12)                     f"and 'unicode'"
147: (8)             )
148: (4)             _use_unicode = True
149: (4)             if style == 'ascii':
150: (8)                 _use_unicode = False
151: (4)             from ._polybase import ABCPolyBase
152: (0)             ABCPolyBase._use_unicode = _use_unicode
153: (0)             from numpy._pytesttester import PytestTester
154: (0)             test = PytestTester(__name__)
155: (0)             del PytestTester

```

---

File 297 - test\_classes.py:

```

1: (0)         """Test inter-conversion of different polynomial classes.
2: (0)         This tests the convert and cast methods of all the polynomial classes.
3: (0)         """
4: (0)         import operator as op
5: (0)         from numbers import Number
6: (0)         import pytest
7: (0)         import numpy as np
8: (0)         from numpy.polynomial import (
9: (4)             Polynomial, Legendre, Chebyshev, Laguerre, Hermite, HermiteE)
10: (0)         from numpy.testing import (
11: (4)             assert_almost_equal, assert_raises, assert_equal, assert_

```

```

12: (4)
13: (0)     )
14: (0)     from numpy.polynomial.polyutils import RankWarning
15: (4)     classes = (
16: (4)         Polynomial, Legendre, Chebyshev, Laguerre,
17: (4)         Hermite, HermiteE
18: (0)     )
19: (0)     classids = tuple(cls.__name__ for cls in classes)
20: (0)     @pytest.fixture(params=classes, ids=classids)
21: (4)     def Poly(request):
22: (0)         return request.param
23: (0)     random = np.random.random
24: (0)     def assert_poly_almost_equal(p1, p2, msg=""):
25: (4)         try:
26: (8)             assert_(np.all(p1.domain == p2.domain))
27: (8)             assert_(np.all(p1.window == p2.window))
28: (8)             assert_almost_equal(p1.coef, p2.coef)
29: (4)         except AssertionError:
30: (8)             msg = f"Result: {p1}\nTarget: {p2}"
31: (8)             raise AssertionError(msg)
32: (0)     Poly1 = Poly
33: (0)     Poly2 = Poly
34: (0)     def test_conversion(Poly1, Poly2):
35: (4)         x = np.linspace(0, 1, 10)
36: (4)         coef = random((3,))
37: (4)         d1 = Poly1.domain + random((2,))*.25
38: (4)         w1 = Poly1.window + random((2,))*.25
39: (4)         p1 = Poly1(coef, domain=d1, window=w1)
40: (4)         d2 = Poly2.domain + random((2,))*.25
41: (4)         w2 = Poly2.window + random((2,))*.25
42: (4)         p2 = p1.convert(kind=Poly2, domain=d2, window=w2)
43: (4)         assert_almost_equal(p2.domain, d2)
44: (4)         assert_almost_equal(p2.window, w2)
45: (4)         assert_almost_equal(p2(x), p1(x))
45: (0)     def test_cast(Poly1, Poly2):
46: (4)         x = np.linspace(0, 1, 10)
47: (4)         coef = random((3,))
48: (4)         d1 = Poly1.domain + random((2,))*.25
49: (4)         w1 = Poly1.window + random((2,))*.25
50: (4)         p1 = Poly1(coef, domain=d1, window=w1)
51: (4)         d2 = Poly2.domain + random((2,))*.25
52: (4)         w2 = Poly2.window + random((2,))*.25
53: (4)         p2 = Poly2.cast(p1, domain=d2, window=w2)
54: (4)         assert_almost_equal(p2.domain, d2)
55: (4)         assert_almost_equal(p2.window, w2)
56: (4)         assert_almost_equal(p2(x), p1(x))
57: (0)     def test_identity(Poly):
58: (4)         d = Poly.domain + random((2,))*.25
59: (4)         w = Poly.window + random((2,))*.25
60: (4)         x = np.linspace(d[0], d[1], 11)
61: (4)         p = Poly.identity(domain=d, window=w)
62: (4)         assert_equal(p.domain, d)
63: (4)         assert_equal(p.window, w)
64: (4)         assert_almost_equal(p(x), x)
65: (0)     def test_basis(Poly):
66: (4)         d = Poly.domain + random((2,))*.25
67: (4)         w = Poly.window + random((2,))*.25
68: (4)         p = Poly.basis(5, domain=d, window=w)
69: (4)         assert_equal(p.domain, d)
70: (4)         assert_equal(p.window, w)
71: (4)         assert_equal(p.coef, [0]*5 + [1])
72: (0)     def test_fromroots(Poly):
73: (4)         d = Poly.domain + random((2,))*.25
74: (4)         w = Poly.window + random((2,))*.25
75: (4)         r = random((5,))
76: (4)         p1 = Poly.fromroots(r, domain=d, window=w)
77: (4)         assert_equal(p1.degree(), len(r))
78: (4)         assert_equal(p1.domain, d)
79: (4)         assert_equal(p1.window, w)
80: (4)         assert_almost_equal(p1(r), 0)

```

```

81: (4)           pdom = Polynomial.domain
82: (4)           pwin = Polynomial.window
83: (4)           p2 = Polynomial.cast(p1, domain=pdom, window=pwin)
84: (4)           assert_almost_equal(p2.coef[-1], 1)
85: (0)           def test_bad_conditioned_fit(Poly):
86: (4)             x = [0., 0., 1.]
87: (4)             y = [1., 2., 3.]
88: (4)             with pytest.warns(RankWarning) as record:
89: (8)               Poly.fit(x, y, 2)
90: (4)             assert record[0].message.args[0] == "The fit may be poorly conditioned"
91: (0)           def test_fit(Poly):
92: (4)             def f(x):
93: (8)               return x*(x - 1)*(x - 2)
94: (4)             x = np.linspace(0, 3)
95: (4)             y = f(x)
96: (4)             p = Poly.fit(x, y, 3)
97: (4)             assert_almost_equal(p.domain, [0, 3])
98: (4)             assert_almost_equal(p(x), y)
99: (4)             assert_equal(p.degree(), 3)
100: (4)            d = Poly.domain + random((2,))*.25
101: (4)            w = Poly.window + random((2,))*.25
102: (4)            p = Poly.fit(x, y, 3, domain=d, window=w)
103: (4)            assert_almost_equal(p(x), y)
104: (4)            assert_almost_equal(p.domain, d)
105: (4)            assert_almost_equal(p.window, w)
106: (4)            p = Poly.fit(x, y, [0, 1, 2, 3], domain=d, window=w)
107: (4)            assert_almost_equal(p(x), y)
108: (4)            assert_almost_equal(p.domain, d)
109: (4)            assert_almost_equal(p.window, w)
110: (4)            p = Poly.fit(x, y, 3, [])
111: (4)            assert_equal(p.domain, Poly.domain)
112: (4)            assert_equal(p.window, Poly.window)
113: (4)            p = Poly.fit(x, y, [0, 1, 2, 3], [])
114: (4)            assert_equal(p.domain, Poly.domain)
115: (4)            assert_equal(p.window, Poly.window)
116: (4)            w = np.zeros_like(x)
117: (4)            z = y + random(y.shape)*.25
118: (4)            w[::2] = 1
119: (4)            p1 = Poly.fit(x[::2], z[::2], 3)
120: (4)            p2 = Poly.fit(x, z, 3, w=w)
121: (4)            p3 = Poly.fit(x, z, [0, 1, 2, 3], w=w)
122: (4)            assert_almost_equal(p1(x), p2(x))
123: (4)            assert_almost_equal(p2(x), p3(x))
124: (0)           def test_equal(Poly):
125: (4)             p1 = Poly([1, 2, 3], domain=[0, 1], window=[2, 3])
126: (4)             p2 = Poly([1, 1, 1], domain=[0, 1], window=[2, 3])
127: (4)             p3 = Poly([1, 2, 3], domain=[1, 2], window=[2, 3])
128: (4)             p4 = Poly([1, 2, 3], domain=[0, 1], window=[1, 2])
129: (4)             assert_(p1 == p1)
130: (4)             assert_(not p1 == p2)
131: (4)             assert_(not p1 == p3)
132: (4)             assert_(not p1 == p4)
133: (0)           def test_not_equal(Poly):
134: (4)             p1 = Poly([1, 2, 3], domain=[0, 1], window=[2, 3])
135: (4)             p2 = Poly([1, 1, 1], domain=[0, 1], window=[2, 3])
136: (4)             p3 = Poly([1, 2, 3], domain=[1, 2], window=[2, 3])
137: (4)             p4 = Poly([1, 2, 3], domain=[0, 1], window=[1, 2])
138: (4)             assert_(not p1 != p1)
139: (4)             assert_(p1 != p2)
140: (4)             assert_(p1 != p3)
141: (4)             assert_(p1 != p4)
142: (0)           def test_add(Poly):
143: (4)             c1 = list(random((4,)) + .5)
144: (4)             c2 = list(random((3,)) + .5)
145: (4)             p1 = Poly(c1)
146: (4)             p2 = Poly(c2)
147: (4)             p3 = p1 + p2
148: (4)             assert_poly_almost_equal(p2 + p1, p3)
149: (4)             assert_poly_almost_equal(p1 + c2, p3)

```

```

150: (4) assert_poly_almost_equal(c2 + p1, p3)
151: (4) assert_poly_almost_equal(p1 + tuple(c2), p3)
152: (4) assert_poly_almost_equal(tuple(c2) + p1, p3)
153: (4) assert_poly_almost_equal(p1 + np.array(c2), p3)
154: (4) assert_poly_almost_equal(np.array(c2) + p1, p3)
155: (4) assert_raises(TypeError, op.add, p1, Poly([0], domain=Poly.domain + 1))
156: (4) assert_raises(TypeError, op.add, p1, Poly([0], window=Poly.window + 1))
157: (4) if Poly is Polynomial:
158: (8)     assert_raises(TypeError, op.add, p1, Chebyshev([0]))
159: (4) else:
160: (8)     assert_raises(TypeError, op.add, p1, Polynomial([0]))
def test_sub(Poly):
161: (0)     c1 = list(random((4,)) + .5)
162: (4)     c2 = list(random((3,)) + .5)
163: (4)     p1 = Poly(c1)
164: (4)     p2 = Poly(c2)
165: (4)     p3 = p1 - p2
166: (4)     assert_poly_almost_equal(p2 - p1, -p3)
167: (4)     assert_poly_almost_equal(p1 - c2, p3)
168: (4)     assert_poly_almost_equal(c2 - p1, -p3)
169: (4)     assert_poly_almost_equal(p1 - tuple(c2), p3)
170: (4)     assert_poly_almost_equal(tuple(c2) - p1, -p3)
171: (4)     assert_poly_almost_equal(p1 - np.array(c2), p3)
172: (4)     assert_poly_almost_equal(np.array(c2) - p1, -p3)
173: (4)     assert_raises(TypeError, op.sub, p1, Poly([0], domain=Poly.domain + 1))
174: (4)     assert_raises(TypeError, op.sub, p1, Poly([0], window=Poly.window + 1))
175: (4) if Poly is Polynomial:
176: (4)     assert_raises(TypeError, op.sub, p1, Chebyshev([0]))
177: (8) else:
178: (4)     assert_raises(TypeError, op.sub, p1, Polynomial([0]))
def test_mul(Poly):
179: (0)     c1 = list(random((4,)) + .5)
180: (4)     c2 = list(random((3,)) + .5)
181: (4)     p1 = Poly(c1)
182: (4)     p2 = Poly(c2)
183: (4)     p3 = p1 * p2
184: (4)     assert_poly_almost_equal(p2 * p1, p3)
185: (4)     assert_poly_almost_equal(p1 * c2, p3)
186: (4)     assert_poly_almost_equal(c2 * p1, p3)
187: (4)     assert_poly_almost_equal(p1 * tuple(c2), p3)
188: (4)     assert_poly_almost_equal(tuple(c2) * p1, p3)
189: (4)     assert_poly_almost_equal(p1 * np.array(c2), p3)
190: (4)     assert_poly_almost_equal(np.array(c2) * p1, p3)
191: (4)     assert_poly_almost_equal(p1 * 2, p1 * Poly([2]))
192: (4)     assert_poly_almost_equal(2 * p1, p1 * Poly([2]))
193: (4)     assert_raises(TypeError, op.mul, p1, Poly([0], domain=Poly.domain + 1))
194: (4)     assert_raises(TypeError, op.mul, p1, Poly([0], window=Poly.window + 1))
195: (4) if Poly is Polynomial:
196: (4)     assert_raises(TypeError, op.mul, p1, Chebyshev([0]))
197: (4) else:
198: (8)     assert_raises(TypeError, op.mul, p1, Polynomial([0]))
199: (4) def test_floordiv(Poly):
200: (8)     assert_raises(TypeError, op.mul, p1, Polynomial([0]))
201: (0)     c1 = list(random((4,)) + .5)
202: (4)     c2 = list(random((3,)) + .5)
203: (4)     c3 = list(random((2,)) + .5)
204: (4)     p1 = Poly(c1)
205: (4)     p2 = Poly(c2)
206: (4)     p3 = Poly(c3)
207: (4)     p4 = p1 * p2 + p3
208: (4)     c4 = list(p4.coef)
209: (4)     assert_poly_almost_equal(p4 // p2, p1)
210: (4)     assert_poly_almost_equal(p4 // c2, p1)
211: (4)     assert_poly_almost_equal(c4 // p2, p1)
212: (4)     assert_poly_almost_equal(p4 // tuple(c2), p1)
213: (4)     assert_poly_almost_equal(tuple(c4) // p2, p1)
214: (4)     assert_poly_almost_equal(p4 // np.array(c2), p1)
215: (4)     assert_poly_almost_equal(np.array(c4) // p2, p1)
216: (4)     assert_poly_almost_equal(2 // p2, Poly([0]))
217: (4)     assert_poly_almost_equal(p2 // 2, 0.5*p2)
218: (4)     assert_poly_almost_equal(p2 // 2, 0.5*p2)

```

```

219: (4)
220: (8)     assert_raises(
221: (4)         TypeError, op.floordiv, p1, Poly([0], domain=Poly.domain + 1))
222: (8)     assert_raises(
223: (4)         TypeError, op.floordiv, p1, Poly([0], window=Poly.window + 1))
223: (4)     if Poly is Polynomial:
224: (8)         assert_raises(TypeError, op.floordiv, p1, Chebyshev([0]))
225: (4)     else:
226: (8)         assert_raises(TypeError, op.floordiv, p1, Polynomial([0]))
227: (0) def test_truediv(Poly):
228: (4)     p1 = Poly([1,2,3])
229: (4)     p2 = p1 * 5
230: (4)     for stype in np.ScalarType:
231: (8)         if not issubclass(stype, Number) or issubclass(stype, bool):
232: (12)             continue
233: (8)         s = stype(5)
234: (8)         assert_poly_almost_equal(op.truediv(p2, s), p1)
235: (8)         assert_raises(TypeError, op.truediv, s, p2)
236: (4)     for stype in (int, float):
237: (8)         s = stype(5)
238: (8)         assert_poly_almost_equal(op.truediv(p2, s), p1)
239: (8)         assert_raises(TypeError, op.truediv, s, p2)
240: (4)     for stype in [complex]:
241: (8)         s = stype(5, 0)
242: (8)         assert_poly_almost_equal(op.truediv(p2, s), p1)
243: (8)         assert_raises(TypeError, op.truediv, s, p2)
244: (4)     for s in [tuple(), list(), dict(), bool(), np.array([1])]:
245: (8)         assert_raises(TypeError, op.truediv, p2, s)
246: (8)         assert_raises(TypeError, op.truediv, s, p2)
247: (4)     for ptype in classes:
248: (8)         assert_raises(TypeError, op.truediv, p2, ptype(1))
249: (0) def test_mod(Poly):
250: (4)     c1 = list(random((4,)) + .5)
251: (4)     c2 = list(random((3,)) + .5)
252: (4)     c3 = list(random((2,)) + .5)
253: (4)     p1 = Poly(c1)
254: (4)     p2 = Poly(c2)
255: (4)     p3 = Poly(c3)
256: (4)     p4 = p1 * p2 + p3
257: (4)     c4 = list(p4.coef)
258: (4)     assert_poly_almost_equal(p4 % p2, p3)
259: (4)     assert_poly_almost_equal(p4 % c2, p3)
260: (4)     assert_poly_almost_equal(c4 % p2, p3)
261: (4)     assert_poly_almost_equal(p4 % tuple(c2), p3)
262: (4)     assert_poly_almost_equal(tuple(c4) % p2, p3)
263: (4)     assert_poly_almost_equal(p4 % np.array(c2), p3)
264: (4)     assert_poly_almost_equal(np.array(c4) % p2, p3)
265: (4)     assert_poly_almost_equal(2 % p2, Poly([2]))
266: (4)     assert_poly_almost_equal(p2 % 2, Poly([0]))
267: (4)     assert_raises(TypeError, op.mod, p1, Poly([0], domain=Poly.domain + 1))
268: (4)     assert_raises(TypeError, op.mod, p1, Poly([0], window=Poly.window + 1))
269: (4)     if Poly is Polynomial:
270: (8)         assert_raises(TypeError, op.mod, p1, Chebyshev([0]))
271: (4)     else:
272: (8)         assert_raises(TypeError, op.mod, p1, Polynomial([0]))
273: (0) def test_divmod(Poly):
274: (4)     c1 = list(random((4,)) + .5)
275: (4)     c2 = list(random((3,)) + .5)
276: (4)     c3 = list(random((2,)) + .5)
277: (4)     p1 = Poly(c1)
278: (4)     p2 = Poly(c2)
279: (4)     p3 = Poly(c3)
280: (4)     p4 = p1 * p2 + p3
281: (4)     c4 = list(p4.coef)
282: (4)     quo, rem = divmod(p4, p2)
283: (4)     assert_poly_almost_equal(quo, p1)
284: (4)     assert_poly_almost_equal(rem, p3)
285: (4)     quo, rem = divmod(p4, c2)
286: (4)     assert_poly_almost_equal(quo, p1)
287: (4)     assert_poly_almost_equal(rem, p3)

```

```

288: (4) quo, rem = divmod(c4, p2)
289: (4) assert_poly_almost_equal(quo, p1)
290: (4) assert_poly_almost_equal(rem, p3)
291: (4) quo, rem = divmod(p4, tuple(c2))
292: (4) assert_poly_almost_equal(quo, p1)
293: (4) assert_poly_almost_equal(rem, p3)
294: (4) quo, rem = divmod(tuple(c4), p2)
295: (4) assert_poly_almost_equal(quo, p1)
296: (4) assert_poly_almost_equal(rem, p3)
297: (4) quo, rem = divmod(p4, np.array(c2))
298: (4) assert_poly_almost_equal(quo, p1)
299: (4) assert_poly_almost_equal(rem, p3)
300: (4) quo, rem = divmod(np.array(c4), p2)
301: (4) assert_poly_almost_equal(quo, p1)
302: (4) assert_poly_almost_equal(rem, p3)
303: (4) quo, rem = divmod(p2, 2)
304: (4) assert_poly_almost_equal(quo, 0.5*p2)
305: (4) assert_poly_almost_equal(rem, Poly([0]))
306: (4) quo, rem = divmod(2, p2)
307: (4) assert_poly_almost_equal(quo, Poly([0]))
308: (4) assert_poly_almost_equal(rem, Poly([2]))
309: (4) assert_raises(TypeError, divmod, p1, Poly([0], domain=Poly.domain + 1))
310: (4) assert_raises(TypeError, divmod, p1, Poly([0], window=Poly.window + 1))
311: (4) if Poly is Polynomial:
312: (8)     assert_raises(TypeError, divmod, p1, Chebyshev([0]))
313: (4) else:
314: (8)     assert_raises(TypeError, divmod, p1, Polynomial([0]))
315: (0) def test_roots(Poly):
316: (4)     d = Poly.domain * 1.25 + .25
317: (4)     w = Poly.window
318: (4)     tgt = np.linspace(d[0], d[1], 5)
319: (4)     res = np.sort(Poly.fromroots(tgt, domain=d, window=w).roots())
320: (4)     assert_almost_equal(res, tgt)
321: (4)     res = np.sort(Poly.fromroots(tgt).roots())
322: (4)     assert_almost_equal(res, tgt)
323: (0) def test_degree(Poly):
324: (4)     p = Poly.basis(5)
325: (4)     assert_equal(p.degree(), 5)
326: (0) def test_copy(Poly):
327: (4)     p1 = Poly.basis(5)
328: (4)     p2 = p1.copy()
329: (4)     assert_(p1 == p2)
330: (4)     assert_(p1 is not p2)
331: (4)     assert_(p1.coef is not p2.coef)
332: (4)     assert_(p1.domain is not p2.domain)
333: (4)     assert_(p1.window is not p2.window)
334: (0) def test_integ(Poly):
335: (4)     P = Polynomial
336: (4)     p0 = Poly.cast(P([1*2, 2*3, 3*4]))
337: (4)     p1 = P.cast(p0.integ())
338: (4)     p2 = P.cast(p0.integ(2))
339: (4)     assert_poly_almost_equal(p1, P([0, 2, 3, 4]))
340: (4)     assert_poly_almost_equal(p2, P([0, 0, 1, 1, 1]))
341: (4)     p0 = Poly.cast(P([1*2, 2*3, 3*4]))
342: (4)     p1 = P.cast(p0.integ(k=1))
343: (4)     p2 = P.cast(p0.integ(2, k=[1, 1]))
344: (4)     assert_poly_almost_equal(p1, P([1, 2, 3, 4]))
345: (4)     assert_poly_almost_equal(p2, P([1, 1, 1, 1, 1]))
346: (4)     p0 = Poly.cast(P([1*2, 2*3, 3*4]))
347: (4)     p1 = P.cast(p0.integ(lbnd=1))
348: (4)     p2 = P.cast(p0.integ(2, lbnd=1))
349: (4)     assert_poly_almost_equal(p1, P([-9, 2, 3, 4]))
350: (4)     assert_poly_almost_equal(p2, P([6, -9, 1, 1, 1]))
351: (4)     d = 2*Poly.domain
352: (4)     p0 = Poly.cast(P([1*2, 2*3, 3*4]), domain=d)
353: (4)     p1 = P.cast(p0.integ())
354: (4)     p2 = P.cast(p0.integ(2))
355: (4)     assert_poly_almost_equal(p1, P([0, 2, 3, 4]))
356: (4)     assert_poly_almost_equal(p2, P([0, 0, 1, 1, 1]))

```

```

357: (0)
358: (4)
359: (4)
360: (4)
361: (4)
362: (4)
363: (4)
364: (4)
365: (4)
366: (4)
367: (4)
368: (4)
369: (4)
370: (0)
371: (4)
372: (4)
373: (4)
374: (4)
375: (4)
376: (4)
377: (4)
378: (4)
379: (4)
380: (4)
381: (4)
382: (4)
383: (4)
384: (0)
385: (4)
386: (4)
387: (4)
388: (4)
389: (4)
390: (8)
391: (8)
392: (4)
393: (4)
394: (4)
395: (8)
396: (8)
397: (4)
398: (4)
399: (0)
400: (4)
401: (4)
402: (4)
403: (4)
404: (4)
405: (4)
406: (4)
407: (0)
408: (4)
409: (4)
410: (4)
411: (4)
412: (4)
413: (4)
414: (4)
415: (0)
416: (4)
417: (4)
418: (4)
419: (4)
420: (4)
421: (4)
422: (4)
423: (0)
424: (4)
425: (4)

def test_deriv(Poly):
    d = Poly.domain + random((2,))*.25
    w = Poly.window + random((2,))*.25
    p1 = Poly([1, 2, 3], domain=d, window=w)
    p2 = p1.integ(2, k=[1, 2])
    p3 = p1.integ(1, k=[1])
    assert_almost_equal(p2.deriv(1).coef, p3.coef)
    assert_almost_equal(p2.deriv(2).coef, p1.coef)
    p1 = Poly([1, 2, 3])
    p2 = p1.integ(2, k=[1, 2])
    p3 = p1.integ(1, k=[1])
    assert_almost_equal(p2.deriv(1).coef, p3.coef)
    assert_almost_equal(p2.deriv(2).coef, p1.coef)

def test_linspace(Poly):
    d = Poly.domain + random((2,))*.25
    w = Poly.window + random((2,))*.25
    p = Poly([1, 2, 3], domain=d, window=w)
    xtgt = np.linspace(d[0], d[1], 20)
    ytgt = p(xtgt)
    xres, yres = p.linspace(20)
    assert_almost_equal(xres, xtgt)
    assert_almost_equal(yres, ytgt)
    xtgt = np.linspace(0, 2, 20)
    ytgt = p(xtgt)
    xres, yres = p.linspace(20, domain=[0, 2])
    assert_almost_equal(xres, xtgt)
    assert_almost_equal(yres, ytgt)

def test_pow(Poly):
    d = Poly.domain + random((2,))*.25
    w = Poly.window + random((2,))*.25
    tgt = Poly([1], domain=d, window=w)
    tst = Poly([1, 2, 3], domain=d, window=w)
    for i in range(5):
        assert_poly_almost_equal(tst**i, tgt)
        tgt = tgt * tst
    tgt = Poly([1])
    tst = Poly([1, 2, 3])
    for i in range(5):
        assert_poly_almost_equal(tst**i, tgt)
        tgt = tgt * tst
    assert_raises(ValueError, op.pow, tgt, 1.5)
    assert_raises(ValueError, op.pow, tgt, -1)

def test_call(Poly):
    P = Polynomial
    d = Poly.domain
    x = np.linspace(d[0], d[1], 11)
    p = Poly.cast(P([1, 2, 3]))
    tgt = 1 + x*(2 + 3*x)
    res = p(x)
    assert_almost_equal(res, tgt)

def test_cutdeg(Poly):
    p = Poly([1, 2, 3])
    assert_raises(ValueError, p.cutdeg, .5)
    assert_raises(ValueError, p.cutdeg, -1)
    assert_equal(len(p.cutdeg(3)), 3)
    assert_equal(len(p.cutdeg(2)), 3)
    assert_equal(len(p.cutdeg(1)), 2)
    assert_equal(len(p.cutdeg(0)), 1)

def test_truncate(Poly):
    p = Poly([1, 2, 3])
    assert_raises(ValueError, p.truncate, .5)
    assert_raises(ValueError, p.truncate, 0)
    assert_equal(len(p.truncate(4)), 3)
    assert_equal(len(p.truncate(3)), 3)
    assert_equal(len(p.truncate(2)), 2)
    assert_equal(len(p.truncate(1)), 1)

def test_trim(Poly):
    c = [1, 1e-6, 1e-12, 0]
    p = Poly(c)

```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

426: (4)             assert_equal(p.trim().coef, c[:3])
427: (4)             assert_equal(p.trim(1e-10).coef, c[:2])
428: (4)             assert_equal(p.trim(1e-5).coef, c[:1])
429: (0)          def test_mapparms(Poly):
430: (4)              d = Poly.domain
431: (4)              w = Poly.window
432: (4)              p = Poly([1], domain=d, window=w)
433: (4)              assert_almost_equal([0, 1], p.mapparms())
434: (4)              w = 2*d + 1
435: (4)              p = Poly([1], domain=d, window=w)
436: (4)              assert_almost_equal([1, 2], p.mapparms())
437: (0)          def test_ufunc_override(Poly):
438: (4)              p = Poly([1, 2, 3])
439: (4)              x = np.ones(3)
440: (4)              assert_raises(TypeError, np.add, p, x)
441: (4)              assert_raises(TypeError, np.add, x, p)
442: (0)      class TestInterpolate:
443: (4)          def f(self, x):
444: (8)              return x * (x - 1) * (x - 2)
445: (4)          def test.raises(self):
446: (8)              assert_raises(ValueError, Chebyshev.interpolate, self.f, -1)
447: (8)              assert_raises(TypeError, Chebyshev.interpolate, self.f, 10.)
448: (4)          def test_dimensions(self):
449: (8)              for deg in range(1, 5):
450: (12)                  assert_(Chebyshev.interpolate(self.f, deg).degree() == deg)
451: (4)          def test_approximation(self):
452: (8)              def powx(x, p):
453: (12)                  return x**p
454: (8)              x = np.linspace(0, 2, 10)
455: (8)              for deg in range(0, 10):
456: (12)                  for t in range(0, deg + 1):
457: (16)                      p = Chebyshev.interpolate(powx, deg, domain=[0, 2], args=(t,))
458: (16)                      assert_almost_equal(p(x), powx(x, t), decimal=11)
```

---

File 298 - test\_chebyshev.py:

```
1: (0)      """Tests for chebyshev module.
2: (0)      """
3: (0)      from functools import reduce
4: (0)      import numpy as np
5: (0)      import numpy.polynomial.chebyshev as cheb
6: (0)      from numpy.polynomial.polynomial import polyval
7: (0)      from numpy.testing import (
8: (4)          assert_almost_equal, assert_raises, assert_equal, assert_,
9: (4)      )
10: (0)     def trim(x):
11: (4)         return cheb.chebtrim(x, tol=1e-6)
12: (0)     T0 = [1]
13: (0)     T1 = [0, 1]
14: (0)     T2 = [-1, 0, 2]
15: (0)     T3 = [0, -3, 0, 4]
16: (0)     T4 = [1, 0, -8, 0, 8]
17: (0)     T5 = [0, 5, 0, -20, 0, 16]
18: (0)     T6 = [-1, 0, 18, 0, -48, 0, 32]
19: (0)     T7 = [0, -7, 0, 56, 0, -112, 0, 64]
20: (0)     T8 = [1, 0, -32, 0, 160, 0, -256, 0, 128]
21: (0)     T9 = [0, 9, 0, -120, 0, 432, 0, -576, 0, 256]
22: (0)     Tlist = [T0, T1, T2, T3, T4, T5, T6, T7, T8, T9]
23: (0)     class TestPrivate:
24: (4)         def test_cseries_to_zseries(self):
25: (8)             for i in range(5):
26: (12)                 inp = np.array([2] + [1]*i, np.double)
27: (12)                 tgt = np.array([.5]*i + [2] + [.5]*i, np.double)
28: (12)                 res = cheb._cseries_to_zseries(inp)
29: (12)                 assert_equal(res, tgt)
30: (4)         def test_zseries_to_cseries(self):
31: (8)             for i in range(5):
```

```

32: (12)             inp = np.array([.5]*i + [2] + [.5]*i, np.double)
33: (12)             tgt = np.array([2] + [1]*i, np.double)
34: (12)             res = cheb._zseries_to_cseries(inp)
35: (12)             assert_equal(res, tgt)
36: (0)              class TestConstants:
37: (4)                def test_chebdomain(self):
38: (8)                  assert_equal(cheb.chebdomain, [-1, 1])
39: (4)                def test_chebzero(self):
40: (8)                  assert_equal(cheb.chebzero, [0])
41: (4)                def test_chebone(self):
42: (8)                  assert_equal(cheb.chebone, [1])
43: (4)                def test_chebx(self):
44: (8)                  assert_equal(cheb.chebx, [0, 1])
45: (0)              class TestArithmetic:
46: (4)                def test_chebadd(self):
47: (8)                  for i in range(5):
48: (12)                    for j in range(5):
49: (16)                      msg = f"At i={i}, j={j}"
50: (16)                      tgt = np.zeros(max(i, j) + 1)
51: (16)                      tgt[i] += 1
52: (16)                      tgt[j] += 1
53: (16)                      res = cheb.chebadd([0]*i + [1], [0]*j + [1])
54: (16)                      assert_equal(trim(res), trim(tgt), err_msg=msg)
55: (4)                def test_chebsub(self):
56: (8)                  for i in range(5):
57: (12)                    for j in range(5):
58: (16)                      msg = f"At i={i}, j={j}"
59: (16)                      tgt = np.zeros(max(i, j) + 1)
60: (16)                      tgt[i] += 1
61: (16)                      tgt[j] -= 1
62: (16)                      res = cheb.chebsub([0]*i + [1], [0]*j + [1])
63: (16)                      assert_equal(trim(res), trim(tgt), err_msg=msg)
64: (4)                def test_chebmux(self):
65: (8)                  assert_equal(cheb.chebmux([0]), [0])
66: (8)                  assert_equal(cheb.chebmux([1]), [0, 1])
67: (8)                  for i in range(1, 5):
68: (12)                    ser = [0]*i + [1]
69: (12)                    tgt = [0]*(i - 1) + [.5, 0, .5]
70: (12)                    assert_equal(cheb.chebmux(ser), tgt)
71: (4)                def test_chebmul(self):
72: (8)                  for i in range(5):
73: (12)                    for j in range(5):
74: (16)                      msg = f"At i={i}, j={j}"
75: (16)                      tgt = np.zeros(i + j + 1)
76: (16)                      tgt[i + j] += .5
77: (16)                      tgt[abs(i - j)] += .5
78: (16)                      res = cheb.chebmul([0]*i + [1], [0]*j + [1])
79: (16)                      assert_equal(trim(res), trim(tgt), err_msg=msg)
80: (4)                def test_chebdiv(self):
81: (8)                  for i in range(5):
82: (12)                    for j in range(5):
83: (16)                      msg = f"At i={i}, j={j}"
84: (16)                      ci = [0]*i + [1]
85: (16)                      cj = [0]*j + [1]
86: (16)                      tgt = cheb.chebadd(ci, cj)
87: (16)                      quo, rem = cheb.chebdiv(tgt, ci)
88: (16)                      res = cheb.chebadd(cheb.chebmul(quo, ci), rem)
89: (16)                      assert_equal(trim(res), trim(tgt), err_msg=msg)
90: (4)                def test_chebpow(self):
91: (8)                  for i in range(5):
92: (12)                    for j in range(5):
93: (16)                      msg = f"At i={i}, j={j}"
94: (16)                      c = np.arange(i + 1)
95: (16)                      tgt = reduce(cheb.chebmul, [c]*j, np.array([1]))
96: (16)                      res = cheb.chebpow(c, j)
97: (16)                      assert_equal(trim(res), trim(tgt), err_msg=msg)
98: (0)              class TestEvaluation:
99: (4)                c1d = np.array([2.5, 2., 1.5])
100: (4)               c2d = np.einsum('i,j->ij', c1d, c1d)

```

```

101: (4)          c3d = np.einsum('i,j,k->ijk', c1d, c1d, c1d)
102: (4)          x = np.random.random((3, 5))*2 - 1
103: (4)          y = polyval(x, [1., 2., 3.])
104: (4)          def test_chebval(self):
105: (8)              assert_equal(cheb.chebval([], [1]).size, 0)
106: (8)              x = np.linspace(-1, 1)
107: (8)              y = [polyval(x, c) for c in Tlist]
108: (8)              for i in range(10):
109: (12)                  msg = f"At i={i}"
110: (12)                  tgt = y[i]
111: (12)                  res = cheb.chebval(x, [0]*i + [1])
112: (12)                  assert_almost_equal(res, tgt, err_msg=msg)
113: (8)              for i in range(3):
114: (12)                  dims = [2]*i
115: (12)                  x = np.zeros(dims)
116: (12)                  assert_equal(cheb.chebval(x, [1]).shape, dims)
117: (12)                  assert_equal(cheb.chebval(x, [1, 0]).shape, dims)
118: (12)                  assert_equal(cheb.chebval(x, [1, 0, 0]).shape, dims)
119: (4)          def test_chebval2d(self):
120: (8)              x1, x2, x3 = self.x
121: (8)              y1, y2, y3 = self.y
122: (8)              assert_raises(ValueError, cheb.chebval2d, x1, x2[:2], self.c2d)
123: (8)              tgt = y1*y2
124: (8)              res = cheb.chebval2d(x1, x2, self.c2d)
125: (8)              assert_almost_equal(res, tgt)
126: (8)              z = np.ones((2, 3))
127: (8)              res = cheb.chebval2d(z, z, self.c2d)
128: (8)              assert_(res.shape == (2, 3))
129: (4)          def test_chebval3d(self):
130: (8)              x1, x2, x3 = self.x
131: (8)              y1, y2, y3 = self.y
132: (8)              assert_raises(ValueError, cheb.chebval3d, x1, x2, x3[:2], self.c3d)
133: (8)              tgt = y1*y2*y3
134: (8)              res = cheb.chebval3d(x1, x2, x3, self.c3d)
135: (8)              assert_almost_equal(res, tgt)
136: (8)              z = np.ones((2, 3))
137: (8)              res = cheb.chebval3d(z, z, z, self.c3d)
138: (8)              assert_(res.shape == (2, 3))
139: (4)          def test_chebgrid2d(self):
140: (8)              x1, x2, x3 = self.x
141: (8)              y1, y2, y3 = self.y
142: (8)              tgt = np.einsum('i,j->ij', y1, y2)
143: (8)              res = cheb.chebgrid2d(x1, x2, self.c2d)
144: (8)              assert_almost_equal(res, tgt)
145: (8)              z = np.ones((2, 3))
146: (8)              res = cheb.chebgrid2d(z, z, self.c2d)
147: (8)              assert_(res.shape == (2, 3)**2)
148: (4)          def test_chebgrid3d(self):
149: (8)              x1, x2, x3 = self.x
150: (8)              y1, y2, y3 = self.y
151: (8)              tgt = np.einsum('i,j,k->ijk', y1, y2, y3)
152: (8)              res = cheb.chebgrid3d(x1, x2, x3, self.c3d)
153: (8)              assert_almost_equal(res, tgt)
154: (8)              z = np.ones((2, 3))
155: (8)              res = cheb.chebgrid3d(z, z, z, self.c3d)
156: (8)              assert_(res.shape == (2, 3)**3)
157: (0)          class TestIntegral:
158: (4)              def test_chebint(self):
159: (8)                  assert_raises(TypeError, cheb.chebint, [0], .5)
160: (8)                  assert_raises(ValueError, cheb.chebint, [0], -1)
161: (8)                  assert_raises(ValueError, cheb.chebint, [0], 1, [0, 0])
162: (8)                  assert_raises(ValueError, cheb.chebint, [0], lbnd=[0])
163: (8)                  assert_raises(ValueError, cheb.chebint, [0], scl=[0])
164: (8)                  assert_raises(TypeError, cheb.chebint, [0], axis=.5)
165: (8)                  for i in range(2, 5):
166: (12)                      k = [0]*(i - 2) + [1]
167: (12)                      res = cheb.chebint([0], m=i, k=k)
168: (12)                      assert_almost_equal(res, [0, 1])
169: (8)                  for i in range(5):

```

```

170: (12)           scl = i + 1
171: (12)           pol = [0]*i + [1]
172: (12)           tgt = [i] + [0]*i + [1/scl]
173: (12)           chebpol = cheb.poly2cheb(pol)
174: (12)           chebint = cheb.chebint(chebpol, m=1, k=[i])
175: (12)           res = cheb.cheb2poly(chebint)
176: (12)           assert_almost_equal(trim(res), trim(tgt))
177: (8)            for i in range(5):
178: (12)              scl = i + 1
179: (12)              pol = [0]*i + [1]
180: (12)              chebpol = cheb.poly2cheb(pol)
181: (12)              chebint = cheb.chebint(chebpol, m=1, k=[i], lbnd=-1)
182: (12)              assert_almost_equal(cheb.chebval(-1, chebint), i)
183: (8)            for i in range(5):
184: (12)              scl = i + 1
185: (12)              pol = [0]*i + [1]
186: (12)              tgt = [i] + [0]*i + [2/scl]
187: (12)              chebpol = cheb.poly2cheb(pol)
188: (12)              chebint = cheb.chebint(chebpol, m=1, k=[i], scl=2)
189: (12)              res = cheb.cheb2poly(chebint)
190: (12)              assert_almost_equal(trim(res), trim(tgt))
191: (8)            for i in range(5):
192: (12)              for j in range(2, 5):
193: (16)                  pol = [0]*i + [1]
194: (16)                  tgt = pol[:]
195: (16)                  for k in range(j):
196: (20)                      tgt = cheb.chebint(tgt, m=1)
197: (16)                      res = cheb.chebint(pol, m=j)
198: (16)                      assert_almost_equal(trim(res), trim(tgt))
199: (8)            for i in range(5):
200: (12)              for j in range(2, 5):
201: (16)                  pol = [0]*i + [1]
202: (16)                  tgt = pol[:]
203: (16)                  for k in range(j):
204: (20)                      tgt = cheb.chebint(tgt, m=1, k=[k])
205: (16)                      res = cheb.chebint(pol, m=j, k=list(range(j)))
206: (16)                      assert_almost_equal(trim(res), trim(tgt))
207: (8)            for i in range(5):
208: (12)              for j in range(2, 5):
209: (16)                  pol = [0]*i + [1]
210: (16)                  tgt = pol[:]
211: (16)                  for k in range(j):
212: (20)                      tgt = cheb.chebint(tgt, m=1, k=[k], lbnd=-1)
213: (16)                      res = cheb.chebint(pol, m=j, k=list(range(j)), lbnd=-1)
214: (16)                      assert_almost_equal(trim(res), trim(tgt))
215: (8)            for i in range(5):
216: (12)              for j in range(2, 5):
217: (16)                  pol = [0]*i + [1]
218: (16)                  tgt = pol[:]
219: (16)                  for k in range(j):
220: (20)                      tgt = cheb.chebint(tgt, m=1, k=[k], scl=2)
221: (16)                      res = cheb.chebint(pol, m=j, k=list(range(j)), scl=2)
222: (16)                      assert_almost_equal(trim(res), trim(tgt))
223: (4)             def test_chebint_axis(self):
224: (8)                 c2d = np.random.random((3, 4))
225: (8)                 tgt = np.vstack([cheb.chebint(c) for c in c2d.T]).T
226: (8)                 res = cheb.chebint(c2d, axis=0)
227: (8)                 assert_almost_equal(res, tgt)
228: (8)                 tgt = np.vstack([cheb.chebint(c) for c in c2d])
229: (8)                 res = cheb.chebint(c2d, axis=1)
230: (8)                 assert_almost_equal(res, tgt)
231: (8)                 tgt = np.vstack([cheb.chebint(c, k=3) for c in c2d])
232: (8)                 res = cheb.chebint(c2d, k=3, axis=1)
233: (8)                 assert_almost_equal(res, tgt)
234: (0)             class TestDerivative:
235: (4)                 def test_chebder(self):
236: (8)                     assert_raises(TypeError, cheb.chebder, [0], .5)
237: (8)                     assert_raises(ValueError, cheb.chebder, [0], -1)
238: (8)                     for i in range(5):

```

```

239: (12)             tgt = [0]*i + [1]
240: (12)             res = cheb.chebder(tgt, m=0)
241: (12)             assert_equal(trim(res), trim(tgt))
242: (8)              for i in range(5):
243: (12)                  for j in range(2, 5):
244: (16)                      tgt = [0]*i + [1]
245: (16)                      res = cheb.chebder(cheb.chebint(tgt, m=j), m=j)
246: (16)                      assert_almost_equal(trim(res), trim(tgt))
247: (8)              for i in range(5):
248: (12)                  for j in range(2, 5):
249: (16)                      tgt = [0]*i + [1]
250: (16)                      res = cheb.chebder(cheb.chebint(tgt, m=j, scl=2), m=j, scl=.5)
251: (16)                      assert_almost_equal(trim(res), trim(tgt))
252: (4)           def test_chebder_axis(self):
253: (8)               c2d = np.random.random((3, 4))
254: (8)               tgt = np.vstack([cheb.chebder(c) for c in c2d.T]).T
255: (8)               res = cheb.chebder(c2d, axis=0)
256: (8)               assert_almost_equal(res, tgt)
257: (8)               tgt = np.vstack([cheb.chebder(c) for c in c2d])
258: (8)               res = cheb.chebder(c2d, axis=1)
259: (8)               assert_almost_equal(res, tgt)
260: (0)    class TestVander:
261: (4)        x = np.random.random((3, 5))*2 - 1
262: (4)        def test_chebvander(self):
263: (8)            x = np.arange(3)
264: (8)            v = cheb.chebvander(x, 3)
265: (8)            assert_(v.shape == (3, 4))
266: (8)            for i in range(4):
267: (12)                coef = [0]*i + [1]
268: (12)                assert_almost_equal(v[..., i], cheb.chebval(x, coef))
269: (8)            x = np.array([[1, 2], [3, 4], [5, 6]])
270: (8)            v = cheb.chebvander(x, 3)
271: (8)            assert_(v.shape == (3, 2, 4))
272: (8)            for i in range(4):
273: (12)                coef = [0]*i + [1]
274: (12)                assert_almost_equal(v[..., i], cheb.chebval(x, coef))
275: (4)        def test_chebvander2d(self):
276: (8)            x1, x2, x3 = self.x
277: (8)            c = np.random.random((2, 3))
278: (8)            van = cheb.chebvander2d(x1, x2, [1, 2])
279: (8)            tgt = cheb.chebval2d(x1, x2, c)
280: (8)            res = np.dot(van, c.flat)
281: (8)            assert_almost_equal(res, tgt)
282: (8)            van = cheb.chebvander2d([x1], [x2], [1, 2])
283: (8)            assert_(van.shape == (1, 5, 6))
284: (4)        def test_chebvander3d(self):
285: (8)            x1, x2, x3 = self.x
286: (8)            c = np.random.random((2, 3, 4))
287: (8)            van = cheb.chebvander3d(x1, x2, x3, [1, 2, 3])
288: (8)            tgt = cheb.chebval3d(x1, x2, x3, c)
289: (8)            res = np.dot(van, c.flat)
290: (8)            assert_almost_equal(res, tgt)
291: (8)            van = cheb.chebvander3d([x1], [x2], [x3], [1, 2, 3])
292: (8)            assert_(van.shape == (1, 5, 24))
293: (0)    class TestFitting:
294: (4)        def test_chebfit(self):
295: (8)            def f(x):
296: (12)                return x*(x - 1)*(x - 2)
297: (8)            def f2(x):
298: (12)                return x**4 + x**2 + 1
299: (8)            assert_raises(ValueError, cheb.chebfit, [1], [1], -1)
300: (8)            assert_raises(TypeError, cheb.chebfit, [[1]], [1], 0)
301: (8)            assert_raises(TypeError, cheb.chebfit, [], [1], 0)
302: (8)            assert_raises(TypeError, cheb.chebfit, [1], [[[1]]], 0)
303: (8)            assert_raises(TypeError, cheb.chebfit, [1, 2], [1], 0)
304: (8)            assert_raises(TypeError, cheb.chebfit, [1], [1, 2], 0)
305: (8)            assert_raises(TypeError, cheb.chebfit, [1], [1], 0, w=[[1]]))
306: (8)            assert_raises(TypeError, cheb.chebfit, [1], [1], 0, w=[1, 1]))
307: (8)            assert_raises(ValueError, cheb.chebfit, [1], [1], [-1,])

```

```

308: (8) assert_raises(ValueError, cheb.chebfit, [1], [1], [2, -1, 6])
309: (8) assert_raises(TypeError, cheb.chebfit, [1], [1], [])
310: (8) x = np.linspace(0, 2)
311: (8) y = f(x)
312: (8) coef3 = cheb.chebfit(x, y, 3)
313: (8) assert_equal(len(coef3), 4)
314: (8) assert_almost_equal(cheb.chebval(x, coef3), y)
315: (8) coef3 = cheb.chebfit(x, y, [0, 1, 2, 3])
316: (8) assert_equal(len(coef3), 4)
317: (8) assert_almost_equal(cheb.chebval(x, coef3), y)
318: (8) coef4 = cheb.chebfit(x, y, 4)
319: (8) assert_equal(len(coef4), 5)
320: (8) assert_almost_equal(cheb.chebval(x, coef4), y)
321: (8) coef4 = cheb.chebfit(x, y, [0, 1, 2, 3, 4])
322: (8) assert_equal(len(coef4), 5)
323: (8) assert_almost_equal(cheb.chebval(x, coef4), y)
324: (8) coef4 = cheb.chebfit(x, y, [2, 3, 4, 1, 0])
325: (8) assert_equal(len(coef4), 5)
326: (8) assert_almost_equal(cheb.chebval(x, coef4), y)
327: (8) coef2d = cheb.chebfit(x, np.array([y, y]).T, 3)
328: (8) assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
329: (8) coef2d = cheb.chebfit(x, np.array([y, y]).T, [0, 1, 2, 3])
330: (8) assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
331: (8) w = np.zeros_like(x)
332: (8) yw = y.copy()
333: (8) w[1::2] = 1
334: (8) y[0::2] = 0
335: (8) wcoef3 = cheb.chebfit(x, yw, 3, w=w)
336: (8) assert_almost_equal(wcoef3, coef3)
337: (8) wcoef3 = cheb.chebfit(x, yw, [0, 1, 2, 3], w=w)
338: (8) assert_almost_equal(wcoef3, coef3)
339: (8) wcoef2d = cheb.chebfit(x, np.array([yw, yw]).T, 3, w=w)
340: (8) assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
341: (8) wcoef2d = cheb.chebfit(x, np.array([yw, yw]).T, [0, 1, 2, 3], w=w)
342: (8) assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
343: (8) x = [1, 1j, -1, -1j]
344: (8) assert_almost_equal(cheb.chebfit(x, x, 1), [0, 1])
345: (8) assert_almost_equal(cheb.chebfit(x, x, [0, 1]), [0, 1])
346: (8) x = np.linspace(-1, 1)
347: (8) y = f2(x)
348: (8) coef1 = cheb.chebfit(x, y, 4)
349: (8) assert_almost_equal(cheb.chebval(x, coef1), y)
350: (8) coef2 = cheb.chebfit(x, y, [0, 2, 4])
351: (8) assert_almost_equal(cheb.chebval(x, coef2), y)
352: (8) assert_almost_equal(coef1, coef2)
353: (0) class TestInterpolate:
354: (4)     def f(self, x):
355: (8)         return x * (x - 1) * (x - 2)
356: (4)     def test.raises(self):
357: (8)         assert_raises(ValueError, cheb.chebinterpolate, self.f, -1)
358: (8)         assert_raises(TypeError, cheb.chebinterpolate, self.f, 10.)
359: (4)     def test_dimensions(self):
360: (8)         for deg in range(1, 5):
361: (12)             assert_(cheb.chebinterpolate(self.f, deg).shape == (deg + 1,))
362: (4)     def test_approximation(self):
363: (8)         def powx(x, p):
364: (12)             return x**p
365: (8)         x = np.linspace(-1, 1, 10)
366: (8)         for deg in range(0, 10):
367: (12)             for p in range(0, deg + 1):
368: (16)                 c = cheb.chebinterpolate(powx, deg, (p,))
369: (16)                 assert_almost_equal(cheb.chebval(x, c), powx(x, p),
decimal=12)
370: (0)         class TestCompanion:
371: (4)             def test.raises(self):
372: (8)                 assert_raises(ValueError, cheb.chebcompanion, [])
373: (8)                 assert_raises(ValueError, cheb.chebcompanion, [1])
374: (4)             def test_dimensions(self):
375: (8)                 for i in range(1, 5):

```

```

376: (12)             coef = [0]*i + [1]
377: (12)             assert_(cheb.chebcompanion(coef).shape == (i, i))
378: (4)              def test_linear_root(self):
379: (8)                assert_(cheb.chebcompanion([1, 2])[0, 0] == -.5)
380: (0)              class TestGauss:
381: (4)                def test_100(self):
382: (8)                  x, w = cheb.chebgauss(100)
383: (8)                  v = cheb.chebvander(x, 99)
384: (8)                  vv = np.dot(v.T * w, v)
385: (8)                  vd = 1/np.sqrt(vv.diagonal())
386: (8)                  vv = vd[:, None] * vv * vd
387: (8)                  assert_almost_equal(vv, np.eye(100))
388: (8)                  tgt = np.pi
389: (8)                  assert_almost_equal(w.sum(), tgt)
390: (0)              class TestMisc:
391: (4)                def test_chebfromroots(self):
392: (8)                  res = cheb.chebfromroots([])
393: (8)                  assert_almost_equal(trim(res), [1])
394: (8)                  for i in range(1, 5):
395: (12)                    roots = np.cos(np.linspace(-np.pi, 0, 2*i + 1)[1::2])
396: (12)                    tgt = [0]*i + [1]
397: (12)                    res = cheb.chebfromroots(roots)*2***(i-1)
398: (12)                    assert_almost_equal(trim(res), trim(tgt))
399: (4)              def test_chebroots(self):
400: (8)                assert_almost_equal(cheb.chebroots([1]), [])
401: (8)                assert_almost_equal(cheb.chebroots([1, 2]), [-.5])
402: (8)                for i in range(2, 5):
403: (12)                  tgt = np.linspace(-1, 1, i)
404: (12)                  res = cheb.chebroots(cheb.chebfromroots(tgt))
405: (12)                  assert_almost_equal(trim(res), trim(tgt))
406: (4)              def test_chebtrim(self):
407: (8)                coef = [2, -1, 1, 0]
408: (8)                assert_raises(ValueError, cheb.chebtrim, coef, -1)
409: (8)                assert_equal(cheb.chebtrim(coef), coef[:-1])
410: (8)                assert_equal(cheb.chebtrim(coef, 1), coef[:-3])
411: (8)                assert_equal(cheb.chebtrim(coef, 2), [0])
412: (4)              def test_chebline(self):
413: (8)                assert_equal(cheb.chebline(3, 4), [3, 4])
414: (4)              def test_cheb2poly(self):
415: (8)                for i in range(10):
416: (12)                  assert_almost_equal(cheb.cheb2poly([0]*i + [1]), Tlist[i])
417: (4)              def test_poly2cheb(self):
418: (8)                for i in range(10):
419: (12)                  assert_almost_equal(cheb.poly2cheb(Tlist[i]), [0]*i + [1])
420: (4)              def test_weight(self):
421: (8)                x = np.linspace(-1, 1, 11)[1:-1]
422: (8)                tgt = 1./((np.sqrt(1 + x) * np.sqrt(1 - x)))
423: (8)                res = cheb.chebweight(x)
424: (8)                assert_almost_equal(res, tgt)
425: (4)              def test_chebpts1(self):
426: (8)                assert_raises(ValueError, cheb.chebpts1, 1.5)
427: (8)                assert_raises(ValueError, cheb.chebpts1, 0)
428: (8)                tgt = [0]
429: (8)                assert_almost_equal(cheb.chebpts1(1), tgt)
430: (8)                tgt = [-0.70710678118654746, 0.70710678118654746]
431: (8)                assert_almost_equal(cheb.chebpts1(2), tgt)
432: (8)                tgt = [-0.86602540378443871, 0, 0.86602540378443871]
433: (8)                assert_almost_equal(cheb.chebpts1(3), tgt)
434: (8)                tgt = [-0.9238795325, -0.3826834323, 0.3826834323, 0.9238795325]
435: (8)                assert_almost_equal(cheb.chebpts1(4), tgt)
436: (4)              def test_chebpts2(self):
437: (8)                assert_raises(ValueError, cheb.chebpts2, 1.5)
438: (8)                assert_raises(ValueError, cheb.chebpts2, 1)
439: (8)                tgt = [-1, 1]
440: (8)                assert_almost_equal(cheb.chebpts2(2), tgt)
441: (8)                tgt = [-1, 0, 1]
442: (8)                assert_almost_equal(cheb.chebpts2(3), tgt)
443: (8)                tgt = [-1, -0.5, .5, 1]
444: (8)                assert_almost_equal(cheb.chebpts2(4), tgt)

```

```
445: (8)             tgt = [-1.0, -0.707106781187, 0, 0.707106781187, 1.0]
446: (8)             assert_almost_equal(cheb.chebpts2(5), tgt)
```

-----  
File 299 - test\_hermite.py:

```
1: (0)             """Tests for hermite module.
2: (0)             """
3: (0)             from functools import reduce
4: (0)             import numpy as np
5: (0)             import numpy.polynomial.hermite as herm
6: (0)             from numpy.polynomial.polynomial import polyval
7: (0)             from numpy.testing import (
8: (4)                 assert_almost_equal, assert_raises, assert_equal, assert_,
9: (4)                 )
10: (0)            H0 = np.array([1])
11: (0)            H1 = np.array([0, 2])
12: (0)            H2 = np.array([-2, 0, 4])
13: (0)            H3 = np.array([0, -12, 0, 8])
14: (0)            H4 = np.array([12, 0, -48, 0, 16])
15: (0)            H5 = np.array([0, 120, 0, -160, 0, 32])
16: (0)            H6 = np.array([-120, 0, 720, 0, -480, 0, 64])
17: (0)            H7 = np.array([0, -1680, 0, 3360, 0, -1344, 0, 128])
18: (0)            H8 = np.array([1680, 0, -13440, 0, 13440, 0, -3584, 0, 256])
19: (0)            H9 = np.array([0, 30240, 0, -80640, 0, 48384, 0, -9216, 0, 512])
20: (0)            Hlist = [H0, H1, H2, H3, H4, H5, H6, H7, H8, H9]
21: (0)            def trim(x):
22: (4)                return herm.hermtrim(x, tol=1e-6)
23: (0)            class TestConstants:
24: (4)                def test_hermdomain(self):
25: (8)                    assert_equal(herm.hermdomain, [-1, 1])
26: (4)                def test_hermzero(self):
27: (8)                    assert_equal(herm.hermzero, [0])
28: (4)                def test_hermone(self):
29: (8)                    assert_equal(herm.hermone, [1])
30: (4)                def test_hermx(self):
31: (8)                    assert_equal(herm.hermx, [0, .5])
32: (0)            class TestArithmetic:
33: (4)                x = np.linspace(-3, 3, 100)
34: (4)                def test_hermadd(self):
35: (8)                    for i in range(5):
36: (12)                        for j in range(5):
37: (16)                            msg = f"At i={i}, j={j}"
38: (16)                            tgt = np.zeros(max(i, j) + 1)
39: (16)                            tgt[i] += 1
40: (16)                            tgt[j] += 1
41: (16)                            res = herm.hermadd([0]*i + [1], [0]*j + [1])
42: (16)                            assert_equal(trim(res), trim(tgt), err_msg=msg)
43: (4)                def test_hermsub(self):
44: (8)                    for i in range(5):
45: (12)                        for j in range(5):
46: (16)                            msg = f"At i={i}, j={j}"
47: (16)                            tgt = np.zeros(max(i, j) + 1)
48: (16)                            tgt[i] += 1
49: (16)                            tgt[j] -= 1
50: (16)                            res = herm.hermsub([0]*i + [1], [0]*j + [1])
51: (16)                            assert_equal(trim(res), trim(tgt), err_msg=msg)
52: (4)                def test_hermmulx(self):
53: (8)                    assert_equal(herm.hermmulx([0]), [0])
54: (8)                    assert_equal(herm.hermmulx([1]), [0, .5])
55: (8)                    for i in range(1, 5):
56: (12)                        ser = [0]*i + [1]
57: (12)                        tgt = [0]^(i - 1) + [i, 0, .5]
58: (12)                        assert_equal(herm.hermmulx(ser), tgt)
59: (4)                def test_hermmul(self):
60: (8)                    for i in range(5):
61: (12)                        pol1 = [0]*i + [1]
62: (12)                        val1 = herm.hermval(self.x, pol1)
```

```

63: (12)             for j in range(5):
64: (16)                 msg = f"At i={i}, j={j}"
65: (16)                 pol2 = [0]*j + [1]
66: (16)                 val2 = herm.hermval(self.x, pol2)
67: (16)                 pol3 = herm.hermmul(pol1, pol2)
68: (16)                 val3 = herm.hermval(self.x, pol3)
69: (16)                 assert_(len(pol3) == i + j + 1, msg)
70: (16)                 assert_almost_equal(val3, val1*val2, err_msg=msg)
71: (4)             def test_hermdiv(self):
72: (8)                 for i in range(5):
73: (12)                     for j in range(5):
74: (16)                         msg = f"At i={i}, j={j}"
75: (16)                         ci = [0]*i + [1]
76: (16)                         cj = [0]*j + [1]
77: (16)                         tgt = herm.hermadd(ci, cj)
78: (16)                         quo, rem = herm.hermdiv(tgt, ci)
79: (16)                         res = herm.hermadd(herm.hermmul(quo, ci), rem)
80: (16)                         assert_equal(trim(res), trim(tgt), err_msg=msg)
81: (4)             def test_hermpow(self):
82: (8)                 for i in range(5):
83: (12)                     for j in range(5):
84: (16)                         msg = f"At i={i}, j={j}"
85: (16)                         c = np.arange(i + 1)
86: (16)                         tgt = reduce(herm.hermmul, [c]*j, np.array([1]))
87: (16)                         res = herm.hermpow(c, j)
88: (16)                         assert_equal(trim(res), trim(tgt), err_msg=msg)
89: (0)         class TestEvaluation:
90: (4)             c1d = np.array([2.5, 1., .75])
91: (4)             c2d = np.einsum('i,j->ij', c1d, c1d)
92: (4)             c3d = np.einsum('i,j,k->ijk', c1d, c1d, c1d)
93: (4)             x = np.random.random((3, 5))*2 - 1
94: (4)             y = polyval(x, [1., 2., 3.])
95: (4)             def test_hermval(self):
96: (8)                 assert_equal(herm.hermval([], [1]).size, 0)
97: (8)                 x = np.linspace(-1, 1)
98: (8)                 y = [polyval(x, c) for c in Hlist]
99: (8)                 for i in range(10):
100: (12)                     msg = f"At i={i}"
101: (12)                     tgt = y[i]
102: (12)                     res = herm.hermval(x, [0]*i + [1])
103: (12)                     assert_almost_equal(res, tgt, err_msg=msg)
104: (8)                 for i in range(3):
105: (12)                     dims = [2]*i
106: (12)                     x = np.zeros(dims)
107: (12)                     assert_equal(herm.hermval(x, [1]).shape, dims)
108: (12)                     assert_equal(herm.hermval(x, [1, 0]).shape, dims)
109: (12)                     assert_equal(herm.hermval(x, [1, 0, 0]).shape, dims)
110: (4)             def test_hermval2d(self):
111: (8)                 x1, x2, x3 = self.x
112: (8)                 y1, y2, y3 = self.y
113: (8)                 assert_raises(ValueError, herm.hermval2d, x1, x2[:2], self.c2d)
114: (8)                 tgt = y1*y2
115: (8)                 res = herm.hermval2d(x1, x2, self.c2d)
116: (8)                 assert_almost_equal(res, tgt)
117: (8)                 z = np.ones((2, 3))
118: (8)                 res = herm.hermval2d(z, z, self.c2d)
119: (8)                 assert_(res.shape == (2, 3))
120: (4)             def test_hermval3d(self):
121: (8)                 x1, x2, x3 = self.x
122: (8)                 y1, y2, y3 = self.y
123: (8)                 assert_raises(ValueError, herm.hermval3d, x1, x2, x3[:2], self.c3d)
124: (8)                 tgt = y1*y2*y3
125: (8)                 res = herm.hermval3d(x1, x2, x3, self.c3d)
126: (8)                 assert_almost_equal(res, tgt)
127: (8)                 z = np.ones((2, 3))
128: (8)                 res = herm.hermval3d(z, z, z, self.c3d)
129: (8)                 assert_(res.shape == (2, 3))
130: (4)             def test_hermgrid2d(self):
131: (8)                 x1, x2, x3 = self.x

```

```

132: (8)             y1, y2, y3 = self.y
133: (8)             tgt = np.einsum('i,j->ij', y1, y2)
134: (8)             res = herm.hermgrid2d(x1, x2, self.c2d)
135: (8)             assert_almost_equal(res, tgt)
136: (8)             z = np.ones((2, 3))
137: (8)             res = herm.hermgrid2d(z, z, self.c2d)
138: (8)             assert_(res.shape == (2, 3)*2)
139: (4)             def test_hermgrid3d(self):
140: (8)                 x1, x2, x3 = self.x
141: (8)                 y1, y2, y3 = self.y
142: (8)                 tgt = np.einsum('i,j,k->ijk', y1, y2, y3)
143: (8)                 res = herm.hermgrid3d(x1, x2, x3, self.c3d)
144: (8)                 assert_almost_equal(res, tgt)
145: (8)                 z = np.ones((2, 3))
146: (8)                 res = herm.hermgrid3d(z, z, z, self.c3d)
147: (8)                 assert_(res.shape == (2, 3)*3)
148: (0)             class TestIntegral:
149: (4)             def test_hermint(self):
150: (8)                 assert_raises(TypeError, herm.hermint, [0], .5)
151: (8)                 assert_raises(ValueError, herm.hermint, [0], -1)
152: (8)                 assert_raises(ValueError, herm.hermint, [0], 1, [0, 0])
153: (8)                 assert_raises(ValueError, herm.hermint, [0], lbnd=[0])
154: (8)                 assert_raises(ValueError, herm.hermint, [0], scl=[0])
155: (8)                 assert_raises(TypeError, herm.hermint, [0], axis=.5)
156: (8)                 for i in range(2, 5):
157: (12)                     k = [0]*(i - 2) + [1]
158: (12)                     res = herm.hermint([0], m=i, k=k)
159: (12)                     assert_almost_equal(res, [0, .5])
160: (8)                 for i in range(5):
161: (12)                     scl = i + 1
162: (12)                     pol = [0]*i + [1]
163: (12)                     tgt = [i] + [0]*i + [1/scl]
164: (12)                     hermpol = herm.poly2herm(pol)
165: (12)                     hermint = herm.hermint(hermpol, m=1, k=[i])
166: (12)                     res = herm.herm2poly(hermint)
167: (12)                     assert_almost_equal(trim(res), trim(tgt))
168: (8)                 for i in range(5):
169: (12)                     scl = i + 1
170: (12)                     pol = [0]*i + [1]
171: (12)                     hermpol = herm.poly2herm(pol)
172: (12)                     hermint = herm.hermint(hermpol, m=1, k=[i], lbnd=-1)
173: (12)                     assert_almost_equal(herm.hermval(-1, hermint), i)
174: (8)                 for i in range(5):
175: (12)                     scl = i + 1
176: (12)                     pol = [0]*i + [1]
177: (12)                     tgt = [i] + [0]*i + [2/scl]
178: (12)                     hermpol = herm.poly2herm(pol)
179: (12)                     hermint = herm.hermint(hermpol, m=1, k=[i], scl=2)
180: (12)                     res = herm.herm2poly(hermint)
181: (12)                     assert_almost_equal(trim(res), trim(tgt))
182: (8)                 for i in range(5):
183: (12)                     for j in range(2, 5):
184: (16)                         pol = [0]*i + [1]
185: (16)                         tgt = pol[:]
186: (16)                         for k in range(j):
187: (20)                             tgt = herm.hermint(tgt, m=1)
188: (16)                         res = herm.hermint(pol, m=j)
189: (16)                         assert_almost_equal(trim(res), trim(tgt))
190: (8)                 for i in range(5):
191: (12)                     for j in range(2, 5):
192: (16)                         pol = [0]*i + [1]
193: (16)                         tgt = pol[:]
194: (16)                         for k in range(j):
195: (20)                             tgt = herm.hermint(tgt, m=1, k=[k])
196: (16)                         res = herm.hermint(pol, m=j, k=list(range(j)))
197: (16)                         assert_almost_equal(trim(res), trim(tgt))
198: (8)                 for i in range(5):
199: (12)                     for j in range(2, 5):
200: (16)                         pol = [0]*i + [1]

```

```

201: (16)                      tgt = pol[:]
202: (16)                      for k in range(j):
203: (20)                          tgt = herm.hermint(tgt, m=1, k=[k], lbnd=-1)
204: (16)                          res = herm.hermint(pol, m=j, k=list(range(j)), lbnd=-1)
205: (16)                          assert_almost_equal(trim(res), trim(tgt))
206: (8)                           for i in range(5):
207: (12)                               for j in range(2, 5):
208: (16)                                   pol = [0]*i + [1]
209: (16)                                   tgt = pol[:]
210: (16)                                   for k in range(j):
211: (20)                                       tgt = herm.hermint(tgt, m=1, k=[k], scl=2)
212: (16)                                       res = herm.hermint(pol, m=j, k=list(range(j)), scl=2)
213: (16)                                       assert_almost_equal(trim(res), trim(tgt))
214: (4)    def test_hermint_axis(self):
215: (8)        c2d = np.random.random((3, 4))
216: (8)        tgt = np.vstack([herm.hermint(c) for c in c2d.T]).T
217: (8)        res = herm.hermint(c2d, axis=0)
218: (8)        assert_almost_equal(res, tgt)
219: (8)        tgt = np.vstack([herm.hermint(c) for c in c2d])
220: (8)        res = herm.hermint(c2d, axis=1)
221: (8)        assert_almost_equal(res, tgt)
222: (8)        tgt = np.vstack([herm.hermint(c, k=3) for c in c2d])
223: (8)        res = herm.hermint(c2d, k=3, axis=1)
224: (8)        assert_almost_equal(res, tgt)
225: (0)    class TestDerivative:
226: (4)        def test_hermder(self):
227: (8)            assert_raises(TypeError, herm.hermder, [0], .5)
228: (8)            assert_raises(ValueError, herm.hermder, [0], -1)
229: (8)            for i in range(5):
230: (12)                tgt = [0]*i + [1]
231: (12)                res = herm.hermder(tgt, m=0)
232: (12)                assert_equal(trim(res), trim(tgt))
233: (8)                for i in range(5):
234: (12)                    for j in range(2, 5):
235: (16)                        tgt = [0]*i + [1]
236: (16)                        res = herm.hermder(herm.hermint(tgt, m=j), m=j)
237: (16)                        assert_almost_equal(trim(res), trim(tgt))
238: (8)                for i in range(5):
239: (12)                    for j in range(2, 5):
240: (16)                        tgt = [0]*i + [1]
241: (16)                        res = herm.hermder(herm.hermint(tgt, m=j, scl=2), m=j, scl=.5)
242: (16)                        assert_almost_equal(trim(res), trim(tgt))
243: (4)        def test_hermder_axis(self):
244: (8)            c2d = np.random.random((3, 4))
245: (8)            tgt = np.vstack([herm.hermder(c) for c in c2d.T]).T
246: (8)            res = herm.hermder(c2d, axis=0)
247: (8)            assert_almost_equal(res, tgt)
248: (8)            tgt = np.vstack([herm.hermder(c) for c in c2d])
249: (8)            res = herm.hermder(c2d, axis=1)
250: (8)            assert_almost_equal(res, tgt)
251: (0)    class TestVander:
252: (4)        x = np.random.random((3, 5))*2 - 1
253: (4)        def test_hermvander(self):
254: (8)            x = np.arange(3)
255: (8)            v = herm.hermvander(x, 3)
256: (8)            assert_(v.shape == (3, 4))
257: (8)            for i in range(4):
258: (12)                coef = [0]*i + [1]
259: (12)                assert_almost_equal(v[..., i], herm.hermval(x, coef))
260: (8)            x = np.array([[1, 2], [3, 4], [5, 6]])
261: (8)            v = herm.hermvander(x, 3)
262: (8)            assert_(v.shape == (3, 2, 4))
263: (8)            for i in range(4):
264: (12)                coef = [0]*i + [1]
265: (12)                assert_almost_equal(v[..., i], herm.hermval(x, coef))
266: (4)        def test_hermvander2d(self):
267: (8)            x1, x2, x3 = self.x
268: (8)            c = np.random.random((2, 3))
269: (8)            van = herm.hermvander2d(x1, x2, [1, 2])

```

```

270: (8)             tgt = herm.hermval2d(x1, x2, c)
271: (8)             res = np.dot(van, c.flat)
272: (8)             assert_almost_equal(res, tgt)
273: (8)             van = herm.hermvander2d([x1], [x2], [1, 2])
274: (8)             assert_(van.shape == (1, 5, 6))
275: (4)             def test_hermvander3d(self):
276: (8)                 x1, x2, x3 = self.x
277: (8)                 c = np.random.random((2, 3, 4))
278: (8)                 van = herm.hermvander3d(x1, x2, x3, [1, 2, 3])
279: (8)                 tgt = herm.hermval3d(x1, x2, x3, c)
280: (8)                 res = np.dot(van, c.flat)
281: (8)                 assert_almost_equal(res, tgt)
282: (8)                 van = herm.hermvander3d([x1], [x2], [x3], [1, 2, 3])
283: (8)                 assert_(van.shape == (1, 5, 24))
284: (0)             class TestFitting:
285: (4)                 def test_hermfit(self):
286: (8)                     def f(x):
287: (12)                         return x*(x - 1)*(x - 2)
288: (8)                     def f2(x):
289: (12)                         return x**4 + x**2 + 1
290: (8)                     assert_raises(ValueError, herm.hermfit, [1], [1], -1)
291: (8)                     assert_raises(TypeError, herm.hermfit, [[1]], [1], 0)
292: (8)                     assert_raises(TypeError, herm.hermfit, [], [1], 0)
293: (8)                     assert_raises(TypeError, herm.hermfit, [1], [[[1]]], 0)
294: (8)                     assert_raises(TypeError, herm.hermfit, [1, 2], [1], 0)
295: (8)                     assert_raises(TypeError, herm.hermfit, [1], [1, 2], 0)
296: (8)                     assert_raises(TypeError, herm.hermfit, [1], [1], 0, w=[[1]])
297: (8)                     assert_raises(TypeError, herm.hermfit, [1], [1], 0, w=[1, 1])
298: (8)                     assert_raises(ValueError, herm.hermfit, [1], [1], [-1,])
299: (8)                     assert_raises(ValueError, herm.hermfit, [1], [1], [2, -1, 6])
300: (8)                     assert_raises(TypeError, herm.hermfit, [1], [1], [])
301: (8)                     x = np.linspace(0, 2)
302: (8)                     y = f(x)
303: (8)                     coef3 = herm.hermfit(x, y, 3)
304: (8)                     assert_equal(len(coef3), 4)
305: (8)                     assert_almost_equal(herm.hermval(x, coef3), y)
306: (8)                     coef3 = herm.hermfit(x, y, [0, 1, 2, 3])
307: (8)                     assert_equal(len(coef3), 4)
308: (8)                     assert_almost_equal(herm.hermval(x, coef3), y)
309: (8)                     coef4 = herm.hermfit(x, y, 4)
310: (8)                     assert_equal(len(coef4), 5)
311: (8)                     assert_almost_equal(herm.hermval(x, coef4), y)
312: (8)                     coef4 = herm.hermfit(x, y, [0, 1, 2, 3, 4])
313: (8)                     assert_equal(len(coef4), 5)
314: (8)                     assert_almost_equal(herm.hermval(x, coef4), y)
315: (8)                     coef4 = herm.hermfit(x, y, [2, 3, 4, 1, 0])
316: (8)                     assert_equal(len(coef4), 5)
317: (8)                     assert_almost_equal(herm.hermval(x, coef4), y)
318: (8)                     coef2d = herm.hermfit(x, np.array([y, y]).T, 3)
319: (8)                     assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
320: (8)                     coef2d = herm.hermfit(x, np.array([y, y]).T, [0, 1, 2, 3])
321: (8)                     assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
322: (8)                     w = np.zeros_like(x)
323: (8)                     yw = y.copy()
324: (8)                     w[1::2] = 1
325: (8)                     y[0::2] = 0
326: (8)                     wcoef3 = herm.hermfit(x, yw, 3, w=w)
327: (8)                     assert_almost_equal(wcoef3, coef3)
328: (8)                     wcoef3 = herm.hermfit(x, yw, [0, 1, 2, 3], w=w)
329: (8)                     assert_almost_equal(wcoef3, coef3)
330: (8)                     wcoef2d = herm.hermfit(x, np.array([yw, yw]).T, 3, w=w)
331: (8)                     assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
332: (8)                     wcoef2d = herm.hermfit(x, np.array([yw, yw]).T, [0, 1, 2, 3], w=w)
333: (8)                     assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
334: (8)                     x = [1, 1j, -1, -1j]
335: (8)                     assert_almost_equal(herm.hermfit(x, x, 1), [0, .5])
336: (8)                     assert_almost_equal(herm.hermfit(x, x, [0, 1]), [0, .5])
337: (8)                     x = np.linspace(-1, 1)
338: (8)                     y = f2(x)

```

```

339: (8)             coef1 = herm.hermfit(x, y, 4)
340: (8)             assert_almost_equal(herm.hermval(x, coef1), y)
341: (8)             coef2 = herm.hermfit(x, y, [0, 2, 4])
342: (8)             assert_almost_equal(herm.hermval(x, coef2), y)
343: (8)             assert_almost_equal(coef1, coef2)
344: (0)             class TestCompanion:
345: (4)                 def test_raises(self):
346: (8)                     assert_raises(ValueError, herm.hermcompanion, [])
347: (8)                     assert_raises(ValueError, herm.hermcompanion, [1])
348: (4)                 def test_dimensions(self):
349: (8)                     for i in range(1, 5):
350: (12)                         coef = [0]*i + [1]
351: (12)                         assert_(herm.hermcompanion(coef).shape == (i, i))
352: (4)                 def test_linear_root(self):
353: (8)                     assert_(herm.hermcompanion([1, 2])[0, 0] == -.25)
354: (0)             class TestGauss:
355: (4)                 def test_100(self):
356: (8)                     x, w = herm.hermgauss(100)
357: (8)                     v = herm.hermvander(x, 99)
358: (8)                     vv = np.dot(v.T * w, v)
359: (8)                     vd = 1/np.sqrt(vv.diagonal())
360: (8)                     vv = vd[:, None] * vv * vd
361: (8)                     assert_almost_equal(vv, np.eye(100))
362: (8)                     tgt = np.sqrt(np.pi)
363: (8)                     assert_almost_equal(w.sum(), tgt)
364: (0)             class TestMisc:
365: (4)                 def test_hermfromroots(self):
366: (8)                     res = herm.hermfromroots([])
367: (8)                     assert_almost_equal(trim(res), [1])
368: (8)                     for i in range(1, 5):
369: (12)                         roots = np.cos(np.linspace(-np.pi, 0, 2*i + 1)[1::2])
370: (12)                         pol = herm.hermfromroots(roots)
371: (12)                         res = herm.hermval(roots, pol)
372: (12)                         tgt = 0
373: (12)                         assert_(len(pol) == i + 1)
374: (12)                         assert_almost_equal(herm.herm2poly(pol)[-1], 1)
375: (12)                         assert_almost_equal(res, tgt)
376: (4)                 def test_hermroots(self):
377: (8)                     assert_almost_equal(herm.hermroots([1]), [])
378: (8)                     assert_almost_equal(herm.hermroots([1, 1]), [-.5])
379: (8)                     for i in range(2, 5):
380: (12)                         tgt = np.linspace(-1, 1, i)
381: (12)                         res = herm.hermroots(herm.hermfromroots(tgt))
382: (12)                         assert_almost_equal(trim(res), trim(tgt))
383: (4)                 def test_hermtrim(self):
384: (8)                     coef = [2, -1, 1, 0]
385: (8)                     assert_raises(ValueError, herm.hermtrim, coef, -1)
386: (8)                     assert_equal(herm.hermtrim(coef), coef[:-1])
387: (8)                     assert_equal(herm.hermtrim(coef, 1), coef[:-3])
388: (8)                     assert_equal(herm.hermtrim(coef, 2), [0])
389: (4)                 def test_hermline(self):
390: (8)                     assert_equal(herm.hermline(3, 4), [3, 2])
391: (4)                 def test_herm2poly(self):
392: (8)                     for i in range(10):
393: (12)                         assert_almost_equal(herm.herm2poly([0]*i + [1]), Hlist[i])
394: (4)                 def test_poly2herm(self):
395: (8)                     for i in range(10):
396: (12)                         assert_almost_equal(herm.poly2herm(Hlist[i]), [0]*i + [1])
397: (4)                 def test_weight(self):
398: (8)                     x = np.linspace(-5, 5, 11)
399: (8)                     tgt = np.exp(-x**2)
400: (8)                     res = herm.hermweight(x)
401: (8)                     assert_almost_equal(res, tgt)

```

-----  
File 300 - test\_hermite\_e.py:

```
1: (0)             """Tests for hermite_e module.
```

```

2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (0)
8: (4)
9: (4)
10: (0)
11: (0)
12: (0)
13: (0)
14: (0)
15: (0)
16: (0)
17: (0)
18: (0)
19: (0)
20: (0)
21: (0)
22: (4)
23: (0)
24: (4)
25: (8)
26: (4)
27: (8)
28: (4)
29: (8)
30: (4)
31: (8)
32: (0)
33: (4)
34: (4)
35: (8)
36: (12)
37: (16)
38: (16)
39: (16)
40: (16)
41: (16)
42: (16)
43: (4)
44: (8)
45: (12)
46: (16)
47: (16)
48: (16)
49: (16)
50: (16)
51: (16)
52: (4)
53: (8)
54: (8)
55: (8)
56: (12)
57: (12)
58: (12)
59: (4)
60: (8)
61: (12)
62: (12)
63: (12)
64: (16)
65: (16)
66: (16)
67: (16)
68: (16)
69: (16)
70: (16)

    """
    from functools import reduce
    import numpy as np
    import numpy.polynomial.hermite_e as herme
    from numpy.polynomial.polynomial import polyval
    from numpy.testing import (
        assert_almost_equal, assert_raises, assert_equal, assert_,
    )
    )
    He0 = np.array([1])
    He1 = np.array([0, 1])
    He2 = np.array([-1, 0, 1])
    He3 = np.array([0, -3, 0, 1])
    He4 = np.array([3, 0, -6, 0, 1])
    He5 = np.array([0, 15, 0, -10, 0, 1])
    He6 = np.array([-15, 0, 45, 0, -15, 0, 1])
    He7 = np.array([0, -105, 0, 105, 0, -21, 0, 1])
    He8 = np.array([105, 0, -420, 0, 210, 0, -28, 0, 1])
    He9 = np.array([0, 945, 0, -1260, 0, 378, 0, -36, 0, 1])
    Helist = [He0, He1, He2, He3, He4, He5, He6, He7, He8, He9]
    def trim(x):
        return herme.hermetrim(x, tol=1e-6)
    class TestConstants:
        def test_hermedomain(self):
            assert_equal(herme.hermedomain, [-1, 1])
        def test_hermezero(self):
            assert_equal(herme.hermezero, [0])
        def test_hermeone(self):
            assert_equal(herme.hermeone, [1])
        def test_hermex(self):
            assert_equal(herme.hermex, [0, 1])
    class TestArithmetic:
        x = np.linspace(-3, 3, 100)
        def test_hermeadd(self):
            for i in range(5):
                for j in range(5):
                    msg = f"At i={i}, j={j}"
                    tgt = np.zeros(max(i, j) + 1)
                    tgt[i] += 1
                    tgt[j] += 1
                    res = herme.hermeadd([0]*i + [1], [0]*j + [1])
                    assert_equal(trim(res), trim(tgt), err_msg=msg)
            def test_hermesub(self):
                for i in range(5):
                    for j in range(5):
                        msg = f"At i={i}, j={j}"
                        tgt = np.zeros(max(i, j) + 1)
                        tgt[i] += 1
                        tgt[j] -= 1
                        res = herme.hermesub([0]*i + [1], [0]*j + [1])
                        assert_equal(trim(res), trim(tgt), err_msg=msg)
            def test_hermemulx(self):
                assert_equal(herme.hermemulx([0]), [0])
                assert_equal(herme.hermemulx([1]), [0, 1])
                for i in range(1, 5):
                    ser = [0]*i + [1]
                    tgt = [0]^(i - 1) + [i, 0, 1]
                    assert_equal(herme.hermemulx(ser), tgt)
            def test_hermemul(self):
                for i in range(5):
                    pol1 = [0]*i + [1]
                    val1 = herme.hermeval(self.x, pol1)
                    for j in range(5):
                        msg = f"At i={i}, j={j}"
                        pol2 = [0]*j + [1]
                        val2 = herme.hermeval(self.x, pol2)
                        pol3 = herme.hermemul(pol1, pol2)
                        val3 = herme.hermeval(self.x, pol3)
                        assert_(len(pol3) == i + j + 1, msg)
                        assert_almost_equal(val3, val1*val2, err_msg=msg)

```

```

71: (4) def test_hermediv(self):
72: (8)     for i in range(5):
73: (12)         for j in range(5):
74: (16)             msg = f"At i={i}, j={j}"
75: (16)             ci = [0]*i + [1]
76: (16)             cj = [0]*j + [1]
77: (16)             tgt = herme.hermadd(ci, cj)
78: (16)             quo, rem = herme.hermdiv(tgt, ci)
79: (16)             res = herme.hermadd(herme.hermemul(quo, ci), rem)
80: (16)             assert_equal(trim(res), trim(tgt), err_msg=msg)
81: (4) def test_hermepow(self):
82: (8)     for i in range(5):
83: (12)         for j in range(5):
84: (16)             msg = f"At i={i}, j={j}"
85: (16)             c = np.arange(i + 1)
86: (16)             tgt = reduce(herme.hermemul, [c]*j, np.array([1]))
87: (16)             res = herme.hermepow(c, j)
88: (16)             assert_equal(trim(res), trim(tgt), err_msg=msg)
89: (0) class TestEvaluation:
90: (4)     c1d = np.array([4., 2., 3.])
91: (4)     c2d = np.einsum('i,j->ij', c1d, c1d)
92: (4)     c3d = np.einsum('i,j,k->ijk', c1d, c1d, c1d)
93: (4)     x = np.random.random((3, 5))*2 - 1
94: (4)     y = polyval(x, [1., 2., 3.])
95: (4)     def test_hermeval(self):
96: (8)         assert_equal(herme.hermeval([], [1]).size, 0)
97: (8)         x = np.linspace(-1, 1)
98: (8)         y = [polyval(x, c) for c in Helist]
99: (8)         for i in range(10):
100: (12)             msg = f"At i={i}"
101: (12)             tgt = y[i]
102: (12)             res = herme.hermeval(x, [0]*i + [1])
103: (12)             assert_almost_equal(res, tgt, err_msg=msg)
104: (8)         for i in range(3):
105: (12)             dims = [2]*i
106: (12)             x = np.zeros(dims)
107: (12)             assert_equal(herme.hermeval(x, [1]).shape, dims)
108: (12)             assert_equal(herme.hermeval(x, [1, 0]).shape, dims)
109: (12)             assert_equal(herme.hermeval(x, [1, 0, 0]).shape, dims)
110: (4)     def test_hermeval2d(self):
111: (8)         x1, x2, x3 = self.x
112: (8)         y1, y2, y3 = self.y
113: (8)         assert_raises(ValueError, herme.hermeval2d, x1, x2[:2], self.c2d)
114: (8)         tgt = y1*y2
115: (8)         res = herme.hermeval2d(x1, x2, self.c2d)
116: (8)         assert_almost_equal(res, tgt)
117: (8)         z = np.ones((2, 3))
118: (8)         res = herme.hermeval2d(z, z, self.c2d)
119: (8)         assert_(res.shape == (2, 3))
120: (4)     def test_hermeval3d(self):
121: (8)         x1, x2, x3 = self.x
122: (8)         y1, y2, y3 = self.y
123: (8)         assert_raises(ValueError, herme.hermeval3d, x1, x2, x3[:2], self.c3d)
124: (8)         tgt = y1*y2*y3
125: (8)         res = herme.hermeval3d(x1, x2, x3, self.c3d)
126: (8)         assert_almost_equal(res, tgt)
127: (8)         z = np.ones((2, 3))
128: (8)         res = herme.hermeval3d(z, z, z, self.c3d)
129: (8)         assert_(res.shape == (2, 3))
130: (4)     def test_hermegrid2d(self):
131: (8)         x1, x2, x3 = self.x
132: (8)         y1, y2, y3 = self.y
133: (8)         tgt = np.einsum('i,j->ij', y1, y2)
134: (8)         res = herme.hermegrid2d(x1, x2, self.c2d)
135: (8)         assert_almost_equal(res, tgt)
136: (8)         z = np.ones((2, 3))
137: (8)         res = herme.hermegrid2d(z, z, self.c2d)
138: (8)         assert_(res.shape == (2, 3)*2)
139: (4)     def test_hermegrid3d(self):

```

```

140: (8)           x1, x2, x3 = self.x
141: (8)           y1, y2, y3 = self.y
142: (8)           tgt = np.einsum('i,j,k->ijk', y1, y2, y3)
143: (8)           res = herme.hermegrid3d(x1, x2, x3, self.c3d)
144: (8)           assert_almost_equal(res, tgt)
145: (8)           z = np.ones((2, 3))
146: (8)           res = herme.hermegrid3d(z, z, z, self.c3d)
147: (8)           assert_(res.shape == (2, 3)*3)
148: (0)           class TestIntegral:
149: (4)             def test_hermeint(self):
150: (8)               assert_raises(TypeError, herme.hermoint, [0], .5)
151: (8)               assert_raises(ValueError, herme.hermoint, [0], -1)
152: (8)               assert_raises(ValueError, herme.hermoint, [0], 1, [0, 0])
153: (8)               assert_raises(ValueError, herme.hermoint, [0], lbnd=[0])
154: (8)               assert_raises(ValueError, herme.hermoint, [0], scl=[0])
155: (8)               assert_raises(TypeError, herme.hermoint, [0], axis=.5)
156: (8)             for i in range(2, 5):
157: (12)               k = [0]*(i - 2) + [1]
158: (12)               res = herme.hermoint([0], m=i, k=k)
159: (12)               assert_almost_equal(res, [0, 1])
160: (8)             for i in range(5):
161: (12)               scl = i + 1
162: (12)               pol = [0]*i + [1]
163: (12)               tgt = [i] + [0]*i + [1/scl]
164: (12)               hermopol = herme.poly2herme(pol)
165: (12)               hermeint = herme.hermoint(hermopol, m=1, k=[i])
166: (12)               res = herme.hermepoly(hermeint)
167: (12)               assert_almost_equal(trim(res), trim(tgt))
168: (8)             for i in range(5):
169: (12)               scl = i + 1
170: (12)               pol = [0]*i + [1]
171: (12)               hermopol = herme.poly2herme(pol)
172: (12)               hermeint = herme.hermoint(hermopol, m=1, k=[i], lbnd=-1)
173: (12)               assert_almost_equal(herme.hermeval(-1, hermeint), i)
174: (8)             for i in range(5):
175: (12)               scl = i + 1
176: (12)               pol = [0]*i + [1]
177: (12)               tgt = [i] + [0]*i + [2/scl]
178: (12)               hermopol = herme.poly2herme(pol)
179: (12)               hermeint = herme.hermoint(hermopol, m=1, k=[i], scl=2)
180: (12)               res = herme.hermepoly(hermeint)
181: (12)               assert_almost_equal(trim(res), trim(tgt))
182: (8)             for i in range(5):
183: (12)               for j in range(2, 5):
184: (16)                 pol = [0]*i + [1]
185: (16)                 tgt = pol[:]
186: (16)                 for k in range(j):
187: (20)                   tgt = herme.hermoint(tgt, m=1)
188: (16)                 res = herme.hermoint(pol, m=j)
189: (16)                 assert_almost_equal(trim(res), trim(tgt))
190: (8)             for i in range(5):
191: (12)               for j in range(2, 5):
192: (16)                 pol = [0]*i + [1]
193: (16)                 tgt = pol[:]
194: (16)                 for k in range(j):
195: (20)                   tgt = herme.hermoint(tgt, m=1, k=[k])
196: (16)                 res = herme.hermoint(pol, m=j, k=list(range(j)))
197: (16)                 assert_almost_equal(trim(res), trim(tgt))
198: (8)             for i in range(5):
199: (12)               for j in range(2, 5):
200: (16)                 pol = [0]*i + [1]
201: (16)                 tgt = pol[:]
202: (16)                 for k in range(j):
203: (20)                   tgt = herme.hermoint(tgt, m=1, k=[k], lbnd=-1)
204: (16)                 res = herme.hermoint(pol, m=j, k=list(range(j)), lbnd=-1)
205: (16)                 assert_almost_equal(trim(res), trim(tgt))
206: (8)             for i in range(5):
207: (12)               for j in range(2, 5):
208: (16)                 pol = [0]*i + [1]

```

```

209: (16)                      tgt = pol[:]
210: (16)                      for k in range(j):
211: (20)                          tgt = herme.hermeint(tgt, m=1, k=[k], scl=2)
212: (16)                          res = herme.hermeint(pol, m=j, k=list(range(j)), scl=2)
213: (16)                          assert_almost_equal(trim(res), trim(tgt))
214: (4)  def test_hermeint_axis(self):
215: (8)      c2d = np.random.random((3, 4))
216: (8)      tgt = np.vstack([herme.hermeint(c) for c in c2d.T]).T
217: (8)      res = herme.hermeint(c2d, axis=0)
218: (8)      assert_almost_equal(res, tgt)
219: (8)      tgt = np.vstack([herme.hermeint(c) for c in c2d])
220: (8)      res = herme.hermeint(c2d, axis=1)
221: (8)      assert_almost_equal(res, tgt)
222: (8)      tgt = np.vstack([herme.hermeint(c, k=3) for c in c2d])
223: (8)      res = herme.hermeint(c2d, k=3, axis=1)
224: (8)      assert_almost_equal(res, tgt)
225: (0)  class TestDerivative:
226: (4)    def test_hermeder(self):
227: (8)        assert_raises(TypeError, herme.hermeder, [0], .5)
228: (8)        assert_raises(ValueError, herme.hermeder, [0], -1)
229: (8)        for i in range(5):
230: (12)            tgt = [0]*i + [1]
231: (12)            res = herme.hermeder(tgt, m=0)
232: (12)            assert_equal(trim(res), trim(tgt))
233: (8)        for i in range(5):
234: (12)            for j in range(2, 5):
235: (16)                tgt = [0]*i + [1]
236: (16)                res = herme.hermeder(herme.hermeint(tgt, m=j), m=j)
237: (16)                assert_almost_equal(trim(res), trim(tgt))
238: (8)        for i in range(5):
239: (12)            for j in range(2, 5):
240: (16)                tgt = [0]*i + [1]
241: (16)                res = herme.hermeder(
242: (20)                    herme.hermeint(tgt, m=j, scl=2), m=j, scl=.5)
243: (16)                assert_almost_equal(trim(res), trim(tgt))
244: (4)    def test_hermeder_axis(self):
245: (8)      c2d = np.random.random((3, 4))
246: (8)      tgt = np.vstack([herme.hermeder(c) for c in c2d.T]).T
247: (8)      res = herme.hermeder(c2d, axis=0)
248: (8)      assert_almost_equal(res, tgt)
249: (8)      tgt = np.vstack([herme.hermeder(c) for c in c2d])
250: (8)      res = herme.hermeder(c2d, axis=1)
251: (8)      assert_almost_equal(res, tgt)
252: (0)  class TestVander:
253: (4)    x = np.random.random((3, 5))*2 - 1
254: (4)    def test_hermevander(self):
255: (8)      x = np.arange(3)
256: (8)      v = herme.hermevander(x, 3)
257: (8)      assert_(v.shape == (3, 4))
258: (8)      for i in range(4):
259: (12)          coef = [0]*i + [1]
260: (12)          assert_almost_equal(v[..., i], herme.hermeval(x, coef))
261: (8)      x = np.array([[1, 2], [3, 4], [5, 6]])
262: (8)      v = herme.hermevander(x, 3)
263: (8)      assert_(v.shape == (3, 2, 4))
264: (8)      for i in range(4):
265: (12)          coef = [0]*i + [1]
266: (12)          assert_almost_equal(v[..., i], herme.hermeval(x, coef))
267: (4)    def test_hermevander2d(self):
268: (8)      x1, x2, x3 = self.x
269: (8)      c = np.random.random((2, 3))
270: (8)      van = herme.hermevander2d(x1, x2, [1, 2])
271: (8)      tgt = herme.hermeval2d(x1, x2, c)
272: (8)      res = np.dot(van, c.flat)
273: (8)      assert_almost_equal(res, tgt)
274: (8)      van = herme.hermevander2d([x1], [x2], [1, 2])
275: (8)      assert_(van.shape == (1, 5, 6))
276: (4)    def test_hermevander3d(self):
277: (8)      x1, x2, x3 = self.x

```

```

278: (8)             c = np.random.random((2, 3, 4))
279: (8)             van = herme.hermevander3d(x1, x2, x3, [1, 2, 3])
280: (8)             tgt = herme.hermeval3d(x1, x2, x3, c)
281: (8)             res = np.dot(van, c.flat)
282: (8)             assert_almost_equal(res, tgt)
283: (8)             van = herme.hermevander3d([x1], [x2], [x3], [1, 2, 3])
284: (8)             assert_(van.shape == (1, 5, 24))
285: (0)             class TestFitting:
286: (4)                 def test_hermefit(self):
287: (8)                     def f(x):
288: (12)                         return x*(x - 1)*(x - 2)
289: (8)                     def f2(x):
290: (12)                         return x**4 + x**2 + 1
291: (8)                     assert_raises(ValueError, herme.hermefit, [1], [1], -1)
292: (8)                     assert_raises(TypeError, herme.hermefit, [[1]], [1], 0)
293: (8)                     assert_raises(TypeError, herme.hermefit, [], [1], 0)
294: (8)                     assert_raises(TypeError, herme.hermefit, [1], [[[1]]], 0)
295: (8)                     assert_raises(TypeError, herme.hermefit, [1, 2], [1], 0)
296: (8)                     assert_raises(TypeError, herme.hermefit, [1], [1, 2], 0)
297: (8)                     assert_raises(TypeError, herme.hermefit, [1], [1], 0, w=[[1]])
298: (8)                     assert_raises(TypeError, herme.hermefit, [1], [1], 0, w=[1, 1])
299: (8)                     assert_raises(ValueError, herme.hermefit, [1], [1], [-1,])
300: (8)                     assert_raises(ValueError, herme.hermefit, [1], [1], [2, -1, 6])
301: (8)                     assert_raises(TypeError, herme.hermefit, [1], [1], [])
302: (8)                     x = np.linspace(0, 2)
303: (8)                     y = f(x)
304: (8)                     coef3 = herme.hermefit(x, y, 3)
305: (8)                     assert_equal(len(coef3), 4)
306: (8)                     assert_almost_equal(herme.hermeval(x, coef3), y)
307: (8)                     coef3 = herme.hermefit(x, y, [0, 1, 2, 3])
308: (8)                     assert_equal(len(coef3), 4)
309: (8)                     assert_almost_equal(herme.hermeval(x, coef3), y)
310: (8)                     coef4 = herme.hermefit(x, y, 4)
311: (8)                     assert_equal(len(coef4), 5)
312: (8)                     assert_almost_equal(herme.hermeval(x, coef4), y)
313: (8)                     coef4 = herme.hermefit(x, y, [0, 1, 2, 3, 4])
314: (8)                     assert_equal(len(coef4), 5)
315: (8)                     assert_almost_equal(herme.hermeval(x, coef4), y)
316: (8)                     coef4 = herme.hermefit(x, y, [2, 3, 4, 1, 0])
317: (8)                     assert_equal(len(coef4), 5)
318: (8)                     assert_almost_equal(herme.hermeval(x, coef4), y)
319: (8)                     coef2d = herme.hermefit(x, np.array([y, y]).T, 3)
320: (8)                     assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
321: (8)                     coef2d = herme.hermefit(x, np.array([y, y]).T, [0, 1, 2, 3])
322: (8)                     assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
323: (8)                     w = np.zeros_like(x)
324: (8)                     yw = y.copy()
325: (8)                     w[1::2] = 1
326: (8)                     y[0::2] = 0
327: (8)                     wcoef3 = herme.hermefit(x, yw, 3, w=w)
328: (8)                     assert_almost_equal(wcoef3, coef3)
329: (8)                     wcoef3 = herme.hermefit(x, yw, [0, 1, 2, 3], w=w)
330: (8)                     assert_almost_equal(wcoef3, coef3)
331: (8)                     wcoef2d = herme.hermefit(x, np.array([yw, yw]).T, 3, w=w)
332: (8)                     assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
333: (8)                     wcoef2d = herme.hermefit(x, np.array([yw, yw]).T, [0, 1, 2, 3], w=w)
334: (8)                     assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
335: (8)                     x = [1, 1j, -1, -1j]
336: (8)                     assert_almost_equal(herme.hermefit(x, x, 1), [0, 1])
337: (8)                     assert_almost_equal(herme.hermefit(x, x, [0, 1]), [0, 1])
338: (8)                     x = np.linspace(-1, 1)
339: (8)                     y = f2(x)
340: (8)                     coef1 = herme.hermefit(x, y, 4)
341: (8)                     assert_almost_equal(herme.hermeval(x, coef1), y)
342: (8)                     coef2 = herme.hermefit(x, y, [0, 2, 4])
343: (8)                     assert_almost_equal(herme.hermeval(x, coef2), y)
344: (8)                     assert_almost_equal(coef1, coef2)
345: (0)             class TestCompanion:
346: (4)                 def test_raises(self):

```

```

347: (8)             assert_raises(ValueError, herme.hermecompanion, [])
348: (8)             assert_raises(ValueError, herme.hermecompanion, [1])
349: (4)             def test_dimensions(self):
350: (8)                 for i in range(1, 5):
351: (12)                     coef = [0]*i + [1]
352: (12)                     assert_(herme.hermecompanion(coef).shape == (i, i))
353: (4)             def test_linear_root(self):
354: (8)                 assert_(herme.hermecompanion([1, 2])[0, 0] == -.5)
355: (0)             class TestGauss:
356: (4)                 def test_100(self):
357: (8)                     x, w = herme.hermegauss(100)
358: (8)                     v = herme.hermevander(x, 99)
359: (8)                     vv = np.dot(v.T * w, v)
360: (8)                     vd = 1/np.sqrt(vv.diagonal())
361: (8)                     vv = vd[:, None] * vv * vd
362: (8)                     assert_almost_equal(vv, np.eye(100))
363: (8)                     tgt = np.sqrt(2*np.pi)
364: (8)                     assert_almost_equal(w.sum(), tgt)
365: (0)             class TestMisc:
366: (4)                 def test_hermefromroots(self):
367: (8)                     res = herme.hermefromroots([])
368: (8)                     assert_almost_equal(trim(res), [1])
369: (8)                     for i in range(1, 5):
370: (12)                         roots = np.cos(np.linspace(-np.pi, 0, 2*i + 1)[1::2])
371: (12)                         pol = herme.hermefromroots(roots)
372: (12)                         res = herme.hermeval(roots, pol)
373: (12)                         tgt = 0
374: (12)                         assert_(len(pol) == i + 1)
375: (12)                         assert_almost_equal(herme.herme2poly(pol)[-1], 1)
376: (12)                         assert_almost_equal(res, tgt)
377: (4)                 def test_hermeroots(self):
378: (8)                     assert_almost_equal(herme.hermeroots([1]), [])
379: (8)                     assert_almost_equal(herme.hermeroots([1, 1]), [-1])
380: (8)                     for i in range(2, 5):
381: (12)                         tgt = np.linspace(-1, 1, i)
382: (12)                         res = herme.hermeroots(herme.hermefromroots(tgt))
383: (12)                         assert_almost_equal(trim(res), trim(tgt))
384: (4)                 def test_hermetrim(self):
385: (8)                     coef = [2, -1, 1, 0]
386: (8)                     assert_raises(ValueError, herme.hermetrim, coef, -1)
387: (8)                     assert_equal(herme.hermetrim(coef), coef[:-1])
388: (8)                     assert_equal(herme.hermetrim(coef, 1), coef[:-3])
389: (8)                     assert_equal(herme.hermetrim(coef, 2), [0])
390: (4)                 def test_hermeline(self):
391: (8)                     assert_equal(herme.hermeline(3, 4), [3, 4])
392: (4)                 def test_herme2poly(self):
393: (8)                     for i in range(10):
394: (12)                         assert_almost_equal(herme.herme2poly([0]*i + [1]), Helist[i])
395: (4)                 def test_poly2herme(self):
396: (8)                     for i in range(10):
397: (12)                         assert_almost_equal(herme.poly2herme(Helist[i]), [0]*i + [1])
398: (4)                 def test_weight(self):
399: (8)                     x = np.linspace(-5, 5, 11)
400: (8)                     tgt = np.exp(-.5*x**2)
401: (8)                     res = herme.hermeweight(x)
402: (8)                     assert_almost_equal(res, tgt)

```

-----  
File 301 - test\_laguerre.py:

```

1: (0)             """Tests for laguerre module.
2: (0)             """
3: (0)             from functools import reduce
4: (0)             import numpy as np
5: (0)             import numpy.polynomial.laguerre as lag
6: (0)             from numpy.polynomial.polynomial import polyval
7: (0)             from numpy.testing import (
8: (4)                 assert_almost_equal, assert_raises, assert_equal, assert_

```

```

9: (4) )
10: (0) L0 = np.array([1])/1
11: (0) L1 = np.array([1, -1])/1
12: (0) L2 = np.array([2, -4, 1])/2
13: (0) L3 = np.array([6, -18, 9, -1])/6
14: (0) L4 = np.array([24, -96, 72, -16, 1])/24
15: (0) L5 = np.array([120, -600, 600, -200, 25, -1])/120
16: (0) L6 = np.array([720, -4320, 5400, -2400, 450, -36, 1])/720
17: (0) Llist = [L0, L1, L2, L3, L4, L5, L6]
18: (0) def trim(x):
19: (4)     return lag.lagtrim(x, tol=1e-6)
20: (0) class TestConstants:
21: (4)     def test_lagdomain(self):
22: (8)         assert_equal(lag.lagdomain, [0, 1])
23: (4)     def test_lagzero(self):
24: (8)         assert_equal(lag.lagzero, [0])
25: (4)     def test_lagone(self):
26: (8)         assert_equal(lag.lagone, [1])
27: (4)     def test_lagx(self):
28: (8)         assert_equal(lag.lagx, [1, -1])
29: (0) class TestArithmetic:
30: (4)     x = np.linspace(-3, 3, 100)
31: (4)     def test_lagadd(self):
32: (8)         for i in range(5):
33: (12)             for j in range(5):
34: (16)                 msg = f"At i={i}, j={j}"
35: (16)                 tgt = np.zeros(max(i, j) + 1)
36: (16)                 tgt[i] += 1
37: (16)                 tgt[j] += 1
38: (16)                 res = lag.lagadd([0]*i + [1], [0]*j + [1])
39: (16)                 assert_equal(trim(res), trim(tgt), err_msg=msg)
40: (4)     def test_lagsub(self):
41: (8)         for i in range(5):
42: (12)             for j in range(5):
43: (16)                 msg = f"At i={i}, j={j}"
44: (16)                 tgt = np.zeros(max(i, j) + 1)
45: (16)                 tgt[i] += 1
46: (16)                 tgt[j] -= 1
47: (16)                 res = lag.lagsub([0]*i + [1], [0]*j + [1])
48: (16)                 assert_equal(trim(res), trim(tgt), err_msg=msg)
49: (4)     def test_lagmulx(self):
50: (8)         assert_equal(lag.lagmulx([0]), [0])
51: (8)         assert_equal(lag.lagmulx([1]), [1, -1])
52: (8)         for i in range(1, 5):
53: (12)             ser = [0]*i + [1]
54: (12)             tgt = [0]*(i - 1) + [-i, 2*i + 1, -(i + 1)]
55: (12)             assert_almost_equal(lag.lagmulx(ser), tgt)
56: (4)     def test_lagmul(self):
57: (8)         for i in range(5):
58: (12)             pol1 = [0]*i + [1]
59: (12)             val1 = lag.lagval(self.x, pol1)
60: (12)             for j in range(5):
61: (16)                 msg = f"At i={i}, j={j}"
62: (16)                 pol2 = [0]*j + [1]
63: (16)                 val2 = lag.lagval(self.x, pol2)
64: (16)                 pol3 = lag.lagmul(pol1, pol2)
65: (16)                 val3 = lag.lagval(self.x, pol3)
66: (16)                 assert_(len(pol3) == i + j + 1, msg)
67: (16)                 assert_almost_equal(val3, val1*val2, err_msg=msg)
68: (4)     def test_lagdiv(self):
69: (8)         for i in range(5):
70: (12)             for j in range(5):
71: (16)                 msg = f"At i={i}, j={j}"
72: (16)                 ci = [0]*i + [1]
73: (16)                 cj = [0]*j + [1]
74: (16)                 tgt = lag.lagadd(ci, cj)
75: (16)                 quo, rem = lag.lagdiv(tgt, ci)
76: (16)                 res = lag.lagadd(lag.lagmul(quo, ci), rem)
77: (16)                 assert_almost_equal(trim(res), trim(tgt), err_msg=msg)

```

```

78: (4)         def test_lagpow(self):
79: (8)             for i in range(5):
80: (12)                 for j in range(5):
81: (16)                     msg = f"At i={i}, j={j}"
82: (16)                     c = np.arange(i + 1)
83: (16)                     tgt = reduce(lag.lagmul, [c]*j, np.array([1]))
84: (16)                     res = lag.lagpow(c, j)
85: (16)                     assert_equal(trim(res), trim(tgt), err_msg=msg)
86: (0)     class TestEvaluation:
87: (4)         c1d = np.array([9., -14., 6.])
88: (4)         c2d = np.einsum('i,j->ij', c1d, c1d)
89: (4)         c3d = np.einsum('i,j,k->ijk', c1d, c1d, c1d)
90: (4)         x = np.random.random((3, 5))*2 - 1
91: (4)         y = polyval(x, [1., 2., 3.])
92: (4)         def test_lagval(self):
93: (8)             assert_equal(lag.lagval([], [1]).size, 0)
94: (8)             x = np.linspace(-1, 1)
95: (8)             y = [polyval(x, c) for c in Llist]
96: (8)             for i in range(7):
97: (12)                 msg = f"At i={i}"
98: (12)                 tgt = y[i]
99: (12)                 res = lag.lagval(x, [0]*i + [1])
100: (12)                assert_almost_equal(res, tgt, err_msg=msg)
101: (8)             for i in range(3):
102: (12)                 dims = [2]*i
103: (12)                 x = np.zeros(dims)
104: (12)                 assert_equal(lag.lagval(x, [1]).shape, dims)
105: (12)                 assert_equal(lag.lagval(x, [1, 0]).shape, dims)
106: (12)                 assert_equal(lag.lagval(x, [1, 0, 0]).shape, dims)
107: (4)         def test_lagval2d(self):
108: (8)             x1, x2, x3 = self.x
109: (8)             y1, y2, y3 = self.y
110: (8)             assert_raises(ValueError, lag.lagval2d, x1, x2[:2], self.c2d)
111: (8)             tgt = y1*y2
112: (8)             res = lag.lagval2d(x1, x2, self.c2d)
113: (8)             assert_almost_equal(res, tgt)
114: (8)             z = np.ones((2, 3))
115: (8)             res = lag.lagval2d(z, z, self.c2d)
116: (8)             assert_(res.shape == (2, 3))
117: (4)         def test_lagval3d(self):
118: (8)             x1, x2, x3 = self.x
119: (8)             y1, y2, y3 = self.y
120: (8)             assert_raises(ValueError, lag.lagval3d, x1, x2, x3[:2], self.c3d)
121: (8)             tgt = y1*y2*y3
122: (8)             res = lag.lagval3d(x1, x2, x3, self.c3d)
123: (8)             assert_almost_equal(res, tgt)
124: (8)             z = np.ones((2, 3))
125: (8)             res = lag.lagval3d(z, z, z, self.c3d)
126: (8)             assert_(res.shape == (2, 3))
127: (4)         def test_laggrid2d(self):
128: (8)             x1, x2, x3 = self.x
129: (8)             y1, y2, y3 = self.y
130: (8)             tgt = np.einsum('i,j->ij', y1, y2)
131: (8)             res = lag.laggrid2d(x1, x2, self.c2d)
132: (8)             assert_almost_equal(res, tgt)
133: (8)             z = np.ones((2, 3))
134: (8)             res = lag.laggrid2d(z, z, self.c2d)
135: (8)             assert_(res.shape == (2, 3)*2)
136: (4)         def test_laggrid3d(self):
137: (8)             x1, x2, x3 = self.x
138: (8)             y1, y2, y3 = self.y
139: (8)             tgt = np.einsum('i,j,k->ijk', y1, y2, y3)
140: (8)             res = lag.laggrid3d(x1, x2, x3, self.c3d)
141: (8)             assert_almost_equal(res, tgt)
142: (8)             z = np.ones((2, 3))
143: (8)             res = lag.laggrid3d(z, z, z, self.c3d)
144: (8)             assert_(res.shape == (2, 3)*3)
145: (0)     class TestIntegral:
146: (4)         def test_lagint(self):

```

```

147: (8) assert_raises(TypeError, lag.lagint, [0], .5)
148: (8) assert_raises(ValueError, lag.lagint, [0], -1)
149: (8) assert_raises(ValueError, lag.lagint, [0], 1, [0, 0])
150: (8) assert_raises(ValueError, lag.lagint, [0], lbnd=[0])
151: (8) assert_raises(ValueError, lag.lagint, [0], scl=[0])
152: (8) assert_raises(TypeError, lag.lagint, [0], axis=.5)
153: (8) for i in range(2, 5):
154: (12)     k = [0]*(i - 2) + [1]
155: (12)     res = lag.lagint([0], m=i, k=k)
156: (12)     assert_almost_equal(res, [1, -1])
157: (8) for i in range(5):
158: (12)     scl = i + 1
159: (12)     pol = [0]*i + [1]
160: (12)     tgt = [i] + [0]*i + [1/scl]
161: (12)     lagpol = lag.poly2lag(pol)
162: (12)     lagint = lag.lagint(lagpol, m=1, k=[i])
163: (12)     res = lag.lag2poly(lagint)
164: (12)     assert_almost_equal(trim(res), trim(tgt))
165: (8) for i in range(5):
166: (12)     scl = i + 1
167: (12)     pol = [0]*i + [1]
168: (12)     lagpol = lag.poly2lag(pol)
169: (12)     lagint = lag.lagint(lagpol, m=1, k=[i], lbnd=-1)
170: (12)     assert_almost_equal(lag.lagval(-1, lagint), i)
171: (8) for i in range(5):
172: (12)     scl = i + 1
173: (12)     pol = [0]*i + [1]
174: (12)     tgt = [i] + [0]*i + [2/scl]
175: (12)     lagpol = lag.poly2lag(pol)
176: (12)     lagint = lag.lagint(lagpol, m=1, k=[i], scl=2)
177: (12)     res = lag.lag2poly(lagint)
178: (12)     assert_almost_equal(trim(res), trim(tgt))
179: (8) for i in range(5):
180: (12)     for j in range(2, 5):
181: (16)         pol = [0]*i + [1]
182: (16)         tgt = pol[:]
183: (16)         for k in range(j):
184: (20)             tgt = lag.lagint(tgt, m=1)
185: (16)             res = lag.lagint(pol, m=j)
186: (16)             assert_almost_equal(trim(res), trim(tgt))
187: (8) for i in range(5):
188: (12)     for j in range(2, 5):
189: (16)         pol = [0]*i + [1]
190: (16)         tgt = pol[:]
191: (16)         for k in range(j):
192: (20)             tgt = lag.lagint(tgt, m=1, k=[k])
193: (16)             res = lag.lagint(pol, m=j, k=list(range(j)))
194: (16)             assert_almost_equal(trim(res), trim(tgt))
195: (8) for i in range(5):
196: (12)     for j in range(2, 5):
197: (16)         pol = [0]*i + [1]
198: (16)         tgt = pol[:]
199: (16)         for k in range(j):
200: (20)             tgt = lag.lagint(tgt, m=1, k=[k], lbnd=-1)
201: (16)             res = lag.lagint(pol, m=j, k=list(range(j)), lbnd=-1)
202: (16)             assert_almost_equal(trim(res), trim(tgt))
203: (8) for i in range(5):
204: (12)     for j in range(2, 5):
205: (16)         pol = [0]*i + [1]
206: (16)         tgt = pol[:]
207: (16)         for k in range(j):
208: (20)             tgt = lag.lagint(tgt, m=1, k=[k], scl=2)
209: (16)             res = lag.lagint(pol, m=j, k=list(range(j)), scl=2)
210: (16)             assert_almost_equal(trim(res), trim(tgt))
211: (4) def test_lagint_axis(self):
212: (8)     c2d = np.random.random((3, 4))
213: (8)     tgt = np.vstack([lag.lagint(c) for c in c2d.T]).T
214: (8)     res = lag.lagint(c2d, axis=0)
215: (8)     assert_almost_equal(res, tgt)

```

```

216: (8)             tgt = np.vstack([lag.lagint(c) for c in c2d])
217: (8)             res = lag.lagint(c2d, axis=1)
218: (8)             assert_almost_equal(res, tgt)
219: (8)             tgt = np.vstack([lag.lagint(c, k=3) for c in c2d])
220: (8)             res = lag.lagint(c2d, k=3, axis=1)
221: (8)             assert_almost_equal(res, tgt)
222: (0)             class TestDerivative:
223: (4)                 def test_lagder(self):
224: (8)                     assert_raises(TypeError, lag.lagder, [0], .5)
225: (8)                     assert_raises(ValueError, lag.lagder, [0], -1)
226: (8)                     for i in range(5):
227: (12)                         tgt = [0]*i + [1]
228: (12)                         res = lag.lagder(tgt, m=0)
229: (12)                         assert_equal(trim(res), trim(tgt))
230: (8)                     for i in range(5):
231: (12)                         for j in range(2, 5):
232: (16)                             tgt = [0]*i + [1]
233: (16)                             res = lag.lagder(lag.lagint(tgt, m=j), m=j)
234: (16)                             assert_almost_equal(trim(res), trim(tgt))
235: (8)                     for i in range(5):
236: (12)                         for j in range(2, 5):
237: (16)                             tgt = [0]*i + [1]
238: (16)                             res = lag.lagder(lag.lagint(tgt, m=j, scl=2), m=j, scl=.5)
239: (16)                             assert_almost_equal(trim(res), trim(tgt))
240: (4)             def test_lagder_axis(self):
241: (8)                 c2d = np.random.random((3, 4))
242: (8)                 tgt = np.vstack([lag.lagder(c) for c in c2d.T]).T
243: (8)                 res = lag.lagder(c2d, axis=0)
244: (8)                 assert_almost_equal(res, tgt)
245: (8)                 tgt = np.vstack([lag.lagder(c) for c in c2d])
246: (8)                 res = lag.lagder(c2d, axis=1)
247: (8)                 assert_almost_equal(res, tgt)
248: (0)             class TestVander:
249: (4)                 x = np.random.random((3, 5))*2 - 1
250: (4)                 def test_lagvander(self):
251: (8)                     x = np.arange(3)
252: (8)                     v = lag.lagvander(x, 3)
253: (8)                     assert_(v.shape == (3, 4))
254: (8)                     for i in range(4):
255: (12)                         coef = [0]*i + [1]
256: (12)                         assert_almost_equal(v[..., i], lag.lagval(x, coef))
257: (8)                     x = np.array([[1, 2], [3, 4], [5, 6]])
258: (8)                     v = lag.lagvander(x, 3)
259: (8)                     assert_(v.shape == (3, 2, 4))
260: (8)                     for i in range(4):
261: (12)                         coef = [0]*i + [1]
262: (12)                         assert_almost_equal(v[..., i], lag.lagval(x, coef))
263: (4)             def test_lagvander2d(self):
264: (8)                 x1, x2, x3 = self.x
265: (8)                 c = np.random.random((2, 3))
266: (8)                 van = lag.lagvander2d(x1, x2, [1, 2])
267: (8)                 tgt = lag.lagval2d(x1, x2, c)
268: (8)                 res = np.dot(van, c.flat)
269: (8)                 assert_almost_equal(res, tgt)
270: (8)                 van = lag.lagvander2d([x1], [x2], [1, 2])
271: (8)                 assert_(van.shape == (1, 5, 6))
272: (4)             def test_lagvander3d(self):
273: (8)                 x1, x2, x3 = self.x
274: (8)                 c = np.random.random((2, 3, 4))
275: (8)                 van = lag.lagvander3d(x1, x2, x3, [1, 2, 3])
276: (8)                 tgt = lag.lagval3d(x1, x2, x3, c)
277: (8)                 res = np.dot(van, c.flat)
278: (8)                 assert_almost_equal(res, tgt)
279: (8)                 van = lag.lagvander3d([x1], [x2], [x3], [1, 2, 3])
280: (8)                 assert_(van.shape == (1, 5, 24))
281: (0)             class TestFitting:
282: (4)                 def test_lagfit(self):
283: (8)                     def f(x):
284: (12)                         return x*(x - 1)*(x - 2)

```

```

285: (8) assert_raises(ValueError, lag.lagfit, [1], [1], -1)
286: (8) assert_raises(TypeError, lag.lagfit, [[1]], [1], 0)
287: (8) assert_raises(TypeError, lag.lagfit, [], [1], 0)
288: (8) assert_raises(TypeError, lag.lagfit, [1], [[[1]]], 0)
289: (8) assert_raises(TypeError, lag.lagfit, [1], [1, 2], [1], 0)
290: (8) assert_raises(TypeError, lag.lagfit, [1], [1, 2], 0)
291: (8) assert_raises(TypeError, lag.lagfit, [1], [1], 0, w=[[1]])
292: (8) assert_raises(TypeError, lag.lagfit, [1], [1], 0, w=[1, 1])
293: (8) assert_raises(ValueError, lag.lagfit, [1], [1], [-1,])
294: (8) assert_raises(ValueError, lag.lagfit, [1], [1], [2, -1, 6])
295: (8) assert_raises(TypeError, lag.lagfit, [1], [1], [])
296: (8) x = np.linspace(0, 2)
297: (8) y = f(x)
298: (8) coef3 = lag.lagfit(x, y, 3)
299: (8) assert_equal(len(coef3), 4)
300: (8) assert_almost_equal(lag.lagval(x, coef3), y)
301: (8) coef3 = lag.lagfit(x, y, [0, 1, 2, 3])
302: (8) assert_equal(len(coef3), 4)
303: (8) assert_almost_equal(lag.lagval(x, coef3), y)
304: (8) coef4 = lag.lagfit(x, y, 4)
305: (8) assert_equal(len(coef4), 5)
306: (8) assert_almost_equal(lag.lagval(x, coef4), y)
307: (8) coef4 = lag.lagfit(x, y, [0, 1, 2, 3, 4])
308: (8) assert_equal(len(coef4), 5)
309: (8) assert_almost_equal(lag.lagval(x, coef4), y)
310: (8) coef2d = lag.lagfit(x, np.array([y, y]).T, 3)
311: (8) assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
312: (8) coef2d = lag.lagfit(x, np.array([y, y]).T, [0, 1, 2, 3])
313: (8) assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
314: (8) w = np.zeros_like(x)
315: (8) yw = y.copy()
316: (8) w[1::2] = 1
317: (8) y[0::2] = 0
318: (8) wcoef3 = lag.lagfit(x, yw, 3, w=w)
319: (8) assert_almost_equal(wcoef3, coef3)
320: (8) wcoef3 = lag.lagfit(x, yw, [0, 1, 2, 3], w=w)
321: (8) assert_almost_equal(wcoef3, coef3)
322: (8) wcoef2d = lag.lagfit(x, np.array([yw, yw]).T, 3, w=w)
323: (8) assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
324: (8) wcoef2d = lag.lagfit(x, np.array([yw, yw]).T, [0, 1, 2, 3], w=w)
325: (8) assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
326: (8) x = [1, 1j, -1, -1j]
327: (8) assert_almost_equal(lag.lagfit(x, x, 1), [1, -1])
328: (8) assert_almost_equal(lag.lagfit(x, x, [0, 1]), [1, -1])
329: (0) class TestCompanion:
330: (4)     def test_raises(self):
331: (8)         assert_raises(ValueError, lag.lagcompanion, [])
332: (8)         assert_raises(ValueError, lag.lagcompanion, [1])
333: (4)     def test_dimensions(self):
334: (8)         for i in range(1, 5):
335: (12)             coef = [0]*i + [1]
336: (12)             assert_(lag.lagcompanion(coef).shape == (i, i))
337: (4)     def test_linear_root(self):
338: (8)         assert_(lag.lagcompanion([1, 2])[0, 0] == 1.5)
339: (0) class TestGauss:
340: (4)     def test_100(self):
341: (8)         x, w = lag.laggauss(100)
342: (8)         v = lag.lagvander(x, 99)
343: (8)         vv = np.dot(v.T * w, v)
344: (8)         vd = 1/np.sqrt(vv.diagonal())
345: (8)         vv = vd[:, None] * vv * vd
346: (8)         assert_almost_equal(vv, np.eye(100))
347: (8)         tgt = 1.0
348: (8)         assert_almost_equal(w.sum(), tgt)
349: (0) class TestMisc:
350: (4)     def test_lagfromroots(self):
351: (8)         res = lag.lagfromroots([])
352: (8)         assert_almost_equal(trim(res), [1])
353: (8)         for i in range(1, 5):

```

```

354: (12)           roots = np.cos(np.linspace(-np.pi, 0, 2*i + 1)[1::2])
355: (12)           pol = lag.lagfromroots(roots)
356: (12)           res = lag.lagval(roots, pol)
357: (12)           tgt = 0
358: (12)           assert_(len(pol) == i + 1)
359: (12)           assert_almost_equal(lag.lag2poly(pol)[-1], 1)
360: (12)           assert_almost_equal(res, tgt)
361: (4)            def test_lagroots(self):
362: (8)              assert_almost_equal(lag.lagroots([1]), [])
363: (8)              assert_almost_equal(lag.lagroots([0, 1]), [1])
364: (8)              for i in range(2, 5):
365: (12)                  tgt = np.linspace(0, 3, i)
366: (12)                  res = lag.lagroots(lag.lagfromroots(tgt))
367: (12)                  assert_almost_equal(trim(res), trim(tgt))
368: (4)            def test_lagtrim(self):
369: (8)              coef = [2, -1, 1, 0]
370: (8)              assert_raises(ValueError, lag.lagtrim, coef, -1)
371: (8)              assert_equal(lag.lagtrim(coef), coef[:-1])
372: (8)              assert_equal(lag.lagtrim(coef, 1), coef[:-3])
373: (8)              assert_equal(lag.lagtrim(coef, 2), [0])
374: (4)            def test_lagline(self):
375: (8)              assert_equal(lag.lagline(3, 4), [7, -4])
376: (4)            def test_lag2poly(self):
377: (8)              for i in range(7):
378: (12)                  assert_almost_equal(lag.lag2poly([0]*i + [1]), Llist[i])
379: (4)            def test_poly2lag(self):
380: (8)              for i in range(7):
381: (12)                  assert_almost_equal(lag.poly2lag(Llist[i]), [0]*i + [1])
382: (4)            def test_weight(self):
383: (8)              x = np.linspace(0, 10, 11)
384: (8)              tgt = np.exp(-x)
385: (8)              res = lag.lagweight(x)
386: (8)              assert_almost_equal(res, tgt)

```

---

## File 302 - test\_legendre.py:

```

1: (0)      """Tests for legendre module.
2: (0)      """
3: (0)      from functools import reduce
4: (0)      import numpy as np
5: (0)      import numpy.polynomial.legendre as leg
6: (0)      from numpy.polynomial.polynomial import polyval
7: (0)      from numpy.testing import (
8: (4)          assert_almost_equal, assert_raises, assert_equal, assert_,
9: (4)      )
10: (0)     L0 = np.array([1])
11: (0)     L1 = np.array([0, 1])
12: (0)     L2 = np.array([-1, 0, 3])/2
13: (0)     L3 = np.array([0, -3, 0, 5])/2
14: (0)     L4 = np.array([3, 0, -30, 0, 35])/8
15: (0)     L5 = np.array([0, 15, 0, -70, 0, 63])/8
16: (0)     L6 = np.array([-5, 0, 105, 0, -315, 0, 231])/16
17: (0)     L7 = np.array([0, -35, 0, 315, 0, -693, 0, 429])/16
18: (0)     L8 = np.array([35, 0, -1260, 0, 6930, 0, -12012, 0, 6435])/128
19: (0)     L9 = np.array([0, 315, 0, -4620, 0, 18018, 0, -25740, 0, 12155])/128
20: (0)     Llist = [L0, L1, L2, L3, L4, L5, L6, L7, L8, L9]
21: (0)     def trim(x):
22: (4)         return leg.legtrim(x, tol=1e-6)
23: (0)     class TestConstants:
24: (4)         def test_legdomain(self):
25: (8)             assert_equal(leg.legdomain, [-1, 1])
26: (4)         def test_legzero(self):
27: (8)             assert_equal(leg.legzero, [0])
28: (4)         def test_legone(self):
29: (8)             assert_equal(leg.legone, [1])
30: (4)         def test_legx(self):
31: (8)             assert_equal(leg.legx, [0, 1])

```

```

32: (0)
33: (4)
34: (4)
35: (8)
36: (12)
37: (16)
38: (16)
39: (16)
40: (16)
41: (16)
42: (16)
43: (4)
44: (8)
45: (12)
46: (16)
47: (16)
48: (16)
49: (16)
50: (16)
51: (16)
52: (4)
53: (8)
54: (8)
55: (8)
56: (12)
57: (12)
58: (12)
59: (12)
60: (4)
61: (8)
62: (12)
63: (12)
64: (12)
65: (16)
66: (16)
67: (16)
68: (16)
69: (16)
70: (16)
71: (16)
72: (4)
73: (8)
74: (12)
75: (16)
76: (16)
77: (16)
78: (16)
79: (16)
80: (16)
81: (16)
82: (4)
83: (8)
84: (12)
85: (16)
86: (16)
87: (16)
88: (16)
89: (16)
90: (0)
91: (4)
92: (4)
93: (4)
94: (4)
95: (4)
96: (4)
97: (8)
98: (8)
99: (8)
100: (8)

    class TestArithmetic:
        x = np.linspace(-1, 1, 100)
        def test_legadd(self):
            for i in range(5):
                for j in range(5):
                    msg = f"At i={i}, j={j}"
                    tgt = np.zeros(max(i, j) + 1)
                    tgt[i] += 1
                    tgt[j] += 1
                    res = leg.legadd([0]*i + [1], [0]*j + [1])
                    assert_equal(trim(res), trim(tgt), err_msg=msg)
        def test_legsub(self):
            for i in range(5):
                for j in range(5):
                    msg = f"At i={i}, j={j}"
                    tgt = np.zeros(max(i, j) + 1)
                    tgt[i] += 1
                    tgt[j] -= 1
                    res = leg.legsub([0]*i + [1], [0]*j + [1])
                    assert_equal(trim(res), trim(tgt), err_msg=msg)
        def test_legmulx(self):
            assert_equal(leg.legmulx([0]), [0])
            assert_equal(leg.legmulx([1]), [0, 1])
            for i in range(1, 5):
                tmp = 2*i + 1
                ser = [0]*i + [1]
                tgt = [0]^(i - 1) + [i/tmp, 0, (i + 1)/tmp]
                assert_equal(leg.legmulx(ser), tgt)
        def test_legmul(self):
            for i in range(5):
                pol1 = [0]*i + [1]
                val1 = leg.legval(self.x, pol1)
                for j in range(5):
                    msg = f"At i={i}, j={j}"
                    pol2 = [0]*j + [1]
                    val2 = leg.legval(self.x, pol2)
                    pol3 = leg.legmul(pol1, pol2)
                    val3 = leg.legval(self.x, pol3)
                    assert_(len(pol3) == i + j + 1, msg)
                    assert_almost_equal(val3, val1*val2, err_msg=msg)
        def test_legdiv(self):
            for i in range(5):
                for j in range(5):
                    msg = f"At i={i}, j={j}"
                    ci = [0]*i + [1]
                    cj = [0]*j + [1]
                    tgt = leg.legadd(ci, cj)
                    quo, rem = leg.legdiv(tgt, ci)
                    res = leg.legadd(leg.legmul(quo, ci), rem)
                    assert_equal(trim(res), trim(tgt), err_msg=msg)
        def test_legpow(self):
            for i in range(5):
                for j in range(5):
                    msg = f"At i={i}, j={j}"
                    c = np.arange(i + 1)
                    tgt = reduce(leg.legmul, [c]*j, np.array([1]))
                    res = leg.legpow(c, j)
                    assert_equal(trim(res), trim(tgt), err_msg=msg)

    class TestEvaluation:
        c1d = np.array([2., 2., 2.])
        c2d = np.einsum('i,j->ij', c1d, c1d)
        c3d = np.einsum('i,j,k->ijk', c1d, c1d, c1d)
        x = np.random.random((3, 5))*2 - 1
        y = polyval(x, [1., 2., 3.])
        def test_legval(self):
            assert_equal(leg.legval([], [1]).size, 0)
            x = np.linspace(-1, 1)
            y = [polyval(x, c) for c in Llist]
            for i in range(10):

```

```

101: (12)                         msg = f"At i={i}"
102: (12)                         tgt = y[i]
103: (12)                         res = leg.legval(x, [0]*i + [1])
104: (12)                         assert_almost_equal(res, tgt, err_msg=msg)
105: (8)                          for i in range(3):
106: (12)                            dims = [2]*i
107: (12)                            x = np.zeros(dims)
108: (12)                            assert_equal(leg.legval(x, [1]).shape, dims)
109: (12)                            assert_equal(leg.legval(x, [1, 0]).shape, dims)
110: (12)                            assert_equal(leg.legval(x, [1, 0, 0]).shape, dims)
111: (4)                           def test_legval2d(self):
112: (8)                             x1, x2, x3 = self.x
113: (8)                             y1, y2, y3 = self.y
114: (8)                             assert_raises(ValueError, leg.legval2d, x1, x2[:2], self.c2d)
115: (8)                             tgt = y1*y2
116: (8)                             res = leg.legval2d(x1, x2, self.c2d)
117: (8)                             assert_almost_equal(res, tgt)
118: (8)                             z = np.ones((2, 3))
119: (8)                             res = leg.legval2d(z, z, self.c2d)
120: (8)                             assert_(res.shape == (2, 3))
121: (4)                           def test_legval3d(self):
122: (8)                             x1, x2, x3 = self.x
123: (8)                             y1, y2, y3 = self.y
124: (8)                             assert_raises(ValueError, leg.legval3d, x1, x2, x3[:2], self.c3d)
125: (8)                             tgt = y1*y2*y3
126: (8)                             res = leg.legval3d(x1, x2, x3, self.c3d)
127: (8)                             assert_almost_equal(res, tgt)
128: (8)                             z = np.ones((2, 3))
129: (8)                             res = leg.legval3d(z, z, z, self.c3d)
130: (8)                             assert_(res.shape == (2, 3))
131: (4)                           def test_leggrid2d(self):
132: (8)                             x1, x2, x3 = self.x
133: (8)                             y1, y2, y3 = self.y
134: (8)                             tgt = np.einsum('i,j->ij', y1, y2)
135: (8)                             res = leg.leggrid2d(x1, x2, self.c2d)
136: (8)                             assert_almost_equal(res, tgt)
137: (8)                             z = np.ones((2, 3))
138: (8)                             res = leg.leggrid2d(z, z, self.c2d)
139: (8)                             assert_(res.shape == (2, 3)*2)
140: (4)                           def test_leggrid3d(self):
141: (8)                             x1, x2, x3 = self.x
142: (8)                             y1, y2, y3 = self.y
143: (8)                             tgt = np.einsum('i,j,k->ijk', y1, y2, y3)
144: (8)                             res = leg.leggrid3d(x1, x2, x3, self.c3d)
145: (8)                             assert_almost_equal(res, tgt)
146: (8)                             z = np.ones((2, 3))
147: (8)                             res = leg.leggrid3d(z, z, z, self.c3d)
148: (8)                             assert_(res.shape == (2, 3)*3)
149: (0)                           class TestIntegral:
150: (4)                             def test_legint(self):
151: (8)                               assert_raises(TypeError, leg.legint, [0], .5)
152: (8)                               assert_raises(ValueError, leg.legint, [0], -1)
153: (8)                               assert_raises(ValueError, leg.legint, [0], 1, [0, 0])
154: (8)                               assert_raises(ValueError, leg.legint, [0], lbnd=[0])
155: (8)                               assert_raises(ValueError, leg.legint, [0], scl=[0])
156: (8)                               assert_raises(TypeError, leg.legint, [0], axis=.5)
157: (8)                               for i in range(2, 5):
158: (12)                                 k = [0] * (i - 2) + [1]
159: (12)                                 res = leg.legint([0], m=i, k=k)
160: (12)                                 assert_almost_equal(res, [0, 1])
161: (8)                               for i in range(5):
162: (12)                                 scl = i + 1
163: (12)                                 pol = [0]*i + [1]
164: (12)                                 tgt = [i] + [0]*i + [1/scl]
165: (12)                                 legpol = leg.poly2leg(pol)
166: (12)                                 legint = leg.legint(legpol, m=1, k=[i])
167: (12)                                 res = leg.leg2poly(legint)
168: (12)                                 assert_almost_equal(trim(res), trim(tgt))
169: (8)                               for i in range(5):

```

```

170: (12)           scl = i + 1
171: (12)           pol = [0]*i + [1]
172: (12)           legpol = leg.poly2leg(pol)
173: (12)           legint = leg.legint(legpol, m=1, k=[i], lbnd=-1)
174: (12)           assert_almost_equal(leg.legval(-1, legint), i)
175: (8)            for i in range(5):
176: (12)               scl = i + 1
177: (12)               pol = [0]*i + [1]
178: (12)               tgt = [i] + [0]*i + [2/scl]
179: (12)               legpol = leg.poly2leg(pol)
180: (12)               legint = leg.legint(legpol, m=1, k=[i], scl=2)
181: (12)               res = leg.leg2poly(legint)
182: (12)               assert_almost_equal(trim(res), trim(tgt))
183: (8)            for i in range(5):
184: (12)               for j in range(2, 5):
185: (16)                 pol = [0]*i + [1]
186: (16)                 tgt = pol[:]
187: (16)                 for k in range(j):
188: (20)                   tgt = leg.legint(tgt, m=1)
189: (16)                   res = leg.legint(pol, m=j)
190: (16)                   assert_almost_equal(trim(res), trim(tgt))
191: (8)            for i in range(5):
192: (12)               for j in range(2, 5):
193: (16)                 pol = [0]*i + [1]
194: (16)                 tgt = pol[:]
195: (16)                 for k in range(j):
196: (20)                   tgt = leg.legint(tgt, m=1, k=[k])
197: (16)                   res = leg.legint(pol, m=j, k=list(range(j)))
198: (16)                   assert_almost_equal(trim(res), trim(tgt))
199: (8)            for i in range(5):
200: (12)               for j in range(2, 5):
201: (16)                 pol = [0]*i + [1]
202: (16)                 tgt = pol[:]
203: (16)                 for k in range(j):
204: (20)                   tgt = leg.legint(tgt, m=1, k=[k], lbnd=-1)
205: (16)                   res = leg.legint(pol, m=j, k=list(range(j)), lbnd=-1)
206: (16)                   assert_almost_equal(trim(res), trim(tgt))
207: (8)            for i in range(5):
208: (12)               for j in range(2, 5):
209: (16)                 pol = [0]*i + [1]
210: (16)                 tgt = pol[:]
211: (16)                 for k in range(j):
212: (20)                   tgt = leg.legint(tgt, m=1, k=[k], scl=2)
213: (16)                   res = leg.legint(pol, m=j, k=list(range(j)), scl=2)
214: (16)                   assert_almost_equal(trim(res), trim(tgt))
215: (4)             def test_legint_axis(self):
216: (8)               c2d = np.random.random((3, 4))
217: (8)               tgt = np.vstack([leg.legint(c) for c in c2d.T]).T
218: (8)               res = leg.legint(c2d, axis=0)
219: (8)               assert_almost_equal(res, tgt)
220: (8)               tgt = np.vstack([leg.legint(c) for c in c2d])
221: (8)               res = leg.legint(c2d, axis=1)
222: (8)               assert_almost_equal(res, tgt)
223: (8)               tgt = np.vstack([leg.legint(c, k=3) for c in c2d])
224: (8)               res = leg.legint(c2d, k=3, axis=1)
225: (8)               assert_almost_equal(res, tgt)
226: (4)             def test_legint_zerointord(self):
227: (8)               assert_equal(leg.legint((1, 2, 3), 0), (1, 2, 3))
228: (0)             class TestDerivative:
229: (4)               def test_legder(self):
230: (8)                 assert_raises(TypeError, leg.legder, [0], .5)
231: (8)                 assert_raises(ValueError, leg.legder, [0], -1)
232: (8)                 for i in range(5):
233: (12)                   tgt = [0]*i + [1]
234: (12)                   res = leg.legder(tgt, m=0)
235: (12)                   assert_equal(trim(res), trim(tgt))
236: (8)                 for i in range(5):
237: (12)                   for j in range(2, 5):
238: (16)                     tgt = [0]*i + [1]

```

```

239: (16)             res = leg.legder(leg.legint(tgt, m=j), m=j)
240: (16)             assert_almost_equal(trim(res), trim(tgt))
241: (8)              for i in range(5):
242: (12)                for j in range(2, 5):
243: (16)                  tgt = [0]*i + [1]
244: (16)                  res = leg.legder(leg.legint(tgt, m=j, scl=2), m=j, scl=.5)
245: (16)                  assert_almost_equal(trim(res), trim(tgt))
246: (4)  def test_legder_axis(self):
247: (8)    c2d = np.random.random((3, 4))
248: (8)    tgt = np.vstack([leg.legder(c) for c in c2d.T]).T
249: (8)    res = leg.legder(c2d, axis=0)
250: (8)    assert_almost_equal(res, tgt)
251: (8)    tgt = np.vstack([leg.legder(c) for c in c2d])
252: (8)    res = leg.legder(c2d, axis=1)
253: (8)    assert_almost_equal(res, tgt)
254: (4)  def test_legder_orderhigherthancoeff(self):
255: (8)    c = (1, 2, 3, 4)
256: (8)    assert_equal(leg.legder(c, 4), [0])
257: (0) class TestVander:
258: (4)  x = np.random.random((3, 5))*2 - 1
259: (4)  def test_legvander(self):
260: (8)    x = np.arange(3)
261: (8)    v = leg.legvander(x, 3)
262: (8)    assert_(v.shape == (3, 4))
263: (8)    for i in range(4):
264: (12)      coef = [0]*i + [1]
265: (12)      assert_almost_equal(v[..., i], leg.legval(x, coef))
266: (8)    x = np.array([[1, 2], [3, 4], [5, 6]])
267: (8)    v = leg.legvander(x, 3)
268: (8)    assert_(v.shape == (3, 2, 4))
269: (8)    for i in range(4):
270: (12)      coef = [0]*i + [1]
271: (12)      assert_almost_equal(v[..., i], leg.legval(x, coef))
272: (4)  def test_legvander2d(self):
273: (8)    x1, x2, x3 = self.x
274: (8)    c = np.random.random((2, 3))
275: (8)    van = leg.legvander2d(x1, x2, [1, 2])
276: (8)    tgt = leg.legval2d(x1, x2, c)
277: (8)    res = np.dot(van, c.flat)
278: (8)    assert_almost_equal(res, tgt)
279: (8)    van = leg.legvander2d([x1], [x2], [1, 2])
280: (8)    assert_(van.shape == (1, 5, 6))
281: (4)  def test_legvander3d(self):
282: (8)    x1, x2, x3 = self.x
283: (8)    c = np.random.random((2, 3, 4))
284: (8)    van = leg.legvander3d(x1, x2, x3, [1, 2, 3])
285: (8)    tgt = leg.legval3d(x1, x2, x3, c)
286: (8)    res = np.dot(van, c.flat)
287: (8)    assert_almost_equal(res, tgt)
288: (8)    van = leg.legvander3d([x1], [x2], [x3], [1, 2, 3])
289: (8)    assert_(van.shape == (1, 5, 24))
290: (4)  def test_legvander_negdeg(self):
291: (8)    assert_raises(ValueError, leg.legvander, (1, 2, 3), -1)
292: (0) class TestFitting:
293: (4)  def test_legfit(self):
294: (8)    def f(x):
295: (12)      return x*(x - 1)*(x - 2)
296: (8)    def f2(x):
297: (12)      return x**4 + x**2 + 1
298: (8)    assert_raises(ValueError, leg.legfit, [1], [1], -1)
299: (8)    assert_raises(TypeError, leg.legfit, [[1]], [1], 0)
300: (8)    assert_raises(TypeError, leg.legfit, [], [1], 0)
301: (8)    assert_raises(TypeError, leg.legfit, [1], [[[1]]], 0)
302: (8)    assert_raises(TypeError, leg.legfit, [1, 2], [1], 0)
303: (8)    assert_raises(TypeError, leg.legfit, [1], [1, 2], 0)
304: (8)    assert_raises(TypeError, leg.legfit, [1], [1], 0, w=[[1]])
305: (8)    assert_raises(TypeError, leg.legfit, [1], [1], 0, w=[1, 1])
306: (8)    assert_raises(ValueError, leg.legfit, [1], [1], [-1,])
307: (8)    assert_raises(ValueError, leg.legfit, [1], [1], [2, -1,])

```

```

308: (8) assert_raises(TypeError, leg.legfit, [1], [1], [])
309: (8) x = np.linspace(0, 2)
310: (8) y = f(x)
311: (8) coef3 = leg.legfit(x, y, 3)
312: (8) assert_equal(len(coef3), 4)
313: (8) assert_almost_equal(leg.legval(x, coef3), y)
314: (8) coef3 = leg.legfit(x, y, [0, 1, 2, 3])
315: (8) assert_equal(len(coef3), 4)
316: (8) assert_almost_equal(leg.legval(x, coef3), y)
317: (8) coef4 = leg.legfit(x, y, 4)
318: (8) assert_equal(len(coef4), 5)
319: (8) assert_almost_equal(leg.legval(x, coef4), y)
320: (8) coef4 = leg.legfit(x, y, [0, 1, 2, 3, 4])
321: (8) assert_equal(len(coef4), 5)
322: (8) assert_almost_equal(leg.legval(x, coef4), y)
323: (8) coef4 = leg.legfit(x, y, [2, 3, 4, 1, 0])
324: (8) assert_equal(len(coef4), 5)
325: (8) assert_almost_equal(leg.legval(x, coef4), y)
326: (8) coef2d = leg.legfit(x, np.array([y, y]).T, 3)
327: (8) assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
328: (8) coef2d = leg.legfit(x, np.array([y, y]).T, [0, 1, 2, 3])
329: (8) assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
330: (8) w = np.zeros_like(x)
331: (8) yw = y.copy()
332: (8) w[1::2] = 1
333: (8) y[0::2] = 0
334: (8) wcoef3 = leg.legfit(x, yw, 3, w=w)
335: (8) assert_almost_equal(wcoef3, coef3)
336: (8) wcoef3 = leg.legfit(x, yw, [0, 1, 2, 3], w=w)
337: (8) assert_almost_equal(wcoef3, coef3)
338: (8) wcoef2d = leg.legfit(x, np.array([yw, yw]).T, 3, w=w)
339: (8) assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
340: (8) wcoef2d = leg.legfit(x, np.array([yw, yw]).T, [0, 1, 2, 3], w=w)
341: (8) assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
342: (8) x = [1, 1j, -1, -1j]
343: (8) assert_almost_equal(leg.legfit(x, x, 1), [0, 1])
344: (8) assert_almost_equal(leg.legfit(x, x, [0, 1]), [0, 1])
345: (8) x = np.linspace(-1, 1)
346: (8) y = f2(x)
347: (8) coef1 = leg.legfit(x, y, 4)
348: (8) assert_almost_equal(leg.legval(x, coef1), y)
349: (8) coef2 = leg.legfit(x, y, [0, 2, 4])
350: (8) assert_almost_equal(leg.legval(x, coef2), y)
351: (8) assert_almost_equal(coef1, coef2)
352: (0) class TestCompanion:
353: (4)     def test_raises(self):
354: (8)         assert_raises(ValueError, leg.legcompanion, [])
355: (8)         assert_raises(ValueError, leg.legcompanion, [1])
356: (4)     def test_dimensions(self):
357: (8)         for i in range(1, 5):
358: (12)             coef = [0]*i + [1]
359: (12)             assert_(leg.legcompanion(coef).shape == (i, i))
360: (4)     def test_linear_root(self):
361: (8)         assert_(leg.legcompanion([1, 2])[0, 0] == -.5)
362: (0) class TestGauss:
363: (4)     def test_100(self):
364: (8)         x, w = leg.leggauss(100)
365: (8)         v = leg.legvander(x, 99)
366: (8)         vv = np.dot(v.T * w, v)
367: (8)         vd = 1/np.sqrt(vv.diagonal())
368: (8)         vv = vd[:, None] * vv * vd
369: (8)         assert_almost_equal(vv, np.eye(100))
370: (8)         tgt = 2.0
371: (8)         assert_almost_equal(w.sum(), tgt)
372: (0) class TestMisc:
373: (4)     def test_legfromroots(self):
374: (8)         res = leg.legfromroots([])
375: (8)         assert_almost_equal(trim(res), [1])
376: (8)         for i in range(1, 5):

```

```

377: (12)           roots = np.cos(np.linspace(-np.pi, 0, 2*i + 1)[1::2])
378: (12)           pol = leg.legfromroots(roots)
379: (12)           res = leg.legval(roots, pol)
380: (12)           tgt = 0
381: (12)           assert_(len(pol) == i + 1)
382: (12)           assert_almost_equal(leg.leg2poly(pol)[-1], 1)
383: (12)           assert_almost_equal(res, tgt)
384: (4)            def test_legroots(self):
385: (8)              assert_almost_equal(leg.legroots([1]), [])
386: (8)              assert_almost_equal(leg.legroots([1, 2]), [-.5])
387: (8)              for i in range(2, 5):
388: (12)                  tgt = np.linspace(-1, 1, i)
389: (12)                  res = leg.legroots(leg.legfromroots(tgt))
390: (12)                  assert_almost_equal(trim(res), trim(tgt))
391: (4)            def test_legtrim(self):
392: (8)              coef = [2, -1, 1, 0]
393: (8)              assert_raises(ValueError, leg.legtrim, coef, -1)
394: (8)              assert_equal(leg.legtrim(coef), coef[:-1])
395: (8)              assert_equal(leg.legtrim(coef, 1), coef[:-3])
396: (8)              assert_equal(leg.legtrim(coef, 2), [0])
397: (4)            def test_lepline(self):
398: (8)              assert_equal(leg.lepline(3, 4), [3, 4])
399: (4)            def test_lepline_zeroscl(self):
400: (8)              assert_equal(leg.lepline(3, 0), [3])
401: (4)            def test_leg2poly(self):
402: (8)              for i in range(10):
403: (12)                  assert_almost_equal(leg.leg2poly([0]*i + [1]), Llist[i])
404: (4)            def test_poly2leg(self):
405: (8)              for i in range(10):
406: (12)                  assert_almost_equal(leg.poly2leg(Llist[i]), [0]*i + [1])
407: (4)            def test_weight(self):
408: (8)              x = np.linspace(-1, 1, 11)
409: (8)              tgt = 1.
410: (8)              res = leg.legweight(x)
411: (8)              assert_almost_equal(res, tgt)

```

---

## File 303 - test\_polyutils.py:

```

1: (0)          """Tests for polyutils module.
2: (0)          """
3: (0)          import numpy as np
4: (0)          import numpy.polynomial.polyutils as pu
5: (0)          from numpy.testing import (
6: (4)              assert_almost_equal, assert_raises, assert_equal, assert_,
7: (4)          )
8: (0)          class TestMisc:
9: (4)              def test_trimseq(self):
10: (8)                  for i in range(5):
11: (12)                      tgt = [1]
12: (12)                      res = pu.trimseq([1] + [0]**5)
13: (12)                      assert_equal(res, tgt)
14: (4)              def test_as_series(self):
15: (8)                  assert_raises(ValueError, pu.as_series, [[]])
16: (8)                  assert_raises(ValueError, pu.as_series, [[[1, 2]]])
17: (8)                  assert_raises(ValueError, pu.as_series, [[1], ['a']])
18: (8)                  types = ['i', 'd', 'O']
19: (8)                  for i in range(len(types)):
20: (12)                      for j in range(i):
21: (16)                          ci = np.ones(1, types[i])
22: (16)                          cj = np.ones(1, types[j])
23: (16)                          [resi, resj] = pu.as_series([ci, cj])
24: (16)                          assert_(resi.dtype.char == resj.dtype.char)
25: (16)                          assert_(resj.dtype.char == types[i])
26: (4)              def test_trimcoef(self):
27: (8)                  coef = [2, -1, 1, 0]
28: (8)                  assert_raises(ValueError, pu.trimcoef, coef, -1)
29: (8)                  assert_equal(pu.trimcoef(coef), coef[:-1])

```

```

30: (8)             assert_equal(pu.trimcoef(coef, 1), coef[:-3])
31: (8)             assert_equal(pu.trimcoef(coef, 2), [0])
32: (4)         def test_vander_nd_exception(self):
33: (8)             assert_raises(ValueError, pu._vander_nd, (), (1, 2, 3), [90])
34: (8)             assert_raises(ValueError, pu._vander_nd, (), (), [90.65])
35: (8)             assert_raises(ValueError, pu._vander_nd, (), (), [])
36: (4)         def test_div_zerodiv(self):
37: (8)             assert_raises(ZeroDivisionError, pu._div, pu._div, (1, 2, 3), [0])
38: (4)         def test_pow_too_large(self):
39: (8)             assert_raises(ValueError, pu._pow, (), [1, 2, 3], 5, 4)
40: (0)     class TestDomain:
41: (4)         def test_getdomain(self):
42: (8)             x = [1, 10, 3, -1]
43: (8)             tgt = [-1, 10]
44: (8)             res = pu.getdomain(x)
45: (8)             assert_almost_equal(res, tgt)
46: (8)             x = [1 + 1j, 1 - 1j, 0, 2]
47: (8)             tgt = [-1j, 2 + 1j]
48: (8)             res = pu.getdomain(x)
49: (8)             assert_almost_equal(res, tgt)
50: (4)         def test_mapdomain(self):
51: (8)             dom1 = [0, 4]
52: (8)             dom2 = [1, 3]
53: (8)             tgt = dom2
54: (8)             res = pu.mapdomain(dom1, dom1, dom2)
55: (8)             assert_almost_equal(res, tgt)
56: (8)             dom1 = [0 - 1j, 2 + 1j]
57: (8)             dom2 = [-2, 2]
58: (8)             tgt = dom2
59: (8)             x = dom1
60: (8)             res = pu.mapdomain(x, dom1, dom2)
61: (8)             assert_almost_equal(res, tgt)
62: (8)             dom1 = [0, 4]
63: (8)             dom2 = [1, 3]
64: (8)             tgt = np.array([dom2, dom2])
65: (8)             x = np.array([dom1, dom1])
66: (8)             res = pu.mapdomain(x, dom1, dom2)
67: (8)             assert_almost_equal(res, tgt)
68: (8)         class MyNDArray(np.ndarray):
69: (12)             pass
70: (8)             dom1 = [0, 4]
71: (8)             dom2 = [1, 3]
72: (8)             x = np.array([dom1, dom1]).view(MyNDArray)
73: (8)             res = pu.mapdomain(x, dom1, dom2)
74: (8)             assert_(isinstance(res, MyNDArray))
75: (4)         def test_mapparms(self):
76: (8)             dom1 = [0, 4]
77: (8)             dom2 = [1, 3]
78: (8)             tgt = [1, .5]
79: (8)             res = pu.mapparms(dom1, dom2)
80: (8)             assert_almost_equal(res, tgt)
81: (8)             dom1 = [0 - 1j, 2 + 1j]
82: (8)             dom2 = [-2, 2]
83: (8)             tgt = [-1 + 1j, 1 - 1j]
84: (8)             res = pu.mapparms(dom1, dom2)
85: (8)             assert_almost_equal(res, tgt)

```

-----

File 304 - test\_polynomial.py:

```

1: (0)         """Tests for polynomial module.
2: (0)         """
3: (0)         from functools import reduce
4: (0)         import numpy as np
5: (0)         import numpy.polynomial.polynomial as poly
6: (0)         import pickle
7: (0)         from copy import deepcopy
8: (0)         from numpy.testing import (

```

```

9: (4) assert_almost_equal, assert_raises, assert_equal, assert_,
10: (4) assert_warns, assert_array_equal, assert_raises_regex)
11: (0) def trim(x):
12: (4)     return poly.polytrim(x, tol=1e-6)
13: (0) T0 = [1]
14: (0) T1 = [0, 1]
15: (0) T2 = [-1, 0, 2]
16: (0) T3 = [0, -3, 0, 4]
17: (0) T4 = [1, 0, -8, 0, 8]
18: (0) T5 = [0, 5, 0, -20, 0, 16]
19: (0) T6 = [-1, 0, 18, 0, -48, 0, 32]
20: (0) T7 = [0, -7, 0, 56, 0, -112, 0, 64]
21: (0) T8 = [1, 0, -32, 0, 160, 0, -256, 0, 128]
22: (0) T9 = [0, 9, 0, -120, 0, 432, 0, -576, 0, 256]
23: (0) Tlist = [T0, T1, T2, T3, T4, T5, T6, T7, T8, T9]
24: (0) class TestConstants:
25: (4)     def test_polydomain(self):
26: (8)         assert_equal(poly.polydomain, [-1, 1])
27: (4)     def test_polyzero(self):
28: (8)         assert_equal(poly.polyzero, [0])
29: (4)     def test_polyone(self):
30: (8)         assert_equal(poly.polyone, [1])
31: (4)     def test_polyx(self):
32: (8)         assert_equal(poly.polyx, [0, 1])
33: (4)     def test_copy(self):
34: (8)         x = poly.Polynomial([1, 2, 3])
35: (8)         y = deepcopy(x)
36: (8)         assert_equal(x, y)
37: (4)     def test_pickle(self):
38: (8)         x = poly.Polynomial([1, 2, 3])
39: (8)         y = pickle.loads(pickle.dumps(x))
40: (8)         assert_equal(x, y)
41: (0) class TestArithmetic:
42: (4)     def test_polyadd(self):
43: (8)         for i in range(5):
44: (12)             for j in range(5):
45: (16)                 msg = f"At i={i}, j={j}"
46: (16)                 tgt = np.zeros(max(i, j) + 1)
47: (16)                 tgt[i] += 1
48: (16)                 tgt[j] += 1
49: (16)                 res = poly.polyadd([0]*i + [1], [0]*j + [1])
50: (16)                 assert_equal(trim(res), trim(tgt), err_msg=msg)
51: (4)     def test_polysub(self):
52: (8)         for i in range(5):
53: (12)             for j in range(5):
54: (16)                 msg = f"At i={i}, j={j}"
55: (16)                 tgt = np.zeros(max(i, j) + 1)
56: (16)                 tgt[i] += 1
57: (16)                 tgt[j] -= 1
58: (16)                 res = poly.polysub([0]*i + [1], [0]*j + [1])
59: (16)                 assert_equal(trim(res), trim(tgt), err_msg=msg)
60: (4)     def test_polymulx(self):
61: (8)         assert_equal(poly.polymulx([0]), [0])
62: (8)         assert_equal(poly.polymulx([1]), [0, 1])
63: (8)         for i in range(1, 5):
64: (12)             ser = [0]*i + [1]
65: (12)             tgt = [0]^(i + 1) + [1]
66: (12)             assert_equal(poly.polymulx(ser), tgt)
67: (4)     def test_polymul(self):
68: (8)         for i in range(5):
69: (12)             for j in range(5):
70: (16)                 msg = f"At i={i}, j={j}"
71: (16)                 tgt = np.zeros(i + j + 1)
72: (16)                 tgt[i + j] += 1
73: (16)                 res = poly.polymul([0]*i + [1], [0]*j + [1])
74: (16)                 assert_equal(trim(res), trim(tgt), err_msg=msg)
75: (4)     def test_polydiv(self):
76: (8)         assert_raises(ZeroDivisionError, poly.polydiv, [1], [0])
77: (8)         quo, rem = poly.polydiv([2], [2])

```

```

78: (8)             assert_equal((quo, rem), (1, 0))
79: (8)             quo, rem = poly.polydiv([2, 2], [2])
80: (8)             assert_equal((quo, rem), ((1, 1), 0))
81: (8)             for i in range(5):
82: (12)                 for j in range(5):
83: (16)                     msg = f"At i={i}, j={j}"
84: (16)                     ci = [0]*i + [1, 2]
85: (16)                     cj = [0]*j + [1, 2]
86: (16)                     tgt = poly.polyadd(ci, cj)
87: (16)                     quo, rem = poly.polydiv(tgt, ci)
88: (16)                     res = poly.polyadd(poly.polymul(quo, ci), rem)
89: (16)                     assert_equal(res, tgt, err_msg=msg)
90: (4)             def test_polypow(self):
91: (8)                 for i in range(5):
92: (12)                     for j in range(5):
93: (16)                         msg = f"At i={i}, j={j}"
94: (16)                         c = np.arange(i + 1)
95: (16)                         tgt = reduce(poly.polymul, [c]*j, np.array([1]))
96: (16)                         res = poly.polypow(c, j)
97: (16)                         assert_equal(trim(res), trim(tgt), err_msg=msg)
98: (0)             class TestEvaluation:
99: (4)                 c1d = np.array([1., 2., 3.])
100: (4)                c2d = np.einsum('i,j->ij', c1d, c1d)
101: (4)                c3d = np.einsum('i,j,k->ijk', c1d, c1d, c1d)
102: (4)                x = np.random.random((3, 5))*2 - 1
103: (4)                y = poly.polyval(x, [1., 2., 3.])
104: (4)                def test_polyval(self):
105: (8)                    assert_equal(poly.polyval([], [1]).size, 0)
106: (8)                    x = np.linspace(-1, 1)
107: (8)                    y = [x**i for i in range(5)]
108: (8)                    for i in range(5):
109: (12)                        tgt = y[i]
110: (12)                        res = poly.polyval(x, [0]*i + [1])
111: (12)                        assert_almost_equal(res, tgt)
112: (8)                    tgt = x*(x**2 - 1)
113: (8)                    res = poly.polyval(x, [0, -1, 0, 1])
114: (8)                    assert_almost_equal(res, tgt)
115: (8)                    for i in range(3):
116: (12)                        dims = [2]*i
117: (12)                        x = np.zeros(dims)
118: (12)                        assert_equal(poly.polyval(x, [1]).shape, dims)
119: (12)                        assert_equal(poly.polyval(x, [1, 0]).shape, dims)
120: (12)                        assert_equal(poly.polyval(x, [1, 0, 0]).shape, dims)
121: (8)                    mask = [False, True, False]
122: (8)                    mx = np.ma.array([1, 2, 3], mask=mask)
123: (8)                    res = np.polyval([7, 5, 3], mx)
124: (8)                    assert_array_equal(res.mask, mask)
125: (8)                    class C(np.ndarray):
126: (12)                        pass
127: (8)                        cx = np.array([1, 2, 3]).view(C)
128: (8)                        assert_equal(type(np.polyval([2, 3, 4], cx)), C)
129: (4)                def test_polyvalfromroots(self):
130: (8)                    assert_raises(ValueError, poly.polyvalfromroots,
131: (22)                        [1], [1], tensor=False)
132: (8)                    assert_equal(poly.polyvalfromroots([], [1]).size, 0)
133: (8)                    assert_(poly.polyvalfromroots([], [1]).shape == (0,))
134: (8)                    assert_equal(poly.polyvalfromroots([], [[1] * 5]).size, 0)
135: (8)                    assert_(poly.polyvalfromroots([], [[1] * 5]).shape == (5, 0))
136: (8)                    assert_equal(poly.polyvalfromroots(1, 1), 0)
137: (8)                    assert_(poly.polyvalfromroots(1, np.ones((3, 3))).shape == (3,))
138: (8)                    x = np.linspace(-1, 1)
139: (8)                    y = [x**i for i in range(5)]
140: (8)                    for i in range(1, 5):
141: (12)                        tgt = y[i]
142: (12)                        res = poly.polyvalfromroots(x, [0]*i)
143: (12)                        assert_almost_equal(res, tgt)
144: (8)                    tgt = x*(x - 1)*(x + 1)
145: (8)                    res = poly.polyvalfromroots(x, [-1, 0, 1])
146: (8)                    assert_almost_equal(res, tgt)

```

```

147: (8)             for i in range(3):
148: (12)            dims = [2]*i
149: (12)            x = np.zeros(dims)
150: (12)            assert_equal(poly.polyvalfromroots(x, [1]).shape, dims)
151: (12)            assert_equal(poly.polyvalfromroots(x, [1, 0]).shape, dims)
152: (12)            assert_equal(poly.polyvalfromroots(x, [1, 0, 0]).shape, dims)
153: (8)             ptest = [15, 2, -16, -2, 1]
154: (8)             r = poly.polyroots(ptest)
155: (8)             x = np.linspace(-1, 1)
156: (8)             assert_almost_equal(poly.polyval(x, ptest),
157: (28)                           poly.polyvalfromroots(x, r))
158: (8)             rshape = (3, 5)
159: (8)             x = np.arange(-3, 2)
160: (8)             r = np.random.randint(-5, 5, size=rshape)
161: (8)             res = poly.polyvalfromroots(x, r, tensor=False)
162: (8)             tgt = np.empty(r.shape[1:])
163: (8)             for ii in range(tgt.size):
164: (12)               tgt[ii] = poly.polyvalfromroots(x[ii], r[:, ii])
165: (8)             assert_equal(res, tgt)
166: (8)             x = np.vstack([x, 2*x])
167: (8)             res = poly.polyvalfromroots(x, r, tensor=True)
168: (8)             tgt = np.empty(r.shape[1:] + x.shape)
169: (8)             for ii in range(r.shape[1]):
170: (12)               for jj in range(x.shape[0]):
171: (16)                 tgt[ii, jj, :] = poly.polyvalfromroots(x[jj], r[:, ii])
172: (8)             assert_equal(res, tgt)
173: (4)              def test_polyval2d(self):
174: (8)                x1, x2, x3 = self.x
175: (8)                y1, y2, y3 = self.y
176: (8)                assert_raises_regex(ValueError, 'incompatible',
177: (28)                               poly.polyval2d, x1, x2[:2], self.c2d)
178: (8)                tgt = y1*y2
179: (8)                res = poly.polyval2d(x1, x2, self.c2d)
180: (8)                assert_almost_equal(res, tgt)
181: (8)                z = np.ones((2, 3))
182: (8)                res = poly.polyval2d(z, z, self.c2d)
183: (8)                assert_(res.shape == (2, 3))
184: (4)              def test_polyval3d(self):
185: (8)                x1, x2, x3 = self.x
186: (8)                y1, y2, y3 = self.y
187: (8)                assert_raises_regex(ValueError, 'incompatible',
188: (22)                               poly.polyval3d, x1, x2, x3[:2], self.c3d)
189: (8)                tgt = y1*y2*y3
190: (8)                res = poly.polyval3d(x1, x2, x3, self.c3d)
191: (8)                assert_almost_equal(res, tgt)
192: (8)                z = np.ones((2, 3))
193: (8)                res = poly.polyval3d(z, z, z, self.c3d)
194: (8)                assert_(res.shape == (2, 3))
195: (4)              def test_polygrid2d(self):
196: (8)                x1, x2, x3 = self.x
197: (8)                y1, y2, y3 = self.y
198: (8)                tgt = np.einsum('i,j->ij', y1, y2)
199: (8)                res = poly.polygrid2d(x1, x2, self.c2d)
200: (8)                assert_almost_equal(res, tgt)
201: (8)                z = np.ones((2, 3))
202: (8)                res = poly.polygrid2d(z, z, self.c2d)
203: (8)                assert_(res.shape == (2, 3)**2)
204: (4)              def test_polygrid3d(self):
205: (8)                x1, x2, x3 = self.x
206: (8)                y1, y2, y3 = self.y
207: (8)                tgt = np.einsum('i,j,k->ijk', y1, y2, y3)
208: (8)                res = poly.polygrid3d(x1, x2, x3, self.c3d)
209: (8)                assert_almost_equal(res, tgt)
210: (8)                z = np.ones((2, 3))
211: (8)                res = poly.polygrid3d(z, z, z, self.c3d)
212: (8)                assert_(res.shape == (2, 3)**3)
213: (0)              class TestIntegral:
214: (4)                def test_polyint(self):
215: (8)                  assert_raises(TypeError, poly.polyint, [0], .5)

```

```

216: (8) assert_raises(ValueError, poly.polyint, [0], -1)
217: (8) assert_raises(ValueError, poly.polyint, [0], 1, [0, 0])
218: (8) assert_raises(ValueError, poly.polyint, [0], lbnd=[0])
219: (8) assert_raises(ValueError, poly.polyint, [0], scl=[0])
220: (8) assert_raises(TypeError, poly.polyint, [0], axis=.5)
221: (8) with assert_warns(DeprecationWarning):
222: (12)     poly.polyint([1, 1], 1.)
223: (8) for i in range(2, 5):
224: (12)     k = [0]*(i - 2) + [1]
225: (12)     res = poly.polyint([0], m=i, k=k)
226: (12)     assert_almost_equal(res, [0, 1])
227: (8) for i in range(5):
228: (12)     scl = i + 1
229: (12)     pol = [0]*i + [1]
230: (12)     tgt = [i] + [0]*i + [1/scl]
231: (12)     res = poly.polyint(pol, m=1, k=[i])
232: (12)     assert_almost_equal(trim(res), trim(tgt))
233: (8) for i in range(5):
234: (12)     scl = i + 1
235: (12)     pol = [0]*i + [1]
236: (12)     res = poly.polyint(pol, m=1, k=[i], lbnd=-1)
237: (12)     assert_almost_equal(poly.polyval(-1, res), i)
238: (8) for i in range(5):
239: (12)     scl = i + 1
240: (12)     pol = [0]*i + [1]
241: (12)     tgt = [i] + [0]*i + [2/scl]
242: (12)     res = poly.polyint(pol, m=1, k=[i], scl=2)
243: (12)     assert_almost_equal(trim(res), trim(tgt))
244: (8) for i in range(5):
245: (12)     for j in range(2, 5):
246: (16)         pol = [0]*i + [1]
247: (16)         tgt = pol[:]
248: (16)         for k in range(j):
249: (20)             tgt = poly.polyint(tgt, m=1)
250: (16)         res = poly.polyint(pol, m=j)
251: (16)         assert_almost_equal(trim(res), trim(tgt))
252: (8) for i in range(5):
253: (12)     for j in range(2, 5):
254: (16)         pol = [0]*i + [1]
255: (16)         tgt = pol[:]
256: (16)         for k in range(j):
257: (20)             tgt = poly.polyint(tgt, m=1, k=[k])
258: (16)         res = poly.polyint(pol, m=j, k=list(range(j)))
259: (16)         assert_almost_equal(trim(res), trim(tgt))
260: (8) for i in range(5):
261: (12)     for j in range(2, 5):
262: (16)         pol = [0]*i + [1]
263: (16)         tgt = pol[:]
264: (16)         for k in range(j):
265: (20)             tgt = poly.polyint(tgt, m=1, k=[k], lbnd=-1)
266: (16)         res = poly.polyint(pol, m=j, k=list(range(j)), lbnd=-1)
267: (16)         assert_almost_equal(trim(res), trim(tgt))
268: (8) for i in range(5):
269: (12)     for j in range(2, 5):
270: (16)         pol = [0]*i + [1]
271: (16)         tgt = pol[:]
272: (16)         for k in range(j):
273: (20)             tgt = poly.polyint(tgt, m=1, k=[k], scl=2)
274: (16)         res = poly.polyint(pol, m=j, k=list(range(j)), scl=2)
275: (16)         assert_almost_equal(trim(res), trim(tgt))
276: (4) def test_polyint_axis(self):
277: (8)     c2d = np.random.random((3, 4))
278: (8)     tgt = np.vstack([poly.polyint(c) for c in c2d.T]).T
279: (8)     res = poly.polyint(c2d, axis=0)
280: (8)     assert_almost_equal(res, tgt)
281: (8)     tgt = np.vstack([poly.polyint(c) for c in c2d])
282: (8)     res = poly.polyint(c2d, axis=1)
283: (8)     assert_almost_equal(res, tgt)
284: (8)     tgt = np.vstack([poly.polyint(c, k=3) for c in c2d])

```

```

285: (8)             res = poly.polyint(c2d, k=3, axis=1)
286: (8)             assert_almost_equal(res, tgt)
287: (0) class TestDerivative:
288: (4)     def test_polyder(self):
289: (8)         assert_raises(TypeError, poly.polyder, [0], .5)
290: (8)         assert_raises(ValueError, poly.polyder, [0], -1)
291: (8)         for i in range(5):
292: (12)             tgt = [0]*i + [1]
293: (12)             res = poly.polyder(tgt, m=0)
294: (12)             assert_equal(trim(res), trim(tgt))
295: (8)         for i in range(5):
296: (12)             for j in range(2, 5):
297: (16)                 tgt = [0]*i + [1]
298: (16)                 res = poly.polyder(poly.polyint(tgt, m=j), m=j)
299: (16)                 assert_almost_equal(trim(res), trim(tgt))
300: (8)         for i in range(5):
301: (12)             for j in range(2, 5):
302: (16)                 tgt = [0]*i + [1]
303: (16)                 res = poly.polyder(poly.polyint(tgt, m=j, scl=2), m=j, scl=.5)
304: (16)                 assert_almost_equal(trim(res), trim(tgt))
305: (4)     def test_polyder_axis(self):
306: (8)         c2d = np.random.random((3, 4))
307: (8)         tgt = np.vstack([poly.polyder(c) for c in c2d.T]).T
308: (8)         res = poly.polyder(c2d, axis=0)
309: (8)         assert_almost_equal(res, tgt)
310: (8)         tgt = np.vstack([poly.polyder(c) for c in c2d])
311: (8)         res = poly.polyder(c2d, axis=1)
312: (8)         assert_almost_equal(res, tgt)
313: (0) class TestVander:
314: (4)     x = np.random.random((3, 5))*2 - 1
315: (4)     def test_polyvander(self):
316: (8)         x = np.arange(3)
317: (8)         v = poly.polyvander(x, 3)
318: (8)         assert_(v.shape == (3, 4))
319: (8)         for i in range(4):
320: (12)             coef = [0]*i + [1]
321: (12)             assert_almost_equal(v[..., i], poly.polyval(x, coef))
322: (8)         x = np.array([[1, 2], [3, 4], [5, 6]])
323: (8)         v = poly.polyvander(x, 3)
324: (8)         assert_(v.shape == (3, 2, 4))
325: (8)         for i in range(4):
326: (12)             coef = [0]*i + [1]
327: (12)             assert_almost_equal(v[..., i], poly.polyval(x, coef))
328: (4)     def test_polyvander2d(self):
329: (8)         x1, x2, x3 = self.x
330: (8)         c = np.random.random((2, 3))
331: (8)         van = poly.polyvander2d(x1, x2, [1, 2])
332: (8)         tgt = poly.polyval2d(x1, x2, c)
333: (8)         res = np.dot(van, c.flat)
334: (8)         assert_almost_equal(res, tgt)
335: (8)         van = poly.polyvander2d([x1], [x2], [1, 2])
336: (8)         assert_(van.shape == (1, 5, 6))
337: (4)     def test_polyvander3d(self):
338: (8)         x1, x2, x3 = self.x
339: (8)         c = np.random.random((2, 3, 4))
340: (8)         van = poly.polyvander3d(x1, x2, x3, [1, 2, 3])
341: (8)         tgt = poly.polyval3d(x1, x2, x3, c)
342: (8)         res = np.dot(van, c.flat)
343: (8)         assert_almost_equal(res, tgt)
344: (8)         van = poly.polyvander3d([x1], [x2], [x3], [1, 2, 3])
345: (8)         assert_(van.shape == (1, 5, 24))
346: (4)     def test_polyvandernegdeg(self):
347: (8)         x = np.arange(3)
348: (8)         assert_raises(ValueError, poly.polyvander, x, -1)
349: (0) class TestCompanion:
350: (4)     def test_raises(self):
351: (8)         assert_raises(ValueError, poly.polycompanion, [])
352: (8)         assert_raises(ValueError, poly.polycompanion, [1])
353: (4)     def test_dimensions(self):

```

```

354: (8)             for i in range(1, 5):
355: (12)            coef = [0]*i + [1]
356: (12)            assert_(poly.polycompanion(coef).shape == (i, i))
357: (4)             def test_linear_root(self):
358: (8)               assert_(poly.polycompanion([1, 2])[0, 0] == -.5)
359: (0)             class TestMisc:
360: (4)               def test_polyfromroots(self):
361: (8)                 res = poly.polyfromroots([])
362: (8)                 assert_almost_equal(trim(res), [1])
363: (8)                 for i in range(1, 5):
364: (12)                   roots = np.cos(np.linspace(-np.pi, 0, 2*i + 1)[1::2])
365: (12)                   tgt = Tlist[i]
366: (12)                   res = poly.polyfromroots(roots)*2***(i-1)
367: (12)                   assert_almost_equal(trim(res), trim(tgt))
368: (4)               def test_polyroots(self):
369: (8)                 assert_almost_equal(poly.polyroots([1]), [])
370: (8)                 assert_almost_equal(poly.polyroots([1, 2]), [-.5])
371: (8)                 for i in range(2, 5):
372: (12)                   tgt = np.linspace(-1, 1, i)
373: (12)                   res = poly.polyroots(poly.polyfromroots(tgt))
374: (12)                   assert_almost_equal(trim(res), trim(tgt))
375: (4)               def test_polyfit(self):
376: (8)                 def f(x):
377: (12)                   return x*(x - 1)*(x - 2)
378: (8)                 def f2(x):
379: (12)                   return x**4 + x**2 + 1
380: (8)                 assert_raises(ValueError, poly.polyfit, [1], [1], -1)
381: (8)                 assert_raises(TypeError, poly.polyfit, [[1]], [1], 0)
382: (8)                 assert_raises(TypeError, poly.polyfit, [], [1], 0)
383: (8)                 assert_raises(TypeError, poly.polyfit, [1], [[[1]]], 0)
384: (8)                 assert_raises(TypeError, poly.polyfit, [1, 2], [1], 0)
385: (8)                 assert_raises(TypeError, poly.polyfit, [1], [1, 2], 0)
386: (8)                 assert_raises(TypeError, poly.polyfit, [1], [1], 0, w=[[1]])
387: (8)                 assert_raises(TypeError, poly.polyfit, [1], [1], 0, w=[1, 1])
388: (8)                 assert_raises(ValueError, poly.polyfit, [1], [1], [-1,])
389: (8)                 assert_raises(ValueError, poly.polyfit, [1], [1], [2, -1, 6])
390: (8)                 assert_raises(TypeError, poly.polyfit, [1], [1], [])
391: (8)                 x = np.linspace(0, 2)
392: (8)                 y = f(x)
393: (8)                 coef3 = poly.polyfit(x, y, 3)
394: (8)                 assert_equal(len(coef3), 4)
395: (8)                 assert_almost_equal(poly.polyval(x, coef3), y)
396: (8)                 coef3 = poly.polyfit(x, y, [0, 1, 2, 3])
397: (8)                 assert_equal(len(coef3), 4)
398: (8)                 assert_almost_equal(poly.polyval(x, coef3), y)
399: (8)                 coef4 = poly.polyfit(x, y, 4)
400: (8)                 assert_equal(len(coef4), 5)
401: (8)                 assert_almost_equal(poly.polyval(x, coef4), y)
402: (8)                 coef4 = poly.polyfit(x, y, [0, 1, 2, 3, 4])
403: (8)                 assert_equal(len(coef4), 5)
404: (8)                 assert_almost_equal(poly.polyval(x, coef4), y)
405: (8)                 coef2d = poly.polyfit(x, np.array([y, y]).T, 3)
406: (8)                 assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
407: (8)                 coef2d = poly.polyfit(x, np.array([y, y]).T, [0, 1, 2, 3])
408: (8)                 assert_almost_equal(coef2d, np.array([coef3, coef3]).T)
409: (8)                 w = np.zeros_like(x)
410: (8)                 yw = y.copy()
411: (8)                 w[1::2] = 1
412: (8)                 yw[0::2] = 0
413: (8)                 wcoef3 = poly.polyfit(x, yw, 3, w=w)
414: (8)                 assert_almost_equal(wcoef3, coef3)
415: (8)                 wcoef3 = poly.polyfit(x, yw, [0, 1, 2, 3], w=w)
416: (8)                 assert_almost_equal(wcoef3, coef3)
417: (8)                 wcoef2d = poly.polyfit(x, np.array([yw, yw]).T, 3, w=w)
418: (8)                 assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
419: (8)                 wcoef2d = poly.polyfit(x, np.array([yw, yw]).T, [0, 1, 2, 3], w=w)
420: (8)                 assert_almost_equal(wcoef2d, np.array([coef3, coef3]).T)
421: (8)                 x = [1, 1j, -1, -1j]
422: (8)                 assert_almost_equal(poly.polyfit(x, x, 1), [0, 1])

```

```

423: (8)             assert_almost_equal(poly.polyfit(x, x, [0, 1]), [0, 1])
424: (8)             x = np.linspace(-1, 1)
425: (8)             y = f2(x)
426: (8)             coef1 = poly.polyfit(x, y, 4)
427: (8)             assert_almost_equal(poly.polyval(x, coef1), y)
428: (8)             coef2 = poly.polyfit(x, y, [0, 2, 4])
429: (8)             assert_almost_equal(poly.polyval(x, coef2), y)
430: (8)             assert_almost_equal(coef1, coef2)
431: (4)             def test_polytrim(self):
432: (8)                 coef = [2, -1, 1, 0]
433: (8)                 assert_raises(ValueError, poly.polytrim, coef, -1)
434: (8)                 assert_equal(poly.polytrim(coef), coef[:-1])
435: (8)                 assert_equal(poly.polytrim(coef, 1), coef[:-3])
436: (8)                 assert_equal(poly.polytrim(coef, 2), [0])
437: (4)             def test_polyline(self):
438: (8)                 assert_equal(poly.polyline(3, 4), [3, 4])
439: (4)             def test_polyline_zero(self):
440: (8)                 assert_equal(poly.polyline(3, 0), [3])

```

---

## File 305 - test\_symbol.py:

```

1: (0)             """
2: (0)             Tests related to the ``symbol`` attribute of the ABCPolyBase class.
3: (0)             """
4: (0)             import pytest
5: (0)             import numpy.polynomial as poly
6: (0)             from numpy.core import array
7: (0)             from numpy.testing import assert_equal, assert_raises, assert_
8: (0)             class TestInit:
9: (4)                 """
10: (4)                 Test polynomial creation with symbol kwarg.
11: (4)                 """
12: (4)                 c = [1, 2, 3]
13: (4)             def test_default_symbol(self):
14: (8)                 p = poly.Polynomial(self.c)
15: (8)                 assert_equal(p.symbol, 'x')
16: (4)             @pytest.mark.parametrize(('bad_input', 'exception'), (
17: (8)                 ('', ValueError),
18: (8)                 ('3', ValueError),
19: (8)                 (None, TypeError),
20: (8)                 (1, TypeError),
21: (4)             ))
22: (4)             def test_symbol_bad_input(self, bad_input, exception):
23: (8)                 with pytest.raises(exception):
24: (12)                     p = poly.Polynomial(self.c, symbol=bad_input)
25: (4)             @pytest.mark.parametrize('symbol', (
26: (8)                 'x',
27: (8)                 'x_1',
28: (8)                 'A',
29: (8)                 'xyz',
30: (8)                 'I^2',
31: (4)             ))
32: (4)             def test_valid_symbols(self, symbol):
33: (8)                 """
34: (8)                 Values for symbol that should pass input validation.
35: (8)                 """
36: (8)                 p = poly.Polynomial(self.c, symbol=symbol)
37: (8)                 assert_equal(p.symbol, symbol)
38: (4)             def test_property(self):
39: (8)                 """
40: (8)                 'symbol' attribute is read only.
41: (8)                 """
42: (8)                 p = poly.Polynomial(self.c, symbol='x')
43: (8)                 with pytest.raises(AttributeError):
44: (12)                     p.symbol = 'z'
45: (4)             def test_change_symbol(self):
46: (8)                 p = poly.Polynomial(self.c, symbol='y')

```

```

47: (8)             pt = poly.Polynomial(p.coef, symbol='t')
48: (8)             assert_equal(pt.symbol, 't')
49: (0) class TestUnaryOperators:
50: (4)             p = poly.Polynomial([1, 2, 3], symbol='z')
51: (4)             def test_neg(self):
52: (8)                 n = -self.p
53: (8)                 assert_equal(n.symbol, 'z')
54: (4)             def test_scalarmul(self):
55: (8)                 out = self.p * 10
56: (8)                 assert_equal(out.symbol, 'z')
57: (4)             def test_rscalarmul(self):
58: (8)                 out = 10 * self.p
59: (8)                 assert_equal(out.symbol, 'z')
60: (4)             def test_pow(self):
61: (8)                 out = self.p ** 3
62: (8)                 assert_equal(out.symbol, 'z')
63: (0) @pytest.mark.parametrize(
64: (4)     'rhs',
65: (4)     (
66: (8)         poly.Polynomial([4, 5, 6], symbol='z'),
67: (8)         array([4, 5, 6]),
68: (4)     ),
69: (0)
70: (0) class TestBinaryOperatorsSameSymbol:
71: (4)     """
72: (4)         Ensure symbol is preserved for numeric operations on polynomials with
73: (4)         the same symbol
74: (4)     """
75: (4)     p = poly.Polynomial([1, 2, 3], symbol='z')
76: (4)     def test_add(self, rhs):
77: (8)         out = self.p + rhs
78: (8)         assert_equal(out.symbol, 'z')
79: (4)     def test_sub(self, rhs):
80: (8)         out = self.p - rhs
81: (8)         assert_equal(out.symbol, 'z')
82: (4)     def test_polymul(self, rhs):
83: (8)         out = self.p * rhs
84: (8)         assert_equal(out.symbol, 'z')
85: (4)     def test_divmod(self, rhs):
86: (8)         for out in divmod(self.p, rhs):
87: (12)             assert_equal(out.symbol, 'z')
88: (4)     def test_radd(self, rhs):
89: (8)         out = rhs + self.p
90: (8)         assert_equal(out.symbol, 'z')
91: (4)     def test_rsub(self, rhs):
92: (8)         out = rhs - self.p
93: (8)         assert_equal(out.symbol, 'z')
94: (4)     def test_rmul(self, rhs):
95: (8)         out = rhs * self.p
96: (8)         assert_equal(out.symbol, 'z')
97: (4)     def test_rdivmod(self, rhs):
98: (8)         for out in divmod(rhs, self.p):
99: (12)             assert_equal(out.symbol, 'z')
100: (0) class TestBinaryOperatorsDifferentSymbol:
101: (4)             p = poly.Polynomial([1, 2, 3], symbol='x')
102: (4)             other = poly.Polynomial([4, 5, 6], symbol='y')
103: (4)             ops = (p.__add__, p.__sub__, p.__mul__, p.__floordiv__, p.__mod__)
104: (4)             @pytest.mark.parametrize('f', ops)
105: (4)             def test_binops_fails(self, f):
106: (8)                 assert_raises(ValueError, f, self.other)
107: (0) class TestEquality:
108: (4)             p = poly.Polynomial([1, 2, 3], symbol='x')
109: (4)             def test_eq(self):
110: (8)                 other = poly.Polynomial([1, 2, 3], symbol='x')
111: (8)                 assert_(self.p == other)
112: (4)             def test_neq(self):
113: (8)                 other = poly.Polynomial([1, 2, 3], symbol='y')
114: (8)                 assert_(not self.p == other)
115: (0) class TestExtraMethods:

```

```

116: (4)
117: (4)     """
118: (4)     Test other methods for manipulating/creating polynomial objects.
119: (4)
120: (4)     """
121: (8)     p = poly.Polynomial([1, 2, 3, 0], symbol='z')
122: (8)     def test_copy(self):
123: (4)         other = self.p.copy()
124: (8)         assert_equal(other.symbol, 'z')
125: (8)     def test_trim(self):
126: (4)         other = self.p.trim()
127: (8)         assert_equal(other.symbol, 'z')
128: (8)     def test_truncate(self):
129: (4)         other = self.p.truncate(2)
130: (8)         assert_equal(other.symbol, 'z')
131: (8)     @pytest.mark.parametrize('kwarg', (
132: (8)         {'domain': [-10, 10]},
133: (8)         {'window': [-10, 10]},
134: (8)         {'kind': poly.Chebyshev},
135: (4)     ))
136: (8)     def test_convert(self, kwarg):
137: (4)         other = self.p.convert(**kwarg)
138: (8)         assert_equal(other.symbol, 'z')
139: (8)     def test_integ(self):
140: (4)         other = self.p.integ()
141: (8)         assert_equal(other.symbol, 'z')
142: (8)     def test_deriv(self):
143: (0)         other = self.p.deriv()
144: (4)         assert_equal(other.symbol, 'z')
145: (4)     def test_composition():
146: (4)         p = poly.Polynomial([3, 2, 1], symbol="t")
147: (4)         q = poly.Polynomial([5, 1, 0, -1], symbol="t")
148: (0)         r = p(q)
149: (4)         assert r.symbol == "t"
150: (4)     def test_fit():
151: (4)         x, y = (range(10),)*2
152: (0)         p = poly.Polynomial.fit(x, y, deg=1, symbol='z')
153: (4)         assert_equal(p.symbol, 'z')
154: (4)     def test_fromroots():
155: (4)         roots = [-2, 2]
156: (0)         p = poly.Polynomial.fromroots(roots, symbol='z')
157: (4)         assert_equal(p.symbol, 'z')
158: (4)     def test_identity():
159: (0)         p = poly.Polynomial.identity(domain=[-1, 1], window=[5, 20], symbol='z')
160: (4)         assert_equal(p.symbol, 'z')
161: (4)     def test_basis():
162: (4)         p = poly.Polynomial.basis(3, symbol='z')
163: (4)         assert_equal(p.symbol, 'z')

```

---

## File 306 - test\_printing.py:

```

1: (0)         from math import nan, inf
2: (0)         import pytest
3: (0)         from numpy.core import array, arange, printoptions
4: (0)         import numpy.polynomial as poly
5: (0)         from numpy.testing import assert_equal, assert_
6: (0)         from fractions import Fraction
7: (0)         from decimal import Decimal
8: (0)         class TestStrUnicodeSuperSubscripts:
9: (4)             @pytest.fixture(scope='class', autouse=True)
10: (4)             def use_unicode(self):
11: (8)                 poly.set_default_printstyle('unicode')
12: (4)             @pytest.mark.parametrize(('inp', 'tgt'), (
13: (8)                 ([1, 2, 3], "1.0 + 2.0·x + 3.0·x²"),
14: (8)                ([-1, 0, 3, -1], "-1.0 + 0.0·x + 3.0·x² - 1.0·x³"),
15: (8)                 (arange(12), ("0.0 + 1.0·x + 2.0·x² + 3.0·x³ + 4.0·x⁴ + "
5.0·x⁵ + "
16: (22)                                     "6.0·x⁶ + 7.0·x⁷. + \n8.0·x⁸, + 9.0·x⁹ + "
10.0·x¹⁰ + "

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

17: (22)                                "11.0·x· $\hat{x}$ "),
18: (4) )
19: (4)     def test_polynomial_str(self, inp, tgt):
20: (8)         res = str(poly.Polynomial(inp))
21: (8)         assert_equal(res, tgt)
22: (4) @pytest.mark.parametrize(('inp', 'tgt'), (
23: (8)     ([1, 2, 3], "1.0 + 2.0·T· $\hat{x}$ (x) + 3.0·T· $\hat{x}$ ,,(x)"),
24: (8)     ([-1, 0, 3, -1], "-1.0 + 0.0·T· $\hat{x}$ (x) + 3.0·T· $\hat{x}$ ,,(x) - 1.0·T· $\hat{x}$ ,f(x)"),
25: (8)     (arange(12), ("0.0 + 1.0·T· $\hat{x}$ (x) + 2.0·T· $\hat{x}$ ,,(x) + 3.0·T· $\hat{x}$ ,f(x) +
4.0·T· $\hat{x}$ ,,,(x) + "
26: (22)                                         "5.0·T· $\hat{x}$ ,...(x) +\n6.0·T· $\hat{x}$ ,†(x) + 7.0·T· $\hat{x}$ ,‡(x) +
8.0·T· $\hat{x}$ ,^(x) + "
27: (22)                                         "9.0·T· $\hat{x}$ ,‰(x) + 10.0·T· $\hat{x}$ ,¤(x) + 11.0·T· $\hat{x}$ ,¤¤(x")),
28: (4) ))
29: (4)     def test_chebyshev_str(self, inp, tgt):
30: (8)         res = str(poly.Chebyshev(inp))
31: (8)         assert_equal(res, tgt)
32: (4) @pytest.mark.parametrize(('inp', 'tgt'), (
33: (8)     ([1, 2, 3], "1.0 + 2.0·P· $\hat{x}$ (x) + 3.0·P· $\hat{x}$ ,,(x)"),
34: (8)     ([-1, 0, 3, -1], "-1.0 + 0.0·P· $\hat{x}$ (x) + 3.0·P· $\hat{x}$ ,,(x) - 1.0·P· $\hat{x}$ ,f(x)"),
35: (8)     (arange(12), ("0.0 + 1.0·P· $\hat{x}$ (x) + 2.0·P· $\hat{x}$ ,,(x) + 3.0·P· $\hat{x}$ ,f(x) +
4.0·P· $\hat{x}$ ,,,(x) + "
36: (22)                                         "5.0·P· $\hat{x}$ ,...(x) +\n6.0·P· $\hat{x}$ ,†(x) + 7.0·P· $\hat{x}$ ,‡(x) +
8.0·P· $\hat{x}$ ,^(x) + "
37: (22)                                         "9.0·P· $\hat{x}$ ,‰(x) + 10.0·P· $\hat{x}$ ,¤(x) + 11.0·P· $\hat{x}$ ,¤¤(x")),
38: (4) ))
39: (4)     def test_legendre_str(self, inp, tgt):
40: (8)         res = str(poly.Legendre(inp))
41: (8)         assert_equal(res, tgt)
42: (4) @pytest.mark.parametrize(('inp', 'tgt'), (
43: (8)     ([1, 2, 3], "1.0 + 2.0·H· $\hat{x}$ (x) + 3.0·H· $\hat{x}$ ,,(x)"),
44: (8)     ([-1, 0, 3, -1], "-1.0 + 0.0·H· $\hat{x}$ (x) + 3.0·H· $\hat{x}$ ,,(x) - 1.0·H· $\hat{x}$ ,f(x)"),
45: (8)     (arange(12), ("0.0 + 1.0·H· $\hat{x}$ (x) + 2.0·H· $\hat{x}$ ,,(x) + 3.0·H· $\hat{x}$ ,f(x) +
4.0·H· $\hat{x}$ ,,,(x) + "
46: (22)                                         "5.0·H· $\hat{x}$ ,...(x) +\n6.0·H· $\hat{x}$ ,†(x) + 7.0·H· $\hat{x}$ ,‡(x) +
8.0·H· $\hat{x}$ ,^(x) + "
47: (22)                                         "9.0·H· $\hat{x}$ ,‰(x) + 10.0·H· $\hat{x}$ ,¤(x) + 11.0·H· $\hat{x}$ ,¤¤(x")),
48: (4) ))
49: (4)     def test_hermite_str(self, inp, tgt):
50: (8)         res = str(poly.Hermite(inp))
51: (8)         assert_equal(res, tgt)
52: (4) @pytest.mark.parametrize(('inp', 'tgt'), (
53: (8)     ([1, 2, 3], "1.0 + 2.0·He· $\hat{x}$ (x) + 3.0·He· $\hat{x}$ ,,(x)"),
54: (8)     ([-1, 0, 3, -1], "-1.0 + 0.0·He· $\hat{x}$ (x) + 3.0·He· $\hat{x}$ ,,(x) -
1.0·He· $\hat{x}$ ,f(x)),",
55: (8)     (arange(12), ("0.0 + 1.0·He· $\hat{x}$ (x) + 2.0·He· $\hat{x}$ ,,(x) + 3.0·He· $\hat{x}$ ,f(x) +
4.0·He· $\hat{x}$ ,,,(x) + 5.0·He· $\hat{x}$ ,...(x) +\n6.0·He· $\hat{x}$ ,†(x) +
7.0·He· $\hat{x}$ ,‡(x) +
8.0·He· $\hat{x}$ ,^(x) + 9.0·He· $\hat{x}$ ,‰(x) + 10.0·He· $\hat{x}$ ,¤(x) +\n"
11.0·He· $\hat{x}$ ,¤¤(x")),
58: (22)                                         "8.0·He· $\hat{x}$ ,^(x) + 9.0·He· $\hat{x}$ ,‰(x) + 10.0·He· $\hat{x}$ ,¤(x) +\n"
11.0·He· $\hat{x}$ ,¤¤(x")),
59: (4) ))
60: (4)     def test_hermiteE_str(self, inp, tgt):
61: (8)         res = str(poly.HermiteE(inp))
62: (8)         assert_equal(res, tgt)
63: (4) @pytest.mark.parametrize(('inp', 'tgt'), (
64: (8)     ([1, 2, 3], "1.0 + 2.0·L· $\hat{x}$ (x) + 3.0·L· $\hat{x}$ ,,(x)"),
65: (8)     ([-1, 0, 3, -1], "-1.0 + 0.0·L· $\hat{x}$ (x) + 3.0·L· $\hat{x}$ ,,(x) - 1.0·L· $\hat{x}$ ,f(x)"),
66: (8)     (arange(12), ("0.0 + 1.0·L· $\hat{x}$ (x) + 2.0·L· $\hat{x}$ ,,(x) + 3.0·L· $\hat{x}$ ,f(x) +
4.0·L· $\hat{x}$ ,,,(x) + "
67: (22)                                         "5.0·L· $\hat{x}$ ,...(x) +\n6.0·L· $\hat{x}$ ,†(x) + 7.0·L· $\hat{x}$ ,‡(x) +
8.0·L· $\hat{x}$ ,^(x) + "
68: (22)                                         "9.0·L· $\hat{x}$ ,‰(x) + 10.0·L· $\hat{x}$ ,¤(x) + 11.0·L· $\hat{x}$ ,¤¤(x)),
69: (4) ))
70: (4)     def test_laguerre_str(self, inp, tgt):
71: (8)         res = str(poly.Laguerre(inp))
72: (8)         assert_equal(res, tgt)
73: (0) class TestStrAscii:
74: (4)     @pytest.fixture(scope='class', autouse=True)
75: (4)     def use_ascii(self):

```

```

76: (8)             poly.set_default_printstyle('ascii')
77: (4)             @pytest.mark.parametrize(('inp', 'tgt'), (
78: (8)                 ([1, 2, 3], "1.0 + 2.0 x + 3.0 x**2"),
79: (8)                 ([-1, 0, 3, -1], "-1.0 + 0.0 x + 3.0 x**2 - 1.0 x**3"),
80: (8)                 (arange(12), ("0.0 + 1.0 x + 2.0 x**2 + 3.0 x**3 + 4.0 x**4 + "
81: (22)                   "5.0 x**5 + 6.0 x**6 +\n7.0 x**7 + 8.0 x**8 + "
82: (22)                   "9.0 x**9 + 10.0 x**10 + 11.0 x**11")),
83: (4)             ))
84: (4)             def test_polynomial_str(self, inp, tgt):
85: (8)                 res = str(poly.Polynomial(inp))
86: (8)                 assert_equal(res, tgt)
87: (4)             @pytest.mark.parametrize(('inp', 'tgt'), (
88: (8)                 ([1, 2, 3], "1.0 + 2.0 T_1(x) + 3.0 T_2(x)"),
89: (8)                 ([-1, 0, 3, -1], "-1.0 + 0.0 T_1(x) + 3.0 T_2(x) - 1.0 T_3(x)"),
90: (8)                 (arange(12), ("0.0 + 1.0 T_1(x) + 2.0 T_2(x) + 3.0 T_3(x) + "
91: (22)                   "4.0 T_4(x) + 5.0 T_5(x) +\n6.0 T_6(x) + 7.0 T_7(x) + "
92: (22)                   "8.0 T_8(x) + 9.0 T_9(x) + 10.0 T_10(x) +\n"
93: (22)                   "11.0 T_11(x))),,
94: (4)             ))
95: (4)             def test_chebyshev_str(self, inp, tgt):
96: (8)                 res = str(poly.Chebyshev(inp))
97: (8)                 assert_equal(res, tgt)
98: (4)             @pytest.mark.parametrize(('inp', 'tgt'), (
99: (8)                 ([1, 2, 3], "1.0 + 2.0 P_1(x) + 3.0 P_2(x)"),
100: (8)                 ([-1, 0, 3, -1], "-1.0 + 0.0 P_1(x) + 3.0 P_2(x) - 1.0 P_3(x)"),
101: (8)                 (arange(12), ("0.0 + 1.0 P_1(x) + 2.0 P_2(x) + 3.0 P_3(x) + "
102: (22)                   "4.0 P_4(x) + 5.0 P_5(x) +\n6.0 P_6(x) + 7.0 P_7(x) + "
103: (22)                   "8.0 P_8(x) + 9.0 P_9(x) + 10.0 P_10(x) +\n"
104: (22)                   "11.0 P_11(x))),,
105: (4)             ))
106: (4)             def test_legendre_str(self, inp, tgt):
107: (8)                 res = str(poly.Legendre(inp))
108: (8)                 assert_equal(res, tgt)
109: (4)             @pytest.mark.parametrize(('inp', 'tgt'), (
110: (8)                 ([1, 2, 3], "1.0 + 2.0 H_1(x) + 3.0 H_2(x)"),
111: (8)                 ([-1, 0, 3, -1], "-1.0 + 0.0 H_1(x) + 3.0 H_2(x) - 1.0 H_3(x)"),
112: (8)                 (arange(12), ("0.0 + 1.0 H_1(x) + 2.0 H_2(x) + 3.0 H_3(x) + "
113: (22)                   "4.0 H_4(x) + 5.0 H_5(x) +\n6.0 H_6(x) + 7.0 H_7(x) + "
114: (22)                   "8.0 H_8(x) + 9.0 H_9(x) + 10.0 H_10(x) +\n"
115: (22)                   "11.0 H_11(x))),,
116: (4)             ))
117: (4)             def test_hermite_str(self, inp, tgt):
118: (8)                 res = str(poly.Hermite(inp))
119: (8)                 assert_equal(res, tgt)
120: (4)             @pytest.mark.parametrize(('inp', 'tgt'), (
121: (8)                 ([1, 2, 3], "1.0 + 2.0 He_1(x) + 3.0 He_2(x)"),
122: (8)                 ([-1, 0, 3, -1], "-1.0 + 0.0 He_1(x) + 3.0 He_2(x) - 1.0 He_3(x)"),
123: (8)                 (arange(12), ("0.0 + 1.0 He_1(x) + 2.0 He_2(x) + 3.0 He_3(x) + "
124: (22)                   "4.0 He_4(x) +\n5.0 He_5(x) + 6.0 He_6(x) + "
125: (22)                   "7.0 He_7(x) + 8.0 He_8(x) + 9.0 He_9(x) +\n"
126: (22)                   "10.0 He_10(x) + 11.0 He_11(x))),,
127: (4)             ))
128: (4)             def test_hermiteE_str(self, inp, tgt):
129: (8)                 res = str(poly.HermiteE(inp))
130: (8)                 assert_equal(res, tgt)
131: (4)             @pytest.mark.parametrize(('inp', 'tgt'), (
132: (8)                 ([1, 2, 3], "1.0 + 2.0 L_1(x) + 3.0 L_2(x)"),
133: (8)                 ([-1, 0, 3, -1], "-1.0 + 0.0 L_1(x) + 3.0 L_2(x) - 1.0 L_3(x)"),
134: (8)                 (arange(12), ("0.0 + 1.0 L_1(x) + 2.0 L_2(x) + 3.0 L_3(x) + "
135: (22)                   "4.0 L_4(x) + 5.0 L_5(x) +\n6.0 L_6(x) + 7.0 L_7(x) + "
136: (22)                   "8.0 L_8(x) + 9.0 L_9(x) + 10.0 L_10(x) +\n"
137: (22)                   "11.0 L_11(x))),,
138: (4)             ))
139: (4)             def test_laguerre_str(self, inp, tgt):
140: (8)                 res = str(poly.Laguerre(inp))
141: (8)                 assert_equal(res, tgt)
142: (0)             class TestLinebreaking:
143: (4)                 @pytest.fixture(scope='class', autouse=True)
144: (4)                 def use_ascii(self):

```

```

145: (8)             poly.set_default_printstyle('ascii')
146: (4)             def test_single_line_one_less(self):
147: (8)                 p = poly.Polynomial([12345678, 12345678, 12345678, 12345678, 123])
148: (8)                 assert_equal(len(str(p)), 74)
149: (8)                 assert_equal(str(p), (
150: (12)                     '12345678.0 + 12345678.0 x + 12345678.0 x**2 + '
151: (12)                     '12345678.0 x**3 + 123.0 x**4'
152: (8)                 ))
153: (4)             def test_num_chars_is_linewidth(self):
154: (8)                 p = poly.Polynomial([12345678, 12345678, 12345678, 12345678, 1234])
155: (8)                 assert_equal(len(str(p)), 75)
156: (8)                 assert_equal(str(p), (
157: (12)                     '12345678.0 + 12345678.0 x + 12345678.0 x**2 + '
158: (12)                     '12345678.0 x**3 +\n1234.0 x**4'
159: (8)                 ))
160: (4)             def test_first_linebreak_multiline_one_less_than_linewidth(self):
161: (8)                 p = poly.Polynomial(
162: (16)                     [12345678, 12345678, 12345678, 12345678, 1, 12345678]
163: (12)                 )
164: (8)                 assert_equal(len(str(p).split('\n')[0]), 74)
165: (8)                 assert_equal(str(p), (
166: (12)                     '12345678.0 + 12345678.0 x + 12345678.0 x**2 + '
167: (12)                     '12345678.0 x**3 + 1.0 x**4 +\n12345678.0 x**5'
168: (8)                 ))
169: (4)             def test_first_linebreak_multiline_on_linewidth(self):
170: (8)                 p = poly.Polynomial(
171: (16)                     [12345678, 12345678, 12345678, 12345678.12, 1, 12345678]
172: (12)                 )
173: (8)                 assert_equal(str(p), (
174: (12)                     '12345678.0 + 12345678.0 x + 12345678.0 x**2 + '
175: (12)                     '12345678.12 x**3 +\n1.0 x**4 + 12345678.0 x**5'
176: (8)                 ))
177: (4)             @pytest.mark.parametrize(('lw', 'tgt'), (
178: (8)                 (75, ('0.0 + 10.0 x + 200.0 x**2 + 3000.0 x**3 + 40000.0 x**4 + '
179: (14)                   '500000.0 x**5 +\n600000.0 x**6 + 70000.0 x**7 + 8000.0 x**8 + '
180: (14)                   '900.0 x**9')),
181: (8)                 (45, ('0.0 + 10.0 x + 200.0 x**2 + 3000.0 x**3 +\n40000.0 x**4 + '
182: (14)                   '500000.0 x**5 +\n600000.0 x**6 + 70000.0 x**7 + 8000.0 x**8
+\n'
183: (14)                   '900.0 x**9')),
184: (8)                 (132, ('0.0 + 10.0 x + 200.0 x**2 + 3000.0 x**3 + 40000.0 x**4 + '
185: (15)                   '500000.0 x**5 + 600000.0 x**6 + 70000.0 x**7 + 8000.0 x**8 +
186: (15)                   '900.0 x**9')),
187: (4)             ))
188: (4)             def test_linewidth_printoption(self, lw, tgt):
189: (8)                 p = poly.Polynomial(
190: (12)                     [0, 10, 200, 3000, 40000, 500000, 600000, 70000, 8000, 900]
191: (8)                 )
192: (8)                 with printoptions(linewidth=lw):
193: (12)                     assert_equal(str(p), tgt)
194: (12)                     for line in str(p).split('\n'):
195: (16)                         assert_(len(line) < lw)
196: (0)             def test_set_default_printoptions():
197: (4)                 p = poly.Polynomial([1, 2, 3])
198: (4)                 c = poly.Chebyshev([1, 2, 3])
199: (4)                 poly.set_default_printstyle('ascii')
200: (4)                 assert_equal(str(p), "1.0 + 2.0 x + 3.0 x**2")
201: (4)                 assert_equal(str(c), "1.0 + 2.0 T_1(x) + 3.0 T_2(x)")
202: (4)                 poly.set_default_printstyle('unicode')
203: (4)                 assert_equal(str(p), "1.0 + 2.0\u00d7x + 3.0\u00d7x\u00b2")
204: (4)                 assert_equal(str(c), "1.0 + 2.0\u00d7T\u00b1(x) + 3.0\u00d7T\u00b2,(x)")
205: (4)                 with pytest.raises(ValueError):
206: (8)                     poly.set_default_printstyle('invalid_input')
207: (0)             def test_complex_coefficients():
208: (4)                 """Test both numpy and built-in complex."""
209: (4)                 coefs = [0+1j, 1+1j, -2+2j, 3+0j]
210: (4)                 p1 = poly.Polynomial(coefs)
211: (4)                 p2 = poly.Polynomial(array(coefs, dtype=object))
212: (4)                 poly.set_default_printstyle('unicode')

```

```

213: (4) assert_equal(str(p1), "1j + (1+1j)·x - (2-2j)·x·x² + (3+0j)·x·x³")
214: (4) assert_equal(str(p2), "1j + (1+1j)·x + (-2+2j)·x·x² + (3+0j)·x·x³")
215: (4) poly.set_default_printstyle('ascii')
216: (4) assert_equal(str(p1), "1j + (1+1j) x - (2-2j) x**2 + (3+0j) x**3")
217: (4) assert_equal(str(p2), "1j + (1+1j) x + (-2+2j) x**2 + (3+0j) x**3")
218: (0) @pytest.mark.parametrize(('coefs', 'tgt'), (
219: (4)     (array([Fraction(1, 2), Fraction(3, 4)], dtype=object), (
220: (8)         "1/2 + 3/4·x"
221: (4)     )),
222: (4)     (array([1, 2, Fraction(5, 7)], dtype=object), (
223: (8)         "1 + 2·x + 5/7·x·x²"
224: (4)     )),
225: (4)     (array([Decimal('1.00'), Decimal('2.2'), 3], dtype=object), (
226: (8)         "1.00 + 2.2·x + 3·x·x²"
227: (4)     )),
228: (0) ))
229: (0) def test_numeric_object_coefficients(coefs, tgt):
230: (4)     p = poly.Polynomial(coefs)
231: (4)     poly.set_default_printstyle('unicode')
232: (4)     assert_equal(str(p), tgt)
233: (0) @pytest.mark.parametrize(('coefs', 'tgt'), (
234: (4)     (array([1, 2, 'f'], dtype=object), '1 + 2·x + f·x·x²'),
235: (4)     (array([1, 2, [3, 4]], dtype=object), '1 + 2·x + [3, 4]·x·x²'),
236: (0) ))
237: (0) def test_nonnumeric_object_coefficients(coefs, tgt):
238: (4)     """
239: (4)         Test coef fallback for object arrays of non-numeric coefficients.
240: (4)     """
241: (4)     p = poly.Polynomial(coefs)
242: (4)     poly.set_default_printstyle('unicode')
243: (4)     assert_equal(str(p), tgt)
244: (0) class TestFormat:
245: (4)     def test_format_unicode(self):
246: (8)         poly.set_default_printstyle('ascii')
247: (8)         p = poly.Polynomial([1, 2, 0, -1])
248: (8)         assert_equal(format(p, 'unicode'), "1.0 + 2.0·x + 0.0·x·x² - 1.0·x·x³")
249: (4)     def test_format_ascii(self):
250: (8)         poly.set_default_printstyle('unicode')
251: (8)         p = poly.Polynomial([1, 2, 0, -1])
252: (8)         assert_equal(
253: (12)             format(p, 'ascii'), "1.0 + 2.0 x + 0.0 x**2 - 1.0 x**3"
254: (8)         )
255: (4)     def test_empty_formatstr(self):
256: (8)         poly.set_default_printstyle('ascii')
257: (8)         p = poly.Polynomial([1, 2, 3])
258: (8)         assert_equal(format(p), "1.0 + 2.0 x + 3.0 x**2")
259: (8)         assert_equal(f"{p}", "1.0 + 2.0 x + 3.0 x**2")
260: (4)     def test_bad_formatstr(self):
261: (8)         p = poly.Polynomial([1, 2, 0, -1])
262: (8)         with pytest.raises(ValueError):
263: (12)             format(p, '.2f')
264: (0) @pytest.mark.parametrize(('poly', 'tgt'), (
265: (4)     (poly.Polynomial, '1.0 + 2.0·z + 3.0·z·z²'),
266: (4)     (poly.Chebyshev, '1.0 + 2.0·Tₐ(z) + 3.0·Tₐ,,(z)'),
267: (4)     (poly.Hermite, '1.0 + 2.0·Hₐ(z) + 3.0·Hₐ,,(z)'),
268: (4)     (poly.HermiteE, '1.0 + 2.0·Hₑₐ(z) + 3.0·Hₑₐ,,(z)'),
269: (4)     (poly.Laguerre, '1.0 + 2.0·Lₐ(z) + 3.0·Lₐ,,(z)'),
270: (4)     (poly.Legendre, '1.0 + 2.0·Pₐ(z) + 3.0·Pₐ,,(z)'),
271: (0) ))
272: (0)     def test_symbol(poly, tgt):
273: (4)         p = poly([1, 2, 3], symbol='z')
274: (4)         assert_equal(f"{p:unicode}", tgt)
275: (0) class TestRepr:
276: (4)     def test_polynomial_str(self):
277: (8)         res = repr(poly.Polynomial([0, 1]))
278: (8)         tgt = (
279: (12)             "Polynomial([0., 1.], domain=[-1, 1], window=[-1, 1], "
280: (12)             "symbol='x')"

```

```

281: (8) )
282: (8) assert_equal(res, tgt)
283: (4) def test_chebyshev_str(self):
284: (8) res = repr(poly.Chebyshev([0, 1]))
285: (8) tgt = (
286: (12) "Chebyshev([0., 1.], domain=[-1, 1], window=[-1, 1], "
287: (12) "symbol='x')")
288: (8) )
289: (8) assert_equal(res, tgt)
290: (4) def test_legendre_repr(self):
291: (8) res = repr(poly.Legendre([0, 1]))
292: (8) tgt = (
293: (12) "Legendre([0., 1.], domain=[-1, 1], window=[-1, 1], "
294: (12) "symbol='x')")
295: (8) )
296: (8) assert_equal(res, tgt)
297: (4) def test_hermite_repr(self):
298: (8) res = repr(poly.Hermite([0, 1]))
299: (8) tgt = (
300: (12) "Hermite([0., 1.], domain=[-1, 1], window=[-1, 1], "
301: (12) "symbol='x')")
302: (8) )
303: (8) assert_equal(res, tgt)
304: (4) def test_hermiteE_repr(self):
305: (8) res = repr(poly.HermiteE([0, 1]))
306: (8) tgt = (
307: (12) "HermiteE([0., 1.], domain=[-1, 1], window=[-1, 1], "
308: (12) "symbol='x')")
309: (8) )
310: (8) assert_equal(res, tgt)
311: (4) def test_laguerre_repr(self):
312: (8) res = repr(poly.Laguerre([0, 1]))
313: (8) tgt = (
314: (12) "Laguerre([0., 1.], domain=[0, 1], window=[0, 1], "
315: (12) "symbol='x')")
316: (8) )
317: (8) assert_equal(res, tgt)
318: (0) class TestLatexRepr:
319: (4) """Test the latex repr used by Jupyter"""
320: (4) def as_latex(self, obj):
321: (8) obj._repr_latex_scalar = lambda x, parens=False: str(x)
322: (8) try:
323: (12) return obj._repr_latex_()
324: (8) finally:
325: (12) del obj._repr_latex_scalar
326: (4) def test_simple_polynomial(self):
327: (8) p = poly.Polynomial([1, 2, 3])
328: (8) assert_equal(self.as_latex(p),
329: (12) r'$x \mapsto 1.0 + 2.0\cdot x + 3.0\cdot x^2$')
330: (8) p = poly.Polynomial([1, 2, 3], domain=[-2, 0])
331: (8) assert_equal(self.as_latex(p),
332: (12) r'$x \mapsto 1.0 + 2.0\cdot \left(1.0 + x\right) + 3.0\cdot \left(1.0 +
x\right)^2$')
333: (8) p = poly.Polynomial([1, 2, 3], domain=[-0.5, 0.5])
334: (8) assert_equal(self.as_latex(p),
335: (12) r'$x \mapsto 1.0 + 2.0\cdot \left(2.0x\right) +
3.0\cdot \left(2.0x\right)^2$')
336: (8) p = poly.Polynomial([1, 2, 3], domain=[-1, 0])
337: (8) assert_equal(self.as_latex(p),
338: (12) r'$x \mapsto 1.0 + 2.0\cdot \left(1.0 + 2.0x\right) + 3.0\cdot \left(1.0 +
2.0x\right)^2$')
339: (4) def test_basis_func(self):
340: (8) p = poly.Chebyshev([1, 2, 3])
341: (8) assert_equal(self.as_latex(p),
342: (12) r'$x \mapsto 1.0\cdot T_0(x) + 2.0\cdot T_1(x) + 3.0\cdot T_2(x)$')
343: (8) p = poly.Chebyshev([1, 2, 3], domain=[-1, 0])
344: (8) assert_equal(self.as_latex(p),
345: (12) r'$x \mapsto 1.0\cdot T_0(1.0 + 2.0x) + 2.0\cdot T_1(1.0 + 2.0x) + '

```

```

3.0\,{T}_\{2\}(1.0 + 2.0x$')
346: (4)         def test_multichar_basis_func(self):
347: (8)             p = poly.HermiteE([1, 2, 3])
348: (8)             assert_equal(self.as_latex(p),
349: (12)                 r'$x \mapsto 1.0\,{\text{He}}_\{0\}(x) + 2.0\,{\text{He}}_\{1\}(x) + 3.0\,{\text{He}}_\{2\}(x)$')
350: (4)         def test_symbol_basic(self):
351: (8)             p = poly.Polynomial([1, 2, 3], symbol='z')
352: (8)             assert_equal(self.as_latex(p),
353: (12)                 r'$z \mapsto 1.0 + 2.0\,z + 3.0\,z^2$')
354: (8)             p = poly.Polynomial([1, 2, 3], domain=[-2, 0], symbol='z')
355: (8)             assert_equal(
356: (12)                 self.as_latex(p),
357: (12)                 (
358: (16)                     r'$z \mapsto 1.0 + 2.0\,\left(1.0 + z\right) + 3.0\,',
359: (16)                     r'\left(1.0 + z\right)^2$'
360: (12)                 ),
361: (8)             )
362: (8)             p = poly.Polynomial([1, 2, 3], domain=[-0.5, 0.5], symbol='z')
363: (8)             assert_equal(
364: (12)                 self.as_latex(p),
365: (12)                 (
366: (16)                     r'$z \mapsto 1.0 + 2.0\,\left(2.0z\right) + 3.0\,',
367: (16)                     r'\left(2.0z\right)^2$'
368: (12)                 ),
369: (8)             )
370: (8)             p = poly.Polynomial([1, 2, 3], domain=[-1, 0], symbol='z')
371: (8)             assert_equal(
372: (12)                 self.as_latex(p),
373: (12)                 (
374: (16)                     r'$z \mapsto 1.0 + 2.0\,\left(1.0 + 2.0z\right) + 3.0\,',
375: (16)                     r'\left(1.0 + 2.0z\right)^2$'
376: (12)                 ),
377: (8)             )
378: (0)             SWITCH_TO_EXP = (
379: (4)                 '1.0 + (1.0e-01) x + (1.0e-02) x**2',
380: (4)                 '1.2 + (1.2e-01) x + (1.2e-02) x**2',
381: (4)                 '1.23 + 0.12 x + (1.23e-02) x**2 + (1.23e-03) x**3',
382: (4)                 '1.235 + 0.123 x + (1.235e-02) x**2 + (1.235e-03) x**3',
383: (4)                 '1.2346 + 0.1235 x + 0.0123 x**2 + (1.2346e-03) x**3 + (1.2346e-04) x**4',
384: (4)                 '1.23457 + 0.12346 x + 0.01235 x**2 + (1.23457e-03) x**3 + '
385: (4)                 '(1.23457e-04) x**4',
386: (4)                 '1.234568 + 0.123457 x + 0.012346 x**2 + 0.001235 x**3 + '
387: (4)                 '(1.234568e-04) x**4 + (1.234568e-05) x**5',
388: (4)                 '1.2345679 + 0.1234568 x + 0.0123457 x**2 + 0.0012346 x**3 + '
389: (4)                 '(1.2345679e-04) x**4 + (1.2345679e-05) x**5')
390: (0)             class TestPrintOptions:
391: (4)                 """
392: (4)                     Test the output is properly configured via printoptions.
393: (4)                     The exponential notation is enabled automatically when the values
394: (4)                     are too small or too large.
395: (4)                 """
396: (4)             @pytest.fixture(scope='class', autouse=True)
397: (4)             def use_ascii(self):
398: (8)                 poly.set_default_printstyle('ascii')
399: (4)             def test_str(self):
400: (8)                 p = poly.Polynomial([1/2, 1/7, 1/7*10**8, 1/7*10**9])
401: (8)                 assert_equal(str(p), '0.5 + 0.14285714 x + 14285714.28571429 x**2 +
402: (29)                               '+ (1.42857143e+08) x**3')
403: (8)                 with printoptions(precision=3):
404: (12)                     assert_equal(str(p), '0.5 + 0.143 x + 14285714.286 x**2 +
405: (33)                               '+ (1.429e+08) x**3')
406: (4)             def test_latex(self):
407: (8)                 p = poly.Polynomial([1/2, 1/7, 1/7*10**8, 1/7*10**9])
408: (8)                 assert_equal(p._repr_latex_(),
409: (12)                     r'$x \mapsto \text{0.5} + \text{0.14285714}\,x + '
410: (12)                     r'\text{14285714.28571429}\,x^2 + '
411: (12)                     r'\text{(1.42857143e+08)}\,x^3$')
412: (8)                 with printoptions(precision=3):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

413: (12)             assert_equal(p._repr_latex_(),
414: (16)                 r'$x \mapsto \text{0.5} + \text{0.143}, x + '
415: (16)                 r'\text{14285714.286}, x^2 + \text{(1.429e+08)}, x^3$')
416: (4) def test_fixed(self):
417: (8)     p = poly.Polynomial([1/2])
418: (8)     assert_equal(str(p), '0.5')
419: (8)     with printoptions(floatmode='fixed'):
420: (12)         assert_equal(str(p), '0.50000000')
421: (8)     with printoptions(floatmode='fixed', precision=4):
422: (12)         assert_equal(str(p), '0.5000')
423: (4) def test_switch_to_exp(self):
424: (8)     for i, s in enumerate(SWITCH_TO_EXP):
425: (12)         with printoptions(precision=i):
426: (16)             p = poly.Polynomial([1.23456789*10**-i
427: (37)                         for i in range(i//2+3)])
428: (16)             assert str(p).replace('\n', ' ') == s
429: (4) def test_non_finite(self):
430: (8)     p = poly.Polynomial([nan, inf])
431: (8)     assert str(p) == 'nan + inf x'
432: (8)     assert p._repr_latex_() == r'$x \mapsto \text{nan} + \text{inf}, x$'
433: (8)     with printoptions(nanstr='NAN', infstr='INF'):
434: (12)         assert str(p) == 'NAN + INF x'
435: (12)         assert p._repr_latex_() == \
436: (16)             r'$x \mapsto \text{NAN} + \text{INF}, x$'

```

-----  
File 307 - \_\_init\_\_.py:

```
1: (0)
```

-----  
File 308 - \_pickle.py:

```

1: (0)     from .mtrand import RandomState
2: (0)     from ._philox import Philox
3: (0)     from ._pcg64 import PCG64, PCG64DXSM
4: (0)     from ._sfc64 import SFC64
5: (0)     from ._generator import Generator
6: (0)     from ._mt19937 import MT19937
7: (0)     BitGenerators = {'MT19937': MT19937,
8: (17)           'PCG64': PCG64,
9: (17)           'PCG64DXSM': PCG64DXSM,
10: (17)          'Philox': Philox,
11: (17)          'SFC64': SFC64,
12: (17)      }
13: (0) def __bit_generator_ctor(bit_generator_name='MT19937'):
14: (4)     """
15: (4)     Pickling helper function that returns a bit generator object
16: (4)     Parameters
17: (4)     -----
18: (4)     bit_generator_name : str
19: (8)         String containing the name of the BitGenerator
20: (4)     Returns
21: (4)     -----
22: (4)     bit_generator : BitGenerator
23: (8)         BitGenerator instance
24: (4)     """
25: (4)     if bit_generator_name in BitGenerators:
26: (8)         bit_generator = BitGenerators[bit_generator_name]
27: (4)     else:
28: (8)         raise ValueError(str(bit_generator_name) + ' is not a known '
29: (51)                                         'BitGenerator module.')
30: (4)     return bit_generator()
31: (0) def __generator_ctor(bit_generator_name="MT19937",
32: (21)                     bit_generator_ctor=__bit_generator_ctor):
33: (4)     """
34: (4)     Pickling helper function that returns a Generator object

```

```

35: (4)             Parameters
36: (4)             -----
37: (4)             bit_generator_name : str
38: (8)                 String containing the core BitGenerator's name
39: (4)             bit_generator_ctor : callable, optional
40: (8)                 Callable function that takes bit_generator_name as its only argument
41: (8)                 and returns an instantized bit generator.
42: (4)             Returns
43: (4)             -----
44: (4)             rg : Generator
45: (8)                 Generator using the named core BitGenerator
46: (4)             """
47: (4)             return Generator(bit_generator_ctor(bit_generator_name))
48: (0)         def __randomstate_ctor(bit_generator_name="MT19937",
49: (23)                         bit_generator_ctor=__bit_generator_ctor):
50: (4)                         """
51: (4)                         Pickling helper function that returns a legacy RandomState-like object
52: (4)                         Parameters
53: (4)                         -----
54: (4)                         bit_generator_name : str
55: (8)                             String containing the core BitGenerator's name
56: (4)                         bit_generator_ctor : callable, optional
57: (8)                             Callable function that takes bit_generator_name as its only argument
58: (8)                             and returns an instantized bit generator.
59: (4)             Returns
60: (4)             -----
61: (4)             rs : RandomState
62: (8)                 Legacy RandomState using the named core BitGenerator
63: (4)             """
64: (4)             return RandomState(bit_generator_ctor(bit_generator_name))

-----

```

#### File 309 - \_\_init\_\_.py:

```

1: (0)             """
2: (0)             =====
3: (0)             Random Number Generation
4: (0)             =====
5: (0)             Use ``default_rng()`` to create a `Generator` and call its methods.
6: (0)             =====
7: (0)             Generator
8: (0)             -----
9: (0)             Generator      Class implementing all of the random number distributions
10: (0)            default_rng    Default constructor for ``Generator``
11: (0)             =====
12: (0)             =====
13: (0)             BitGenerator Streams that work with Generator
14: (0)             -----
15: (0)             MT19937
16: (0)             PCG64
17: (0)             PCG64DXSM
18: (0)             Philox
19: (0)             SFC64
20: (0)             =====
21: (0)             =====
22: (0)             Getting entropy to initialize a BitGenerator
23: (0)             -----
24: (0)             SeedSequence
25: (0)             =====
26: (0)             Legacy
27: (0)             -----
28: (0)             For backwards compatibility with previous versions of numpy before 1.17, the
29: (0)             various aliases to the global `RandomState` methods are left alone and do not
30: (0)             use the new `Generator` API.
31: (0)             =====
32: (0)             Utility functions
33: (0)             -----
34: (0)             random          Uniformly distributed floats over ``[0, 1)``

```

```

35: (0)           bytes          Uniformly distributed random bytes.
36: (0)           permutation   Randomly permute a sequence / generate a random sequence.
37: (0)           shuffle        Randomly permute a sequence in place.
38: (0)           choice         Random sample from 1-D array.
39: (0)           =====
40: (0)           =====
41: (0)           Compatibility
42: (0)           functions - removed
43: (0)           in the new API
44: (0)           -----
45: (0)           rand          Uniformly distributed values.
46: (0)           randn         Normally distributed values.
47: (0)           ranf          Uniformly distributed floating point numbers.
48: (0)           random_integers Uniformly distributed integers in a given range.
49: (21)          (deprecated, use ``integers(..., closed=True)`` instead)
50: (0)           random_sample  Alias for `random_sample`
51: (0)           randint       Uniformly distributed integers in a given range
52: (0)           seed          Seed the legacy random number generator.
53: (0)           =====
54: (0)           =====
55: (0)           Univariate
56: (0)           distributions
57: (0)           -----
58: (0)           beta          Beta distribution over ``[0, 1]``.
59: (0)           binomial      Binomial distribution.
60: (0)           chisquare     :math:`\chi^2` distribution.
61: (0)           exponential   Exponential distribution.
62: (0)           f             F (Fisher-Snedecor) distribution.
63: (0)           gamma         Gamma distribution.
64: (0)           geometric     Geometric distribution.
65: (0)           gumbel        Gumbel distribution.
66: (0)           hypergeometric Hypergeometric distribution.
67: (0)           laplace       Laplace distribution.
68: (0)           logistic      Logistic distribution.
69: (0)           lognormal     Log-normal distribution.
70: (0)           logseries    Logarithmic series distribution.
71: (0)           negative_binomial Negative binomial distribution.
72: (0)           noncentral_chisquare Non-central chi-square distribution.
73: (0)           noncentral_f   Non-central F distribution.
74: (0)           normal        Normal / Gaussian distribution.
75: (0)           pareto        Pareto distribution.
76: (0)           poisson       Poisson distribution.
77: (0)           power         Power distribution.
78: (0)           rayleigh      Rayleigh distribution.
79: (0)           triangular    Triangular distribution.
80: (0)           uniform       Uniform distribution.
81: (0)           vonmises     Von Mises circular distribution.
82: (0)           wald          Wald (inverse Gaussian) distribution.
83: (0)           weibull       Weibull distribution.
84: (0)           zipf          Zipf's distribution over ranked data.
85: (0)           =====
86: (0)           =====
87: (0)           Multivariate
88: (0)           distributions
89: (0)           -----
-
90: (0)           dirichlet     Multivariate generalization of Beta distribution.
91: (0)           multinomial   Multivariate generalization of the binomial distribution.
92: (0)           multivariate_normal Multivariate generalization of the normal distribution.
93: (0)           =====
94: (0)           =====
95: (0)           Standard
96: (0)           distributions
97: (0)           -----
98: (0)           standard_cauchy Standard Cauchy-Lorentz distribution.
99: (0)           standard_exponential Standard exponential distribution.
100: (0)          standard_gamma Standard Gamma distribution.

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

101: (0) standard_normal      Standard normal distribution.
102: (0) standard_t          Standard Student's t-distribution.
103: (0)
104: (0)
105: (0) Internal functions
106: (0)
107: (0) get_state           Get tuple representing internal state of generator.
108: (0) set_state            Set state of generator.
109: (0)
110: (0)
111: (0)     """
112: (4)     __all__ = [
113: (4)         'beta',
114: (4)         'binomial',
115: (4)         'bytes',
116: (4)         'chisquare',
117: (4)         'choice',
118: (4)         'dirichlet',
119: (4)         'exponential',
120: (4)         'f',
121: (4)         'gamma',
122: (4)         'geometric',
123: (4)         'get_state',
124: (4)         'gumbel',
125: (4)         'hypergeometric',
126: (4)         'laplace',
127: (4)         'logistic',
128: (4)         'lognormal',
129: (4)         'logseries',
130: (4)         'multinomial',
131: (4)         'multivariate_normal',
132: (4)         'negative_binomial',
133: (4)         'noncentral_chisquare',
134: (4)         'noncentral_f',
135: (4)         'normal',
136: (4)         'pareto',
137: (4)         'permutation',
138: (4)         'poisson',
139: (4)         'power',
140: (4)         'rand',
141: (4)         'randint',
142: (4)         'randn',
143: (4)         'random',
144: (4)         'random_integers',
145: (4)         'random_sample',
146: (4)         'ranf',
147: (4)         'rayleigh',
148: (4)         'sample',
149: (4)         'seed',
150: (4)         'set_state',
151: (4)         'shuffle',
152: (4)         'standard_cauchy',
153: (4)         'standard_exponential',
154: (4)         'standard_gamma',
155: (4)         'standard_normal',
156: (4)         'standard_t',
157: (4)         'triangular',
158: (4)         'uniform',
159: (4)         'vonmises',
160: (4)         'wald',
161: (4)         'weibull',
162: (0)         'zipf',
163: (0)     ]
164: (0)     from . import _pickle
165: (0)     from . import _common
166: (0)     from . import _bounded_integers
167: (0)     from ._generator import Generator, default_rng
168: (0)     from .bit_generator import SeedSequence, BitGenerator
169: (0)     from ._mt19937 import MT19937
169: (0)     from ._pcg64 import PCG64, PCG64DXSM

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

170: (0)         from ._philox import Philox
171: (0)         from ._sfc64 import SFC64
172: (0)         from .mtrand import *
173: (0)         __all__ += ['Generator', 'RandomState', 'SeedSequence', 'MT19937',
174: (12)           'Philox', 'PCG64', 'PCG64DXSM', 'SFC64', 'default_rng',
175: (12)           'BitGenerator']
176: (0)         def __RandomState_ctor():
177: (4)             """Return a RandomState instance.
178: (4)             This function exists solely to assist (un)pickling.
179: (4)             Note that the state of the RandomState returned here is irrelevant, as
this
180: (4)             function's entire purpose is to return a newly allocated RandomState whose
181: (4)             state pickle can set. Consequently the RandomState returned by this
function
182: (4)             is a freshly allocated copy with a seed=0.
183: (4)             See https://github.com/numpy/numpy/issues/4763 for a detailed discussion
184: (4)
185: (4)             """
186: (0)             return RandomState(seed=0)
187: (0)         from numpy._pytesttester import PytestTester
188: (0)         test = PytestTester(__name__)
189: (0)         del PytestTester

```

---

File 310 - test\_direct.py:

```

1: (0)             import os
2: (0)             from os.path import join
3: (0)             import sys
4: (0)             import numpy as np
5: (0)             from numpy.testing import (assert_equal, assert_allclose, assert_array_equal,
6: (27)               assert_raises)
7: (0)             import pytest
8: (0)             from numpy.random import (
9: (4)                 Generator, MT19937, PCG64, PCG64DXSM, Philox, RandomState, SeedSequence,
10: (4)                 SFC64, default_rng
11: (0)             )
12: (0)             from numpy.random._common import interface
13: (0)             try:
14: (4)                 import cffi # noqa: F401
15: (4)                 MISSING_CFFI = False
16: (0)             except ImportError:
17: (4)                 MISSING_CFFI = True
18: (0)             try:
19: (4)                 import ctypes # noqa: F401
20: (4)                 MISSING_CTYPES = False
21: (0)             except ImportError:
22: (4)                 MISSING_CTYPES = False
23: (0)             if sys.flags.optimize > 1:
24: (4)                 MISSING_CFFI = True
25: (0)             pwd = os.path.dirname(os.path.abspath(__file__))
26: (0)             def assert_state_equal(actual, target):
27: (4)                 for key in actual:
28: (8)                     if isinstance(actual[key], dict):
29: (12)                         assert_state_equal(actual[key], target[key])
30: (8)                     elif isinstance(actual[key], np.ndarray):
31: (12)                         assert_array_equal(actual[key], target[key])
32: (8)                     else:
33: (12)                         assert actual[key] == target[key]
34: (0)             def uint32_to_float32(u):
35: (4)                 return ((u >> np.uint32(8)) * (1.0 / 2**24)).astype(np.float32)
36: (0)             def uniform32_from_uint64(x):
37: (4)                 x = np.uint64(x)
38: (4)                 upper = np.array(x >> np.uint64(32), dtype=np.uint32)
39: (4)                 lower = np.uint64(0xffffffff)
40: (4)                 lower = np.array(x & lower, dtype=np.uint32)
41: (4)                 joined = np.column_stack([lower, upper]).ravel()
42: (4)                 return uint32_to_float32(joined)
43: (0)             def uniform32_from_uint53(x):

```

```

44: (4)          x = np.uint64(x) >> np.uint64(16)
45: (4)          x = np.uint32(x & np.uint64(0xffffffff))
46: (4)          return uint32_to_float32(x)
47: (0)          def uniform32_from_uint32(x):
48: (4)              return uint32_to_float32(x)
49: (0)          def uniform32_from_uint(x, bits):
50: (4)              if bits == 64:
51: (8)                  return uniform32_from_uint64(x)
52: (4)              elif bits == 53:
53: (8)                  return uniform32_from_uint53(x)
54: (4)              elif bits == 32:
55: (8)                  return uniform32_from_uint32(x)
56: (4)              else:
57: (8)                  raise NotImplementedError
58: (0)          def uniform_from_uint(x, bits):
59: (4)              if bits in (64, 63, 53):
60: (8)                  return uniform_from_uint64(x)
61: (4)              elif bits == 32:
62: (8)                  return uniform_from_uint32(x)
63: (0)          def uniform_from_uint64(x):
64: (4)              return (x >> np.uint64(11)) * (1.0 / 9007199254740992.0)
65: (0)          def uniform_from_uint32(x):
66: (4)              out = np.empty(len(x) // 2)
67: (4)              for i in range(0, len(x), 2):
68: (8)                  a = x[i] >> 5
69: (8)                  b = x[i + 1] >> 6
70: (8)                  out[i // 2] = (a * 67108864.0 + b) / 9007199254740992.0
71: (4)              return out
72: (0)          def uniform_from_dsfmt(x):
73: (4)              return x.view(np.double) - 1.0
74: (0)          def gauss_from_uint(x, n, bits):
75: (4)              if bits in (64, 63):
76: (8)                  doubles = uniform_from_uint64(x)
77: (4)              elif bits == 32:
78: (8)                  doubles = uniform_from_uint32(x)
79: (4)              else: # bits == 'dsfmt'
80: (8)                  doubles = uniform_from_dsfmt(x)
81: (4)              gauss = []
82: (4)              loc = 0
83: (4)              x1 = x2 = 0.0
84: (4)              while len(gauss) < n:
85: (8)                  r2 = 2
86: (8)                  while r2 >= 1.0 or r2 == 0.0:
87: (12)                      x1 = 2.0 * doubles[loc] - 1.0
88: (12)                      x2 = 2.0 * doubles[loc + 1] - 1.0
89: (12)                      r2 = x1 * x1 + x2 * x2
90: (12)                      loc += 2
91: (8)                  f = np.sqrt(-2.0 * np.log(r2) / r2)
92: (8)                  gauss.append(f * x2)
93: (8)                  gauss.append(f * x1)
94: (4)              return gauss[:n]
95: (0)          def test_seedsequence():
96: (4)              from numpy.random.bit_generator import (ISeedSequence,
97: (44)                                  ISpawnableSeedSequence,
98: (44)                                  SeedlessSeedSequence)
99: (4)              s1 = SeedSequence(range(10), spawn_key=(1, 2), pool_size=6)
100: (4)             s1.spawn(10)
101: (4)             s2 = SeedSequence(**s1.state)
102: (4)             assert_equal(s1.state, s2.state)
103: (4)             assert_equal(s1.n_children_spawned, s2.n_children_spawned)
104: (4)             assert_raises(TypeError, ISeedSequence)
105: (4)             assert_raises(TypeError, ISpawnableSeedSequence)
106: (4)             dummy = SeedlessSeedSequence()
107: (4)             assert_raises(NotImplementedError, dummy.generate_state, 10)
108: (4)             assert len(dummy.spawn(10)) == 10
109: (0)          def test_generator_spawning():
110: (4)              """ Test spawning new generators and bit_generators directly.
111: (4)              """
112: (4)              rng = np.random.default_rng()

```

```

113: (4)           seq = rng.bit_generator.seed_seq
114: (4)           new_ss = seq.spawn(5)
115: (4)           expected_keys = [seq.spawn_key + (i,) for i in range(5)]
116: (4)           assert [c.spawn_key for c in new_ss] == expected_keys
117: (4)           new_bgs = rng.bit_generator.spawn(5)
118: (4)           expected_keys = [seq.spawn_key + (i,) for i in range(5, 10)]
119: (4)           assert [bg.seed_seq.spawn_key for bg in new_bgs] == expected_keys
120: (4)           new_rngs = rng.spawn(5)
121: (4)           expected_keys = [seq.spawn_key + (i,) for i in range(10, 15)]
122: (4)           found_keys = [rng.bit_generator.seed_seq.spawn_key for rng in new_rngs]
123: (4)           assert found_keys == expected_keys
124: (4)           assert new_rngs[0].uniform() != new_rngs[1].uniform()
125: (0) def test_non_spawnable():
126: (4)     from numpy.random.bit_generator import ISeedSequence
127: (4)     class FakeSeedSequence:
128: (8)         def generate_state(self, n_words, dtype=np.uint32):
129: (12)             return np.zeros(n_words, dtype=dtype)
130: (4)     ISeedSequence.register(FakeSeedSequence)
131: (4)     rng = np.random.default_rng(FakeSeedSequence())
132: (4)     with pytest.raises(TypeError, match="The underlying SeedSequence"):
133: (8)         rng.spawn(5)
134: (4)     with pytest.raises(TypeError, match="The underlying SeedSequence"):
135: (8)         rng.bit_generator.spawn(5)
136: (0) class Base:
137: (4)     dtype = np.uint64
138: (4)     data2 = data1 = {}
139: (4)     @classmethod
140: (4)     def setup_class(cls):
141: (8)         cls.bit_generator = PCG64
142: (8)         cls.bits = 64
143: (8)         cls.dtype = np.uint64
144: (8)         cls.seed_error_type = TypeError
145: (8)         cls.invalid_init_types = []
146: (8)         cls.invalid_init_values = []
147: (4)     @classmethod
148: (4)     def _read_csv(cls, filename):
149: (8)         with open(filename) as csv:
150: (12)             seed = csv.readline()
151: (12)             seed = seed.split(',')
152: (12)             seed = [int(s.strip(), 0) for s in seed[1:]]
153: (12)             data = []
154: (12)             for line in csv:
155: (16)                 data.append(int(line.split(',')[-1].strip(), 0))
156: (12)             return {'seed': seed, 'data': np.array(data, dtype=cls.dtype)}
157: (4) def test_raw(self):
158: (8)     bit_generator = self.bit_generator(*self.data1['seed'])
159: (8)     uints = bit_generator.random_raw(1000)
160: (8)     assert_equal(uints, self.data1['data'])
161: (8)     bit_generator = self.bit_generator(*self.data1['seed'])
162: (8)     uints = bit_generator.random_raw()
163: (8)     assert_equal(uints, self.data1['data'][0])
164: (8)     bit_generator = self.bit_generator(*self.data2['seed'])
165: (8)     uints = bit_generator.random_raw(1000)
166: (8)     assert_equal(uints, self.data2['data'])
167: (4) def test_random_raw(self):
168: (8)     bit_generator = self.bit_generator(*self.data1['seed'])
169: (8)     uints = bit_generator.random_raw(output=False)
170: (8)     assert uints is None
171: (8)     uints = bit_generator.random_raw(1000, output=False)
172: (8)     assert uints is None
173: (4) def test_gauss_inv(self):
174: (8)     n = 25
175: (8)     rs = RandomState(self.bit_generator(*self.data1['seed']))
176: (8)     gauss = rs.standard_normal(n)
177: (8)     assert_allclose(gauss,
178: (24)                     gauss_from_uint(self.data1['data'], n, self.bits))
179: (8)     rs = RandomState(self.bit_generator(*self.data2['seed']))
180: (8)     gauss = rs.standard_normal(25)
181: (8)     assert_allclose(gauss,

```

```

182: (24)                                     gauss_from_uint(self.data2['data'], n, self.bits))
183: (4)          def test_uniform_double(self):
184: (8)              rs = Generator(self.bit_generator(*self.data1['seed']))
185: (8)              vals = uniform_from_uint(self.data1['data'], self.bits)
186: (8)              uniforms = rs.random(len(vals))
187: (8)              assert_allclose(uniforms, vals)
188: (8)              assert_equal(uniforms.dtype, np.float64)
189: (8)              rs = Generator(self.bit_generator(*self.data2['seed']))
190: (8)              vals = uniform_from_uint(self.data2['data'], self.bits)
191: (8)              uniforms = rs.random(len(vals))
192: (8)              assert_allclose(uniforms, vals)
193: (8)              assert_equal(uniforms.dtype, np.float64)
194: (4)          def test_uniform_float(self):
195: (8)              rs = Generator(self.bit_generator(*self.data1['seed']))
196: (8)              vals = uniform32_from_uint(self.data1['data'], self.bits)
197: (8)              uniforms = rs.random(len(vals), dtype=np.float32)
198: (8)              assert_allclose(uniforms, vals)
199: (8)              assert_equal(uniforms.dtype, np.float32)
200: (8)              rs = Generator(self.bit_generator(*self.data2['seed']))
201: (8)              vals = uniform32_from_uint(self.data2['data'], self.bits)
202: (8)              uniforms = rs.random(len(vals), dtype=np.float32)
203: (8)              assert_allclose(uniforms, vals)
204: (8)              assert_equal(uniforms.dtype, np.float32)
205: (4)          def test_repr(self):
206: (8)              rs = Generator(self.bit_generator(*self.data1['seed']))
207: (8)              assert 'Generator' in repr(rs)
208: (8)              assert f'{id(rs):#x}'.upper().replace('X', 'x') in repr(rs)
209: (4)          def test_str(self):
210: (8)              rs = Generator(self.bit_generator(*self.data1['seed']))
211: (8)              assert 'Generator' in str(rs)
212: (8)              assert str(self.bit_generator.__name__) in str(rs)
213: (8)              assert f'{id(rs):#x}'.upper().replace('X', 'x') not in str(rs)
214: (4)          def test_pickle(self):
215: (8)              import pickle
216: (8)              bit_generator = self.bit_generator(*self.data1['seed'])
217: (8)              state = bit_generator.state
218: (8)              bitgen_pk1 = pickle.dumps(bit_generator)
219: (8)              reloaded = pickle.loads(bitgen_pk1)
220: (8)              reloaded_state = reloaded.state
221: (8)              assert_array_equal(Generator(bit_generator).standard_normal(1000),
222: (27)                                Generator(reloaded).standard_normal(1000))
223: (8)              assert bit_generator is not reloaded
224: (8)              assert_state_equal(reloaded_state, state)
225: (8)              ss = SeedSequence(100)
226: (8)              aa = pickle.loads(pickle.dumps(ss))
227: (8)              assert_equal(ss.state, aa.state)
228: (4)          def test_invalid_state_type(self):
229: (8)              bit_generator = self.bit_generator(*self.data1['seed'])
230: (8)              with pytest.raises(TypeError):
231: (12)                  bit_generator.state = {'1'}
232: (4)          def test_invalid_state_value(self):
233: (8)              bit_generator = self.bit_generator(*self.data1['seed'])
234: (8)              state = bit_generator.state
235: (8)              state['bit_generator'] = 'otherBitGenerator'
236: (8)              with pytest.raises(ValueError):
237: (12)                  bit_generator.state = state
238: (4)          def test_invalid_init_type(self):
239: (8)              bit_generator = self.bit_generator
240: (8)              for st in self.invalid_init_types:
241: (12)                  with pytest.raises(TypeError):
242: (16)                      bit_generator(*st)
243: (4)          def test_invalid_init_values(self):
244: (8)              bit_generator = self.bit_generator
245: (8)              for st in self.invalid_init_values:
246: (12)                  with pytest.raises((ValueError, OverflowError)):
247: (16)                      bit_generator(*st)
248: (4)          def test_benchmark(self):
249: (8)              bit_generator = self.bit_generator(*self.data1['seed'])
250: (8)              bit_generator._benchmark(1)

```

```

251: (8)             bit_generator._benchmark(1, 'double')
252: (8)             with pytest.raises(ValueError):
253: (12)                 bit_generator._benchmark(1, 'int32')
254: (4)             @pytest.mark.skipif(MISSING_CFFI, reason='cffi not available')
255: (4)             def test_cffi(self):
256: (8)                 bit_generator = self.bit_generator(*self.data1['seed'])
257: (8)                 cffi_interface = bit_generator.cffi
258: (8)                 assert isinstance(cffi_interface, interface)
259: (8)                 other_cffi_interface = bit_generator.cffi
260: (8)                 assert other_cffi_interface is cffi_interface
261: (4)             @pytest.mark.skipif(MISSING_CTYPE, reason='ctypes not available')
262: (4)             def test_ctypes(self):
263: (8)                 bit_generator = self.bit_generator(*self.data1['seed'])
264: (8)                 ctypes_interface = bit_generator.ctypes
265: (8)                 assert isinstance(ctypes_interface, interface)
266: (8)                 other_ctypes_interface = bit_generator.ctypes
267: (8)                 assert other_ctypes_interface is ctypes_interface
268: (4)             def test_getstate(self):
269: (8)                 bit_generator = self.bit_generator(*self.data1['seed'])
270: (8)                 state = bit_generator.state
271: (8)                 alt_state = bit_generator.__getstate__()
272: (8)                 assert state_equal(state, alt_state)
273: (0)             class TestPhilox(Base):
274: (4)                 @classmethod
275: (4)                 def setup_class(cls):
276: (8)                     cls.bit_generator = Philox
277: (8)                     cls.bits = 64
278: (8)                     cls.dtype = np.uint64
279: (8)                     cls.data1 = cls._read_csv(
280: (12)                         join(pwd, './data/philox-testset-1.csv'))
281: (8)                     cls.data2 = cls._read_csv(
282: (12)                         join(pwd, './data/philox-testset-2.csv'))
283: (8)                     cls.seed_error_type = TypeError
284: (8)                     cls.invalid_init_types = []
285: (8)                     cls.invalid_init_values = [(1, None, 1), (-1,), (None, None, 2 ** 257
+ 1)]
286: (4)             def test_set_key(self):
287: (8)                 bit_generator = self.bit_generator(*self.data1['seed'])
288: (8)                 state = bit_generator.state
289: (8)                 keyed = self.bit_generator(counter=state['state']['counter'],
290: (35)                                 key=state['state']['key'])
291: (8)                 assert state_equal(bit_generator.state, keyed.state)
292: (0)             class TestPCG64(Base):
293: (4)                 @classmethod
294: (4)                 def setup_class(cls):
295: (8)                     cls.bit_generator = PCG64
296: (8)                     cls.bits = 64
297: (8)                     cls.dtype = np.uint64
298: (8)                     cls.data1 = cls._read_csv(join(pwd, './data/pcg64-testset-1.csv'))
299: (8)                     cls.data2 = cls._read_csv(join(pwd, './data/pcg64-testset-2.csv'))
300: (8)                     cls.seed_error_type = (ValueError, TypeError)
301: (8)                     cls.invalid_init_types = [(3.2,), ([None],), (1, None)]
302: (8)                     cls.invalid_init_values = [(-1,)]
303: (4)             def test_advance_symmetry(self):
304: (8)                 rs = Generator(self.bit_generator(*self.data1['seed']))
305: (8)                 state = rs.bit_generator.state
306: (8)                 step = -0x9e3779b97f4a7c150000000000000000
307: (8)                 rs.bit_generator.advance(step)
308: (8)                 val_neg = rs.integers(10)
309: (8)                 rs.bit_generator.state = state
310: (8)                 rs.bit_generator.advance(2**128 + step)
311: (8)                 val_pos = rs.integers(10)
312: (8)                 rs.bit_generator.state = state
313: (8)                 rs.bit_generator.advance(10 * 2**128 + step)
314: (8)                 val_big = rs.integers(10)
315: (8)                 assert val_neg == val_pos
316: (8)                 assert val_big == val_pos
317: (4)             def test_advange_large(self):
318: (8)                 rs = Generator(self.bit_generator(38219308213743))

```

```

319: (8)          pcg = rs.bit_generator
320: (8)          state = pcg.state["state"]
321: (8)          initial_state = 287608843259529770491897792873167516365
322: (8)          assert state["state"] == initial_state
323: (8)          pcg.advance(sum(2**i for i in (96, 64, 32, 16, 8, 4, 2, 1)))
324: (8)          state = pcg.state["state"]
325: (8)          advanced_state = 135275564607035429730177404003164635391
326: (8)          assert state["state"] == advanced_state
327: (0)          class TestPCG64DXSM(Base):
328: (4)          @classmethod
329: (4)          def setup_class(cls):
330: (8)          cls.bit_generator = PCG64DXSM
331: (8)          cls.bits = 64
332: (8)          cls.dtype = np.uint64
333: (8)          cls.data1 = cls._read_csv(join(pwd, './data/pcg64dxsm-testset-1.csv'))
334: (8)          cls.data2 = cls._read_csv(join(pwd, './data/pcg64dxsm-testset-2.csv'))
335: (8)          cls.seed_error_type = (ValueError, TypeError)
336: (8)          cls.invalid_init_types = [(3.2,), ([None],), (1, None)]
337: (8)          cls.invalid_init_values = [(-1,)]
338: (4)          def test_advance_symmetry(self):
339: (8)          rs = Generator(self.bit_generator(*self.data1['seed']))
340: (8)          state = rs.bit_generator.state
341: (8)          step = -0x9e3779b97f4a7c15000000000000000000000000
342: (8)          rs.bit_generator.advance(step)
343: (8)          val_neg = rs.integers(10)
344: (8)          rs.bit_generator.state = state
345: (8)          rs.bit_generator.advance(2**128 + step)
346: (8)          val_pos = rs.integers(10)
347: (8)          rs.bit_generator.state = state
348: (8)          rs.bit_generator.advance(10 * 2**128 + step)
349: (8)          val_big = rs.integers(10)
350: (8)          assert val_neg == val_pos
351: (8)          assert val_big == val_pos
352: (4)          def test_advange_large(self):
353: (8)          rs = Generator(self.bit_generator(38219308213743))
354: (8)          pcg = rs.bit_generator
355: (8)          state = pcg.state
356: (8)          initial_state = 287608843259529770491897792873167516365
357: (8)          assert state["state"]["state"] == initial_state
358: (8)          pcg.advance(sum(2**i for i in (96, 64, 32, 16, 8, 4, 2, 1)))
359: (8)          state = pcg.state["state"]
360: (8)          advanced_state = 277778083536782149546677086420637664879
361: (8)          assert state["state"] == advanced_state
362: (0)          class TestMT19937(Base):
363: (4)          @classmethod
364: (4)          def setup_class(cls):
365: (8)          cls.bit_generator = MT19937
366: (8)          cls.bits = 32
367: (8)          cls.dtype = np.uint32
368: (8)          cls.data1 = cls._read_csv(join(pwd, './data/mt19937-testset-1.csv'))
369: (8)          cls.data2 = cls._read_csv(join(pwd, './data/mt19937-testset-2.csv'))
370: (8)          cls.seed_error_type = ValueError
371: (8)          cls.invalid_init_types = []
372: (8)          cls.invalid_init_values = [(-1,)]
373: (4)          def test_seed_float_array(self):
374: (8)          assert_raises(TypeError, self.bit_generator, np.array([np.pi]))
375: (8)          assert_raises(TypeError, self.bit_generator, np.array([-np.pi]))
376: (8)          assert_raises(TypeError, self.bit_generator, np.array([np.pi, -np.pi]))
377: (8)          assert_raises(TypeError, self.bit_generator, np.array([0, np.pi]))
378: (8)          assert_raises(TypeError, self.bit_generator, [np.pi])
379: (8)          assert_raises(TypeError, self.bit_generator, [0, np.pi])
380: (4)          def test_state_tuple(self):
381: (8)          rs = Generator(self.bit_generator(*self.data1['seed']))
382: (8)          bit_generator = rs.bit_generator
383: (8)          state = bit_generator.state
384: (8)          desired = rs.integers(2 ** 16)
385: (8)          tup = (state['bit_generator'], state['state']['key'],
386: (15)             state['state']['pos'])

```

```

387: (8)             bit_generator.state = tup
388: (8)             actual = rs.integers(2 ** 16)
389: (8)             assert_equal(actual, desired)
390: (8)             tup = tup + (0, 0.0)
391: (8)             bit_generator.state = tup
392: (8)             actual = rs.integers(2 ** 16)
393: (8)             assert_equal(actual, desired)
394: (0)          class TestSFC64(Base):
395: (4)          @classmethod
396: (4)          def setup_class(cls):
397: (8)              cls.bit_generator = SFC64
398: (8)              cls.bits = 64
399: (8)              cls.dtype = np.uint64
400: (8)              cls.data1 = cls._read_csv(
401: (12)                  join(pwd, './data/sfc64-testset-1.csv'))
402: (8)              cls.data2 = cls._read_csv(
403: (12)                  join(pwd, './data/sfc64-testset-2.csv'))
404: (8)              cls.seed_error_type = (ValueError, TypeError)
405: (8)              cls.invalid_init_types = [(3.2,), ([None],), (1, None)]
406: (8)              cls.invalid_init_values = [(-1,)]
407: (0)          class TestDefaultRNG:
408: (4)          def test_seed(self):
409: (8)              for args in [(), (None,), (1234,), ([1234, 5678],)]:
410: (12)                  rg = default_rng(*args)
411: (12)                  assert isinstance(rg.bit_generator, PCG64)
412: (4)          def test_passthrough(self):
413: (8)              bg = Philox()
414: (8)              rg = default_rng(bg)
415: (8)              assert rg.bit_generator is bg
416: (8)              rg2 = default_rng(rg)
417: (8)              assert rg2 is rg
418: (8)              assert rg2.bit_generator is bg

```

-----

## File 311 - test\_extending.py:

```

1: (0)          from importlib.util import spec_from_file_location, module_from_spec
2: (0)          import os
3: (0)          import pathlib
4: (0)          import pytest
5: (0)          import shutil
6: (0)          import subprocess
7: (0)          import sys
8: (0)          import sysconfig
9: (0)          import textwrap
10: (0)         import warnings
11: (0)         import numpy as np
12: (0)         from numpy.testing import IS_WASM
13: (0)         try:
14: (4)             import cffi
15: (0)         except ImportError:
16: (4)             cffi = None
17: (0)         if sys.flags.optimize > 1:
18: (4)             cffi = None
19: (0)         try:
20: (4)             with warnings.catch_warnings(record=True) as w:
21: (8)                 warnings.filterwarnings('always', '', DeprecationWarning)
22: (8)                 import numba
23: (0)             except (ImportError, SystemError):
24: (4)                 numba = None
25: (0)             try:
26: (4)                 import cython
27: (4)                 from Cython.Compiler.Version import version as cython_version
28: (0)             except ImportError:
29: (4)                 cython = None
30: (0)             else:
31: (4)                 from numpy._utils import _pep440
32: (4)                 required_version = '0.29.35'

```

```

33: (4)         if _pep440.parse(cython_version) < _pep440.Version(required_version):
34: (8)             cython = None
35: (0)             @pytest.mark.skipif(
36: (8)                 sys.platform == "win32" and sys.maxsize < 2**32,
37: (8)                 reason="Failing in 32-bit Windows wheel build job, skip for now"
38: (0)             )
39: (0)             @pytest.mark.skipif(IS_WASM, reason="Can't start subprocess")
40: (0)             @pytest.mark.skipif(cython is None, reason="requires cython")
41: (0)             @pytest.mark.slow
42: (0)         def test_cython(tmp_path):
43: (4)             import glob
44: (4)             srccdir = os.path.join(os.path.dirname(__file__), '...')
45: (4)             shutil.copytree(srccdir, tmp_path / 'random')
46: (4)             build_dir = tmp_path / 'random' / '_examples' / 'cython'
47: (4)             target_dir = build_dir / "build"
48: (4)             os.makedirs(target_dir, exist_ok=True)
49: (4)             if sys.platform == "win32":
50: (8)                 subprocess.check_call(["meson", "setup",
51: (31)                     "--buildtype=release",
52: (31)                     "--vsenv", str(build_dir)],
53: (30)                     cwd=target_dir,
54: (30)                     )
55: (4)             else:
56: (8)                 subprocess.check_call(["meson", "setup", str(build_dir)],
57: (30)                     cwd=target_dir
58: (30)                     )
59: (4)             subprocess.check_call(["meson", "compile", "-vv"], cwd=target_dir)
60: (4)             g = glob.glob(str(target_dir / "*" / "extending.pyx.c"))
61: (4)             with open(g[0]) as fid:
62: (8)                 txt_to_find = 'NumPy API declarations from "numpy/__init__'
63: (8)                 for i, line in enumerate(fid):
64: (12)                     if txt_to_find in line:
65: (16)                         break
66: (8)                 else:
67: (12)                     assert False, ("Could not find '{}' in C file, "
68: (27)                         "wrong pxd used").format(txt_to_find))
69: (4)             suffix = sysconfig.get_config_var('EXT_SUFFIX')
70: (4)             def load(modname):
71: (8)                 so = (target_dir / modname).with_suffix(suffix)
72: (8)                 spec = spec_from_file_location(modname, so)
73: (8)                 mod = module_from_spec(spec)
74: (8)                 spec.loader.exec_module(mod)
75: (8)                 return mod
76: (4)             load("extending")
77: (4)             load("extending_cpp")
78: (4)             extending_distributions = load("extending_distributions")
79: (4)             from numpy.random import PCG64
80: (4)             values = extending_distributions.uniforms_ex(PCG64(0), 10, 'd')
81: (4)             assert values.shape == (10,)
82: (4)             assert values.dtype == np.float64
83: (0)             @pytest.mark.skipif(numba is None or cffi is None,
84: (20)                             reason="requires numba and cffi")
85: (0)             def test_numba():
86: (4)                 from numpy.random._examples.numba import extending # noqa: F401
87: (0)                 @pytest.mark.skipif(cffi is None, reason="requires cffi")
88: (0)                 def test_cffi():
89: (4)                     from numpy.random._examples.cffi import extending # noqa: F401

```

-----  
File 312 - test\_generator\_mt19937.py:

```

1: (0)         import sys
2: (0)         import hashlib
3: (0)         import pytest
4: (0)         import numpy as np
5: (0)         from numpy.linalg import LinAlgError
6: (0)         from numpy.testing import (
7: (4)             assert_, assert_raises, assert_equal, assert_allclose,

```

```

8: (4)             assert_warnings, assert_no_warnings, assert_array_equal,
9: (4)             assert_array_almost_equal, suppress_warnings, IS_WASM)
10: (0)            from numpy.random import Generator, MT19937, SeedSequence, RandomState
11: (0)            random = Generator(MT19937())
12: (0)            JUMP_TEST_DATA = [
13: (4)              {
14: (8)                "seed": 0,
15: (8)                "steps": 10,
16: (8)                "initial": {"key_sha256":
"bb1636883c2707b51c5b7fc26c6927af4430f2e0785a8c7bc886337f919f9edf", "pos": 9},
17: (8)                  "jumped": {"key_sha256":
"ff682ac12bb140f2d72fba8d3506cf4e46817a0db27aae1683867629031d8d55", "pos": 598},
18: (4)                },
19: (4)              {
20: (8)                "seed":384908324,
21: (8)                "steps":312,
22: (8)                "initial": {"key_sha256":
"16b791a1e04886ccbbb4d448d6ff791267dc458ae599475d08d5cced29d11614", "pos": 311},
23: (8)                  "jumped": {"key_sha256":
"a0110a2cf23b56be0fea8f787a7fc84bef0cb5623003d75b26bdfa1c18002c", "pos": 276},
24: (4)                },
25: (4)              {
26: (8)                "seed": [839438204, 980239840, 859048019, 821],
27: (8)                "steps": 511,
28: (8)                "initial": {"key_sha256":
"d306cf01314d51bd37892d874308200951a35265ede54d200f1e065004c3e9ea", "pos": 510},
29: (8)                  "jumped": {"key_sha256":
"0e00ab449f01a5195a83b4aee0dfbc2ce8d46466a640b92e33977d2e42f777f8", "pos": 475},
30: (4)                },
31: (0)              ]
32: (0)          @pytest.fixture(scope='module', params=[True, False])
33: (0)          def endpoint(request):
34: (4)            return request.param
35: (0)          class TestSeed:
36: (4)            def test_scalar(self):
37: (8)              s = Generator(MT19937(0))
38: (8)              assert_equal(s.integers(1000), 479)
39: (8)              s = Generator(MT19937(4294967295))
40: (8)              assert_equal(s.integers(1000), 324)
41: (4)            def test_array(self):
42: (8)              s = Generator(MT19937(range(10)))
43: (8)              assert_equal(s.integers(1000), 465)
44: (8)              s = Generator(MT19937(np.arange(10)))
45: (8)              assert_equal(s.integers(1000), 465)
46: (8)              s = Generator(MT19937([0]))
47: (8)              assert_equal(s.integers(1000), 479)
48: (8)              s = Generator(MT19937([4294967295]))
49: (8)              assert_equal(s.integers(1000), 324)
50: (4)            def test_seedsequence(self):
51: (8)              s = MT19937(SeedSequence(0))
52: (8)              assert_equal(s.random_raw(1), 2058676884)
53: (4)            def test_invalid_scalar(self):
54: (8)              assert_raises(TypeError, MT19937, -0.5)
55: (8)              assert_raises(ValueError, MT19937, -1)
56: (4)            def test_invalid_array(self):
57: (8)              assert_raises(TypeError, MT19937, [-0.5])
58: (8)              assert_raises(ValueError, MT19937, [-1])
59: (8)              assert_raises(ValueError, MT19937, [1, -2, 4294967296])
60: (4)            def test_noninstantiated_bitgen(self):
61: (8)              assert_raises(ValueError, Generator, MT19937)
62: (0)          class TestBinomial:
63: (4)            def test_n_zero(self):
64: (8)              zeros = np.zeros(2, dtype='int')
65: (8)              for p in [0, .5, 1]:
66: (12)                assert_(random.binomial(0, p) == 0)
67: (12)                assert_array_equal(random.binomial(zeros, p), zeros)
68: (4)            def test_p_is_nan(self):
69: (8)              assert_raises(ValueError, random.binomial, 1, np.nan)
70: (0)          class TestMultinomial:

```

```

71: (4) def test_basic(self):
72: (8)     random.multinomial(100, [0.2, 0.8])
73: (4) def test_zero_probability(self):
74: (8)     random.multinomial(100, [0.2, 0.8, 0.0, 0.0, 0.0])
75: (4) def test_int_negative_interval(self):
76: (8)     assert_(-5 <= random.integers(-5, -1) < -1)
77: (8)     x = random.integers(-5, -1, 5)
78: (8)     assert_(np.all(-5 <= x))
79: (8)     assert_(np.all(x < -1))
80: (4) def test_size(self):
81: (8)     p = [0.5, 0.5]
82: (8)     assert_equal(random.multinomial(1, p, np.uint32(1)).shape, (1, 2))
83: (8)     assert_equal(random.multinomial(1, p, np.uint32(1)).shape, (1, 2))
84: (8)     assert_equal(random.multinomial(1, p, np.uint32(1)).shape, (1, 2))
85: (8)     assert_equal(random.multinomial(1, p, [2, 2]).shape, (2, 2, 2))
86: (8)     assert_equal(random.multinomial(1, p, (2, 2)).shape, (2, 2, 2))
87: (8)     assert_equal(random.multinomial(1, p, np.array((2, 2))).shape,
88: (21)         (2, 2, 2))
89: (8)     assert_raises(TypeError, random.multinomial, 1, p,
90: (22)             float(1))
91: (4) def test_invalid_prob(self):
92: (8)     assert_raises(ValueError, random.multinomial, 100, [1.1, 0.2])
93: (8)     assert_raises(ValueError, random.multinomial, 100, [-.1, 0.9])
94: (4) def test_invalid_n(self):
95: (8)     assert_raises(ValueError, random.multinomial, -1, [0.8, 0.2])
96: (8)     assert_raises(ValueError, random.multinomial, [-1] * 10, [0.8, 0.2])
97: (4) def test_p_non_contiguous(self):
98: (8)     p = np.arange(15.)
99: (8)     p /= np.sum(p[1::3])
100: (8)    pvals = p[1::3]
101: (8)    random = Generator(MT19937(1432985819))
102: (8)    non_contig = random.multinomial(100, pvals=pvals)
103: (8)    random = Generator(MT19937(1432985819))
104: (8)    contig = random.multinomial(100, pvals=np.ascontiguousarray(pvals))
105: (8)    assert_array_equal(non_contig, contig)
106: (4) def test_multinomial_pvals_float32(self):
107: (8)    x = np.array([9.9e-01, 9.9e-01, 1.0e-09, 1.0e-09, 1.0e-09, 1.0e-09,
108: (22)        1.0e-09, 1.0e-09, 1.0e-09, 1.0e-09], dtype=np.float32)
109: (8)    pvals = x / x.sum()
110: (8)    random = Generator(MT19937(1432985819))
111: (8)    match = r"\w\s*pvals array is cast to 64-bit floating"
112: (8)    with pytest.raises(ValueError, match=match):
113: (12)        random.multinomial(1, pvals)
114: (0) class TestMultivariateHypergeometric:
115: (4)     def setup_method(self):
116: (8)         self.seed = 8675309
117: (4)     def test_argument_validation(self):
118: (8)         assert_raises(ValueError, random.multivariate_hypergeometric,
119: (22)             10, 4)
120: (8)         assert_raises(ValueError, random.multivariate_hypergeometric,
121: (22)             [2, 3, 4], -1)
122: (8)         assert_raises(ValueError, random.multivariate_hypergeometric,
123: (22)             [-1, 2, 3], 2)
124: (8)         assert_raises(ValueError, random.multivariate_hypergeometric,
125: (22)             [2, 3, 4], 10)
126: (8)         assert_raises(ValueError, random.multivariate_hypergeometric,
127: (22)             [], 1)
128: (8)         assert_raises(ValueError, random.multivariate_hypergeometric,
129: (22)             [999999999, 101], 5, 1, 'marginals')
130: (8)         int64_info = np.iinfo(np.int64)
131: (8)         max_int64 = int64_info.max
132: (8)         max_int64_index = max_int64 // int64_info.dtype.itemsize
133: (8)         assert_raises(ValueError, random.multivariate_hypergeometric,
134: (22)             [max_int64_index - 100, 101], 5, 1, 'count')
135: (4) @pytest.mark.parametrize('method', ['count', 'marginals'])
136: (4)     def test_edge_cases(self, method):
137: (8)         random = Generator(MT19937(self.seed))
138: (8)         x = random.multivariate_hypergeometric([0, 0, 0], 0, method=method)
139: (8)         assert_array_equal(x, [0, 0, 0])

```

```

140: (8)             x = random.multivariate_hypergeometric([], 0, method=method)
141: (8)             assert_array_equal(x, [])
142: (8)             x = random.multivariate_hypergeometric([], 0, size=1, method=method)
143: (8)             assert_array_equal(x, np.empty((1, 0), dtype=np.int64))
144: (8)             x = random.multivariate_hypergeometric([1, 2, 3], 0, method=method)
145: (8)             assert_array_equal(x, [0, 0, 0])
146: (8)             x = random.multivariate_hypergeometric([9, 0, 0], 3, method=method)
147: (8)             assert_array_equal(x, [3, 0, 0])
148: (8)             colors = [1, 1, 0, 1, 1]
149: (8)             x = random.multivariate_hypergeometric(colors, sum(colors),
150: (47)                           method=method)
151: (8)             assert_array_equal(x, colors)
152: (8)             x = random.multivariate_hypergeometric([3, 4, 5], 12, size=3,
153: (47)                           method=method)
154: (8)             assert_array_equal(x, [[3, 4, 5]]*3)
155: (4) @pytest.mark.parametrize('nsample', [8, 25, 45, 55])
156: (4) @pytest.mark.parametrize('method', ['count', 'marginals'])
157: (4) @pytest.mark.parametrize('size', [5, (2, 3), 150000])
158: (4) def test_typical_cases(self, nsample, method, size):
159: (8)     random = Generator(MT19937(self.seed))
160: (8)     colors = np.array([10, 5, 20, 25])
161: (8)     sample = random.multivariate_hypergeometric(colors, nsample, size,
162: (52)                           method=method)
163: (8)     if isinstance(size, int):
164: (12)         expected_shape = (size,) + colors.shape
165: (8)     else:
166: (12)         expected_shape = size + colors.shape
167: (8)     assert_equal(sample.shape, expected_shape)
168: (8)     assert_((sample >= 0).all())
169: (8)     assert_((sample <= colors).all())
170: (8)     assert_array_equal(sample.sum(axis=-1),
171: (27)                     np.full(size, fill_value=nsample, dtype=int))
172: (8)     if isinstance(size, int) and size >= 100000:
173: (12)         assert_allclose(sample.mean(axis=0),
174: (28)             nsample * colors / colors.sum(),
175: (28)             rtol=1e-3, atol=0.005)
176: (4) def test_repeatability1(self):
177: (8)     random = Generator(MT19937(self.seed))
178: (8)     sample = random.multivariate_hypergeometric([3, 4, 5], 5, size=5,
179: (52)                           method='count')
180: (8)     expected = np.array([[2, 1, 2],
181: (29)                   [2, 1, 2],
182: (29)                   [1, 1, 3],
183: (29)                   [2, 0, 3],
184: (29)                   [2, 1, 2]])
185: (8)     assert_array_equal(sample, expected)
186: (4) def test_repeatability2(self):
187: (8)     random = Generator(MT19937(self.seed))
188: (8)     sample = random.multivariate_hypergeometric([20, 30, 50], 50,
189: (52)                           size=5,
190: (52)                           method='marginals')
191: (8)     expected = np.array([[ 9, 17, 24],
192: (29)                   [ 7, 13, 30],
193: (29)                   [ 9, 15, 26],
194: (29)                   [ 9, 17, 24],
195: (29)                   [12, 14, 24]])
196: (8)     assert_array_equal(sample, expected)
197: (4) def test_repeatability3(self):
198: (8)     random = Generator(MT19937(self.seed))
199: (8)     sample = random.multivariate_hypergeometric([20, 30, 50], 12,
200: (52)                           size=5,
201: (52)                           method='marginals')
202: (8)     expected = np.array([[ 2,  3,  7],
203: (29)                   [ 5,  3,  4],
204: (29)                   [ 2,  5,  5],
205: (29)                   [ 5,  3,  4],
206: (29)                   [ 1,  5,  6]])
207: (8)     assert_array_equal(sample, expected)
208: (0) class TestSetState:

```

```

209: (4)           def setup_method(self):
210: (8)             self.seed = 1234567890
211: (8)             self.rg = Generator(MT19937(self.seed))
212: (8)             self.bit_generator = self.rg.bit_generator
213: (8)             self.state = self.bit_generator.state
214: (8)             self.legacy_state = (self.state['bit_generator'],
215: (29)                           self.state['state']['key'],
216: (29)                           self.state['state']['pos'])
217: (4)           def test_gaussian_reset(self):
218: (8)             old = self.rg.standard_normal(size=3)
219: (8)             self.bit_generator.state = self.state
220: (8)             new = self.rg.standard_normal(size=3)
221: (8)             assert_(np.all(old == new))
222: (4)           def test_gaussian_reset_in_media_res(self):
223: (8)             self.rg.standard_normal()
224: (8)             state = self.bit_generator.state
225: (8)             old = self.rg.standard_normal(size=3)
226: (8)             self.bit_generator.state = state
227: (8)             new = self.rg.standard_normal(size=3)
228: (8)             assert_(np.all(old == new))
229: (4)           def test_negative_binomial(self):
230: (8)             self.rg.negative_binomial(0.5, 0.5)
231: (0)         class TestIntegers:
232: (4)           rfunc = random.integers
233: (4)           itype = [bool, np.int8, np.uint8, np.int16, np.uint16,
234: (13)                         np.int32, np.uint32, np.int64, np.uint64]
235: (4)           def test_unsupported_type(self, endpoint):
236: (8)             assert_raises(TypeError, self.rfunc, 1, endpoint=endpoint,
237: (4)               dtype=float)
238: (8)           def test_bounds_checking(self, endpoint):
239: (12)             for dt in self.itype:
240: (12)               lbnd = 0 if dt is bool else np.iinfo(dt).min
241: (12)               ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
242: (12)               ubnd = ubnd - 1 if endpoint else ubnd
243: (26)               assert_raises(ValueError, self.rfunc, lbnd - 1, ubnd,
244: (12)                             endpoint=endpoint, dtype=dt)
245: (26)               assert_raises(ValueError, self.rfunc, lbnd, ubnd + 1,
246: (12)                             endpoint=endpoint, dtype=dt)
247: (26)               assert_raises(ValueError, self.rfunc, ubnd, lbnd,
248: (12)                             endpoint=endpoint, dtype=dt)
249: (26)               assert_raises(ValueError, self.rfunc, 1, 0, endpoint=endpoint,
250: (12)                             dtype=dt)
251: (26)               assert_raises(ValueError, self.rfunc, [lbnd - 1], ubnd,
252: (12)                             endpoint=endpoint, dtype=dt)
253: (26)               assert_raises(ValueError, self.rfunc, [lbnd], [ubnd + 1],
254: (12)                             endpoint=endpoint, dtype=dt)
255: (26)               assert_raises(ValueError, self.rfunc, [ubnd], [lbnd],
256: (12)                             endpoint=endpoint, dtype=dt)
257: (26)               assert_raises(ValueError, self.rfunc, 1, [0],
258: (12)                             endpoint=endpoint, dtype=dt)
259: (26)               assert_raises(ValueError, self.rfunc, [ubnd+1], [ubnd],
260: (4)                             endpoint=endpoint, dtype=dt)
261: (8)           def test_bounds_checking_array(self, endpoint):
262: (12)             for dt in self.itype:
263: (12)               lbnd = 0 if dt is bool else np.iinfo(dt).min
264: (12)               ubnd = 2 if dt is bool else np.iinfo(dt).max + (not endpoint)
265: (26)               assert_raises(ValueError, self.rfunc, [lbnd - 1] * 2, [ubnd] * 2,
266: (12)                             endpoint=endpoint, dtype=dt)
267: (26)               assert_raises(ValueError, self.rfunc, [lbnd] * 2,
268: (12)                             [ubnd + 1] * 2, endpoint=endpoint, dtype=dt)
269: (26)               assert_raises(ValueError, self.rfunc, ubnd, [lbnd] * 2,
270: (12)                             endpoint=endpoint, dtype=dt)
271: (26)               assert_raises(ValueError, self.rfunc, [1] * 2, 0,
272: (4)                             endpoint=endpoint, dtype=dt)
273: (8)           def test_rng_zero_and_extremes(self, endpoint):
274: (12)             for dt in self.itype:
275: (12)               lbnd = 0 if dt is bool else np.iinfo(dt).min
276: (12)               ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
276: (12)               ubnd = ubnd - 1 if endpoint else ubnd

```

```

277: (12)           is_open = not endpoint
278: (12)           tgt = ubnd - 1
279: (12)           assert_equal(self.rfunc(tgt, tgt + is_open, size=1000,
280: (36)                         endpoint=endpoint, dtype=dt), tgt)
281: (12)           assert_equal(self.rfunc([tgt], tgt + is_open, size=1000,
282: (36)                         endpoint=endpoint, dtype=dt), tgt)
283: (12)           tgt = lbnd
284: (12)           assert_equal(self.rfunc(tgt, tgt + is_open, size=1000,
285: (36)                         endpoint=endpoint, dtype=dt), tgt)
286: (12)           assert_equal(self.rfunc(tgt, [tgt + is_open], size=1000,
287: (36)                         endpoint=endpoint, dtype=dt), tgt)
288: (12)           tgt = (lbnd + ubnd) // 2
289: (12)           assert_equal(self.rfunc(tgt, tgt + is_open, size=1000,
290: (36)                         endpoint=endpoint, dtype=dt), tgt)
291: (12)           assert_equal(self.rfunc([tgt], [tgt + is_open],
292: (36)                         size=1000, endpoint=endpoint, dtype=dt),
293: (25)                           tgt)
294: (4)            def test_rng_zero_and_extremes_array(self, endpoint):
295: (8)              size = 1000
296: (8)              for dt in self.dtype:
297: (12)                lbnd = 0 if dt is bool else np.iinfo(dt).min
298: (12)                ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
299: (12)                ubnd = ubnd - 1 if endpoint else ubnd
300: (12)                tgt = ubnd - 1
301: (12)                assert_equal(self.rfunc([tgt], [tgt + 1],
302: (36)                  size=size, dtype=dt), tgt)
303: (12)                assert_equal(self.rfunc(
304: (16)                  [tgt] * size, [tgt + 1] * size, dtype=dt), tgt)
305: (12)                assert_equal(self.rfunc(
306: (16)                  [tgt] * size, [tgt + 1] * size, size=size, dtype=dt), tgt)
307: (12)                tgt = lbnd
308: (12)                assert_equal(self.rfunc([tgt], [tgt + 1],
309: (36)                  size=size, dtype=dt), tgt)
310: (12)                assert_equal(self.rfunc(
311: (16)                  [tgt] * size, [tgt + 1] * size, dtype=dt), tgt)
312: (12)                assert_equal(self.rfunc(
313: (16)                  [tgt] * size, [tgt + 1] * size, size=size, dtype=dt), tgt)
314: (12)                tgt = (lbnd + ubnd) // 2
315: (12)                assert_equal(self.rfunc([tgt], [tgt + 1],
316: (36)                  size=size, dtype=dt), tgt)
317: (12)                assert_equal(self.rfunc(
318: (16)                  [tgt] * size, [tgt + 1] * size, dtype=dt), tgt)
319: (12)                assert_equal(self.rfunc(
320: (16)                  [tgt] * size, [tgt + 1] * size, size=size, dtype=dt), tgt)
321: (4)            def test_full_range(self, endpoint):
322: (8)              for dt in self.dtype:
323: (12)                lbnd = 0 if dt is bool else np.iinfo(dt).min
324: (12)                ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
325: (12)                ubnd = ubnd - 1 if endpoint else ubnd
326: (12)                try:
327: (16)                  self.rfunc(lbnd, ubnd, endpoint=endpoint, dtype=dt)
328: (12)                except Exception as e:
329: (16)                  raise AssertionError("No error should have been raised, "
330: (37)                      "but one was with the following "
331: (37)                      "message:\n\n%s" % str(e))
332: (4)            def test_full_range_array(self, endpoint):
333: (8)              for dt in self.dtype:
334: (12)                lbnd = 0 if dt is bool else np.iinfo(dt).min
335: (12)                ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
336: (12)                ubnd = ubnd - 1 if endpoint else ubnd
337: (12)                try:
338: (16)                  self.rfunc([lbnd] * 2, [ubnd], endpoint=endpoint, dtype=dt)
339: (12)                except Exception as e:
340: (16)                  raise AssertionError("No error should have been raised, "
341: (37)                      "but one was with the following "
342: (37)                      "message:\n\n%s" % str(e))
343: (4)            def test_in_bounds_fuzz(self, endpoint):
344: (8)              random = Generator(MT19937())
345: (8)              for dt in self.dtype[1:]:

```

```

346: (12)             for ubnd in [4, 8, 16]:
347: (16)                 vals = self.rfunc(2, ubnd - endpoint, size=2 ** 16,
348: (34)                             endpoint=endpoint, dtype=dt)
349: (16)                     assert_(vals.max() < ubnd)
350: (16)                     assert_(vals.min() >= 2)
351: (8)             vals = self.rfunc(0, 2 - endpoint, size=2 ** 16, endpoint=endpoint,
352: (26)                             dtype=bool)
353: (8)                 assert_(vals.max() < 2)
354: (8)                 assert_(vals.min() >= 0)
355: (4)             def test_scalar_array_equiv(self, endpoint):
356: (8)                 for dt in self.dtype:
357: (12)                     lbnd = 0 if dt is bool else np.iinfo(dt).min
358: (12)                     ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
359: (12)                     ubnd = ubnd - 1 if endpoint else ubnd
360: (12)                     size = 1000
361: (12)                     random = Generator(MT19937(1234))
362: (12)                     scalar = random.integers(lbnd, ubnd, size=size, endpoint=endpoint,
363: (32)                             dtype=dt)
364: (12)                     random = Generator(MT19937(1234))
365: (12)                     scalar_array = random.integers([lbnd], [ubnd], size=size,
366: (38)                             endpoint=endpoint, dtype=dt)
367: (12)                     random = Generator(MT19937(1234))
368: (12)                     array = random.integers([lbnd] * size, [ubnd] *
369: (31)                         size, size=size, endpoint=endpoint, dtype=dt)
370: (12)                     assert_array_equal(scalar, scalar_array)
371: (12)                     assert_array_equal(scalar, array)
372: (4)             def test_repeatability(self, endpoint):
373: (8)                 tgt = {'bool':
'053594a9b82d656f967c54869bc6970aa0358cf94ad469c81478459c6a90eee3',
374: (15)                     'int16':
'54de9072b6ee9ff7f20b58329556a46a447a8a29d67db51201bf88baa6e4e5d4',
375: (15)                     'int32':
'd3a0d5efb04542b25ac712e50d21f39ac30f312a5052e9bbb1ad3baa791ac84b',
376: (15)                     'int64':
'14e224389ac4580bfbdc5697d6190b496f91227cf67df60989de3d546389b1',
377: (15)                     'int8':
'0e203226ff3fbcd1580f15da4621e5f7164d0d8d6b51696dd42d004ece2cbec1',
378: (15)                     'uint16':
'54de9072b6ee9ff7f20b58329556a46a447a8a29d67db51201bf88baa6e4e5d4',
379: (15)                     'uint32':
'd3a0d5efb04542b25ac712e50d21f39ac30f312a5052e9bbb1ad3baa791ac84b',
380: (15)                     'uint64':
'14e224389ac4580bfbdc5697d6190b496f91227cf67df60989de3d546389b1',
381: (15)                     'uint8':
'0e203226ff3fbcd1580f15da4621e5f7164d0d8d6b51696dd42d004ece2cbec1'}
382: (8)             for dt in self.dtype[1:]:
383: (12)                 random = Generator(MT19937(1234))
384: (12)                 if sys.byteorder == 'little':
385: (16)                     val = random.integers(0, 6 - endpoint, size=1000,
endpoint=endpoint,
386: (33)                                         dtype=dt)
387: (12)
388: (16)                     val = random.integers(0, 6 - endpoint, size=1000,
endpoint=endpoint,
389: (33)                                         dtype=dt).byteswap()
390: (12)                     res = hashlib.sha256(val).hexdigest()
391: (12)                     assert_(tgt[np.dtype(dt).name] == res)
392: (8)             random = Generator(MT19937(1234))
393: (8)             val = random.integers(0, 2 - endpoint, size=1000, endpoint=endpoint,
394: (25)                             dtype=bool).view(np.int8)
395: (8)             res = hashlib.sha256(val).hexdigest()
396: (8)             assert_(tgt[np.dtype(bool).name] == res)
397: (4)             def test_repeatability_broadcasting(self, endpoint):
398: (8)                 for dt in self.dtype:
399: (12)                     lbnd = 0 if dt in (bool, np.bool_) else np.iinfo(dt).min
400: (12)                     ubnd = 2 if dt in (bool, np.bool_) else np.iinfo(dt).max + 1
401: (12)                     ubnd = ubnd - 1 if endpoint else ubnd
402: (12)                     random = Generator(MT19937(1234))
403: (12)                     val = random.integers(lbnd, ubnd, size=1000, endpoint=endpoint,

```

```

404: (29)                               dtype=dt)
405: (12)                               random = Generator(MT19937(1234))
406: (12)                               val_bc = random.integers([lbnd] * 1000, ubnd, endpoint=endpoint,
407: (32)                               dtype=dt)
408: (12)                               assert_array_equal(val, val_bc)
409: (12)                               random = Generator(MT19937(1234))
410: (12)                               val_bc = random.integers([lbnd] * 1000, [ubnd] * 1000,
411: (32)                               endpoint=endpoint, dtype=dt)
412: (12)                               assert_array_equal(val, val_bc)
413: (4) @pytest.mark.parametrize(
414: (8)     'bound, expected',
415: (8)     [(2**32 - 1, np.array([517043486, 1364798665, 1733884389, 1353720612,
416: (31)         3769704066, 1170797179, 4108474671])),]
417: (9)     (2**32, np.array([517043487, 1364798666, 1733884390, 1353720613,
418: (27)         3769704067, 1170797180, 4108474672])),]
419: (9)     (2**32 + 1, np.array([517043487, 1733884390, 3769704068, 4108474673,
420: (31)         1831631863, 1215661561, 3869512430]))]
421: (4)
422: (4) def test_repeatability_32bit_boundary(self, bound, expected):
423: (8)     for size in [None, len(expected)]:
424: (12)         random = Generator(MT19937(1234))
425: (12)         x = random.integers(bound, size=size)
426: (12)         assert_equal(x, expected if size is not None else expected[0])
427: (4) def test_repeatability_32bit_boundary_broadcasting(self):
428: (8)     desired = np.array([[1622936284, 3620788691, 1659384060],
429: (29)         [1417365545, 760222891, 1909653332],
430: (29)         [3788118662, 660249498, 4092002593]],
431: (28)         [[3625610153, 2979601262, 3844162757],
432: (29)             [685800658, 120261497, 2694012896],
433: (29)             [1207779440, 1586594375, 3854335050]],
434: (28)         [[3004074748, 2310761796, 3012642217],
435: (29)             [2067714190, 2786677879, 1363865881],
436: (29)             [791663441, 1867303284, 2169727960]],
437: (28)         [[1939603804, 1250951100, 298950036],
438: (29)             [1040128489, 3791912209, 3317053765],
439: (29)             [3155528714, 61360675, 2305155588]],
440: (28)         [[817688762, 1335621943, 3288952434],
441: (29)             [1770890872, 1102951817, 1957607470],
442: (29)             [3099996017, 798043451, 48334215]]])
443: (8)     for size in [None, (5, 3, 3)]:
444: (12)         random = Generator(MT19937(12345))
445: (12)         x = random.integers([-1], [0], [1],
446: (32)             [2**32 - 1, 2**32, 2**32 + 1],
447: (32)             size=size)
448: (12)         assert_array_equal(x, desired if size is not None else desired[0])
449: (4) def test_int64_uint64_broadcast_exceptions(self, endpoint):
450: (8)     configs = {np.uint64: ((0, 2**65), (-1, 2**62), (10, 9), (0, 0)),
451: (19)         np.int64: ((0, 2**64), (-2**64), 2**62, (10, 9), (0, 0),
452: (30)             (-2**63-1, -2**63-1))}
453: (8)     for dtype in configs:
454: (12)         for config in configs[dtype]:
455: (16)             low, high = config
456: (16)             high = high - endpoint
457: (16)             low_a = np.array([[low]*10])
458: (16)             high_a = np.array([high] * 10)
459: (16)             assert_raises(ValueError, random.integers, low, high,
460: (30)                 endpoint=endpoint, dtype=dtype)
461: (16)             assert_raises(ValueError, random.integers, low_a, high,
462: (30)                 endpoint=endpoint, dtype=dtype)
463: (16)             assert_raises(ValueError, random.integers, low, high_a,
464: (30)                 endpoint=endpoint, dtype=dtype)
465: (16)             assert_raises(ValueError, random.integers, low_a, high_a,
466: (30)                 endpoint=endpoint, dtype=dtype)
467: (16)             low_o = np.array([[low]*10], dtype=object)
468: (16)             high_o = np.array([high] * 10, dtype=object)
469: (16)             assert_raises(ValueError, random.integers, low_o, high,
470: (30)                 endpoint=endpoint, dtype=dtype)
471: (16)             assert_raises(ValueError, random.integers, low, high_o,
472: (30)                 endpoint=endpoint, dtype=dtype)

```

```

473: (16) assert_raises(ValueError, random.integers, low_o, high_o,
474: (30)                                     endpoint=endpoint, dtype=dtype)
475: (4) def test_int64_uint64_corner_case(self, endpoint):
476: (8)     dt = np.int64
477: (8)     tgt = np.iinfo(np.int64).max
478: (8)     lbnd = np.int64(np.iinfo(np.int64).max)
479: (8)     ubnd = np.uint64(np.iinfo(np.int64).max + 1 - endpoint)
480: (8)     actual = random.integers(lbnd, ubnd, endpoint=endpoint, dtype=dt)
481: (8)     assert_equal(actual, tgt)
482: (4) def test_respect_dtype_singleton(self, endpoint):
483: (8)     for dt in self.dtype:
484: (12)         lbnd = 0 if dt is bool else np.iinfo(dt).min
485: (12)         ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
486: (12)         ubnd = ubnd - 1 if endpoint else ubnd
487: (12)         dt = np.bool_ if dt is bool else dt
488: (12)         sample = self.rfunc(lbnd, ubnd, endpoint=endpoint, dtype=dt)
489: (12)         assert_equal(sample.dtype, dt)
490: (8)     for dt in (bool, int):
491: (12)         lbnd = 0 if dt is bool else np.iinfo(dt).min
492: (12)         ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
493: (12)         ubnd = ubnd - 1 if endpoint else ubnd
494: (12)         sample = self.rfunc(lbnd, ubnd, endpoint=endpoint, dtype=dt)
495: (12)         assert not hasattr(sample, 'dtype')
496: (12)         assert_equal(type(sample), dt)
497: (4) def test_respect_dtype_array(self, endpoint):
498: (8)     for dt in self.dtype:
499: (12)         lbnd = 0 if dt is bool else np.iinfo(dt).min
500: (12)         ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
501: (12)         ubnd = ubnd - 1 if endpoint else ubnd
502: (12)         dt = np.bool_ if dt is bool else dt
503: (12)         sample = self.rfunc([lbnd], [ubnd], endpoint=endpoint, dtype=dt)
504: (12)         assert_equal(sample.dtype, dt)
505: (12)         sample = self.rfunc([lbnd] * 2, [ubnd] * 2, endpoint=endpoint,
506: (32)                           dtype=dt)
507: (12)         assert_equal(sample.dtype, dt)
508: (4) def test_zero_size(self, endpoint):
509: (8)     for dt in self.dtype:
510: (12)         sample = self.rfunc(0, 0, (3, 0, 4), endpoint=endpoint, dtype=dt)
511: (12)         assert sample.shape == (3, 0, 4)
512: (12)         assert sample.dtype == dt
513: (12)         assert self.rfunc(0, -10, 0, endpoint=endpoint,
514: (30)                           dtype=dt).shape == (0,)
515: (12)         assert_equal(random.integers(0, 0, size=(3, 0, 4)).shape,
516: (25)                           (3, 0, 4))
517: (12)         assert_equal(random.integers(0, -10, size=0).shape, (0,))
518: (12)         assert_equal(random.integers(10, 10, size=0).shape, (0,)))
519: (4) def test_error_byteorder(self):
520: (8)     other_byteord_dt = '<i4' if sys.byteorder == 'big' else '>i4'
521: (8)     with pytest.raises(ValueError):
522: (12)         random.integers(0, 200, size=10, dtype=other_byteord_dt)
523: (4) @pytest.mark.slow
524: (4) @pytest.mark.parametrize('sample_size,high,dtype,chi2max',
525: (8)     [(5000000, 5, np.int8, 125.0),           # p-value ~4.6e-25
526: (9)      (5000000, 7, np.uint8, 150.0),        # p-value ~7.7e-30
527: (9)      (10000000, 2500, np.int16, 3300.0),   # p-value ~3.0e-25
528: (9)      (50000000, 5000, np.uint16, 6500.0),  # p-value ~3.5e-25
529: (8)    ])
530: (4) def test_integers_small_dtype_chisquared(self, sample_size, high,
531: (45)                               dtype, chi2max):
532: (8)     samples = random.integers(high, size=sample_size, dtype=dtype)
533: (8)     values, counts = np.unique(samples, return_counts=True)
534: (8)     expected = sample_size / high
535: (8)     chi2 = ((counts - expected)**2 / expected).sum()
536: (8)     assert chi2 < chi2max
537: (0) class TestRandomDist:
538: (4)     def setup_method(self):
539: (8)         self.seed = 1234567890
540: (4)     def test_integers(self):
541: (8)         random = Generator(MT19937(self.seed))

```

```

542: (8)           actual = random.integers(-99, 99, size=(3, 2))
543: (8)           desired = np.array([-80, -56], [41, 37], [-83, -16]))
544: (8)           assert_array_equal(actual, desired)
545: (4) def test_integers_masked(self):
546: (8)           random = Generator(MT19937(self.seed))
547: (8)           actual = random.integers(0, 99, size=(3, 2), dtype=np.uint32)
548: (8)           desired = np.array([[9, 21], [70, 68], [8, 41]], dtype=np.uint32)
549: (8)           assert_array_equal(actual, desired)
550: (4) def test_integers_closed(self):
551: (8)           random = Generator(MT19937(self.seed))
552: (8)           actual = random.integers(-99, 99, size=(3, 2), endpoint=True)
553: (8)           desired = np.array([-80, -56], [41, 38], [-83, -15]))
554: (8)           assert_array_equal(actual, desired)
555: (4) def test_integers_max_int(self):
556: (8)           actual = random.integers(np.iinfo('l').max, np.iinfo('l').max,
557: (33)                           endpoint=True)
558: (8)           desired = np.iinfo('l').max
559: (8)           assert_equal(actual, desired)
560: (4) def test_random(self):
561: (8)           random = Generator(MT19937(self.seed))
562: (8)           actual = random.random((3, 2))
563: (8)           desired = np.array([0.096999199829214, 0.707517457682192],
564: (28)                         [0.084364834598269, 0.767731206553125],
565: (28)                         [0.665069021359413, 0.715487190596693]])
566: (8)           assert_array_almost_equal(actual, desired, decimal=15)
567: (8)           random = Generator(MT19937(self.seed))
568: (8)           actual = random.random()
569: (8)           assert_array_almost_equal(actual, desired[0, 0], decimal=15)
570: (4) def test_random_float(self):
571: (8)           random = Generator(MT19937(self.seed))
572: (8)           actual = random.random((3, 2))
573: (8)           desired = np.array([0.0969992, 0.70751746],
574: (28)                         [0.08436483, 0.76773121],
575: (28)                         [0.66506902, 0.71548719])
576: (8)           assert_array_almost_equal(actual, desired, decimal=7)
577: (4) def test_random_float_scalar(self):
578: (8)           random = Generator(MT19937(self.seed))
579: (8)           actual = random.random(dtype=np.float32)
580: (8)           desired = 0.0969992
581: (8)           assert_array_almost_equal(actual, desired, decimal=7)
582: (4) @pytest.mark.parametrize('dtype, uint_view_type',
583: (29)                 [(np.float32, np.uint32),
584: (30)                   (np.float64, np.uint64)])
585: (4) def test_random_distribution_of_lsb(self, dtype, uint_view_type):
586: (8)           random = Generator(MT19937(self.seed))
587: (8)           sample = random.random(100000, dtype=dtype)
588: (8)           num_ones_in_lsb = np.count_nonzero(sample.view(uint_view_type) & 1)
589: (8)           assert 24100 < num_ones_in_lsb < 25900
590: (4) def test_random_unsupported_type(self):
591: (8)           assert_raises(TypeError, random.random, dtype='int32')
592: (4) def test_choice_uniform_replace(self):
593: (8)           random = Generator(MT19937(self.seed))
594: (8)           actual = random.choice(4, 4)
595: (8)           desired = np.array([0, 0, 2, 2], dtype=np.int64)
596: (8)           assert_array_equal(actual, desired)
597: (4) def test_choice_nonuniform_replace(self):
598: (8)           random = Generator(MT19937(self.seed))
599: (8)           actual = random.choice(4, 4, p=[0.4, 0.4, 0.1, 0.1])
600: (8)           desired = np.array([0, 1, 0, 1], dtype=np.int64)
601: (8)           assert_array_equal(actual, desired)
602: (4) def test_choice_uniform_noreplace(self):
603: (8)           random = Generator(MT19937(self.seed))
604: (8)           actual = random.choice(4, 3, replace=False)
605: (8)           desired = np.array([2, 0, 3], dtype=np.int64)
606: (8)           assert_array_equal(actual, desired)
607: (8)           actual = random.choice(4, 4, replace=False, shuffle=False)
608: (8)           desired = np.arange(4, dtype=np.int64)
609: (8)           assert_array_equal(actual, desired)
610: (4) def test_choice_nonuniform_noreplace(self):

```

```

611: (8)             random = Generator(MT19937(self.seed))
612: (8)             actual = random.choice(4, 3, replace=False, p=[0.1, 0.3, 0.5, 0.1])
613: (8)             desired = np.array([0, 2, 3], dtype=np.int64)
614: (8)             assert_array_equal(actual, desired)
615: (4)             def test_choice_noninteger(self):
616: (8)                 random = Generator(MT19937(self.seed))
617: (8)                 actual = random.choice(['a', 'b', 'c', 'd'], 4)
618: (8)                 desired = np.array(['a', 'a', 'c', 'c'])
619: (8)                 assert_array_equal(actual, desired)
620: (4)             def test_choice_multidimensional_default_axis(self):
621: (8)                 random = Generator(MT19937(self.seed))
622: (8)                 actual = random.choice([[0, 1], [2, 3], [4, 5], [6, 7]], 3)
623: (8)                 desired = np.array([[0, 1], [0, 1], [4, 5]])
624: (8)                 assert_array_equal(actual, desired)
625: (4)             def test_choice_multidimensional_custom_axis(self):
626: (8)                 random = Generator(MT19937(self.seed))
627: (8)                 actual = random.choice([[0, 1], [2, 3], [4, 5], [6, 7]], 1, axis=1)
628: (8)                 desired = np.array([[0], [2], [4], [6]])
629: (8)                 assert_array_equal(actual, desired)
630: (4)             def test_choice_exceptions(self):
631: (8)                 sample = random.choice
632: (8)                 assert_raises(ValueError, sample, -1, 3)
633: (8)                 assert_raises(ValueError, sample, 3., 3)
634: (8)                 assert_raises(ValueError, sample, [], 3)
635: (8)                 assert_raises(ValueError, sample, [1, 2, 3, 4], 3,
636: (22)                   p=[[0.25, 0.25], [0.25, 0.25]])
637: (8)                 assert_raises(ValueError, sample, [1, 2], 3, p=[0.4, 0.4, 0.2])
638: (8)                 assert_raises(ValueError, sample, [1, 2], 3, p=[1.1, -0.1])
639: (8)                 assert_raises(ValueError, sample, [1, 2], 3, p=[0.4, 0.4])
640: (8)                 assert_raises(ValueError, sample, [1, 2, 3], 4, replace=False)
641: (8)                 assert_raises(ValueError, sample, [1, 2, 3], -2, replace=False)
642: (8)                 assert_raises(ValueError, sample, [1, 2, 3], (-1,), replace=False)
643: (8)                 assert_raises(ValueError, sample, [1, 2, 3], (-1, 1), replace=False)
644: (8)                 assert_raises(ValueError, sample, [1, 2, 3], 2,
645: (22)                   replace=False, p=[1, 0, 0])
646: (4)             def test_choice_return_shape(self):
647: (8)                 p = [0.1, 0.9]
648: (8)                 assert_(np.isscalar(random.choice(2, replace=True)))
649: (8)                 assert_(np.isscalar(random.choice(2, replace=False)))
650: (8)                 assert_(np.isscalar(random.choice(2, replace=True, p=p)))
651: (8)                 assert_(np.isscalar(random.choice(2, replace=False, p=p)))
652: (8)                 assert_(np.isscalar(random.choice([1, 2], replace=True)))
653: (8)                 assert_(random.choice([None], replace=True) is None)
654: (8)                 a = np.array([1, 2])
655: (8)                 arr = np.empty(1, dtype=object)
656: (8)                 arr[0] = a
657: (8)                 assert_(random.choice(arr, replace=True) is a)
658: (8)                 s = tuple()
659: (8)                 assert_(not np.isscalar(random.choice(2, s, replace=True)))
660: (8)                 assert_(not np.isscalar(random.choice(2, s, replace=False)))
661: (8)                 assert_(not np.isscalar(random.choice(2, s, replace=True, p=p)))
662: (8)                 assert_(not np.isscalar(random.choice(2, s, replace=False, p=p)))
663: (8)                 assert_(not np.isscalar(random.choice([1, 2], s, replace=True)))
664: (8)                 assert_(random.choice([None], s, replace=True).ndim == 0)
665: (8)                 a = np.array([1, 2])
666: (8)                 arr = np.empty(1, dtype=object)
667: (8)                 arr[0] = a
668: (8)                 assert_(random.choice(arr, s, replace=True).item() is a)
669: (8)                 s = (2, 3)
670: (8)                 p = [0.1, 0.1, 0.1, 0.1, 0.4, 0.2]
671: (8)                 assert_equal(random.choice(6, s, replace=True).shape, s)
672: (8)                 assert_equal(random.choice(6, s, replace=False).shape, s)
673: (8)                 assert_equal(random.choice(6, s, replace=True, p=p).shape, s)
674: (8)                 assert_equal(random.choice(6, s, replace=False, p=p).shape, s)
675: (8)                 assert_equal(random.choice(np.arange(6), s, replace=True).shape, s)
676: (8)                 assert_equal(random.integers(0, 0, size=(3, 0, 4)).shape, (3, 0, 4))
677: (8)                 assert_equal(random.integers(0, -10, size=0).shape, (0,))
678: (8)                 assert_equal(random.integers(10, 10, size=0).shape, (0,))
679: (8)                 assert_equal(random.choice(0, size=0).shape, (0,))
```

```

680: (8) assert_equal(random.choice([], size=(0,)).shape, (0,))
681: (8) assert_equal(random.choice(['a', 'b'], size=(3, 0, 4)).shape,
682: (21) (3, 0, 4))
683: (8) assert_raises(ValueError, random.choice, [], 10)
684: (4) def test_choice_nan_probabilities(self):
685: (8)     a = np.array([42, 1, 2])
686: (8)     p = [None, None, None]
687: (8)     assert_raises(ValueError, random.choice, a, p=p)
688: (4) def test_choice_p_non_contiguous(self):
689: (8)     p = np.ones(10) / 5
690: (8)     p[1::2] = 3.0
691: (8)     random = Generator(MT19937(self.seed))
692: (8)     non_contig = random.choice(5, 3, p=p[::2])
693: (8)     random = Generator(MT19937(self.seed))
694: (8)     contig = random.choice(5, 3, p=np.ascontiguousarray(p[::2]))
695: (8)     assert_array_equal(non_contig, contig)
696: (4) def test_choice_return_type(self):
697: (8)     p = np.ones(4) / 4.
698: (8)     actual = random.choice(4, 2)
699: (8)     assert actual.dtype == np.int64
700: (8)     actual = random.choice(4, 2, replace=False)
701: (8)     assert actual.dtype == np.int64
702: (8)     actual = random.choice(4, 2, p=p)
703: (8)     assert actual.dtype == np.int64
704: (8)     actual = random.choice(4, 2, p=p, replace=False)
705: (8)     assert actual.dtype == np.int64
706: (4) def test_choice_large_sample(self):
707: (8)     choice_hash =
'4266599d12bfcfb815213303432341c06b4349f5455890446578877bb322e222'
708: (8)     random = Generator(MT19937(self.seed))
709: (8)     actual = random.choice(10000, 5000, replace=False)
710: (8)     if sys.byteorder != 'little':
711: (12)         actual = actual.byteswap()
712: (8)     res = hashlib.sha256(actual.view(np.int8)).hexdigest()
713: (8)     assert_(choice_hash == res)
714: (4) def test_bytes(self):
715: (8)     random = Generator(MT19937(self.seed))
716: (8)     actual = random.bytes(10)
717: (8)     desired = b'\x86\xf0\xd4\x18\xe1\x81\t8%\xdd'
718: (8)     assert_equal(actual, desired)
719: (4) def test_shuffle(self):
720: (8)     for conv in [lambda x: np.array([]),
721: (21)         lambda x: x,
722: (21)         lambda x: np.asarray(x).astype(np.int8),
723: (21)         lambda x: np.asarray(x).astype(np.float32),
724: (21)         lambda x: np.asarray(x).astype(np.complex64),
725: (21)         lambda x: np.asarray(x).astype(object),
726: (21)         lambda x: [(i, i) for i in x],
727: (21)         lambda x: np.asarray([[i, i] for i in x]),
728: (21)         lambda x: np.vstack([x, x]).T,
729: (21)         lambda x: (np.asarray([(i, i) for i in x],
730: (43)             [("a", int), ("b", int)])
731: (32)                 .view(np.recarray)),
732: (21)                 lambda x: np.asarray([(i, i) for i in x],
733: (42)                     [("a", object, (1,)), ("b", np.int32, (1,))])]:
734: (43)                     random = Generator(MT19937(self.seed))
735: (12)                     alist = conv([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
736: (12)                     random.shuffle(alist)
737: (12)                     actual = alist
738: (12)                     desired = conv([4, 1, 9, 8, 0, 5, 3, 6, 2, 7])
739: (12)                     assert_array_equal(actual, desired)
740: (12) def test_shuffle_custom_axis(self):
741: (4)     random = Generator(MT19937(self.seed))
742: (8)     actual = np.arange(16).reshape((4, 4))
743: (8)     random.shuffle(actual, axis=1)
744: (8)     desired = np.array([[0, 3, 1, 2],
745: (8)                     [4, 7, 5, 6],
746: (28)                     [8, 11, 9, 10],
747: (28)

```

```

748: (28)                               [12, 15, 13, 14]])
749: (8)       assert_array_equal(actual, desired)
750: (8)       random = Generator(MT19937(self.seed))
751: (8)       actual = np.arange(16).reshape((4, 4))
752: (8)       random.shuffle(actual, axis=-1)
753: (8)       assert_array_equal(actual, desired)
754: (4) def test_shuffle_custom_axis_empty(self):
755: (8)     random = Generator(MT19937(self.seed))
756: (8)     desired = np.array([]).reshape((0, 6))
757: (8)     for axis in (0, 1):
758: (12)         actual = np.array([]).reshape((0, 6))
759: (12)         random.shuffle(actual, axis=axis)
760: (12)         assert_array_equal(actual, desired)
761: (4) def test_shuffle_axis_nonsquare(self):
762: (8)     y1 = np.arange(20).reshape(2, 10)
763: (8)     y2 = y1.copy()
764: (8)     random = Generator(MT19937(self.seed))
765: (8)     random.shuffle(y1, axis=1)
766: (8)     random = Generator(MT19937(self.seed))
767: (8)     random.shuffle(y2.T)
768: (8)     assert_array_equal(y1, y2)
769: (4) def test_shuffle_masked(self):
770: (8)     a = np.ma.masked_values(np.reshape(range(20), (5, 4)) % 3 - 1, -1)
771: (8)     b = np.ma.masked_values(np.arange(20) % 3 - 1, -1)
772: (8)     a_orig = a.copy()
773: (8)     b_orig = b.copy()
774: (8)     for i in range(50):
775: (12)         random.shuffle(a)
776: (12)         assert_equal(
777: (16)             sorted(a.data[~a.mask]), sorted(a_orig.data[~a_orig.mask]))
778: (12)         random.shuffle(b)
779: (12)         assert_equal(
780: (16)             sorted(b.data[~b.mask]), sorted(b_orig.data[~b_orig.mask]))
781: (4) def test_shuffle_exceptions(self):
782: (8)     random = Generator(MT19937(self.seed))
783: (8)     arr = np.arange(10)
784: (8)     assert_raises(np.AxisError, random.shuffle, arr, 1)
785: (8)     arr = np.arange(9).reshape((3, 3))
786: (8)     assert_raises(np.AxisError, random.shuffle, arr, 3)
787: (8)     assert_raises(TypeError, random.shuffle, arr, slice(1, 2, None))
788: (8)     arr = [[1, 2, 3], [4, 5, 6]]
789: (8)     assert_raises(NotImplementedError, random.shuffle, arr, 1)
790: (8)     arr = np.array(3)
791: (8)     assert_raises(TypeError, random.shuffle, arr)
792: (8)     arr = np.ones((3, 2))
793: (8)     assert_raises(np.AxisError, random.shuffle, arr, 2)
794: (4) def test_shuffle_not_writeable(self):
795: (8)     random = Generator(MT19937(self.seed))
796: (8)     a = np.zeros(5)
797: (8)     a.flags.writeable = False
798: (8)     with pytest.raises(ValueError, match='read-only'):
799: (12)         random.shuffle(a)
800: (4) def test_permutation(self):
801: (8)     random = Generator(MT19937(self.seed))
802: (8)     alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
803: (8)     actual = random.permutation(alist)
804: (8)     desired = [4, 1, 9, 8, 0, 5, 3, 6, 2, 7]
805: (8)     assert_array_equal(actual, desired)
806: (8)     random = Generator(MT19937(self.seed))
807: (8)     arr_2d = np.atleast_2d([1, 2, 3, 4, 5, 6, 7, 8, 9, 0]).T
808: (8)     actual = random.permutation(arr_2d)
809: (8)     assert_array_equal(actual, np.atleast_2d(desired).T)
810: (8)     bad_x_str = "abcd"
811: (8)     assert_raises(np.AxisError, random.permutation, bad_x_str)
812: (8)     bad_x_float = 1.2
813: (8)     assert_raises(np.AxisError, random.permutation, bad_x_float)
814: (8)     random = Generator(MT19937(self.seed))
815: (8)     integer_val = 10
816: (8)     desired = [3, 0, 8, 7, 9, 4, 2, 5, 1, 6]

```

```

817: (8)             actual = random.permutation(integer_val)
818: (8)             assert_array_equal(actual, desired)
819: (4)             def test_permutation_custom_axis(self):
820: (8)                 a = np.arange(16).reshape((4, 4))
821: (8)                 desired = np.array([[ 0,  3,  1,  2],
822: (28)                         [ 4,  7,  5,  6],
823: (28)                         [ 8, 11,  9, 10],
824: (28)                         [12, 15, 13, 14]])
825: (8)                 random = Generator(MT19937(self.seed))
826: (8)                 actual = random.permutation(a, axis=1)
827: (8)                 assert_array_equal(actual, desired)
828: (8)                 random = Generator(MT19937(self.seed))
829: (8)                 actual = random.permutation(a, axis=-1)
830: (8)                 assert_array_equal(actual, desired)
831: (4)             def test_permutation_exceptions(self):
832: (8)                 random = Generator(MT19937(self.seed))
833: (8)                 arr = np.arange(10)
834: (8)                 assert_raises(np.AxisError, random.permutation, arr, 1)
835: (8)                 arr = np.arange(9).reshape((3, 3))
836: (8)                 assert_raises(np.AxisError, random.permutation, arr, 3)
837: (8)                 assert_raises(TypeError, random.permutation, arr, slice(1, 2, None))
838: (4)             @pytest.mark.parametrize("dtype", [int, object])
839: (4)             @pytest.mark.parametrize("axis, expected",
840: (29)                         [(None, np.array([[3, 7, 0, 9, 10, 11],
841: (47)                             [8, 4, 2, 5, 1, 6]])),
842: (30)                             (0, np.array([[6, 1, 2, 9, 10, 11],
843: (44)                                 [0, 7, 8, 3, 4, 5]])),
844: (30)                             (1, np.array([[ 5, 3, 4, 0, 2, 1],
845: (44)                                 [11, 9, 10, 6, 8, 7]]))])
846: (4)             def test_permuted(self, dtype, axis, expected):
847: (8)                 random = Generator(MT19937(self.seed))
848: (8)                 x = np.arange(12).reshape(2, 6).astype(dtype)
849: (8)                 random.permuted(x, axis=axis, out=x)
850: (8)                 assert_array_equal(x, expected)
851: (8)                 random = Generator(MT19937(self.seed))
852: (8)                 x = np.arange(12).reshape(2, 6).astype(dtype)
853: (8)                 y = random.permuted(x, axis=axis)
854: (8)                 assert y.dtype == dtype
855: (8)                 assert_array_equal(y, expected)
856: (4)             def test_permuted_with_strides(self):
857: (8)                 random = Generator(MT19937(self.seed))
858: (8)                 x0 = np.arange(22).reshape(2, 11)
859: (8)                 x1 = x0.copy()
860: (8)                 x = x0[:, ::3]
861: (8)                 y = random.permuted(x, axis=1, out=x)
862: (8)                 expected = np.array([[0, 9, 3, 6],
863: (29)                               [14, 20, 11, 17]])
864: (8)                 assert_array_equal(y, expected)
865: (8)                 x1[:, ::3] = expected
866: (8)                 assert_array_equal(x1, x0)
867: (4)             def test_permuted_empty(self):
868: (8)                 y = random.permuted([])
869: (8)                 assert_array_equal(y, [])
870: (4)             @pytest.mark.parametrize('outshape', [(2, 3), 5])
871: (4)             def test_permuted_out_with_wrong_shape(self, outshape):
872: (8)                 a = np.array([1, 2, 3])
873: (8)                 out = np.zeros(outshape, dtype=a.dtype)
874: (8)                 with pytest.raises(ValueError, match='same shape'):
875: (12)                     random.permuted(a, out=out)
876: (4)             def test_permuted_out_with_wrong_type(self):
877: (8)                 out = np.zeros((3, 5), dtype=np.int32)
878: (8)                 x = np.ones((3, 5))
879: (8)                 with pytest.raises(TypeError, match='Cannot cast'):
880: (12)                     random.permuted(x, axis=1, out=out)
881: (4)             def test_permuted_not_writeable(self):
882: (8)                 x = np.zeros((2, 5))
883: (8)                 x.flags.writeable = False
884: (8)                 with pytest.raises(ValueError, match='read-only'):
885: (12)                     random.permuted(x, axis=1, out=x)

```

```

886: (4)
887: (8)
888: (8)
889: (8)
890: (12)
891: (13)
892: (13)
893: (8)
894: (4)
895: (8)
896: (8)
897: (8)
898: (28)
899: (28)
900: (8)
901: (8)
902: (8)
903: (8)
904: (8)
905: (4)
906: (8)
907: (8)
908: (8)
909: (28)
910: (28)
911: (8)
912: (4)
913: (8)
914: (8)
915: (8)
916: (8)
917: (29)
918: (28)
919: (29)
920: (28)
921: (29)
922: (8)
923: (8)
924: (8)
925: (8)
926: (8)
927: (8)
928: (8)
929: (4)
930: (8)
931: (8)
932: (8)
933: (8)
934: (8)
935: (8)
936: (8)
937: (8)
938: (4)
939: (8)
940: (8)
941: (8)
942: (8)
943: (8)
944: (8)
945: (4)
946: (8)
947: (8)
948: (8)
949: (8)
950: (8)
951: (8)
952: (34)
953: (8)

def test_beta(self):
    random = Generator(MT19937(self.seed))
    actual = random.beta(.1, .9, size=(3, 2))
    desired = np.array([
        [1.083029353267698e-10, 2.449965303168024e-11],
        [2.397085162969853e-02, 3.590779671820755e-08],
        [2.830254190078299e-04, 1.744709918330393e-01]])
    assert_array_almost_equal(actual, desired, decimal=15)

def test_binomial(self):
    random = Generator(MT19937(self.seed))
    actual = random.binomial(100.123, .456, size=(3, 2))
    desired = np.array([[42, 41],
                       [42, 48],
                       [44, 50]])
    assert_array_equal(actual, desired)

random = Generator(MT19937(self.seed))
actual = random.binomial(100.123, .456)
desired = 42
assert_array_equal(actual, desired)

def test_chisquare(self):
    random = Generator(MT19937(self.seed))
    actual = random.chisquare(50, size=(3, 2))
    desired = np.array([[32.9850547060149, 39.0219480493301],
                       [56.2006134779419, 57.3474165711485],
                       [55.4243733880198, 55.4209797925213]])
    assert_array_almost_equal(actual, desired, decimal=13)

def test_dirichlet(self):
    random = Generator(MT19937(self.seed))
    alpha = np.array([51.72840233779265162, 39.74494232180943953])
    actual = random.dirichlet(alpha, size=(3, 2))
    desired = np.array([[0.5439892869558927, 0.45601071304410745],
                       [0.5588917345860708, 0.4411082654139292],
                       [[0.5632074165063435, 0.43679258349365657],
                        [0.54862581112627, 0.45137418887373015]],
                       [[0.49961831357047226, 0.5003816864295278],
                        [0.52374806183482, 0.47625193816517997]]])
    assert_array_almost_equal(actual, desired, decimal=15)

bad_alpha = np.array([5.4e-01, -1.0e-16])
assert_raises(ValueError, random.dirichlet, bad_alpha)
random = Generator(MT19937(self.seed))
alpha = np.array([51.72840233779265162, 39.74494232180943953])
actual = random.dirichlet(alpha)
assert_array_almost_equal(actual, desired[0, 0], decimal=15)

def test_dirichlet_size(self):
    p = np.array([51.72840233779265162, 39.74494232180943953])
    assert_equal(random.dirichlet(p, np.uint32(1)).shape, (1, 2))
    assert_equal(random.dirichlet(p, np.uint32(1)).shape, (1, 2))
    assert_equal(random.dirichlet(p, np.uint32(1)).shape, (1, 2))
    assert_equal(random.dirichlet(p, [2, 2]).shape, (2, 2, 2))
    assert_equal(random.dirichlet(p, (2, 2)).shape, (2, 2, 2))
    assert_equal(random.dirichlet(p, np.array((2, 2))).shape, (2, 2, 2))
    assert_raises(TypeError, random.dirichlet, p, float(1))

def test_dirichlet_bad_alpha(self):
    alpha = np.array([5.4e-01, -1.0e-16])
    assert_raises(ValueError, random.dirichlet, alpha)
    assert_raises(ValueError, random.dirichlet, [[5, 1]])
    assert_raises(ValueError, random.dirichlet, [[5], [1]])
    assert_raises(ValueError, random.dirichlet, [[[5], [1]], [[1], [5]]])
    assert_raises(ValueError, random.dirichlet, np.array([[5, 1], [1, 5]]))

def test_dirichlet_alpha_non_contiguous(self):
    a = np.array([51.72840233779265162, -1.0, 39.74494232180943953])
    alpha = a[::-2]
    random = Generator(MT19937(self.seed))
    non_contig = random.dirichlet(alpha, size=(3, 2))
    random = Generator(MT19937(self.seed))
    contig = random.dirichlet(np.ascontiguousarray(alpha),
                             size=(3, 2))
    assert_array_almost_equal(non_contig, contig)

```

```

954: (4) def test_dirichlet_small_alpha(self):
955: (8)     eps = 1.0e-9 # 1.0e-10 -> runtime x 10; 1e-11 -> runtime x 200, etc.
956: (8)     alpha = eps * np.array([1., 1.0e-3])
957: (8)     random = Generator(MT19937(self.seed))
958: (8)     actual = random.dirichlet(alpha, size=(3, 2))
959: (8)     expected = np.array([
960: (12)         [[1., 0.],
961: (13)             [1., 0.]],
962: (12)             [[1., 0.],
963: (13)                 [1., 0.]],
964: (12)                 [[1., 0.],
965: (13)                     [1., 0.]]
966: (8)             ])
967: (8)             assert_array_almost_equal(actual, expected, decimal=15)
968: (4) @pytest.mark.slow
969: (4) def test_dirichlet_moderately_small_alpha(self):
970: (8)     alpha = np.array([0.02, 0.04, 0.03])
971: (8)     exact_mean = alpha / alpha.sum()
972: (8)     random = Generator(MT19937(self.seed))
973: (8)     sample = random.dirichlet(alpha, size=20000000)
974: (8)     sample_mean = sample.mean(axis=0)
975: (8)     assert_allclose(sample_mean, exact_mean, rtol=1e-3)
976: (4) @pytest.mark.parametrize(
977: (8)     'alpha',
978: (8)     [[5, 9, 0, 8],
979: (9)         [0.5, 0, 0, 0],
980: (9)             [1, 5, 0, 0, 1.5, 0, 0, 0],
981: (9)                 [0.01, 0.03, 0, 0.005],
982: (9)                     [1e-5, 0, 0, 0],
983: (9)                         [0.002, 0.015, 0, 0, 0.04, 0, 0, 0],
984: (9)                             [0.0],
985: (9)                                 [0, 0, 0]],
986: (4)             )
987: (4)             def test_dirichlet_multiple_zeros_in_alpha(self, alpha):
988: (8)                 alpha = np.array(alpha)
989: (8)                 y = random.dirichlet(alpha)
990: (8)                 assert_equal(y[alpha == 0], 0.0)
991: (4)             def test_exponential(self):
992: (8)                 random = Generator(MT19937(self.seed))
993: (8)                 actual = random.exponential(1.1234, size=(3, 2))
994: (8)                 desired = np.array([[0.098845481066258, 1.560752510746964],
995: (28)                     [0.075730916041636, 1.769098974710777],
996: (28)                         [1.488602544592235, 2.49684815275751]])
997: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
998: (4)             def test_exponential_0(self):
999: (8)                 assert_equal(random.exponential(scale=0), 0)
1000: (8)                 assert_raises(ValueError, random.exponential, scale=-0.)
1001: (4)             def test_f(self):
1002: (8)                 random = Generator(MT19937(self.seed))
1003: (8)                 actual = random.f(12, 77, size=(3, 2))
1004: (8)                 desired = np.array([[0.461720027077085, 1.100441958872451],
1005: (28)                     [1.100337455217484, 0.91421736740018],
1006: (28)                         [0.500811891303113, 0.826802454552058]])
1007: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
1008: (4)             def test_gamma(self):
1009: (8)                 random = Generator(MT19937(self.seed))
1010: (8)                 actual = random.gamma(5, 3, size=(3, 2))
1011: (8)                 desired = np.array([[ 5.03850858902096, 7.9228656732049],
1012: (28)                     [18.73983605132985, 19.57961681699238],
1013: (28)                         [18.17897755150825, 18.17653912505234]])
1014: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1015: (4)             def test_gamma_0(self):
1016: (8)                 assert_equal(random.gamma(shape=0, scale=0), 0)
1017: (8)                 assert_raises(ValueError, random.gamma, shape=-0., scale=-0.)
1018: (4)             def test_geometric(self):
1019: (8)                 random = Generator(MT19937(self.seed))
1020: (8)                 actual = random.geometric(.123456789, size=(3, 2))
1021: (8)                 desired = np.array([[1, 11],
1022: (28)                     [1, 12],

```

```

1023: (28) [11, 17]])
1024: (8) assert_array_equal(actual, desired)
1025: (4) def test_geometric_exceptions(self):
1026: (8)     assert_raises(ValueError, random.geometric, 1.1)
1027: (8)     assert_raises(ValueError, random.geometric, [1.1] * 10)
1028: (8)     assert_raises(ValueError, random.geometric, -0.1)
1029: (8)     assert_raises(ValueError, random.geometric, [-0.1] * 10)
1030: (8)     with np.errstate(invalid='ignore'):
1031: (12)         assert_raises(ValueError, random.geometric, np.nan)
1032: (12)         assert_raises(ValueError, random.geometric, [np.nan] * 10)
1033: (4) def test_gumbel(self):
1034: (8)     random = Generator(MT19937(self.seed))
1035: (8)     actual = random.gumbel(loc=.123456789, scale=2.0, size=(3, 2))
1036: (8)     desired = np.array([[ 4.688397515056245, -0.289514845417841],
1037: (28)             [ 4.981176042584683, -0.633224272589149],
1038: (28)             [-0.055915275687488, -0.333962478257953]])
1039: (8)     assert_array_almost_equal(actual, desired, decimal=15)
1040: (4) def test_gumbel_0(self):
1041: (8)     assert_equal(random.gumbel(scale=0), 0)
1042: (8)     assert_raises(ValueError, random.gumbel, scale=-0.)
1043: (4) def test_hypergeometric(self):
1044: (8)     random = Generator(MT19937(self.seed))
1045: (8)     actual = random.hypergeometric(10.1, 5.5, 14, size=(3, 2))
1046: (8)     desired = np.array([[ 9, 9],
1047: (28)         [ 9, 9],
1048: (28)         [10, 9]])
1049: (8)     assert_array_equal(actual, desired)
1050: (8)     actual = random.hypergeometric(5, 0, 3, size=4)
1051: (8)     desired = np.array([3, 3, 3, 3])
1052: (8)     assert_array_equal(actual, desired)
1053: (8)     actual = random.hypergeometric(15, 0, 12, size=4)
1054: (8)     desired = np.array([12, 12, 12, 12])
1055: (8)     assert_array_equal(actual, desired)
1056: (8)     actual = random.hypergeometric(0, 5, 3, size=4)
1057: (8)     desired = np.array([0, 0, 0, 0])
1058: (8)     assert_array_equal(actual, desired)
1059: (8)     actual = random.hypergeometric(0, 15, 12, size=4)
1060: (8)     desired = np.array([0, 0, 0, 0])
1061: (8)     assert_array_equal(actual, desired)
1062: (4) def test_laplace(self):
1063: (8)     random = Generator(MT19937(self.seed))
1064: (8)     actual = random.laplace(loc=.123456789, scale=2.0, size=(3, 2))
1065: (8)     desired = np.array([-3.156353949272393, 1.195863024830054],
1066: (28)         [-3.435458081645966, 1.656882398925444],
1067: (28)         [ 0.924824032467446, 1.251116432209336])
1068: (8)     assert_array_almost_equal(actual, desired, decimal=15)
1069: (4) def test_laplace_0(self):
1070: (8)     assert_equal(random.laplace(scale=0), 0)
1071: (8)     assert_raises(ValueError, random.laplace, scale=-0.)
1072: (4) def test_logistic(self):
1073: (8)     random = Generator(MT19937(self.seed))
1074: (8)     actual = random.logistic(loc=.123456789, scale=2.0, size=(3, 2))
1075: (8)     desired = np.array([-4.338584631510999, 1.890171436749954],
1076: (28)         [-4.64547787337966, 2.514545562919217],
1077: (28)         [ 1.495389489198666, 1.967827627577474])
1078: (8)     assert_array_almost_equal(actual, desired, decimal=15)
1079: (4) def test_lognormal(self):
1080: (8)     random = Generator(MT19937(self.seed))
1081: (8)     actual = random.lognormal(mean=.123456789, sigma=2.0, size=(3, 2))
1082: (8)     desired = np.array([[ 0.0268252166335, 13.9534486483053],
1083: (28)         [ 0.1204014788936, 2.2422077497792],
1084: (28)         [ 4.2484199496128, 12.0093343977523]])
1085: (8)     assert_array_almost_equal(actual, desired, decimal=13)
1086: (4) def test_lognormal_0(self):
1087: (8)     assert_equal(random.lognormal(sigma=0), 1)
1088: (8)     assert_raises(ValueError, random.lognormal, sigma=-0.)
1089: (4) def test_logseries(self):
1090: (8)     random = Generator(MT19937(self.seed))
1091: (8)     actual = random.logseries(p=.923456789, size=(3, 2))

```

```

1092: (8)             desired = np.array([[14, 17],
1093: (28)                 [3, 18],
1094: (28)                 [5, 1]])
1095: (8)             assert_array_equal(actual, desired)
1096: (4)             def test_logseries_zero(self):
1097: (8)                 random = Generator(MT19937(self.seed))
1098: (8)                 assert random.logseries(0) == 1
1099: (4)             @pytest.mark.parametrize("value", [np.nextafter(0., -1), 1., np.nan, 5.])
1100: (4)             def test_logseries_exceptions(self, value):
1101: (8)                 random = Generator(MT19937(self.seed))
1102: (8)                 with np.errstate(invalid="ignore"):
1103: (12)                     with pytest.raises(ValueError):
1104: (16)                         random.logseries(value)
1105: (12)                     with pytest.raises(ValueError):
1106: (16)                         random.logseries(np.array([value] * 10))
1107: (12)                     with pytest.raises(ValueError):
1108: (16)                         random.logseries(np.array([value] * 10)[::2])
1109: (4)             def test_multinomial(self):
1110: (8)                 random = Generator(MT19937(self.seed))
1111: (8)                 actual = random.multinomial(20, [1 / 6.] * 6, size=(3, 2))
1112: (8)                 desired = np.array([[[1, 5, 1, 6, 4, 3],
1113: (29)                     [4, 2, 6, 2, 4, 2]],
1114: (28)                     [[5, 3, 2, 6, 3, 1],
1115: (29)                     [4, 4, 0, 2, 3, 7]],
1116: (28)                     [[6, 3, 1, 5, 3, 2],
1117: (29)                     [5, 5, 3, 1, 2, 4]]])
1118: (8)             assert_array_equal(actual, desired)
1119: (4)             @pytest.mark.skipif(IS_WASM, reason="fp errors don't work in wasm")
1120: (4)             @pytest.mark.parametrize("method", ["svd", "eigh", "cholesky"])
1121: (4)             def test_multivariate_normal(self, method):
1122: (8)                 random = Generator(MT19937(self.seed))
1123: (8)                 mean = (.123456789, 10)
1124: (8)                 cov = [[1, 0], [0, 1]]
1125: (8)                 size = (3, 2)
1126: (8)                 actual = random.multivariate_normal(mean, cov, size, method=method)
1127: (8)                 desired = np.array([[-1.747478062846581, 11.25613495182354 ],
1128: (29)                     [-0.9967333370066214, 10.342002097029821 ],
1129: (28)                     [0.7850019631242964, 11.181113712443013 ],
1130: (29)                     [0.8901349653255224, 8.873825399642492 ],
1131: (28)                     [0.713026010743003, 9.551628690083056 ],
1132: (29)                     [0.7127098726541128, 11.991709234143173 ]])
1133: (8)             assert_array_almost_equal(actual, desired, decimal=15)
1134: (8)             actual = random.multivariate_normal(mean, cov, method=method)
1135: (8)             desired = np.array([0.233278563284287, 9.424140804347195])
1136: (8)             assert_array_almost_equal(actual, desired, decimal=15)
1137: (8)             mean = [0, 0]
1138: (8)             cov = [[1, 2], [1, 2]]
1139: (8)             assert_raises(ValueError, random.multivariate_normal, mean, cov,
1140: (22)                             check_valid='raise')
1141: (8)             cov = [[1, 2], [2, 1]]
1142: (8)             assert_warnings(RuntimeWarning, random.multivariate_normal, mean, cov)
1143: (8)             assert_warnings(RuntimeWarning, random.multivariate_normal, mean, cov,
1144: (21)                             method='eigh')
1145: (8)             assert_raises(LinAlgError, random.multivariate_normal, mean, cov,
1146: (22)                             method='cholesky')
1147: (8)             assert_no_warnings(random.multivariate_normal, mean, cov,
1148: (27)                             check_valid='ignore')
1149: (8)             assert_raises(ValueError, random.multivariate_normal, mean, cov,
1150: (22)                             check_valid='raise')
1151: (8)             assert_raises(ValueError, random.multivariate_normal, mean, cov,
1152: (22)                             check_valid='raise', method='eigh')
1153: (8)             cov = [[1, 1], [1, 1]]
1154: (8)             if method in ('svd', 'eigh'):
1155: (12)                 samples = random.multivariate_normal(mean, cov, size=(3, 2),
1156: (49)                                 method=method)
1157: (12)                 assert_array_almost_equal(samples[..., 0], samples[..., 1],
1158: (38)                                 decimal=6)
1159: (8)             else:
1160: (12)                 assert_raises(LinAlgError, random.multivariate_normal, mean, cov,

```

```

1161: (26)
1162: (8)
1163: (8)
1164: (12)
1165: (12)
1166: (12)
1167: (8)
1168: (8)
1169: (8)
1170: (22)
1171: (8)
1172: (22)
1173: (8)
1174: (22)
1175: (8)
1176: (22)
1177: (4)
1178: (4)
1179: (8)
1180: (8)
1181: (12)
1182: (4)
1183: (4)
1184: (8)
1185: (8)
1186: (8)
1187: (8)
1188: (8)
1189: (8)
1190: (8)
1191: (8)
1192: (8)
1193: (4)
1194: (8)
1195: (8)
1196: (8)
1197: (28)
1198: (28)
1199: (8)
1200: (4)
1201: (8)
1202: (12)
1203: (12)
1204: (26)
1205: (4)
1206: (8)
1207: (12)
1208: (4)
1209: (8)
1210: (12)
1211: (12)
1212: [0.1])
1213: (4)
1214: (8)
1215: (8)
1216: (28)
1217: (28)
1218: (8)
1219: (8)
1220: (8)
1221: (28)
1222: (28)
1223: (8)
1224: (8)
1225: (8)
1226: (8)
1227: (28)
1228: (28)

        method='cholesky')
    cov = np.array([[1, 0.1], [0.1, 1]], dtype=np.float32)
    with suppress_warnings() as sup:
        random.multivariate_normal(mean, cov, method=method)
        w = sup.record(RuntimeWarning)
        assert len(w) == 0
    mu = np.zeros(2)
    cov = np.eye(2)
    assert_raises(ValueError, random.multivariate_normal, mean, cov,
                  check_valid='other')
    assert_raises(ValueError, random.multivariate_normal,
                  np.zeros((2, 1, 1)), cov)
    assert_raises(ValueError, random.multivariate_normal,
                  mu, np.empty((3, 2)))
    assert_raises(ValueError, random.multivariate_normal,
                  mu, np.eye(3))

@pytest.mark.parametrize('mean, cov', [[(0), [[1+1j]]], ([0j], [[1]])])
def test_multivariate_normal_disallow_complex(self, mean, cov):
    random = Generator(MT19937(self.seed))
    with pytest.raises(TypeError, match="must not be complex"):
        random.multivariate_normal(mean, cov)

@pytest.mark.parametrize("method", ["svd", "eigh", "cholesky"])
def test_multivariate_normal_basic_stats(self, method):
    random = Generator(MT19937(self.seed))
    n_s = 1000
    mean = np.array([1, 2])
    cov = np.array([[2, 1], [1, 2]])
    s = random.multivariate_normal(mean, cov, size=(n_s,), method=method)
    s_center = s - mean
    cov_emp = (s_center.T @ s_center) / (n_s - 1)
    assert np.all(np.abs(s_center.mean(-2)) < 0.1)
    assert np.all(np.abs(cov_emp - cov) < 0.2)

def test_negative_binomial(self):
    random = Generator(MT19937(self.seed))
    actual = random.negative_binomial(n=100, p=.12345, size=(3, 2))
    desired = np.array([[543, 727],
                      [775, 760],
                      [600, 674]])
    assert_array_equal(actual, desired)

def test_negative_binomial_exceptions(self):
    with np.errstate(invalid='ignore'):
        assert_raises(ValueError, random.negative_binomial, 100, np.nan)
        assert_raises(ValueError, random.negative_binomial, 100,
                      [np.nan] * 10)

def test_negative_binomial_p0_exception(self):
    with assert_raises(ValueError):
        x = random.negative_binomial(1, 0)

def test_negative_binomial_invalid_p_n_combination(self):
    with np.errstate(invalid='ignore'):
        assert_raises(ValueError, random.negative_binomial, 2**62, 0.1)
        assert_raises(ValueError, random.negative_binomial, [2**62], [0.1])

def test_noncentral_chisquare(self):
    random = Generator(MT19937(self.seed))
    actual = random.noncentral_chisquare(df=5, nonc=5, size=(3, 2))
    desired = np.array([[ 1.70561552362133, 15.97378184942111],
                      [13.71483425173724, 20.17859633310629],
                      [11.3615477156643 , 3.67891108738029]])
    assert_array_almost_equal(actual, desired, decimal=14)
    actual = random.noncentral_chisquare(df=.5, nonc=.2, size=(3, 2))
    desired = np.array([[9.41427665607629e-04, 1.70473157518850e-04],
                      [1.14554372041263e+00, 1.38187755933435e-03],
                      [1.90659181905387e+00, 1.21772577941822e+00]])
    assert_array_almost_equal(actual, desired, decimal=14)
    random = Generator(MT19937(self.seed))
    actual = random.noncentral_chisquare(df=5, nonc=0, size=(3, 2))
    desired = np.array([[0.82947954590419, 1.80139670767078],
                      [6.58720057417794, 7.00491463609814],
                      [6.31101879073157, 6.30982307753005]])

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1229: (8) assert_array_almost_equal(actual, desired, decimal=14)
1230: (4) def test_noncentral_f(self):
1231: (8)     random = Generator(MT19937(self.seed))
1232: (8)     actual = random.noncentral_f(dfnum=5, dfden=2, nonc=1,
1233: (37)                     size=(3, 2))
1234: (8)     desired = np.array([[0.060310671139, 0.23866058175939],
1235: (28)                 [0.86860246709073, 0.2668510459738],
1236: (28)                 [0.23375780078364, 1.88922102885943]])
1237: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1238: (4) def test_noncentral_f_nan(self):
1239: (8)     random = Generator(MT19937(self.seed))
1240: (8)     actual = random.noncentral_f(dfnum=5, dfden=2, nonc=np.nan)
1241: (8)     assert np.isnan(actual)
1242: (4) def test_normal(self):
1243: (8)     random = Generator(MT19937(self.seed))
1244: (8)     actual = random.normal(loc=.123456789, scale=2.0, size=(3, 2))
1245: (8)     desired = np.array([-3.618412914693162, 2.635726692647081],
1246: (28)           [-2.116923463013243, 0.807460983059643],
1247: (28)           [1.446547137248593, 2.485684213886024])
1248: (8)     assert_array_almost_equal(actual, desired, decimal=15)
1249: (4) def test_normal_0(self):
1250: (8)     assert_equal(random.normal(scale=0), 0)
1251: (8)     assert_raises(ValueError, random.normal, scale=-0.)
1252: (4) def test_pareto(self):
1253: (8)     random = Generator(MT19937(self.seed))
1254: (8)     actual = random.pareto(a=.123456789, size=(3, 2))
1255: (8)     desired = np.array([1.0394926776069018e+00, 7.7142534343505773e+04],
1256: (28)           [7.2640150889064703e-01, 3.4650454783825594e+05],
1257: (28)           [4.5852344481994740e+04, 6.5851383009539105e+07])
1258: (8)     np.testing.assert_array_almost_equal(actual, desired, nulp=30)
1259: (4) def test_poisson(self):
1260: (8)     random = Generator(MT19937(self.seed))
1261: (8)     actual = random.poisson(lam=.123456789, size=(3, 2))
1262: (8)     desired = np.array([[0, 0],
1263: (28)           [0, 0],
1264: (28)           [0, 0]])
1265: (8)     assert_array_equal(actual, desired)
1266: (4) def test_poisson_exceptions(self):
1267: (8)     lambig = np.iinfo('int64').max
1268: (8)     lamneg = -1
1269: (8)     assert_raises(ValueError, random.poisson, lamneg)
1270: (8)     assert_raises(ValueError, random.poisson, [lamneg] * 10)
1271: (8)     assert_raises(ValueError, random.poisson, lambig)
1272: (8)     assert_raises(ValueError, random.poisson, [lambig] * 10)
1273: (8)     with np.errstate(invalid='ignore'):
1274: (12)         assert_raises(ValueError, random.poisson, np.nan)
1275: (12)         assert_raises(ValueError, random.poisson, [np.nan] * 10)
1276: (4) def test_power(self):
1277: (8)     random = Generator(MT19937(self.seed))
1278: (8)     actual = random.power(a=.123456789, size=(3, 2))
1279: (8)     desired = np.array([[1.977857368842754e-09, 9.806792196620341e-02],
1280: (28)           [2.482442984543471e-10, 1.527108843266079e-01],
1281: (28)           [8.188283434244285e-02, 3.950547209346948e-01]])
1282: (8)     assert_array_almost_equal(actual, desired, decimal=15)
1283: (4) def test_rayleigh(self):
1284: (8)     random = Generator(MT19937(self.seed))
1285: (8)     actual = random.rayleigh(scale=10, size=(3, 2))
1286: (8)     desired = np.array([[4.19494429102666, 16.66920198906598],
1287: (28)           [3.67184544902662, 17.74695521962917],
1288: (28)           [16.27935397855501, 21.08355560691792]])
1289: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1290: (4) def test_rayleigh_0(self):
1291: (8)     assert_equal(random.rayleigh(scale=0), 0)
1292: (8)     assert_raises(ValueError, random.rayleigh, scale=-0.)
1293: (4) def test_standard_cauchy(self):
1294: (8)     random = Generator(MT19937(self.seed))
1295: (8)     actual = random.standard_cauchy(size=(3, 2))
1296: (8)     desired = np.array([-1.48943778266206, -3.275389641569784],
1297: (28)           [0.560102864910406, -0.680780916282552]),

```

```

1298: (28)                                     [-1.314912905226277,  0.295852965660225]])]
1299: (8)          assert_array_almost_equal(actual, desired, decimal=15)
1300: (4)          def test_standard_exponential(self):
1301: (8)              random = Generator(MT19937(self.seed))
1302: (8)              actual = random.standard_exponential(size=(3, 2), method='inv')
1303: (8)              desired = np.array([[0.102031839440643, 1.229350298474972],
1304: (28)                                [0.088137284693098, 1.459859985522667],
1305: (28)                                [1.093830802293668, 1.256977002164613]])
1306: (8)          assert_array_almost_equal(actual, desired, decimal=15)
1307: (4)          def test_standard_exponential_type_error(self):
1308: (8)              assert_raises(TypeError, random.standard_exponential, dtype=np.int32)
1309: (4)          def test_standard_gamma(self):
1310: (8)              random = Generator(MT19937(self.seed))
1311: (8)              actual = random.standard_gamma(shape=3, size=(3, 2))
1312: (8)              desired = np.array([[0.62970724056362, 1.22379851271008],
1313: (28)                                [3.899412530884, 4.12479964250139],
1314: (28)                                [3.74994102464584, 3.74929307690815]])
1315: (8)          assert_array_almost_equal(actual, desired, decimal=14)
1316: (4)          def test_standard_gamma_scalar_float(self):
1317: (8)              random = Generator(MT19937(self.seed))
1318: (8)              actual = random.standard_gamma(3, dtype=np.float32)
1319: (8)              desired = 2.9242148399353027
1320: (8)          assert_array_almost_equal(actual, desired, decimal=6)
1321: (4)          def test_standard_gamma_float(self):
1322: (8)              random = Generator(MT19937(self.seed))
1323: (8)              actual = random.standard_gamma(shape=3, size=(3, 2))
1324: (8)              desired = np.array([[0.62971, 1.2238],
1325: (28)                                [3.89941, 4.1248],
1326: (28)                                [3.74994, 3.74929]])
1327: (8)          assert_array_almost_equal(actual, desired, decimal=5)
1328: (4)          def test_standard_gamma_float_out(self):
1329: (8)              actual = np.zeros((3, 2), dtype=np.float32)
1330: (8)              random = Generator(MT19937(self.seed))
1331: (8)              random.standard_gamma(10.0, out=actual, dtype=np.float32)
1332: (8)              desired = np.array([[10.14987, 7.87012],
1333: (29)                                [9.46284, 12.56832],
1334: (29)                                [13.82495, 7.81533]], dtype=np.float32)
1335: (8)          assert_array_almost_equal(actual, desired, decimal=5)
1336: (8)          random = Generator(MT19937(self.seed))
1337: (8)          random.standard_gamma(10.0, out=actual, size=(3, 2), dtype=np.float32)
1338: (8)          assert_array_almost_equal(actual, desired, decimal=5)
1339: (4)          def test_standard_gamma_unknown_type(self):
1340: (8)              assert_raises(TypeError, random.standard_gamma, 1.,
1341: (22)                            dtype='int32')
1342: (4)          def test_out_size_mismatch(self):
1343: (8)              out = np.zeros(10)
1344: (8)              assert_raises(ValueError, random.standard_gamma, 10.0, size=20,
1345: (22)                                out=out)
1346: (8)              assert_raises(ValueError, random.standard_gamma, 10.0, size=(10, 1),
1347: (22)                                out=out)
1348: (4)          def test_standard_gamma_0(self):
1349: (8)              assert_equal(random.standard_gamma(shape=0), 0)
1350: (8)              assert_raises(ValueError, random.standard_gamma, shape=-0.)
1351: (4)          def test_standard_normal(self):
1352: (8)              random = Generator(MT19937(self.seed))
1353: (8)              actual = random.standard_normal(size=(3, 2))
1354: (8)              desired = np.array([-1.870934851846581, 1.25613495182354],
1355: (28)                                [-1.120190126006621, 0.342002097029821],
1356: (28)                                [0.661545174124296, 1.181113712443012])
1357: (8)          assert_array_almost_equal(actual, desired, decimal=15)
1358: (4)          def test_standard_normal_unsupported_type(self):
1359: (8)              assert_raises(TypeError, random.standard_normal, dtype=np.int32)
1360: (4)          def test_standard_t(self):
1361: (8)              random = Generator(MT19937(self.seed))
1362: (8)              actual = random.standard_t(df=10, size=(3, 2))
1363: (8)              desired = np.array([-1.484666193042647, 0.30597891831161],
1364: (28)                                [1.056684299648085, -0.407312602088507],
1365: (28)                                [0.130704414281157, -2.038053410490321])
1366: (8)          assert_array_almost_equal(actual, desired, decimal=15)

```

```

1367: (4) def test_triangular(self):
1368: (8)     random = Generator(MT19937(self.seed))
1369: (8)     actual = random.triangular(left=5.12, mode=10.23, right=20.34,
1370: (35)                     size=(3, 2))
1371: (8)     desired = np.array([[ 7.86664070590917, 13.6313848513185 ],
1372: (28)                 [ 7.68152445215983, 14.36169131136546],
1373: (28)                 [13.16105603911429, 13.72341621856971]])
1374: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1375: (4) def test_uniform(self):
1376: (8)     random = Generator(MT19937(self.seed))
1377: (8)     actual = random.uniform(low=1.23, high=10.54, size=(3, 2))
1378: (8)     desired = np.array([[2.13306255040998 , 7.816987531021207],
1379: (28)                 [2.015436610109887, 8.377577533009589],
1380: (28)                 [7.421792588856135, 7.891185744455209]])
1381: (8)     assert_array_almost_equal(actual, desired, decimal=15)
1382: (4) def test_uniform_range_bounds(self):
1383: (8)     fmin = np.finfo('float').min
1384: (8)     fmax = np.finfo('float').max
1385: (8)     func = random.uniform
1386: (8)     assert_raises(OverflowError, func, -np.inf, 0)
1387: (8)     assert_raises(OverflowError, func, 0, np.inf)
1388: (8)     assert_raises(OverflowError, func, fmin, fmax)
1389: (8)     assert_raises(OverflowError, func, [-np.inf], [0])
1390: (8)     assert_raises(OverflowError, func, [0], [np.inf])
1391: (8)     random.uniform(low=np.nextafter(fmin, 1), high=fmax / 1e17)
1392: (4) def test_uniform_zero_range(self):
1393: (8)     func = random.uniform
1394: (8)     result = func(1.5, 1.5)
1395: (8)     assert_allclose(result, 1.5)
1396: (8)     result = func([0.0, np.pi], [0.0, np.pi])
1397: (8)     assert_allclose(result, [0.0, np.pi])
1398: (8)     result = func([[2145.12], [2145.12]], [2145.12, 2145.12])
1399: (8)     assert_allclose(result, 2145.12 + np.zeros((2, 2)))
1400: (4) def test_uniform_neg_range(self):
1401: (8)     func = random.uniform
1402: (8)     assert_raises(ValueError, func, 2, 1)
1403: (8)     assert_raises(ValueError, func, [1, 2], [1, 1])
1404: (8)     assert_raises(ValueError, func, [[0, 1],[2, 3]], 2)
1405: (4) def test_scalar_exception_propagation(self):
1406: (8)     class ThrowingFloat(np.ndarray):
1407: (12)         def __float__(self):
1408: (16)             raise TypeError
1409: (8)     throwing_float = np.array(1.0).view(ThrowingFloat)
1410: (8)     assert_raises(TypeError, random.uniform, throwing_float,
1411: (22)             throwing_float)
1412: (8)     class ThrowingInteger(np.ndarray):
1413: (12)         def __int__(self):
1414: (16)             raise TypeError
1415: (8)     throwing_int = np.array(1).view(ThrowingInteger)
1416: (8)     assert_raises(TypeError, random.hypergeometric, throwing_int, 1, 1)
1417: (4) def test_vonmises(self):
1418: (8)     random = Generator(MT19937(self.seed))
1419: (8)     actual = random.vonmises(mu=1.23, kappa=1.54, size=(3, 2))
1420: (8)     desired = np.array([[ 1.107972248690106, 2.841536476232361],
1421: (28)                 [ 1.832602376042457, 1.945511926976032],
1422: (28)                 [-0.260147475776542, 2.058047492231698]])
1423: (8)     assert_array_almost_equal(actual, desired, decimal=15)
1424: (4) def test_vonmises_small(self):
1425: (8)     random = Generator(MT19937(self.seed))
1426: (8)     r = random.vonmises(mu=0., kappa=1.1e-8, size=10**6)
1427: (8)     assert_(np.isfinite(r).all())
1428: (4) def test_vonmises_nan(self):
1429: (8)     random = Generator(MT19937(self.seed))
1430: (8)     r = random.vonmises(mu=0., kappa=np.nan)
1431: (8)     assert_(np.isnan(r))
1432: (4) @pytest.mark.parametrize("kappa", [1e4, 1e15])
1433: (4) def test_vonmises_large_kappa(self, kappa):
1434: (8)     random = Generator(MT19937(self.seed))
1435: (8)     rs = RandomState(random.bit_generator)

```

```

1436: (8)             state = random.bit_generator.state
1437: (8)             random_state_vals = rs.vonmises(0, kappa, size=10)
1438: (8)             random.bit_generator.state = state
1439: (8)             gen_vals = random.vonmises(0, kappa, size=10)
1440: (8)             if kappa < 1e6:
1441: (12)                 assert_allclose(random_state_vals, gen_vals)
1442: (8)             else:
1443: (12)                 assert np.all(random_state_vals != gen_vals)
1444: (4)             @pytest.mark.parametrize("mu", [-7., -np.pi, -3.1, np.pi, 3.2])
1445: (4)             @pytest.mark.parametrize("kappa", [1e-9, 1e-6, 1, 1e3, 1e15])
1446: (4)             def test_vonmises_large_kappa_range(self, mu, kappa):
1447: (8)                 random = Generator(MT19937(self.seed))
1448: (8)                 r = random.vonmises(mu, kappa, 50)
1449: (8)                 assert_(np.all(r > -np.pi) and np.all(r <= np.pi))
1450: (4)             def test_wald(self):
1451: (8)                 random = Generator(MT19937(self.seed))
1452: (8)                 actual = random.wald(mean=1.23, scale=1.54, size=(3, 2))
1453: (8)                 desired = np.array([[0.26871721804551, 3.2233942732115],
1454: (28)                               [2.20328374987066, 2.40958405189353],
1455: (28)                               [2.07093587449261, 0.73073890064369]])
1456: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1457: (4)             def test_weibull(self):
1458: (8)                 random = Generator(MT19937(self.seed))
1459: (8)                 actual = random.weibull(a=1.23, size=(3, 2))
1460: (8)                 desired = np.array([[0.138613914769468, 1.306463419753191],
1461: (28)                               [0.111623365934763, 1.446570494646721],
1462: (28)                               [1.257145775276011, 1.914247725027957]])
1463: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
1464: (4)             def test_weibull_0(self):
1465: (8)                 random = Generator(MT19937(self.seed))
1466: (8)                 assert_equal(random.weibull(a=0, size=12), np.zeros(12))
1467: (8)                 assert_raises(ValueError, random.weibull, a=-0.)
1468: (4)             def test_zipf(self):
1469: (8)                 random = Generator(MT19937(self.seed))
1470: (8)                 actual = random.zipf(a=1.23, size=(3, 2))
1471: (8)                 desired = np.array([[ 1,   1],
1472: (28)                               [ 10, 867],
1473: (28)                               [354,   2]])
1474: (8)                 assert_array_equal(actual, desired)
1475: (0)             class TestBroadcast:
1476: (4)                 def setup_method(self):
1477: (8)                     self.seed = 123456789
1478: (4)                 def test_uniform(self):
1479: (8)                     random = Generator(MT19937(self.seed))
1480: (8)                     low = [0]
1481: (8)                     high = [1]
1482: (8)                     uniform = random.uniform
1483: (8)                     desired = np.array([0.16693771389729, 0.19635129550675,
0.75563050964095])
1484: (8)                     random = Generator(MT19937(self.seed))
1485: (8)                     actual = random.uniform(low * 3, high)
1486: (8)                     assert_array_almost_equal(actual, desired, decimal=14)
1487: (8)                     random = Generator(MT19937(self.seed))
1488: (8)                     actual = random.uniform(low, high * 3)
1489: (8)                     assert_array_almost_equal(actual, desired, decimal=14)
1490: (4)             def test_normal(self):
1491: (8)                 loc = [0]
1492: (8)                 scale = [1]
1493: (8)                 bad_scale = [-1]
1494: (8)                 random = Generator(MT19937(self.seed))
1495: (8)                 desired = np.array([-0.38736406738527, 0.79594375042255,
0.0197076236097])
1496: (8)                 random = Generator(MT19937(self.seed))
1497: (8)                 actual = random.normal(loc * 3, scale)
1498: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1499: (8)                 assert_raises(ValueError, random.normal, loc * 3, bad_scale)
1500: (8)                 random = Generator(MT19937(self.seed))
1501: (8)                 normal = random.normal
1502: (8)                 actual = normal(loc, scale * 3)

```

```

1503: (8) assert_array_almost_equal(actual, desired, decimal=14)
1504: (8) assert_raises(ValueError, normal, loc, bad_scale * 3)
1505: (4) def test_beta(self):
1506: (8)     a = [1]
1507: (8)     b = [2]
1508: (8)     bad_a = [-1]
1509: (8)     bad_b = [-2]
1510: (8)     desired = np.array([0.18719338682602, 0.73234824491364,
0.17928615186455])
1511: (8)     random = Generator(MT19937(self.seed))
1512: (8)     beta = random.beta
1513: (8)     actual = beta(a * 3, b)
1514: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1515: (8)     assert_raises(ValueError, beta, bad_a * 3, b)
1516: (8)     assert_raises(ValueError, beta, a * 3, bad_b)
1517: (8)     random = Generator(MT19937(self.seed))
1518: (8)     actual = random.beta(a, b * 3)
1519: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1520: (4) def test_exponential(self):
1521: (8)     scale = [1]
1522: (8)     bad_scale = [-1]
1523: (8)     desired = np.array([0.67245993212806, 0.21380495318094,
0.7177848928629])
1524: (8)     random = Generator(MT19937(self.seed))
1525: (8)     actual = random.exponential(scale * 3)
1526: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1527: (8)     assert_raises(ValueError, random.exponential, bad_scale * 3)
1528: (4) def test_standard_gamma(self):
1529: (8)     shape = [1]
1530: (8)     bad_shape = [-1]
1531: (8)     desired = np.array([0.67245993212806, 0.21380495318094,
0.7177848928629])
1532: (8)     random = Generator(MT19937(self.seed))
1533: (8)     std_gamma = random.standard_gamma
1534: (8)     actual = std_gamma(shape * 3)
1535: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1536: (8)     assert_raises(ValueError, std_gamma, bad_shape * 3)
1537: (4) def test_gamma(self):
1538: (8)     shape = [1]
1539: (8)     scale = [2]
1540: (8)     bad_shape = [-1]
1541: (8)     bad_scale = [-2]
1542: (8)     desired = np.array([1.34491986425611, 0.42760990636187,
1.4355697857258])
1543: (8)     random = Generator(MT19937(self.seed))
1544: (8)     gamma = random.gamma
1545: (8)     actual = gamma(shape * 3, scale)
1546: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1547: (8)     assert_raises(ValueError, gamma, bad_shape * 3, scale)
1548: (8)     assert_raises(ValueError, gamma, shape * 3, bad_scale)
1549: (8)     random = Generator(MT19937(self.seed))
1550: (8)     gamma = random.gamma
1551: (8)     actual = gamma(shape, scale * 3)
1552: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1553: (8)     assert_raises(ValueError, gamma, bad_shape, scale * 3)
1554: (8)     assert_raises(ValueError, gamma, shape, bad_scale * 3)
1555: (4) def test_f(self):
1556: (8)     dfnum = [1]
1557: (8)     dfden = [2]
1558: (8)     bad_dfnum = [-1]
1559: (8)     bad_dfden = [-2]
1560: (8)     desired = np.array([0.07765056244107, 7.72951397913186,
0.05786093891763])
1561: (8)     random = Generator(MT19937(self.seed))
1562: (8)     f = random.f
1563: (8)     actual = f(dfnum * 3, dfden)
1564: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1565: (8)     assert_raises(ValueError, f, bad_dfnum * 3, dfden)
1566: (8)     assert_raises(ValueError, f, dfnum * 3, bad_dfden)

```

```

1567: (8)             random = Generator(MT19937(self.seed))
1568: (8)             f = random.f
1569: (8)             actual = f(dfnum, dfden * 3)
1570: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1571: (8)             assert_raises(ValueError, f, bad_dfnum, dfden * 3)
1572: (8)             assert_raises(ValueError, f, dfnum, bad_dfden * 3)
1573: (4)             def test_noncentral_f(self):
1574: (8)                 dfnum = [2]
1575: (8)                 dfden = [3]
1576: (8)                 nonc = [4]
1577: (8)                 bad_dfnum = [0]
1578: (8)                 bad_dfden = [-1]
1579: (8)                 bad_nonc = [-2]
1580: (8)                 desired = np.array([2.02434240411421, 12.91838601070124,
1.24395160354629])
1581: (8)             random = Generator(MT19937(self.seed))
1582: (8)             nonc_f = random.noncentral_f
1583: (8)             actual = nonc_f(dfnum * 3, dfden, nonc)
1584: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1585: (8)             assert np.all(np.isnan(nonc_f(dfnum, dfden, [np.nan] * 3)))
1586: (8)             assert_raises(ValueError, nonc_f, bad_dfnum * 3, dfden, nonc)
1587: (8)             assert_raises(ValueError, nonc_f, dfnum * 3, bad_dfden, nonc)
1588: (8)             assert_raises(ValueError, nonc_f, dfnum * 3, dfden, bad_nonc)
1589: (8)             random = Generator(MT19937(self.seed))
1590: (8)             nonc_f = random.noncentral_f
1591: (8)             actual = nonc_f(dfnum, dfden * 3, nonc)
1592: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1593: (8)             assert_raises(ValueError, nonc_f, bad_dfnum, dfden * 3, nonc)
1594: (8)             assert_raises(ValueError, nonc_f, dfnum, bad_dfden * 3, nonc)
1595: (8)             assert_raises(ValueError, nonc_f, dfnum, dfden * 3, bad_nonc)
1596: (8)             random = Generator(MT19937(self.seed))
1597: (8)             nonc_f = random.noncentral_f
1598: (8)             actual = nonc_f(dfnum, dfden, nonc * 3)
1599: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1600: (8)             assert_raises(ValueError, nonc_f, bad_dfnum, dfden, nonc * 3)
1601: (8)             assert_raises(ValueError, nonc_f, dfnum, bad_dfden, nonc * 3)
1602: (8)             assert_raises(ValueError, nonc_f, dfnum, dfden, bad_nonc * 3)
1603: (4)             def test_noncentral_f_small_df(self):
1604: (8)                 random = Generator(MT19937(self.seed))
1605: (8)                 desired = np.array([0.04714867120827, 0.1239390327694])
1606: (8)                 actual = random.noncentral_f(0.9, 0.9, 2, size=2)
1607: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1608: (4)             def test_chisquare(self):
1609: (8)                 df = [1]
1610: (8)                 bad_df = [-1]
1611: (8)                 desired = np.array([0.05573640064251, 1.47220224353539,
2.9469379318589])
1612: (8)             random = Generator(MT19937(self.seed))
1613: (8)             actual = random.chisquare(df * 3)
1614: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1615: (8)             assert_raises(ValueError, random.chisquare, bad_df * 3)
1616: (4)             def test_noncentral_chisquare(self):
1617: (8)                 df = [1]
1618: (8)                 nonc = [2]
1619: (8)                 bad_df = [-1]
1620: (8)                 bad_nonc = [-2]
1621: (8)                 desired = np.array([0.07710766249436, 5.27829115110304,
0.630732147399])
1622: (8)             random = Generator(MT19937(self.seed))
1623: (8)             nonc_chi = random.noncentral_chisquare
1624: (8)             actual = nonc_chi(df * 3, nonc)
1625: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1626: (8)             assert_raises(ValueError, nonc_chi, bad_df * 3, nonc)
1627: (8)             assert_raises(ValueError, nonc_chi, df * 3, bad_nonc)
1628: (8)             random = Generator(MT19937(self.seed))
1629: (8)             nonc_chi = random.noncentral_chisquare
1630: (8)             actual = nonc_chi(df, nonc * 3)
1631: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1632: (8)             assert_raises(ValueError, nonc_chi, bad_df, nonc * 3)

```

```

1633: (8) assert_raises(ValueError, nonc_chi, df, bad_nonc * 3)
1634: (4) def test_standard_t(self):
1635: (8)     df = [1]
1636: (8)     bad_df = [-1]
1637: (8)     desired = np.array([-1.39498829447098, -1.23058658835223,
0.17207021065983])
1638: (8) random = Generator(MT19937(self.seed))
1639: (8) actual = random.standard_t(df * 3)
1640: (8) assert_array_almost_equal(actual, desired, decimal=14)
1641: (8) assert_raises(ValueError, random.standard_t, bad_df * 3)
1642: (4) def test_vonmises(self):
1643: (8)     mu = [2]
1644: (8)     kappa = [1]
1645: (8)     bad_kappa = [-1]
1646: (8)     desired = np.array([2.25935584988528, 2.23326261461399,
-2.84152146503326])
1647: (8) random = Generator(MT19937(self.seed))
1648: (8) actual = random.vonmises(mu * 3, kappa)
1649: (8) assert_array_almost_equal(actual, desired, decimal=14)
1650: (8) assert_raises(ValueError, random.vonmises, mu * 3, bad_kappa)
1651: (8) random = Generator(MT19937(self.seed))
1652: (8) actual = random.vonmises(mu, kappa * 3)
1653: (8) assert_array_almost_equal(actual, desired, decimal=14)
1654: (8) assert_raises(ValueError, random.vonmises, mu, bad_kappa * 3)
1655: (4) def test_pareto(self):
1656: (8)     a = [1]
1657: (8)     bad_a = [-1]
1658: (8)     desired = np.array([0.95905052946317, 0.2383810889437 ,
1.04988745750013])
1659: (8) random = Generator(MT19937(self.seed))
1660: (8) actual = random.pareto(a * 3)
1661: (8) assert_array_almost_equal(actual, desired, decimal=14)
1662: (8) assert_raises(ValueError, random.pareto, bad_a * 3)
1663: (4) def test_weibull(self):
1664: (8)     a = [1]
1665: (8)     bad_a = [-1]
1666: (8)     desired = np.array([0.67245993212806, 0.21380495318094,
0.7177848928629])
1667: (8) random = Generator(MT19937(self.seed))
1668: (8) actual = random.weibull(a * 3)
1669: (8) assert_array_almost_equal(actual, desired, decimal=14)
1670: (8) assert_raises(ValueError, random.weibull, bad_a * 3)
1671: (4) def test_power(self):
1672: (8)     a = [1]
1673: (8)     bad_a = [-1]
1674: (8)     desired = np.array([0.48954864361052, 0.19249412888486,
0.51216834058807])
1675: (8) random = Generator(MT19937(self.seed))
1676: (8) actual = random.power(a * 3)
1677: (8) assert_array_almost_equal(actual, desired, decimal=14)
1678: (8) assert_raises(ValueError, random.power, bad_a * 3)
1679: (4) def test_laplace(self):
1680: (8)     loc = [0]
1681: (8)     scale = [1]
1682: (8)     bad_scale = [-1]
1683: (8)     desired = np.array([-1.09698732625119, -0.93470271947368,
0.71592671378202])
1684: (8) random = Generator(MT19937(self.seed))
1685: (8) laplace = random.laplace
1686: (8) actual = laplace(loc * 3, scale)
1687: (8) assert_array_almost_equal(actual, desired, decimal=14)
1688: (8) assert_raises(ValueError, laplace, loc * 3, bad_scale)
1689: (8) random = Generator(MT19937(self.seed))
1690: (8) laplace = random.laplace
1691: (8) actual = laplace(loc, scale * 3)
1692: (8) assert_array_almost_equal(actual, desired, decimal=14)
1693: (8) assert_raises(ValueError, laplace, loc, bad_scale * 3)
1694: (4) def test_gumbel(self):
1695: (8)     loc = [0]

```

```

1696: (8) scale = [1]
1697: (8) bad_scale = [-1]
1698: (8) desired = np.array([1.70020068231762, 1.52054354273631,
-0.34293267607081])
1699: (8) random = Generator(MT19937(self.seed))
1700: (8) gumbel = random.gumbel
1701: (8) actual = gumbel(loc * 3, scale)
1702: (8) assert_array_almost_equal(actual, desired, decimal=14)
1703: (8) assert_raises(ValueError, gumbel, loc * 3, bad_scale)
1704: (8) random = Generator(MT19937(self.seed))
1705: (8) gumbel = random.gumbel
1706: (8) actual = gumbel(loc, scale * 3)
1707: (8) assert_array_almost_equal(actual, desired, decimal=14)
1708: (8) assert_raises(ValueError, gumbel, loc, bad_scale * 3)
1709: (4) def test_logistic(self):
1710: (8) loc = [0]
1711: (8) scale = [1]
1712: (8) bad_scale = [-1]
1713: (8) desired = np.array([-1.607487640433, -1.40925686003678,
1.12887112820397])
1714: (8) random = Generator(MT19937(self.seed))
1715: (8) actual = random.logistic(loc * 3, scale)
1716: (8) assert_array_almost_equal(actual, desired, decimal=14)
1717: (8) assert_raises(ValueError, random.logistic, loc * 3, bad_scale)
1718: (8) random = Generator(MT19937(self.seed))
1719: (8) actual = random.logistic(loc, scale * 3)
1720: (8) assert_array_almost_equal(actual, desired, decimal=14)
1721: (8) assert_raises(ValueError, random.logistic, loc, bad_scale * 3)
1722: (8) assert_equal(random.logistic(1.0, 0.0), 1.0)
1723: (4) def test_lognormal(self):
1724: (8) mean = [0]
1725: (8) sigma = [1]
1726: (8) bad_sigma = [-1]
1727: (8) desired = np.array([0.67884390500697, 2.21653186290321,
1.01990310084276])
1728: (8) random = Generator(MT19937(self.seed))
1729: (8) lognormal = random.lognormal
1730: (8) actual = lognormal(mean * 3, sigma)
1731: (8) assert_array_almost_equal(actual, desired, decimal=14)
1732: (8) assert_raises(ValueError, lognormal, mean * 3, bad_sigma)
1733: (8) random = Generator(MT19937(self.seed))
1734: (8) actual = random.lognormal(mean, sigma * 3)
1735: (8) assert_raises(ValueError, random.lognormal, mean, bad_sigma * 3)
1736: (4) def test_rayleigh(self):
1737: (8) scale = [1]
1738: (8) bad_scale = [-1]
1739: (8) desired = np.array(
1740: (12) [1.1597068009872629,
1741: (13) 0.6539188836253857,
1742: (13) 1.1981526554349398]
1743: (8) )
1744: (8) random = Generator(MT19937(self.seed))
1745: (8) actual = random.rayleigh(scale * 3)
1746: (8) assert_array_almost_equal(actual, desired, decimal=14)
1747: (8) assert_raises(ValueError, random.rayleigh, bad_scale * 3)
1748: (4) def test_wald(self):
1749: (8) mean = [0.5]
1750: (8) scale = [1]
1751: (8) bad_mean = [0]
1752: (8) bad_scale = [-2]
1753: (8) desired = np.array([0.38052407392905, 0.50701641508592,
0.484935249864])
1754: (8) random = Generator(MT19937(self.seed))
1755: (8) actual = random.wald(mean * 3, scale)
1756: (8) assert_array_almost_equal(actual, desired, decimal=14)
1757: (8) assert_raises(ValueError, random.wald, bad_mean * 3, scale)
1758: (8) assert_raises(ValueError, random.wald, mean * 3, bad_scale)
1759: (8) random = Generator(MT19937(self.seed))
1760: (8) actual = random.wald(mean, scale * 3)

```

```

1761: (8) assert_array_almost_equal(actual, desired, decimal=14)
1762: (8) assert_raises(ValueError, random.wald, bad_mean, scale * 3)
1763: (8) assert_raises(ValueError, random.wald, mean, bad_scale * 3)
1764: (4) def test_triangular(self):
1765: (8)     left = [1]
1766: (8)     right = [3]
1767: (8)     mode = [2]
1768: (8)     bad_left_one = [3]
1769: (8)     bad_mode_one = [4]
1770: (8)     bad_left_two, bad_mode_two = right * 2
1771: (8)     desired = np.array([1.57781954604754, 1.62665986867957,
2.30090130831326])
1772: (8)     random = Generator(MT19937(self.seed))
1773: (8)     triangular = random.triangular
1774: (8)     actual = triangular(left * 3, mode, right)
1775: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1776: (8)     assert_raises(ValueError, triangular, bad_left_one * 3, mode, right)
1777: (8)     assert_raises(ValueError, triangular, left * 3, bad_mode_one, right)
1778: (8)     assert_raises(ValueError, triangular, bad_left_two * 3, bad_mode_two,
1779: (22)             right)
1780: (8)     random = Generator(MT19937(self.seed))
1781: (8)     triangular = random.triangular
1782: (8)     actual = triangular(left, mode * 3, right)
1783: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1784: (8)     assert_raises(ValueError, triangular, bad_left_one, mode * 3, right)
1785: (8)     assert_raises(ValueError, triangular, left, bad_mode_one * 3, right)
1786: (8)     assert_raises(ValueError, triangular, bad_left_two, bad_mode_two * 3,
1787: (22)             right)
1788: (8)     random = Generator(MT19937(self.seed))
1789: (8)     triangular = random.triangular
1790: (8)     actual = triangular(left, mode, right * 3)
1791: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1792: (8)     assert_raises(ValueError, triangular, bad_left_one, mode, right * 3)
1793: (8)     assert_raises(ValueError, triangular, left, bad_mode_one, right * 3)
1794: (8)     assert_raises(ValueError, triangular, bad_left_two, bad_mode_two,
1795: (22)             right * 3)
1796: (8)     assert_raises(ValueError, triangular, 10., 0., 20.)
1797: (8)     assert_raises(ValueError, triangular, 10., 25., 20.)
1798: (8)     assert_raises(ValueError, triangular, 10., 10., 10.)
1799: (4) def test_binomial(self):
1800: (8)     n = [1]
1801: (8)     p = [0.5]
1802: (8)     bad_n = [-1]
1803: (8)     bad_p_one = [-1]
1804: (8)     bad_p_two = [1.5]
1805: (8)     desired = np.array([0, 0, 1])
1806: (8)     random = Generator(MT19937(self.seed))
1807: (8)     binom = random.binomial
1808: (8)     actual = binom(n * 3, p)
1809: (8)     assert_array_equal(actual, desired)
1810: (8)     assert_raises(ValueError, binom, bad_n * 3, p)
1811: (8)     assert_raises(ValueError, binom, n * 3, bad_p_one)
1812: (8)     assert_raises(ValueError, binom, n * 3, bad_p_two)
1813: (8)     random = Generator(MT19937(self.seed))
1814: (8)     actual = random.binomial(n, p * 3)
1815: (8)     assert_array_equal(actual, desired)
1816: (8)     assert_raises(ValueError, binom, bad_n, p * 3)
1817: (8)     assert_raises(ValueError, binom, n, bad_p_one * 3)
1818: (8)     assert_raises(ValueError, binom, n, bad_p_two * 3)
1819: (4) def test_negative_binomial(self):
1820: (8)     n = [1]
1821: (8)     p = [0.5]
1822: (8)     bad_n = [-1]
1823: (8)     bad_p_one = [-1]
1824: (8)     bad_p_two = [1.5]
1825: (8)     desired = np.array([0, 2, 1], dtype=np.int64)
1826: (8)     random = Generator(MT19937(self.seed))
1827: (8)     neg_binom = random.negative_binomial
1828: (8)     actual = neg_binom(n * 3, p)

```

```

1829: (8)             assert_array_equal(actual, desired)
1830: (8)             assert_raises(ValueError, neg_binom, bad_n * 3, p)
1831: (8)             assert_raises(ValueError, neg_binom, n * 3, bad_p_one)
1832: (8)             assert_raises(ValueError, neg_binom, n * 3, bad_p_two)
1833: (8)             random = Generator(MT19937(self.seed))
1834: (8)             neg_binom = random.negative_binomial
1835: (8)             actual = neg_binom(n, p * 3)
1836: (8)             assert_array_equal(actual, desired)
1837: (8)             assert_raises(ValueError, neg_binom, bad_n, p * 3)
1838: (8)             assert_raises(ValueError, neg_binom, n, bad_p_one * 3)
1839: (8)             assert_raises(ValueError, neg_binom, n, bad_p_two * 3)
1840: (4) def test_poisson(self):
1841: (8)     lam = [1]
1842: (8)     bad_lam_one = [-1]
1843: (8)     desired = np.array([0, 0, 3])
1844: (8)     random = Generator(MT19937(self.seed))
1845: (8)     max_lam = random._poisson_lam_max
1846: (8)     bad_lam_two = [max_lam * 2]
1847: (8)     poisson = random.poisson
1848: (8)     actual = poisson(lam * 3)
1849: (8)     assert_array_equal(actual, desired)
1850: (8)     assert_raises(ValueError, poisson, bad_lam_one * 3)
1851: (8)     assert_raises(ValueError, poisson, bad_lam_two * 3)
1852: (4) def test_zipf(self):
1853: (8)     a = [2]
1854: (8)     bad_a = [0]
1855: (8)     desired = np.array([1, 8, 1])
1856: (8)     random = Generator(MT19937(self.seed))
1857: (8)     zipf = random.zipf
1858: (8)     actual = zipf(a * 3)
1859: (8)     assert_array_equal(actual, desired)
1860: (8)     assert_raises(ValueError, zipf, bad_a * 3)
1861: (8)     with np.errstate(invalid='ignore'):
1862: (12)         assert_raises(ValueError, zipf, np.nan)
1863: (12)         assert_raises(ValueError, zipf, [0, 0, np.nan])
1864: (4) def test_geometric(self):
1865: (8)     p = [0.5]
1866: (8)     bad_p_one = [-1]
1867: (8)     bad_p_two = [1.5]
1868: (8)     desired = np.array([1, 1, 3])
1869: (8)     random = Generator(MT19937(self.seed))
1870: (8)     geometric = random.geometric
1871: (8)     actual = geometric(p * 3)
1872: (8)     assert_array_equal(actual, desired)
1873: (8)     assert_raises(ValueError, geometric, bad_p_one * 3)
1874: (8)     assert_raises(ValueError, geometric, bad_p_two * 3)
1875: (4) def test_hypergeometric(self):
1876: (8)     ngood = [1]
1877: (8)     nbad = [2]
1878: (8)     nsample = [2]
1879: (8)     bad_ngood = [-1]
1880: (8)     bad_nbad = [-2]
1881: (8)     bad_nsampel_one = [-1]
1882: (8)     bad_nsampel_two = [4]
1883: (8)     desired = np.array([0, 0, 1])
1884: (8)     random = Generator(MT19937(self.seed))
1885: (8)     actual = random.hypergeometric(ngood * 3, nbad, nsample)
1886: (8)     assert_array_equal(actual, desired)
1887: (8)     assert_raises(ValueError, random.hypergeometric, bad_ngood * 3, nbad,
1888: (8)     nsample)
1889: (8)     assert_raises(ValueError, random.hypergeometric, ngood * 3, bad_nbad,
1890: (8)     nsample)
1891: (8)     assert_raises(ValueError, random.hypergeometric, ngood * 3, nbad,
1892: (8)     bad_nsampel_one)
1893: (8)     assert_raises(ValueError, random.hypergeometric, ngood * 3, nbad,
1894: (8)     bad_nsampel_two)
1895: (8)     random = Generator(MT19937(self.seed))
1896: (8)     actual = random.hypergeometric(ngood, nbad * 3, nsample)
1897: (8)     assert_array_equal(actual, desired)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1894: (8) assert_raises(ValueError, random.hypergeometric, bad_ngood, nbad * 3,
nsample)
1895: (8) assert_raises(ValueError, random.hypergeometric, ngood, bad_nbad * 3,
nsample)
1896: (8) assert_raises(ValueError, random.hypergeometric, ngood, nbad * 3,
bad_nsampel_one)
1897: (8) assert_raises(ValueError, random.hypergeometric, ngood, nbad * 3,
bad_nsampel_two)
1898: (8) random = Generator(MT19937(self.seed))
1899: (8) hypergeom = random.hypergeometric
1900: (8) actual = hypergeom(ngood, nbad, nsample * 3)
1901: (8) assert_array_equal(actual, desired)
1902: (8) assert_raises(ValueError, hypergeom, bad_ngood, nbad, nsample * 3)
1903: (8) assert_raises(ValueError, hypergeom, ngood, bad_nbad, nsample * 3)
1904: (8) assert_raises(ValueError, hypergeom, ngood, nbad, bad_nsampel_one * 3)
1905: (8) assert_raises(ValueError, hypergeom, ngood, nbad, bad_nsampel_two * 3)
1906: (8) assert_raises(ValueError, hypergeom, -1, 10, 20)
1907: (8) assert_raises(ValueError, hypergeom, 10, -1, 20)
1908: (8) assert_raises(ValueError, hypergeom, 10, 10, -1)
1909: (8) assert_raises(ValueError, hypergeom, 10, 10, 25)
1910: (8) assert_raises(ValueError, hypergeom, 2**30, 10, 20)
1911: (8) assert_raises(ValueError, hypergeom, 999, 2**31, 50)
1912: (8) assert_raises(ValueError, hypergeom, 999, [2**29, 2**30], 1000)
1913: (4) def test_logseries(self):
1914: (8)     p = [0.5]
1915: (8)     bad_p_one = [2]
1916: (8)     bad_p_two = [-1]
1917: (8)     desired = np.array([1, 1, 1])
1918: (8)     random = Generator(MT19937(self.seed))
1919: (8)     logseries = random.logseries
1920: (8)     actual = logseries(p * 3)
1921: (8)     assert_array_equal(actual, desired)
1922: (8)     assert_raises(ValueError, logseries, bad_p_one * 3)
1923: (8)     assert_raises(ValueError, logseries, bad_p_two * 3)
1924: (4) def test_multinomial(self):
1925: (8)     random = Generator(MT19937(self.seed))
1926: (8)     actual = random.multinomial([5, 20], [1 / 6.] * 6, size=(3, 2))
1927: (8)     desired = np.array([[0, 0, 2, 1, 2, 0],
1928: (29)                 [2, 3, 6, 4, 2, 3],
1929: (28)                 [[1, 0, 1, 0, 2, 1],
1930: (29)                 [7, 2, 2, 1, 4, 4],
1931: (28)                 [[0, 2, 0, 1, 2, 0],
1932: (29)                 [3, 2, 3, 3, 4, 5]]], dtype=np.int64)
1933: (8)     assert_array_equal(actual, desired)
1934: (8)     random = Generator(MT19937(self.seed))
1935: (8)     actual = random.multinomial([5, 20], [1 / 6.] * 6)
1936: (8)     desired = np.array([[0, 0, 2, 1, 2, 0],
1937: (28)                 [2, 3, 6, 4, 2, 3]], dtype=np.int64)
1938: (8)     assert_array_equal(actual, desired)
1939: (8)     random = Generator(MT19937(self.seed))
1940: (8)     actual = random.multinomial([5, 20], [[1 / 6.] * 6] * 2)
1941: (8)     desired = np.array([[0, 0, 2, 1, 2, 0],
1942: (28)                 [2, 3, 6, 4, 2, 3]], dtype=np.int64)
1943: (8)     assert_array_equal(actual, desired)
1944: (8)     random = Generator(MT19937(self.seed))
1945: (8)     actual = random.multinomial([[5], [20]], [[1 / 6.] * 6] * 2)
1946: (8)     desired = np.array([[0, 0, 2, 1, 2, 0],
1947: (29)                 [0, 0, 2, 1, 1, 1],
1948: (28)                 [[4, 2, 3, 3, 5, 3],
1949: (29)                 [7, 2, 2, 1, 4, 4]]], dtype=np.int64)
1950: (8)     assert_array_equal(actual, desired)
1951: (4)     @pytest.mark.parametrize("n", [10,
1952: (35)                     np.array([10, 10]),
1953: (35)                     np.array([[10]], [[10]]),
1954: (35)                     ],
1955: (29)                     )
1956: (4)     def test_multinomial_pval_broadcast(self, n):
1957: (8)         random = Generator(MT19937(self.seed))
1958: (8)         pvals = np.array([1 / 4] * 4)

```

```

1959: (8)             actual = random.multinomial(n, pvals)
1960: (8)             n_shape = tuple() if isinstance(n, int) else n.shape
1961: (8)             expected_shape = n_shape + (4,)
1962: (8)             assert actual.shape == expected_shape
1963: (8)             pvals = np.vstack([pvals, pvals])
1964: (8)             actual = random.multinomial(n, pvals)
1965: (8)             expected_shape = np.broadcast_shapes(n_shape, pvals.shape[:-1]) + (4,)
1966: (8)             assert actual.shape == expected_shape
1967: (8)             pvals = np.vstack([[pvals], [pvals]])
1968: (8)             actual = random.multinomial(n, pvals)
1969: (8)             expected_shape = np.broadcast_shapes(n_shape, pvals.shape[:-1])
1970: (8)             assert actual.shape == expected_shape + (4,)
1971: (8)             actual = random.multinomial(n, pvals, size=(3, 2) + expected_shape)
1972: (8)             assert actual.shape == (3, 2) + expected_shape + (4,)
1973: (8)             with pytest.raises(ValueError):
1974: (12)                 actual = random.multinomial(n, pvals, size=(1,) * 6)
1975: (4)             def test_invalid_pvals_broadcast(self):
1976: (8)                 random = Generator(MT19937(self.seed))
1977: (8)                 pvals = [[1 / 6] * 6, [1 / 4] * 6]
1978: (8)                 assert_raises(ValueError, random.multinomial, 1, pvals)
1979: (8)                 assert_raises(ValueError, random.multinomial, 6, 0.5)
1980: (4)             def test_empty_outputs(self):
1981: (8)                 random = Generator(MT19937(self.seed))
1982: (8)                 actual = random.multinomial(np.empty((10, 0, 6), "i8"), [1 / 6] * 6)
1983: (8)                 assert actual.shape == (10, 0, 6, 6)
1984: (8)                 actual = random.multinomial(12, np.empty((10, 0, 10)))
1985: (8)                 assert actual.shape == (10, 0, 10)
1986: (8)                 actual = random.multinomial(np.empty((3, 0, 7), "i8"),
1987: (36)                               np.empty((3, 0, 7, 4)))
1988: (8)                 assert actual.shape == (3, 0, 7, 4)
1989: (0)             @pytest.mark.skipif(IS_WASM, reason="can't start thread")
1990: (0)             class TestThread:
1991: (4)                 def setup_method(self):
1992: (8)                     self.seeds = range(4)
1993: (4)                 def check_function(self, function, sz):
1994: (8)                     from threading import Thread
1995: (8)                     out1 = np.empty((len(self.seeds),) + sz)
1996: (8)                     out2 = np.empty((len(self.seeds),) + sz)
1997: (8)                     t = [Thread(target=function, args=(Generator(MT19937(s)), o))
1998: (13)                         for s, o in zip(self.seeds, out1)]
1999: (8)                     [x.start() for x in t]
2000: (8)                     [x.join() for x in t]
2001: (8)                     for s, o in zip(self.seeds, out2):
2002: (12)                         function(Generator(MT19937(s)), o)
2003: (8)                     if np.intp().dtype.itemsize == 4 and sys.platform == "win32":
2004: (12)                         assert_array_almost_equal(out1, out2)
2005: (8)                     else:
2006: (12)                         assert_array_equal(out1, out2)
2007: (4)             def test_normal(self):
2008: (8)                 def gen_random(state, out):
2009: (12)                     out[...] = state.normal(size=10000)
2010: (8)                     self.check_function(gen_random, sz=(10000,))
2011: (4)             def test_exp(self):
2012: (8)                 def gen_random(state, out):
2013: (12)                     out[...] = state.exponential(scale=np.ones((100, 1000)))
2014: (8)                     self.check_function(gen_random, sz=(100, 1000))
2015: (4)             def test_multinomial(self):
2016: (8)                 def gen_random(state, out):
2017: (12)                     out[...] = state.multinomial(10, [1 / 6.] * 6, size=10000)
2018: (8)                     self.check_function(gen_random, sz=(10000, 6))
2019: (0)             class TestSingleEltArrayInput:
2020: (4)                 def setup_method(self):
2021: (8)                     self.argOne = np.array([2])
2022: (8)                     self.argTwo = np.array([3])
2023: (8)                     self.argThree = np.array([4])
2024: (8)                     self.tgtShape = (1,)
2025: (4)                 def test_one_arg_funcs(self):
2026: (8)                     funcs = (random.exponential, random.standard_gamma,
2027: (17)                           random.chisquare, random.standard_t,

```

```

2028: (17)                     random.pareto, random.weibull,
2029: (17)                     random.power, random.rayleigh,
2030: (17)                     random.poisson, random.zipf,
2031: (17)                     random.geometric, random.logseries)
2032: (8)                      probfuncs = (random.geometric, random.logseries)
2033: (8)                      for func in funcs:
2034: (12)                        if func in probfuncs: # p < 1.0
2035: (16)                          out = func(np.array([0.5]))
2036: (12)                        else:
2037: (16)                          out = func(self.argOne)
2038: (12)                          assert_equal(out.shape, self.tgtShape)
2039: (4)                           def test_two_arg_funcs(self):
2040: (8)                             funcs = (random.uniform, random.normal,
2041: (17)                               random.beta, random.gamma,
2042: (17)                               random.f, random.noncentral_chisquare,
2043: (17)                               random.vonmises, random.laplace,
2044: (17)                               random.gumbel, random.logistic,
2045: (17)                               random.lognormal, random.wald,
2046: (17)                               random.binomial, random.negative_binomial)
2047: (8)                             probfuncs = (random.binomial, random.negative_binomial)
2048: (8)                             for func in funcs:
2049: (12)                               if func in probfuncs: # p <= 1
2050: (16)                                 argTwo = np.array([0.5])
2051: (12)                               else:
2052: (16)                                 argTwo = self.argTwo
2053: (12)                                 out = func(self.argOne, argTwo)
2054: (12)                                 assert_equal(out.shape, self.tgtShape)
2055: (12)                                 out = func(self.argOne[0], argTwo)
2056: (12)                                 assert_equal(out.shape, self.tgtShape)
2057: (12)                                 out = func(self.argOne, argTwo[0])
2058: (12)                                 assert_equal(out.shape, self.tgtShape)
2059: (4)                           def test_integers(self, endpoint):
2060: (8)                             itype = [np.bool_, np.int8, np.uint8, np.int16, np.uint16,
2061: (17)                               np.int32, np.uint32, np.int64, np.uint64]
2062: (8)                             func = random.integers
2063: (8)                             high = np.array([1])
2064: (8)                             low = np.array([0])
2065: (8)                             for dt in itype:
2066: (12)                               out = func(low, high, endpoint=endpoint, dtype=dt)
2067: (12)                               assert_equal(out.shape, self.tgtShape)
2068: (12)                               out = func(low[0], high, endpoint=endpoint, dtype=dt)
2069: (12)                               assert_equal(out.shape, self.tgtShape)
2070: (12)                               out = func(low, high[0], endpoint=endpoint, dtype=dt)
2071: (12)                               assert_equal(out.shape, self.tgtShape)
2072: (4)                           def test_three_arg_funcs(self):
2073: (8)                             funcs = [random.noncentral_f, random.triangular,
2074: (17)                               random.hypergeometric]
2075: (8)                             for func in funcs:
2076: (12)                               out = func(self.argOne, self.argTwo, self.argThree)
2077: (12)                               assert_equal(out.shape, self.tgtShape)
2078: (12)                               out = func(self.argOne[0], self.argTwo, self.argThree)
2079: (12)                               assert_equal(out.shape, self.tgtShape)
2080: (12)                               out = func(self.argOne, self.argTwo[0], self.argThree)
2081: (12)                               assert_equal(out.shape, self.tgtShape)
2082: (0) @pytest.mark.parametrize("config", JUMP_TEST_DATA)
2083: (0) def test_jumped(config):
2084: (4)   seed = config["seed"]
2085: (4)   steps = config["steps"]
2086: (4)   mt19937 = MT19937(seed)
2087: (4)   mt19937.random_raw(steps)
2088: (4)   key = mt19937.state["state"]["key"]
2089: (4)   if sys.byteorder == 'big':
2090: (8)     key = key.byteswap()
2091: (4)   sha256 = hashlib.sha256(key)
2092: (4)   assert mt19937.state["state"]["pos"] == config["initial"]["pos"]
2093: (4)   assert sha256.hexdigest() == config["initial"]["key_sha256"]
2094: (4)   jumped = mt19937.jumped()
2095: (4)   key = jumped.state["state"]["key"]
2096: (4)   if sys.byteorder == 'big':

```

```

2097: (8)             key = key.byteswap()
2098: (4)             sha256 = hashlib.sha256(key)
2099: (4)             assert jumped.state["state"]["pos"] == config["jumped"]["pos"]
2100: (4)             assert sha256.hexdigest() == config["jumped"]["key_sha256"]
2101: (0)             def test_broadcast_size_error():
2102: (4)                 mu = np.ones(3)
2103: (4)                 sigma = np.ones((4, 3))
2104: (4)                 size = (10, 4, 2)
2105: (4)                 assert random.normal(mu, sigma, size=(5, 4, 3)).shape == (5, 4, 3)
2106: (4)                 with pytest.raises(ValueError):
2107: (8)                     random.normal(mu, sigma, size=size)
2108: (4)                 with pytest.raises(ValueError):
2109: (8)                     random.normal(mu, sigma, size=(1, 3))
2110: (4)                 with pytest.raises(ValueError):
2111: (8)                     random.normal(mu, sigma, size=(4, 1, 1))
2112: (4)                 shape = np.ones((4, 3))
2113: (4)                 with pytest.raises(ValueError):
2114: (8)                     random.standard_gamma(shape, size=size)
2115: (4)                 with pytest.raises(ValueError):
2116: (8)                     random.standard_gamma(shape, size=(3,))
2117: (4)                 with pytest.raises(ValueError):
2118: (8)                     random.standard_gamma(shape, size=3)
2119: (4)                 out = np.empty(size)
2120: (4)                 with pytest.raises(ValueError):
2121: (8)                     random.standard_gamma(shape, out=out)
2122: (4)                 with pytest.raises(ValueError):
2123: (8)                     random.binomial(1, [0.3, 0.7], size=(2, 1))
2124: (4)                 with pytest.raises(ValueError):
2125: (8)                     random.binomial([1, 2], 0.3, size=(2, 1))
2126: (4)                 with pytest.raises(ValueError):
2127: (8)                     random.binomial([1, 2], [0.3, 0.7], size=(2, 1))
2128: (4)                 with pytest.raises(ValueError):
2129: (8)                     random.multinomial([2, 2], [.3, .7], size=(2, 1))
2130: (4)                 a = random.chisquare(5, size=3)
2131: (4)                 b = random.chisquare(5, size=(4, 3))
2132: (4)                 c = random.chisquare(5, size=(5, 4, 3))
2133: (4)                 assert random.noncentral_f(a, b, c).shape == (5, 4, 3)
2134: (4)                 with pytest.raises(ValueError, match=r"Output size \[6, 5, 1, 1\] is"):
2135: (8)                     random.noncentral_f(a, b, c, size=(6, 5, 1, 1))
2136: (0)             def test_broadcast_size_scalar():
2137: (4)                 mu = np.ones(3)
2138: (4)                 sigma = np.ones(3)
2139: (4)                 random.normal(mu, sigma, size=3)
2140: (4)                 with pytest.raises(ValueError):
2141: (8)                     random.normal(mu, sigma, size=2)
2142: (0)             def test_ragged_shuffle():
2143: (4)                 seq = [[], [], 1]
2144: (4)                 gen = Generator(MT19937(0))
2145: (4)                 assert_no_warnings(gen.shuffle, seq)
2146: (4)                 assert seq == [1, [], []]
2147: (0)                 @pytest.mark.parametrize("high", [-2, [-2]])
2148: (0)                 @pytest.mark.parametrize("endpoint", [True, False])
2149: (0)                 def test_single_arg_integer_exception(high, endpoint):
2150: (4)                     gen = Generator(MT19937(0))
2151: (4)                     msg = 'high < 0' if endpoint else 'high <= 0'
2152: (4)                     with pytest.raises(ValueError, match=msg):
2153: (8)                         gen.integers(high, endpoint=endpoint)
2154: (4)                     msg = 'low > high' if endpoint else 'low >= high'
2155: (4)                     with pytest.raises(ValueError, match=msg):
2156: (8)                         gen.integers(-1, high, endpoint=endpoint)
2157: (4)                     with pytest.raises(ValueError, match=msg):
2158: (8)                         gen.integers([-1], high, endpoint=endpoint)
2159: (0)                 @pytest.mark.parametrize("dtype", ["f4", "f8"])
2160: (0)                 def test_c_contig_req_out(dtype):
2161: (4)                     out = np.empty((2, 3), order="F", dtype=dtype)
2162: (4)                     shape = [1, 2, 3]
2163: (4)                     with pytest.raises(ValueError, match="Supplied output array"):
2164: (8)                         random.standard_gamma(shape, out=out, dtype=dtype)
2165: (4)                     with pytest.raises(ValueError, match="Supplied output array"):

```

```

2166: (8)             random.standard_gamma(shape, out=out, size=out.shape, dtype=dtype)
2167: (0)             @pytest.mark.parametrize("dtype", ["f4", "f8"])
2168: (0)             @pytest.mark.parametrize("order", ["F", "C"])
2169: (0)             @pytest.mark.parametrize("dist", [random.standard_normal, random.random])
2170: (0)             def test_contig_req_out(dist, order, dtype):
2171: (4)                 out = np.empty((2, 3), dtype=dtype, order=order)
2172: (4)                 variates = dist(out=out, dtype=dtype)
2173: (4)                 assert variates is out
2174: (4)                 variates = dist(out=out, dtype=dtype, size=out.shape)
2175: (4)                 assert variates is out
2176: (0)             def test_generator_ctor_old_style_pickle():
2177: (4)                 rg = np.random.Generator(np.random.PCG64DXSM(0))
2178: (4)                 rg.standard_normal(1)
2179: (4)                 ctor, args, state_a = rg.__reduce__()
2180: (4)                 assert args[:1] == ("PCG64DXSM",)
2181: (4)                 b = ctor(*args[:1])
2182: (4)                 b.bit_generator.state = state_a
2183: (4)                 state_b = b.bit_generator.state
2184: (4)                 assert state_a == state_b
-----
```

File 313 - test\_generator\_mt19937\_regressions.py:

```

1: (0)             from numpy.testing import (assert_, assert_array_equal)
2: (0)             import numpy as np
3: (0)             import pytest
4: (0)             from numpy.random import Generator, MT19937
5: (0)             class TestRegression:
6: (4)                 def setup_method(self):
7: (8)                     self.mt19937 = Generator(MT19937(121263137472525314065))
8: (4)                 def test_vonmises_range(self):
9: (8)                     for mu in np.linspace(-7., 7., 5):
10: (12)                         r = self.mt19937.vonmises(mu, 1, 50)
11: (12)                         assert_(np.all(r > -np.pi) and np.all(r <= np.pi))
12: (4)                 def test_hypergeometric_range(self):
13: (8)                     assert_(np.all(self.mt19937.hypergeometric(3, 18, 11, size=10) < 4))
14: (8)                     assert_(np.all(self.mt19937.hypergeometric(18, 3, 11, size=10) > 0))
15: (8)                     args = (2**20 - 2, 2**20 - 2, 2**20 - 2) # Check for 32-bit systems
16: (8)                     assert_(self.mt19937.hypergeometric(*args) > 0)
17: (4)                 def test_logseries_convergence(self):
18: (8)                     N = 1000
19: (8)                     rvsn = self.mt19937.logseries(0.8, size=N)
20: (8)                     freq = np.sum(rvsn == 1) / N
21: (8)                     msg = f'Frequency was {freq:f}, should be > 0.45'
22: (8)                     assert_(freq > 0.45, msg)
23: (8)                     freq = np.sum(rvsn == 2) / N
24: (8)                     msg = f'Frequency was {freq:f}, should be < 0.23'
25: (8)                     assert_(freq < 0.23, msg)
26: (4)                 def test_shuffle_mixed_dimension(self):
27: (8)                     for t in [[1, 2, 3, None],
28: (18)                         [(1, 1), (2, 2), (3, 3), None],
29: (18)                         [1, (2, 2), (3, 3), None],
30: (18)                         [(1, 1), 2, 3, None]]:
31: (12)                         mt19937 = Generator(MT19937(12345))
32: (12)                         shuffled = np.array(t, dtype=object)
33: (12)                         mt19937.shuffle(shuffled)
34: (12)                         expected = np.array([t[2], t[0], t[3], t[1]], dtype=object)
35: (12)                         assert_array_equal(np.array(shuffled, dtype=object), expected)
36: (4)                 def test_call_within_randomstate(self):
37: (8)                     res = np.array([1, 8, 0, 1, 5, 3, 3, 8, 1, 4])
38: (8)                     for i in range(3):
39: (12)                         mt19937 = Generator(MT19937(i))
40: (12)                         m = Generator(MT19937(4321))
41: (12)                         assert_array_equal(m.choice(10, size=10, p=np.ones(10)/10.), res)
42: (4)                 def test_multivariate_normal_size_types(self):
43: (8)                     self.mt19937.multivariate_normal([0], [[0]], size=1)
44: (8)                     self.mt19937.multivariate_normal([0], [[0]], size=np.int_(1))
45: (8)                     self.mt19937.multivariate_normal([0], [[0]], size=np.int64(1))
```

```

46: (4)         def test_beta_small_parameters(self):
47: (8)             x = self.mt19937.beta(0.0001, 0.0001, size=100)
48: (8)             assert_(not np.any(np.isnan(x)), 'Nans in mt19937.beta')
49: (4)         def test_beta_very_small_parameters(self):
50: (8)             self.mt19937.beta(1e-49, 1e-40)
51: (4)         def test_beta_ridiculously_small_parameters(self):
52: (8)             tiny = np.finfo(1.0).tiny
53: (8)             x = self.mt19937.beta(tiny/32, tiny/40, size=50)
54: (8)             assert not np.any(np.isnan(x))
55: (4)         def test_choice_sum_of_probs_tolerance(self):
56: (8)             a = [1, 2, 3]
57: (8)             counts = [4, 4, 2]
58: (8)             for dt in np.float16, np.float32, np.float64:
59: (12)                 probs = np.array(counts, dtype=dt) / sum(counts)
60: (12)                 c = self.mt19937.choice(a, p=probs)
61: (12)                 assert_(c in a)
62: (12)                 with pytest.raises(ValueError):
63: (16)                     self.mt19937.choice(a, p=probs*0.9)
64: (4)         def test_shuffle_of_array_of_different_length_strings(self):
65: (8)             a = np.array(['a', 'a' * 1000])
66: (8)             for _ in range(1000):
67: (12)                 self.mt19937.shuffle(a)
68: (8)             import gc
69: (8)             gc.collect()
70: (4)         def test_shuffle_of_array_of_objects(self):
71: (8)             a = np.array([np.arange(1), np.arange(4)], dtype=object)
72: (8)             for _ in range(1000):
73: (12)                 self.mt19937.shuffle(a)
74: (8)             import gc
75: (8)             gc.collect()
76: (4)         def test_permutation_subclass(self):
77: (8)             class N(np.ndarray):
78: (12)                 pass
79: (8)             mt19937 = Generator(MT19937(1))
80: (8)             orig = np.arange(3).view(N)
81: (8)             perm = mt19937.permutation(orig)
82: (8)             assert_array_equal(perm, np.array([2, 0, 1]))
83: (8)             assert_array_equal(orig, np.arange(3).view(N))
84: (8)
85: (12)             class M:
86: (12)                 a = np.arange(5)
87: (16)                 def __array__(self):
88: (16)                     return self.a
89: (8)             mt19937 = Generator(MT19937(1))
90: (8)             m = M()
91: (8)             perm = mt19937.permutation(m)
92: (8)             assert_array_equal(perm, np.array([4, 1, 3, 0, 2]))
93: (8)             assert_array_equal(m.__array__(), np.arange(5))
94: (4)         def test_gamma_0(self):
95: (8)             assert self.mt19937.standard_gamma(0.0) == 0.0
96: (8)             assert_array_equal(self.mt19937.standard_gamma([0.0]), 0.0)
97: (8)             actual = self.mt19937.standard_gamma([0.0], dtype='float')
98: (8)             expected = np.array([0.], dtype=np.float32)
99: (4)             assert_array_equal(actual, expected)
100: (8)         def test_geometric_tiny_prob(self):
101: (27)             assert_array_equal(self.mt19937.geometric(p=1e-30, size=3),
101: (27)                             np.iinfo(np.int64).max)

```

-----  
File 314 - test\_random.py:

```

1: (0)         import warnings
2: (0)         import pytest
3: (0)         import numpy as np
4: (0)         from numpy.testing import (
5: (8)             assert_, assert_raises, assert_equal, assert_warns,
6: (8)             assert_no_warnings, assert_array_equal, assert_array_almost_equal,
7: (8)             suppress_warnings, IS_WASM
8: (8)         )

```

```

9: (0)
10: (0)
11: (0)
12: (4)
13: (8)
14: (8)
15: (8)
16: (8)
17: (4)
18: (8)
19: (8)
20: (8)
21: (8)
22: (8)
23: (8)
24: (8)
25: (8)
26: (4)
27: (8)
28: (8)
29: (4)
30: (8)
31: (8)
32: (8)
33: (8)
34: (8)
35: (4)
36: (8)
37: (22)
38: (8)
39: (8)
40: (58)
41: (0)
42: (4)
43: (8)
44: (8)
45: (12)
46: (12)
47: (4)
48: (8)
49: (0)
50: (4)
51: (8)
52: (4)
53: (8)
54: (4)
55: (8)
56: (8)
57: (8)
58: (8)
59: (4)
60: (8)
61: (8)
62: (8)
63: (8)
64: (8)
65: (8)
66: (8)
67: (21)
68: (8)
69: (22)
70: (4)
71: (8)
72: (8)
73: (8)
[[1], [0]]])
74: (8)
[1, 0]]))
75: (0)
         class TestSetState:
from numpy import random
import sys
class TestSeed:
    def test_scalar(self):
        s = np.random.RandomState(0)
        assert_equal(s.randint(1000), 684)
        s = np.random.RandomState(4294967295)
        assert_equal(s.randint(1000), 419)
    def test_array(self):
        s = np.random.RandomState(range(10))
        assert_equal(s.randint(1000), 468)
        s = np.random.RandomState(np.arange(10))
        assert_equal(s.randint(1000), 468)
        s = np.random.RandomState([0])
        assert_equal(s.randint(1000), 973)
        s = np.random.RandomState([4294967295])
        assert_equal(s.randint(1000), 265)
    def test_invalid_scalar(self):
        assert_raises(TypeError, np.random.RandomState, -0.5)
        assert_raises(ValueError, np.random.RandomState, -1)
    def test_invalid_array(self):
        assert_raises(TypeError, np.random.RandomState, [-0.5])
        assert_raises(ValueError, np.random.RandomState, [-1])
        assert_raises(ValueError, np.random.RandomState, [4294967296])
        assert_raises(ValueError, np.random.RandomState, [1, 2, 4294967296])
        assert_raises(ValueError, np.random.RandomState, [1, -2, 4294967296])
    def test_invalid_array_shape(self):
        assert_raises(ValueError, np.random.RandomState,
                     np.array([], dtype=np.int64))
        assert_raises(ValueError, np.random.RandomState, [[1, 2, 3]])
        assert_raises(ValueError, np.random.RandomState, [[1, 2, 3],
                                                       [4, 5, 6]])
class TestBinomial:
    def test_n_zero(self):
        zeros = np.zeros(2, dtype='int')
        for p in [0, .5, 1]:
            assert_(random.binomial(0, p) == 0)
            assert_array_equal(random.binomial(zeros, p), zeros)
    def test_p_is_nan(self):
        assert_raises(ValueError, random.binomial, 1, np.nan)
class TestMultinomial:
    def test_basic(self):
        random.multinomial(100, [0.2, 0.8])
    def test_zero_probability(self):
        random.multinomial(100, [0.2, 0.8, 0.0, 0.0, 0.0])
    def test_int_negative_interval(self):
        assert_(-5 <= random.randint(-5, -1) < -1)
        x = random.randint(-5, -1, 5)
        assert_(np.all(-5 <= x))
        assert_(np.all(x < -1))
    def test_size(self):
        p = [0.5, 0.5]
        assert_equal(np.random.multinomial(1, p, np.uint32(1)).shape, (1, 2))
        assert_equal(np.random.multinomial(1, p, np.uint32(1)).shape, (1, 2))
        assert_equal(np.random.multinomial(1, p, np.uint32(1)).shape, (1, 2))
        assert_equal(np.random.multinomial(1, p, [2, 2]).shape, (2, 2, 2))
        assert_equal(np.random.multinomial(1, p, (2, 2)).shape, (2, 2, 2))
        assert_equal(np.random.multinomial(1, p, np.array((2, 2))).shape,
                    (2, 2, 2))
        assert_raises(TypeError, np.random.multinomial, 1, p,
                      float(1))
    def test_multidimensional_pvals(self):
        assert_raises(ValueError, np.random.multinomial, 10, [[0, 1]])
        assert_raises(ValueError, np.random.multinomial, 10, [[0], [1]])
        assert_raises(ValueError, np.random.multinomial, 10, [[[0], [1]],
[[1], [0]]]))
        assert_raises(ValueError, np.random.multinomial, 10, np.array([[0, 1],
[1, 0]])))

```

```

76: (4)         def setup_method(self):
77: (8)             self.seed = 1234567890
78: (8)             self.prng = random.RandomState(self.seed)
79: (8)             self.state = self.prng.get_state()
80: (4)         def test_basic(self):
81: (8)             old = self.prng.tomaxint(16)
82: (8)             self.prng.set_state(self.state)
83: (8)             new = self.prng.tomaxint(16)
84: (8)             assert_(np.all(old == new))
85: (4)         def test_gaussian_reset(self):
86: (8)             old = self.prng.standard_normal(size=3)
87: (8)             self.prng.set_state(self.state)
88: (8)             new = self.prng.standard_normal(size=3)
89: (8)             assert_(np.all(old == new))
90: (4)         def test_gaussian_reset_in_media_res(self):
91: (8)             self.prng.standard_normal()
92: (8)             state = self.prng.get_state()
93: (8)             old = self.prng.standard_normal(size=3)
94: (8)             self.prng.set_state(state)
95: (8)             new = self.prng.standard_normal(size=3)
96: (8)             assert_(np.all(old == new))
97: (4)         def test_backwards_compatibility(self):
98: (8)             old_state = self.state[:-2]
99: (8)             x1 = self.prng.standard_normal(size=16)
100: (8)            self.prng.set_state(old_state)
101: (8)            x2 = self.prng.standard_normal(size=16)
102: (8)            self.prng.set_state(self.state)
103: (8)            x3 = self.prng.standard_normal(size=16)
104: (8)            assert_(np.all(x1 == x2))
105: (8)            assert_(np.all(x1 == x3))
106: (4)         def test_negative_binomial(self):
107: (8)             self.prng.negative_binomial(0.5, 0.5)
108: (4)         def test_set_invalid_state(self):
109: (8)             with pytest.raises(IndexError):
110: (12)                 self.prng.set_state(())
111: (0)         class TestRandint:
112: (4)             rfunc = np.random.randint
113: (4)             itype = [np.bool_, np.int8, np.uint8, np.int16, np.uint16,
114: (13)                 np.int32, np.uint32, np.int64, np.uint64]
115: (4)         def test_unsupported_type(self):
116: (8)             assert_raises(TypeError, self.rfunc, 1, dtype=float)
117: (4)         def test_bounds_checking(self):
118: (8)             for dt in self.itype:
119: (12)                 lbnd = 0 if dt is np.bool_ else np.iinfo(dt).min
120: (12)                 ubnd = 2 if dt is np.bool_ else np.iinfo(dt).max + 1
121: (12)                 assert_raises(ValueError, self.rfunc, lbnd - 1, ubnd, dtype=dt)
122: (12)                 assert_raises(ValueError, self.rfunc, lbnd, ubnd + 1, dtype=dt)
123: (12)                 assert_raises(ValueError, self.rfunc, ubnd, lbnd, dtype=dt)
124: (12)                 assert_raises(ValueError, self.rfunc, 1, 0, dtype=dt)
125: (4)         def test_rng_zero_and_extremes(self):
126: (8)             for dt in self.itype:
127: (12)                 lbnd = 0 if dt is np.bool_ else np.iinfo(dt).min
128: (12)                 ubnd = 2 if dt is np.bool_ else np.iinfo(dt).max + 1
129: (12)                 tgt = ubnd - 1
130: (12)                 assert_equal(self.rfunc(tgt, tgt + 1, size=1000, dtype=dt), tgt)
131: (12)                 tgt = lbnd
132: (12)                 assert_equal(self.rfunc(tgt, tgt + 1, size=1000, dtype=dt), tgt)
133: (12)                 tgt = (lbnd + ubnd)//2
134: (12)                 assert_equal(self.rfunc(tgt, tgt + 1, size=1000, dtype=dt), tgt)
135: (4)         def test_full_range(self):
136: (8)             for dt in self.itype:
137: (12)                 lbnd = 0 if dt is np.bool_ else np.iinfo(dt).min
138: (12)                 ubnd = 2 if dt is np.bool_ else np.iinfo(dt).max + 1
139: (12)                 try:
140: (16)                     self.rfunc(lbnd, ubnd, dtype=dt)
141: (12)                 except Exception as e:
142: (16)                     raise AssertionError("No error should have been raised, "
143: (37)                         "but one was with the following "
144: (37)                         "message:\n\n%s" % str(e))

```

```

145: (4)             def test_in_bounds_fuzz(self):
146: (8)                 np.random.seed()
147: (8)                 for dt in self.dtype[1:]:
148: (12)                     for ubnd in [4, 8, 16]:
149: (16)                         vals = self.rfunc(2, ubnd, size=2**16, dtype=dt)
150: (16)                         assert_(vals.max() < ubnd)
151: (16)                         assert_(vals.min() >= 2)
152: (8)                         vals = self.rfunc(0, 2, size=2**16, dtype=np.bool_)
153: (8)                         assert_(vals.max() < 2)
154: (8)                         assert_(vals.min() >= 0)
155: (4)             def test_repeatability(self):
156: (8)                 import hashlib
157: (8)                 tgt = {'bool':
'509aea74d792fb931784c4b0135392c65aec64beee12b0cc167548a2c3d31e71',
158: (15)                     'int16':
'7b07f1a920e46f6d0fe02314155a2330bcfd7635e708da50e536c5ebb631a7d4',
159: (15)                     'int32':
'e577bfed6c935de944424667e3da285012e741892dc7051a8f1ce68ab05c92f',
160: (15)                     'int64':
'0fbead0b06759df2cfb55e43148822d4a1ff953c7eb19a5b08445a63bb64fa9e',
161: (15)                     'int8':
'001aac3a5acb935a9b186cbe14a1ca064b8bb2dd0b045d48abeacf74d0203404',
162: (15)                     'uint16':
'7b07f1a920e46f6d0fe02314155a2330bcfd7635e708da50e536c5ebb631a7d4',
163: (15)                     'uint32':
'e577bfed6c935de944424667e3da285012e741892dc7051a8f1ce68ab05c92f',
164: (15)                     'uint64':
'0fbead0b06759df2cfb55e43148822d4a1ff953c7eb19a5b08445a63bb64fa9e',
165: (15)                     'uint8':
'001aac3a5acb935a9b186cbe14a1ca064b8bb2dd0b045d48abeacf74d0203404'}
166: (8)             for dt in self.dtype[1:]:
167: (12)                 np.random.seed(1234)
168: (12)                 if sys.byteorder == 'little':
169: (16)                     val = self.rfunc(0, 6, size=1000, dtype=dt)
170: (12)                 else:
171: (16)                     val = self.rfunc(0, 6, size=1000, dtype=dt).byteswap()
172: (12)                 res = hashlib.sha256(val.view(np.int8)).hexdigest()
173: (12)                 assert_(tgt[np.dtype(dt).name] == res)
174: (8)             np.random.seed(1234)
175: (8)             val = self.rfunc(0, 2, size=1000, dtype=bool).view(np.int8)
176: (8)             res = hashlib.sha256(val).hexdigest()
177: (8)             assert_(tgt[np.dtype(bool).name] == res)
178: (4)             def test_int64_uint64_corner_case(self):
179: (8)                 dt = np.int64
180: (8)                 tgt = np.iinfo(np.int64).max
181: (8)                 lbind = np.int64(np.iinfo(np.int64).max)
182: (8)                 ubnd = np.uint64(np.iinfo(np.int64).max + 1)
183: (8)                 actual = np.random.randint(lbind, ubnd, dtype=dt)
184: (8)                 assert_equal(actual, tgt)
185: (4)             def test_respect_dtype_singleton(self):
186: (8)                 for dt in self.dtype:
187: (12)                     lbind = 0 if dt is np.bool_ else np.iinfo(dt).min
188: (12)                     ubnd = 2 if dt is np.bool_ else np.iinfo(dt).max + 1
189: (12)                     sample = self.rfunc(lbind, ubnd, dtype=dt)
190: (12)                     assert_equal(sample.dtype, np.dtype(dt))
191: (8)                 for dt in (bool, int):
192: (12)                     lbind = 0 if dt is bool else np.iinfo(dt).min
193: (12)                     ubnd = 2 if dt is bool else np.iinfo(dt).max + 1
194: (12)                     sample = self.rfunc(lbind, ubnd, dtype=dt)
195: (12)                     assert_(not hasattr(sample, 'dtype'))
196: (12)                     assert_equal(type(sample), dt)
197: (0)             class TestRandomDist:
198: (4)                 def setup_method(self):
199: (8)                     self.seed = 1234567890
200: (4)                 def test_rand(self):
201: (8)                     np.random.seed(self.seed)
202: (8)                     actual = np.random.rand(3, 2)
203: (8)                     desired = np.array([[0.61879477158567997, 0.59162362775974664],
204: (28)                               [0.88868358904449662, 0.89165480011560816],
```

```

205: (28)                                     [0.4575674820298663, 0.7781880808593471]])
206: (8)          assert_array_almost_equal(actual, desired, decimal=15)
207: (4)          def test_rndn(self):
208: (8)              np.random.seed(self.seed)
209: (8)              actual = np.random.rndn(3, 2)
210: (8)              desired = np.array([[1.34016345771863121, 1.73759122771936081],
211: (27)                  [1.498988344300628, -0.2286433324536169],
212: (27)                  [2.031033998682787, 2.17032494605655257]])
213: (8)          assert_array_almost_equal(actual, desired, decimal=15)
214: (4)          def test_randint(self):
215: (8)              np.random.seed(self.seed)
216: (8)              actual = np.random.randint(-99, 99, size=(3, 2))
217: (8)              desired = np.array([[31, 3],
218: (28)                  [-52, 41],
219: (28)                  [-48, -66]])
220: (8)          assert_array_equal(actual, desired)
221: (4)          def test_random_integers(self):
222: (8)              np.random.seed(self.seed)
223: (8)              with suppress_warnings() as sup:
224: (12)                  w = sup.record(DeprecationWarning)
225: (12)                  actual = np.random.random_integers(-99, 99, size=(3, 2))
226: (12)                  assert_(len(w) == 1)
227: (8)              desired = np.array([[31, 3],
228: (28)                  [-52, 41],
229: (28)                  [-48, -66]])
230: (8)          assert_array_equal(actual, desired)
231: (4)          def test_random_integers_max_int(self):
232: (8)              with suppress_warnings() as sup:
233: (12)                  w = sup.record(DeprecationWarning)
234: (12)                  actual = np.random.random_integers(np.iinfo('l').max,
235: (47)                      np.iinfo('l').max)
236: (12)                  assert_(len(w) == 1)
237: (8)              desired = np.iinfo('l').max
238: (8)              assert_equal(actual, desired)
239: (4)          def test_random_integers_DEPRECATED(self):
240: (8)              with warnings.catch_warnings():
241: (12)                  warnings.simplefilter("error", DeprecationWarning)
242: (12)                  assert_raises(DeprecationWarning,
243: (26)                      np.random.random_integers,
244: (26)                      np.iinfo('l').max)
245: (12)                  assert_raises(DeprecationWarning,
246: (26)                      np.random.random_integers,
247: (26)                      np.iinfo('l').max, np.iinfo('l').max)
248: (4)          def test_random(self):
249: (8)              np.random.seed(self.seed)
250: (8)              actual = np.random.random((3, 2))
251: (8)              desired = np.array([[0.61879477158567997, 0.59162362775974664],
252: (28)                  [0.88868358904449662, 0.89165480011560816],
253: (28)                  [0.4575674820298663, 0.7781880808593471]])
254: (8)          assert_array_almost_equal(actual, desired, decimal=15)
255: (4)          def test_choice_uniform_replace(self):
256: (8)              np.random.seed(self.seed)
257: (8)              actual = np.random.choice(4, 4)
258: (8)              desired = np.array([2, 3, 2, 3])
259: (8)              assert_array_equal(actual, desired)
260: (4)          def test_choice_nonuniform_replace(self):
261: (8)              np.random.seed(self.seed)
262: (8)              actual = np.random.choice(4, 4, p=[0.4, 0.4, 0.1, 0.1])
263: (8)              desired = np.array([1, 1, 2, 2])
264: (8)              assert_array_equal(actual, desired)
265: (4)          def test_choice_uniform_noreplace(self):
266: (8)              np.random.seed(self.seed)
267: (8)              actual = np.random.choice(4, 3, replace=False)
268: (8)              desired = np.array([0, 1, 3])
269: (8)              assert_array_equal(actual, desired)
270: (4)          def test_choice_nonuniform_noreplace(self):
271: (8)              np.random.seed(self.seed)
272: (8)              actual = np.random.choice(4, 3, replace=False,
273: (34)                  p=[0.1, 0.3, 0.5, 0.1])

```

```

274: (8)             desired = np.array([2, 3, 1])
275: (8)             assert_array_equal(actual, desired)
276: (4)             def test_choice_noninteger(self):
277: (8)                 np.random.seed(self.seed)
278: (8)                 actual = np.random.choice(['a', 'b', 'c', 'd'], 4)
279: (8)                 desired = np.array(['c', 'd', 'c', 'd'])
280: (8)                 assert_array_equal(actual, desired)
281: (4)             def test_choice_exceptions(self):
282: (8)                 sample = np.random.choice
283: (8)                 assert_raises(ValueError, sample, -1, 3)
284: (8)                 assert_raises(ValueError, sample, 3., 3)
285: (8)                 assert_raises(ValueError, sample, [[1, 2], [3, 4]], 3)
286: (8)                 assert_raises(ValueError, sample, [], 3)
287: (8)                 assert_raises(ValueError, sample, [1, 2, 3, 4], 3,
288: (22)                   p=[[0.25, 0.25], [0.25, 0.25]])
289: (8)                 assert_raises(ValueError, sample, [1, 2], 3, p=[0.4, 0.4, 0.2])
290: (8)                 assert_raises(ValueError, sample, [1, 2], 3, p=[1.1, -0.1])
291: (8)                 assert_raises(ValueError, sample, [1, 2], 3, p=[0.4, 0.4])
292: (8)                 assert_raises(ValueError, sample, [1, 2, 3], 4, replace=False)
293: (8)                 assert_raises(ValueError, sample, [1, 2, 3], -2, replace=False)
294: (8)                 assert_raises(ValueError, sample, [1, 2, 3], (-1,), replace=False)
295: (8)                 assert_raises(ValueError, sample, [1, 2, 3], (-1, 1), replace=False)
296: (8)                 assert_raises(ValueError, sample, [1, 2, 3], 2,
297: (22)                   replace=False, p=[1, 0, 0])
298: (4)             def test_choice_return_shape(self):
299: (8)                 p = [0.1, 0.9]
300: (8)                 assert_(np.isscalar(np.random.choice(2, replace=True)))
301: (8)                 assert_(np.isscalar(np.random.choice(2, replace=False)))
302: (8)                 assert_(np.isscalar(np.random.choice(2, replace=True, p=p)))
303: (8)                 assert_(np.isscalar(np.random.choice(2, replace=False, p=p)))
304: (8)                 assert_(np.isscalar(np.random.choice([1, 2], replace=True)))
305: (8)                 assert_(np.random.choice([None], replace=True) is None)
306: (8)                 a = np.array([1, 2])
307: (8)                 arr = np.empty(1, dtype=object)
308: (8)                 arr[0] = a
309: (8)                 assert_(np.random.choice(arr, replace=True) is a)
310: (8)                 s = tuple()
311: (8)                 assert_(not np.isscalar(np.random.choice(2, s, replace=True)))
312: (8)                 assert_(not np.isscalar(np.random.choice(2, s, replace=False)))
313: (8)                 assert_(not np.isscalar(np.random.choice(2, s, replace=True, p=p)))
314: (8)                 assert_(not np.isscalar(np.random.choice(2, s, replace=False, p=p)))
315: (8)                 assert_(not np.isscalar(np.random.choice([1, 2], s, replace=True)))
316: (8)                 assert_(np.random.choice([None], s, replace=True).ndim == 0)
317: (8)                 a = np.array([1, 2])
318: (8)                 arr = np.empty(1, dtype=object)
319: (8)                 arr[0] = a
320: (8)                 assert_(np.random.choice(arr, s, replace=True).item() is a)
321: (8)                 s = (2, 3)
322: (8)                 p = [0.1, 0.1, 0.1, 0.1, 0.4, 0.2]
323: (8)                 assert_equal(np.random.choice(6, s, replace=True).shape, s)
324: (8)                 assert_equal(np.random.choice(6, s, replace=False).shape, s)
325: (8)                 assert_equal(np.random.choice(6, s, replace=True, p=p).shape, s)
326: (8)                 assert_equal(np.random.choice(6, s, replace=False, p=p).shape, s)
327: (8)                 assert_equal(np.random.choice(np.arange(6), s, replace=True).shape, s)
328: (8)                 assert_equal(np.random.randint(0, 0, size=(3, 0, 4)).shape, (3, 0, 4))
329: (8)                 assert_equal(np.random.randint(0, -10, size=0).shape, (0,))
330: (8)                 assert_equal(np.random.randint(10, 10, size=0).shape, (0,))
331: (8)                 assert_equal(np.random.choice(0, size=0).shape, (0,))
332: (8)                 assert_equal(np.random.choice([], size=(0,)).shape, (0,))
333: (8)                 assert_equal(np.random.choice(['a', 'b'], size=(3, 0, 4)).shape,
334: (21)                   (3, 0, 4))
335: (8)                 assert_raises(ValueError, np.random.choice, [], 10)
336: (4)             def test_choice_nan_probabilities(self):
337: (8)                 a = np.array([42, 1, 2])
338: (8)                 p = [None, None, None]
339: (8)                 assert_raises(ValueError, np.random.choice, a, p=p)
340: (4)             def test_bytes(self):
341: (8)                 np.random.seed(self.seed)
342: (8)                 actual = np.random.bytes(10)

```

```

343: (8)             desired = b'\x82Ui\x9e\xff\x97+Wf\x a5'
344: (8)             assert_equal(actual, desired)
345: (4)             def test_shuffle(self):
346: (8)                 for conv in [lambda x: np.array([]),
347: (21)                     lambda x: x,
348: (21)                     lambda x: np.asarray(x).astype(np.int8),
349: (21)                     lambda x: np.asarray(x).astype(np.float32),
350: (21)                     lambda x: np.asarray(x).astype(np.complex64),
351: (21)                     lambda x: np.asarray(x).astype(object),
352: (21)                     lambda x: [(i, i) for i in x],
353: (21)                     lambda x: np.asarray([[i, i] for i in x]),
354: (21)                     lambda x: np.vstack([x, x]).T,
355: (21)                     lambda x: (np.asarray([(i, i) for i in x],
356: (43)                         [("a", int), ("b", int)])
357: (32)                             .view(np.recarray)),
358: (21)                     lambda x: np.asarray([(i, i) for i in x],
359: (42)                         [("a", object), ("b", np.int32)]):
360: (12)                         np.random.seed(self.seed)
361: (12)                         alist = conv([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
362: (12)                         np.random.shuffle(alist)
363: (12)                         actual = alist
364: (12)                         desired = conv([0, 1, 9, 6, 2, 4, 5, 8, 7, 3])
365: (12)                         assert_array_equal(actual, desired)
366: (4)             def test_shuffle_masked(self):
367: (8)                 a = np.ma.masked_values(np.reshape(range(20), (5, 4)) % 3 - 1, -1)
368: (8)                 b = np.ma.masked_values(np.arange(20) % 3 - 1, -1)
369: (8)                 a_orig = a.copy()
370: (8)                 b_orig = b.copy()
371: (8)                 for i in range(50):
372: (12)                     np.random.shuffle(a)
373: (12)                     assert_equal(
374: (16)                         sorted(a.data[~a.mask]), sorted(a_orig.data[~a_orig.mask]))
375: (12)                     np.random.shuffle(b)
376: (12)                     assert_equal(
377: (16)                         sorted(b.data[~b.mask]), sorted(b_orig.data[~b_orig.mask]))
378: (4)             @pytest.mark.parametrize("random",
379: (12)                 [np.random, np.random.RandomState(), np.random.default_rng()])
380: (4)             def test_shuffle_untyped_warning(self, random):
381: (8)                 values = {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6}
382: (8)                 with pytest.warns(UserWarning,
383: (16)                     match="you are shuffling a 'dict' object") as rec:
384: (12)                     random.shuffle(values)
385: (8)                     assert "test_random" in rec[0].filename
386: (4)             @pytest.mark.parametrize("random",
387: (8)                 [np.random, np.random.RandomState(), np.random.default_rng()])
388: (4)             @pytest.mark.parametrize("use_array_like", [True, False])
389: (4)             def test_shuffle_no_object_unpacking(self, random, use_array_like):
390: (8)                 class MyArr(np.ndarray):
391: (12)                     pass
392: (8)                     items = [
393: (12)                         None, np.array([3]), np.float64(3), np.array(10), np.float64(7)
394: (8)                     ]
395: (8)                     arr = np.array(items, dtype=object)
396: (8)                     item_ids = {id(i) for i in items}
397: (8)                     if use_array_like:
398: (12)                         arr = arr.view(MyArr)
399: (8)                     assert all(id(i) in item_ids for i in arr)
400: (8)                     if use_array_like and not isinstance(random, np.random.Generator):
401: (12)                         with pytest.warns(UserWarning,
402: (20)                             match="Shuffling a one dimensional array.*"):
403: (16)                             random.shuffle(arr)
404: (8)                     else:
405: (12)                         random.shuffle(arr)
406: (12)                         assert all(id(i) in item_ids for i in arr)
407: (4)             def test_shuffle_memoryview(self):
408: (8)                 np.random.seed(self.seed)
409: (8)                 a = np.arange(5).data
410: (8)                 np.random.shuffle(a)
411: (8)                 assert_equal(np.asarray(a), [0, 1, 4, 3, 2])

```

```

412: (8)             rng = np.random.RandomState(self.seed)
413: (8)             rng.shuffle(a)
414: (8)             assert_equal(np.asarray(a), [0, 1, 2, 3, 4])
415: (8)             rng = np.random.default_rng(self.seed)
416: (8)             rng.shuffle(a)
417: (8)             assert_equal(np.asarray(a), [4, 1, 0, 3, 2])
418: (4)             def test_shuffle_not_writeable(self):
419: (8)                 a = np.zeros(3)
420: (8)                 a.flags.writeable = False
421: (8)                 with pytest.raises(ValueError, match='read-only'):
422: (12)                     np.random.shuffle(a)
423: (4)             def test_beta(self):
424: (8)                 np.random.seed(self.seed)
425: (8)                 actual = np.random.beta(.1, .9, size=(3, 2))
426: (8)                 desired = np.array(
427: (16)                     [[1.45341850513746058e-02, 5.31297615662868145e-04],
428: (17)                     [1.85366619058432324e-06, 4.19214516800110563e-03],
429: (17)                     [1.58405155108498093e-04, 1.26252891949397652e-04]])
430: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
431: (4)             def test_binomial(self):
432: (8)                 np.random.seed(self.seed)
433: (8)                 actual = np.random.binomial(100, .456, size=(3, 2))
434: (8)                 desired = np.array([[37, 43],
435: (28)                     [42, 48],
436: (28)                     [46, 45]])
437: (8)                 assert_array_equal(actual, desired)
438: (4)             def test_chisquare(self):
439: (8)                 np.random.seed(self.seed)
440: (8)                 actual = np.random.chisquare(50, size=(3, 2))
441: (8)                 desired = np.array([[63.87858175501090585, 68.68407748911370447],
442: (28)                     [65.77116116901505904, 47.09686762438974483],
443: (28)                     [72.3828403199695174, 74.18408615260374006]])
444: (8)                 assert_array_almost_equal(actual, desired, decimal=13)
445: (4)             def test_dirichlet(self):
446: (8)                 np.random.seed(self.seed)
447: (8)                 alpha = np.array([51.72840233779265162, 39.74494232180943953])
448: (8)                 actual = np.random.mtrand.dirichlet(alpha, size=(3, 2))
449: (8)                 desired = np.array([[0.54539444573611562, 0.45460555426388438],
450: (29)                     [0.62345816822039413, 0.37654183177960598]],
451: (28)                     [[0.55206000085785778, 0.44793999914214233],
452: (29)                     [0.58964023305154301, 0.41035976694845688]],
453: (28)                     [[0.59266909280647828, 0.40733090719352177],
454: (29)                     [0.56974431743975207, 0.43025568256024799]]])
455: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
456: (4)             def test_dirichlet_size(self):
457: (8)                 p = np.array([51.72840233779265162, 39.74494232180943953])
458: (8)                 assert_equal(np.random.dirichlet(p, np.uint32(1)).shape, (1, 2))
459: (8)                 assert_equal(np.random.dirichlet(p, np.uint32(1)).shape, (1, 2))
460: (8)                 assert_equal(np.random.dirichlet(p, np.uint32(1)).shape, (1, 2))
461: (8)                 assert_equal(np.random.dirichlet(p, [2, 2]).shape, (2, 2, 2))
462: (8)                 assert_equal(np.random.dirichlet(p, (2, 2)).shape, (2, 2, 2))
463: (8)                 assert_equal(np.random.dirichlet(p, np.array((2, 2))).shape, (2, 2,
2))
464: (8)                 assert_raises(TypeError, np.random.dirichlet, p, float(1))
465: (4)             def test_dirichlet_bad_alpha(self):
466: (8)                 alpha = np.array([5.4e-01, -1.0e-16])
467: (8)                 assert_raises(ValueError, np.random.mtrand.dirichlet, alpha)
468: (8)                 assert_raises(ValueError, random.dirichlet, [[5, 1]])
469: (8)                 assert_raises(ValueError, random.dirichlet, [[5], [1]])
470: (8)                 assert_raises(ValueError, random.dirichlet, [[[5], [1]], [[1], [5]]])
471: (8)                 assert_raises(ValueError, random.dirichlet, np.array([[5, 1], [1,
5]])))
472: (4)             def test_exponential(self):
473: (8)                 np.random.seed(self.seed)
474: (8)                 actual = np.random.exponential(1.1234, size=(3, 2))
475: (8)                 desired = np.array([[1.08342649775011624, 1.00607889924557314],
476: (28)                     [2.46628830085216721, 2.49668106809923884],
477: (28)                     [0.68717433461363442, 1.69175666993575979]])
478: (8)                 assert_array_almost_equal(actual, desired, decimal=15)

```

```

479: (4) def test_exponential_0(self):
480: (8)     assert_equal(np.random.exponential(scale=0), 0)
481: (8)     assert_raises(ValueError, np.random.exponential, scale=-0.)
482: (4) def test_f(self):
483: (8)     np.random.seed(self.seed)
484: (8)     actual = np.random.f(12, 77, size=(3, 2))
485: (8)     desired = np.array([[1.21975394418575878, 1.75135759791559775],
486: (28)             [1.44803115017146489, 1.22108959480396262],
487: (28)             [1.02176975757740629, 1.34431827623300415]])
488: (8)     assert_array_almost_equal(actual, desired, decimal=15)
489: (4) def test_gamma(self):
490: (8)     np.random.seed(self.seed)
491: (8)     actual = np.random.gamma(5, 3, size=(3, 2))
492: (8)     desired = np.array([[24.60509188649287182, 28.54993563207210627],
493: (28)             [26.13476110204064184, 12.56988482927716078],
494: (28)             [31.71863275789960568, 33.30143302795922011]])
495: (8)     assert_array_almost_equal(actual, desired, decimal=14)
496: (4) def test_gamma_0(self):
497: (8)     assert_equal(np.random.gamma(shape=0, scale=0), 0)
498: (8)     assert_raises(ValueError, np.random.gamma, shape=-0., scale=-0.)
499: (4) def test_geometric(self):
500: (8)     np.random.seed(self.seed)
501: (8)     actual = np.random.geometric(.123456789, size=(3, 2))
502: (8)     desired = np.array([[8, 7],
503: (28)         [17, 17],
504: (28)         [5, 12]])
505: (8)     assert_array_equal(actual, desired)
506: (4) def test_gumbel(self):
507: (8)     np.random.seed(self.seed)
508: (8)     actual = np.random.gumbel(loc=.123456789, scale=2.0, size=(3, 2))
509: (8)     desired = np.array([[0.19591898743416816, 0.34405539668096674],
510: (28)         [-1.4492522252274278, -1.47374816298446865],
511: (28)         [1.10651090478803416, -0.69535848626236174]])
512: (8)     assert_array_almost_equal(actual, desired, decimal=15)
513: (4) def test_gumbel_0(self):
514: (8)     assert_equal(np.random.gumbel(scale=0), 0)
515: (8)     assert_raises(ValueError, np.random.gumbel, scale=-0.)
516: (4) def test_hypergeometric(self):
517: (8)     np.random.seed(self.seed)
518: (8)     actual = np.random.hypergeometric(10, 5, 14, size=(3, 2))
519: (8)     desired = np.array([[10, 10],
520: (28)         [10, 10],
521: (28)         [9, 9]])
522: (8)     assert_array_equal(actual, desired)
523: (8)     actual = np.random.hypergeometric(5, 0, 3, size=4)
524: (8)     desired = np.array([3, 3, 3, 3])
525: (8)     assert_array_equal(actual, desired)
526: (8)     actual = np.random.hypergeometric(15, 0, 12, size=4)
527: (8)     desired = np.array([12, 12, 12, 12])
528: (8)     assert_array_equal(actual, desired)
529: (8)     actual = np.random.hypergeometric(0, 5, 3, size=4)
530: (8)     desired = np.array([0, 0, 0, 0])
531: (8)     assert_array_equal(actual, desired)
532: (8)     actual = np.random.hypergeometric(0, 15, 12, size=4)
533: (8)     desired = np.array([0, 0, 0, 0])
534: (8)     assert_array_equal(actual, desired)
535: (4) def test_laplace(self):
536: (8)     np.random.seed(self.seed)
537: (8)     actual = np.random.laplace(loc=.123456789, scale=2.0, size=(3, 2))
538: (8)     desired = np.array([[0.66599721112760157, 0.52829452552221945],
539: (28)         [3.12791959514407125, 3.18202813572992005],
540: (28)         [-0.05391065675859356, 1.74901336242837324]])
541: (8)     assert_array_almost_equal(actual, desired, decimal=15)
542: (4) def test_laplace_0(self):
543: (8)     assert_equal(np.random.laplace(scale=0), 0)
544: (8)     assert_raises(ValueError, np.random.laplace, scale=-0.)
545: (4) def test_logistic(self):
546: (8)     np.random.seed(self.seed)
547: (8)     actual = np.random.logistic(loc=.123456789, scale=2.0, size=(3, 2))

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

548: (8)             desired = np.array([[1.09232835305011444, 0.8648196662399954],
549: (28)           [4.27818590694950185, 4.33897006346929714],
550: (28)           [-0.21682183359214885, 2.63373365386060332]])
551: (8)             assert_array_almost_equal(actual, desired, decimal=15)
552: (4)             def test_lognormal(self):
553: (8)               np.random.seed(self.seed)
554: (8)               actual = np.random.lognormal(mean=.123456789, sigma=2.0, size=(3, 2))
555: (8)               desired = np.array([[16.50698631688883822, 36.54846706092654784],
556: (28)             [22.67886599981281748, 0.71617561058995771],
557: (28)             [65.72798501792723869, 86.84341601437161273]])
558: (8)             assert_array_almost_equal(actual, desired, decimal=13)
559: (4)             def test_lognormal_0(self):
560: (8)               assert_equal(np.random.lognormal(sigma=0), 1)
561: (8)               assert_raises(ValueError, np.random.lognormal, sigma=-0.)
562: (4)             def test_logseries(self):
563: (8)               np.random.seed(self.seed)
564: (8)               actual = np.random.logseries(p=.923456789, size=(3, 2))
565: (8)               desired = np.array([[2, 2],
566: (28)             [6, 17],
567: (28)             [3, 6]])
568: (8)               assert_array_equal(actual, desired)
569: (4)             def test_multinomial(self):
570: (8)               np.random.seed(self.seed)
571: (8)               actual = np.random.multinomial(20, [1/6.]**6, size=(3, 2))
572: (8)               desired = np.array([[4, 3, 5, 4, 2, 2],
573: (29)             [5, 2, 8, 2, 2, 1]],
574: (28)             [[3, 4, 3, 6, 0, 4],
575: (29)             [2, 1, 4, 3, 6, 4]],
576: (28)             [[4, 4, 2, 5, 2, 3],
577: (29)             [4, 3, 4, 2, 3, 4]]])
578: (8)               assert_array_equal(actual, desired)
579: (4)             def test_multivariate_normal(self):
580: (8)               np.random.seed(self.seed)
581: (8)               mean = (.123456789, 10)
582: (8)               cov = [[1, 0], [0, 1]]
583: (8)               size = (3, 2)
584: (8)               actual = np.random.multivariate_normal(mean, cov, size)
585: (8)               desired = np.array([[1.463620246718631, 11.73759122771936],
586: (29)             [1.622445133300628, 9.771356667546383]],
587: (28)             [[2.154490787682787, 12.170324946056553],
588: (29)             [1.719909438201865, 9.230548443648306]],
589: (28)             [[0.689515026297799, 9.880729819607714],
590: (29)             [-0.023054015651998, 9.201096623542879]])
591: (8)               assert_array_almost_equal(actual, desired, decimal=15)
592: (8)               actual = np.random.multivariate_normal(mean, cov)
593: (8)               desired = np.array([0.895289569463708, 9.17180864067987])
594: (8)               assert_array_almost_equal(actual, desired, decimal=15)
595: (8)               mean = [0, 0]
596: (8)               cov = [[1, 2], [2, 1]]
597: (8)               assert_warns(RuntimeWarning, np.random.multivariate_normal, mean, cov)
598: (8)               assert_no_warnings(np.random.multivariate_normal, mean, cov,
599: (27)                 check_valid='ignore')
600: (8)               assert_raises(ValueError, np.random.multivariate_normal, mean, cov,
601: (22)                 check_valid='raise')
602: (8)               cov = np.array([[1, 0.1], [0.1, 1]], dtype=np.float32)
603: (8)               with suppress_warnings() as sup:
604: (12)                 np.random.multivariate_normal(mean, cov)
605: (12)                 w = sup.record(RuntimeWarning)
606: (12)                 assert len(w) == 0
607: (4)             def test_negative_binomial(self):
608: (8)               np.random.seed(self.seed)
609: (8)               actual = np.random.negative_binomial(n=100, p=.12345, size=(3, 2))
610: (8)               desired = np.array([[848, 841],
611: (28)             [892, 611],
612: (28)             [779, 647]])
613: (8)               assert_array_equal(actual, desired)
614: (4)             def test_noncentral_chisquare(self):
615: (8)               np.random.seed(self.seed)
616: (8)               actual = np.random.noncentral_chisquare(df=5, nonc=5, size=(3, 2))

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

617: (8)             desired = np.array([[23.91905354498517511, 13.35324692733826346],
618: (28)           [31.22452661329736401, 16.60047399466177254],
619: (28)           [5.03461598262724586, 17.94973089023519464]])
620: (8)             assert_array_almost_equal(actual, desired, decimal=14)
621: (8)             actual = np.random.noncentral_chisquare(df=.5, nonc=.2, size=(3, 2))
622: (8)             desired = np.array([[1.47145377828516666, 0.15052899268012659],
623: (28)           [0.00943803056963588, 1.02647251615666169],
624: (28)           [0.332334982684171, 0.15451287602753125]])
625: (8)             assert_array_almost_equal(actual, desired, decimal=14)
626: (8)             np.random.seed(self.seed)
627: (8)             actual = np.random.noncentral_chisquare(df=5, nonc=0, size=(3, 2))
628: (8)             desired = np.array([[9.597154162763948, 11.725484450296079],
629: (28)           [10.413711048138335, 3.694475922923986],
630: (28)           [13.484222138963087, 14.377255424602957]])
631: (8)             assert_array_almost_equal(actual, desired, decimal=14)
632: (4)             def test_noncentral_f(self):
633: (8)               np.random.seed(self.seed)
634: (8)               actual = np.random.noncentral_f(dfnum=5, dfden=2, nonc=1,
635: (40)                 size=(3, 2))
636: (8)               desired = np.array([[1.40598099674926669, 0.34207973179285761],
637: (28)             [3.57715069265772545, 7.92632662577829805],
638: (28)             [0.43741599463544162, 1.1774208752428319]])
639: (8)             assert_array_almost_equal(actual, desired, decimal=14)
640: (4)             def test_normal(self):
641: (8)               np.random.seed(self.seed)
642: (8)               actual = np.random.normal(loc=.123456789, scale=2.0, size=(3, 2))
643: (8)               desired = np.array([[2.80378370443726244, 3.59863924443872163],
644: (28)             [3.121433477601256, -0.33382987590723379],
645: (28)             [4.18552478636557357, 4.46410668111310471]])
646: (8)             assert_array_almost_equal(actual, desired, decimal=15)
647: (4)             def test_normal_0(self):
648: (8)               assert_equal(np.random.normal(scale=0), 0)
649: (8)               assert_raises(ValueError, np.random.normal, scale=-0.)
650: (4)             def test_pareto(self):
651: (8)               np.random.seed(self.seed)
652: (8)               actual = np.random.pareto(a=.123456789, size=(3, 2))
653: (8)               desired = np.array(
654: (16)                 [[2.46852460439034849e+03, 1.41286880810518346e+03],
655: (17)                   [5.28287797029485181e+07, 6.57720981047328785e+07],
656: (17)                   [1.40840323350391515e+02, 1.98390255135251704e+05]])
657: (8)               np.testing.assert_array_almost_equal_nulp(actual, desired, nulp=30)
658: (4)             def test_poisson(self):
659: (8)               np.random.seed(self.seed)
660: (8)               actual = np.random.poisson(lam=.123456789, size=(3, 2))
661: (8)               desired = np.array([[0, 0],
662: (28)                 [1, 0],
663: (28)                 [0, 0]])
664: (8)               assert_array_equal(actual, desired)
665: (4)             def test_poisson_exceptions(self):
666: (8)               lambig = np.iinfo('l').max
667: (8)               lamneg = -1
668: (8)               assert_raises(ValueError, np.random.poisson, lamneg)
669: (8)               assert_raises(ValueError, np.random.poisson, [lamneg]*10)
670: (8)               assert_raises(ValueError, np.random.poisson, lambig)
671: (8)               assert_raises(ValueError, np.random.poisson, [lambig]*10)
672: (4)             def test_power(self):
673: (8)               np.random.seed(self.seed)
674: (8)               actual = np.random.power(a=.123456789, size=(3, 2))
675: (8)               desired = np.array([[0.02048932883240791, 0.01424192241128213],
676: (28)                 [0.38446073748535298, 0.39499689943484395],
677: (28)                 [0.00177699707563439, 0.13115505880863756]])
678: (8)               assert_array_almost_equal(actual, desired, decimal=15)
679: (4)             def test_rayleigh(self):
680: (8)               np.random.seed(self.seed)
681: (8)               actual = np.random.rayleigh(scale=10, size=(3, 2))
682: (8)               desired = np.array([[13.8882496494248393, 13.383318339044731],
683: (28)                 [20.95413364294492098, 21.08285015800712614],
684: (28)                 [11.06066537006854311, 17.35468505778271009]])
685: (8)               assert_array_almost_equal(actual, desired, decimal=14)

```

```

686: (4) def test_rayleigh_0(self):
687: (8)     assert_equal(np.random.rayleigh(scale=0), 0)
688: (8)     assert_raises(ValueError, np.random.rayleigh, scale=-0.)
689: (4) def test_standard_cauchy(self):
690: (8)     np.random.seed(self.seed)
691: (8)     actual = np.random.standard_cauchy(size=(3, 2))
692: (28)    desired = np.array([[0.77127660196445336, -6.55601161955910605],
693: (28)                  [0.93582023391158309, -2.07479293013759447],
694: (28)                  [-4.74601644297011926, 0.18338989290760804]])
695: (8)     assert_array_almost_equal(actual, desired, decimal=15)
696: (4) def test_standard_exponential(self):
697: (8)     np.random.seed(self.seed)
698: (8)     actual = np.random.standard_exponential(size=(3, 2))
699: (8)     desired = np.array([[0.96441739162374596, 0.89556604882105506],
700: (28)                  [2.1953785836319808, 2.22243285392490542],
701: (28)                  [0.6116915921431676, 1.50592546727413201]])
702: (8)     assert_array_almost_equal(actual, desired, decimal=15)
703: (4) def test_standard_gamma(self):
704: (8)     np.random.seed(self.seed)
705: (8)     actual = np.random.standard_gamma(shape=3, size=(3, 2))
706: (8)     desired = np.array([[5.50841531318455058, 6.62953470301903103],
707: (28)                  [5.93988484943779227, 2.31044849402133989],
708: (28)                  [7.54838614231317084, 8.012756093271868]])
709: (8)     assert_array_almost_equal(actual, desired, decimal=14)
710: (4) def test_standard_gamma_0(self):
711: (8)     assert_equal(np.random.standard_gamma(shape=0), 0)
712: (8)     assert_raises(ValueError, np.random.standard_gamma, shape=-0.)
713: (4) def test_standard_normal(self):
714: (8)     np.random.seed(self.seed)
715: (8)     actual = np.random.standard_normal(size=(3, 2))
716: (8)     desired = np.array([[1.34016345771863121, 1.73759122771936081],
717: (28)                  [1.498988344300628, -0.2286433324536169],
718: (28)                  [2.031033998682787, 2.17032494605655257]])
719: (8)     assert_array_almost_equal(actual, desired, decimal=15)
720: (4) def test_standard_t(self):
721: (8)     np.random.seed(self.seed)
722: (8)     actual = np.random.standard_t(df=10, size=(3, 2))
723: (8)     desired = np.array([[0.97140611862659965, -0.08830486548450577],
724: (28)                  [1.36311143689505321, -0.55317463909867071],
725: (28)                  [-0.18473749069684214, 0.61181537341755321]])
726: (8)     assert_array_almost_equal(actual, desired, decimal=15)
727: (4) def test_triangular(self):
728: (8)     np.random.seed(self.seed)
729: (8)     actual = np.random.triangular(left=5.12, mode=10.23, right=20.34,
730: (38)                  size=(3, 2))
731: (8)     desired = np.array([[12.68117178949215784, 12.4129206149193152],
732: (28)                  [16.20131377335158263, 16.25692138747600524],
733: (28)                  [11.20400690911820263, 14.4978144835829923]])
734: (8)     assert_array_almost_equal(actual, desired, decimal=14)
735: (4) def test_uniform(self):
736: (8)     np.random.seed(self.seed)
737: (8)     actual = np.random.uniform(low=1.23, high=10.54, size=(3, 2))
738: (8)     desired = np.array([[6.99097932346268003, 6.73801597444323974],
739: (28)                  [9.50364421400426274, 9.53130618907631089],
740: (28)                  [5.48995325769805476, 8.47493103280052118]])
741: (8)     assert_array_almost_equal(actual, desired, decimal=15)
742: (4) def test_uniform_range_bounds(self):
743: (8)     fmin = np.finfo('float').min
744: (8)     fmax = np.finfo('float').max
745: (8)     func = np.random.uniform
746: (8)     assert_raises(OverflowError, func, -np.inf, 0)
747: (8)     assert_raises(OverflowError, func, 0, np.inf)
748: (8)     assert_raises(OverflowError, func, fmin, fmax)
749: (8)     assert_raises(OverflowError, func, [-np.inf], [0])
750: (8)     assert_raises(OverflowError, func, [0], [np.inf])
751: (8)     np.random.uniform(low=np.nextafter(fmin, 1), high=fmax / 1e17)
752: (4) def test_scalar_exception_propagation(self):
753: (8)     class ThrowingFloat(np.ndarray):
754: (12)         def __float__(self):

```

```

755: (16)                     raise TypeError
756: (8)                      throwing_float = np.array(1.0).view(ThrowingFloat)
757: (8)                      assert_raises(TypeError, np.random.uniform, throwing_float,
758: (22)                         throwing_float)
759: (8)                      class ThrowingInteger(np.ndarray):
760: (12)                        def __int__(self):
761: (16)                          raise TypeError
762: (12)                          __index__ = __int__
763: (8)                      throwing_int = np.array(1).view(ThrowingInteger)
764: (8)                      assert_raises(TypeError, np.random.hypergeometric, throwing_int, 1, 1)
765: (4)                      def test_vonmises(self):
766: (8)                        np.random.seed(self.seed)
767: (8)                        actual = np.random.vonmises(mu=1.23, kappa=1.54, size=(3, 2))
768: (8)                        desired = np.array([[2.28567572673902042, 2.89163838442285037],
769: (28)                           [0.38198375564286025, 2.57638023113890746],
770: (28)                           [1.19153771588353052, 1.83509849681825354]])
771: (8)                        assert_array_almost_equal(actual, desired, decimal=15)
772: (4)                      def test_vonmises_small(self):
773: (8)                        np.random.seed(self.seed)
774: (8)                        r = np.random.vonmises(mu=0., kappa=1.1e-8, size=10**6)
775: (8)                        np.testing.assert_(np.isfinite(r).all())
776: (4)                      def test_wald(self):
777: (8)                        np.random.seed(self.seed)
778: (8)                        actual = np.random.wald(mean=1.23, scale=1.54, size=(3, 2))
779: (8)                        desired = np.array([[3.82935265715889983, 5.13125249184285526],
780: (28)                           [0.35045403618358717, 1.50832396872003538],
781: (28)                           [0.24124319895843183, 0.22031101461955038]])
782: (8)                        assert_array_almost_equal(actual, desired, decimal=14)
783: (4)                      def test_weibull(self):
784: (8)                        np.random.seed(self.seed)
785: (8)                        actual = np.random.weibull(a=1.23, size=(3, 2))
786: (8)                        desired = np.array([[0.97097342648766727, 0.91422896443565516],
787: (28)                           [1.89517770034962929, 1.91414357960479564],
788: (28)                           [0.67057783752390987, 1.39494046635066793]])
789: (8)                        assert_array_almost_equal(actual, desired, decimal=15)
790: (4)                      def test_weibull_0(self):
791: (8)                        np.random.seed(self.seed)
792: (8)                        assert_equal(np.random.weibull(a=0, size=12), np.zeros(12))
793: (8)                        assert_raises(ValueError, np.random.weibull, a=-0.)
794: (4)                      def test_zipf(self):
795: (8)                        np.random.seed(self.seed)
796: (8)                        actual = np.random.zipf(a=1.23, size=(3, 2))
797: (8)                        desired = np.array([[66, 29],
798: (28)                           [1, 1],
799: (28)                           [3, 13]])
800: (8)                        assert_array_equal(actual, desired)
801: (0)                      class TestBroadcast:
802: (4)                        def setup_method(self):
803: (8)                          self.seed = 123456789
804: (4)                        def setSeed(self):
805: (8)                          np.random.seed(self.seed)
806: (4)                        def test_uniform(self):
807: (8)                          low = [0]
808: (8)                          high = [1]
809: (8)                          uniform = np.random.uniform
810: (8)                          desired = np.array([0.53283302478975902,
811: (28)                            0.53413660089041659,
812: (28)                            0.50955303552646702])
813: (8)                          self.setSeed()
814: (8)                          actual = uniform(low * 3, high)
815: (8)                          assert_array_almost_equal(actual, desired, decimal=14)
816: (8)                          self.setSeed()
817: (8)                          actual = uniform(low, high * 3)
818: (8)                          assert_array_almost_equal(actual, desired, decimal=14)
819: (4)                      def test_normal(self):
820: (8)                        loc = [0]
821: (8)                        scale = [1]
822: (8)                        bad_scale = [-1]
823: (8)                        normal = np.random.normal

```

```

824: (8)             desired = np.array([2.2129019979039612,
825: (28)           2.1283977976520019,
826: (28)           1.8417114045748335])
827: (8)             self.setSeed()
828: (8)             actual = normal(loc * 3, scale)
829: (8)             assert_array_almost_equal(actual, desired, decimal=14)
830: (8)             assert_raises(ValueError, normal, loc * 3, bad_scale)
831: (8)             self.setSeed()
832: (8)             actual = normal(loc, scale * 3)
833: (8)             assert_array_almost_equal(actual, desired, decimal=14)
834: (8)             assert_raises(ValueError, normal, loc, bad_scale * 3)
835: (4)             def test_beta(self):
836: (8)               a = [1]
837: (8)               b = [2]
838: (8)               bad_a = [-1]
839: (8)               bad_b = [-2]
840: (8)               beta = np.random.beta
841: (8)               desired = np.array([0.19843558305989056,
842: (28)                 0.075230336409423643,
843: (28)                 0.24976865978980844])
844: (8)               self.setSeed()
845: (8)               actual = beta(a * 3, b)
846: (8)               assert_array_almost_equal(actual, desired, decimal=14)
847: (8)               assert_raises(ValueError, beta, bad_a * 3, b)
848: (8)               assert_raises(ValueError, beta, a * 3, bad_b)
849: (8)               self.setSeed()
850: (8)               actual = beta(a, b * 3)
851: (8)               assert_array_almost_equal(actual, desired, decimal=14)
852: (8)               assert_raises(ValueError, beta, bad_a, b * 3)
853: (8)               assert_raises(ValueError, beta, a, bad_b * 3)
854: (4)             def test_exponential(self):
855: (8)               scale = [1]
856: (8)               bad_scale = [-1]
857: (8)               exponential = np.random.exponential
858: (8)               desired = np.array([0.76106853658845242,
859: (28)                 0.76386282278691653,
860: (28)                 0.71243813125891797])
861: (8)               self.setSeed()
862: (8)               actual = exponential(scale * 3)
863: (8)               assert_array_almost_equal(actual, desired, decimal=14)
864: (8)               assert_raises(ValueError, exponential, bad_scale * 3)
865: (4)             def test_standard_gamma(self):
866: (8)               shape = [1]
867: (8)               bad_shape = [-1]
868: (8)               std_gamma = np.random.standard_gamma
869: (8)               desired = np.array([0.76106853658845242,
870: (28)                 0.76386282278691653,
871: (28)                 0.71243813125891797])
872: (8)               self.setSeed()
873: (8)               actual = std_gamma(shape * 3)
874: (8)               assert_array_almost_equal(actual, desired, decimal=14)
875: (8)               assert_raises(ValueError, std_gamma, bad_shape * 3)
876: (4)             def test_gamma(self):
877: (8)               shape = [1]
878: (8)               scale = [2]
879: (8)               bad_shape = [-1]
880: (8)               bad_scale = [-2]
881: (8)               gamma = np.random.gamma
882: (8)               desired = np.array([1.5221370731769048,
883: (28)                 1.5277256455738331,
884: (28)                 1.4248762625178359])
885: (8)               self.setSeed()
886: (8)               actual = gamma(shape * 3, scale)
887: (8)               assert_array_almost_equal(actual, desired, decimal=14)
888: (8)               assert_raises(ValueError, gamma, bad_shape * 3, scale)
889: (8)               assert_raises(ValueError, gamma, shape * 3, bad_scale)
890: (8)               self.setSeed()
891: (8)               actual = gamma(shape, scale * 3)
892: (8)               assert_array_almost_equal(actual, desired, decimal=14)

```

```

893: (8) assert_raises(ValueError, gamma, bad_shape, scale * 3)
894: (8) assert_raises(ValueError, gamma, shape, bad_scale * 3)
895: (4) def test_f(self):
896: (8)     dfnum = [1]
897: (8)     dfden = [2]
898: (8)     bad_dfnum = [-1]
899: (8)     bad_dfden = [-2]
900: (8)     f = np.random.f
901: (8)     desired = np.array([0.80038951638264799,
902: (28)                     0.86768719635363512,
903: (28)                     2.7251095168386801])
904: (8)     self.setSeed()
905: (8)     actual = f(dfnum * 3, dfden)
906: (8)     assert_array_almost_equal(actual, desired, decimal=14)
907: (8)     assert_raises(ValueError, f, bad_dfnum * 3, dfden)
908: (8)     assert_raises(ValueError, f, dfnum * 3, bad_dfden)
909: (8)     self.setSeed()
910: (8)     actual = f(dfnum, dfden * 3)
911: (8)     assert_array_almost_equal(actual, desired, decimal=14)
912: (8)     assert_raises(ValueError, f, bad_dfnum, dfden * 3)
913: (8)     assert_raises(ValueError, f, dfnum, bad_dfden * 3)
914: (4) def test_noncentral_f(self):
915: (8)     dfnum = [2]
916: (8)     dfden = [3]
917: (8)     nonc = [4]
918: (8)     bad_dfnum = [0]
919: (8)     bad_dfden = [-1]
920: (8)     bad_nonc = [-2]
921: (8)     nonc_f = np.random.noncentral_f
922: (8)     desired = np.array([9.1393943263705211,
923: (28)                     13.025456344595602,
924: (28)                     8.8018098359100545])
925: (8)     self.setSeed()
926: (8)     actual = nonc_f(dfnum * 3, dfden, nonc)
927: (8)     assert_array_almost_equal(actual, desired, decimal=14)
928: (8)     assert_raises(ValueError, nonc_f, bad_dfnum * 3, dfden, nonc)
929: (8)     assert_raises(ValueError, nonc_f, dfnum * 3, bad_dfden, nonc)
930: (8)     assert_raises(ValueError, nonc_f, dfnum * 3, dfden, bad_nonc)
931: (8)     self.setSeed()
932: (8)     actual = nonc_f(dfnum, dfden * 3, nonc)
933: (8)     assert_array_almost_equal(actual, desired, decimal=14)
934: (8)     assert_raises(ValueError, nonc_f, bad_dfnum, dfden * 3, nonc)
935: (8)     assert_raises(ValueError, nonc_f, dfnum, bad_dfden * 3, nonc)
936: (8)     assert_raises(ValueError, nonc_f, dfnum, dfden * 3, bad_nonc)
937: (8)     self.setSeed()
938: (8)     actual = nonc_f(dfnum, dfden, nonc * 3)
939: (8)     assert_array_almost_equal(actual, desired, decimal=14)
940: (8)     assert_raises(ValueError, nonc_f, bad_dfnum, dfden, nonc * 3)
941: (8)     assert_raises(ValueError, nonc_f, dfnum, bad_dfden, nonc * 3)
942: (8)     assert_raises(ValueError, nonc_f, dfnum, dfden, bad_nonc * 3)
943: (4) def test_noncentral_f_small_df(self):
944: (8)     self.setSeed()
945: (8)     desired = np.array([6.869638627492048, 0.785880199263955])
946: (8)     actual = np.random.noncentral_f(0.9, 0.9, 2, size=2)
947: (8)     assert_array_almost_equal(actual, desired, decimal=14)
948: (4) def test_chisquare(self):
949: (8)     df = [1]
950: (8)     bad_df = [-1]
951: (8)     chisquare = np.random.chisquare
952: (8)     desired = np.array([0.57022801133088286,
953: (28)                     0.51947702108840776,
954: (28)                     0.1320969254923558])
955: (8)     self.setSeed()
956: (8)     actual = chisquare(df * 3)
957: (8)     assert_array_almost_equal(actual, desired, decimal=14)
958: (8)     assert_raises(ValueError, chisquare, bad_df * 3)
959: (4) def test_noncentral_chisquare(self):
960: (8)     df = [1]
961: (8)     nonc = [2]

```

```

962: (8)             bad_df = [-1]
963: (8)             bad_nonc = [-2]
964: (8)             nonc_chi = np.random.noncentral_chisquare
965: (8)             desired = np.array([9.0015599467913763,
966: (28)                 4.5804135049718742,
967: (28)                 6.0872302432834564])
968: (8)             self.setSeed()
969: (8)             actual = nonc_chi(df * 3, nonc)
970: (8)             assert_array_almost_equal(actual, desired, decimal=14)
971: (8)             assert_raises(ValueError, nonc_chi, bad_df * 3, nonc)
972: (8)             assert_raises(ValueError, nonc_chi, df * 3, bad_nonc)
973: (8)             self.setSeed()
974: (8)             actual = nonc_chi(df, nonc * 3)
975: (8)             assert_array_almost_equal(actual, desired, decimal=14)
976: (8)             assert_raises(ValueError, nonc_chi, bad_df, nonc * 3)
977: (8)             assert_raises(ValueError, nonc_chi, df, bad_nonc * 3)
978: (4)              def test_standard_t(self):
979: (8)                  df = [1]
980: (8)                  bad_df = [-1]
981: (8)                  t = np.random.standard_t
982: (8)                  desired = np.array([3.0702872575217643,
983: (28)                      5.8560725167361607,
984: (28)                      1.0274791436474273])
985: (8)                  self.setSeed()
986: (8)                  actual = t(df * 3)
987: (8)                  assert_array_almost_equal(actual, desired, decimal=14)
988: (8)                  assert_raises(ValueError, t, bad_df * 3)
989: (4)              def test_vonmises(self):
990: (8)                  mu = [2]
991: (8)                  kappa = [1]
992: (8)                  bad_kappa = [-1]
993: (8)                  vonmises = np.random.vonmises
994: (8)                  desired = np.array([2.9883443664201312,
995: (28)                      -2.7064099483995943,
996: (28)                      -1.8672476700665914])
997: (8)                  self.setSeed()
998: (8)                  actual = vonmises(mu * 3, kappa)
999: (8)                  assert_array_almost_equal(actual, desired, decimal=14)
1000: (8)                 assert_raises(ValueError, vonmises, mu * 3, bad_kappa)
1001: (8)                 self.setSeed()
1002: (8)                 actual = vonmises(mu, kappa * 3)
1003: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1004: (8)                 assert_raises(ValueError, vonmises, mu, bad_kappa * 3)
1005: (4)              def test_pareto(self):
1006: (8)                  a = [1]
1007: (8)                  bad_a = [-1]
1008: (8)                  pareto = np.random.pareto
1009: (8)                  desired = np.array([1.1405622680198362,
1010: (28)                      1.1465519762044529,
1011: (28)                      1.0389564467453547])
1012: (8)                  self.setSeed()
1013: (8)                  actual = pareto(a * 3)
1014: (8)                  assert_array_almost_equal(actual, desired, decimal=14)
1015: (8)                  assert_raises(ValueError, pareto, bad_a * 3)
1016: (4)              def test_weibull(self):
1017: (8)                  a = [1]
1018: (8)                  bad_a = [-1]
1019: (8)                  weibull = np.random.weibull
1020: (8)                  desired = np.array([0.76106853658845242,
1021: (28)                      0.76386282278691653,
1022: (28)                      0.71243813125891797])
1023: (8)                  self.setSeed()
1024: (8)                  actual = weibull(a * 3)
1025: (8)                  assert_array_almost_equal(actual, desired, decimal=14)
1026: (8)                  assert_raises(ValueError, weibull, bad_a * 3)
1027: (4)              def test_power(self):
1028: (8)                  a = [1]
1029: (8)                  bad_a = [-1]
1030: (8)                  power = np.random.power

```

```

1031: (8)             desired = np.array([0.53283302478975902,
1032: (28)             0.53413660089041659,
1033: (28)             0.50955303552646702])
1034: (8)             self.setSeed()
1035: (8)             actual = power(a * 3)
1036: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1037: (8)             assert_raises(ValueError, power, bad_a * 3)
1038: (4)             def test_laplace(self):
1039: (8)                 loc = [0]
1040: (8)                 scale = [1]
1041: (8)                 bad_scale = [-1]
1042: (8)                 laplace = np.random.laplace
1043: (8)                 desired = np.array([0.067921356028507157,
1044: (28)                   0.070715642226971326,
1045: (28)                   0.019290950698972624])
1046: (8)             self.setSeed()
1047: (8)             actual = laplace(loc * 3, scale)
1048: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1049: (8)             assert_raises(ValueError, laplace, loc * 3, bad_scale)
1050: (8)             self.setSeed()
1051: (8)             actual = laplace(loc, scale * 3)
1052: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1053: (8)             assert_raises(ValueError, laplace, loc, bad_scale * 3)
1054: (4)             def test_gumbel(self):
1055: (8)                 loc = [0]
1056: (8)                 scale = [1]
1057: (8)                 bad_scale = [-1]
1058: (8)                 gumbel = np.random.gumbel
1059: (8)                 desired = np.array([0.2730318639556768,
1060: (28)                   0.26936705726291116,
1061: (28)                   0.33906220393037939])
1062: (8)             self.setSeed()
1063: (8)             actual = gumbel(loc * 3, scale)
1064: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1065: (8)             assert_raises(ValueError, gumbel, loc * 3, bad_scale)
1066: (8)             self.setSeed()
1067: (8)             actual = gumbel(loc, scale * 3)
1068: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1069: (8)             assert_raises(ValueError, gumbel, loc, bad_scale * 3)
1070: (4)             def test_logistic(self):
1071: (8)                 loc = [0]
1072: (8)                 scale = [1]
1073: (8)                 bad_scale = [-1]
1074: (8)                 logistic = np.random.logistic
1075: (8)                 desired = np.array([0.13152135837586171,
1076: (28)                   0.13675915696285773,
1077: (28)                   0.038216792802833396])
1078: (8)             self.setSeed()
1079: (8)             actual = logistic(loc * 3, scale)
1080: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1081: (8)             assert_raises(ValueError, logistic, loc * 3, bad_scale)
1082: (8)             self.setSeed()
1083: (8)             actual = logistic(loc, scale * 3)
1084: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1085: (8)             assert_raises(ValueError, logistic, loc, bad_scale * 3)
1086: (4)             def test_lognormal(self):
1087: (8)                 mean = [0]
1088: (8)                 sigma = [1]
1089: (8)                 bad_sigma = [-1]
1090: (8)                 lognormal = np.random.lognormal
1091: (8)                 desired = np.array([9.1422086044848427,
1092: (28)                   8.4013952870126261,
1093: (28)                   6.3073234116578671])
1094: (8)             self.setSeed()
1095: (8)             actual = lognormal(mean * 3, sigma)
1096: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1097: (8)             assert_raises(ValueError, lognormal, mean * 3, bad_sigma)
1098: (8)             self.setSeed()
1099: (8)             actual = lognormal(mean, sigma * 3)

```

```

1100: (8) assert_array_almost_equal(actual, desired, decimal=14)
1101: (8) assert_raises(ValueError, lognormal, mean, bad_sigma * 3)
1102: (4) def test_rayleigh(self):
1103: (8)     scale = [1]
1104: (8)     bad_scale = [-1]
1105: (8)     rayleigh = np.random.rayleigh
1106: (8)     desired = np.array([1.2337491937897689,
1107: (28)             1.2360119924878694,
1108: (28)             1.1936818095781789])
1109: (8)     self.setSeed()
1110: (8)     actual = rayleigh(scale * 3)
1111: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1112: (8)     assert_raises(ValueError, rayleigh, bad_scale * 3)
1113: (4) def test_wald(self):
1114: (8)     mean = [0.5]
1115: (8)     scale = [1]
1116: (8)     bad_mean = [0]
1117: (8)     bad_scale = [-2]
1118: (8)     wald = np.random.wald
1119: (8)     desired = np.array([0.11873681120271318,
1120: (28)         0.12450084820795027,
1121: (28)         0.9096122728408238])
1122: (8)     self.setSeed()
1123: (8)     actual = wald(mean * 3, scale)
1124: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1125: (8)     assert_raises(ValueError, wald, bad_mean * 3, scale)
1126: (8)     assert_raises(ValueError, wald, mean * 3, bad_scale)
1127: (8)     self.setSeed()
1128: (8)     actual = wald(mean, scale * 3)
1129: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1130: (8)     assert_raises(ValueError, wald, bad_mean, scale * 3)
1131: (8)     assert_raises(ValueError, wald, mean, bad_scale * 3)
1132: (8)     assert_raises(ValueError, wald, 0.0, 1)
1133: (8)     assert_raises(ValueError, wald, 0.5, 0.0)
1134: (4) def test_triangular(self):
1135: (8)     left = [1]
1136: (8)     right = [3]
1137: (8)     mode = [2]
1138: (8)     bad_left_one = [3]
1139: (8)     bad_mode_one = [4]
1140: (8)     bad_left_two, bad_mode_two = right * 2
1141: (8)     triangular = np.random.triangular
1142: (8)     desired = np.array([2.03339048710429,
1143: (28)         2.0347400359389356,
1144: (28)         2.0095991069536208])
1145: (8)     self.setSeed()
1146: (8)     actual = triangular(left * 3, mode, right)
1147: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1148: (8)     assert_raises(ValueError, triangular, bad_left_one * 3, mode, right)
1149: (8)     assert_raises(ValueError, triangular, left * 3, bad_mode_one, right)
1150: (8)     assert_raises(ValueError, triangular, bad_left_two * 3, bad_mode_two,
1151: (22)             right)
1152: (8)     self.setSeed()
1153: (8)     actual = triangular(left, mode * 3, right)
1154: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1155: (8)     assert_raises(ValueError, triangular, bad_left_one, mode * 3, right)
1156: (8)     assert_raises(ValueError, triangular, left, bad_mode_one * 3, right)
1157: (8)     assert_raises(ValueError, triangular, bad_left_two, bad_mode_two * 3,
1158: (22)             right)
1159: (8)     self.setSeed()
1160: (8)     actual = triangular(left, mode, right * 3)
1161: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1162: (8)     assert_raises(ValueError, triangular, bad_left_one, mode, right * 3)
1163: (8)     assert_raises(ValueError, triangular, left, bad_mode_one, right * 3)
1164: (8)     assert_raises(ValueError, triangular, bad_left_two, bad_mode_two,
1165: (22)             right * 3)
1166: (4) def test_binomial(self):
1167: (8)     n = [1]
1168: (8)     p = [0.5]

```

```

1169: (8)             bad_n = [-1]
1170: (8)             bad_p_one = [-1]
1171: (8)             bad_p_two = [1.5]
1172: (8)             binom = np.random.binomial
1173: (8)             desired = np.array([1, 1, 1])
1174: (8)             self.setSeed()
1175: (8)             actual = binom(n * 3, p)
1176: (8)             assert_array_equal(actual, desired)
1177: (8)             assert_raises(ValueError, binom, bad_n * 3, p)
1178: (8)             assert_raises(ValueError, binom, n * 3, bad_p_one)
1179: (8)             assert_raises(ValueError, binom, n * 3, bad_p_two)
1180: (8)             self.setSeed()
1181: (8)             actual = binom(n, p * 3)
1182: (8)             assert_array_equal(actual, desired)
1183: (8)             assert_raises(ValueError, binom, bad_n, p * 3)
1184: (8)             assert_raises(ValueError, binom, n, bad_p_one * 3)
1185: (8)             assert_raises(ValueError, binom, n, bad_p_two * 3)
1186: (4)             def test_negative_binomial(self):
1187: (8)                 n = [1]
1188: (8)                 p = [0.5]
1189: (8)                 bad_n = [-1]
1190: (8)                 bad_p_one = [-1]
1191: (8)                 bad_p_two = [1.5]
1192: (8)                 neg_binom = np.random.negative_binomial
1193: (8)                 desired = np.array([1, 0, 1])
1194: (8)                 self.setSeed()
1195: (8)                 actual = neg_binom(n * 3, p)
1196: (8)                 assert_array_equal(actual, desired)
1197: (8)                 assert_raises(ValueError, neg_binom, bad_n * 3, p)
1198: (8)                 assert_raises(ValueError, neg_binom, n * 3, bad_p_one)
1199: (8)                 assert_raises(ValueError, neg_binom, n * 3, bad_p_two)
1200: (8)                 self.setSeed()
1201: (8)                 actual = neg_binom(n, p * 3)
1202: (8)                 assert_array_equal(actual, desired)
1203: (8)                 assert_raises(ValueError, neg_binom, bad_n, p * 3)
1204: (8)                 assert_raises(ValueError, neg_binom, n, bad_p_one * 3)
1205: (8)                 assert_raises(ValueError, neg_binom, n, bad_p_two * 3)
1206: (4)             def test_poisson(self):
1207: (8)                 max_lam = np.random.RandomState().poisson_lam_max
1208: (8)                 lam = [1]
1209: (8)                 bad_lam_one = [-1]
1210: (8)                 bad_lam_two = [max_lam * 2]
1211: (8)                 poisson = np.random.poisson
1212: (8)                 desired = np.array([1, 1, 0])
1213: (8)                 self.setSeed()
1214: (8)                 actual = poisson(lam * 3)
1215: (8)                 assert_array_equal(actual, desired)
1216: (8)                 assert_raises(ValueError, poisson, bad_lam_one * 3)
1217: (8)                 assert_raises(ValueError, poisson, bad_lam_two * 3)
1218: (4)             def test_zipf(self):
1219: (8)                 a = [2]
1220: (8)                 bad_a = [0]
1221: (8)                 zipf = np.random.zipf
1222: (8)                 desired = np.array([2, 2, 1])
1223: (8)                 self.setSeed()
1224: (8)                 actual = zipf(a * 3)
1225: (8)                 assert_array_equal(actual, desired)
1226: (8)                 assert_raises(ValueError, zipf, bad_a * 3)
1227: (8)                 with np.errstate(invalid='ignore'):
1228: (12)                     assert_raises(ValueError, zipf, np.nan)
1229: (12)                     assert_raises(ValueError, zipf, [0, 0, np.nan])
1230: (4)             def test_geometric(self):
1231: (8)                 p = [0.5]
1232: (8)                 bad_p_one = [-1]
1233: (8)                 bad_p_two = [1.5]
1234: (8)                 geom = np.random.geometric
1235: (8)                 desired = np.array([2, 2, 2])
1236: (8)                 self.setSeed()
1237: (8)                 actual = geom(p * 3)

```

```

1238: (8)             assert_array_equal(actual, desired)
1239: (8)             assert_raises(ValueError, geom, bad_p_one * 3)
1240: (8)             assert_raises(ValueError, geom, bad_p_two * 3)
1241: (4)             def test_hypergeometric(self):
1242: (8)                 ngood = [1]
1243: (8)                 nbad = [2]
1244: (8)                 nsample = [2]
1245: (8)                 bad_ngood = [-1]
1246: (8)                 bad_nbad = [-2]
1247: (8)                 bad_nsampel_one = [0]
1248: (8)                 bad_nsampel_two = [4]
1249: (8)                 hypergeom = np.random.hypergeometric
1250: (8)                 desired = np.array([1, 1, 1])
1251: (8)                 self.setSeed()
1252: (8)                 actual = hypergeom/ngood * 3, nbad, nsample)
1253: (8)                 assert_array_equal(actual, desired)
1254: (8)                 assert_raises(ValueError, hypergeom, bad_ngood * 3, nbad, nsample)
1255: (8)                 assert_raises(ValueError, hypergeom, ngood * 3, bad_nbad, nsample)
1256: (8)                 assert_raises(ValueError, hypergeom, ngood * 3, nbad, bad_nsampel_one)
1257: (8)                 assert_raises(ValueError, hypergeom, ngood * 3, nbad, bad_nsampel_two)
1258: (8)                 self.setSeed()
1259: (8)                 actual = hypergeom/ngood, nbad * 3, nsample)
1260: (8)                 assert_array_equal(actual, desired)
1261: (8)                 assert_raises(ValueError, hypergeom, bad_ngood, nbad * 3, nsample)
1262: (8)                 assert_raises(ValueError, hypergeom, ngood, bad_nbad * 3, nsample)
1263: (8)                 assert_raises(ValueError, hypergeom, ngood, nbad * 3, bad_nsampel_one)
1264: (8)                 assert_raises(ValueError, hypergeom, ngood, nbad * 3, bad_nsampel_two)
1265: (8)                 self.setSeed()
1266: (8)                 actual = hypergeom/ngood, nbad, nsample * 3)
1267: (8)                 assert_array_equal(actual, desired)
1268: (8)                 assert_raises(ValueError, hypergeom, bad_ngood, nbad, nsample * 3)
1269: (8)                 assert_raises(ValueError, hypergeom, ngood, bad_nbad, nsample * 3)
1270: (8)                 assert_raises(ValueError, hypergeom, ngood, nbad, bad_nsampel_one * 3)
1271: (8)                 assert_raises(ValueError, hypergeom, ngood, nbad, bad_nsampel_two * 3)
1272: (4)             def test_logseries(self):
1273: (8)                 p = [0.5]
1274: (8)                 bad_p_one = [2]
1275: (8)                 bad_p_two = [-1]
1276: (8)                 logseries = np.random.logseries
1277: (8)                 desired = np.array([1, 1, 1])
1278: (8)                 self.setSeed()
1279: (8)                 actual = logseries(p * 3)
1280: (8)                 assert_array_equal(actual, desired)
1281: (8)                 assert_raises(ValueError, logseries, bad_p_one * 3)
1282: (8)                 assert_raises(ValueError, logseries, bad_p_two * 3)
1283: (0)             @pytest.mark.skipif(IS_WASM, reason="can't start thread")
1284: (0)             class TestThread:
1285: (4)                 def setup_method(self):
1286: (8)                     self.seeds = range(4)
1287: (4)                 def check_function(self, function, sz):
1288: (8)                     from threading import Thread
1289: (8)                     out1 = np.empty((len(self.seeds),) + sz)
1290: (8)                     out2 = np.empty((len(self.seeds),) + sz)
1291: (8)                     t = [Thread(target=function, args=(np.random.RandomState(s), o))
1292: (13)                         for s, o in zip(self.seeds, out1)]
1293: (8)                     [x.start() for x in t]
1294: (8)                     [x.join() for x in t]
1295: (8)                     for s, o in zip(self.seeds, out2):
1296: (12)                         function(np.random.RandomState(s), o)
1297: (8)                     if np.intp().dtype.itemsize == 4 and sys.platform == "win32":
1298: (12)                         assert_array_almost_equal(out1, out2)
1299: (8)                     else:
1300: (12)                         assert_array_equal(out1, out2)
1301: (4)             def test_normal(self):
1302: (8)                 def gen_random(state, out):
1303: (12)                     out[...] = state.normal(size=10000)
1304: (8)                     self.check_function(gen_random, sz=(10000,))
1305: (4)             def test_exp(self):
1306: (8)                 def gen_random(state, out):

```

```

1307: (12)             out[...] = state.exponential(scale=np.ones((100, 1000)))
1308: (8)              self.check_function(gen_random, sz=(100, 1000))
1309: (4)              def test_multinomial(self):
1310: (8)                def gen_random(state, out):
1311: (12)                  out[...] = state.multinomial(10, [1/6.]*6, size=10000)
1312: (8)                  self.check_function(gen_random, sz=(10000, 6))
1313: (0)              class TestSingleEltArrayInput:
1314: (4)                def setup_method(self):
1315: (8)                  self.argOne = np.array([2])
1316: (8)                  self.argTwo = np.array([3])
1317: (8)                  self.argThree = np.array([4])
1318: (8)                  self.tgtShape = (1,)
1319: (4)                def test_one_arg_funcs(self):
1320: (8)                  funcs = (np.random.exponential, np.random.standard_gamma,
1321: (17)                      np.random.chisquare, np.random.standard_t,
1322: (17)                      np.random.pareto, np.random.weibull,
1323: (17)                      np.random.power, np.random.rayleigh,
1324: (17)                      np.random.poisson, np.random.zipf,
1325: (17)                      np.random.geometric, np.random.logseries)
1326: (8)                  probfuncs = (np.random.geometric, np.random.logseries)
1327: (8)                  for func in funcs:
1328: (12)                    if func in probfuncs: # p < 1.0
1329: (16)                      out = func(np.array([0.5]))
1330: (12)                    else:
1331: (16)                      out = func(self.argOne)
1332: (12)                      assert_equal(out.shape, self.tgtShape)
1333: (4)                def test_two_arg_funcs(self):
1334: (8)                  funcs = (np.random.uniform, np.random.normal,
1335: (17)                      np.random.beta, np.random.gamma,
1336: (17)                      np.random.f, np.random.noncentral_chisquare,
1337: (17)                      np.random.vonmises, np.random.laplace,
1338: (17)                      np.random.gumbel, np.random.logistic,
1339: (17)                      np.random.lognormal, np.random.wald,
1340: (17)                      np.random.binomial, np.random.negative_binomial)
1341: (8)                  probfuncs = (np.random.binomial, np.random.negative_binomial)
1342: (8)                  for func in funcs:
1343: (12)                    if func in probfuncs: # p <= 1
1344: (16)                      argTwo = np.array([0.5])
1345: (12)                    else:
1346: (16)                      argTwo = self.argTwo
1347: (12)                      out = func(self.argOne, argTwo)
1348: (12)                      assert_equal(out.shape, self.tgtShape)
1349: (12)                      out = func(self.argOne[0], argTwo)
1350: (12)                      assert_equal(out.shape, self.tgtShape)
1351: (12)                      out = func(self.argOne, argTwo[0])
1352: (12)                      assert_equal(out.shape, self.tgtShape)
1353: (4)                def test_randint(self):
1354: (8)                  itype = [bool, np.int8, np.uint8, np.int16, np.uint16,
1355: (17)                      np.int32, np.uint32, np.int64, np.uint64]
1356: (8)                  func = np.random.randint
1357: (8)                  high = np.array([1])
1358: (8)                  low = np.array([0])
1359: (8)                  for dt in itype:
1360: (12)                    out = func(low, high, dtype=dt)
1361: (12)                    assert_equal(out.shape, self.tgtShape)
1362: (12)                    out = func(low[0], high, dtype=dt)
1363: (12)                    assert_equal(out.shape, self.tgtShape)
1364: (12)                    out = func(low, high[0], dtype=dt)
1365: (12)                    assert_equal(out.shape, self.tgtShape)
1366: (4)                def test_three_arg_funcs(self):
1367: (8)                  funcs = [np.random.noncentral_f, np.random.triangular,
1368: (17)                      np.random.hypergeometric]
1369: (8)                  for func in funcs:
1370: (12)                    out = func(self.argOne, self.argTwo, self.argThree)
1371: (12)                    assert_equal(out.shape, self.tgtShape)
1372: (12)                    out = func(self.argOne[0], self.argTwo, self.argThree)
1373: (12)                    assert_equal(out.shape, self.tgtShape)
1374: (12)                    out = func(self.argOne, self.argTwo[0], self.argThree)
1375: (12)                    assert_equal(out.shape, self.tgtShape)

```

File 315 - test\_randomstate.py:

```

1: (0)          import hashlib
2: (0)          import pickle
3: (0)          import sys
4: (0)          import warnings
5: (0)          import numpy as np
6: (0)          import pytest
7: (0)          from numpy.testing import (
8: (8)              assert_, assert_raises, assert_equal, assert_warns,
9: (8)              assert_no_warnings, assert_array_equal, assert_array_almost_equal,
10: (8)              suppress_warnings, IS_WASM
11: (8)          )
12: (0)          from numpy.random import MT19937, PCG64
13: (0)          from numpy import random
14: (0)          INT_FUNCS = {'binomial': (100.0, 0.6),
15: (13)              'geometric': (.5,),
16: (13)              'hypergeometric': (20, 20, 10),
17: (13)              'logseries': (.5,),
18: (13)              'multinomial': (20, np.ones(6) / 6.0),
19: (13)              'negative_binomial': (100, .5),
20: (13)              'poisson': (10.0,),
21: (13)              'zipf': (2,),
22: (13)          }
23: (0)          if np.iinfo(int).max < 2**32:
24: (4)              INT_FUNC_HASHES = {'binomial':
'2fbead005fc63942dec5326d36a1f32fe2c9d32c904ee61e46866b88447c263',
25: (23)                  'logseries':
'23ead5dcde35d4cf4ef2c105e4c3d43304b45dc1b1444b7823b9ee4fa144ebb',
26: (23)                  'geometric':
'0d764db64f5c3bad48c8c33551c13b4d07a1e7b470f77629bef6c985cac76fcf',
27: (23)                  'hypergeometric':
'7b59bf2f1691626c5815cdcd9a49e1dd68697251d4521575219e4d2a1b8b2c67',
28: (23)                  'multinomial':
'd754fa5b92943a38ec07630de92362dd2e02c43577fc147417dc5b9db94ccdd3',
29: (23)                  'negative_binomial':
'8eb216f7cb2a63cf55605422845caaff002fddc64a7dc8b2d45acd477a49e824',
30: (23)                  'poisson':
'70c891d76104013ebd6f6bcf30d403a9074b886ff62e4e6b8eb605bf1a4673b7',
31: (23)                  'zipf':
'01f074f97517cd5d21747148ac6ca4074dde7fcb7acbaec0a936606fecacd93f',
32: (23)          }
33: (0)          else:
34: (4)              INT_FUNC_HASHES = {'binomial':
'8626dd9d052cb608e93d8868de0a7b347258b199493871a1dc56e2a26cacb112',
35: (23)                  'geometric':
'8edd53d272e49c4fc8fbbe6c7d08d563d62e482921f3131d0a0e068af30f0db9',
36: (23)                  'hypergeometric':
'83496cc4281c77b786c9b7ad88b74d42e01603a55c60577ebab81c3ba8d45657',
37: (23)                  'logseries':
'65878a38747c176bc00e930ebafebb69d4e1e16cd3a704e264ea8f5e24f548db',
38: (23)                  'multinomial':
'7a984ae6dca26fd25374479e118b22f55db0aedcccd5a0f2584ceada33db98605',
39: (23)                  'negative_binomial':
'd636d968e6a24ae92ab52fe11c46ac45b0897e98714426764e820a7d77602a61',
40: (23)                  'poisson':
'956552176f77e7c9cb20d0118fc9cf690be488d790ed4b4c4747b965e61b0bb4',
41: (23)                  'zipf':
'f84ba7feffd41e606e20b28dfc0f1ea9964a74574513d4a4cbc98433a8bfa45',
42: (23)          }
43: (0)          @pytest.fixture(scope='module', params=INT_FUNCS)
44: (0)          def int_func(request):
45: (4)              return (request.param, INT_FUNCS[request.param],
46: (12)                  INT_FUNC_HASHES[request.param])
47: (0)          @pytest.fixture
48: (0)          def restore_singleton_bitgen():

```

```

49: (4)         """Ensures that the singleton bitgen is restored after a test"""
50: (4)         orig_bitgen = np.random.get_bit_generator()
51: (4)         yield
52: (4)         np.random.set_bit_generator(orig_bitgen)
53: (0)     def assert_mt19937_state_equal(a, b):
54: (4)         assert_equal(a['bit_generator'], b['bit_generator'])
55: (4)         assert_array_equal(a['state']['key'], b['state']['key'])
56: (4)         assert_array_equal(a['state']['pos'], b['state']['pos'])
57: (4)         assert_equal(a['has_gauss'], b['has_gauss'])
58: (4)         assert_equal(a['gauss'], b['gauss'])
59: (0)     class TestSeed:
60: (4)         def test_scalar(self):
61: (8)             s = random.RandomState(0)
62: (8)             assert_equal(s.randint(1000), 684)
63: (8)             s = random.RandomState(4294967295)
64: (8)             assert_equal(s.randint(1000), 419)
65: (4)         def test_array(self):
66: (8)             s = random.RandomState(range(10))
67: (8)             assert_equal(s.randint(1000), 468)
68: (8)             s = random.RandomState(np.arange(10))
69: (8)             assert_equal(s.randint(1000), 468)
70: (8)             s = random.RandomState([0])
71: (8)             assert_equal(s.randint(1000), 973)
72: (8)             s = random.RandomState([4294967295])
73: (8)             assert_equal(s.randint(1000), 265)
74: (4)         def test_invalid_scalar(self):
75: (8)             assert_raises(TypeError, random.RandomState, -0.5)
76: (8)             assert_raises(ValueError, random.RandomState, -1)
77: (4)         def test_invalid_array(self):
78: (8)             assert_raises(TypeError, random.RandomState, [-0.5])
79: (8)             assert_raises(ValueError, random.RandomState, [-1])
80: (8)             assert_raises(ValueError, random.RandomState, [4294967296])
81: (8)             assert_raises(ValueError, random.RandomState, [1, 2, 4294967296])
82: (8)             assert_raises(ValueError, random.RandomState, [1, -2, 4294967296])
83: (4)         def test_invalid_array_shape(self):
84: (8)             assert_raises(ValueError, random.RandomState, np.array([], 
85: (63)           dtype=np.int64))
86: (8)             assert_raises(ValueError, random.RandomState, [[1, 2, 3]])
87: (8)             assert_raises(ValueError, random.RandomState, [[1, 2, 3],
88: (55)                           [4, 5, 6]])
89: (4)         def test_CANNOT_SEED(self):
90: (8)             rs = random.RandomState(PCG64(0))
91: (8)             with assert_raises(TypeError):
92: (12)                 rs.seed(1234)
93: (4)         def test_INVALID_INITIALIZATION(self):
94: (8)             assert_raises(ValueError, random.RandomState, MT19937)
95: (0)     class TestBinomial:
96: (4)         def test_n_zero(self):
97: (8)             zeros = np.zeros(2, dtype='int')
98: (8)             for p in [0, .5, 1]:
99: (12)                 assert_(random.binomial(0, p) == 0)
100: (12)                assert_array_equal(random.binomial(zeros, p), zeros)
101: (4)         def test_p_is_nan(self):
102: (8)             assert_raises(ValueError, random.binomial, 1, np.nan)
103: (0)     class TestMultinomial:
104: (4)         def test_basic(self):
105: (8)             random.multinomial(100, [0.2, 0.8])
106: (4)         def test_zero_probability(self):
107: (8)             random.multinomial(100, [0.2, 0.8, 0.0, 0.0, 0.0])
108: (4)         def test_int_negative_interval(self):
109: (8)             assert_(-5 <= random.randint(-5, -1) < -1)
110: (8)             x = random.randint(-5, -1, 5)
111: (8)             assert_(np.all(-5 <= x))
112: (8)             assert_(np.all(x < -1))
113: (4)         def test_size(self):
114: (8)             p = [0.5, 0.5]
115: (8)             assert_equal(random.multinomial(1, p, np.uint32(1)).shape, (1, 2))
116: (8)             assert_equal(random.multinomial(1, p, np.uint32(1)).shape, (1, 2))

```

```

117: (8) assert_equal(random.multinomial(1, p, np.uint32(1)).shape, (1, 2))
118: (8) assert_equal(random.multinomial(1, p, [2, 2]).shape, (2, 2, 2))
119: (8) assert_equal(random.multinomial(1, p, (2, 2)).shape, (2, 2, 2))
120: (8) assert_equal(random.multinomial(1, p, np.array((2, 2))).shape,
121: (21) (2, 2, 2))
122: (8) assert_raises(TypeError, random.multinomial, 1, p,
123: (22) float(1))
124: (4) def test_invalid_prob(self):
125: (8) assert_raises(ValueError, random.multinomial, 100, [1.1, 0.2])
126: (8) assert_raises(ValueError, random.multinomial, 100, [-.1, 0.9])
127: (4) def test_invalid_n(self):
128: (8) assert_raises(ValueError, random.multinomial, -1, [0.8, 0.2])
129: (4) def test_p_non_contiguous(self):
130: (8) p = np.arange(15.)
131: (8) p /= np.sum(p[1::3])
132: (8) pvals = p[1::3]
133: (8) random.seed(1432985819)
134: (8) non_contig = random.multinomial(100, pvals=pvals)
135: (8) random.seed(1432985819)
136: (8) contig = random.multinomial(100, pvals=np.ascontiguousarray(pvals))
137: (8) assert_array_equal(non_contig, contig)
138: (4) def test_multinomial_pvals_float32(self):
139: (8) x = np.array([9.9e-01, 9.9e-01, 1.0e-09, 1.0e-09, 1.0e-09, 1.0e-09,
140: (22) 1.0e-09, 1.0e-09, 1.0e-09, 1.0e-09], dtype=np.float32)
141: (8) pvals = x / x.sum()
142: (8) match = r"[\w\s]*pvals array is cast to 64-bit floating"
143: (8) with pytest.raises(ValueError, match=match):
144: (12)     random.multinomial(1, pvals)
145: (4) def test_multinomial_n_float(self):
146: (8)     random.multinomial(100.5, [0.2, 0.8])
147: (0) class TestSetState:
148: (4)     def setup_method(self):
149: (8)         self.seed = 1234567890
150: (8)         self.random_state = random.RandomState(self.seed)
151: (8)         self.state = self.random_state.get_state()
152: (4)     def test_basic(self):
153: (8)         old = self.random_state.tomaxint(16)
154: (8)         self.random_state.set_state(self.state)
155: (8)         new = self.random_state.tomaxint(16)
156: (8)         assert_(np.all(old == new))
157: (4)     def test_gaussian_reset(self):
158: (8)         old = self.random_state.standard_normal(size=3)
159: (8)         self.random_state.set_state(self.state)
160: (8)         new = self.random_state.standard_normal(size=3)
161: (8)         assert_(np.all(old == new))
162: (4)     def test_gaussian_reset_in_media_res(self):
163: (8)         self.random_state.standard_normal()
164: (8)         state = self.random_state.get_state()
165: (8)         old = self.random_state.standard_normal(size=3)
166: (8)         self.random_state.set_state(state)
167: (8)         new = self.random_state.standard_normal(size=3)
168: (8)         assert_(np.all(old == new))
169: (4)     def test_backwards_compatibility(self):
170: (8)         old_state = self.state[:-2]
171: (8)         x1 = self.random_state.standard_normal(size=16)
172: (8)         self.random_state.set_state(old_state)
173: (8)         x2 = self.random_state.standard_normal(size=16)
174: (8)         self.random_state.set_state(self.state)
175: (8)         x3 = self.random_state.standard_normal(size=16)
176: (8)         assert_(np.all(x1 == x2))
177: (8)         assert_(np.all(x1 == x3))
178: (4)     def test_negative_binomial(self):
179: (8)         self.random_state.negative_binomial(0.5, 0.5)
180: (4)     def test_get_state_warning(self):
181: (8)         rs = random.RandomState(PCG64())
182: (8)         with suppress_warnings() as sup:
183: (12)             w = sup.record(RuntimeWarning)
184: (12)             state = rs.get_state()
185: (12)             assert_(len(w) == 1)

```

```

186: (12)             assert isinstance(state, dict)
187: (12)             assert state['bit_generator'] == 'PCG64'
188: (4)              def test_invalid_legacy_state_setting(self):
189: (8)                state = self.random_state.get_state()
190: (8)                new_state = ('Unknown', ) + state[1:]
191: (8)                assert_raises(ValueError, self.random_state.set_state, new_state)
192: (8)                assert_raises(TypeError, self.random_state.set_state,
193: (22)                  np.array(new_state, dtype=object))
194: (8)                state = self.random_state.get_state(legacy=False)
195: (8)                del state['bit_generator']
196: (8)                assert_raises(ValueError, self.random_state.set_state, state)
197: (4)              def test_pickle(self):
198: (8)                self.random_state.seed(0)
199: (8)                self.random_state.random_sample(100)
200: (8)                self.random_state.standard_normal()
201: (8)                pickled = self.random_state.get_state(legacy=False)
202: (8)                assert_equal(pickled['has_gauss'], 1)
203: (8)                rs_unpick = pickle.loads(pickle.dumps(self.random_state))
204: (8)                unpickled = rs_unpick.get_state(legacy=False)
205: (8)                assert_mt19937_state_equal(pickled, unpickled)
206: (4)              def test_state_setting(self):
207: (8)                attr_state = self.random_state.__getstate__()
208: (8)                self.random_state.standard_normal()
209: (8)                self.random_state.__setstate__(attr_state)
210: (8)                state = self.random_state.get_state(legacy=False)
211: (8)                assert_mt19937_state_equal(attr_state, state)
212: (4)              def test_repr(self):
213: (8)                assert repr(self.random_state).startswith('RandomState(MT19937)')
214: (0)              class TestRandint:
215: (4)                rfunc = random.randint
216: (4)                itype = [np.bool_, np.int8, np.uint8, np.int16, np.uint16,
217: (13)                  np.int32, np.uint32, np.int64, np.uint64]
218: (4)                def test_unsupported_type(self):
219: (8)                  assert_raises(TypeError, self.rfunc, 1, dtype=float)
220: (4)                def test_bounds_checking(self):
221: (8)                  for dt in self.itype:
222: (12)                    lbind = 0 if dt is np.bool_ else np.iinfo(dt).min
223: (12)                    ubnd = 2 if dt is np.bool_ else np.iinfo(dt).max + 1
224: (12)                    assert_raises(ValueError, self.rfunc, lbind - 1, ubnd, dtype=dt)
225: (12)                    assert_raises(ValueError, self.rfunc, lbind, ubnd + 1, dtype=dt)
226: (12)                    assert_raises(ValueError, self.rfunc, ubnd, lbind, dtype=dt)
227: (12)                    assert_raises(ValueError, self.rfunc, 1, 0, dtype=dt)
228: (4)                def test_rng_zero_and_extremes(self):
229: (8)                  for dt in self.itype:
230: (12)                    lbind = 0 if dt is np.bool_ else np.iinfo(dt).min
231: (12)                    ubnd = 2 if dt is np.bool_ else np.iinfo(dt).max + 1
232: (12)                    tgt = ubnd - 1
233: (12)                    assert_equal(self.rfunc(tgt, tgt + 1, size=1000, dtype=dt), tgt)
234: (12)                    tgt = lbind
235: (12)                    assert_equal(self.rfunc(tgt, tgt + 1, size=1000, dtype=dt), tgt)
236: (12)                    tgt = (lbind + ubnd)//2
237: (12)                    assert_equal(self.rfunc(tgt, tgt + 1, size=1000, dtype=dt), tgt)
238: (4)                def test_full_range(self):
239: (8)                  for dt in self.itype:
240: (12)                    lbind = 0 if dt is np.bool_ else np.iinfo(dt).min
241: (12)                    ubnd = 2 if dt is np.bool_ else np.iinfo(dt).max + 1
242: (12)                    try:
243: (16)                      self.rfunc(lbind, ubnd, dtype=dt)
244: (12)                    except Exception as e:
245: (16)                      raise AssertionError("No error should have been raised, "
246: (37)                        "but one was with the following "
247: (37)                        "message:\n\n%s" % str(e))
248: (4)                def test_in_bounds_fuzz(self):
249: (8)                  random.seed()
250: (8)                  for dt in self.itype[1:]:
251: (12)                    for ubnd in [4, 8, 16]:
252: (16)                      vals = self.rfunc(2, ubnd, size=2**16, dtype=dt)
253: (16)                      assert_(vals.max() < ubnd)
254: (16)                      assert_(vals.min() >= 2)

```

```

255: (8)             vals = self.rfunc(0, 2, size=2**16, dtype=np.bool_)
256: (8)             assert_(vals.max() < 2)
257: (8)             assert_(vals.min() >= 0)
258: (4)             def test_repeatability(self):
259: (8)                 tgt = {'bool':
'509aea74d792fb931784c4b0135392c65aec64beee12b0cc167548a2c3d31e71',
260: (15)                  'int16':
'7b07f1a920e46f6d0fe02314155a2330bcfd7635e708da50e536c5ebb631a7d4',
261: (15)                  'int32':
'e577bfed6c935de944424667e3da285012e741892dc7051a8f1ce68ab05c92f',
262: (15)                  'int64':
'0fbead0b06759df2cfb55e43148822d4a1ff953c7eb19a5b08445a63bb64fa9e',
263: (15)                  'int8':
'001aac3a5acb935a9b186cbe14a1ca064b8bb2dd0b045d48abeacf74d0203404',
264: (15)                  'uint16':
'7b07f1a920e46f6d0fe02314155a2330bcfd7635e708da50e536c5ebb631a7d4',
265: (15)                  'uint32':
'e577bfed6c935de944424667e3da285012e741892dc7051a8f1ce68ab05c92f',
266: (15)                  'uint64':
'0fbead0b06759df2cfb55e43148822d4a1ff953c7eb19a5b08445a63bb64fa9e',
267: (15)                  'uint8':
'001aac3a5acb935a9b186cbe14a1ca064b8bb2dd0b045d48abeacf74d0203404'}
268: (8)             for dt in self.dtype[1:]:
269: (12)                 random.seed(1234)
270: (12)                 if sys.byteorder == 'little':
271: (16)                     val = self.rfunc(0, 6, size=1000, dtype=dt)
272: (12)                 else:
273: (16)                     val = self.rfunc(0, 6, size=1000, dtype=dt).byteswap()
274: (12)                 res = hashlib.sha256(val.view(np.int8)).hexdigest()
275: (12)                 assert_(tgt[np.dtype(dt).name] == res)
276: (8)                 random.seed(1234)
277: (8)                 val = self.rfunc(0, 2, size=1000, dtype=bool).view(np.int8)
278: (8)                 res = hashlib.sha256(val).hexdigest()
279: (8)                 assert_(tgt[np.dtype(bool).name] == res)
280: (4)                 @pytest.mark.skipif(np.iinfo('l').max < 2**32,
281: (24)                               reason='Cannot test with 32-bit C long')
282: (4)             def test_repeatability_32bit_boundary_broadcasting(self):
283: (8)                 desired = np.array([[3992670689, 2438360420, 2557845020],
284: (29)                               [4107320065, 4142558326, 3216529513],
285: (29)                               [1605979228, 2807061240, 665605495]],
286: (28)                               [[3211410639, 4128781000, 457175120],
287: (29)                               [1712592594, 1282922662, 3081439808],
288: (29)                               [3997822960, 2008322436, 1563495165]],
289: (28)                               [[1398375547, 4269260146, 115316740],
290: (29)                               [3414372578, 3437564012, 2112038651],
291: (29)                               [3572980305, 2260248732, 3908238631]],
292: (28)                               [[2561372503, 223155946, 3127879445],
293: (29)                               [441282060, 3514786552, 2148440361],
294: (29)                               [1629275283, 3479737011, 3003195987]],
295: (28)                               [[412181688, 940383289, 3047321305],
296: (29)                               [2978368172, 764731833, 2282559898],
297: (29)                               [105711276, 720447391, 3596512484]]])
298: (8)             for size in [None, (5, 3, 3)]:
299: (12)                 random.seed(12345)
300: (12)                 x = self.rfunc([-1], [0], [1]), [2**32 - 1, 2**32, 2**32 + 1],
301: (27)                               size=size)
302: (12)                 assert_array_equal(x, desired if size is not None else desired[0])
303: (4)             def test_int64_uint64_corner_case(self):
304: (8)                 dt = np.int64
305: (8)                 tgt = np.iinfo(np.int64).max
306: (8)                 lbnd = np.int64(np.iinfo(np.int64).max)
307: (8)                 ubnd = np.uint64(np.iinfo(np.int64).max + 1)
308: (8)                 actual = random.randint(lbnd, ubnd, dtype=dt)
309: (8)                 assert_equal(actual, tgt)
310: (4)             def test_respect_dtype_singleton(self):
311: (8)                 for dt in self.dtype:
312: (12)                     lbnd = 0 if dt is np.bool_ else np.iinfo(dt).min
313: (12)                     ubnd = 2 if dt is np.bool_ else np.iinfo(dt).max + 1
314: (12)                     sample = self.rfunc(lbnd, ubnd, dtype=dt)

```

```

315: (12)             assert_equal(sample.dtype, np.dtype(dt))
316: (8)              for dt in (bool, int):
317: (12)                lbind = 0 if dt is bool else np.iinfo(dt).min
318: (12)                ubind = 2 if dt is bool else np.iinfo(dt).max + 1
319: (12)                sample = self.rfunc(lbind, ubind, dtype=dt)
320: (12)                assert_(not hasattr(sample, 'dtype'))
321: (12)                assert_equal(type(sample), dt)
322: (0)               class TestRandomDist:
323: (4)                 def setup_method(self):
324: (8)                   self.seed = 1234567890
325: (4)                 def test_rand(self):
326: (8)                   random.seed(self.seed)
327: (8)                   actual = random.rand(3, 2)
328: (8)                   desired = np.array([[0.61879477158567997, 0.59162362775974664],
329: (28)                           [0.88868358904449662, 0.89165480011560816],
330: (28)                           [0.4575674820298663, 0.7781880808593471]])
331: (8)                   assert_array_almost_equal(actual, desired, decimal=15)
332: (4)                 def test_rand_singleton(self):
333: (8)                   random.seed(self.seed)
334: (8)                   actual = random.rand()
335: (8)                   desired = 0.61879477158567997
336: (8)                   assert_array_almost_equal(actual, desired, decimal=15)
337: (4)                 def test_ranndn(self):
338: (8)                   random.seed(self.seed)
339: (8)                   actual = random.ranndn(3, 2)
340: (8)                   desired = np.array([[1.34016345771863121, 1.73759122771936081],
341: (27)                           [1.498988344300628, -0.2286433324536169],
342: (27)                           [2.031033998682787, 2.17032494605655257]])
343: (8)                   assert_array_almost_equal(actual, desired, decimal=15)
344: (8)                   random.seed(self.seed)
345: (8)                   actual = random.ranndn()
346: (8)                   assert_array_almost_equal(actual, desired[0, 0], decimal=15)
347: (4)                 def test_randint(self):
348: (8)                   random.seed(self.seed)
349: (8)                   actual = random.randint(-99, 99, size=(3, 2))
350: (8)                   desired = np.array([[31, 3],
351: (28)                           [-52, 41],
352: (28)                           [-48, -66]])
353: (8)                   assert_array_equal(actual, desired)
354: (4)                 def test_random_integers(self):
355: (8)                   random.seed(self.seed)
356: (8)                   with suppress_warnings() as sup:
357: (12)                     w = sup.record(DeprecationWarning)
358: (12)                     actual = random.random_integers(-99, 99, size=(3, 2))
359: (12)                     assert_(len(w) == 1)
360: (8)                   desired = np.array([[31, 3],
361: (28)                           [-52, 41],
362: (28)                           [-48, -66]])
363: (8)                   assert_array_equal(actual, desired)
364: (8)                   random.seed(self.seed)
365: (8)                   with suppress_warnings() as sup:
366: (12)                     w = sup.record(DeprecationWarning)
367: (12)                     actual = random.random_integers(198, size=(3, 2))
368: (12)                     assert_(len(w) == 1)
369: (8)                   assert_array_equal(actual, desired + 100)
370: (4)                 def test_tomaxint(self):
371: (8)                   random.seed(self.seed)
372: (8)                   rs = random.RandomState(self.seed)
373: (8)                   actual = rs.tomaxint(size=(3, 2))
374: (8)                   if np.iinfo(int).max == 2147483647:
375: (12)                     desired = np.array([[1328851649, 731237375],
376: (32)                           [1270502067, 320041495],
377: (32)                           [1908433478, 499156889]], dtype=np.int64)
378: (8)                   else:
379: (12)                     desired = np.array([[5707374374421908479, 5456764827585442327],
380: (32)                           [8196659375100692377, 8224063923314595285],
381: (32)                           [4220315081820346526, 7177518203184491332]],
382: (31)                           dtype=np.int64)
383: (8)                   assert_equal(actual, desired)

```

```

384: (8)           rs.seed(self.seed)
385: (8)           actual = rs.tomaxint()
386: (8)           assert_equal(actual, desired[0, 0])
387: (4)           def test_random_integers_max_int(self):
388: (8)             with suppress_warnings() as sup:
389: (12)               w = sup.record(DeprecationWarning)
390: (12)               actual = random.random_integers(np.iinfo('l').max,
391: (44)                                     np.iinfo('l').max)
392: (12)               assert_(len(w) == 1)
393: (8)             desired = np.iinfo('l').max
394: (8)             assert_equal(actual, desired)
395: (8)             with suppress_warnings() as sup:
396: (12)               w = sup.record(DeprecationWarning)
397: (12)               typer = np.dtype('l').type
398: (12)               actual = random.random_integers(typer(np.iinfo('l').max),
399: (44)                                     typer(np.iinfo('l').max))
400: (12)               assert_(len(w) == 1)
401: (8)             assert_equal(actual, desired)
402: (4)           def test_random_integers_deprecated(self):
403: (8)             with warnings.catch_warnings():
404: (12)               warnings.simplefilter("error", DeprecationWarning)
405: (12)               assert_raises(DeprecationWarning,
406: (26)                 random.random_integers,
407: (26)                 np.iinfo('l').max)
408: (12)               assert_raises(DeprecationWarning,
409: (26)                 random.random_integers,
410: (26)                 np.iinfo('l').max, np.iinfo('l').max)
411: (4)           def test_random_sample(self):
412: (8)             random.seed(self.seed)
413: (8)             actual = random.random_sample((3, 2))
414: (8)             desired = np.array([[0.61879477158567997, 0.59162362775974664],
415: (28)                   [0.88868358904449662, 0.89165480011560816],
416: (28)                   [0.4575674820298663, 0.7781880808593471]])
417: (8)             assert_array_almost_equal(actual, desired, decimal=15)
418: (8)             random.seed(self.seed)
419: (8)             actual = random.random_sample()
420: (8)             assert_array_almost_equal(actual, desired[0, 0], decimal=15)
421: (4)           def test_choice_uniform_replace(self):
422: (8)             random.seed(self.seed)
423: (8)             actual = random.choice(4, 4)
424: (8)             desired = np.array([2, 3, 2, 3])
425: (8)             assert_array_equal(actual, desired)
426: (4)           def test_choice_nonuniform_replace(self):
427: (8)             random.seed(self.seed)
428: (8)             actual = random.choice(4, 4, p=[0.4, 0.4, 0.1, 0.1])
429: (8)             desired = np.array([1, 1, 2, 2])
430: (8)             assert_array_equal(actual, desired)
431: (4)           def test_choice_uniform_noreplace(self):
432: (8)             random.seed(self.seed)
433: (8)             actual = random.choice(4, 3, replace=False)
434: (8)             desired = np.array([0, 1, 3])
435: (8)             assert_array_equal(actual, desired)
436: (4)           def test_choice_nonuniform_noreplace(self):
437: (8)             random.seed(self.seed)
438: (8)             actual = random.choice(4, 3, replace=False, p=[0.1, 0.3, 0.5, 0.1])
439: (8)             desired = np.array([2, 3, 1])
440: (8)             assert_array_equal(actual, desired)
441: (4)           def test_choice_noninteger(self):
442: (8)             random.seed(self.seed)
443: (8)             actual = random.choice(['a', 'b', 'c', 'd'], 4)
444: (8)             desired = np.array(['c', 'd', 'c', 'd'])
445: (8)             assert_array_equal(actual, desired)
446: (4)           def test_choice_exceptions(self):
447: (8)             sample = random.choice
448: (8)             assert_raises(ValueError, sample, -1, 3)
449: (8)             assert_raises(ValueError, sample, 3., 3)
450: (8)             assert_raises(ValueError, sample, [[1, 2], [3, 4]], 3)
451: (8)             assert_raises(ValueError, sample, [], 3)
452: (8)             assert_raises(ValueError, sample, [1, 2, 3, 4], 3,

```

```

453: (22)
454: (8)           p=[[0.25, 0.25], [0.25, 0.25]])
455: (8)           assert_raises(ValueError, sample, [1, 2], 3, p=[0.4, 0.4, 0.2])
456: (8)           assert_raises(ValueError, sample, [1, 2], 3, p=[1.1, -0.1])
457: (8)           assert_raises(ValueError, sample, [1, 2], 3, p=[0.4, 0.4])
458: (8)           assert_raises(ValueError, sample, [1, 2, 3], 4, replace=False)
459: (8)           assert_raises(ValueError, sample, [1, 2, 3], -2, replace=False)
460: (8)           assert_raises(ValueError, sample, [1, 2, 3], (-1,), replace=False)
461: (8)           assert_raises(ValueError, sample, [1, 2, 3], (-1, 1), replace=False)
462: (22)          assert_raises(ValueError, sample, [1, 2, 3], 2,
463: (4)                  replace=False, p=[1, 0, 0])
464: (8)
465: (8)           def test_choice_return_shape(self):
466: (8)               p = [0.1, 0.9]
467: (8)               assert_(np.isscalar(random.choice(2, replace=True)))
468: (8)               assert_(np.isscalar(random.choice(2, replace=False)))
469: (8)               assert_(np.isscalar(random.choice(2, replace=True, p=p)))
470: (8)               assert_(np.isscalar(random.choice(2, replace=False, p=p)))
471: (8)               assert_(np.isscalar(random.choice([1, 2], replace=True)))
472: (8)               assert_(random.choice([None], replace=True) is None)
473: (8)               a = np.array([1, 2])
474: (8)               arr = np.empty(1, dtype=object)
475: (8)               arr[0] = a
476: (8)               assert_(random.choice(arr, replace=True) is a)
477: (8)               s = tuple()
478: (8)               assert_(not np.isscalar(random.choice(2, s, replace=True)))
479: (8)               assert_(not np.isscalar(random.choice(2, s, replace=False)))
480: (8)               assert_(not np.isscalar(random.choice(2, s, replace=True, p=p)))
481: (8)               assert_(not np.isscalar(random.choice(2, s, replace=False, p=p)))
482: (8)               assert_(not np.isscalar(random.choice([1, 2], s, replace=True)))
483: (8)               assert_(random.choice([None], s, replace=True).ndim == 0)
484: (8)               a = np.array([1, 2])
485: (8)               arr = np.empty(1, dtype=object)
486: (8)               arr[0] = a
487: (8)               assert_(random.choice(arr, s, replace=True).item() is a)
488: (8)               s = (2, 3)
489: (8)               p = [0.1, 0.1, 0.1, 0.1, 0.4, 0.2]
490: (8)               assert_equal(random.choice(6, s, replace=True).shape, s)
491: (8)               assert_equal(random.choice(6, s, replace=False).shape, s)
492: (8)               assert_equal(random.choice(6, s, replace=True, p=p).shape, s)
493: (8)               assert_equal(random.choice(6, s, replace=False, p=p).shape, s)
494: (8)               assert_equal(random.choice(np.arange(6), s, replace=True).shape, s)
495: (8)               assert_equal(random.randint(0, 0, size=(3, 0, 4)).shape, (3, 0, 4))
496: (8)               assert_equal(random.randint(0, -10, size=0).shape, (0,))
497: (8)               assert_equal(random.randint(10, 10, size=0).shape, (0,))
498: (8)               assert_equal(random.choice(0, size=0).shape, (0,))
499: (21)              assert_equal(random.choice([], size=(0,)).shape, (0,))
500: (8)              assert_equal(random.choice(['a', 'b'], size=(3, 0, 4)).shape,
501: (4)                  (3, 0, 4))
502: (8)              assert_raises(ValueError, random.choice, [], 10)
503: (8)
504: (8)              def test_choice_nan_probabilities(self):
505: (4)                  a = np.array([42, 1, 2])
506: (8)                  p = [None, None, None]
507: (8)                  assert_raises(ValueError, random.choice, a, p=p)
508: (8)
509: (8)              def test_choice_p_non_contiguous(self):
510: (8)                  p = np.ones(10) / 5
511: (8)                  p[1::2] = 3.0
512: (8)                  random.seed(self.seed)
513: (4)                  non_contig = random.choice(5, 3, p=p[::2])
514: (8)                  random.seed(self.seed)
515: (8)                  contig = random.choice(5, 3, p=np.ascontiguousarray(p[::2]))
516: (8)                  assert_array_equal(non_contig, contig)
517: (8)
518: (4)              def test_bytes(self):
519: (8)                  random.seed(self.seed)
520: (8)                  actual = random.bytes(10)
521: (21)                 desired = b'\x82Ui\x9e\xff\x97+WF\xA5'

```

```

522: (21)           lambda x: np.asarray(x).astype(np.float32),
523: (21)           lambda x: np.asarray(x).astype(np.complex64),
524: (21)           lambda x: np.asarray(x).astype(object),
525: (21)           lambda x: [(i, i) for i in x],
526: (21)           lambda x: np.asarray([[i, i] for i in x]),
527: (21)           lambda x: np.vstack([x, x]).T,
528: (21)           lambda x: (np.asarray([(i, i) for i in x],
529: (43)                         [("a", int), ("b", int)])
530: (32)                           .view(np.recarray)),
531: (21)           lambda x: np.asarray([(i, i) for i in x],
532: (42)                         [("a", object, (1,)),
533: (43)                           ("b", np.int32, (1,))]):]
534: (12)           random.seed(self.seed)
535: (12)           alist = conv([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
536: (12)           random.shuffle(alist)
537: (12)           actual = alist
538: (12)           desired = conv([0, 1, 9, 6, 2, 4, 5, 8, 7, 3])
539: (12)           assert_array_equal(actual, desired)
540: (4)            def test_shuffle_masked(self):
541: (8)              a = np.ma.masked_values(np.reshape(range(20), (5, 4)) % 3 - 1, -1)
542: (8)              b = np.ma.masked_values(np.arange(20) % 3 - 1, -1)
543: (8)              a_orig = a.copy()
544: (8)              b_orig = b.copy()
545: (8)              for i in range(50):
546: (12)                  random.shuffle(a)
547: (12)                  assert_equal(
548: (16)                      sorted(a.data[~a.mask]), sorted(a_orig.data[~a_orig.mask]))
549: (12)                  random.shuffle(b)
550: (12)                  assert_equal(
551: (16)                      sorted(b.data[~b.mask]), sorted(b_orig.data[~b_orig.mask]))
552: (8)            def test_shuffle_invalid_objects(self):
553: (12)              x = np.array(3)
554: (12)              assert_raises(TypeError, random.shuffle, x)
555: (4)            def test_permutation(self):
556: (8)              random.seed(self.seed)
557: (8)              alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
558: (8)              actual = random.permutation(alist)
559: (8)              desired = [0, 1, 9, 6, 2, 4, 5, 8, 7, 3]
560: (8)              assert_array_equal(actual, desired)
561: (8)              random.seed(self.seed)
562: (8)              arr_2d = np.atleast_2d([1, 2, 3, 4, 5, 6, 7, 8, 9, 0]).T
563: (8)              actual = random.permutation(arr_2d)
564: (8)              assert_array_equal(actual, np.atleast_2d(desired).T)
565: (8)              random.seed(self.seed)
566: (8)              bad_x_str = "abcd"
567: (8)              assert_raises(IndexError, random.permutation, bad_x_str)
568: (8)              random.seed(self.seed)
569: (8)              bad_x_float = 1.2
570: (8)              assert_raises(IndexError, random.permutation, bad_x_float)
571: (8)              integer_val = 10
572: (8)              desired = [9, 0, 8, 5, 1, 3, 4, 7, 6, 2]
573: (8)              random.seed(self.seed)
574: (8)              actual = random.permutation(integer_val)
575: (8)              assert_array_equal(actual, desired)
576: (4)            def test_beta(self):
577: (8)              random.seed(self.seed)
578: (8)              actual = random.beta(.1, .9, size=(3, 2))
579: (8)              desired = np.array(
580: (16)                  [[1.45341850513746058e-02, 5.31297615662868145e-04],
581: (17)                  [1.85366619058432324e-06, 4.19214516800110563e-03],
582: (17)                  [1.58405155108498093e-04, 1.26252891949397652e-04]])
583: (8)              assert_array_almost_equal(actual, desired, decimal=15)
584: (4)            def test_binomial(self):
585: (8)              random.seed(self.seed)
586: (8)              actual = random.binomial(100.123, .456, size=(3, 2))
587: (8)              desired = np.array([[37, 43],
588: (28)                            [42, 48],
589: (28)                            [46, 45]])
590: (8)              assert_array_equal(actual, desired)

```

```

591: (8)             random.seed(self.seed)
592: (8)             actual = random.binomial(100.123, .456)
593: (8)             desired = 37
594: (8)             assert_array_equal(actual, desired)
595: (4)             def test_chisquare(self):
596: (8)                 random.seed(self.seed)
597: (8)                 actual = random.chisquare(50, size=(3, 2))
598: (8)                 desired = np.array([[63.87858175501090585, 68.68407748911370447],
599: (28)                               [65.77116116901505904, 47.09686762438974483],
600: (28)                               [72.3828403199695174, 74.18408615260374006]])
601: (8)                 assert_array_almost_equal(actual, desired, decimal=13)
602: (4)             def test_dirichlet(self):
603: (8)                 random.seed(self.seed)
604: (8)                 alpha = np.array([51.72840233779265162, 39.74494232180943953])
605: (8)                 actual = random.dirichlet(alpha, size=(3, 2))
606: (8)                 desired = np.array([[0.54539444573611562, 0.45460555426388438],
607: (29)                               [0.62345816822039413, 0.37654183177960598]],
608: (28)                               [[0.55206000085785778, 0.44793999914214233],
609: (29)                               [0.58964023305154301, 0.41035976694845688]],
610: (28)                               [[0.59266909280647828, 0.40733090719352177],
611: (29)                               [0.56974431743975207, 0.43025568256024799]]])
612: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
613: (8)                 bad_alpha = np.array([5.4e-01, -1.0e-16])
614: (8)                 assert_raises(ValueError, random.dirichlet, bad_alpha)
615: (8)                 random.seed(self.seed)
616: (8)                 alpha = np.array([51.72840233779265162, 39.74494232180943953])
617: (8)                 actual = random.dirichlet(alpha)
618: (8)                 assert_array_almost_equal(actual, desired[0, 0], decimal=15)
619: (4)             def test_dirichlet_size(self):
620: (8)                 p = np.array([51.72840233779265162, 39.74494232180943953])
621: (8)                 assert_equal(random.dirichlet(p, np.uint32(1)).shape, (1, 2))
622: (8)                 assert_equal(random.dirichlet(p, np.uint32(1)).shape, (1, 2))
623: (8)                 assert_equal(random.dirichlet(p, np.uint32(1)).shape, (1, 2))
624: (8)                 assert_equal(random.dirichlet(p, [2, 2]).shape, (2, 2, 2))
625: (8)                 assert_equal(random.dirichlet(p, (2, 2)).shape, (2, 2, 2))
626: (8)                 assert_equal(random.dirichlet(p, np.array((2, 2))).shape, (2, 2, 2))
627: (8)                 assert_raises(TypeError, random.dirichlet, p, float(1))
628: (4)             def test_dirichlet_bad_alpha(self):
629: (8)                 alpha = np.array([5.4e-01, -1.0e-16])
630: (8)                 assert_raises(ValueError, random.dirichlet, alpha)
631: (4)             def test_dirichlet_alpha_non_contiguous(self):
632: (8)                 a = np.array([51.72840233779265162, -1.0, 39.74494232180943953])
633: (8)                 alpha = a[::-2]
634: (8)                 random.seed(self.seed)
635: (8)                 non_contig = random.dirichlet(alpha, size=(3, 2))
636: (8)                 random.seed(self.seed)
637: (8)                 contig = random.dirichlet(np.ascontiguousarray(alpha),
638: (34)                               size=(3, 2))
639: (8)                 assert_array_almost_equal(non_contig, contig)
640: (4)             def test_exponential(self):
641: (8)                 random.seed(self.seed)
642: (8)                 actual = random.exponential(1.1234, size=(3, 2))
643: (8)                 desired = np.array([[1.08342649775011624, 1.00607889924557314],
644: (28)                               [2.46628830085216721, 2.49668106809923884],
645: (28)                               [0.68717433461363442, 1.6917566993575979]])
646: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
647: (4)             def test_exponential_0(self):
648: (8)                 assert_equal(random.exponential(scale=0), 0)
649: (8)                 assert_raises(ValueError, random.exponential, scale=-0.)
650: (4)             def test_f(self):
651: (8)                 random.seed(self.seed)
652: (8)                 actual = random.f(12, 77, size=(3, 2))
653: (8)                 desired = np.array([[1.21975394418575878, 1.75135759791559775],
654: (28)                               [1.44803115017146489, 1.22108959480396262],
655: (28)                               [1.02176975757740629, 1.34431827623300415]])
656: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
657: (4)             def test_gamma(self):
658: (8)                 random.seed(self.seed)
659: (8)                 actual = random.gamma(5, 3, size=(3, 2))

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

660: (8)             desired = np.array([[24.60509188649287182, 28.54993563207210627],
661: (28)           [26.13476110204064184, 12.56988482927716078],
662: (28)           [31.71863275789960568, 33.30143302795922011]])
663: (8)             assert_array_almost_equal(actual, desired, decimal=14)
664: (4) def test_gamma_0(self):
665: (8)     assert_equal(random.gamma(shape=0, scale=0), 0)
666: (8)     assert_raises(ValueError, random.gamma, shape=-0., scale=-0.)
667: (4) def test_geometric(self):
668: (8)     random.seed(self.seed)
669: (8)     actual = random.geometric(.123456789, size=(3, 2))
670: (8)     desired = np.array([[8, 7],
671: (28)       [17, 17],
672: (28)       [5, 12]])
673: (8)     assert_array_equal(actual, desired)
674: (4) def test_geometric_exceptions(self):
675: (8)     assert_raises(ValueError, random.geometric, 1.1)
676: (8)     assert_raises(ValueError, random.geometric, [1.1] * 10)
677: (8)     assert_raises(ValueError, random.geometric, -0.1)
678: (8)     assert_raises(ValueError, random.geometric, [-0.1] * 10)
679: (8)     with suppress_warnings() as sup:
680: (12)         sup.record(RuntimeWarning)
681: (12)         assert_raises(ValueError, random.geometric, np.nan)
682: (12)         assert_raises(ValueError, random.geometric, [np.nan] * 10)
683: (4) def test_gumbel(self):
684: (8)     random.seed(self.seed)
685: (8)     actual = random.gumbel(loc=.123456789, scale=2.0, size=(3, 2))
686: (8)     desired = np.array([[0.19591898743416816, 0.34405539668096674],
687: (28)       [-1.4492522252274278, -1.47374816298446865],
688: (28)       [1.10651090478803416, -0.69535848626236174]])
689: (8)     assert_array_almost_equal(actual, desired, decimal=15)
690: (4) def test_gumbel_0(self):
691: (8)     assert_equal(random.gumbel(scale=0), 0)
692: (8)     assert_raises(ValueError, random.gumbel, scale=-0.)
693: (4) def test_hypergeometric(self):
694: (8)     random.seed(self.seed)
695: (8)     actual = random.hypergeometric(10.1, 5.5, 14, size=(3, 2))
696: (8)     desired = np.array([[10, 10],
697: (28)       [10, 10],
698: (28)       [9, 9]])
699: (8)     assert_array_equal(actual, desired)
700: (8)     actual = random.hypergeometric(5, 0, 3, size=4)
701: (8)     desired = np.array([3, 3, 3])
702: (8)     assert_array_equal(actual, desired)
703: (8)     actual = random.hypergeometric(15, 0, 12, size=4)
704: (8)     desired = np.array([12, 12, 12, 12])
705: (8)     assert_array_equal(actual, desired)
706: (8)     actual = random.hypergeometric(0, 5, 3, size=4)
707: (8)     desired = np.array([0, 0, 0])
708: (8)     assert_array_equal(actual, desired)
709: (8)     actual = random.hypergeometric(0, 15, 12, size=4)
710: (8)     desired = np.array([0, 0, 0])
711: (8)     assert_array_equal(actual, desired)
712: (4) def test_laplace(self):
713: (8)     random.seed(self.seed)
714: (8)     actual = random.laplace(loc=.123456789, scale=2.0, size=(3, 2))
715: (8)     desired = np.array([[0.66599721112760157, 0.52829452552221945],
716: (28)       [3.12791959514407125, 3.18202813572992005],
717: (28)       [-0.05391065675859356, 1.74901336242837324]])
718: (8)     assert_array_almost_equal(actual, desired, decimal=15)
719: (4) def test_laplace_0(self):
720: (8)     assert_equal(random.laplace(scale=0), 0)
721: (8)     assert_raises(ValueError, random.laplace, scale=-0.)
722: (4) def test_logistic(self):
723: (8)     random.seed(self.seed)
724: (8)     actual = random.logistic(loc=.123456789, scale=2.0, size=(3, 2))
725: (8)     desired = np.array([[1.09232835305011444, 0.8648196662399954],
726: (28)       [4.27818590694950185, 4.33897006346929714],
727: (28)       [-0.21682183359214885, 2.63373365386060332]])
728: (8)     assert_array_almost_equal(actual, desired, decimal=15)

```

```

729: (4)
730: (8)
731: (8)
732: (8)
733: (28)
734: (28)
735: (8)
736: (4)
737: (8)
738: (8)
739: (4)
740: (8)
741: (8)
742: (8)
743: (28)
744: (28)
745: (8)
746: (4)
747: (8)
748: (4)
749: (4)
750: (8)
751: (12)
752: (16)
753: (12)
754: (16)
755: (12)
756: (16)
757: (4)
758: (8)
759: (8)
760: (8)
761: (29)
762: (28)
763: (29)
764: (28)
765: (29)
766: (8)
767: (4)
768: (8)
769: (8)
770: (8)
771: (8)
772: (8)
773: (8)
774: (29)
775: (28)
776: (29)
777: (28)
778: (29)
779: (8)
780: (8)
781: (8)
782: (8)
783: (8)
784: (8)
785: (8)
786: (8)
787: (27)
788: (8)
789: (22)
790: (8)
791: (8)
792: (12)
793: (12)
794: (12)
795: (8)
796: (8)
797: (8)

        def test_lognormal(self):
            random.seed(self.seed)
            actual = random.lognormal(mean=.123456789, sigma=2.0, size=(3, 2))
            desired = np.array([[16.50698631688883822, 36.54846706092654784],
                               [22.67886599981281748, 0.71617561058995771],
                               [65.72798501792723869, 86.84341601437161273]])
            assert_array_almost_equal(actual, desired, decimal=13)
        def test_lognormal_0(self):
            assert_equal(random.lognormal(sigma=0), 1)
            assert_raises(ValueError, random.lognormal, sigma=-0.)
        def test_logseries(self):
            random.seed(self.seed)
            actual = random.logseries(p=.923456789, size=(3, 2))
            desired = np.array([[2, 2],
                               [6, 17],
                               [3, 6]])
            assert_array_equal(actual, desired)
        def test_logseries_zero(self):
            assert random.logseries(0) == 1
@ pytest.mark.parametrize("value", [np.nextafter(0., -1), 1., np.nan, 5.])
        def test_logseries_exceptions(self, value):
            with np.errstate(invalid="ignore"):
                with pytest.raises(ValueError):
                    random.logseries(value)
                with pytest.raises(ValueError):
                    random.logseries(np.array([value] * 10))
                with pytest.raises(ValueError):
                    random.logseries(np.array([value] * 10)[:2])
        def test_multinomial(self):
            random.seed(self.seed)
            actual = random.multinomial(20, [1 / 6.] * 6, size=(3, 2))
            desired = np.array([[4, 3, 5, 4, 2, 2],
                               [5, 2, 8, 2, 2, 1],
                               [3, 4, 3, 6, 0, 4],
                               [2, 1, 4, 3, 6, 4],
                               [4, 4, 2, 5, 2, 3],
                               [4, 3, 4, 2, 3, 4]])
            assert_array_equal(actual, desired)
        def test_multivariate_normal(self):
            random.seed(self.seed)
            mean = (.123456789, 10)
            cov = [[1, 0], [0, 1]]
            size = (3, 2)
            actual = random.multivariate_normal(mean, cov, size)
            desired = np.array([[1.463620246718631, 11.73759122771936],
                               [1.622445133300628, 9.771356667546383],
                               [2.154490787682787, 12.170324946056553],
                               [1.719909438201865, 9.230548443648306],
                               [0.689515026297799, 9.880729819607714],
                               [-0.023054015651998, 9.201096623542879]])
            assert_array_almost_equal(actual, desired, decimal=15)
            actual = random.multivariate_normal(mean, cov)
            desired = np.array([0.895289569463708, 9.17180864067987])
            assert_array_almost_equal(actual, desired, decimal=15)
            mean = [0, 0]
            cov = [[1, 2], [2, 1]]
            assert_warns(RuntimeWarning, random.multivariate_normal, mean, cov)
            assert_no_warnings(random.multivariate_normal, mean, cov,
                               check_valid='ignore')
            assert_raises(ValueError, random.multivariate_normal, mean, cov,
                          check_valid='raise')
            cov = np.array([[1, 0.1], [0.1, 1]], dtype=np.float32)
            with suppress_warnings() as sup:
                random.multivariate_normal(mean, cov)
                w = sup.record(RuntimeWarning)
                assert len(w) == 0
            mu = np.zeros(2)
            cov = np.eye(2)
            assert_raises(ValueError, random.multivariate_normal, mean, cov,

```

```

798: (22)                                check_valid='other')
799: (8)       assert_raises(ValueError, random.multivariate_normal,
800: (22)                           np.zeros((2, 1, 1)), cov)
801: (8)       assert_raises(ValueError, random.multivariate_normal,
802: (22)                           mu, np.empty((3, 2)))
803: (8)       assert_raises(ValueError, random.multivariate_normal,
804: (22)                           mu, np.eye(3))
805: (4) def test_negative_binomial(self):
806: (8)     random.seed(self.seed)
807: (8)     actual = random.negative_binomial(n=100, p=.12345, size=(3, 2))
808: (8)     desired = np.array([[848, 841],
809: (28)                     [892, 611],
810: (28)                     [779, 647]])
811: (8)     assert_array_equal(actual, desired)
812: (4) def test_negative_binomial_exceptions(self):
813: (8)     with suppress_warnings() as sup:
814: (12)         sup.record(RuntimeWarning)
815: (12)         assert_raises(ValueError, random.negative_binomial, 100, np.nan)
816: (12)         assert_raises(ValueError, random.negative_binomial, 100,
817: (26)                           [np.nan] * 10)
818: (4) def test_noncentral_chisquare(self):
819: (8)     random.seed(self.seed)
820: (8)     actual = random.noncentral_chisquare(df=5, nonc=5, size=(3, 2))
821: (8)     desired = np.array([[23.91905354498517511, 13.35324692733826346],
822: (28)                     [31.22452661329736401, 16.60047399466177254],
823: (28)                     [5.03461598262724586, 17.94973089023519464]])
824: (8)     assert_array_almost_equal(actual, desired, decimal=14)
825: (8)     actual = random.noncentral_chisquare(df=.5, nonc=.2, size=(3, 2))
826: (8)     desired = np.array([[1.47145377828516666, 0.15052899268012659],
827: (28)                     [0.00943803056963588, 1.02647251615666169],
828: (28)                     [0.332334982684171, 0.15451287602753125]])
829: (8)     assert_array_almost_equal(actual, desired, decimal=14)
830: (8)     random.seed(self.seed)
831: (8)     actual = random.noncentral_chisquare(df=5, nonc=0, size=(3, 2))
832: (8)     desired = np.array([[9.597154162763948, 11.725484450296079],
833: (28)                     [10.413711048138335, 3.694475922923986],
834: (28)                     [13.484222138963087, 14.377255424602957]])
835: (8)     assert_array_almost_equal(actual, desired, decimal=14)
836: (4) def test_noncentral_f(self):
837: (8)     random.seed(self.seed)
838: (8)     actual = random.noncentral_f(dfnum=5, dfden=2, nonc=1,
839: (37)                     size=(3, 2))
840: (8)     desired = np.array([[1.40598099674926669, 0.34207973179285761],
841: (28)                     [3.57715069265772545, 7.92632662577829805],
842: (28)                     [0.43741599463544162, 1.1774208752428319]])
843: (8)     assert_array_almost_equal(actual, desired, decimal=14)
844: (4) def test_noncentral_f_nan(self):
845: (8)     random.seed(self.seed)
846: (8)     actual = random.noncentral_f(dfnum=5, dfden=2, nonc=np.nan)
847: (8)     assert np.isnan(actual)
848: (4) def test_normal(self):
849: (8)     random.seed(self.seed)
850: (8)     actual = random.normal(loc=.123456789, scale=2.0, size=(3, 2))
851: (8)     desired = np.array([[2.80378370443726244, 3.59863924443872163],
852: (28)                     [3.121433477601256, -0.33382987590723379],
853: (28)                     [4.18552478636557357, 4.46410668111310471]])
854: (8)     assert_array_almost_equal(actual, desired, decimal=15)
855: (4) def test_normal_0(self):
856: (8)     assert_equal(random.normal(scale=0), 0)
857: (8)     assert_raises(ValueError, random.normal, scale=-0.)
858: (4) def test_pareto(self):
859: (8)     random.seed(self.seed)
860: (8)     actual = random.pareto(a=.123456789, size=(3, 2))
861: (8)     desired = np.array(
862: (16)         [[2.46852460439034849e+03, 1.41286880810518346e+03],
863: (17)                     [5.28287797029485181e+07, 6.57720981047328785e+07],
864: (17)                     [1.40840323350391515e+02, 1.98390255135251704e+05]])
865: (8)     np.testing.assert_array_almost_equal(actual, desired, nulp=30)
866: (4) def test_poisson(self):

```

```

867: (8)           random.seed(self.seed)
868: (8)           actual = random.poisson(lam=.123456789, size=(3, 2))
869: (8)           desired = np.array([[0, 0],
870: (28)                 [1, 0],
871: (28)                 [0, 0]])
872: (8)           assert_array_equal(actual, desired)
873: (4) def test_poisson_exceptions(self):
874: (8)     lambig = np.iinfo('l').max
875: (8)     lamneg = -1
876: (8)     assert_raises(ValueError, random.poisson, lambig)
877: (8)     assert_raises(ValueError, random.poisson, [lamneg] * 10)
878: (8)     assert_raises(ValueError, random.poisson, lambig)
879: (8)     assert_raises(ValueError, random.poisson, [lambig] * 10)
880: (8)     with suppress_warnings() as sup:
881: (12)         sup.record(RuntimeWarning)
882: (12)         assert_raises(ValueError, random.poisson, np.nan)
883: (12)         assert_raises(ValueError, random.poisson, [np.nan] * 10)
884: (4) def test_power(self):
885: (8)     random.seed(self.seed)
886: (8)     actual = random.power(a=.123456789, size=(3, 2))
887: (8)     desired = np.array([[0.02048932883240791, 0.01424192241128213],
888: (28)                   [0.38446073748535298, 0.39499689943484395],
889: (28)                   [0.00177699707563439, 0.13115505880863756]])
890: (8)     assert_array_almost_equal(actual, desired, decimal=15)
891: (4) def test_rayleigh(self):
892: (8)     random.seed(self.seed)
893: (8)     actual = random.rayleigh(scale=10, size=(3, 2))
894: (8)     desired = np.array([[13.8882496494248393, 13.383318339044731],
895: (28)                   [20.95413364294492098, 21.08285015800712614],
896: (28)                   [11.06066537006854311, 17.35468505778271009]])
897: (8)     assert_array_almost_equal(actual, desired, decimal=14)
898: (4) def test_rayleigh_0(self):
899: (8)     assert_equal(random.rayleigh(scale=0), 0)
900: (8)     assert_raises(ValueError, random.rayleigh, scale=-0.)
901: (4) def test_standard_cauchy(self):
902: (8)     random.seed(self.seed)
903: (8)     actual = random.standard_cauchy(size=(3, 2))
904: (8)     desired = np.array([[0.77127660196445336, -6.55601161955910605],
905: (28)                   [0.93582023391158309, -2.07479293013759447],
906: (28)                   [-4.74601644297011926, 0.18338989290760804]])
907: (8)     assert_array_almost_equal(actual, desired, decimal=15)
908: (4) def test_standard_exponential(self):
909: (8)     random.seed(self.seed)
910: (8)     actual = random.standard_exponential(size=(3, 2))
911: (8)     desired = np.array([[0.96441739162374596, 0.89556604882105506],
912: (28)                   [2.1953785836319808, 2.22243285392490542],
913: (28)                   [0.6116915921431676, 1.50592546727413201]])
914: (8)     assert_array_almost_equal(actual, desired, decimal=15)
915: (4) def test_standard_gamma(self):
916: (8)     random.seed(self.seed)
917: (8)     actual = random.standard_gamma(shape=3, size=(3, 2))
918: (8)     desired = np.array([[5.50841531318455058, 6.62953470301903103],
919: (28)                   [5.93988484943779227, 2.31044849402133989],
920: (28)                   [7.54838614231317084, 8.012756093271868]])
921: (8)     assert_array_almost_equal(actual, desired, decimal=14)
922: (4) def test_standard_gamma_0(self):
923: (8)     assert_equal(random.standard_gamma(shape=0), 0)
924: (8)     assert_raises(ValueError, random.standard_gamma, shape=-0.)
925: (4) def test_standard_normal(self):
926: (8)     random.seed(self.seed)
927: (8)     actual = random.standard_normal(size=(3, 2))
928: (8)     desired = np.array([[1.34016345771863121, 1.73759122771936081],
929: (28)                   [1.498988344300628, -0.2286433324536169],
930: (28)                   [2.031033998682787, 2.17032494605655257]])
931: (8)     assert_array_almost_equal(actual, desired, decimal=15)
932: (4) def test_randn_singleton(self):
933: (8)     random.seed(self.seed)
934: (8)     actual = random.randn()
935: (8)     desired = np.array(1.34016345771863121)

```

```

936: (8)             assert_array_almost_equal(actual, desired, decimal=15)
937: (4)             def test_standard_t(self):
938: (8)                 random.seed(self.seed)
939: (8)                 actual = random.standard_t(df=10, size=(3, 2))
940: (8)                 desired = np.array([[0.97140611862659965, -0.08830486548450577],
941: (28)                               [1.36311143689505321, -0.55317463909867071],
942: (28)                               [-0.18473749069684214, 0.61181537341755321]]))
943: (8)             assert_array_almost_equal(actual, desired, decimal=15)
944: (4)             def test_triangular(self):
945: (8)                 random.seed(self.seed)
946: (8)                 actual = random.triangular(left=5.12, mode=10.23, right=20.34,
947: (35)                               size=(3, 2))
948: (8)                 desired = np.array([[12.68117178949215784, 12.4129206149193152],
949: (28)                               [16.20131377335158263, 16.25692138747600524],
950: (28)                               [11.20400690911820263, 14.4978144835829923]]))
951: (8)             assert_array_almost_equal(actual, desired, decimal=14)
952: (4)             def test_uniform(self):
953: (8)                 random.seed(self.seed)
954: (8)                 actual = random.uniform(low=1.23, high=10.54, size=(3, 2))
955: (8)                 desired = np.array([[6.99097932346268003, 6.73801597444323974],
956: (28)                               [9.50364421400426274, 9.53130618907631089],
957: (28)                               [5.48995325769805476, 8.47493103280052118]]))
958: (8)             assert_array_almost_equal(actual, desired, decimal=15)
959: (4)             def test_uniform_range_bounds(self):
960: (8)                 fmin = np.finfo('float').min
961: (8)                 fmax = np.finfo('float').max
962: (8)                 func = random.uniform
963: (8)                 assert_raises(OverflowError, func, -np.inf, 0)
964: (8)                 assert_raises(OverflowError, func, 0, np.inf)
965: (8)                 assert_raises(OverflowError, func, fmin, fmax)
966: (8)                 assert_raises(OverflowError, func, [-np.inf], [0])
967: (8)                 assert_raises(OverflowError, func, [0], [np.inf])
968: (8)                 random.uniform(low=np.nextafter(fmin, 1), high=fmax / 1e17)
969: (4)             def test_scalar_exception_propagation(self):
970: (8)                 class ThrowingFloat(np.ndarray):
971: (12)                     def __float__(self):
972: (16)                         raise TypeError
973: (8)                     throwing_float = np.array(1.0).view(ThrowingFloat)
974: (8)                     assert_raises(TypeError, random.uniform, throwing_float,
975: (22)                           throwing_float)
976: (8)                     class ThrowingInteger(np.ndarray):
977: (12)                         def __int__(self):
978: (16)                             raise TypeError
979: (8)                         throwing_int = np.array(1).view(ThrowingInteger)
980: (8)                         assert_raises(TypeError, random.hypergeometric, throwing_int, 1, 1)
981: (4)             def test_vonmises(self):
982: (8)                 random.seed(self.seed)
983: (8)                 actual = random.vonmises(mu=1.23, kappa=1.54, size=(3, 2))
984: (8)                 desired = np.array([[2.28567572673902042, 2.89163838442285037],
985: (28)                               [0.38198375564286025, 2.57638023113890746],
986: (28)                               [1.19153771588353052, 1.83509849681825354]]))
987: (8)                 assert_array_almost_equal(actual, desired, decimal=15)
988: (4)             def test_vonmises_small(self):
989: (8)                 random.seed(self.seed)
990: (8)                 r = random.vonmises(mu=0., kappa=1.1e-8, size=10**6)
991: (8)                 assert_(np.isfinite(r).all())
992: (4)             def test_vonmises_large(self):
993: (8)                 random.seed(self.seed)
994: (8)                 actual = random.vonmises(mu=0., kappa=1e7, size=3)
995: (8)                 desired = np.array([4.63425374852111e-04,
996: (28)                               3.558873596114509e-04,
997: (28)                               -2.337119622577433e-04])
998: (8)                 assert_array_almost_equal(actual, desired, decimal=8)
999: (4)             def test_vonmises_nan(self):
1000: (8)                 random.seed(self.seed)
1001: (8)                 r = random.vonmises(mu=0., kappa=np.nan)
1002: (8)                 assert_(np.isnan(r))
1003: (4)             def test_wald(self):
1004: (8)                 random.seed(self.seed)

```

```

1005: (8)           actual = random.wald(mean=1.23, scale=1.54, size=(3, 2))
1006: (8)           desired = np.array([[3.82935265715889983, 5.13125249184285526],
1007: (28)          [0.35045403618358717, 1.50832396872003538],
1008: (28)          [0.24124319895843183, 0.22031101461955038]])]
1009: (8)           assert_array_almost_equal(actual, desired, decimal=14)
1010: (4)           def test_weibull(self):
1011: (8)             random.seed(self.seed)
1012: (8)             actual = random.weibull(a=1.23, size=(3, 2))
1013: (8)             desired = np.array([[0.97097342648766727, 0.91422896443565516],
1014: (28)            [1.89517770034962929, 1.91414357960479564],
1015: (28)            [0.67057783752390987, 1.39494046635066793]])]
1016: (8)             assert_array_almost_equal(actual, desired, decimal=15)
1017: (4)           def test_weibull_0(self):
1018: (8)             random.seed(self.seed)
1019: (8)             assert_equal(random.weibull(a=0, size=12), np.zeros(12))
1020: (8)             assert_raises(ValueError, random.weibull, a=-0.)
1021: (4)           def test_zipf(self):
1022: (8)             random.seed(self.seed)
1023: (8)             actual = random.zipf(a=1.23, size=(3, 2))
1024: (8)             desired = np.array([[66, 29],
1025: (28)            [1, 1],
1026: (28)            [3, 13]])]
1027: (8)             assert_array_equal(actual, desired)
1028: (0)           class TestBroadcast:
1029: (4)             def setup_method(self):
1030: (8)               self.seed = 123456789
1031: (4)             def set_seed(self):
1032: (8)               random.seed(self.seed)
1033: (4)             def test_uniform(self):
1034: (8)               low = [0]
1035: (8)               high = [1]
1036: (8)               uniform = random.uniform
1037: (8)               desired = np.array([0.53283302478975902,
1038: (28)                 0.53413660089041659,
1039: (28)                 0.50955303552646702])
1040: (8)               self.set_seed()
1041: (8)               actual = uniform(low * 3, high)
1042: (8)               assert_array_almost_equal(actual, desired, decimal=14)
1043: (8)               self.set_seed()
1044: (8)               actual = uniform(low, high * 3)
1045: (8)               assert_array_almost_equal(actual, desired, decimal=14)
1046: (4)             def test_normal(self):
1047: (8)               loc = [0]
1048: (8)               scale = [1]
1049: (8)               bad_scale = [-1]
1050: (8)               normal = random.normal
1051: (8)               desired = np.array([2.2129019979039612,
1052: (28)                 2.1283977976520019,
1053: (28)                 1.8417114045748335])
1054: (8)               self.set_seed()
1055: (8)               actual = normal(loc * 3, scale)
1056: (8)               assert_array_almost_equal(actual, desired, decimal=14)
1057: (8)               assert_raises(ValueError, normal, loc * 3, bad_scale)
1058: (8)               self.set_seed()
1059: (8)               actual = normal(loc, scale * 3)
1060: (8)               assert_array_almost_equal(actual, desired, decimal=14)
1061: (8)               assert_raises(ValueError, normal, loc, bad_scale * 3)
1062: (4)             def test_beta(self):
1063: (8)               a = [1]
1064: (8)               b = [2]
1065: (8)               bad_a = [-1]
1066: (8)               bad_b = [-2]
1067: (8)               beta = random.beta
1068: (8)               desired = np.array([0.19843558305989056,
1069: (28)                 0.075230336409423643,
1070: (28)                 0.24976865978980844])
1071: (8)               self.set_seed()
1072: (8)               actual = beta(a * 3, b)
1073: (8)               assert_array_almost_equal(actual, desired, decimal=14)

```

```

1074: (8)             assert_raises(ValueError, beta, bad_a * 3, b)
1075: (8)             assert_raises(ValueError, beta, a * 3, bad_b)
1076: (8)             self.set_seed()
1077: (8)             actual = beta(a, b * 3)
1078: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1079: (8)             assert_raises(ValueError, beta, bad_a, b * 3)
1080: (8)             assert_raises(ValueError, beta, a, bad_b * 3)
1081: (4)             def test_exponential(self):
1082: (8)                 scale = [1]
1083: (8)                 bad_scale = [-1]
1084: (8)                 exponential = random.exponential
1085: (8)                 desired = np.array([0.76106853658845242,
1086: (28)                     0.76386282278691653,
1087: (28)                     0.71243813125891797])
1088: (8)                 self.set_seed()
1089: (8)                 actual = exponential(scale * 3)
1090: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1091: (8)                 assert_raises(ValueError, exponential, bad_scale * 3)
1092: (4)             def test_standard_gamma(self):
1093: (8)                 shape = [1]
1094: (8)                 bad_shape = [-1]
1095: (8)                 std_gamma = random.standard_gamma
1096: (8)                 desired = np.array([0.76106853658845242,
1097: (28)                     0.76386282278691653,
1098: (28)                     0.71243813125891797])
1099: (8)                 self.set_seed()
1100: (8)                 actual = std_gamma(shape * 3)
1101: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1102: (8)                 assert_raises(ValueError, std_gamma, bad_shape * 3)
1103: (4)             def test_gamma(self):
1104: (8)                 shape = [1]
1105: (8)                 scale = [2]
1106: (8)                 bad_shape = [-1]
1107: (8)                 bad_scale = [-2]
1108: (8)                 gamma = random.gamma
1109: (8)                 desired = np.array([1.5221370731769048,
1110: (28)                     1.5277256455738331,
1111: (28)                     1.4248762625178359])
1112: (8)                 self.set_seed()
1113: (8)                 actual = gamma(shape * 3, scale)
1114: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1115: (8)                 assert_raises(ValueError, gamma, bad_shape * 3, scale)
1116: (8)                 assert_raises(ValueError, gamma, shape * 3, bad_scale)
1117: (8)                 self.set_seed()
1118: (8)                 actual = gamma(shape, scale * 3)
1119: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1120: (8)                 assert_raises(ValueError, gamma, bad_shape, scale * 3)
1121: (8)                 assert_raises(ValueError, gamma, shape, bad_scale * 3)
1122: (4)             def test_f(self):
1123: (8)                 dfnum = [1]
1124: (8)                 dfden = [2]
1125: (8)                 bad_dfnum = [-1]
1126: (8)                 bad_dfden = [-2]
1127: (8)                 f = random.f
1128: (8)                 desired = np.array([0.80038951638264799,
1129: (28)                     0.86768719635363512,
1130: (28)                     2.7251095168386801])
1131: (8)                 self.set_seed()
1132: (8)                 actual = f(dfnum * 3, dfden)
1133: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1134: (8)                 assert_raises(ValueError, f, bad_dfnum * 3, dfden)
1135: (8)                 assert_raises(ValueError, f, dfnum * 3, bad_dfden)
1136: (8)                 self.set_seed()
1137: (8)                 actual = f(dfnum, dfden * 3)
1138: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1139: (8)                 assert_raises(ValueError, f, bad_dfnum, dfden * 3)
1140: (8)                 assert_raises(ValueError, f, dfnum, bad_dfden * 3)
1141: (4)             def test_noncentral_f(self):
1142: (8)                 dfnum = [2]

```

```

1143: (8)         dfden = [3]
1144: (8)         nonc = [4]
1145: (8)         bad_dfnum = [0]
1146: (8)         bad_dfden = [-1]
1147: (8)         bad_nonc = [-2]
1148: (8)         nonc_f = random.noncentral_f
1149: (8)         desired = np.array([9.1393943263705211,
1150: (28)             13.025456344595602,
1151: (28)             8.8018098359100545])
1152: (8)         self.set_seed()
1153: (8)         actual = nonc_f(dfnum * 3, dfden, nonc)
1154: (8)         assert_array_almost_equal(actual, desired, decimal=14)
1155: (8)         assert np.all(np.isnan(nonc_f(dfnum, dfden, [np.nan] * 3)))
1156: (8)         assert_raises(ValueError, nonc_f, bad_dfnum * 3, dfden, nonc)
1157: (8)         assert_raises(ValueError, nonc_f, dfnum * 3, bad_dfden, nonc)
1158: (8)         assert_raises(ValueError, nonc_f, dfnum * 3, dfden, bad_nonc)
1159: (8)         self.set_seed()
1160: (8)         actual = nonc_f(dfnum, dfden * 3, nonc)
1161: (8)         assert_array_almost_equal(actual, desired, decimal=14)
1162: (8)         assert_raises(ValueError, nonc_f, bad_dfnum, dfden * 3, nonc)
1163: (8)         assert_raises(ValueError, nonc_f, dfnum, bad_dfden * 3, nonc)
1164: (8)         assert_raises(ValueError, nonc_f, dfnum, dfden * 3, bad_nonc)
1165: (8)         self.set_seed()
1166: (8)         actual = nonc_f(dfnum, dfden, nonc * 3)
1167: (8)         assert_array_almost_equal(actual, desired, decimal=14)
1168: (8)         assert_raises(ValueError, nonc_f, bad_dfnum, dfden, nonc * 3)
1169: (8)         assert_raises(ValueError, nonc_f, dfnum, bad_dfden, nonc * 3)
1170: (8)         assert_raises(ValueError, nonc_f, dfnum, dfden, bad_nonc * 3)
1171: (4)         def test_noncentral_f_small_df(self):
1172: (8)             self.set_seed()
1173: (8)             desired = np.array([6.869638627492048, 0.785880199263955])
1174: (8)             actual = random.noncentral_f(0.9, 0.9, 2, size=2)
1175: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1176: (4)         def test_chisquare(self):
1177: (8)             df = [1]
1178: (8)             bad_df = [-1]
1179: (8)             chisquare = random.chisquare
1180: (8)             desired = np.array([0.57022801133088286,
1181: (28)                 0.51947702108840776,
1182: (28)                 0.1320969254923558])
1183: (8)             self.set_seed()
1184: (8)             actual = chisquare(df * 3)
1185: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1186: (8)             assert_raises(ValueError, chisquare, bad_df * 3)
1187: (4)         def test_noncentral_chisquare(self):
1188: (8)             df = [1]
1189: (8)             nonc = [2]
1190: (8)             bad_df = [-1]
1191: (8)             bad_nonc = [-2]
1192: (8)             nonc_chi = random.noncentral_chisquare
1193: (8)             desired = np.array([9.0015599467913763,
1194: (28)                 4.5804135049718742,
1195: (28)                 6.0872302432834564])
1196: (8)             self.set_seed()
1197: (8)             actual = nonc_chi(df * 3, nonc)
1198: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1199: (8)             assert_raises(ValueError, nonc_chi, bad_df * 3, nonc)
1200: (8)             assert_raises(ValueError, nonc_chi, df * 3, bad_nonc)
1201: (8)             self.set_seed()
1202: (8)             actual = nonc_chi(df, nonc * 3)
1203: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1204: (8)             assert_raises(ValueError, nonc_chi, bad_df, nonc * 3)
1205: (8)             assert_raises(ValueError, nonc_chi, df, bad_nonc * 3)
1206: (4)         def test_standard_t(self):
1207: (8)             df = [1]
1208: (8)             bad_df = [-1]
1209: (8)             t = random.standard_t
1210: (8)             desired = np.array([3.0702872575217643,
1211: (28)                 5.8560725167361607,

```

```

1212: (28)                                1.0274791436474273])
1213: (8)          self.set_seed()
1214: (8)          actual = t(df * 3)
1215: (8)          assert_array_almost_equal(actual, desired, decimal=14)
1216: (8)          assert_raises(ValueError, t, bad_df * 3)
1217: (8)          assert_raises(ValueError, random.standard_t, bad_df * 3)
1218: (4)          def test_vonmises(self):
1219: (8)              mu = [2]
1220: (8)              kappa = [1]
1221: (8)              bad_kappa = [-1]
1222: (8)              vonmises = random.vonmises
1223: (8)              desired = np.array([2.9883443664201312,
1224: (28)                  -2.7064099483995943,
1225: (28)                  -1.8672476700665914])
1226: (8)          self.set_seed()
1227: (8)          actual = vonmises(mu * 3, kappa)
1228: (8)          assert_array_almost_equal(actual, desired, decimal=14)
1229: (8)          assert_raises(ValueError, vonmises, mu * 3, bad_kappa)
1230: (8)          self.set_seed()
1231: (8)          actual = vonmises(mu, kappa * 3)
1232: (8)          assert_array_almost_equal(actual, desired, decimal=14)
1233: (8)          assert_raises(ValueError, vonmises, mu, bad_kappa * 3)
1234: (4)          def test_pareto(self):
1235: (8)              a = [1]
1236: (8)              bad_a = [-1]
1237: (8)              pareto = random.pareto
1238: (8)              desired = np.array([1.1405622680198362,
1239: (28)                  1.1465519762044529,
1240: (28)                  1.0389564467453547])
1241: (8)          self.set_seed()
1242: (8)          actual = pareto(a * 3)
1243: (8)          assert_array_almost_equal(actual, desired, decimal=14)
1244: (8)          assert_raises(ValueError, pareto, bad_a * 3)
1245: (8)          assert_raises(ValueError, random.pareto, bad_a * 3)
1246: (4)          def test_weibull(self):
1247: (8)              a = [1]
1248: (8)              bad_a = [-1]
1249: (8)              weibull = random.weibull
1250: (8)              desired = np.array([0.76106853658845242,
1251: (28)                  0.76386282278691653,
1252: (28)                  0.71243813125891797])
1253: (8)          self.set_seed()
1254: (8)          actual = weibull(a * 3)
1255: (8)          assert_array_almost_equal(actual, desired, decimal=14)
1256: (8)          assert_raises(ValueError, weibull, bad_a * 3)
1257: (8)          assert_raises(ValueError, random.weibull, bad_a * 3)
1258: (4)          def test_power(self):
1259: (8)              a = [1]
1260: (8)              bad_a = [-1]
1261: (8)              power = random.power
1262: (8)              desired = np.array([0.53283302478975902,
1263: (28)                  0.53413660089041659,
1264: (28)                  0.50955303552646702])
1265: (8)          self.set_seed()
1266: (8)          actual = power(a * 3)
1267: (8)          assert_array_almost_equal(actual, desired, decimal=14)
1268: (8)          assert_raises(ValueError, power, bad_a * 3)
1269: (8)          assert_raises(ValueError, random.power, bad_a * 3)
1270: (4)          def test_laplace(self):
1271: (8)              loc = [0]
1272: (8)              scale = [1]
1273: (8)              bad_scale = [-1]
1274: (8)              laplace = random.laplace
1275: (8)              desired = np.array([0.067921356028507157,
1276: (28)                  0.070715642226971326,
1277: (28)                  0.019290950698972624])
1278: (8)          self.set_seed()
1279: (8)          actual = laplace(loc * 3, scale)
1280: (8)          assert_array_almost_equal(actual, desired, decimal=14)

```

```

1281: (8)             assert_raises(ValueError, laplace, loc * 3, bad_scale)
1282: (8)             self.set_seed()
1283: (8)             actual = laplace(loc, scale * 3)
1284: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1285: (8)             assert_raises(ValueError, laplace, loc, bad_scale * 3)
1286: (4)             def test_gumbel(self):
1287: (8)                 loc = [0]
1288: (8)                 scale = [1]
1289: (8)                 bad_scale = [-1]
1290: (8)                 gumbel = random.gumbel
1291: (8)                 desired = np.array([0.2730318639556768,
1292: (28)                         0.26936705726291116,
1293: (28)                         0.33906220393037939])
1294: (8)                 self.set_seed()
1295: (8)                 actual = gumbel(loc * 3, scale)
1296: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1297: (8)                 assert_raises(ValueError, gumbel, loc * 3, bad_scale)
1298: (8)                 self.set_seed()
1299: (8)                 actual = gumbel(loc, scale * 3)
1300: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1301: (8)                 assert_raises(ValueError, gumbel, loc, bad_scale * 3)
1302: (4)             def test_logistic(self):
1303: (8)                 loc = [0]
1304: (8)                 scale = [1]
1305: (8)                 bad_scale = [-1]
1306: (8)                 logistic = random.logistic
1307: (8)                 desired = np.array([0.13152135837586171,
1308: (28)                     0.13675915696285773,
1309: (28)                     0.038216792802833396])
1310: (8)                 self.set_seed()
1311: (8)                 actual = logistic(loc * 3, scale)
1312: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1313: (8)                 assert_raises(ValueError, logistic, loc * 3, bad_scale)
1314: (8)                 self.set_seed()
1315: (8)                 actual = logistic(loc, scale * 3)
1316: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1317: (8)                 assert_raises(ValueError, logistic, loc, bad_scale * 3)
1318: (8)                 assert_equal(random.logistic(1.0, 0.0), 1.0)
1319: (4)             def test_lognormal(self):
1320: (8)                 mean = [0]
1321: (8)                 sigma = [1]
1322: (8)                 bad_sigma = [-1]
1323: (8)                 lognormal = random.lognormal
1324: (8)                 desired = np.array([9.1422086044848427,
1325: (28)                     8.4013952870126261,
1326: (28)                     6.3073234116578671])
1327: (8)                 self.set_seed()
1328: (8)                 actual = lognormal(mean * 3, sigma)
1329: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1330: (8)                 assert_raises(ValueError, lognormal, mean * 3, bad_sigma)
1331: (8)                 assert_raises(ValueError, random.lognormal, mean * 3, bad_sigma)
1332: (8)                 self.set_seed()
1333: (8)                 actual = lognormal(mean, sigma * 3)
1334: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1335: (8)                 assert_raises(ValueError, lognormal, mean, bad_sigma * 3)
1336: (8)                 assert_raises(ValueError, random.lognormal, mean, bad_sigma * 3)
1337: (4)             def test_rayleigh(self):
1338: (8)                 scale = [1]
1339: (8)                 bad_scale = [-1]
1340: (8)                 rayleigh = random.rayleigh
1341: (8)                 desired = np.array([1.2337491937897689,
1342: (28)                     1.2360119924878694,
1343: (28)                     1.1936818095781789])
1344: (8)                 self.set_seed()
1345: (8)                 actual = rayleigh(scale * 3)
1346: (8)                 assert_array_almost_equal(actual, desired, decimal=14)
1347: (8)                 assert_raises(ValueError, rayleigh, bad_scale * 3)
1348: (4)             def test_wald(self):
1349: (8)                 mean = [0.5]

```

```

1350: (8)             scale = [1]
1351: (8)             bad_mean = [0]
1352: (8)             bad_scale = [-2]
1353: (8)             wald = random.wald
1354: (8)             desired = np.array([0.11873681120271318,
1355: (28)                 0.12450084820795027,
1356: (28)                 0.9096122728408238])
1357: (8)             self.set_seed()
1358: (8)             actual = wald(mean * 3, scale)
1359: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1360: (8)             assert_raises(ValueError, wald, bad_mean * 3, scale)
1361: (8)             assert_raises(ValueError, wald, mean * 3, bad_scale)
1362: (8)             assert_raises(ValueError, random.wald, bad_mean * 3, scale)
1363: (8)             assert_raises(ValueError, random.wald, mean * 3, bad_scale)
1364: (8)             self.set_seed()
1365: (8)             actual = wald(mean, scale * 3)
1366: (8)             assert_array_almost_equal(actual, desired, decimal=14)
1367: (8)             assert_raises(ValueError, wald, bad_mean, scale * 3)
1368: (8)             assert_raises(ValueError, wald, mean, bad_scale * 3)
1369: (8)             assert_raises(ValueError, wald, 0.0, 1)
1370: (8)             assert_raises(ValueError, wald, 0.5, 0.0)
1371: (4) def test_triangular(self):
1372: (8)     left = [1]
1373: (8)     right = [3]
1374: (8)     mode = [2]
1375: (8)     bad_left_one = [3]
1376: (8)     bad_mode_one = [4]
1377: (8)     bad_left_two, bad_mode_two = right * 2
1378: (8)     triangular = random.triangular
1379: (8)     desired = np.array([2.03339048710429,
1380: (28)                 2.0347400359389356,
1381: (28)                 2.0095991069536208])
1382: (8)     self.set_seed()
1383: (8)     actual = triangular(left * 3, mode, right)
1384: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1385: (8)     assert_raises(ValueError, triangular, bad_left_one * 3, mode, right)
1386: (8)     assert_raises(ValueError, triangular, left * 3, bad_mode_one, right)
1387: (8)     assert_raises(ValueError, triangular, bad_left_two * 3, bad_mode_two,
1388: (22)                     right)
1389: (8)     self.set_seed()
1390: (8)     actual = triangular(left, mode * 3, right)
1391: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1392: (8)     assert_raises(ValueError, triangular, bad_left_one, mode * 3, right)
1393: (8)     assert_raises(ValueError, triangular, left, bad_mode_one * 3, right)
1394: (8)     assert_raises(ValueError, triangular, bad_left_two, bad_mode_two * 3,
1395: (22)                     right)
1396: (8)     self.set_seed()
1397: (8)     actual = triangular(left, mode, right * 3)
1398: (8)     assert_array_almost_equal(actual, desired, decimal=14)
1399: (8)     assert_raises(ValueError, triangular, bad_left_one, mode, right * 3)
1400: (8)     assert_raises(ValueError, triangular, left, bad_mode_one, right * 3)
1401: (8)     assert_raises(ValueError, triangular, bad_left_two, bad_mode_two,
1402: (22)                     right * 3)
1403: (8)     assert_raises(ValueError, triangular, 10., 0., 20.)
1404: (8)     assert_raises(ValueError, triangular, 10., 25., 20.)
1405: (8)     assert_raises(ValueError, triangular, 10., 10., 10.)
1406: (4) def test_binomial(self):
1407: (8)     n = [1]
1408: (8)     p = [0.5]
1409: (8)     bad_n = [-1]
1410: (8)     bad_p_one = [-1]
1411: (8)     bad_p_two = [1.5]
1412: (8)     binom = random.binomial
1413: (8)     desired = np.array([1, 1, 1])
1414: (8)     self.set_seed()
1415: (8)     actual = binom(n * 3, p)
1416: (8)     assert_array_equal(actual, desired)
1417: (8)     assert_raises(ValueError, binom, bad_n * 3, p)
1418: (8)     assert_raises(ValueError, binom, n * 3, bad_p_one)

```

```

1419: (8)             assert_raises(ValueError, binom, n * 3, bad_p_two)
1420: (8)             self.set_seed()
1421: (8)             actual = binom(n, p * 3)
1422: (8)             assert_array_equal(actual, desired)
1423: (8)             assert_raises(ValueError, binom, bad_n, p * 3)
1424: (8)             assert_raises(ValueError, binom, n, bad_p_one * 3)
1425: (8)             assert_raises(ValueError, binom, n, bad_p_two * 3)
1426: (4)             def test_negative_binomial(self):
1427: (8)                 n = [1]
1428: (8)                 p = [0.5]
1429: (8)                 bad_n = [-1]
1430: (8)                 bad_p_one = [-1]
1431: (8)                 bad_p_two = [1.5]
1432: (8)                 neg_binom = random.negative_binomial
1433: (8)                 desired = np.array([1, 0, 1])
1434: (8)                 self.set_seed()
1435: (8)                 actual = neg_binom(n * 3, p)
1436: (8)                 assert_array_equal(actual, desired)
1437: (8)                 assert_raises(ValueError, neg_binom, bad_n * 3, p)
1438: (8)                 assert_raises(ValueError, neg_binom, n * 3, bad_p_one)
1439: (8)                 assert_raises(ValueError, neg_binom, n * 3, bad_p_two)
1440: (8)                 self.set_seed()
1441: (8)                 actual = neg_binom(n, p * 3)
1442: (8)                 assert_array_equal(actual, desired)
1443: (8)                 assert_raises(ValueError, neg_binom, bad_n, p * 3)
1444: (8)                 assert_raises(ValueError, neg_binom, n, bad_p_one * 3)
1445: (8)                 assert_raises(ValueError, neg_binom, n, bad_p_two * 3)
1446: (4)             def test_poisson(self):
1447: (8)                 max_lam = random.RandomState()._poisson_lam_max
1448: (8)                 lam = [1]
1449: (8)                 bad_lam_one = [-1]
1450: (8)                 bad_lam_two = [max_lam * 2]
1451: (8)                 poisson = random.poisson
1452: (8)                 desired = np.array([1, 1, 0])
1453: (8)                 self.set_seed()
1454: (8)                 actual = poisson(lam * 3)
1455: (8)                 assert_array_equal(actual, desired)
1456: (8)                 assert_raises(ValueError, poisson, bad_lam_one * 3)
1457: (8)                 assert_raises(ValueError, poisson, bad_lam_two * 3)
1458: (4)             def test_zipf(self):
1459: (8)                 a = [2]
1460: (8)                 bad_a = [0]
1461: (8)                 zipf = random.zipf
1462: (8)                 desired = np.array([2, 2, 1])
1463: (8)                 self.set_seed()
1464: (8)                 actual = zipf(a * 3)
1465: (8)                 assert_array_equal(actual, desired)
1466: (8)                 assert_raises(ValueError, zipf, bad_a * 3)
1467: (8)                 with np.errstate(invalid='ignore'):
1468: (12)                     assert_raises(ValueError, zipf, np.nan)
1469: (12)                     assert_raises(ValueError, zipf, [0, 0, np.nan])
1470: (4)             def test_geometric(self):
1471: (8)                 p = [0.5]
1472: (8)                 bad_p_one = [-1]
1473: (8)                 bad_p_two = [1.5]
1474: (8)                 geom = random.geometric
1475: (8)                 desired = np.array([2, 2, 2])
1476: (8)                 self.set_seed()
1477: (8)                 actual = geom(p * 3)
1478: (8)                 assert_array_equal(actual, desired)
1479: (8)                 assert_raises(ValueError, geom, bad_p_one * 3)
1480: (8)                 assert_raises(ValueError, geom, bad_p_two * 3)
1481: (4)             def test_hypergeometric(self):
1482: (8)                 ngood = [1]
1483: (8)                 nbad = [2]
1484: (8)                 nsample = [2]
1485: (8)                 bad_ngood = [-1]
1486: (8)                 bad_nbad = [-2]
1487: (8)                 bad_nsamp

```

```

1488: (8)             bad_nsamp le _two = [4]
1489: (8)             hypergeom = random.hypergeometric
1490: (8)             desired = np.array([1, 1, 1])
1491: (8)             self.set_seed()
1492: (8)             actual = hypergeom(ngood * 3, nbad, nsamp le)
1493: (8)             assert_array_equal(actual, desired)
1494: (8)             assert_raises(ValueError, hypergeom, bad_ngood * 3, nbad, nsamp le)
1495: (8)             assert_raises(ValueError, hypergeom, ngood * 3, bad_nbad, nsamp le)
1496: (8)             assert_raises(ValueError, hypergeom, ngood * 3, nbad, bad_nsamp le_one)
1497: (8)             assert_raises(ValueError, hypergeom, ngood * 3, nbad, bad_nsamp le_two)
1498: (8)             self.set_seed()
1499: (8)             actual = hypergeom(ngood, nbad * 3, nsamp le)
1500: (8)             assert_array_equal(actual, desired)
1501: (8)             assert_raises(ValueError, hypergeom, bad_ngood, nbad * 3, nsamp le)
1502: (8)             assert_raises(ValueError, hypergeom, ngood, bad_nbad * 3, nsamp le)
1503: (8)             assert_raises(ValueError, hypergeom, ngood, nbad * 3, bad_nsamp le_one)
1504: (8)             assert_raises(ValueError, hypergeom, ngood, nbad * 3, bad_nsamp le_two)
1505: (8)             self.set_seed()
1506: (8)             actual = hypergeom(ngood, nbad, nsamp le * 3)
1507: (8)             assert_array_equal(actual, desired)
1508: (8)             assert_raises(ValueError, hypergeom, bad_ngood, nbad, nsamp le * 3)
1509: (8)             assert_raises(ValueError, hypergeom, ngood, bad_nbad, nsamp le * 3)
1510: (8)             assert_raises(ValueError, hypergeom, ngood, nbad, bad_nsamp le_one * 3)
1511: (8)             assert_raises(ValueError, hypergeom, ngood, nbad, bad_nsamp le_two * 3)
1512: (8)             assert_raises(ValueError, hypergeom, -1, 10, 20)
1513: (8)             assert_raises(ValueError, hypergeom, 10, -1, 20)
1514: (8)             assert_raises(ValueError, hypergeom, 10, 10, 0)
1515: (8)             assert_raises(ValueError, hypergeom, 10, 10, 25)
1516: (4)             def test_logseries(self):
1517: (8)                 p = [0.5]
1518: (8)                 bad_p_one = [2]
1519: (8)                 bad_p_two = [-1]
1520: (8)                 logseries = random.logseries
1521: (8)                 desired = np.array([1, 1, 1])
1522: (8)                 self.set_seed()
1523: (8)                 actual = logseries(p * 3)
1524: (8)                 assert_array_equal(actual, desired)
1525: (8)                 assert_raises(ValueError, logseries, bad_p_one * 3)
1526: (8)                 assert_raises(ValueError, logseries, bad_p_two * 3)
1527: (0)                 @pytest.mark.skipif(IS_WASM, reason="can't start thread")
1528: (0)                 class TestThread:
1529: (4)                     def setup_method(self):
1530: (8)                         self.seeds = range(4)
1531: (4)                     def check_function(self, function, sz):
1532: (8)                         from threading import Thread
1533: (8)                         out1 = np.empty((len(self.seeds),) + sz)
1534: (8)                         out2 = np.empty((len(self.seeds),) + sz)
1535: (8)                         t = [Thread(target=function, args=(random.RandomState(s), o))
1536: (13)                             for s, o in zip(self.seeds, out1)]
1537: (8)                         [x.start() for x in t]
1538: (8)                         [x.join() for x in t]
1539: (8)                         for s, o in zip(self.seeds, out2):
1540: (12)                             function(random.RandomState(s), o)
1541: (8)                         if np.intp().dtype.itemsize == 4 and sys.platform == "win32":
1542: (12)                             assert_array_almost_equal(out1, out2)
1543: (8)                         else:
1544: (12)                             assert_array_equal(out1, out2)
1545: (4)             def test_normal(self):
1546: (8)                 def gen_random(state, out):
1547: (12)                     out[...] = state.normal(size=10000)
1548: (8)                     self.check_function(gen_random, sz=(10000,))
1549: (4)             def test_exp(self):
1550: (8)                 def gen_random(state, out):
1551: (12)                     out[...] = state.exponential(scale=np.ones((100, 1000)))
1552: (8)                     self.check_function(gen_random, sz=(100, 1000))
1553: (4)             def test_multinomial(self):
1554: (8)                 def gen_random(state, out):
1555: (12)                     out[...] = state.multinomial(10, [1 / 6.] * 6, size=10000)
1556: (8)                     self.check_function(gen_random, sz=(10000, 6))

```

```

1557: (0)
1558: (4)
1559: (8)
1560: (8)
1561: (8)
1562: (8)
1563: (4)
1564: (8)
1565: (17)
1566: (17)
1567: (17)
1568: (17)
1569: (17)
1570: (8)
1571: (8)
1572: (12)
1573: (16)
1574: (12)
1575: (16)
1576: (12)
1577: (4)
1578: (8)
1579: (17)
1580: (17)
1581: (17)
1582: (17)
1583: (17)
1584: (17)
1585: (8)
1586: (8)
1587: (12)
1588: (16)
1589: (12)
1590: (16)
1591: (12)
1592: (12)
1593: (12)
1594: (12)
1595: (12)
1596: (12)
1597: (4)
1598: (8)
1599: (17)
1600: (8)
1601: (12)
1602: (12)
1603: (12)
1604: (12)
1605: (12)
1606: (12)
1607: (0)
1608: (4)
1609: (4)
1610: (4)
1611: (4)
1612: (4)
1613: (0)
1614: (4)
1615: (4)
1616: (4)
1617: (4)
1618: (4)
1619: (8)
1620: (4)
1621: (4)
1622: (0)
1623: (4)
1624: (8)
1625: (4)

class TestSingleEltArrayInput:
    def setup_method(self):
        self.argOne = np.array([2])
        self.argTwo = np.array([3])
        self.argThree = np.array([4])
        self.tgtShape = (1,)
    def test_one_arg_funcs(self):
        funcs = (random.exponential, random.standard_gamma,
                 random.chisquare, random.standard_t,
                 random.pareto, random.weibull,
                 random.power, random.rayleigh,
                 random.poisson, random.zipf,
                 random.geometric, random.logseries)
        probfuncs = (random.geometric, random.logseries)
        for func in funcs:
            if func in probfuncs: # p < 1.0
                out = func(np.array([0.5]))
            else:
                out = func(self.argOne)
            assert_equal(out.shape, self.tgtShape)
    def test_two_arg_funcs(self):
        funcs = (random.uniform, random.normal,
                 random.beta, random.gamma,
                 random.f, random.noncentral_chisquare,
                 random.vonmises, random.laplace,
                 random.gumbel, random.logistic,
                 random.lognormal, random.wald,
                 random.binomial, random.negative_binomial)
        probfuncs = (random.binomial, random.negative_binomial)
        for func in funcs:
            if func in probfuncs: # p <= 1
                argTwo = np.array([0.5])
            else:
                argTwo = self.argTwo
            out = func(self.argOne, argTwo)
            assert_equal(out.shape, self.tgtShape)
            out = func(self.argOne[0], argTwo)
            assert_equal(out.shape, self.tgtShape)
            out = func(self.argOne, argTwo[0])
            assert_equal(out.shape, self.tgtShape)
    def test_three_arg_funcs(self):
        funcs = [random.noncentral_f, random.triangular,
                 random.hypergeometric]
        for func in funcs:
            out = func(self.argOne, self.argTwo, self.argThree)
            assert_equal(out.shape, self.tgtShape)
            out = func(self.argOne[0], self.argTwo, self.argThree)
            assert_equal(out.shape, self.tgtShape)
            out = func(self.argOne, self.argTwo[0], self.argThree)
            assert_equal(out.shape, self.tgtShape)
    def test_integer_dtype(int_func):
        random.seed(123456789)
        fname, args, sha256 = int_func
        f = getattr(random, fname)
        actual = f(*args, size=2)
        assert_(actual.dtype == np.dtype('l'))
    def test_integer_repeat(int_func):
        random.seed(123456789)
        fname, args, sha256 = int_func
        f = getattr(random, fname)
        val = f(*args, size=1000000)
        if sys.byteorder != 'little':
            val = val.byteswap()
        res = hashlib.sha256(val.view(np.int8)).hexdigest()
        assert_(res == sha256)
    def test_broadcast_size_error():
        with pytest.raises(ValueError):
            random.binomial(1, [0.3, 0.7], size=(2, 1))
        with pytest.raises(ValueError):

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1626: (8)             random.binomial([1, 2], 0.3, size=(2, 1))
1627: (4)             with pytest.raises(ValueError):
1628: (8)                 random.binomial([1, 2], [0.3, 0.7], size=(2, 1))
1629: (0)             def test_randomstate_ctor_old_style_pickle():
1630: (4)                 rs = np.random.RandomState(MT19937(0))
1631: (4)                 rs.standard_normal(1)
1632: (4)                 ctor, args, state_a = rs.__reduce__()
1633: (4)                 assert args[:1] == ("MT19937",)
1634: (4)                 b = ctor(*args[:1])
1635: (4)                 b.set_state(state_a)
1636: (4)                 state_b = b.get_state(legacy=False)
1637: (4)                 assert_equal(state_a['bit_generator'], state_b['bit_generator'])
1638: (4)                 assert_array_equal(state_a['state']['key'], state_b['state']['key'])
1639: (4)                 assert_array_equal(state_a['state']['pos'], state_b['state']['pos'])
1640: (4)                 assert_equal(state_a['has_gauss'], state_b['has_gauss'])
1641: (4)                 assert_equal(state_a['gauss'], state_b['gauss'])
1642: (0)             def test_hot_swap(restored_singleton_bitgen):
1643: (4)                 def_bg = np.random.default_rng(0)
1644: (4)                 bg = def_bg.bit_generator
1645: (4)                 np.random.set_bit_generator(bg)
1646: (4)                 assert isinstance(np.random.mtrand._rand._bit_generator, type(bg))
1647: (4)                 second_bg = np.random.get_bit_generator()
1648: (4)                 assert bg is second_bg
1649: (0)             def test_seed_alt_bit_gen(restored_singleton_bitgen):
1650: (4)                 bg = PCG64(0)
1651: (4)                 np.random.set_bit_generator(bg)
1652: (4)                 state = np.random.get_state(legacy=False)
1653: (4)                 np.random.seed(1)
1654: (4)                 new_state = np.random.get_state(legacy=False)
1655: (4)                 print(state)
1656: (4)                 print(new_state)
1657: (4)                 assert state["bit_generator"] == "PCG64"
1658: (4)                 assert state["state"]["state"] != new_state["state"]["state"]
1659: (4)                 assert state["state"]["inc"] != new_state["state"]["inc"]
1660: (0)             def test_state_error_alt_bit_gen(restored_singleton_bitgen):
1661: (4)                 state = np.random.get_state()
1662: (4)                 bg = PCG64(0)
1663: (4)                 np.random.set_bit_generator(bg)
1664: (4)                 with pytest.raises(ValueError, match="state must be for a PCG64"):
1665: (8)                     np.random.set_state(state)
1666: (0)             def test_swap_worked(restored_singleton_bitgen):
1667: (4)                 np.random.seed(98765)
1668: (4)                 vals = np.random.randint(0, 2 ** 30, 10)
1669: (4)                 bg = PCG64(0)
1670: (4)                 state = bg.state
1671: (4)                 np.random.set_bit_generator(bg)
1672: (4)                 state_direct = np.random.get_state(legacy=False)
1673: (4)                 for field in state:
1674: (8)                     assert state[field] == state_direct[field]
1675: (4)                 np.random.seed(98765)
1676: (4)                 pcg_vals = np.random.randint(0, 2 ** 30, 10)
1677: (4)                 assert not np.all(vals == pcg_vals)
1678: (4)                 new_state = bg.state
1679: (4)                 assert new_state["state"]["state"] != state["state"]["state"]
1680: (4)                 assert new_state["state"]["inc"] == new_state["state"]["inc"]
1681: (0)             def test_swapped_singleton_against_direct(restored_singleton_bitgen):
1682: (4)                 np.random.set_bit_generator(PCG64(98765))
1683: (4)                 singleton_vals = np.random.randint(0, 2 ** 30, 10)
1684: (4)                 rg = np.random.RandomState(PCG64(98765))
1685: (4)                 non_singleton_vals = rg.randint(0, 2 ** 30, 10)
1686: (4)                 assert_equal(non_singleton_vals, singleton_vals)

```

-----  
File 316 - test\_randomstate\_regression.py:

```

1: (0)             import sys
2: (0)             import pytest
3: (0)             from numpy.testing import (

```

```

4: (4)             assert_, assert_array_equal, assert_raises,
5: (4)             )
6: (0)             import numpy as np
7: (0)             from numpy import random
8: (0)             class TestRegression:
9: (4)                 def test_VonMises_range(self):
10: (8)                     for mu in np.linspace(-7., 7., 5):
11: (12)                         r = random.vonmises(mu, 1, 50)
12: (12)                         assert_(np.all(r > -np.pi) and np.all(r <= np.pi))
13: (4)                 def test_hypergeometric_range(self):
14: (8)                     assert_(np.all(random.hypergeometric(3, 18, 11, size=10) < 4))
15: (8)                     assert_(np.all(random.hypergeometric(18, 3, 11, size=10) > 0))
16: (8)                     args = [
17: (12)                         (2**20 - 2, 2**20 - 2, 2**20 - 2), # Check for 32-bit systems
18: (8)                     ]
19: (8)                     is_64bits = sys.maxsize > 2**32
20: (8)                     if is_64bits and sys.platform != 'win32':
21: (12)                         args.append((2**40 - 2, 2**40 - 2, 2**40 - 2))
22: (8)                     for arg in args:
23: (12)                         assert_(random.hypergeometric(*arg) > 0)
24: (4)                 def test_logseries_convergence(self):
25: (8)                     N = 1000
26: (8)                     random.seed(0)
27: (8)                     rvsn = random.logseries(0.8, size=N)
28: (8)                     freq = np.sum(rvsn == 1) / N
29: (8)                     msg = f'Frequency was {freq:f}, should be > 0.45'
30: (8)                     assert_(freq > 0.45, msg)
31: (8)                     freq = np.sum(rvsn == 2) / N
32: (8)                     msg = f'Frequency was {freq:f}, should be < 0.23'
33: (8)                     assert_(freq < 0.23, msg)
34: (4)                 def test_shuffle_mixed_dimension(self):
35: (8)                     for t in [[1, 2, 3, None],
36: (18)                         [(1, 1), (2, 2), (3, 3), None],
37: (18)                         [1, (2, 2), (3, 3), None],
38: (18)                         [(1, 1), 2, 3, None]]:
39: (12)                         random.seed(12345)
40: (12)                         shuffled = list(t)
41: (12)                         random.shuffle(shuffled)
42: (12)                         expected = np.array([t[0], t[3], t[1], t[2]], dtype=object)
43: (12)                         assert_array_equal(np.array(shuffled, dtype=object), expected)
44: (4)                 def test_call_within_randomstate(self):
45: (8)                     m = random.RandomState()
46: (8)                     res = np.array([0, 8, 7, 2, 1, 9, 4, 7, 0, 3])
47: (8)                     for i in range(3):
48: (12)                         random.seed(i)
49: (12)                         m.seed(4321)
50: (12)                         assert_array_equal(m.choice(10, size=10, p=np.ones(10)/10.), res)
51: (4)                 def test_multivariate_normal_size_types(self):
52: (8)                     random.multivariate_normal([0], [[0]], size=1)
53: (8)                     random.multivariate_normal([0], [[0]], size=np.int_(1))
54: (8)                     random.multivariate_normal([0], [[0]], size=np.int64(1))
55: (4)                 def test_beta_small_parameters(self):
56: (8)                     random.seed(1234567890)
57: (8)                     x = random.beta(0.0001, 0.0001, size=100)
58: (8)                     assert_(not np.any(np.isnan(x)), 'Nans in random.beta')
59: (4)                 def test_choice_sum_of_probs_tolerance(self):
60: (8)                     random.seed(1234)
61: (8)                     a = [1, 2, 3]
62: (8)                     counts = [4, 4, 2]
63: (8)                     for dt in np.float16, np.float32, np.float64:
64: (12)                         probs = np.array(counts, dtype=dt) / sum(counts)
65: (12)                         c = random.choice(a, p=probs)
66: (12)                         assert_(c in a)
67: (12)                         assert_raises(ValueError, random.choice, a, p=probs*0.9)
68: (4)                 def test_shuffle_of_array_of_different_length_strings(self):
69: (8)                     random.seed(1234)
70: (8)                     a = np.array(['a', 'a' * 1000])
71: (8)                     for _ in range(100):
72: (12)                         random.shuffle(a)

```

```

73: (8)             import gc
74: (8)             gc.collect()
75: (4)         def test_shuffle_of_array_of_objects(self):
76: (8)             random.seed(1234)
77: (8)             a = np.array([np.arange(1), np.arange(4)], dtype=object)
78: (8)             for _ in range(1000):
79: (12)                 random.shuffle(a)
80: (8)             import gc
81: (8)             gc.collect()
82: (4)         def test_permutation_subclass(self):
83: (8)             class N(np.ndarray):
84: (12)                 pass
85: (8)             random.seed(1)
86: (8)             orig = np.arange(3).view(N)
87: (8)             perm = random.permutation(orig)
88: (8)             assert_array_equal(perm, np.array([0, 2, 1]))
89: (8)             assert_array_equal(orig, np.arange(3).view(N))
90: (8)         class M:
91: (12)             a = np.arange(5)
92: (12)             def __array__(self):
93: (16)                 return self.a
94: (8)             random.seed(1)
95: (8)             m = M()
96: (8)             perm = random.permutation(m)
97: (8)             assert_array_equal(perm, np.array([2, 1, 4, 0, 3]))
98: (8)             assert_array_equal(m.__array__(), np.arange(5))
99: (4)         def test_warns_byteorder(self):
100: (8)             other_byteord_dt = '<i4' if sys.byteorder == 'big' else '>i4'
101: (8)             with pytest.deprecated_call(match='non-native byteorder is not'):
102: (12)                 random.randint(0, 200, size=10, dtype=other_byteord_dt)
103: (4)         def test_named_argument_initialization(self):
104: (8)             rs1 = np.random.RandomState(123456789)
105: (8)             rs2 = np.random.RandomState(seed=123456789)
106: (8)             assert rs1.randint(0, 100) == rs2.randint(0, 100)
107: (4)         def test_choice_retun_dtype(self):
108: (8)             c = np.random.choice(10, p=[.1]*10, size=2)
109: (8)             assert c.dtype == np.dtype(int)
110: (8)             c = np.random.choice(10, p=[.1]*10, replace=False, size=2)
111: (8)             assert c.dtype == np.dtype(int)
112: (8)             c = np.random.choice(10, size=2)
113: (8)             assert c.dtype == np.dtype(int)
114: (8)             c = np.random.choice(10, replace=False, size=2)
115: (8)             assert c.dtype == np.dtype(int)
116: (4)             @pytest.mark.skipif(np.iinfo('l').max < 2**32,
117: (24)                             reason='Cannot test with 32-bit C long')
118: (4)         def test_randint_117(self):
119: (8)             random.seed(0)
120: (8)             expected = np.array([2357136044, 2546248239, 3071714933, 3626093760,
121: (29)                             2588848963, 3684848379, 2340255427, 3638918503,
122: (29)                             1819583497, 2678185683], dtype='int64')
123: (8)             actual = random.randint(2**32, size=10)
124: (8)             assert_array_equal(actual, expected)
125: (4)         def test_p_zero_stream(self):
126: (8)             np.random.seed(12345)
127: (8)             assert_array_equal(random.binomial(1, [0, 0.25, 0.5, 0.75, 1]),
128: (27)                             [0, 0, 0, 1, 1])
129: (4)         def test_n_zero_stream(self):
130: (8)             np.random.seed(8675309)
131: (8)             expected = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
132: (29)                             [3, 4, 2, 3, 3, 1, 5, 3, 1, 3]])
133: (8)             assert_array_equal(random.binomial([[0], [10]], 0.25, size=(2, 10)),
134: (27)                             expected)
135: (0)         def test_multinomial_empty():
136: (4)             assert random.multinomial(10, []).shape == (0,)
137: (4)             assert random.multinomial(3, [], size=(7, 5, 3)).shape == (7, 5, 3, 0)
138: (0)         def test_multinomial_1d_pval():
139: (4)             with pytest.raises(TypeError, match="pvals must be a 1-d"):
140: (8)                 random.multinomial(10, 0.3)

```

## File 317 - test\_regression.py:

```

1: (0)          import sys
2: (0)          from numpy.testing import (
3: (4)              assert_, assert_array_equal, assert_raises,
4: (4)          )
5: (0)          from numpy import random
6: (0)          import numpy as np
7: (0)          class TestRegression:
8: (4)              def test_VonMises_range(self):
9: (8)                  for mu in np.linspace(-7., 7., 5):
10: (12)                      r = random.mtrand.vonmises(mu, 1, 50)
11: (12)                      assert_(np.all(r > -np.pi) and np.all(r <= np.pi))
12: (4)              def test_hypergeometric_range(self):
13: (8)                  assert_(np.all(np.random.hypergeometric(3, 18, 11, size=10) < 4))
14: (8)                  assert_(np.all(np.random.hypergeometric(18, 3, 11, size=10) > 0))
15: (8)                  args = [
16: (12)                      (2**20 - 2, 2**20 - 2, 2**20 - 2), # Check for 32-bit systems
17: (8)                  ]
18: (8)                  is_64bits = sys.maxsize > 2**32
19: (8)                  if is_64bits and sys.platform != 'win32':
20: (12)                      args.append((2**40 - 2, 2**40 - 2, 2**40 - 2))
21: (8)                  for arg in args:
22: (12)                      assert_(np.random.hypergeometric(*arg) > 0)
23: (4)              def test_logseries_convergence(self):
24: (8)                  N = 1000
25: (8)                  np.random.seed(0)
26: (8)                  rvsn = np.random.logseries(0.8, size=N)
27: (8)                  freq = np.sum(rvsn == 1) / N
28: (8)                  msg = f'Frequency was {freq:f}, should be > 0.45'
29: (8)                  assert_(freq > 0.45, msg)
30: (8)                  freq = np.sum(rvsn == 2) / N
31: (8)                  msg = f'Frequency was {freq:f}, should be < 0.23'
32: (8)                  assert_(freq < 0.23, msg)
33: (4)              def test_shuffle_mixed_dimension(self):
34: (8)                  for t in [[1, 2, 3, None],
35: (18)                      [(1, 1), (2, 2), (3, 3), None],
36: (18)                      [1, (2, 2), (3, 3), None],
37: (18)                      [(1, 1), 2, 3, None]]:
38: (12)                      np.random.seed(12345)
39: (12)                      shuffled = list(t)
40: (12)                      random.shuffle(shuffled)
41: (12)                      expected = np.array([t[0], t[3], t[1], t[2]], dtype=object)
42: (12)                      assert_array_equal(np.array(shuffled, dtype=object), expected)
43: (4)              def test_call_within_randomstate(self):
44: (8)                  m = np.random.RandomState()
45: (8)                  res = np.array([0, 8, 7, 2, 1, 9, 4, 7, 0, 3])
46: (8)                  for i in range(3):
47: (12)                      np.random.seed(i)
48: (12)                      m.seed(4321)
49: (12)                      assert_array_equal(m.choice(10, size=10, p=np.ones(10)/10.), res)
50: (4)              def test_multivariate_normal_size_types(self):
51: (8)                  np.random.multivariate_normal([0], [[0]], size=1)
52: (8)                  np.random.multivariate_normal([0], [[0]], size=np.int_(1))
53: (8)                  np.random.multivariate_normal([0], [[0]], size=np.int64(1))
54: (4)              def test_beta_small_parameters(self):
55: (8)                  np.random.seed(1234567890)
56: (8)                  x = np.random.beta(0.0001, 0.0001, size=100)
57: (8)                  assert_(not np.any(np.isnan(x)), 'Nans in np.random.beta')
58: (4)              def test_choice_sum_of_probs_tolerance(self):
59: (8)                  np.random.seed(1234)
60: (8)                  a = [1, 2, 3]
61: (8)                  counts = [4, 4, 2]
62: (8)                  for dt in np.float16, np.float32, np.float64:
63: (12)                      probs = np.array(counts, dtype=dt) / sum(counts)
64: (12)                      c = np.random.choice(a, p=probs)
65: (12)                      assert_(c in a)

```

```

66: (12)             assert_raises(ValueError, np.random.choice, a, p=probs*0.9)
67: (4)      def test_shuffle_of_array_of_different_length_strings(self):
68: (8)          np.random.seed(1234)
69: (8)          a = np.array(['a', 'a' * 1000])
70: (8)          for _ in range(100):
71: (12)              np.random.shuffle(a)
72: (8)          import gc
73: (8)          gc.collect()
74: (4)      def test_shuffle_of_array_of_objects(self):
75: (8)          np.random.seed(1234)
76: (8)          a = np.array([np.arange(1), np.arange(4)], dtype=object)
77: (8)          for _ in range(1000):
78: (12)              np.random.shuffle(a)
79: (8)          import gc
80: (8)          gc.collect()
81: (4)      def test_permutation_subclass(self):
82: (8)          class N(np.ndarray):
83: (12)              pass
84: (8)          np.random.seed(1)
85: (8)          orig = np.arange(3).view(N)
86: (8)          perm = np.random.permutation(orig)
87: (8)          assert_array_equal(perm, np.array([0, 2, 1]))
88: (8)          assert_array_equal(orig, np.arange(3).view(N))
89: (8)
90: (12)      class M:
91: (12)          a = np.arange(5)
92: (16)          def __array__(self):
93: (16)              return self.a
94: (8)          np.random.seed(1)
95: (8)          m = M()
96: (8)          perm = np.random.permutation(m)
97: (8)          assert_array_equal(perm, np.array([2, 1, 4, 0, 3]))
97: (8)          assert_array_equal(m.__array__(), np.arange(5))

```

---

## File 318 - test\_seed\_sequence.py:

```

1: (0)      import numpy as np
2: (0)      from numpy.testing import assert_array_equal, assert_array_compare
3: (0)      from numpy.random import SeedSequence
4: (0)      def test_reference_data():
5: (4)          """ Check that SeedSequence generates data the same as the C++ reference.
6: (4)          https://gist.github.com/imbeme/540829265469e673d045
7: (4)          """
8: (4)          inputs = [
9: (8)              [3735928559, 195939070, 229505742, 305419896],
10: (8)              [3668361503, 4165561550, 1661411377, 3634257570],
11: (8)              [164546577, 4166754639, 1765190214, 1303880213],
12: (8)              [446610472, 3941463886, 522937693, 1882353782],
13: (8)              [1864922766, 1719732118, 3882010307, 1776744564],
14: (8)              [4141682960, 3310988675, 553637289, 902896340],
15: (8)              [1134851934, 2352871630, 3699409824, 2648159817],
16: (8)              [1240956131, 3107113773, 1283198141, 1924506131],
17: (8)              [2669565031, 579818610, 3042504477, 2774880435],
18: (8)              [2766103236, 2883057919, 4029656435, 862374500],
19: (4)
20: (4)          outputs = [
21: (8)              [3914649087, 576849849, 3593928901, 2229911004],
22: (8)              [2240804226, 3691353228, 1365957195, 2654016646],
23: (8)              [3562296087, 3191708229, 1147942216, 3726991905],
24: (8)              [1403443605, 3591372999, 1291086759, 441919183],
25: (8)              [1086200464, 2191331643, 560336446, 3658716651],
26: (8)              [3249937430, 2346751812, 847844327, 2996632307],
27: (8)              [2584285912, 4034195531, 3523502488, 169742686],
28: (8)              [959045797, 3875435559, 1886309314, 359682705],
29: (8)              [3978441347, 432478529, 3223635119, 138903045],
30: (8)              [296367413, 4262059219, 13109864, 3283683422],
31: (4)
32: (4)          ]
            outputs64 = [

```

```

33: (8) [2477551240072187391, 9577394838764454085],
34: (8) [15854241394484835714, 11398914698975566411],
35: (8) [13708282465491374871, 16007308345579681096],
36: (8) [15424829579845884309, 1898028439751125927],
37: (8) [9411697742461147792, 15714068361935982142],
38: (8) [10079222287618677782, 12870437757549876199],
39: (8) [17326737873898640088, 729039288628699544],
40: (8) [16644868984619524261, 1544825456798124994],
41: (8) [1857481142255628931, 596584038813451439],
42: (8) [18305404959516669237, 14103312907920476776],
43: (4)
44: (4) ]
45: (8) for seed, expected, expected64 in zip(inputs, outputs, outputs64):
46: (8)     expected = np.array(expected, dtype=np.uint32)
47: (8)     ss = SeedSequence(seed)
48: (8)     state = ss.generate_state(len(expected))
49: (8)     assert_array_equal(state, expected)
50: (8)     state64 = ss.generate_state(len(expected64), dtype=np.uint64)
51: (8)     assert_array_equal(state64, expected64)
52: (0) def test_zero_padding():
53: (4)     """ Ensure that the implicit zero-padding does not cause problems.
54: (4)     """
55: (4)     ss0 = SeedSequence(42)
56: (4)     ss1 = SeedSequence(42 << 32)
57: (8)     assert_array_compare(
58: (8)         np.not_equal,
59: (8)         ss0.generate_state(4),
60: (8)         ss1.generate_state(4))
61: (26)     expected42 = np.array([3444837047, 2669555309, 2046530742, 3581440988],
62: (4)             dtype=np.uint32)
63: (4)     assert_array_equal(SeedSequence(42).generate_state(4), expected42)
64: (4)     assert_array_compare(
65: (8)         np.not_equal,
66: (8)         SeedSequence(42, spawn_key=(0,)).generate_state(4),
66: (8)         expected42)

-----

```

## File 319 - \_\_init\_\_.py:

```
1: (0)
```

## File 320 - test\_smoke.py:

```

1: (0)         import pickle
2: (0)         from functools import partial
3: (0)         import numpy as np
4: (0)         import pytest
5: (0)         from numpy.testing import assert_equal, assert_, assert_array_equal
6: (0)         from numpy.random import (Generator, MT19937, PCG64, PCG64DXSM, Philox, SFC64)
7: (0) @pytest.fixture(scope='module',
8: (16)             params=(np.bool_, np.int8, np.int16, np.int32, np.int64,
9: (24)                 np.uint8, np.uint16, np.uint32, np.uint64))
10: (0) def dtype(request):
11: (4)     return request.param
12: (0) def params_0(f):
13: (4)     val = f()
14: (4)     assert_(np.isscalar(val))
15: (4)     val = f(10)
16: (4)     assert_(val.shape == (10,))
17: (4)     val = f((10, 10))
18: (4)     assert_(val.shape == (10, 10))
19: (4)     val = f((10, 10, 10))
20: (4)     assert_(val.shape == (10, 10, 10))
21: (4)     val = f(size=(5, 5))
22: (4)     assert_(val.shape == (5, 5))
23: (0) def params_1(f, bounded=False):
24: (4)     a = 5.0

```

```

25: (4)          b = np.arange(2.0, 12.0)
26: (4)          c = np.arange(2.0, 102.0).reshape((10, 10))
27: (4)          d = np.arange(2.0, 1002.0).reshape((10, 10, 10))
28: (4)          e = np.array([2.0, 3.0])
29: (4)          g = np.arange(2.0, 12.0).reshape((1, 10, 1))
30: (4)          if bounded:
31: (8)              a = 0.5
32: (8)              b = b / (1.5 * b.max())
33: (8)              c = c / (1.5 * c.max())
34: (8)              d = d / (1.5 * d.max())
35: (8)              e = e / (1.5 * e.max())
36: (8)              g = g / (1.5 * g.max())
37: (4)          f(a)
38: (4)          f(a, size=(10, 10))
39: (4)          f(b)
40: (4)          f(c)
41: (4)          f(d)
42: (4)          f(b, size=10)
43: (4)          f(e, size=(10, 2))
44: (4)          f(g, size=(10, 10, 10))
45: (0)          def comp_state(state1, state2):
46: (4)              identical = True
47: (4)              if isinstance(state1, dict):
48: (8)                  for key in state1:
49: (12)                      identical &= comp_state(state1[key], state2[key])
50: (4)              elif type(state1) != type(state2):
51: (8)                  identical &= type(state1) == type(state2)
52: (4)              else:
53: (8)                  if (isinstance(state1, (list, tuple, np.ndarray)) and isinstance(
54: (16)                      state2, (list, tuple, np.ndarray))):
55: (12)                      for s1, s2 in zip(state1, state2):
56: (16)                          identical &= comp_state(s1, s2)
57: (8)                  else:
58: (12)                      identical &= state1 == state2
59: (4)              return identical
60: (0)          def warmup(rng, n=None):
61: (4)              if n is None:
62: (8)                  n = 11 + np.random.randint(0, 20)
63: (4)                  rg.standard_normal(n)
64: (4)                  rg.standard_normal(n)
65: (4)                  rg.standard_normal(n, dtype=np.float32)
66: (4)                  rg.standard_normal(n, dtype=np.float32)
67: (4)                  rg.integers(0, 2 ** 24, n, dtype=np.uint64)
68: (4)                  rg.integers(0, 2 ** 48, n, dtype=np.uint64)
69: (4)                  rg.standard_gamma(11.0, n)
70: (4)                  rg.standard_gamma(11.0, n, dtype=np.float32)
71: (4)                  rg.random(n, dtype=np.float64)
72: (4)                  rg.random(n, dtype=np.float32)
73: (0)          class RNG:
74: (4)              @classmethod
75: (4)              def setup_class(cls):
76: (8)                  cls.bit_generator = PCG64
77: (8)                  cls.advance = None
78: (8)                  cls.seed = [12345]
79: (8)                  cls.rg = Generator(cls.bit_generator(*cls.seed))
80: (8)                  cls.initial_state = cls.rg.bit_generator.state
81: (8)                  cls.seed_vector_bits = 64
82: (8)                  cls._extra_setup()
83: (4)              @classmethod
84: (4)              def _extra_setup(cls):
85: (8)                  cls.vec_1d = np.arange(2.0, 102.0)
86: (8)                  cls.vec_2d = np.arange(2.0, 102.0)[None, :]
87: (8)                  cls.mat = np.arange(2.0, 102.0, 0.01).reshape((100, 100))
88: (8)                  cls.seed_error = TypeError
89: (4)              def _reset_state(self):
90: (8)                  self.rg.bit_generator.state = self.initial_state
91: (4)              def test_init(self):
92: (8)                  rg = Generator(self.bit_generator())
93: (8)                  state = rg.bit_generator.state

```

```

94: (8)             rg.standard_normal(1)
95: (8)             rg.standard_normal(1)
96: (8)             rg.bit_generator.state = state
97: (8)             new_state = rg.bit_generator.state
98: (8)             assert_(comp_state(state, new_state))
99: (4)             def test_advance(self):
100: (8)             state = self.rg.bit_generator.state
101: (8)             if hasattr(self.rg.bit_generator, 'advance'):
102: (12)             self.rg.bit_generator.advance(self.advance)
103: (12)             assert_(not comp_state(state, self.rg.bit_generator.state))
104: (8)         else:
105: (12)             bitgen_name = self.rg.bit_generator.__class__.__name__
106: (12)             pytest.skip(f'Advance is not supported by {bitgen_name}')
107: (4)             def test_jump(self):
108: (8)             state = self.rg.bit_generator.state
109: (8)             if hasattr(self.rg.bit_generator, 'jumped'):
110: (12)             bit_gen2 = self.rg.bit_generator.jumped()
111: (12)             jumped_state = bit_gen2.state
112: (12)             assert_(not comp_state(state, jumped_state))
113: (12)             self.rg.random(2 * 3 * 5 * 7 * 11 * 13 * 17)
114: (12)             self.rg.bit_generator.state = state
115: (12)             bit_gen3 = self.rg.bit_generator.jumped()
116: (12)             rejumped_state = bit_gen3.state
117: (12)             assert_(comp_state(jumped_state, rejumped_state))
118: (8)         else:
119: (12)             bitgen_name = self.rg.bit_generator.__class__.__name__
120: (12)             if bitgen_name not in ('SFC64',):
121: (16)                 raise AttributeError(f'no "jumped" in {bitgen_name}')
122: (12)             pytest.skip(f'Jump is not supported by {bitgen_name}')
123: (4)             def test_uniform(self):
124: (8)             r = self.rg.uniform(-1.0, 0.0, size=10)
125: (8)             assert_(len(r) == 10)
126: (8)             assert_((r > -1).all())
127: (8)             assert_((r <= 0).all())
128: (4)             def test_uniform_array(self):
129: (8)             r = self.rg.uniform(np.array([-1.0] * 10), 0.0, size=10)
130: (8)             assert_(len(r) == 10)
131: (8)             assert_((r > -1).all())
132: (8)             assert_((r <= 0).all())
133: (8)             r = self.rg.uniform(np.array([-1.0] * 10),
134: (28)                           np.array([0.0] * 10), size=10)
135: (8)             assert_(len(r) == 10)
136: (8)             assert_((r > -1).all())
137: (8)             assert_((r <= 0).all())
138: (8)             r = self.rg.uniform(-1.0, np.array([0.0] * 10), size=10)
139: (8)             assert_(len(r) == 10)
140: (8)             assert_((r > -1).all())
141: (8)             assert_((r <= 0).all())
142: (4)             def test_random(self):
143: (8)             assert_(len(self.rg.random(10)) == 10)
144: (8)             params_0(self.rg.random)
145: (4)             def test_standard_normal_zig(self):
146: (8)             assert_(len(self.rg.standard_normal(10)) == 10)
147: (4)             def test_standard_normal(self):
148: (8)             assert_(len(self.rg.standard_normal(10)) == 10)
149: (8)             params_0(self.rg.standard_normal)
150: (4)             def test_standard_gamma(self):
151: (8)             assert_(len(self.rg.standard_gamma(10, 10)) == 10)
152: (8)             assert_(len(self.rg.standard_gamma(np.array([10] * 10), 10)) == 10)
153: (8)             params_1(self.rg.standard_gamma)
154: (4)             def test_standard_exponential(self):
155: (8)             assert_(len(self.rg.standard_exponential(10)) == 10)
156: (8)             params_0(self.rg.standard_exponential)
157: (4)             def test_standard_exponential_float(self):
158: (8)             randoms = self.rg.standard_exponential(10, dtype='float32')
159: (8)             assert_(len(randoms) == 10)
160: (8)             assert randoms.dtype == np.float32
161: (8)             params_0(partial(self.rg.standard_exponential, dtype='float32'))
162: (4)             def test_standard_exponential_float_log(self):

```

```

163: (8)           randoms = self.rg.standard_exponential(10, dtype='float32',
164: (47)                     method='inv')
165: (8)           assert_(len(randoms) == 10)
166: (8)           assert randoms.dtype == np.float32
167: (8)           params_0(partial(self.rg.standard_exponential, dtype='float32',
168: (25)                         method='inv'))
169: (4)           def test_standard_cauchy(self):
170: (8)             assert_(len(self.rg.standard_cauchy(10)) == 10)
171: (8)             params_0(self.rg.standard_cauchy)
172: (4)           def test_standard_t(self):
173: (8)             assert_(len(self.rg.standard_t(10, 10)) == 10)
174: (8)             params_1(self.rg.standard_t)
175: (4)           def test_binomial(self):
176: (8)             assert_(self.rg.binomial(10, .5) >= 0)
177: (8)             assert_(self.rg.binomial(1000, .5) >= 0)
178: (4)           def test_reset_state(self):
179: (8)             state = self.rg.bit_generator.state
180: (8)             int_1 = self.rg.integers(2**31)
181: (8)             self.rg.bit_generator.state = state
182: (8)             int_2 = self.rg.integers(2**31)
183: (8)             assert_(int_1 == int_2)
184: (4)           def test_entropy_init(self):
185: (8)             rg = Generator(self.bit_generator())
186: (8)             rg2 = Generator(self.bit_generator())
187: (8)             assert_(not comp_state(rg.bit_generator.state,
188: (31)                           rg2.bit_generator.state))
189: (4)           def test_seed(self):
190: (8)             rg = Generator(self.bit_generator(*self.seed))
191: (8)             rg2 = Generator(self.bit_generator(*self.seed))
192: (8)             rg.random()
193: (8)             rg2.random()
194: (8)             assert_(comp_state(rg.bit_generator.state, rg2.bit_generator.state))
195: (4)           def test_reset_state_gauss(self):
196: (8)             rg = Generator(self.bit_generator(*self.seed))
197: (8)             rg.standard_normal()
198: (8)             state = rg.bit_generator.state
199: (8)             n1 = rg.standard_normal(size=10)
200: (8)             rg2 = Generator(self.bit_generator())
201: (8)             rg2.bit_generator.state = state
202: (8)             n2 = rg2.standard_normal(size=10)
203: (8)             assert_array_equal(n1, n2)
204: (4)           def test_reset_state_uint32(self):
205: (8)             rg = Generator(self.bit_generator(*self.seed))
206: (8)             rg.integers(0, 2 ** 24, 120, dtype=np.uint32)
207: (8)             state = rg.bit_generator.state
208: (8)             n1 = rg.integers(0, 2 ** 24, 10, dtype=np.uint32)
209: (8)             rg2 = Generator(self.bit_generator())
210: (8)             rg2.bit_generator.state = state
211: (8)             n2 = rg2.integers(0, 2 ** 24, 10, dtype=np.uint32)
212: (8)             assert_array_equal(n1, n2)
213: (4)           def test_reset_state_float(self):
214: (8)             rg = Generator(self.bit_generator(*self.seed))
215: (8)             rg.random(dtype='float32')
216: (8)             state = rg.bit_generator.state
217: (8)             n1 = rg.random(size=10, dtype='float32')
218: (8)             rg2 = Generator(self.bit_generator())
219: (8)             rg2.bit_generator.state = state
220: (8)             n2 = rg2.random(size=10, dtype='float32')
221: (8)             assert_((n1 == n2).all())
222: (4)           def test_shuffle(self):
223: (8)             original = np.arange(200, 0, -1)
224: (8)             permuted = self.rg.permutation(original)
225: (8)             assert_((original != permuted).any())
226: (4)           def test_permutation(self):
227: (8)             original = np.arange(200, 0, -1)
228: (8)             permuted = self.rg.permutation(original)
229: (8)             assert_((original != permuted).any())
230: (4)           def test_beta(self):
231: (8)             vals = self.rg.beta(2.0, 2.0, 10)

```

```

232: (8)             assert_(len(vals) == 10)
233: (8)             vals = self.rg.beta(np.array([2.0] * 10), 2.0)
234: (8)             assert_(len(vals) == 10)
235: (8)             vals = self.rg.beta(2.0, np.array([2.0] * 10))
236: (8)             assert_(len(vals) == 10)
237: (8)             vals = self.rg.beta(np.array([2.0] * 10), np.array([2.0] * 10))
238: (8)             assert_(len(vals) == 10)
239: (8)             vals = self.rg.beta(np.array([2.0] * 10), np.array([[2.0]] * 10))
240: (8)             assert_(vals.shape == (10, 10))
241: (4)             def test_bytes(self):
242: (8)                 vals = self.rg.bytes(10)
243: (8)                 assert_(len(vals) == 10)
244: (4)             def test_chisquare(self):
245: (8)                 vals = self.rg.chisquare(2.0, 10)
246: (8)                 assert_(len(vals) == 10)
247: (8)                 params_1(self.rg.chisquare)
248: (4)             def test_exponential(self):
249: (8)                 vals = self.rg.exponential(2.0, 10)
250: (8)                 assert_(len(vals) == 10)
251: (8)                 params_1(self.rg.exponential)
252: (4)             def test_f(self):
253: (8)                 vals = self.rg.f(3, 1000, 10)
254: (8)                 assert_(len(vals) == 10)
255: (4)             def test_gamma(self):
256: (8)                 vals = self.rg.gamma(3, 2, 10)
257: (8)                 assert_(len(vals) == 10)
258: (4)             def test_geometric(self):
259: (8)                 vals = self.rg.geometric(0.5, 10)
260: (8)                 assert_(len(vals) == 10)
261: (8)                 params_1(self.rg.exponential, bounded=True)
262: (4)             def test_gumbel(self):
263: (8)                 vals = self.rg.gumbel(2.0, 2.0, 10)
264: (8)                 assert_(len(vals) == 10)
265: (4)             def test_laplace(self):
266: (8)                 vals = self.rg.laplace(2.0, 2.0, 10)
267: (8)                 assert_(len(vals) == 10)
268: (4)             def test_logistic(self):
269: (8)                 vals = self.rg.logistic(2.0, 2.0, 10)
270: (8)                 assert_(len(vals) == 10)
271: (4)             def test_logseries(self):
272: (8)                 vals = self.rg.logseries(0.5, 10)
273: (8)                 assert_(len(vals) == 10)
274: (4)             def test_negative_binomial(self):
275: (8)                 vals = self.rg.negative_binomial(10, 0.2, 10)
276: (8)                 assert_(len(vals) == 10)
277: (4)             def test_noncentral_chisquare(self):
278: (8)                 vals = self.rg.noncentral_chisquare(10, 2, 10)
279: (8)                 assert_(len(vals) == 10)
280: (4)             def test_noncentral_f(self):
281: (8)                 vals = self.rg.noncentral_f(3, 1000, 2, 10)
282: (8)                 assert_(len(vals) == 10)
283: (8)                 vals = self.rg.noncentral_f(np.array([3] * 10), 1000, 2)
284: (8)                 assert_(len(vals) == 10)
285: (8)                 vals = self.rg.noncentral_f(3, np.array([1000] * 10), 2)
286: (8)                 assert_(len(vals) == 10)
287: (8)                 vals = self.rg.noncentral_f(3, 1000, np.array([2] * 10))
288: (8)                 assert_(len(vals) == 10)
289: (4)             def test_normal(self):
290: (8)                 vals = self.rg.normal(10, 0.2, 10)
291: (8)                 assert_(len(vals) == 10)
292: (4)             def test_pareto(self):
293: (8)                 vals = self.rg.pareto(3.0, 10)
294: (8)                 assert_(len(vals) == 10)
295: (4)             def test_poisson(self):
296: (8)                 vals = self.rg.poisson(10, 10)
297: (8)                 assert_(len(vals) == 10)
298: (8)                 vals = self.rg.poisson(np.array([10] * 10))
299: (8)                 assert_(len(vals) == 10)
300: (8)                 params_1(self.rg.poisson)

```

```

301: (4)           def test_power(self):
302: (8)             vals = self.rg.power(0.2, 10)
303: (8)             assert_(len(vals) == 10)
304: (4)           def test_integers(self):
305: (8)             vals = self.rg.integers(10, 20, 10)
306: (8)             assert_(len(vals) == 10)
307: (4)           def test_rayleigh(self):
308: (8)             vals = self.rg.rayleigh(0.2, 10)
309: (8)             assert_(len(vals) == 10)
310: (8)             params_1(self.rg.rayleigh, bounded=True)
311: (4)           def test_vonmises(self):
312: (8)             vals = self.rg.vonmises(10, 0.2, 10)
313: (8)             assert_(len(vals) == 10)
314: (4)           def test_wald(self):
315: (8)             vals = self.rg.wald(1.0, 1.0, 10)
316: (8)             assert_(len(vals) == 10)
317: (4)           def test_weibull(self):
318: (8)             vals = self.rg.weibull(1.0, 10)
319: (8)             assert_(len(vals) == 10)
320: (4)           def test_zipf(self):
321: (8)             vals = self.rg.zipf(10, 10)
322: (8)             assert_(len(vals) == 10)
323: (8)             vals = self.rg.zipf(self.vec_1d)
324: (8)             assert_(len(vals) == 100)
325: (8)             vals = self.rg.zipf(self.vec_2d)
326: (8)             assert_(vals.shape == (1, 100))
327: (8)             vals = self.rg.zipf(self.mat)
328: (8)             assert_(vals.shape == (100, 100))
329: (4)           def test_hypergeometric(self):
330: (8)             vals = self.rg.hypergeometric(25, 25, 20)
331: (8)             assert_(np.isscalar(vals))
332: (8)             vals = self.rg.hypergeometric(np.array([25] * 10), 25, 20)
333: (8)             assert_(vals.shape == (10,))
334: (4)           def test_triangular(self):
335: (8)             vals = self.rg.triangular(-5, 0, 5)
336: (8)             assert_(np.isscalar(vals))
337: (8)             vals = self.rg.triangular(-5, np.array([0] * 10), 5)
338: (8)             assert_(vals.shape == (10,))
339: (4)           def test_multivariate_normal(self):
340: (8)             mean = [0, 0]
341: (8)             cov = [[1, 0], [0, 100]] # diagonal covariance
342: (8)             x = self.rg.multivariate_normal(mean, cov, 5000)
343: (8)             assert_(x.shape == (5000, 2))
344: (8)             x_zig = self.rg.multivariate_normal(mean, cov, 5000)
345: (8)             assert_(x_zig.shape == (5000, 2))
346: (8)             x_inv = self.rg.multivariate_normal(mean, cov, 5000)
347: (8)             assert_(x_inv.shape == (5000, 2))
348: (8)             assert_((x_zig != x_inv).any())
349: (4)           def test_multinomial(self):
350: (8)             vals = self.rg.multinomial(100, [1.0 / 3, 2.0 / 3])
351: (8)             assert_(vals.shape == (2,))
352: (8)             vals = self.rg.multinomial(100, [1.0 / 3, 2.0 / 3], size=10)
353: (8)             assert_(vals.shape == (10, 2))
354: (4)           def test_dirichlet(self):
355: (8)             s = self.rg.dirichlet((10, 5, 3), 20)
356: (8)             assert_(s.shape == (20, 3))
357: (4)           def test_pickle(self):
358: (8)             pick = pickle.dumps(self.rg)
359: (8)             unpick = pickle.loads(pick)
360: (8)             assert_((type(self.rg) == type(unpick)))
361: (8)             assert_(comp_state(self.rg.bit_generator.state,
362: (27)                           unpick.bit_generator.state))
363: (8)             pick = pickle.dumps(self.rg)
364: (8)             unpick = pickle.loads(pick)
365: (8)             assert_((type(self.rg) == type(unpick)))
366: (8)             assert_(comp_state(self.rg.bit_generator.state,
367: (27)                           unpick.bit_generator.state))
368: (4)           def test_seed_array(self):
369: (8)             if self.seed_vector_bits is None:

```

```

370: (12)           bitgen_name = self.bit_generator.__name__
371: (12)           pytest.skip(f'Vector seeding is not supported by {bitgen_name}')
372: (8)            if self.seed_vector_bits == 32:
373: (12)              dtype = np.uint32
374: (8)            else:
375: (12)              dtype = np.uint64
376: (8)            seed = np.array([1], dtype=dtype)
377: (8)            bg = self.bit_generator(seed)
378: (8)            state1 = bg.state
379: (8)            bg = self.bit_generator(1)
380: (8)            state2 = bg.state
381: (8)            assert_(comp_state(state1, state2))
382: (8)            seed = np.arange(4, dtype=dtype)
383: (8)            bg = self.bit_generator(seed)
384: (8)            state1 = bg.state
385: (8)            bg = self.bit_generator(seed[0])
386: (8)            state2 = bg.state
387: (8)            assert_(not comp_state(state1, state2))
388: (8)            seed = np.arange(1500, dtype=dtype)
389: (8)            bg = self.bit_generator(seed)
390: (8)            state1 = bg.state
391: (8)            bg = self.bit_generator(seed[0])
392: (8)            state2 = bg.state
393: (8)            assert_(not comp_state(state1, state2))
394: (8)            seed = 2 ** np.mod(np.arange(1500, dtype=dtype),
395: (27)                           self.seed_vector_bits - 1) + 1
396: (8)            bg = self.bit_generator(seed)
397: (8)            state1 = bg.state
398: (8)            bg = self.bit_generator(seed[0])
399: (8)            state2 = bg.state
400: (8)            assert_(not comp_state(state1, state2))
401: (4) def test_uniform_float(self):
402: (8)     rg = Generator(self.bit_generator(12345))
403: (8)     warmup(rg)
404: (8)     state = rg.bit_generator.state
405: (8)     r1 = rg.random(11, dtype=np.float32)
406: (8)     rg2 = Generator(self.bit_generator())
407: (8)     warmup(rg2)
408: (8)     rg2.bit_generator.state = state
409: (8)     r2 = rg2.random(11, dtype=np.float32)
410: (8)     assert_array_equal(r1, r2)
411: (8)     assert_equal(r1.dtype, np.float32)
412: (8)     assert_(comp_state(rg.bit_generator.state, rg2.bit_generator.state))
413: (4) def test_gamma_floats(self):
414: (8)     rg = Generator(self.bit_generator())
415: (8)     warmup(rg)
416: (8)     state = rg.bit_generator.state
417: (8)     r1 = rg.standard_gamma(4.0, 11, dtype=np.float32)
418: (8)     rg2 = Generator(self.bit_generator())
419: (8)     warmup(rg2)
420: (8)     rg2.bit_generator.state = state
421: (8)     r2 = rg2.standard_gamma(4.0, 11, dtype=np.float32)
422: (8)     assert_array_equal(r1, r2)
423: (8)     assert_equal(r1.dtype, np.float32)
424: (8)     assert_(comp_state(rg.bit_generator.state, rg2.bit_generator.state))
425: (4) def test_normal_floats(self):
426: (8)     rg = Generator(self.bit_generator())
427: (8)     warmup(rg)
428: (8)     state = rg.bit_generator.state
429: (8)     r1 = rg.standard_normal(11, dtype=np.float32)
430: (8)     rg2 = Generator(self.bit_generator())
431: (8)     warmup(rg2)
432: (8)     rg2.bit_generator.state = state
433: (8)     r2 = rg2.standard_normal(11, dtype=np.float32)
434: (8)     assert_array_equal(r1, r2)
435: (8)     assert_equal(r1.dtype, np.float32)
436: (8)     assert_(comp_state(rg.bit_generator.state, rg2.bit_generator.state))
437: (4) def test_normal_zig_floats(self):
438: (8)     rg = Generator(self.bit_generator())

```

```
439: (8)             warmup(rg)
440: (8)             state = rg.bit_generator.state
441: (8)             r1 = rg.standard_normal(11, dtype=np.float32)
442: (8)             rg2 = Generator(self.bit_generator())
443: (8)             warmup(rg2)
444: (8)             rg2.bit_generator.state = state
445: (8)             r2 = rg2.standard_normal(11, dtype=np.float32)
446: (8)             assert_array_equal(r1, r2)
447: (8)             assert_equal(r1.dtype, np.float32)
448: (8)             assert_(comp_state(rg.bit_generator.state, rg2.bit_generator.state))
449: (4)             def test_output_fill(self):
450: (8)                 rg = self.rg
451: (8)                 state = rg.bit_generator.state
452: (8)                 size = (31, 7, 97)
453: (8)                 existing = np.empty(size)
454: (8)                 rg.bit_generator.state = state
455: (8)                 rg.standard_normal(out=existing)
456: (8)                 rg.bit_generator.state = state
457: (8)                 direct = rg.standard_normal(size=size)
458: (8)                 assert_equal(direct, existing)
459: (8)                 sized = np.empty(size)
460: (8)                 rg.bit_generator.state = state
461: (8)                 rg.standard_normal(out=sized, size=sized.shape)
462: (8)                 existing = np.empty(size, dtype=np.float32)
463: (8)                 rg.bit_generator.state = state
464: (8)                 rg.standard_normal(out=existing, dtype=np.float32)
465: (8)                 rg.bit_generator.state = state
466: (8)                 direct = rg.standard_normal(size=size, dtype=np.float32)
467: (8)                 assert_equal(direct, existing)
468: (4)             def test_output_filling_uniform(self):
469: (8)                 rg = self.rg
470: (8)                 state = rg.bit_generator.state
471: (8)                 size = (31, 7, 97)
472: (8)                 existing = np.empty(size)
473: (8)                 rg.bit_generator.state = state
474: (8)                 rg.random(out=existing)
475: (8)                 rg.bit_generator.state = state
476: (8)                 direct = rg.random(size=size)
477: (8)                 assert_equal(direct, existing)
478: (8)                 existing = np.empty(size, dtype=np.float32)
479: (8)                 rg.bit_generator.state = state
480: (8)                 rg.random(out=existing, dtype=np.float32)
481: (8)                 rg.bit_generator.state = state
482: (8)                 direct = rg.random(size=size, dtype=np.float32)
483: (8)                 assert_equal(direct, existing)
484: (4)             def test_output_filling_exponential(self):
485: (8)                 rg = self.rg
486: (8)                 state = rg.bit_generator.state
487: (8)                 size = (31, 7, 97)
488: (8)                 existing = np.empty(size)
489: (8)                 rg.bit_generator.state = state
490: (8)                 rg.standard_exponential(out=existing)
491: (8)                 rg.bit_generator.state = state
492: (8)                 direct = rg.standard_exponential(size=size)
493: (8)                 assert_equal(direct, existing)
494: (8)                 existing = np.empty(size, dtype=np.float32)
495: (8)                 rg.bit_generator.state = state
496: (8)                 rg.standard_exponential(out=existing, dtype=np.float32)
497: (8)                 rg.bit_generator.state = state
498: (8)                 direct = rg.standard_exponential(size=size, dtype=np.float32)
499: (8)                 assert_equal(direct, existing)
500: (4)             def test_output_filling_gamma(self):
501: (8)                 rg = self.rg
502: (8)                 state = rg.bit_generator.state
503: (8)                 size = (31, 7, 97)
504: (8)                 existing = np.zeros(size)
505: (8)                 rg.bit_generator.state = state
506: (8)                 rg.standard_gamma(1.0, out=existing)
507: (8)                 rg.bit_generator.state = state
```

```

508: (8)             direct = rg.standard_gamma(1.0, size=size)
509: (8)             assert_equal(direct, existing)
510: (8)             existing = np.zeros(size, dtype=np.float32)
511: (8)             rg.bit_generator.state = state
512: (8)             rg.standard_gamma(1.0, out=existing, dtype=np.float32)
513: (8)             rg.bit_generator.state = state
514: (8)             direct = rg.standard_gamma(1.0, size=size, dtype=np.float32)
515: (8)             assert_equal(direct, existing)
516: (4)             def test_output_filling_gamma_broadcast(self):
517: (8)                 rg = self.rg
518: (8)                 state = rg.bit_generator.state
519: (8)                 size = (31, 7, 97)
520: (8)                 mu = np.arange(97.0) + 1.0
521: (8)                 existing = np.zeros(size)
522: (8)                 rg.bit_generator.state = state
523: (8)                 rg.standard_gamma(mu, out=existing)
524: (8)                 rg.bit_generator.state = state
525: (8)                 direct = rg.standard_gamma(mu, size=size)
526: (8)                 assert_equal(direct, existing)
527: (8)                 existing = np.zeros(size, dtype=np.float32)
528: (8)                 rg.bit_generator.state = state
529: (8)                 rg.standard_gamma(mu, out=existing, dtype=np.float32)
530: (8)                 rg.bit_generator.state = state
531: (8)                 direct = rg.standard_gamma(mu, size=size, dtype=np.float32)
532: (8)                 assert_equal(direct, existing)
533: (4)             def test_output_fill_error(self):
534: (8)                 rg = self.rg
535: (8)                 size = (31, 7, 97)
536: (8)                 existing = np.empty(size)
537: (8)                 with pytest.raises(TypeError):
538: (12)                     rg.standard_normal(out=existing, dtype=np.float32)
539: (8)                 with pytest.raises(ValueError):
540: (12)                     rg.standard_normal(out=existing[:3])
541: (8)                     existing = np.empty(size, dtype=np.float32)
542: (8)                 with pytest.raises(TypeError):
543: (12)                     rg.standard_normal(out=existing, dtype=np.float64)
544: (8)                     existing = np.zeros(size, dtype=np.float32)
545: (8)                 with pytest.raises(TypeError):
546: (12)                     rg.standard_gamma(1.0, out=existing, dtype=np.float64)
547: (8)                 with pytest.raises(ValueError):
548: (12)                     rg.standard_gamma(1.0, out=existing[:3], dtype=np.float32)
549: (8)                     existing = np.zeros(size, dtype=np.float64)
550: (8)                 with pytest.raises(TypeError):
551: (12)                     rg.standard_gamma(1.0, out=existing, dtype=np.float32)
552: (8)                 with pytest.raises(ValueError):
553: (12)                     rg.standard_gamma(1.0, out=existing[:3])
554: (4)             def test_integers_broadcast(self, dtype):
555: (8)                 if dtype == np.bool_:
556: (12)                     upper = 2
557: (12)                     lower = 0
558: (8)                 else:
559: (12)                     info = np.iinfo(dtype)
560: (12)                     upper = int(info.max) + 1
561: (12)                     lower = info.min
562: (8)                     self._reset_state()
563: (8)                     a = self.rg.integers(lower, [upper] * 10, dtype=dtype)
564: (8)                     self._reset_state()
565: (8)                     b = self.rg.integers([lower] * 10, upper, dtype=dtype)
566: (8)                     assert_equal(a, b)
567: (8)                     self._reset_state()
568: (8)                     c = self.rg.integers(lower, upper, size=10, dtype=dtype)
569: (8)                     assert_equal(a, c)
570: (8)                     self._reset_state()
571: (8)                     d = self.rg.integers(np.array(
572: (12)                         [lower] * 10), np.array([upper], dtype=object), size=10,
573: (12)                         dtype=dtype)
574: (8)                     assert_equal(a, d)
575: (8)                     self._reset_state()
576: (8)                     e = self.rg.integers(

```

```

577: (12)           np.array([lower] * 10), np.array([upper] * 10), size=10,
578: (12)           dtype=dtype)
579: (8)            assert_equal(a, e)
580: (8)            self._reset_state()
581: (8)            a = self.rg.integers(0, upper, size=10, dtype=dtype)
582: (8)            self._reset_state()
583: (8)            b = self.rg.integers([upper] * 10, dtype=dtype)
584: (8)            assert_equal(a, b)
585: (4)             def test_integers_numpy(self, dtype):
586: (8)               high = np.array([1])
587: (8)               low = np.array([0])
588: (8)               out = self.rg.integers(low, high, dtype=dtype)
589: (8)               assert out.shape == (1,)
590: (8)               out = self.rg.integers(low[0], high, dtype=dtype)
591: (8)               assert out.shape == (1,)
592: (8)               out = self.rg.integers(low, high[0], dtype=dtype)
593: (8)               assert out.shape == (1,)
594: (4)             def test_integers_broadcast_errors(self, dtype):
595: (8)               if dtype == np.bool_:
596: (12)                 upper = 2
597: (12)                 lower = 0
598: (8)               else:
599: (12)                 info = np.iinfo(dtype)
600: (12)                 upper = int(info.max) + 1
601: (12)                 lower = info.min
602: (8)               with pytest.raises(ValueError):
603: (12)                 self.rg.integers(lower, [upper + 1] * 10, dtype=dtype)
604: (8)               with pytest.raises(ValueError):
605: (12)                 self.rg.integers(lower - 1, [upper] * 10, dtype=dtype)
606: (8)               with pytest.raises(ValueError):
607: (12)                 self.rg.integers([lower - 1], [upper] * 10, dtype=dtype)
608: (8)               with pytest.raises(ValueError):
609: (12)                 self.rg.integers([0], [0], dtype=dtype)
610: (0)              class TestMT19937(RNG):
611: (4)                @classmethod
612: (4)                def setup_class(cls):
613: (8)                  cls.bit_generator = MT19937
614: (8)                  cls.advance = None
615: (8)                  cls.seed = [2 ** 21 + 2 ** 16 + 2 ** 5 + 1]
616: (8)                  cls.rg = Generator(cls.bit_generator(*cls.seed))
617: (8)                  cls.initial_state = cls.rg.bit_generator.state
618: (8)                  cls.seed_vector_bits = 32
619: (8)                  cls._extra_setup()
620: (8)                  cls.seed_error = ValueError
621: (4)                def test_numpy_state(self):
622: (8)                  nprg = np.random.RandomState()
623: (8)                  nprg.standard_normal(99)
624: (8)                  state = nprg.get_state()
625: (8)                  self.rg.bit_generator.state = state
626: (8)                  state2 = self.rg.bit_generator.state
627: (8)                  assert_(state[1] == state2['state']['key']).all()
628: (8)                  assert_(state[2] == state2['state']['pos']))
629: (0)              class TestPhilox(RNG):
630: (4)                @classmethod
631: (4)                def setup_class(cls):
632: (8)                  cls.bit_generator = Philox
633: (8)                  cls.advance = 2**63 + 2**31 + 2**15 + 1
634: (8)                  cls.seed = [12345]
635: (8)                  cls.rg = Generator(cls.bit_generator(*cls.seed))
636: (8)                  cls.initial_state = cls.rg.bit_generator.state
637: (8)                  cls.seed_vector_bits = 64
638: (8)                  cls._extra_setup()
639: (0)              class TestSFC64(RNG):
640: (4)                @classmethod
641: (4)                def setup_class(cls):
642: (8)                  cls.bit_generator = SFC64
643: (8)                  cls.advance = None
644: (8)                  cls.seed = [12345]
645: (8)                  cls.rg = Generator(cls.bit_generator(*cls.seed))

```

```

646: (8)             cls.initial_state = cls.rg.bit_generator.state
647: (8)             cls.seed_vector_bits = 192
648: (8)             cls._extra_setup()
649: (0)          class TestPCG64(RNG):
650: (4)              @classmethod
651: (4)                  def setup_class(cls):
652: (8)                      cls.bit_generator = PCG64
653: (8)                      cls.advance = 2**63 + 2**31 + 2**15 + 1
654: (8)                      cls.seed = [12345]
655: (8)                      cls.rg = Generator(cls.bit_generator(*cls.seed))
656: (8)                      cls.initial_state = cls.rg.bit_generator.state
657: (8)                      cls.seed_vector_bits = 64
658: (8)                      cls._extra_setup()
659: (0)          class TestPCG64DXSM(RNG):
660: (4)              @classmethod
661: (4)                  def setup_class(cls):
662: (8)                      cls.bit_generator = PCG64DXSM
663: (8)                      cls.advance = 2**63 + 2**31 + 2**15 + 1
664: (8)                      cls.seed = [12345]
665: (8)                      cls.rg = Generator(cls.bit_generator(*cls.seed))
666: (8)                      cls.initial_state = cls.rg.bit_generator.state
667: (8)                      cls.seed_vector_bits = 64
668: (8)                      cls._extra_setup()
669: (0)          class TestDefaultRNG(RNG):
670: (4)              @classmethod
671: (4)                  def setup_class(cls):
672: (8)                      cls.bit_generator = PCG64
673: (8)                      cls.advance = 2**63 + 2**31 + 2**15 + 1
674: (8)                      cls.seed = [12345]
675: (8)                      cls.rg = np.random.default_rng(*cls.seed)
676: (8)                      cls.initial_state = cls.rg.bit_generator.state
677: (8)                      cls.seed_vector_bits = 64
678: (8)                      cls._extra_setup()
679: (4)                  def test_default_is_pcg64(self):
680: (8)                      assert_(isinstance(self.rg.bit_generator, PCG64))
681: (4)                  def test_seed(self):
682: (8)                      np.random.default_rng()
683: (8)                      np.random.default_rng(None)
684: (8)                      np.random.default_rng(12345)
685: (8)                      np.random.default_rng(0)
686: (8)                      np.random.default_rng(43660444402423911716352051725018508569)
687: (8)                      np.random.default_rng([43660444402423911716352051725018508569,
688: (31)                                         279705150948142787361475340226491943209])
689: (8)                      with pytest.raises(ValueError):
690: (12)                          np.random.default_rng(-1)
691: (8)                      with pytest.raises(ValueError):
692: (12)                          np.random.default_rng([12345, -1])

```

-----

## File 321 - \_\_init\_\_.py:

1: (0)

## File 322 - extending.py:

```

1: (0)          """
2: (0)          Use cffi to access any of the underlying C functions from distributions.h
3: (0)          """
4: (0)          import os
5: (0)          import numpy as np
6: (0)          import cffi
7: (0)          from .parse import parse_distributions_h
8: (0)          ffi = cffi.FFI()
9: (0)          inc_dir = os.path.join(np.get_include(), 'numpy')
10: (0)          ffi.cdef('''
11: (4)              typedef intptr_t npy_intp;

```

12/16/24, 6:27 PM

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```
12: (4)         typedef unsigned char npy_bool;
13: (0)         '''
14: (0)         parse_distributions_h(ffi, inc_dir)
15: (0)         lib = ffi.dlopen(np.random._generator.__file__)
16: (0)         bit_gen = np.random.PCG64()
17: (0)         rng = np.random.Generator(bit_gen)
18: (0)         state = bit_gen.state
19: (0)         interface = rng.bit_generator.cffi
20: (0)         n = 100
21: (0)         vals_cffi = ffi.new('double[%d]' % n)
22: (0)         lib.random_standard_normal_fill(interface.bit_generator, n, vals_cffi)
23: (0)         bit_gen.state = state
24: (0)         vals = rng.standard_normal(n)
25: (0)         for i in range(n):
26: (4)             assert vals[i] == vals_cffi[i]
```

---

File 323 - parse.py:

```
1: (0)         import os
2: (0)         def parse_distributions_h(ffi, inc_dir):
3: (4)             """
4: (4)             Parse distributions.h located in inc_dir for CFFI, filling in the ffi.cdef
5: (4)             Read the function declarations without the "#define ..." macros that will
6: (4)             be filled in when loading the library.
7: (4)             """
8: (4)             with open(os.path.join(inc_dir, 'random', 'bitgen.h')) as fid:
9: (8)                 s = []
10: (8)                 for line in fid:
11: (12)                     if line.strip().startswith('#'):
12: (16)                         continue
13: (12)                     s.append(line)
14: (8)                     ffi.cdef('\n'.join(s))
15: (4)                     with open(os.path.join(inc_dir, 'random', 'distributions.h')) as fid:
16: (8)                         s = []
17: (8)                         in_skip = 0
18: (8)                         ignoring = False
19: (8)                         for line in fid:
20: (12)                             if ignoring:
21: (16)                                 if line.strip().startswith('#endif'):
22: (20)                                     ignoring = False
23: (16)                                     continue
24: (12)                                     if line.strip().startswith('#ifdef __cplusplus'):
25: (16)                                         ignoring = True
26: (12)                                         if line.strip().startswith('#'):
27: (16)                                             continue
28: (12)                                             if line.strip().startswith('static inline'):
29: (16)                                                 in_skip += line.count('{')
30: (16)                                                 continue
31: (12)                                                 elif in_skip > 0:
32: (16)                                                     in_skip += line.count('{')
33: (16)                                                     in_skip -= line.count('}')
34: (16)                                                     continue
35: (12)                                                     line = line.replace('DECLDIR', '')
36: (12)                                                     line = line.replace('RAND_INT_TYPE', 'int64_t')
37: (12)                                                     s.append(line)
38: (8)                     ffi.cdef('\n'.join(s))
```

---

File 324 - extending.py:

```
1: (0)             import numpy as np
2: (0)             import numba as nb
3: (0)             from numpy.random import PCG64
4: (0)             from timeit import timeit
5: (0)             bit_gen = PCG64()
6: (0)             next_d = bit_gen.cffi.next_double
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

7: (0) state_addr = bit_gen.cffi.state_address
8: (0)
9: (4) def normals(n, state):
10: (4)     out = np.empty(n)
11: (8)     for i in range((n + 1) // 2):
12: (8)         x1 = 2.0 * next_d(state) - 1.0
13: (8)         x2 = 2.0 * next_d(state) - 1.0
14: (8)         r2 = x1 * x1 + x2 * x2
15: (12)         while r2 >= 1.0 or r2 == 0.0:
16: (12)             x1 = 2.0 * next_d(state) - 1.0
17: (12)             x2 = 2.0 * next_d(state) - 1.0
18: (12)             r2 = x1 * x1 + x2 * x2
19: (8)             f = np.sqrt(-2.0 * np.log(r2) / r2)
20: (8)             out[2 * i] = f * x1
21: (8)             if 2 * i + 1 < n:
22: (12)                 out[2 * i + 1] = f * x2
23: (0)     return out
24: (0) normalsj = nb.jit(normals, nopython=True)
25: (0) n = 10000
26: (0) def numbacall():
27: (4)     return normalsj(n, state_addr)
28: (0) rg = np.random.Generator(PCG64())
29: (0) def numpycall():
30: (4)     return rg.normal(size=n)
31: (0) r1 = numbacall()
32: (0) r2 = numpycall()
33: (0) assert r1.shape == (n,)
34: (0) assert r1.shape == r2.shape
35: (0) t1 = timeit(numbacall, number=1000)
36: (0) print(f'{t1:.2f} secs for {n} PCG64 (Numba/PCG64) gaussian randoms')
37: (0) t2 = timeit(numpycall, number=1000)
38: (0) print(f'{t2:.2f} secs for {n} PCG64 (NumPy/PCG64) gaussian randoms')
39: (0) next_u32 = bit_gen.ctypes.next_uint32
40: (0) ctypes_state = bit_gen.ctypes.state
41: (0) @nb.jit(nopython=True)
42: (0) def bounded_uint(lb, ub, state):
43: (4)     mask = delta = ub - lb
44: (4)     mask |= mask >> 1
45: (4)     mask |= mask >> 2
46: (4)     mask |= mask >> 4
47: (4)     mask |= mask >> 8
48: (4)     mask |= mask >> 16
49: (4)     val = next_u32(state) & mask
50: (8)     while val > delta:
51: (4)         val = next_u32(state) & mask
52: (0)     return lb + val
53: (0) print(bounded_uint(323, 2394691, ctypes_state.value))
54: (0) @nb.jit(nopython=True)
55: (0) def bounded_uints(lb, ub, n, state):
56: (4)     out = np.empty(n, dtype=np.uint32)
57: (4)     for i in range(n):
58: (8)         out[i] = bounded_uint(lb, ub, state)
58: (0) bounded_uints(323, 2394691, 10000000, ctypes_state.value)

```

---

File 325 - extending\_distributions.py:

```

1: (0) """
2: (0) Building the required library in this example requires a source distribution
3: (0) of NumPy or clone of the NumPy git repository since distributions.c is not
4: (0) included in binary distributions.
5: (0) On *nix, execute in numpy/random/src/distributions
6: (0) export ${PYTHON_VERSION}=3.8 # Python version
7: (0) export PYTHON_INCLUDE=#path to Python's include folder, usually \
8: (4)     ${PYTHON_HOME}/include/python${PYTHON_VERSION}
9: (0) export NUMPY_INCLUDE=#path to numpy's include folder, usually \
10: (4)      ${PYTHON_HOME}/lib/python${PYTHON_VERSION}/site-
packages/numpy/core/include
11: (0)      gcc -shared -o libdistributions.so -fPIC distributions.c \

```

```

12: (4)          -I${NUMPY_INCLUDE} -I${PYTHON_INCLUDE}
13: (0)          mv libdistributions.so ../../examples/numba/
14: (0)          On Windows
15: (0)          rem PYTHON_HOME and PYTHON_VERSION are setup dependent, this is an example
16: (0)          set PYTHON_HOME=c:\Anaconda
17: (0)          set PYTHON_VERSION=38
18: (0)          cl.exe /LD ./distributions.c -DDLL_EXPORT \
19: (4)          -I%PYTHON_HOME%\lib\site-packages\numpy\core\include \
20: (4)          -I%PYTHON_HOME%\include %PYTHON_HOME%\libs\python%PYTHON_VERSION%.lib
21: (0)          move distributions.dll ../../examples/numba/
22: (0)          """
23: (0)          import os
24: (0)          import numba as nb
25: (0)          import numpy as np
26: (0)          from cffi import FFI
27: (0)          from numpy.random import PCG64
28: (0)          ffi = FFI()
29: (0)          if os.path.exists('./distributions.dll'):
30: (4)              lib = ffi.dlopen('./distributions.dll')
31: (0)          elif os.path.exists('./libdistributions.so'):
32: (4)              lib = ffi.dlopen('./libdistributions.so')
33: (0)          else:
34: (4)              raise RuntimeError('Required DLL/so file was not found.')
35: (0)          ffi.cdef("""
36: (0)          double random_standard_normal(void *bitgen_state);
37: (0)          """)
38: (0)          x = PCG64()
39: (0)          xffi = x.cffi
40: (0)          bit_generator = xffi.bit_generator
41: (0)          random_standard_normal = lib.random_standard_normal
42: (0)          def normals(n, bit_generator):
43: (4)              out = np.empty(n)
44: (4)              for i in range(n):
45: (8)                  out[i] = random_standard_normal(bit_generator)
46: (4)              return out
47: (0)          normalsj = nb.jit(normals, nopython=True)
48: (0)          bit_generator_address = int(ffi.cast('uintptr_t', bit_generator))
49: (0)          norm = normalsj(1000, bit_generator_address)
50: (0)          print(norm[:12])

```

---

## File 326 - overrides.py:

```

1: (0)          """Tools for testing implementations of __array_function__ and ufunc overrides
2: (0)          """
3: (0)          from numpy.core.overrides import ARRAY_FUNCTIONS as __array_functions
4: (0)          from numpy import ufunc as __ufunc
5: (0)          import numpy.core.umath as __umath
6: (0)          def get_overridable_numpy_ufuncs():
7: (4)              """List all numpy ufuncs overridable via `__array_ufunc__`"""
8: (4)              Parameters
9: (4)              -----
10: (4)              None
11: (4)              Returns
12: (4)              -----
13: (4)              set
14: (8)                  A set containing all overridable ufuncs in the public numpy API.
15: (4)              """
16: (4)              ufuncs = {obj for obj in __umath.__dict__.values()
17: (14)                  if isinstance(obj, __ufunc)}
18: (4)              return ufuncs
19: (0)          def allows_array_ufunc_override(func):
20: (4)              """Determine if a function can be overridden via `__array_ufunc__`"""
21: (4)              Parameters
22: (4)              -----
23: (4)              func : callable
24: (8)                  Function that may be overridable via `__array_ufunc__`
25: (4)              Returns

```

```

26: (4)      -----
27: (4)      bool
28: (8)          `True` if `func` is overridable via `__array_ufunc__` and
29: (8)          `False` otherwise.
30: (4)      Notes
31: (4)      -----
32: (4)          This function is equivalent to ``isinstance(func, np.ufunc)`` and
33: (4)          will work correctly for ufuncs defined outside of Numpy.
34: (4)          """
35: (4)      return isinstance(func, np.ufunc)
36: (0)  def get_overridable_numpy_array_functions():
37: (4)      """List all numpy functions overridable via `__array_function__`"""
38: (4)      Parameters
39: (4)      -----
40: (4)      None
41: (4)      Returns
42: (4)      -----
43: (4)      set
44: (8)          A set containing all functions in the public numpy API that are
45: (8)          overridable via `__array_function__`.
46: (4)          """
47: (4)      from numpy.lib import recfunctions
48: (4)      return _array_functions.copy()
49: (0)  def allows_array_function_override(func):
50: (4)      """Determine if a Numpy function can be overridden via
`__array_function__`"""
51: (4)      Parameters
52: (4)      -----
53: (4)      func : callable
54: (8)          Function that may be overridable via `__array_function__`
55: (4)      Returns
56: (4)      -----
57: (4)      bool
58: (8)          `True` if `func` is a function in the Numpy API that is
59: (8)          overridable via `__array_function__` and `False` otherwise.
60: (4)          """
61: (4)      return func in _array_functions

```

---

### File 327 - print\_coercion\_tables.py:

```

1: (0)      """Prints type-coercion tables for the built-in NumPy types
2: (0)      """
3: (0)      import numpy as np
4: (0)      from collections import namedtuple
5: (0)      class GenericObject:
6: (4)          def __init__(self, v):
7: (8)              self.v = v
8: (4)          def __add__(self, other):
9: (8)              return self
10: (4)          def __radd__(self, other):
11: (8)              return self
12: (4)          dtype = np.dtype('O')
13: (0)      def print_cancast_table(ntypes):
14: (4)          print('X', end=' ')
15: (4)          for char in ntypes:
16: (8)              print(char, end=' ')
17: (4)          print()
18: (4)          for row in ntypes:
19: (8)              print(row, end=' ')
20: (8)              for col in ntypes:
21: (12)                  if np.can_cast(row, col, "equiv"):
22: (16)                      cast = "#"
23: (12)                  elif np.can_cast(row, col, "safe"):
24: (16)                      cast = "="
25: (12)                  elif np.can_cast(row, col, "same_kind"):
26: (16)                      cast = "~"
27: (12)                  elif np.can_cast(row, col, "unsafe"):
```

```

28: (16)           cast = "."
29: (12)         else:
30: (16)           cast = " "
31: (12)           print(cast, end=' ')
32: (8)         print()
33: (0)     def print_coercion_table(ntypes, inputfirstvalue, inputsecondvalue,
firstarray, use_promote_types=False):
34: (4)           print('+', end=' ')
35: (4)           for char in ntypes:
36: (8)             print(char, end=' ')
37: (4)           print()
38: (4)           for row in ntypes:
39: (8)             if row == '0':
40: (12)               rowtype = GenericObject
41: (8)             else:
42: (12)               rowtype = np.obj2sctype(row)
43: (8)             print(row, end=' ')
44: (8)             for col in ntypes:
45: (12)               if col == '0':
46: (16)                 coltype = GenericObject
47: (12)               else:
48: (16)                 coltype = np.obj2sctype(col)
49: (12)             try:
50: (16)               if firstarray:
51: (20)                 rowvalue = np.array([rowtype(inputfirstvalue)],
52: (16)                               dtype=rowtype)
53: (20)
54: (16)
55: (16)
56: (20)
57: (16)
58: (20)
59: (20)
60: (24)
61: (20)
62: (24)
63: (12)
64: (16)
65: (12)
66: (16)
67: (12)
68: (16)
69: (12)
70: (8)
71: (0)     def print_new_cast_table(*, can_cast=True, legacy=False, flags=False):
72: (4)       """Prints new casts, the values given are default "can-cast" values, not
73: (4)       actual ones.
74: (4)       """
75: (4)       from numpy.core._multiarray_tests import get_all_cast_information
76: (4)       cast_table = {
77: (8)         -1: " ",
78: (8)         0: "#", # No cast (classify as equivalent here)
79: (8)         1: "#", # equivalent casting
80: (8)         2: "=", # safe casting
81: (8)         3: "~", # same-kind casting
82: (8)         4: ".", # unsafe casting
83: (4)
84: (4)       flags_table = {
85: (8)         0 : "â--", 7: "â-^",
86: (8)         1: "â-š", 2: "â- ", 4: "â-„",
87: (16)           3: "â-  ", 5: "â-    ",
88: (24)             6: "â-    ",
89: (4)
90: (4)       cast_info = namedtuple("cast_info", ["can_cast", "legacy", "flags"])
91: (4)       no_cast_info = cast_info(" ", " ", " ")
92: (4)       casts = get_all_cast_information()
93: (4)       table = {}

```

```

94: (4)         dtypes = set()
95: (4)         for cast in casts:
96: (8)             dtypes.add(cast["from"])
97: (8)             dtypes.add(cast["to"])
98: (8)             if cast["from"] not in table:
99: (12)                 table[cast["from"]] = {}
100: (8)                to_dict = table[cast["from"]]
101: (8)                can_cast = cast_table[cast["casting"]]
102: (8)                legacy = "L" if cast["legacy"] else "."
103: (8)                flags = 0
104: (8)                if cast["requires_pyapi"]:
105: (12)                    flags |= 1
106: (8)                if cast["supports_unaligned"]:
107: (12)                    flags |= 2
108: (8)                if cast["no_floatingpoint_errors"]:
109: (12)                    flags |= 4
110: (8)                flags = flags_table[flags]
111: (8)                to_dict[cast["to"]] = cast_info(can_cast=can_cast, legacy=legacy,
flags=flags)
112: (4)        types = np.typecodes["All"]
113: (4)        def sorter(x):
114: (8)            dtype = np.dtype(x.type)
115: (8)            try:
116: (12)                indx = types.index(dtype.char)
117: (8)            except ValueError:
118: (12)                indx = np.inf
119: (8)            return (indx, dtype.char)
120: (4)        dtypes = sorted(dtypes, key=sorter)
121: (4)        def print_table(field="can_cast"):
122: (8)            print('X', end=' ')
123: (8)            for dt in dtypes:
124: (12)                print(np.dtype(dt.type).char, end=' ')
125: (8)            print()
126: (8)            for from_dt in dtypes:
127: (12)                print(np.dtype(from_dt.type).char, end=' ')
128: (12)                row = table.get(from_dt, {})
129: (12)                for to_dt in dtypes:
130: (16)                    print(getattr(row.get(to_dt, no_cast_info), field), end=' ')
131: (12)                print()
132: (4)            if can_cast:
133: (8)                print()
134: (8)                print("Casting: # is equivalent, = is safe, ~ is same-kind, and . is
unsafe")
135: (8)                print()
136: (8)                print_table("can_cast")
137: (4)            if legacy:
138: (8)                print()
139: (8)                print("L denotes a legacy cast . a non-legacy one.")
140: (8)                print()
141: (8)                print_table("legacy")
142: (4)            if flags:
143: (8)                print()
144: (8)                print(f"{flags_table[0]}: no flags, {flags_table[1]}: PyAPI, "
145: (14)                  f"{flags_table[2]}: supports unaligned, {flags_table[4]}: no-
float-errors")
146: (8)                    print()
147: (8)                    print_table("flags")
148: (0)            if __name__ == '__main__':
149: (4)                print("can cast")
150: (4)                print_cancast_table(np.typecodes['All'])
151: (4)                print()
152: (4)                print("In these tables, ValueError is '!', OverflowError is '@', TypeError
is '#'")
153: (4)                print()
154: (4)                print("scalar + scalar")
155: (4)                print_coercion_table(np.typecodes['All'], 0, 0, False)
156: (4)                print()
157: (4)                print("scalar + neg scalar")
158: (4)                print_coercion_table(np.typecodes['All'], 0, -1, False)

```

```

159: (4)         print()
160: (4)         print("array + scalar")
161: (4)         print_coercion_table(np.typecodes['All'], 0, 0, True)
162: (4)         print()
163: (4)         print("array + neg scalar")
164: (4)         print_coercion_table(np.typecodes['All'], 0, -1, True)
165: (4)         print()
166: (4)         print("promote_types")
167: (4)         print_coercion_table(np.typecodes['All'], 0, 0, False, True)
168: (4)         print("New casting type promotion:")
169: (4)         print_new_cast_table(can_cast=True, legacy=True, flags=True)
-----
```

## File 328 - setup.py:

```

1: (0)     def configuration(parent_package='', top_path=None):
2: (4)         from numpy.distutils.misc_util import Configuration
3: (4)         config = Configuration('testing', parent_package, top_path)
4: (4)         config.add_subpackage('_private')
5: (4)         config.add_subpackage('tests')
6: (4)         config.add_data_files('.pyi')
7: (4)         config.add_data_files('_private/*.pyi')
8: (4)         return config
9: (0)     if __name__ == '__main__':
10: (4)         from numpy.distutils.core import setup
11: (4)         setup(maintainer="NumPy Developers",
12: (10)             maintainer_email="numpy-dev@numpy.org",
13: (10)             description="NumPy test module",
14: (10)             url="https://www.numpy.org",
15: (10)             license="NumPy License (BSD Style)",
16: (10)             configuration=configuration,
17: (10)         )
-----
```

## File 329 - \_\_init\_\_.py:

```

1: (0)     """Common test support for all numpy test scripts.
2: (0)     This single module should provide all the common functionality for numpy tests
3: (0)     in a single location, so that test scripts can just import it and work right
4: (0)     away.
5: (0)     """
6: (0)     from unittest import TestCase
7: (0)     from . import _private
8: (0)     from ._private.utils import *
9: (0)     from ._private.utils import (_assert_valid_refcount, _gen_alignment_data)
10: (0)    from ._private import extbuild
11: (0)    from . import overrides
12: (0)    __all__ = (
13: (4)        _private.utils.__all__ + ['TestCase', 'overrides']
14: (0)    )
15: (0)    from numpy._pytesttester import PytestTester
16: (0)    test = PytestTester(__name__)
17: (0)    del PytestTester
-----
```

## File 330 - test\_utils.py:

```

1: (0)         import warnings
2: (0)         import sys
3: (0)         import os
4: (0)         import itertools
5: (0)         import pytest
6: (0)         import weakref
7: (0)         import numpy as np
8: (0)         from numpy.testing import (
9: (4)             assert_equal, assert_array_equal, assert_almost_equal,
-----
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

10: (4)           assert_array_almost_equal, assert_array_less, build_err_msg,
11: (4)           assert_raises, assert_warns, assert_no_warnings, assert_allclose,
12: (4)           assert_approx_equal, assert_array_almost_equal_nulp, assert_array_max_ulp,
13: (4)           clear_and_catch_warnings, suppress_warnings, assert_string_equal, assert_,
14: (4)           tempdir, temppath, assert_no_gc_cycles, HAS_REFCOUNT
15: (4)
16: (0) class _GenericTest:
17: (4)     def _test_equal(self, a, b):
18: (8)         self._assert_func(a, b)
19: (4)     def _test_not_equal(self, a, b):
20: (8)         with assert_raises(AssertionError):
21: (12)             self._assert_func(a, b)
22: (4)     def test_array_rank1_eq(self):
23: (8)         """Test two equal array of rank 1 are found equal."""
24: (8)         a = np.array([1, 2])
25: (8)         b = np.array([1, 2])
26: (8)         self._test_equal(a, b)
27: (4)     def test_array_rank1_noteq(self):
28: (8)         """Test two different array of rank 1 are found not equal."""
29: (8)         a = np.array([1, 2])
30: (8)         b = np.array([2, 2])
31: (8)         self._test_not_equal(a, b)
32: (4)     def test_array_rank2_eq(self):
33: (8)         """Test two equal array of rank 2 are found equal."""
34: (8)         a = np.array([[1, 2], [3, 4]])
35: (8)         b = np.array([[1, 2], [3, 4]])
36: (8)         self._test_equal(a, b)
37: (4)     def test_array_diffshape(self):
38: (8)         """Test two arrays with different shapes are found not equal."""
39: (8)         a = np.array([1, 2])
40: (8)         b = np.array([[1, 2], [1, 2]])
41: (8)         self._test_not_equal(a, b)
42: (4)     def test_objarray(self):
43: (8)         """Test object arrays."""
44: (8)         a = np.array([1, 1], dtype=object)
45: (8)         self._test_equal(a, 1)
46: (4)     def test_array_likes(self):
47: (8)         self._test_equal([1, 2, 3], (1, 2, 3))
48: (0) class TestArrayEqual(_GenericTest):
49: (4)     def setup_method(self):
50: (8)         self._assert_func = assert_array_equal
51: (4)     def test_generic_rank1(self):
52: (8)         """Test rank 1 array for all dtypes."""
53: (8)         def foo(t):
54: (12)             a = np.empty(2, t)
55: (12)             a.fill(1)
56: (12)             b = a.copy()
57: (12)             c = a.copy()
58: (12)             c.fill(0)
59: (12)             self._test_equal(a, b)
60: (12)             self._test_not_equal(c, b)
61: (8)             for t in '?bhilqpBHILQPfdgFDG':
62: (12)                 foo(t)
63: (8)             for t in ['S1', 'U1']:
64: (12)                 foo(t)
65: (4)         def test_0_ndim_array(self):
66: (8)             x = np.array(473963742225900817127911193656584771)
67: (8)             y = np.array(18535119325151578301457182298393896)
68: (8)             assert_raises(AssertionError, self._assert_func, x, y)
69: (8)             y = x
70: (8)             self._assert_func(x, y)
71: (8)             x = np.array(43)
72: (8)             y = np.array(10)
73: (8)             assert_raises(AssertionError, self._assert_func, x, y)
74: (8)             y = x
75: (8)             self._assert_func(x, y)
76: (4)         def test_generic_rank3(self):
77: (8)             """Test rank 3 array for all dtypes."""
78: (8)             def foo(t):

```

```

79: (12)             a = np.empty((4, 2, 3), t)
80: (12)             a.fill(1)
81: (12)             b = a.copy()
82: (12)             c = a.copy()
83: (12)             c.fill(0)
84: (12)             self._test_equal(a, b)
85: (12)             self._test_not_equal(c, b)
86: (8)              for t in '?bhilqpBHILQPfdgFDG':
87: (12)                  foo(t)
88: (8)              for t in ['S1', 'U1']:
89: (12)                  foo(t)
90: (4)              def test_nan_array(self):
91: (8)                  """Test arrays with nan values in them."""
92: (8)                  a = np.array([1, 2, np.nan])
93: (8)                  b = np.array([1, 2, np.nan])
94: (8)                  self._test_equal(a, b)
95: (8)                  c = np.array([1, 2, 3])
96: (8)                  self._test_not_equal(c, b)
97: (4)              def test_string_arrays(self):
98: (8)                  """Test two arrays with different shapes are found not equal."""
99: (8)                  a = np.array(['floupi', 'floupa'])
100: (8)                 b = np.array(['floupi', 'floupa'])
101: (8)                 self._test_equal(a, b)
102: (8)                 c = np.array(['floupipipi', 'floupa'])
103: (8)                 self._test_not_equal(c, b)
104: (4)              def test_recarrays(self):
105: (8)                  """Test record arrays."""
106: (8)                  a = np.empty(2, [('floupi', float), ('floupa', float)])
107: (8)                  a['floupi'] = [1, 2]
108: (8)                  a['floupa'] = [1, 2]
109: (8)                  b = a.copy()
110: (8)                  self._test_equal(a, b)
111: (8)                  c = np.empty(2, [('floupipipi', float),
112: (25)                               ('floupi', float), ('floupa', float)])
113: (8)                  c['floupipipi'] = a['floupi'].copy()
114: (8)                  c['floupa'] = a['floupa'].copy()
115: (8)                  with pytest.raises(TypeError):
116: (12)                      self._test_not_equal(c, b)
117: (4)              def test_masked_nan_inf(self):
118: (8)                  a = np.ma.MaskedArray([3., 4., 6.5], mask=[False, True, False])
119: (8)                  b = np.array([3., np.nan, 6.5])
120: (8)                  self._test_equal(a, b)
121: (8)                  self._test_equal(b, a)
122: (8)                  a = np.ma.MaskedArray([3., 4., 6.5], mask=[True, False, False])
123: (8)                  b = np.array([np.inf, 4., 6.5])
124: (8)                  self._test_equal(a, b)
125: (8)                  self._test_equal(b, a)
126: (4)              def test_subclass_that_overrides_eq(self):
127: (8)                  class MyArray(np.ndarray):
128: (12)                      def __eq__(self, other):
129: (16)                          return bool(np.equal(self, other).all())
130: (12)                      def __ne__(self, other):
131: (16)                          return not self == other
132: (8)                      a = np.array([1., 2.]).view(MyArray)
133: (8)                      b = np.array([2., 3.]).view(MyArray)
134: (8)                      assert_(type(a == a), bool)
135: (8)                      assert_(a == a)
136: (8)                      assert_(a != b)
137: (8)                      self._test_equal(a, a)
138: (8)                      self._test_not_equal(a, b)
139: (8)                      self._test_not_equal(b, a)
140: (4)              def test_subclass_that_does_not_implement_npall(self):
141: (8)                  class MyArray(np.ndarray):
142: (12)                      def __array_function__(self, *args, **kwargs):
143: (16)                          return NotImplemented
144: (8)                      a = np.array([1., 2.]).view(MyArray)
145: (8)                      b = np.array([2., 3.]).view(MyArray)
146: (8)                      with assert_raises(TypeError):
147: (12)                          np.all(a)

```

```

148: (8)             self._test_equal(a, a)
149: (8)             self._test_not_equal(a, b)
150: (8)             self._test_not_equal(b, a)
151: (4)             def test_suppress_overflow_warnings(self):
152: (8)                 with pytest.raises(AssertionError):
153: (12)                     with np.errstate(all="raise"):
154: (16)                         np.testing.assert_array_equal(
155: (20)                             np.array([1, 2, 3], np.float32),
156: (20)                             np.array([1, 1e-40, 3], np.float32))
157: (4)             def test_array_vs_scalar_is_equal(self):
158: (8)                 """Test comparing an array with a scalar when all values are equal."""
159: (8)                 a = np.array([1., 1., 1.])
160: (8)                 b = 1.
161: (8)                 self._test_equal(a, b)
162: (4)             def test_array_vs_scalar_not_equal(self):
163: (8)                 """Test comparing an array with a scalar when not all values equal."""
164: (8)                 a = np.array([1., 2., 3.])
165: (8)                 b = 1.
166: (8)                 self._test_not_equal(a, b)
167: (4)             def test_array_vs_scalar_strict(self):
168: (8)                 """Test comparing an array with a scalar with strict option."""
169: (8)                 a = np.array([1., 1., 1.])
170: (8)                 b = 1.
171: (8)                 with pytest.raises(AssertionError):
172: (12)                     assert_array_equal(a, b, strict=True)
173: (4)             def test_array_vs_array_strict(self):
174: (8)                 """Test comparing two arrays with strict option."""
175: (8)                 a = np.array([1., 1., 1.])
176: (8)                 b = np.array([1., 1., 1.])
177: (8)                 assert_array_equal(a, b, strict=True)
178: (4)             def test_array_vs_float_array_strict(self):
179: (8)                 """Test comparing two arrays with strict option."""
180: (8)                 a = np.array([1, 1, 1])
181: (8)                 b = np.array([1., 1., 1.])
182: (8)                 with pytest.raises(AssertionError):
183: (12)                     assert_array_equal(a, b, strict=True)
184: (0)             class TestBuildErrorMessage:
185: (4)                 def test_build_err_msg_defaults(self):
186: (8)                     x = np.array([1.00001, 2.00002, 3.00003])
187: (8)                     y = np.array([1.00002, 2.00003, 3.00004])
188: (8)                     err_msg = 'There is a mismatch'
189: (8)                     a = build_err_msg([x, y], err_msg)
190: (8)                     b = ('\nItems are not equal: There is a mismatch\n ACTUAL: array(['
191: (13)                         '1.00001, 2.00002, 3.00003])\n DESIRED: array([1.00002,
192: (13)                         '2.00003, 3.00004])')
193: (8)                     assert_equal(a, b)
194: (4)                 def test_build_err_msg_no_verbose(self):
195: (8)                     x = np.array([1.00001, 2.00002, 3.00003])
196: (8)                     y = np.array([1.00002, 2.00003, 3.00004])
197: (8)                     err_msg = 'There is a mismatch'
198: (8)                     a = build_err_msg([x, y], err_msg, verbose=False)
199: (8)                     b = '\nItems are not equal: There is a mismatch'
200: (8)                     assert_equal(a, b)
201: (4)                 def test_build_err_msg_custom_names(self):
202: (8)                     x = np.array([1.00001, 2.00002, 3.00003])
203: (8)                     y = np.array([1.00002, 2.00003, 3.00004])
204: (8)                     err_msg = 'There is a mismatch'
205: (8)                     a = build_err_msg([x, y], err_msg, names=('FOO', 'BAR'))
206: (8)                     b = ('\nItems are not equal: There is a mismatch\n FOO: array([
207: (13)                         '1.00001, 2.00002, 3.00003])\n BAR: array([1.00002, 2.00003,
208: (13)                         '3.00004])')
209: (8)                     assert_equal(a, b)
210: (4)                 def test_build_err_msg_custom_precision(self):
211: (8)                     x = np.array([1.000000001, 2.00002, 3.00003])
212: (8)                     y = np.array([1.000000002, 2.00003, 3.00004])
213: (8)                     err_msg = 'There is a mismatch'
214: (8)                     a = build_err_msg([x, y], err_msg, precision=10)
215: (8)                     b = ('\nItems are not equal: There is a mismatch\n ACTUAL: array([
216: (13)                         '1.000000001, 2.00002      , 3.00003      ]) \n DESIRED: array([

```

```

217: (13)           '1.000000002, 2.00003 , 3.00004 ])')
218: (8)             assert_equal(a, b)
219: (0)             class TestEqual(TestArrayEqual):
220: (4)               def setup_method(self):
221: (8)                 self._assert_func = assert_equal
222: (4)               def test_nan_items(self):
223: (8)                 self._assert_func(np.nan, np.nan)
224: (8)                 self._assert_func([np.nan], [np.nan])
225: (8)                 self._test_not_equal(np.nan, [np.nan])
226: (8)                 self._test_not_equal(np.nan, 1)
227: (4)               def test_inf_items(self):
228: (8)                 self._assert_func(np.inf, np.inf)
229: (8)                 self._assert_func([np.inf], [np.inf])
230: (8)                 self._test_not_equal(np.inf, [np.inf])
231: (4)               def test_datetime(self):
232: (8)                 self._test_equal(
233: (12)                   np.datetime64("2017-01-01", "s"),
234: (12)                   np.datetime64("2017-01-01", "s")
235: (8)               )
236: (8)               self._test_equal(
237: (12)                   np.datetime64("2017-01-01", "s"),
238: (12)                   np.datetime64("2017-01-01", "m")
239: (8)               )
240: (8)               self._test_not_equal(
241: (12)                   np.datetime64("2017-01-01", "s"),
242: (12)                   np.datetime64("2017-01-02", "s")
243: (8)               )
244: (8)               self._test_not_equal(
245: (12)                   np.datetime64("2017-01-01", "s"),
246: (12)                   np.datetime64("2017-01-02", "m")
247: (8)               )
248: (4)             def test_nat_items(self):
249: (8)               nadt_no_unit = np.datetime64("NaT")
250: (8)               nadt_s = np.datetime64("NaT", "s")
251: (8)               nadt_d = np.datetime64("NaT", "ns")
252: (8)               natd_no_unit = np.timedelta64("NaT")
253: (8)               natd_s = np.timedelta64("NaT", "s")
254: (8)               natd_d = np.timedelta64("NaT", "ns")
255: (8)               dts = [nadt_no_unit, nadt_s, nadt_d]
256: (8)               tds = [natd_no_unit, natd_s, natd_d]
257: (8)               for a, b in itertools.product(dts, dts):
258: (12)                 self._assert_func(a, b)
259: (12)                 self._assert_func([a], [b])
260: (12)                 self._test_not_equal([a], b)
261: (8)               for a, b in itertools.product(tds, tds):
262: (12)                 self._assert_func(a, b)
263: (12)                 self._assert_func([a], [b])
264: (12)                 self._test_not_equal([a], b)
265: (8)               for a, b in itertools.product(tds, dts):
266: (12)                 self._test_not_equal(a, b)
267: (12)                 self._test_not_equal(a, [b])
268: (12)                 self._test_not_equal([a], [b])
269: (12)                 self._test_not_equal([a], np.datetime64("2017-01-01", "s"))
270: (12)                 self._test_not_equal([b], np.datetime64("2017-01-01", "s"))
271: (12)                 self._test_not_equal([a], np.timedelta64(123, "s"))
272: (12)                 self._test_not_equal([b], np.timedelta64(123, "s"))
273: (4)             def test_non_numeric(self):
274: (8)               self._assert_func('ab', 'ab')
275: (8)               self._test_not_equal('ab', 'abb')
276: (4)             def test_complex_item(self):
277: (8)               self._assert_func(complex(1, 2), complex(1, 2))
278: (8)               self._assert_func(complex(1, np.nan), complex(1, np.nan))
279: (8)               self._test_not_equal(complex(1, np.nan), complex(1, 2))
280: (8)               self._test_not_equal(complex(np.nan, 1), complex(1, np.nan))
281: (8)               self._test_not_equal(complex(np.nan, np.inf), complex(np.nan, 2))
282: (4)             def test_negative_zero(self):
283: (8)               self._test_not_equal(np.PZERO, np.NZERO)
284: (4)             def test_complex(self):
285: (8)               x = np.array([complex(1, 2), complex(1, np.nan)])

```

```

286: (8)             y = np.array([complex(1, 2), complex(1, 2)])
287: (8)             self._assert_func(x, x)
288: (8)             self._test_not_equal(x, y)
289: (4)             def test_object(self):
290: (8)                 import datetime
291: (8)                 a = np.array([datetime.datetime(2000, 1, 1),
292: (22)                               datetime.datetime(2000, 1, 2)])
293: (8)                 self._test_not_equal(a, a[::-1])
294: (0)             class TestArrayAlmostEqual(_GenericTest):
295: (4)                 def setup_method(self):
296: (8)                     self._assert_func = assert_array_almost_equal
297: (4)                 def test_closeness(self):
298: (8)                     self._assert_func(1.499999, 0.0, decimal=0)
299: (8)                     assert_raises(AssertionError,
300: (26)                         lambda: self._assert_func(1.5, 0.0, decimal=0))
301: (8)                     self._assert_func([1.499999], [0.0], decimal=0)
302: (8)                     assert_raises(AssertionError,
303: (26)                         lambda: self._assert_func([1.5], [0.0], decimal=0))
304: (4)                 def test_simple(self):
305: (8)                     x = np.array([1234.2222])
306: (8)                     y = np.array([1234.2223])
307: (8)                     self._assert_func(x, y, decimal=3)
308: (8)                     self._assert_func(x, y, decimal=4)
309: (8)                     assert_raises(AssertionError,
310: (16)                         lambda: self._assert_func(x, y, decimal=5))
311: (4)                 def test_nan(self):
312: (8)                     anan = np.array([np.nan])
313: (8)                     aone = np.array([1])
314: (8)                     ainf = np.array([np.inf])
315: (8)                     self._assert_func(anan, anan)
316: (8)                     assert_raises(AssertionError,
317: (16)                         lambda: self._assert_func(anan, aone))
318: (8)                     assert_raises(AssertionError,
319: (16)                         lambda: self._assert_func(anan, ainf))
320: (8)                     assert_raises(AssertionError,
321: (16)                         lambda: self._assert_func(ainf, anan))
322: (4)                 def test_inf(self):
323: (8)                     a = np.array([[1., 2.], [3., 4.]])
324: (8)                     b = a.copy()
325: (8)                     a[0, 0] = np.inf
326: (8)                     assert_raises(AssertionError,
327: (16)                         lambda: self._assert_func(a, b))
328: (8)                     b[0, 0] = -np.inf
329: (8)                     assert_raises(AssertionError,
330: (16)                         lambda: self._assert_func(a, b))
331: (4)                 def test_subclass(self):
332: (8)                     a = np.array([[1., 2.], [3., 4.]])
333: (8)                     b = np.ma.masked_array([[1., 2.], [0., 4.]],
334: (31)                           [[False, False], [True, False]])
335: (8)                     self._assert_func(a, b)
336: (8)                     self._assert_func(b, a)
337: (8)                     self._assert_func(b, b)
338: (8)                     a = np.ma.MaskedArray(3.5, mask=True)
339: (8)                     b = np.array([3., 4., 6.5])
340: (8)                     self._test_equal(a, b)
341: (8)                     self._test_equal(b, a)
342: (8)                     a = np.ma.masked
343: (8)                     b = np.array([3., 4., 6.5])
344: (8)                     self._test_equal(a, b)
345: (8)                     self._test_equal(b, a)
346: (8)                     a = np.ma.MaskedArray([3., 4., 6.5], mask=[True, True, True])
347: (8)                     b = np.array([1., 2., 3.])
348: (8)                     self._test_equal(a, b)
349: (8)                     self._test_equal(b, a)
350: (8)                     a = np.ma.MaskedArray([3., 4., 6.5], mask=[True, True, True])
351: (8)                     b = np.array(1.)
352: (8)                     self._test_equal(a, b)
353: (8)                     self._test_equal(b, a)
354: (4)             def test_subclass_that_CANNOT_be_bool(self):

```

```

355: (8)
356: (12)
357: (16)
358: (12)
359: (16)
360: (12)
361: (16)
362: (8)
363: (8)
364: (0)
365: (4)
366: (8)
367: (4)
368: (8)
369: (8)
370: (22)
371: (8)
372: (8)
373: (22)
374: (4)
375: (8)
376: (8)
377: (22)
378: (8)
379: (22)
380: (8)
381: (22)
382: (4)
383: (8)
384: (8)
385: (8)
386: (22)
387: (8)
388: (22)
389: (4)
390: (8)
391: (4)
392: (8)
393: (8)
394: (8)
395: (8)
396: (8)
397: (8)
398: (4)
399: (8)
400: (8)
401: (8)
402: (8)
403: (8)
404: (8)
405: (4)
406: (8)
407: (11)
408: (8)
409: (8)
410: (8)
411: (12)
412: (8)
413: (8)
414: (8)
415: (8)
416: (8)
417: (12)
418: (12)
419: (8)
420: (12)
421: (12)
422: (8)
423: (12)

            class MyArray(np.ndarray):
                def __eq__(self, other):
                    return super().__eq__(other).view(np.ndarray)
                def __lt__(self, other):
                    return super().__lt__(other).view(np.ndarray)
                def all(self, *args, **kwargs):
                    raise NotImplementedError
                a = np.array([1., 2.]).view(MyArray)
                self._assert_func(a, a)
            class TestAlmostEqual(_GenericTest):
                def setup_method(self):
                    self._assert_func = assert_almost_equal
                def test_closeness(self):
                    self._assert_func(1.499999, 0.0, decimal=0)
                    assert_raises(AssertionError,
                                  lambda: self._assert_func(1.5, 0.0, decimal=0))
                    self._assert_func([1.499999], [0.0], decimal=0)
                    assert_raises(AssertionError,
                                  lambda: self._assert_func([1.5], [0.0], decimal=0))
                def test_nan_item(self):
                    self._assert_func(np.nan, np.nan)
                    assert_raises(AssertionError,
                                  lambda: self._assert_func(np.nan, 1))
                    assert_raises(AssertionError,
                                  lambda: self._assert_func(np.nan, np.inf))
                    assert_raises(AssertionError,
                                  lambda: self._assert_func(np.inf, np.nan))
                def test_inf_item(self):
                    self._assert_func(np.inf, np.inf)
                    self._assert_func(-np.inf, -np.inf)
                    assert_raises(AssertionError,
                                  lambda: self._assert_func(np.inf, 1))
                    assert_raises(AssertionError,
                                  lambda: self._assert_func(-np.inf, np.inf))
                def test_simple_item(self):
                    self._test_not_equal(1, 2)
                def test_complex_item(self):
                    self._assert_func(complex(1, 2), complex(1, 2))
                    self._assert_func(complex(1, np.nan), complex(1, np.nan))
                    self._assert_func(complex(np.inf, np.nan), complex(np.inf, np.nan))
                    self._test_not_equal(complex(1, np.nan), complex(1, 2))
                    self._test_not_equal(complex(np.nan, 1), complex(1, np.nan))
                    self._test_not_equal(complex(np.nan, np.inf), complex(np.nan, 2))
                def test_complex(self):
                    x = np.array([complex(1, 2), complex(1, np.nan)])
                    z = np.array([complex(1, 2), complex(np.nan, 1)])
                    y = np.array([complex(1, 2), complex(1, 2)])
                    self._assert_func(x, x)
                    self._test_not_equal(x, y)
                    self._test_not_equal(x, z)
                def test_error_message(self):
                    """Check the message is formatted correctly for the decimal value.
                    Also check the message when input includes inf or nan (gh12200)"""
                    x = np.array([1.0000000001, 2.0000000002, 3.0003])
                    y = np.array([1.0000000002, 2.0000000003, 3.0004])
                    with pytest.raises(AssertionError) as exc_info:
                        self._assert_func(x, y, decimal=12)
                    msgs = str(exc_info.value).split('\n')
                    assert_equal(msgs[3], 'Mismatched elements: 3 / 3 (100%)')
                    assert_equal(msgs[4], 'Max absolute difference: 1.e-05')
                    assert_equal(msgs[5], 'Max relative difference: 3.33328889e-06')
                    assert_equal(
                        msgs[6],
                        ' x: array([1.0000000001, 2.0000000002, 3.0003           ])')
                    assert_equal(
                        msgs[7],
                        ' y: array([1.0000000002, 2.0000000003, 3.0004           ])')
                    with pytest.raises(AssertionError) as exc_info:
                        self._assert_func(x, y)

```

```

424: (8)                         msgs = str(exc_info.value).split('\n')
425: (8)                         assert_equal(msgs[3], 'Mismatched elements: 1 / 3 (33.3%)')
426: (8)                         assert_equal(msgs[4], 'Max absolute difference: 1.e-05')
427: (8)                         assert_equal(msgs[5], 'Max relative difference: 3.33328889e-06')
428: (8)                         assert_equal(msgs[6], ' x: array([1.        , 2.        , 3.00003]))')
429: (8)                         assert_equal(msgs[7], ' y: array([1.        , 2.        , 3.00004]))')
430: (8)                         x = np.array([np.inf, 0])
431: (8)                         y = np.array([np.inf, 1])
432: (8)                         with pytest.raises(AssertionError) as exc_info:
433: (12)                             self._assert_func(x, y)
434: (8)                         msgs = str(exc_info.value).split('\n')
435: (8)                         assert_equal(msgs[3], 'Mismatched elements: 1 / 2 (50%)')
436: (8)                         assert_equal(msgs[4], 'Max absolute difference: 1.')
437: (8)                         assert_equal(msgs[5], 'Max relative difference: 1.')
438: (8)                         assert_equal(msgs[6], ' x: array([inf,  0.]))')
439: (8)                         assert_equal(msgs[7], ' y: array([inf,  1.]))')
440: (8)                         x = np.array([1, 2])
441: (8)                         y = np.array([0, 0])
442: (8)                         with pytest.raises(AssertionError) as exc_info:
443: (12)                             self._assert_func(x, y)
444: (8)                         msgs = str(exc_info.value).split('\n')
445: (8)                         assert_equal(msgs[3], 'Mismatched elements: 2 / 2 (100%)')
446: (8)                         assert_equal(msgs[4], 'Max absolute difference: 2')
447: (8)                         assert_equal(msgs[5], 'Max relative difference: inf')
448: (4)                         def test_error_message_2(self):
449: (8)                             """Check the message is formatted correctly when either x or y is a
scalar."""
450: (8)                             x = 2
451: (8)                             y = np.ones(20)
452: (8)                             with pytest.raises(AssertionError) as exc_info:
453: (12)                                 self._assert_func(x, y)
454: (8)                             msgs = str(exc_info.value).split('\n')
455: (8)                             assert_equal(msgs[3], 'Mismatched elements: 20 / 20 (100%)')
456: (8)                             assert_equal(msgs[4], 'Max absolute difference: 1.')
457: (8)                             assert_equal(msgs[5], 'Max relative difference: 1.')
458: (8)                             y = 2
459: (8)                             x = np.ones(20)
460: (8)                             with pytest.raises(AssertionError) as exc_info:
461: (12)                                 self._assert_func(x, y)
462: (8)                             msgs = str(exc_info.value).split('\n')
463: (8)                             assert_equal(msgs[3], 'Mismatched elements: 20 / 20 (100%)')
464: (8)                             assert_equal(msgs[4], 'Max absolute difference: 1.')
465: (8)                             assert_equal(msgs[5], 'Max relative difference: 0.5')
466: (4)                         def test_subclass_that_cannot_be_bool(self):
467: (8)                             class MyArray(np.ndarray):
468: (12)                                 def __eq__(self, other):
469: (16)                                     return super().__eq__(other).view(np.ndarray)
470: (12)                                 def __lt__(self, other):
471: (16)                                     return super().__lt__(other).view(np.ndarray)
472: (12)                                 def all(self, *args, **kwargs):
473: (16)                                     raise NotImplementedError
474: (8)                                 a = np.array([1., 2.]).view(MyArray)
475: (8)                                 self._assert_func(a, a)
476: (0)                         class TestApproxEqual:
477: (4)                             def setup_method(self):
478: (8)                                 self._assert_func = assert_approx_equal
479: (4)                             def test_simple_0d_arrays(self):
480: (8)                                 x = np.array(1234.22)
481: (8)                                 y = np.array(1234.23)
482: (8)                                 self._assert_func(x, y, significant=5)
483: (8)                                 self._assert_func(x, y, significant=6)
484: (8)                                 assert_raises(AssertionError,
485: (22)                                     lambda: self._assert_func(x, y, significant=7))
486: (4)                             def test_simple_items(self):
487: (8)                                 x = 1234.22
488: (8)                                 y = 1234.23
489: (8)                                 self._assert_func(x, y, significant=4)
490: (8)                                 self._assert_func(x, y, significant=5)
491: (8)                                 self._assert_func(x, y, significant=6)

```

```

492: (8) assert_raises(AssertionError,
493: (22)         lambda: self._assert_func(x, y, significant=7))
494: (4) def test_nan_array(self):
495: (8)     anan = np.array(np.nan)
496: (8)     aone = np.array(1)
497: (8)     ainf = np.array(np.inf)
498: (8)     self._assert_func(anan, anan)
499: (8)     assert_raises(AssertionError, lambda: self._assert_func(anan, aone))
500: (8)     assert_raises(AssertionError, lambda: self._assert_func(anan, ainf))
501: (8)     assert_raises(AssertionError, lambda: self._assert_func(ainf, anan))
502: (4) def test_nan_items(self):
503: (8)     anan = np.array(np.nan)
504: (8)     aone = np.array(1)
505: (8)     ainf = np.array(np.inf)
506: (8)     self._assert_func(anan, anan)
507: (8)     assert_raises(AssertionError, lambda: self._assert_func(anan, aone))
508: (8)     assert_raises(AssertionError, lambda: self._assert_func(anan, ainf))
509: (8)     assert_raises(AssertionError, lambda: self._assert_func(ainf, anan))
510: (0) class TestArrayAssertLess:
511: (4)     def setup_method(self):
512: (8)         self._assert_func = assert_array_less
513: (4)     def test_simple_arrays(self):
514: (8)         x = np.array([1.1, 2.2])
515: (8)         y = np.array([1.2, 2.3])
516: (8)         self._assert_func(x, y)
517: (8)         assert_raises(AssertionError, lambda: self._assert_func(y, x))
518: (8)         y = np.array([1.0, 2.3])
519: (8)         assert_raises(AssertionError, lambda: self._assert_func(x, y))
520: (8)         assert_raises(AssertionError, lambda: self._assert_func(y, x))
521: (4)     def test_rank2(self):
522: (8)         x = np.array([[1.1, 2.2], [3.3, 4.4]])
523: (8)         y = np.array([[1.2, 2.3], [3.4, 4.5]])
524: (8)         self._assert_func(x, y)
525: (8)         assert_raises(AssertionError, lambda: self._assert_func(y, x))
526: (8)         y = np.array([[1.0, 2.3], [3.4, 4.5]])
527: (8)         assert_raises(AssertionError, lambda: self._assert_func(x, y))
528: (8)         assert_raises(AssertionError, lambda: self._assert_func(y, x))
529: (4)     def test_rank3(self):
530: (8)         x = np.ones(shape=(2, 2, 2))
531: (8)         y = np.ones(shape=(2, 2, 2))+1
532: (8)         self._assert_func(x, y)
533: (8)         assert_raises(AssertionError, lambda: self._assert_func(y, x))
534: (8)         y[0, 0, 0] = 0
535: (8)         assert_raises(AssertionError, lambda: self._assert_func(x, y))
536: (8)         assert_raises(AssertionError, lambda: self._assert_func(y, x))
537: (4)     def test_simple_items(self):
538: (8)         x = 1.1
539: (8)         y = 2.2
540: (8)         self._assert_func(x, y)
541: (8)         assert_raises(AssertionError, lambda: self._assert_func(y, x))
542: (8)         y = np.array([2.2, 3.3])
543: (8)         self._assert_func(x, y)
544: (8)         assert_raises(AssertionError, lambda: self._assert_func(y, x))
545: (8)         y = np.array([1.0, 3.3])
546: (8)         assert_raises(AssertionError, lambda: self._assert_func(x, y))
547: (4)     def test_nan_noncompare(self):
548: (8)         anan = np.array(np.nan)
549: (8)         aone = np.array(1)
550: (8)         ainf = np.array(np.inf)
551: (8)         self._assert_func(anan, anan)
552: (8)         assert_raises(AssertionError, lambda: self._assert_func(aone, anan))
553: (8)         assert_raises(AssertionError, lambda: self._assert_func(anan, aone))
554: (8)         assert_raises(AssertionError, lambda: self._assert_func(anan, ainf))
555: (8)         assert_raises(AssertionError, lambda: self._assert_func(ainf, anan))
556: (4)     def test_nan_noncompare_array(self):
557: (8)         x = np.array([1.1, 2.2, 3.3])
558: (8)         anan = np.array(np.nan)
559: (8)         assert_raises(AssertionError, lambda: self._assert_func(x, anan))
560: (8)         assert_raises(AssertionError, lambda: self._assert_func(anan, x))

```

```

561: (8)             x = np.array([1.1, 2.2, np.nan])
562: (8)             assert_raises(AssertionError, lambda: self._assert_func(x, anan))
563: (8)             assert_raises(AssertionError, lambda: self._assert_func(anan, x))
564: (8)             y = np.array([1.0, 2.0, np.nan])
565: (8)             self._assert_func(y, x)
566: (8)             assert_raises(AssertionError, lambda: self._assert_func(x, y))
567: (4)             def test_inf_compare(self):
568: (8)                 aone = np.array(1)
569: (8)                 ainf = np.array(np.inf)
570: (8)                 self._assert_func(aone, ainf)
571: (8)                 self._assert_func(-ainf, aone)
572: (8)                 self._assert_func(-ainf, ainf)
573: (8)                 assert_raises(AssertionError, lambda: self._assert_func(ainf, aone))
574: (8)                 assert_raises(AssertionError, lambda: self._assert_func(aone, -ainf))
575: (8)                 assert_raises(AssertionError, lambda: self._assert_func(ainf, ainf))
576: (8)                 assert_raises(AssertionError, lambda: self._assert_func(ainf, -ainf))
577: (8)                 assert_raises(AssertionError, lambda: self._assert_func(-ainf, -ainf))
578: (4)             def test_inf_compare_array(self):
579: (8)                 x = np.array([1.1, 2.2, np.inf])
580: (8)                 ainf = np.array(np.inf)
581: (8)                 assert_raises(AssertionError, lambda: self._assert_func(x, ainf))
582: (8)                 assert_raises(AssertionError, lambda: self._assert_func(ainf, x))
583: (8)                 assert_raises(AssertionError, lambda: self._assert_func(x, -ainf))
584: (8)                 assert_raises(AssertionError, lambda: self._assert_func(-x, -ainf))
585: (8)                 assert_raises(AssertionError, lambda: self._assert_func(-ainf, -x))
586: (8)                 self._assert_func(-ainf, x)
587: (0)             class TestWarns:
588: (4)                 def test_warn(self):
589: (8)                     def f():
590: (12)                         warnings.warn("yo")
591: (12)                         return 3
592: (8)                     before_filters = sys.modules['warnings'].filters[:]
593: (8)                     assert_equal(assert_warns(UserWarning, f), 3)
594: (8)                     after_filters = sys.modules['warnings'].filters
595: (8)                     assert_raises(AssertionError, assert_no_warnings, f)
596: (8)                     assert_equal(assert_no_warnings(lambda x: x, 1), 1)
597: (8)                     assert_equal(before_filters, after_filters,
598: (21)                         "assert_warns does not preserver warnings state")
599: (4)                 def test_context_manager(self):
600: (8)                     before_filters = sys.modules['warnings'].filters[:]
601: (8)                     with assert_warns(UserWarning):
602: (12)                         warnings.warn("yo")
603: (8)                     after_filters = sys.modules['warnings'].filters
604: (8)                     def no_warnings():
605: (12)                         with assert_no_warnings():
606: (16)                             warnings.warn("yo")
607: (8)                         assert_raises(AssertionError, no_warnings)
608: (8)                         assert_equal(before_filters, after_filters,
609: (21)                             "assert_warns does not preserver warnings state")
610: (4)                 def test_warn_wrong_warning(self):
611: (8)                     def f():
612: (12)                         warnings.warn("yo", DeprecationWarning)
613: (8)                     failed = False
614: (8)                     with warnings.catch_warnings():
615: (12)                         warnings.simplefilter("error", DeprecationWarning)
616: (12)                         try:
617: (16)                             assert_warns(UserWarning, f)
618: (16)                             failed = True
619: (12)                         except DeprecationWarning:
620: (16)                             pass
621: (8)                         if failed:
622: (12)                             raise AssertionError("wrong warning caught by assert_warn")
623: (0)             class TestAssertAllclose:
624: (4)                 def test_simple(self):
625: (8)                     x = 1e-3
626: (8)                     y = 1e-9
627: (8)                     assert_allclose(x, y, atol=1)
628: (8)                     assert_raises(AssertionError, assert_allclose, x, y)
629: (8)                     a = np.array([x, y, x, y])

```

```

630: (8)             b = np.array([x, y, x, x])
631: (8)             assert_allclose(a, b, atol=1)
632: (8)             assert_raises(AssertionError, assert_allclose, a, b)
633: (8)             b[-1] = y * (1 + 1e-8)
634: (8)             assert_allclose(a, b)
635: (8)             assert_raises(AssertionError, assert_allclose, a, b, rtol=1e-9)
636: (8)             assert_allclose(6, 10, rtol=0.5)
637: (8)             assert_raises(AssertionError, assert_allclose, 10, 6, rtol=0.5)
638: (4) def test_min_int(self):
639: (8)     a = np.array([np.iinfo(np.int_).min], dtype=np.int_)
640: (8)     assert_allclose(a, a)
641: (4) def test_report_fail_percentage(self):
642: (8)     a = np.array([1, 1, 1, 1])
643: (8)     b = np.array([1, 1, 1, 2])
644: (8)     with pytest.raises(AssertionError) as exc_info:
645: (12)         assert_allclose(a, b)
646: (8)     msg = str(exc_info.value)
647: (8)     assert_('_Mismatched elements: 1 / 4 (25%)\n' +
648: (16)         'Max absolute difference: 1\n' +
649: (16)         'Max relative difference: 0.5' in msg)
650: (4) def test_equal_nan(self):
651: (8)     a = np.array([np.nan])
652: (8)     b = np.array([np.nan])
653: (8)     assert_allclose(a, b, equal_nan=True)
654: (4) def test_not_equal_nan(self):
655: (8)     a = np.array([np.nan])
656: (8)     b = np.array([np.nan])
657: (8)     assert_raises(AssertionError, assert_allclose, a, b, equal_nan=False)
658: (4) def test_equal_nan_default(self):
659: (8)     a = np.array([np.nan])
660: (8)     b = np.array([np.nan])
661: (8)     assert_array_equal(a, b)
662: (8)     assert_array_almost_equal(a, b)
663: (8)     assert_array_less(a, b)
664: (8)     assert_allclose(a, b)
665: (4) def test_report_max_relative_error(self):
666: (8)     a = np.array([0, 1])
667: (8)     b = np.array([0, 2])
668: (8)     with pytest.raises(AssertionError) as exc_info:
669: (12)         assert_allclose(a, b)
670: (8)     msg = str(exc_info.value)
671: (8)     assert_('_Max relative difference: 0.5' in msg)
672: (4) def test_timedelta(self):
673: (8)     a = np.array([[1, 2, 3, "NaT"]], dtype="m8[ns]")
674: (8)     assert_allclose(a, a)
675: (4) def test_error_message_unsigned(self):
676: (8)     """Check the message is formatted correctly when overflow can
occur
677: (11)     (gh21768)"""
678: (8)     x = np.asarray([0, 1, 8], dtype='uint8')
679: (8)     y = np.asarray([4, 4, 4], dtype='uint8')
680: (8)     with pytest.raises(AssertionError) as exc_info:
681: (12)         assert_allclose(x, y, atol=3)
682: (8)     msgs = str(exc_info.value).split('\n')
683: (8)     assert_equal(msgs[4], 'Max absolute difference: 4')
684: (0) class TestArrayAlmostEqualNulp:
685: (4)     def test_float64_pass(self):
686: (8)         nulp = 5
687: (8)         x = np.linspace(-20, 20, 50, dtype=np.float64)
688: (8)         x = 10**x
689: (8)         x = np.r_[-x, x]
690: (8)         eps = np.finfo(x.dtype).eps
691: (8)         y = x + x*eps*nulp/2.
692: (8)         assert_array_almost_nulp(x, y, nulp)
693: (8)         epsneg = np.finfo(x.dtype).epsneg
694: (8)         y = x - x*epsneg*nulp/2.
695: (8)         assert_array_almost_nulp(x, y, nulp)
696: (4)     def test_float64_fail(self):
697: (8)         nulp = 5

```

```

698: (8)           x = np.linspace(-20, 20, 50, dtype=np.float64)
699: (8)           x = 10**x
700: (8)           x = np.r_[ -x, x]
701: (8)           eps = np.finfo(x.dtype).eps
702: (8)           y = x + x*eps*nulp*2.
703: (8)           assert_raises(AssertionError, assert_array_almost_equal_nulp,
704: (22)                 x, y, nulp)
705: (8)           epsneg = np.finfo(x.dtype).epsneg
706: (8)           y = x - x*epsneg*nulp*2.
707: (8)           assert_raises(AssertionError, assert_array_almost_equal_nulp,
708: (22)                 x, y, nulp)
709: (4)           def test_float64_ignore_nan(self):
710: (8)             offset = np.uint64(0xffffffff)
711: (8)             nan1_i64 = np.array(np.nan, dtype=np.float64).view(np.uint64)
712: (8)             nan2_i64 = nan1_i64 ^ offset # nan payload on MIPS is all ones.
713: (8)             nan1_f64 = nan1_i64.view(np.float64)
714: (8)             nan2_f64 = nan2_i64.view(np.float64)
715: (8)             assert_array_max_ulp(nan1_f64, nan2_f64, 0)
716: (4)           def test_float32_pass(self):
717: (8)             nulp = 5
718: (8)             x = np.linspace(-20, 20, 50, dtype=np.float32)
719: (8)             x = 10**x
720: (8)             x = np.r_[ -x, x]
721: (8)             eps = np.finfo(x.dtype).eps
722: (8)             y = x + x*eps*nulp/2.
723: (8)             assert_array_almost_equal_nulp(x, y, nulp)
724: (8)             epsneg = np.finfo(x.dtype).epsneg
725: (8)             y = x - x*epsneg*nulp/2.
726: (8)             assert_array_almost_equal_nulp(x, y, nulp)
727: (4)           def test_float32_fail(self):
728: (8)             nulp = 5
729: (8)             x = np.linspace(-20, 20, 50, dtype=np.float32)
730: (8)             x = 10**x
731: (8)             x = np.r_[ -x, x]
732: (8)             eps = np.finfo(x.dtype).eps
733: (8)             y = x + x*eps*nulp*2.
734: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
735: (22)                   x, y, nulp)
736: (8)             epsneg = np.finfo(x.dtype).epsneg
737: (8)             y = x - x*epsneg*nulp*2.
738: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
739: (22)                   x, y, nulp)
740: (4)           def test_float32_ignore_nan(self):
741: (8)             offset = np.uint32(0xffff)
742: (8)             nan1_i32 = np.array(np.nan, dtype=np.float32).view(np.uint32)
743: (8)             nan2_i32 = nan1_i32 ^ offset # nan payload on MIPS is all ones.
744: (8)             nan1_f32 = nan1_i32.view(np.float32)
745: (8)             nan2_f32 = nan2_i32.view(np.float32)
746: (8)             assert_array_max_ulp(nan1_f32, nan2_f32, 0)
747: (4)           def test_float16_pass(self):
748: (8)             nulp = 5
749: (8)             x = np.linspace(-4, 4, 10, dtype=np.float16)
750: (8)             x = 10**x
751: (8)             x = np.r_[ -x, x]
752: (8)             eps = np.finfo(x.dtype).eps
753: (8)             y = x + x*eps*nulp/2.
754: (8)             assert_array_almost_equal_nulp(x, y, nulp)
755: (8)             epsneg = np.finfo(x.dtype).epsneg
756: (8)             y = x - x*epsneg*nulp/2.
757: (8)             assert_array_almost_equal_nulp(x, y, nulp)
758: (4)           def test_float16_fail(self):
759: (8)             nulp = 5
760: (8)             x = np.linspace(-4, 4, 10, dtype=np.float16)
761: (8)             x = 10**x
762: (8)             x = np.r_[ -x, x]
763: (8)             eps = np.finfo(x.dtype).eps
764: (8)             y = x + x*eps*nulp*2.
765: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
766: (22)                   x, y, nulp)

```

```

767: (8)           epsneg = np.finfo(x.dtype).epsneg
768: (8)           y = x - x*epsneg*nulp*2.
769: (8)           assert_raises(AssertionError, assert_array_almost_equal_nulp,
770: (22)             x, y, nulp)
771: (4)           def test_float16_ignore_nan(self):
772: (8)             offset = np.uint16(0xff)
773: (8)             nan1_i16 = np.array(np.nan, dtype=np.float16).view(np.uint16)
774: (8)             nan2_i16 = nan1_i16 ^ offset # nan payload on MIPS is all ones.
775: (8)             nan1_f16 = nan1_i16.view(np.float16)
776: (8)             nan2_f16 = nan2_i16.view(np.float16)
777: (8)             assert_array_max_ulp(nan1_f16, nan2_f16, 0)
778: (4)           def test_complex128_pass(self):
779: (8)             nulp = 5
780: (8)             x = np.linspace(-20, 20, 50, dtype=np.float64)
781: (8)             x = 10**x
782: (8)             x = np.r_[-x, x]
783: (8)             xi = x + x*1j
784: (8)             eps = np.finfo(x.dtype).eps
785: (8)             y = x + x*eps*nulp/2.
786: (8)             assert_array_almost_equal_nulp(xi, x + y*1j, nulp)
787: (8)             assert_array_almost_equal_nulp(xi, y + x*1j, nulp)
788: (8)             y = x + x*eps*nulp/4.
789: (8)             assert_array_almost_equal_nulp(xi, y + y*1j, nulp)
790: (8)             epsneg = np.finfo(x.dtype).epsneg
791: (8)             y = x - x*epsneg*nulp/2.
792: (8)             assert_array_almost_equal_nulp(xi, x + y*1j, nulp)
793: (8)             assert_array_almost_equal_nulp(xi, y + x*1j, nulp)
794: (8)             y = x - x*epsneg*nulp/4.
795: (8)             assert_array_almost_equal_nulp(xi, y + y*1j, nulp)
796: (4)           def test_complex128_fail(self):
797: (8)             nulp = 5
798: (8)             x = np.linspace(-20, 20, 50, dtype=np.float64)
799: (8)             x = 10**x
800: (8)             x = np.r_[-x, x]
801: (8)             xi = x + x*1j
802: (8)             eps = np.finfo(x.dtype).eps
803: (8)             y = x + x*eps*nulp*2.
804: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
805: (22)               xi, x + y*1j, nulp)
806: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
807: (22)               xi, y + x*1j, nulp)
808: (8)             y = x + x*eps*nulp
809: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
810: (22)               xi, y + y*1j, nulp)
811: (8)             epsneg = np.finfo(x.dtype).epsneg
812: (8)             y = x - x*epsneg*nulp*2.
813: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
814: (22)               xi, x + y*1j, nulp)
815: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
816: (22)               xi, y + x*1j, nulp)
817: (8)             y = x - x*epsneg*nulp
818: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
819: (22)               xi, y + y*1j, nulp)
820: (4)           def test_complex64_pass(self):
821: (8)             nulp = 5
822: (8)             x = np.linspace(-20, 20, 50, dtype=np.float32)
823: (8)             x = 10**x
824: (8)             x = np.r_[-x, x]
825: (8)             xi = x + x*1j
826: (8)             eps = np.finfo(x.dtype).eps
827: (8)             y = x + x*eps*nulp/2.
828: (8)             assert_array_almost_equal_nulp(xi, x + y*1j, nulp)
829: (8)             assert_array_almost_equal_nulp(xi, y + x*1j, nulp)
830: (8)             y = x + x*eps*nulp/4.
831: (8)             assert_array_almost_equal_nulp(xi, y + y*1j, nulp)
832: (8)             epsneg = np.finfo(x.dtype).epsneg
833: (8)             y = x - x*epsneg*nulp/2.
834: (8)             assert_array_almost_equal_nulp(xi, x + y*1j, nulp)
835: (8)             assert_array_almost_equal_nulp(xi, y + x*1j, nulp)

```

```

836: (8)             y = x - x*epsneg*nulp/4.
837: (8)             assert_array_almost_equal_nulp(xi, y + y*1j, nulp)
838: (4)             def test_complex64_fail(self):
839: (8)                 nulp = 5
840: (8)                 x = np.linspace(-20, 20, 50, dtype=np.float32)
841: (8)                 x = 10**x
842: (8)                 x = np.r_[ -x, x]
843: (8)                 xi = x + x*1j
844: (8)                 eps = np.finfo(x.dtype).eps
845: (8)                 y = x + x*eps*nulp*2.
846: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
847: (22)                         xi, x + y*1j, nulp)
848: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
849: (22)                         xi, y + x*1j, nulp)
850: (8)             y = x + x*eps*nulp
851: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
852: (22)                         xi, y + y*1j, nulp)
853: (8)             epsneg = np.finfo(x.dtype).epsneg
854: (8)             y = x - x*epsneg*nulp*2.
855: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
856: (22)                         xi, x + y*1j, nulp)
857: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
858: (22)                         xi, y + x*1j, nulp)
859: (8)             y = x - x*epsneg*nulp
860: (8)             assert_raises(AssertionError, assert_array_almost_equal_nulp,
861: (22)                         xi, y + y*1j, nulp)
862: (0)             class TestULP:
863: (4)                 def test_equal(self):
864: (8)                     x = np.random.randn(10)
865: (8)                     assert_array_max_ulp(x, x, maxulp=0)
866: (4)                 def test_single(self):
867: (8)                     x = np.ones(10).astype(np.float32)
868: (8)                     x += 0.01 * np.random.randn(10).astype(np.float32)
869: (8)                     eps = np.finfo(np.float32).eps
870: (8)                     assert_array_max_ulp(x, x+eps, maxulp=20)
871: (4)                 def test_double(self):
872: (8)                     x = np.ones(10).astype(np.float64)
873: (8)                     x += 0.01 * np.random.randn(10).astype(np.float64)
874: (8)                     eps = np.finfo(np.float64).eps
875: (8)                     assert_array_max_ulp(x, x+eps, maxulp=200)
876: (4)                 def test_inf(self):
877: (8)                     for dt in [np.float32, np.float64]:
878: (12)                         inf = np.array([np.inf]).astype(dt)
879: (12)                         big = np.array([np.finfo(dt).max])
880: (12)                         assert_array_max_ulp(inf, big, maxulp=200)
881: (4)                 def test_nan(self):
882: (8)                     for dt in [np.float32, np.float64]:
883: (12)                         if dt == np.float32:
884: (16)                             maxulp = 1e6
885: (12)                         else:
886: (16)                             maxulp = 1e12
887: (12)                         inf = np.array([np.inf]).astype(dt)
888: (12)                         nan = np.array([np.nan]).astype(dt)
889: (12)                         big = np.array([np.finfo(dt).max])
890: (12)                         tiny = np.array([np.finfo(dt).tiny])
891: (12)                         zero = np.array([np.PZERO]).astype(dt)
892: (12)                         nzero = np.array([np.NZERO]).astype(dt)
893: (12)                         assert_raises(AssertionError,
894: (26)                             lambda: assert_array_max_ulp(nan, inf,
895: (26)                                 maxulp=maxulp))
896: (12)                         assert_raises(AssertionError,
897: (26)                             lambda: assert_array_max_ulp(nan, big,
898: (26)                                 maxulp=maxulp))
899: (12)                         assert_raises(AssertionError,
900: (26)                             lambda: assert_array_max_ulp(nan, tiny,
901: (26)                                 maxulp=maxulp))
902: (12)                         assert_raises(AssertionError,
903: (26)                             lambda: assert_array_max_ulp(nan, zero,
904: (26)                                 maxulp=maxulp))

```

```

905: (12) assert_raises(AssertError,
906: (26)         lambda: assert_array_max_ulp(nan, nzero,
907: (26)             maxulp=maxulp))
908: (0) class TestStringEqual:
909: (4)     def test_simple(self):
910: (8)         assert_string_equal("hello", "hello")
911: (8)         assert_string_equal("hello\nmultiline", "hello\nmultiline")
912: (8)         with pytest.raises(AssertError) as exc_info:
913: (12)             assert_string_equal("foo\nbar", "hello\nbar")
914: (8)             msg = str(exc_info.value)
915: (8)             assert_equal(msg, "Differences in strings:\n- foo\n+ hello")
916: (8)             assert_raises(AssertError,
917: (22)                 lambda: assert_string_equal("foo", "hello"))
918: (4)     def test_regex(self):
919: (8)         assert_string_equal("a+*b", "a+*b")
920: (8)         assert_raises(AssertError,
921: (22)             lambda: assert_string_equal("aaa", "a+b"))
922: (0) def assert_warn_len_equal(mod, n_in_context):
923: (4)     try:
924: (8)         mod.warns = mod.__warningregistry__
925: (4)     except AttributeError:
926: (8)         mod.warns = {}
927: (4)     num_warns = len(mod.warns)
928: (4)     if 'version' in mod.warns:
929: (8)         num_warns -= 1
930: (4)     assert_equal(num_warns, n_in_context)
931: (0) def test_warn_len_equal_call_scenarios():
932: (4)     class mod:
933: (8)         pass
934: (4)         mod_inst = mod()
935: (4)         assert_warn_len_equal(mod=mod_inst,
936: (26)             n_in_context=0)
937: (4)     class mod:
938: (8)         def __init__(self):
939: (12)             self.__warningregistry__ = {'warning1':1,
940: (40)                             'warning2':2}
941: (4)         mod_inst = mod()
942: (4)         assert_warn_len_equal(mod=mod_inst,
943: (26)             n_in_context=2)
944: (0) def _get_fresh_mod():
945: (4)     my_mod = sys.modules[__name__]
946: (4)     try:
947: (8)         my_mod.__warningregistry__.clear()
948: (4)     except AttributeError:
949: (8)         pass
950: (4)     return my_mod
951: (0) def test_clear_and_catch_warnings():
952: (4)     my_mod = _get_fresh_mod()
953: (4)     assert_equal(getattr(my_mod, '__warningregistry__', {}), {})
954: (4)     with clear_and_catch_warnings(modules=[my_mod]):
955: (8)         warnings.simplefilter('ignore')
956: (8)         warnings.warn('Some warning')
957: (4)         assert_equal(my_mod.__warningregistry__, {})
958: (4)         with clear_and_catch_warnings():
959: (8)             warnings.simplefilter('ignore')
960: (8)             warnings.warn('Some warning')
961: (4)             assert_warn_len_equal(my_mod, 0)
962: (4)             my_mod.__warningregistry__ = {'warning1': 1,
963: (34)                             'warning2': 2}
964: (4)             with clear_and_catch_warnings(modules=[my_mod]):
965: (8)                 warnings.simplefilter('ignore')
966: (8)                 warnings.warn('Another warning')
967: (4)                 assert_warn_len_equal(my_mod, 2)
968: (4)                 with clear_and_catch_warnings():
969: (8)                     warnings.simplefilter('ignore')
970: (8)                     warnings.warn('Another warning')
971: (4)                     assert_warn_len_equal(my_mod, 0)
972: (0) def test_suppress_warnings_module():
973: (4)     my_mod = _get_fresh_mod()

```

```

974: (4) assert_equal(getattr(my_mod, '__warningregistry__', {}), {})
975: (4) def warn_other_module():
976: (8)     def warn(arr):
977: (12)         warnings.warn("Some warning 2", stacklevel=2)
978: (12)         return arr
979: (8)     np.apply_along_axis(warn, 0, [0])
980: (4) assert_warn_len_equal(my_mod, 0)
981: (4) with suppress_warnings() as sup:
982: (8)     sup.record(UserWarning)
983: (8)     sup.filter(module=np.lib.shape_base)
984: (8)     warnings.warn("Some warning")
985: (8)     warn_other_module()
986: (4)     assert_equal(len(sup.log), 1)
987: (4)     assert_equal(sup.log[0].message.args[0], "Some warning")
988: (4)     assert_warn_len_equal(my_mod, 0)
989: (4)     sup = suppress_warnings()
990: (4)     sup.filter(module=my_mod)
991: (4)     with sup:
992: (8)         warnings.warn('Some warning')
993: (4)     assert_warn_len_equal(my_mod, 0)
994: (4)     sup.filter(module=my_mod)
995: (4)     with sup:
996: (8)         warnings.warn('Some warning')
997: (4)     assert_warn_len_equal(my_mod, 0)
998: (4)     with suppress_warnings():
999: (8)         warnings.simplefilter('ignore')
1000: (8)         warnings.warn('Some warning')
1001: (4)     assert_warn_len_equal(my_mod, 0)
1002: (0) def test_suppress_warnings_type():
1003: (4)     my_mod = _get_fresh_mod()
1004: (4)     assert_equal(getattr(my_mod, '__warningregistry__', {}), {})
1005: (4)     with suppress_warnings() as sup:
1006: (8)         sup.filter(UserWarning)
1007: (8)         warnings.warn('Some warning')
1008: (4)     assert_warn_len_equal(my_mod, 0)
1009: (4)     sup = suppress_warnings()
1010: (4)     sup.filter(UserWarning)
1011: (4)     with sup:
1012: (8)         warnings.warn('Some warning')
1013: (4)     assert_warn_len_equal(my_mod, 0)
1014: (4)     sup.filter(module=my_mod)
1015: (4)     with sup:
1016: (8)         warnings.warn('Some warning')
1017: (4)     assert_warn_len_equal(my_mod, 0)
1018: (4)     with suppress_warnings():
1019: (8)         warnings.simplefilter('ignore')
1020: (8)         warnings.warn('Some warning')
1021: (4)     assert_warn_len_equal(my_mod, 0)
1022: (0) def test_suppress_warnings_decorate_no_record():
1023: (4)     sup = suppress_warnings()
1024: (4)     sup.filter(UserWarning)
1025: (4)     @sup
1026: (4)     def warn(category):
1027: (8)         warnings.warn('Some warning', category)
1028: (4)     with warnings.catch_warnings(record=True) as w:
1029: (8)         warnings.simplefilter("always")
1030: (8)         warn(UserWarning) # should be suppressed
1031: (8)         warn(RuntimeWarning)
1032: (8)         assert_equal(len(w), 1)
1033: (0) def test_suppress_warnings_record():
1034: (4)     sup = suppress_warnings()
1035: (4)     log1 = sup.record()
1036: (4)     with sup:
1037: (8)         log2 = sup.record(message='Some other warning 2')
1038: (8)         sup.filter(message='Some warning')
1039: (8)         warnings.warn('Some warning')
1040: (8)         warnings.warn('Some other warning')
1041: (8)         warnings.warn('Some other warning 2')
1042: (8)         assert_equal(len(sup.log), 2)

```

```

1043: (8)             assert_equal(len(log1), 1)
1044: (8)             assert_equal(len(log2),1)
1045: (8)             assert_equal(log2[0].message.args[0], 'Some other warning 2')
1046: (4)             with sup:
1047: (8)                 log2 = sup.record(message='Some other warning 2')
1048: (8)                 sup.filter(message='Some warning')
1049: (8)                 warnings.warn('Some warning')
1050: (8)                 warnings.warn('Some other warning')
1051: (8)                 warnings.warn('Some other warning 2')
1052: (8)                 assert_equal(len(sup.log), 2)
1053: (8)                 assert_equal(len(log1), 1)
1054: (8)                 assert_equal(len(log2), 1)
1055: (8)                 assert_equal(log2[0].message.args[0], 'Some other warning 2')
1056: (4)             with suppress_warnings() as sup:
1057: (8)                 sup.record()
1058: (8)                 with suppress_warnings() as sup2:
1059: (12)                     sup2.record(message='Some warning')
1060: (12)                     warnings.warn('Some warning')
1061: (12)                     warnings.warn('Some other warning')
1062: (12)                     assert_equal(len(sup2.log), 1)
1063: (8)                     assert_equal(len(sup.log), 1)
1064: (0)             def test_suppress_warnings_forwarding():
1065: (4)                 def warn_other_module():
1066: (8)                     def warn(arr):
1067: (12)                         warnings.warn("Some warning", stacklevel=2)
1068: (12)                         return arr
1069: (8)                         np.apply_along_axis(warn, 0, [0])
1070: (4)             with suppress_warnings() as sup:
1071: (8)                 sup.record()
1072: (8)                 with suppress_warnings("always"):
1073: (12)                     for i in range(2):
1074: (16)                         warnings.warn("Some warning")
1075: (8)                     assert_equal(len(sup.log), 2)
1076: (4)             with suppress_warnings() as sup:
1077: (8)                 sup.record()
1078: (8)                 with suppress_warnings("location"):
1079: (12)                     for i in range(2):
1080: (16)                         warnings.warn("Some warning")
1081: (16)                         warnings.warn("Some warning")
1082: (8)                     assert_equal(len(sup.log), 2)
1083: (4)             with suppress_warnings() as sup:
1084: (8)                 sup.record()
1085: (8)                 with suppress_warnings("module"):
1086: (12)                     for i in range(2):
1087: (16)                         warnings.warn("Some warning")
1088: (16)                         warnings.warn("Some warning")
1089: (16)                         warn_other_module()
1090: (8)                     assert_equal(len(sup.log), 2)
1091: (4)             with suppress_warnings() as sup:
1092: (8)                 sup.record()
1093: (8)                 with suppress_warnings("once"):
1094: (12)                     for i in range(2):
1095: (16)                         warnings.warn("Some warning")
1096: (16)                         warnings.warn("Some other warning")
1097: (16)                         warn_other_module()
1098: (8)                     assert_equal(len(sup.log), 2)
1099: (0)             def test_tempdir():
1100: (4)                 with tempdir() as tdir:
1101: (8)                     fpath = os.path.join(tdir, 'tmp')
1102: (8)                     with open(fpath, 'w'):
1103: (12)                         pass
1104: (4)                     assert_(not os.path.isdir(tdir))
1105: (4)                     raised = False
1106: (4)                     try:
1107: (8)                         with tempdir() as tdir:
1108: (12)                             raise ValueError()
1109: (4)                     except ValueError:
1110: (8)                         raised = True
1111: (4)                     assert_(raised)

```

```

1112: (4)             assert_(not os.path.isdir(tdir))
1113: (0)
1114: (4)
1115: (8)
1116: (12)
1117: (4)
1118: (4)
1119: (4)
1120: (8)
1121: (12)
1122: (4)
1123: (8)
1124: (4)
1125: (4)
1126: (0)
1127: (4)
1128: (0)
1129: (4)
1130: (4)
1131: (8)
1132: (8)
1133: (4)
1134: (0)
1135: (0)
1136: (4)             """ Test assert_no_gc_cycles """
1137: (4)
1138: (8)
1139: (12)
1140: (12)
1141: (12)
1142: (8)
1143: (12)
1144: (8)
1145: (4)
1146: (8)
1147: (12)
1148: (12)
1149: (12)
1150: (12)
1151: (8)
1152: (12)
1153: (16)
1154: (8)
1155: (12)
1156: (4)
1157: (4)
1158: (8)
1159: (8)
1160: (8)
1161: (8)
1162: (8)
1163: (12)
1164: (12)
new
1165: (12)             An object that not only contains a reference cycle, but creates
1166: (12)             cycles whenever it's garbage-collected and its __del__ runs
1167: (12)             """
1168: (12)             make_cycle = True
1169: (16)             def __init__(self):
1170: (12)                 self.cycle = self
1171: (16)             def __del__(self):
1172: (16)                 self.cycle = None
1173: (20)                 if ReferenceCycleInDel.make_cycle:
1174: (8)                     ReferenceCycleInDel()
try:
1175: (12)             w = weakref.ref(ReferenceCycleInDel())
1176: (12)             try:
1177: (16)                 with assert_raises(RuntimeError):
1178: (20)                     assert_no_gc_cycles(lambda: None)
1179: (12)             except AssertionError:

```

```

1180: (16)             if w() is not None:
1181: (20)                 pytest.skip("GC does not call __del__ on cyclic objects")
1182: (20)                     raise
1183: (8)             finally:
1184: (12)                 ReferenceCycleInDel.make_cycle = False
-----
```

File 331 - `__init__.py`:

```
1: (0)
```

File 332 - `extbuild.py`:

```

1: (0) """
2: (0)     Build a c-extension module on-the-fly in tests.
3: (0)     See build_and_import_extensions for usage hints
4: (0) """
5: (0)     import os
6: (0)     import pathlib
7: (0)     import subprocess
8: (0)     import sys
9: (0)     import sysconfig
10: (0)    import textwrap
11: (0)    __all__ = ['build_and_import_extension', 'compile_extension_module']
12: (0) def build_and_import_extension(
13: (8)     modname, functions, *, prologue="", build_dir=None,
14: (8)         include_dirs=[], more_init=""):
15: (4) """
16: (4)     Build and imports a c-extension module `modname` from a list of function
17: (4)     fragments `functions`.
18: (4)     Parameters
19: (4)     -----
20: (4)     functions : list of fragments
21: (8)         Each fragment is a sequence of func_name, calling convention, snippet.
22: (4)     prologue : string
23: (8)         Code to precede the rest, usually extra ``#include`` or ``#define``
24: (8)         macros.
25: (4)     build_dir : pathlib.Path
26: (8)         Where to build the module, usually a temporary directory
27: (4)     include_dirs : list
28: (8)         Extra directories to find include files when compiling
29: (4)     more_init : string
30: (8)         Code to appear in the module PyMODINIT_FUNC
31: (4)     Returns
32: (4)     -----
33: (4)     out: module
34: (8)         The module will have been loaded and is ready for use
35: (4)     Examples
36: (4)     -----
37: (4)     >>> functions = [("test_bytes", "METH_O", \\\"\\\"\\"
38: (8)         if ( !PyBytesCheck(args)) {
39: (12)             Py_RETURN_FALSE;
40: (8)         }
41: (8)             Py_RETURN_TRUE;
42: (4)         \\\"\\\"\\")]
43: (4)     >>> mod = build_and_import_extension("testme", functions)
44: (4)     >>> assert not mod.test_bytes(u'abc')
45: (4)     >>> assert mod.test_bytes(b'abc')
46: (4) """
47: (4)     body = prologue + _make_methods(functions, modname)
48: (4)     init = """PyObject *mod = PyModule_Create(&moduledesc);
49: (11)     """
50: (4)     if not build_dir:
51: (8)         build_dir = pathlib.Path('.')
52: (4)     if more_init:
53: (8)         init += """#define INITERROR return NULL
```

```

54: (16)
55: (8)         """
56: (4)         init += more_init
57: (4)         init += "\nreturn mod;"
58: (4)         source_string = _make_source(modname, init, body)
59: (4)         try:
60: (8)             mod_so = compile_extension_module(
61: (8)                 modname, build_dir, include_dirs, source_string)
62: (4)         except Exception as e:
63: (8)             raise RuntimeError(f"could not compile in {build_dir}:") from e
64: (4)         import importlib.util
65: (4)         spec = importlib.util.spec_from_file_location(modname, mod_so)
66: (4)         foo = importlib.util.module_from_spec(spec)
67: (4)         spec.loader.exec_module(foo)
68: (4)         return foo
69: (0)     def compile_extension_module(
70: (8)         name, builddir, include_dirs,
71: (8)         source_string, libraries=[], library_dirs=[]):
72: (4)         """
73: (4)             Build an extension module and return the filename of the resulting
74: (4)             native code file.
75: (4)             Parameters
76: (4)             -----
77: (4)             name : string
78: (8)                 name of the module, possibly including dots if it is a module inside a
79: (8)                 package.
80: (4)             builddir : pathlib.Path
81: (8)                 Where to build the module, usually a temporary directory
82: (4)             include_dirs : list
83: (8)                 Extra directories to find include files when compiling
84: (4)             libraries : list
85: (8)                 Libraries to link into the extension module
86: (4)             library_dirs: list
87: (8)                 Where to find the libraries, ``-L`` passed to the linker
88: (4)             """
89: (4)             modname = name.split('.')[-1]
90: (4)             dirname = builddir / name
91: (4)             dirname.mkdir(exist_ok=True)
92: (4)             cfile = _convert_str_to_file(source_string, dirname)
93: (4)             include_dirs = include_dirs + [sysconfig.get_config_var('INCLUDEPY')]
94: (4)             return _c_compile(
95: (8)                 cfile, outputfilename=dirname / modname,
96: (8)                 include_dirs=include_dirs, libraries=[], library_dirs=[],
97: (4)             )
98: (0)     def _convert_str_to_file(source, dirname):
99: (4)         """
100: (4)             Helper function to create a file ``source.c`` in `dirname` that
101: (4)             contains
102: (4)             the string in `source`. Returns the file name
103: (4)             """
104: (8)             filename = dirname / 'source.c'
105: (4)             with filename.open('w') as f:
106: (8)                 f.write(str(source))
107: (4)             return filename
108: (0)     def _make_methods(functions, modname):
109: (4)         """
110: (4)             Turns the name, signature, code in functions into complete functions
111: (4)             and lists them in a methods_table. Then turns the methods_table into a
112: (4)             ``PyMethodDef`` structure and returns the resulting code fragment ready
113: (4)             for compilation
114: (4)             """
115: (8)             methods_table = []
116: (8)             codes = []
117: (8)             for funcname, flags, code in functions:
118: (12)                 cfuncname = "%s_%s" % (modname, funcname)
119: (8)                 if 'METH_KEYWORDS' in flags:
120: (12)                     signature = '(PyObject *self, PyObject *args, PyObject *kwargs)'
121: (8)                 else:
122: (12)                     signature = '(PyObject *self, PyObject *args)'
123: (8)                 methods_table.append(
124: (8)                     "{\"%s\", (PyCFunction)%s, %s},\n" % (funcname, cfuncname, flags))
125: (4)             func_code = """

```

```

122: (8)             static PyObject* {cfuncname}{signature}
123: (8)             {{
124: (8)             {code}
125: (8)             }}
126: (8)             """ .format(cfuncname=cfuncname, signature=signature, code=code)
127: (8)             codes.append(func_code)
128: (4)             body = "\n".join(codes) + """
129: (4)             static PyMethodDef methods[] = {
130: (4)             %(methods)s
131: (4)             { NULL }
132: (4)             };
133: (4)             static struct PyModuleDef moduledef = {
134: (8)             PyModuleDef_HEAD_INIT,
135: (8)             "%(modname)s", /* m_name */
136: (8)             NULL,           /* m_doc */
137: (8)             -1,            /* m_size */
138: (8)             methods,        /* m_methods */
139: (4)             };
140: (4)             """ % dict(methods='`n'.join(methods_table), modname=modname)
141: (4)             return body
142: (0)             def _make_source(name, init, body):
143: (4)                 """ Combines the code fragments into source code ready to be compiled
144: (4)                 """
145: (4)                 code = """
146: (4)                 %(body)s
147: (4)                 PyMODINIT_FUNC
148: (4)                 PyInit_%(name)s(void) {
149: (4)                 %(init)s
150: (4)                 }
151: (4)                 """ % dict(
152: (8)                 name=name, init=init, body=body,
153: (4)                 )
154: (4)                 return code
155: (0)             def _c_compile(cfile, outputfilename, include_dirs=[], libraries=[],
156: (15)                         library_dirs=[]):
157: (4)                 if sys.platform == 'win32':
158: (8)                     compile_extra = ["/we4013"]
159: (8)                     link_extra = ["/LIBPATH:" + os.path.join(sys.base_prefix, 'libs')]
160: (4)                 elif sys.platform.startswith('linux'):
161: (8)                     compile_extra = [
162: (12)                     "-O0", "-g", "-Werror=implicit-function-declaration", "-fPIC"]
163: (8)                     link_extra = []
164: (4)                 else:
165: (8)                     compile_extra = link_extra = []
166: (8)                     pass
167: (4)                 if sys.platform == 'win32':
168: (8)                     link_extra = link_extra + ['/DEBUG'] # generate .pdb file
169: (4)                 if sys.platform == 'darwin':
170: (8)                     for s in ('/sw/', '/opt/local/'):
171: (12)                     if (s + 'include' not in include_dirs
172: (20)                         and os.path.exists(s + 'include')):
173: (16)                         include_dirs.append(s + 'include')
174: (12)                     if s + 'lib' not in library_dirs and os.path.exists(s + 'lib'):
175: (16)                         library_dirs.append(s + 'lib')
176: (4)                 outputfilename = outputfilename.with_suffix(get_so_suffix())
177: (4)                 build(
178: (8)                     cfile, outputfilename,
179: (8)                     compile_extra, link_extra,
180: (8)                     include_dirs, libraries, library_dirs)
181: (4)                 return outputfilename
182: (0)             def build(cfile, outputfilename, compile_extra, link_extra,
183: (10)                 include_dirs, libraries, library_dirs):
184: (4)                 "use meson to build"
185: (4)                 build_dir = cfile.parent / "build"
186: (4)                 os.makedirs(build_dir, exist_ok=True)
187: (4)                 so_name = outputfilename.parts[-1]
188: (4)                 with open(cfile.parent / "meson.build", "wt") as fid:
189: (8)                     includes = ['-I' + d for d in include_dirs]
190: (8)                     link_dirs = ['-L' + d for d in library_dirs]

```

```

191: (8)             fid.write(textwrap.dedent(f"""\
192: (12)             project('foo', 'c')
193: (12)             shared_module('{so_name}', '{cfile.parts[-1]}'),
194: (16)                 c_args: {includes} + {compile_extra},
195: (16)                 link_args: {link_dirs} + {link_extra},
196: (16)                 link_with: {libraries},
197: (16)                 name_prefix: '',
198: (16)                 name_suffix: 'dummy',
199: (12)
200: (8)             )
201: (4)             if sys.platform == "win32":
202: (8)                 subprocess.check_call(["meson", "setup",
203: (31)                     "--buildtype=release",
204: (31)                     "--vsevn", ".."],
205: (30)                     cwd=build_dir,
206: (30)             )
207: (4)             else:
208: (8)                 subprocess.check_call(["meson", "setup", "--vsevn", ".."],
209: (30)                     cwd=build_dir
210: (30)             )
211: (4)             subprocess.check_call(["meson", "compile"], cwd=build_dir)
212: (4)             os.rename(str(build_dir / so_name) + ".dummy", cfile.parent / so_name)
213: (0)         def get_so_suffix():
214: (4)             ret = sysconfig.get_config_var('EXT_SUFFIX')
215: (4)             assert ret
216: (4)             return ret
-----
```

## File 333 - utils.py:

```

1: (0)             """
2: (0)             Utility function to facilitate testing.
3: (0)             """
4: (0)             import os
5: (0)             import sys
6: (0)             import platform
7: (0)             import re
8: (0)             import gc
9: (0)             import operator
10: (0)            import warnings
11: (0)            from functools import partial, wraps
12: (0)            import shutil
13: (0)            import contextlib
14: (0)            from tempfile import mkdtemp, mkstemp
15: (0)            from unittest.case import SkipTest
16: (0)            from warnings import WarningMessage
17: (0)            import pprint
18: (0)            import sysconfig
19: (0)            import numpy as np
20: (0)            from numpy.core import (
21: (5)                intp, float32, empty, arange, array_repr, ndarray, isnat, array)
22: (0)            from numpy import isnan, isinf, isinfinite
23: (0)            import numpy.linalg._umath_linalg
24: (0)            from io import StringIO
25: (0)            __all__ = [
26: (8)                'assert_equal', 'assert_almost_equal', 'assert_approx_equal',
27: (8)                'assert_array_equal', 'assert_array_less', 'assert_string_equal',
28: (8)                'assert_array_almost_equal', 'assert_raises', 'build_err_msg',
29: (8)                'decorate_methods', 'jiffies', 'memusage', 'print_assert_equal',
30: (8)                'rundocs', 'runstring', 'verbose', 'measure',
31: (8)                'assert_', 'assert_array_almost_equal_nulp', 'assert_raises_regex',
32: (8)                'assert_array_max_ulp', 'assert_warns', 'assert_no_warnings',
33: (8)                'assert_allclose', 'IgnoreException', 'clear_and_catch_warnings',
34: (8)                'SkipTest', 'KnownFailureException', 'temppath', 'tempdir', 'IS_PYPY',
35: (8)                'HAS_REFCOUNT', 'IS_WASM', 'suppress_warnings',
'assert_array_compare',
36: (8)                'assert_no_gc_cycles', 'break_cycles', 'HAS_LAPACK64', 'IS_PYTHON',
37: (8)                '_OLD_PROMOTION', 'IS_MUSL', '_SUPPORTS_SVE'
```

```

38: (8) ]
39: (0) class KnownFailureException(Exception):
40: (4)     '''Raise this exception to mark a test as a known failing test.'''
41: (4)     pass
42: (0) KnownFailureTest = KnownFailureException # backwards compat
43: (0) verbose = 0
44: (0) IS_WASM = platform.machine() in ["wasm32", "wasm64"]
45: (0) IS_PYPY = sys.implementation.name == 'pypy'
46: (0) IS_PYTHON = hasattr(sys, "pyston_version_info")
47: (0) HAS_REFCOUNT = getattr(sys, 'getrefcount', None) is not None and not IS_PYTHON
48: (0) HAS_LAPACK64 = numpy.linalg._umath_linalg._ilp64
49: (0) _OLD_PROMOTION = lambda: np._get_promotion_state() == 'legacy'
50: (0) IS_MUSL = False
51: (0) _v = sysconfig.get_config_var('HOST_GNU_TYPE') or ''
52: (0) if 'musl' in _v:
53: (4)     IS_MUSL = True
54: (0) def assert_(val, msg=''):
55: (4) """
56: (4)     Assert that works in release mode.
57: (4)     Accepts callable msg to allow deferring evaluation until failure.
58: (4)     The Python built-in ``assert`` does not work when executing code in
59: (4)     optimized mode (the ``-O`` flag) - no byte-code is generated for it.
60: (4)     For documentation on usage, refer to the Python documentation.
61: (4) """
62: (4)     __tracebackhide__ = True # Hide traceback for py.test
63: (4) if not val:
64: (8)     try:
65: (12)         smsg = msg()
66: (8)     except TypeError:
67: (12)         smsg = msg
68: (8)     raise AssertionError(smsg)
69: (0) if os.name == 'nt':
70: (4)     def GetPerformanceAttributes(object, counter, instance=None,
71: (33)                     inum=-1, format=None, machine=None):
72: (8)         import win32pdh
73: (8)         if format is None:
74: (12)             format = win32pdh.PDH_FMT_LONG
75: (8)         path = win32pdh.MakeCounterPath( (machine, object, instance, None,
76: (42)                         inum, counter))
77: (8)         hq = win32pdh.OpenQuery()
78: (8)         try:
79: (12)             hc = win32pdh.AddCounter(hq, path)
80: (12)             try:
81: (16)                 win32pdh.CollectQueryData(hq)
82: (16)                 type, val = win32pdh.GetFormattedCounterValue(hc, format)
83: (16)                 return val
84: (12)             finally:
85: (16)                 win32pdh.RemoveCounter(hc)
86: (8)             finally:
87: (12)                 win32pdh.CloseQuery(hq)
88: (4)     def memusage(processName="python", instance=0):
89: (8)         import win32pdh
90: (8)         return GetPerformanceAttributes("Process", "Virtual Bytes",
91: (40)                         processName, instance,
92: (40)                         win32pdh.PDH_FMT_LONG, None)
93: (0) elif sys.platform[:5] == 'linux':
94: (4)     def memusage(_proc_pid_stat=f'/proc/{os.getpid()}/stat'):
95: (8) """
96: (8)     Return virtual memory size in bytes of the running python.
97: (8) """
98: (8)     try:
99: (12)         with open(_proc_pid_stat) as f:
100: (16)             l = f.readline().split(' ')
101: (12)             return int(l[22])
102: (8)         except Exception:
103: (12)             return
104: (0)     else:
105: (4)         def memusage():
106: (8) """

```

```

107: (8)                                Return memory usage of running python. [Not implemented]
108: (8)
109: (8)                                raise NotImplementedError
110: (0)                                if sys.platform[:5] == 'linux':
111: (4)                                def jiffies(_proc_pid_stat=f'/proc/{os.getpid()}/stat', _load_time=[]):
112: (8)                                """
113: (8)                                Return number of jiffies elapsed.
114: (8)                                Return number of jiffies (1/100ths of a second) that this
115: (8)                                process has been scheduled in user mode. See man 5 proc.
116: (8)
117: (8)
118: (8)
119: (12)                               import time
120: (8)                                if not _load_time:
121: (12)                                _load_time.append(time.time())
122: (16)                               try:
123: (12)                                with open(_proc_pid_stat) as f:
124: (8)                                    l = f.readline().split(' ')
125: (12)                                    return int(l[13])
126: (0)                                except Exception:
127: (4)                                    return int(100*(time.time()-_load_time[0]))
128: (8)
129: (8)                                else:
130: (4)                                def jiffies(_load_time=[]):
131: (8)                                """
132: (8)                                Return number of jiffies elapsed.
133: (8)                                Return number of jiffies (1/100ths of a second) that this
134: (8)                                process has been scheduled in user mode. See man 5 proc.
135: (8)
136: (8)                                import time
137: (0)                                if not _load_time:
138: (18)                                _load_time.append(time.time())
139: (4)                                return int(100*(time.time()-_load_time[0]))
140: (4)                                def build_err_msg(arrays, err_msg, header='Items are not equal:',
141: (8)                                    verbose=True, names=('ACTUAL', 'DESIRED'), precision=8):
142: (12)                                msg = ['\n' + header]
143: (8)                                if err_msg:
144: (12)                                    if err_msg.find('\n') == -1 and len(err_msg) < 79-len(header):
145: (4)                                        msg = [msg[0] + ' ' + err_msg]
146: (8)                                    else:
147: (12)                                        msg.append(err_msg)
148: (16)                                if verbose:
149: (12)                                    for i, a in enumerate(arrays):
150: (16)                                        if isinstance(a, ndarray):
151: (12)                                            r_func = partial(array_repr, precision=precision)
152: (16)                                        else:
153: (12)                                            r_func = repr
154: (16)                                        try:
155: (12)                                            r = r_func(a)
156: (16)                                        except Exception as exc:
157: (16)                                            r = f'[repr failed for <{type(a).__name__}>: {exc}]'
158: (12)                                        if r.count('\n') > 3:
159: (4)                                            r = '\n'.join(r.splitlines()[:3])
160: (0)                                            r += '...'
161: (4)                                            msg.append(f' {names[i]}: {r}')
162: (4)                                return '\n'.join(msg)
163: (4)                                def assert_equal(actual, desired, err_msg='', verbose=True):
164: (8)                                """
165: (4)                                Raises an AssertionError if two objects are not equal.
166: (4)                                Given two objects (scalars, lists, tuples, dictionaries or numpy arrays),
167: (4)                                check that all elements of these objects are equal. An exception is raised
168: (4)                                at the first conflicting values.
169: (4)                                When one of `actual` and `desired` is a scalar and the other is
array_like,
170: (4)                                the function checks that each element of the array_like object is equal to
171: (4)                                the scalar.
172: (4)                                This function handles NaN comparisons as if NaN was a "normal" number.
173: (4)                                That is, AssertionError is not raised if both objects have NaNs in the
same
174: (4)                                positions. This is in contrast to the IEEE standard on NaNs, which says
175: (4)                                that NaN compared to anything must return False.
176: (4)                                Parameters

```

```
174: (4)
175: (4)
176: (8)
177: (4)
178: (8)
179: (4)
180: (8)
181: (4)
182: (8)
183: (4)
184: (4)
185: (4)
186: (8)
187: (4)
188: (4)
189: (4)
190: (4)
191: (8)
192: (4)
193: (4)
194: (4)
195: (5)
196: (5)
197: (4)
198: (4)
199: (4)
200: (4)
201: (4)
202: (4)
203: (8)
204: (12)
205: (8)
206: (8)
207: (12)
208: (16)
209: (12)
210: (25)
211: (8)
212: (4)
213: (8)
214: (8)
215: (12)
216: (25)
217: (8)
218: (4)
219: (4)
220: (4)
221: (8)
222: (4)
223: (4)
224: (8)
225: (4)
226: (8)
227: (4)
228: (8)
229: (12)
230: (12)
231: (8)
232: (12)
233: (12)
234: (8)
235: (12)
236: (12)
237: (8)
238: (12)
239: (12)
240: (8)
241: (12)

    -----
    actual : array_like
        The object to check.
    desired : array_like
        The expected object.
    err_msg : str, optional
        The error message to be printed in case of failure.
    verbose : bool, optional
        If True, the conflicting values are appended to the error message.
Raises
-----
AssertionError
    If actual and desired are not equal.
Examples
-----
>>> np.testing.assert_equal([4,5], [4,6])
Traceback (most recent call last):
...
AssertionError:
Items are not equal:
item=1
    ACTUAL: 5
    DESIRED: 6
The following comparison does not raise an exception. There are NaNs
in the inputs, but they are in the same positions.
>>> np.testing.assert_equal(np.array([1.0, 2.0, np.nan]), [1, 2, np.nan])
"""
    _tracebackhide__ = True # Hide traceback for py.test
if isinstance(desired, dict):
    if not isinstance(actual, dict):
        raise AssertionError(repr(type(actual)))
    assert_equal(len(actual), len(desired), err_msg, verbose)
    for k, i in desired.items():
        if k not in actual:
            raise AssertionError(repr(k))
        assert_equal(actual[k], desired[k], f'key={k}\n{err_msg}', verbose)
    return
if isinstance(desired, (list, tuple)) and isinstance(actual, (list,
tuple)):
    assert_equal(len(actual), len(desired), err_msg, verbose)
    for k in range(len(desired)):
        assert_equal(actual[k], desired[k], f'item={k}\n{err_msg}', verbose)
    return
from numpy.core import ndarray, isscalar, signbit
from numpy.lib import iscomplexobj, real, imag
if isinstance(actual, ndarray) or isinstance(desired, ndarray):
    return assert_array_equal(actual, desired, err_msg, verbose)
msg = build_err_msg([actual, desired], err_msg, verbose=verbose)
try:
    usecomplex = iscomplexobj(actual) or iscomplexobj(desired)
except (ValueError, TypeError):
    usecomplex = False
if usecomplex:
    if iscomplexobj(actual):
        actualr = real(actual)
        actuali = imag(actual)
    else:
        actualr = actual
        actuali = 0
    if iscomplexobj(desired):
        desiredr = real(desired)
        desiredi = imag(desired)
    else:
        desiredr = desired
        desiredi = 0
    try:
        assert_equal(actualr, desiredr)
```

```

242: (12)                     assert_equal(actuali, desiredi)
243: (8)                      except AssertionError:
244: (12)                        raise AssertionError(msg)
245: (4)  if isscalar(desired) != isscalar(actual):
246: (8)    raise AssertionError(msg)
247: (4)  try:
248: (8)    isdesnat = isnat(desired)
249: (8)    isactnat = isnat(actual)
250: (8)    dtypes_match = (np.asarray(desired).dtype.type ==
251: (24)                  np.asarray(actual).dtype.type)
252: (8)    if isdesnat and isactnat:
253: (12)      if dtypes_match:
254: (16)        return
255: (12)      else:
256: (16)        raise AssertionError(msg)
257: (4)  except (TypeError, ValueError, NotImplementedError):
258: (8)    pass
259: (4)  try:
260: (8)    isdesnan = isnan(desired)
261: (8)    isactnan = isnan(actual)
262: (8)    if isdesnan and isactnan:
263: (12)      return # both nan, so equal
264: (8)    array_actual = np.asarray(actual)
265: (8)    array_desired = np.asarray(desired)
266: (8)    if (array_actual.dtype.char in 'Mm' or
267: (16)          array_desired.dtype.char in 'Mm'):
268: (12)      raise NotImplementedError('cannot compare to a scalar '
269: (38)          'with a different type')
270: (8)    if desired == 0 and actual == 0:
271: (12)      if not signbit(desired) == signbit(actual):
272: (16)        raise AssertionError(msg)
273: (4)  except (TypeError, ValueError, NotImplementedError):
274: (8)    pass
275: (4)  try:
276: (8)    if not (desired == actual):
277: (12)      raise AssertionError(msg)
278: (4)  except (DeprecationWarning, FutureWarning) as e:
279: (8)    if 'elementwise == comparison' in e.args[0]:
280: (12)      raise AssertionError(msg)
281: (8)    else:
282: (12)      raise
283: (0) def print_assert_equal(test_string, actual, desired):
284: (4) """
285: (4) Test if two objects are equal, and print an error message if test fails.
286: (4) The test is performed with ``actual == desired``.
287: (4) Parameters
288: (4) -----
289: (4) test_string : str
290: (8)     The message supplied to AssertionError.
291: (4) actual : object
292: (8)     The object to test for equality against `desired`.
293: (4) desired : object
294: (8)     The expected result.
295: (4) Examples
296: (4) -----
297: (4) >>> np.testing.print_assert_equal('Test XYZ of func xyz', [0, 1], [0, 1])
298: (4) >>> np.testing.print_assert_equal('Test XYZ of func xyz', [0, 1], [0, 2])
299: (4) Traceback (most recent call last):
300: (4) ...
301: (4) AssertionError: Test XYZ of func xyz failed
302: (4) ACTUAL:
303: (4) [0, 1]
304: (4) DESIRED:
305: (4) [0, 2]
306: (4) """
307: (4) __tracebackhide__ = True # Hide traceback for py.test
308: (4) import pprint
309: (4) if not (actual == desired):
310: (8)   msg = StringIO()

```

```

311: (8)                         msg.write(test_string)
312: (8)                         msg.write(' failed\nACTUAL: \n')
313: (8)                         pprint.pprint(actual, msg)
314: (8)                         msg.write('DESIRED: \n')
315: (8)                         pprint.pprint(desired, msg)
316: (8)                         raise AssertionError(msg.getvalue())
317: (0) @np._no_nep50_warning()
318: (0) def assert_almost_equal(actual, desired, decimal=7, err_msg='', verbose=True):
319: (4)     """
320: (4)         Raises an AssertionError if two items are not equal up to desired
321: (4)         precision.
322: (4)         .. note:: It is recommended to use one of `assert_allclose`,
323: (14)             `assert_array_almost_equal_nulp` or `assert_array_max_ulp`
324: (14)             instead of this function for more consistent floating point
325: (14)             comparisons.
326: (4)         The test verifies that the elements of `actual` and `desired` satisfy.
327: (8)             ``abs(desired-actual) < float64(1.5 * 10**(-decimal))``
328: (4)         That is a looser test than originally documented, but agrees with what the
329: (4)         actual implementation in `assert_array_almost_equal` did up to rounding
330: (4)         vagaries. An exception is raised at conflicting values. For ndarrays this
331: (4)         delegates to assert_array_almost_equal
332: (4)         Parameters
333: (4)             -----
334: (4)             actual : array_like
335: (8)                 The object to check.
336: (4)             desired : array_like
337: (8)                 The expected object.
338: (4)             decimal : int, optional
339: (8)                 Desired precision, default is 7.
340: (4)             err_msg : str, optional
341: (8)                 The error message to be printed in case of failure.
342: (4)             verbose : bool, optional
343: (8)                 If True, the conflicting values are appended to the error message.
344: (4)             Raises
345: (4)             -----
346: (4)             AssertionError
347: (6)                 If actual and desired are not equal up to specified precision.
348: (4)             See Also
349: (4)             -----
350: (4)             assert_allclose: Compare two array_like objects for equality with desired
351: (21)                 relative and/or absolute precision.
352: (4)             assert_array_almost_nulp, assert_array_max_ulp, assert_equal
353: (4)             Examples
354: (4)             -----
355: (4)             >>> from numpy.testing import assert_almost_equal
356: (4)             >>> assert_almost_equal(2.3333333333333, 2.33333334)
357: (4)             >>> assert_almost_equal(2.3333333333333, 2.33333334, decimal=10)
358: (4)             Traceback (most recent call last):
359: (8)                 ...
360: (4)             AssertionError:
361: (4)                 Arrays are not almost equal to 10 decimals
362: (5)                 ACTUAL: 2.3333333333333
363: (5)                 DESIRED: 2.33333334
364: (4)                 >>> assert_almost_equal(np.array([1.0, 2.3333333333333]), ...
365: (4)                               np.array([1.0, 2.33333334]), decimal=9)
366: (4)                 Traceback (most recent call last):
367: (8)                     ...
368: (4)                 AssertionError:
369: (4)                 Arrays are not almost equal to 9 decimals
370: (4)                 <BLANKLINE>
371: (4)                 Mismatched elements: 1 / 2 (50%)
372: (4)                 Max absolute difference: 6.66669964e-09
373: (4)                 Max relative difference: 2.85715698e-09
374: (5)                 x: array([1.          , 2.333333333])
375: (5)                 y: array([1.          , 2.33333334])
376: (4)                 """
377: (4)                 __tracebackhide__ = True # Hide traceback for py.test
378: (4)                 from numpy.core import ndarray
379: (4)                 from numpy.lib import iscomplexobj, real, imag

```

```

380: (4)
381: (8)     try:
382: (4)         usecomplex = iscomplexobj(actual) or iscomplexobj(desired)
383: (8)     except ValueError:
384: (4)         usecomplex = False
385: (8)     def _build_err_msg():
386: (8)         header = ('Arrays are not almost equal to %d decimals' % decimal)
387: (29)         return build_err_msg([actual, desired], err_msg, verbose=verbose,
388: (4)                         header=header)
389: (8)     if usecomplex:
390: (12)         if iscomplexobj(actual):
391: (12)             actualr = real(actual)
392: (8)             actuali = imag(actual)
393: (12)         else:
394: (12)             actualr = actual
395: (8)             actuali = 0
396: (12)         if iscomplexobj(desired):
397: (12)             desiredr = real(desired)
398: (8)             desiredi = imag(desired)
399: (12)         else:
400: (12)             desiredr = desired
401: (8)             desiredi = 0
402: (12)     try:
403: (12)         assert_almost_equal(actualr, desiredr, decimal=decimal)
404: (8)         assert_almost_equal(actuali, desiredi, decimal=decimal)
405: (12)     except AssertionError:
406: (4)         raise AssertionError(_build_err_msg())
407: (12)     if isinstance(actual, (ndarray, tuple, list)) \
408: (8)         or isinstance(desired, (ndarray, tuple, list)):
409: (4)         return assert_array_almost_equal(actual, desired, decimal, err_msg)
410: (8)     try:
411: (12)         if not (isfinite(desired) and isfinite(actual)):
412: (16)             if isnan(desired) or isnan(actual):
413: (20)                 if not (isnan(desired) and isnan(actual)):
414: (12)                     raise AssertionError(_build_err_msg())
415: (16)                 else:
416: (20)                     if not desired == actual:
417: (12)                         raise AssertionError(_build_err_msg())
418: (4)         return
419: (8)     except (NotImplementedError, TypeError):
420: (4)         pass
421: (8)     if abs(desired - actual) >= np.float64(1.5 * 10.0**(-decimal)):
422: (0)         raise AssertionError(_build_err_msg())
@np._no_nep50_warning()
423: (0)     def assert_approx_equal(actual, desired, significant=7, err_msg='',
424: (24)                     verbose=True):
425: (4)
426: (4)         """ Raises an AssertionError if two items are not equal up to significant
427: (4)         digits.
428: (4)         .. note:: It is recommended to use one of `assert_allclose`,
429: (14)             `assert_array_almost_equal_nulp` or `assert_array_max_ulp`
430: (14)             instead of this function for more consistent floating point
431: (14)             comparisons.
432: (4)         Given two numbers, check that they are approximately equal.
433: (4)         Approximately equal is defined as the number of significant digits
434: (4)         that agree.
435: (4)         Parameters
436: (4)         -----
437: (4)         actual : scalar
438: (8)             The object to check.
439: (4)         desired : scalar
440: (8)             The expected object.
441: (4)         significant : int, optional
442: (8)             Desired precision, default is 7.
443: (4)         err_msg : str, optional
444: (8)             The error message to be printed in case of failure.
445: (4)         verbose : bool, optional
446: (8)             If True, the conflicting values are appended to the error message.
447: (4)         Raises
448: (4)         -----

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

449: (4)           AssertionError
450: (6)             If actual and desired are not equal up to specified precision.
451: (4)             See Also
452: (4)
453: (4)             -----
454: (21)            assert_allclose: Compare two array_like objects for equality with desired
455: (4)               relative and/or absolute precision.
456: (4)             assert_array_almost_equal_nulp, assert_array_max_ulp, assert_equal
457: (4)             Examples
458: (4)             -----
459: (4)             >>> np.testing.assert_approx_equal(0.12345677777777e-20, 0.1234567e-20)
460: (4)             >>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345671e-20,
461: (4)                           ...                                         significant=8)
462: (4)             >>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345672e-20,
463: (4)                           ...                                         significant=8)
464: (8)             Traceback (most recent call last):
465: (4)               ...
466: (4)             AssertionError:
467: (5)               Items are not equal to 8 significant digits:
468: (5)               ACTUAL: 1.234567e-21
469: (4)               DESIRED: 1.2345672e-21
470: (4)               the evaluated condition that raises the exception is
471: (4)               >>> abs(0.12345670e-20/1e-21 - 0.12345672e-20/1e-21) >= 10**-(8-1)
472: (4)               True
473: (4)               """
474: (4)               __tracebackhide__ = True # Hide traceback for py.test
475: (4)               import numpy as np
476: (4)               (actual, desired) = map(float, (actual, desired))
477: (8)               if desired == actual:
478: (4)                   return
479: (8)               with np.errstate(invalid='ignore'):
480: (8)                   scale = 0.5*(np.abs(desired) + np.abs(actual))
481: (4)                   scale = np.power(10, np.floor(np.log10(scale)))
482: (8)               try:
483: (4)                   sc_desired = desired/scale
484: (8)               except ZeroDivisionError:
485: (4)                   sc_desired = 0.0
486: (8)               try:
487: (4)                   sc_actual = actual/scale
488: (8)               except ZeroDivisionError:
489: (4)                   sc_actual = 0.0
490: (8)               msg = build_err_msg(
491: (8)                   [actual, desired], err_msg,
492: (8)                   header='Items are not equal to %d significant digits:' % significant,
493: (4)                   verbose=verbose)
494: (8)               try:
495: (12)                   if not (isfinite(desired) and isfinite(actual)):
496: (16)                       if isnan(desired) or isnan(actual):
497: (20)                           if not (isnan(desired) and isnan(actual)):
498: (12)                               raise AssertionError(msg)
499: (16)
500: (20)
501: (12)
502: (4)
503: (8)
504: (4)
505: (8)
506: (0)
507: (0)
header='',
508: (25)
509: (25)
510: (4)
511: (4)
512: (28)
513: (4)
514: (4)
515: (4)
516: (4)
@np._no_nep50_warning()
def assert_array_compare(comparison, x, y, err_msg='', verbose=True,
precision=6, equal_nan=True, equal_inf=True,
*, strict=False):
    __tracebackhide__ = True # Hide traceback for py.test
    from numpy.core import (array2string, isnan, inf, bool_, errstate,
all, max, object_)
    x = np.asanyarray(x)
    y = np.asanyarray(y)
    ox, oy = x, y
    def isnumber(x):

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

517: (8)           return x.dtype.char in '?bhilqpBHILQPefdgFDG'
518: (4)           def istime(x):
519: (8)             return x.dtype.char in "Mm"
520: (4)           def func_assert_same_pos(x, y, func=isnan, hasval='nan'):
521: (8)             """Handling nan/inf.
522: (8)             Combine results of running func on x and y, checking that they are
True
523: (8)             at the same locations.
524: (8)
525: (8)             """
526: (8)             __tracebackhide__ = True # Hide traceback for py.test
527: (8)             x_id = func(x)
528: (8)             y_id = func(y)
529: (12)            if bool_(x_id == y_id).all() != True:
530: (32)              msg = build_err_msg([x, y],
531: (32)                  err_msg + '\nx and y %s location mismatch:' %
532: (32)                      (hasval), verbose=verbose, header=header,
533: (12)                      names=('x', 'y'), precision=precision)
534: (8)              raise AssertionError(msg)
535: (12)            if isinstance(x_id, bool) or x_id.ndim == 0:
536: (8)              return bool_(x_id)
537: (12)            elif isinstance(y_id, bool) or y_id.ndim == 0:
538: (8)              return bool_(y_id)
539: (12)            else:
540: (4)              return y_id
try:
541: (8)            if strict:
542: (12)              cond = x.shape == y.shape and x.dtype == y.dtype
543: (8)
544: (12)            else:
545: (8)              cond = (x.shape == () or y.shape == ()) or x.shape == y.shape
546: (12)            if not cond:
547: (16)              if x.shape != y.shape:
548: (12)                reason = f'\n(shapes {x.shape}, {y.shape} mismatch)'
549: (16)              else:
550: (12)                reason = f'\n(dtypes {x.dtype}, {y.dtype} mismatch)'
551: (32)            msg = build_err_msg([x, y],
552: (32)                err_msg
553: (32)                + reason,
554: (32)                verbose=verbose, header=header,
555: (12)                names=('x', 'y'), precision=precision)
556: (8)            raise AssertionError(msg)
flagged = bool_(False)
557: (8)            if isnumber(x) and isnumber(y):
558: (12)              if equal_nan:
559: (16)                flagged = func_assert_same_pos(x, y, func=isnan, hasval='nan')
560: (12)              if equal_inf:
561: (16)                flagged |= func_assert_same_pos(x, y,
562: (48)                    func=lambda xy: xy == +inf,
563: (48)                    hasval='+inf')
564: (16)                flagged |= func_assert_same_pos(x, y,
565: (48)                    func=lambda xy: xy == -inf,
566: (48)                    hasval='-inf')
567: (8)            elif istime(x) and istime(y):
568: (12)              if equal_nan and x.dtype.type == y.dtype.type:
569: (16)                flagged = func_assert_same_pos(x, y, func=isnat, hasval="NaT")
570: (8)            if flagged.ndim > 0:
571: (12)              x, y = x[~flagged], y[~flagged]
572: (12)              if x.size == 0:
573: (16)                return
574: (8)              elif flagged:
575: (12)                return
576: (8)              val = comparison(x, y)
577: (8)              if isinstance(val, bool):
578: (12)                cond = val
579: (12)                reduced = array([val])
580: (8)
581: (12)              else:
582: (12)                reduced = val.ravel()
583: (8)                cond = reduced.all()
584: (12)              if cond != True:
n_mismatch = reduced.size - reduced.sum(dtype=intp)

```

```

585: (12)           n_elements = flagged.size if flagged.ndim != 0 else reduced.size
586: (12)           percent_mismatch = 100 * n_mismatch / n_elements
587: (12)           remarks = [
588: (16)             'Mismatched elements: {} / {} ({:.3g}%)'.format(
589: (20)               n_mismatch, n_elements, percent_mismatch)]
590: (12)           with errstate(all='ignore'):
591: (16)             with contextlib.suppress(TypeError):
592: (20)               error = abs(x - y)
593: (20)               if np.issubdtype(x.dtype, np.unsignedinteger):
594: (24)                 error2 = abs(y - x)
595: (24)                 np.minimum(error, error2, out=error)
596: (20)               max_abs_error = max(error)
597: (20)               if getattr(error, 'dtype', object_) == object_:
598: (24)                 remarks.append('Max absolute difference: '
599: (39)                   + str(max_abs_error))
600: (20)             else:
601: (24)               remarks.append('Max absolute difference: '
602: (39)                   + array2string(max_abs_error))
603: (20)             nonzero = bool_(y != 0)
604: (20)             if all(~nonzero):
605: (24)               max_rel_error = array(inf)
606: (20)             else:
607: (24)               max_rel_error = max(error[nonzero] / abs(y[nonzero]))
608: (20)             if getattr(error, 'dtype', object_) == object_:
609: (24)               remarks.append('Max relative difference: '
610: (39)                   + str(max_rel_error))
611: (20)             else:
612: (24)               remarks.append('Max relative difference: '
613: (39)                   + array2string(max_rel_error))
614: (12)             err_msg += '\n' + '\n'.join(remarks)
615: (12)             msg = build_err_msg([ox, oy], err_msg,
616: (32)                           verbose=verbose, header=header,
617: (32)                           names=('x', 'y'), precision=precision)
618: (12)             raise AssertionError(msg)
619: (4)           except ValueError:
620: (8)             import traceback
621: (8)             efmt = traceback.format_exc()
622: (8)             header = f'error during assertion:\n\n{efmt}\n\n{header}'
623: (8)             msg = build_err_msg([x, y], err_msg, verbose=verbose, header=header,
624: (28)                           names=('x', 'y'), precision=precision)
625: (8)             raise ValueError(msg)
626: (0)           def assert_array_equal(x, y, err_msg='', verbose=True, *, strict=False):
627: (4)             """
628: (4)             Raises an AssertionError if two array_like objects are not equal.
629: (4)             Given two array_like objects, check that the shape is equal and all
630: (4)             elements of these objects are equal (but see the Notes for the special
631: (4)             handling of a scalar). An exception is raised at shape mismatch or
632: (4)             conflicting values. In contrast to the standard usage in numpy, NaNs
633: (4)             are compared like numbers, no assertion is raised if both objects have
634: (4)             NaNs in the same positions.
635: (4)             The usual caution for verifying equality with floating point numbers is
636: (4)             advised.
637: (4)             Parameters
638: (4)             -----
639: (4)             x : array_like
640: (8)               The actual object to check.
641: (4)             y : array_like
642: (8)               The desired, expected object.
643: (4)             err_msg : str, optional
644: (8)               The error message to be printed in case of failure.
645: (4)             verbose : bool, optional
646: (8)               If True, the conflicting values are appended to the error message.
647: (4)             strict : bool, optional
648: (8)               If True, raise an AssertionError when either the shape or the data
649: (8)               type of the array_like objects does not match. The special
650: (8)               handling for scalars mentioned in the Notes section is disabled.
651: (8)               .. versionadded:: 1.24.0
652: (4)             Raises
653: (4)             -----

```

```

654: (4)           AssertionError
655: (8)             If actual and desired objects are not equal.
656: (4)           See Also
657: (4)             -----
658: (4)               assert_allclose: Compare two array_like objects for equality with desired
659: (21)                 relative and/or absolute precision.
660: (4)               assert_array_almost_equal_nulp, assert_array_max_ulp, assert_equal
661: (4)           Notes
662: (4)             -----
663: (4)               When one of `x` and `y` is a scalar and the other is array_like, the
664: (4)                 function checks that each element of the array_like object is equal to
665: (4)                 the scalar. This behaviour can be disabled with the `strict` parameter.
666: (4)           Examples
667: (4)             -----
668: (4)               The first assert does not raise an exception:
669: (4)                 >>> np.testing.assert_array_equal([1.0, 2.33333, np.nan],
670: (4)                               ...                           [np.exp(0), 2.33333, np.nan])
671: (4)               Assert fails with numerical imprecision with floats:
672: (4)                 >>> np.testing.assert_array_equal([1.0, np.pi, np.nan],
673: (4)                               ...                           [1, np.sqrt(np.pi)**2, np.nan])
674: (4)               Traceback (most recent call last):
675: (8)                 ...
676: (4)           AssertionError:
677: (4)             Arrays are not equal
678: (4)             <BLANKLINE>
679: (4)               Mismatched elements: 1 / 3 (33.3%)
680: (4)               Max absolute difference: 4.4408921e-16
681: (4)               Max relative difference: 1.41357986e-16
682: (5)                 x: array([1.          , 3.141593,       nan])
683: (5)                 y: array([1.          , 3.141593,       nan])
684: (4)               Use `assert_allclose` or one of the nulp (number of floating point values)
685: (4)               functions for these cases instead:
686: (4)                 >>> np.testing.assert_allclose([1.0, np.pi, np.nan],
687: (4)                               ...                           [1, np.sqrt(np.pi)**2, np.nan],
688: (4)                               ...                           rtol=1e-10, atol=0)
689: (4)               As mentioned in the Notes section, `assert_array_equal` has special
690: (4)               handling for scalars. Here the test checks that each value in `x` is 3:
691: (4)                 >>> x = np.full((2, 5), fill_value=3)
692: (4)                 >>> np.testing.assert_array_equal(x, 3)
693: (4)               Use `strict` to raise an AssertionError when comparing a scalar with an
694: (4)               array:
695: (4)                 >>> np.testing.assert_array_equal(x, 3, strict=True)
696: (4)               Traceback (most recent call last):
697: (8)                 ...
698: (4)           AssertionError:
699: (4)             Arrays are not equal
700: (4)             <BLANKLINE>
701: (4)               (shapes (2, 5), () mismatch)
702: (5)                 x: array([[3, 3, 3, 3, 3],
703: (11)                   [3, 3, 3, 3, 3]])
704: (5)                 y: array(3)
705: (4)               The `strict` parameter also ensures that the array data types match:
706: (4)                 >>> x = np.array([2, 2, 2])
707: (4)                 >>> y = np.array([2., 2., 2.], dtype=np.float32)
708: (4)                 >>> np.testing.assert_array_equal(x, y, strict=True)
709: (4)               Traceback (most recent call last):
710: (8)                 ...
711: (4)           AssertionError:
712: (4)             Arrays are not equal
713: (4)             <BLANKLINE>
714: (4)               (dtypes int64, float32 mismatch)
715: (5)                 x: array([2, 2, 2])
716: (5)                 y: array([2., 2., 2.], dtype=float32)
717: (4)               """
718: (4)                 __tracebackhide__ = True # Hide traceback for py.test
719: (4)                 assert_array_compare(operator.__eq__, x, y, err_msg=err_msg,
720: (25)                               verbose=verbose, header='Arrays are not equal',
721: (25)                               strict=strict)
722: (0)               @np._no_nep50_warning()

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

723: (0)
724: (4)
725: (4)
726: (4)
727: (4)
728: (14)
729: (14)
730: (14)
731: (4)
732: (4)
733: (8)
734: (4)
735: (4)
736: (4)
737: (4)
738: (4)
739: (4)
740: (4)
741: (4)
742: (8)
743: (4)
744: (8)
745: (4)
746: (8)
747: (4)
748: (6)
749: (4)
750: (8)
751: (4)
752: (4)
753: (4)
754: (8)
755: (4)
756: (4)
757: (4)
758: (21)
759: (4)
760: (4)
761: (4)
762: (4)
763: (4)
764: (4)
765: (4)
766: (4)
767: (4)
768: (8)
769: (4)
770: (4)
771: (4)
772: (4)
773: (4)
774: (4)
775: (5)
776: (5)
777: (4)
778: (4)
779: (4)
780: (8)
781: (4)
782: (4)
783: (4)
784: (4)
785: (5)
786: (5)
787: (4)
788: (4)
789: (4)
790: (4)
791: (4)

def assert_array_almost_equal(x, y, decimal=6, err_msg='', verbose=True):
    """
    Raises an AssertionError if two objects are not equal up to desired
    precision.
    .. note:: It is recommended to use one of `assert_allclose`,
              `assert_array_almost_equal_nulp` or `assert_array_max_ulp`
              instead of this function for more consistent floating point
              comparisons.
    The test verifies identical shapes and that the elements of ``actual`` and
    ``desired`` satisfy.
        ``abs(desired-actual) < 1.5 * 10**(-decimal)``
    That is a looser test than originally documented, but agrees with what the
    actual implementation did up to rounding vagaries. An exception is raised
    at shape mismatch or conflicting values. In contrast to the standard usage
    in numpy, NaNs are compared like numbers, no assertion is raised if both
    objects have NaNs in the same positions.
    Parameters
    -----
    x : array_like
        The actual object to check.
    y : array_like
        The desired, expected object.
    decimal : int, optional
        Desired precision, default is 6.
    err_msg : str, optional
        The error message to be printed in case of failure.
    verbose : bool, optional
        If True, the conflicting values are appended to the error message.
    Raises
    -----
    AssertionError
        If actual and desired are not equal up to specified precision.
    See Also
    -----
    assert_allclose: Compare two array_like objects for equality with desired
                    relative and/or absolute precision.
    assert_array_almost_equal_nulp, assert_array_max_ulp, assert_equal
    Examples
    -----
    the first assert does not raise an exception
    >>> np.testing.assert_array_almost_equal([1.0,2.333,np.nan],
...                                     [1.0,2.333,np.nan])
    >>> np.testing.assert_array_almost_equal([1.0,2.33333,np.nan],
...                                     [1.0,2.33339,np.nan], decimal=5)
    Traceback (most recent call last):
    ...
    AssertionError:
    Arrays are not almost equal to 5 decimals
    <BLANKLINE>
    Mismatched elements: 1 / 3 (33.3%)
    Max absolute difference: 6.e-05
    Max relative difference: 2.57136612e-05
    x: array([1.      , 2.33333,     nan])
    y: array([1.      , 2.33339,     nan])
    >>> np.testing.assert_array_almost_equal([1.0,2.33333,np.nan],
...                                     [1.0,2.33333, 5], decimal=5)
    Traceback (most recent call last):
    ...
    AssertionError:
    Arrays are not almost equal to 5 decimals
    <BLANKLINE>
    x and y nan location mismatch:
    x: array([1.      , 2.33333,     nan])
    y: array([1.      , 2.33333, 5.      ])
    """
    _tracebackhide_ = True # Hide traceback for py.test
    from numpy.core import number, float_, result_type
    from numpy.core.numerictypes import issubdtype
    from numpy.core.fromnumeric import any as npany

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

792: (4)
793: (8)
794: (12)
795: (16)
796: (16)
797: (16)
798: (20)
799: (16)
800: (20)
801: (16)
802: (16)
803: (8)
804: (12)
805: (8)
806: (8)
807: (8)
808: (8)
809: (12)
810: (8)
811: (4)
812: (13)
813: (13)
814: (0)
815: (4)
816: (4)
817: (4)
818: (4)
819: (4)
820: (4)
821: (4)
822: (4)
823: (4)
824: (4)
825: (4)
826: (4)
827: (4)
828: (6)
829: (4)
830: (6)
831: (4)
832: (6)
833: (4)
834: (8)
835: (4)
836: (4)
837: (4)
838: (6)
839: (4)
840: (4)
841: (4)
842: (4)
843: (4)
844: (4)
845: (4)
846: (4)
847: (4)
848: (8)
849: (4)
850: (4)
851: (4)
852: (4)
853: (4)
854: (4)
855: (5)
856: (5)
857: (4)
858: (4)
859: (8)
860: (4)

def compare(x, y):
    try:
        if np.isnan(isinf(x)) or np.isnan(isinf(y)):
            xinfid = isinf(x)
            yinfid = isinf(y)
            if not (xinfid == yinfid).all():
                return False
            if x.size == y.size == 1:
                return x == y
            x = x[~xinfid]
            y = y[~yinfid]
        except (TypeError, NotImplementedError):
            pass
        dtype = result_type(y, 1.)
        y = np.asarray(y, dtype)
        z = abs(x - y)
        if not issubdtype(z.dtype, number):
            z = z.astype(float_) # handle object arrays
        return z < 1.5 * 10.0**(-decimal)
    assert_array_compare(compare, x, y, err_msg=err_msg, verbose=verbose,
                         header=('Arrays are not almost equal to %d decimals' % decimal),
                         precision=decimal)
def assert_array_less(x, y, err_msg='', verbose=True):
    """
    Raises an AssertionError if two array_like objects are not ordered by less
    than.
    Given two array_like objects, check that the shape is equal and all
    elements of the first object are strictly smaller than those of the
    second object. An exception is raised at shape mismatch or incorrectly
    ordered values. Shape mismatch does not raise if an object has zero
    dimension. In contrast to the standard usage in numpy, NaNs are
    compared, no assertion is raised if both objects have NaNs in the same
    positions.
    Parameters
    -----
    x : array_like
        The smaller object to check.
    y : array_like
        The larger object to compare.
    err_msg : string
        The error message to be printed in case of failure.
    verbose : bool
        If True, the conflicting values are appended to the error message.
    Raises
    -----
    AssertionError
        If x is not strictly smaller than y, element-wise.
    See Also
    -----
    assert_array_equal: tests objects for equality
    assert_array_almost_equal: test objects for equality up to precision
    Examples
    -----
    >>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1.1, 2.0, np.nan])
    >>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1, 2.0, np.nan])
    Traceback (most recent call last):
    ...
    AssertionError:
    Arrays are not less-ordered
    <BLANKLINE>
    Mismatched elements: 1 / 3 (33.3%)
    Max absolute difference: 1.
    Max relative difference: 0.5
    x: array([ 1.,  1., nan])
    y: array([ 1.,  2., nan])
    >>> np.testing.assert_array_less([1.0, 4.0], 3)
    Traceback (most recent call last):
    ...
    AssertionError:

```

```

861: (4)          Arrays are not less-ordered
862: (4)          <BLANKLINE>
863: (4)          Mismatched elements: 1 / 2 (50%)
864: (4)          Max absolute difference: 2.
865: (4)          Max relative difference: 0.66666667
866: (5)          x: array([1., 4.])
867: (5)          y: array(3)
868: (4)          >>> np.testing.assert_array_less([1.0, 2.0, 3.0], [4])
869: (4)          Traceback (most recent call last):
870: (8)          ...
871: (4)          AssertionError:
872: (4)          Arrays are not less-ordered
873: (4)          <BLANKLINE>
874: (4)          (shapes (3,), (1,) mismatch)
875: (5)          x: array([1., 2., 3.])
876: (5)          y: array([4])
877: (4)          """
878: (4)          __tracebackhide__ = True # Hide traceback for py.test
879: (4)          assert_array_compare(operator._lt_, x, y, err_msg=err_msg,
880: (25)                  verbose=verbose,
881: (25)                  header='Arrays are not less-ordered',
882: (25)                  equal_inf=False)
883: (0)          def runstring(astr, dict):
884: (4)          exec(astr, dict)
885: (0)          def assert_string_equal(actual, desired):
886: (4)          """
887: (4)          Test if two strings are equal.
888: (4)          If the given strings are equal, `assert_string_equal` does nothing.
889: (4)          If they are not equal, an AssertionError is raised, and the diff
890: (4)          between the strings is shown.
891: (4)          Parameters
892: (4)          -----
893: (4)          actual : str
894: (8)          The string to test for equality against the expected string.
895: (4)          desired : str
896: (8)          The expected string.
897: (4)          Examples
898: (4)          -----
899: (4)          >>> np.testing.assert_string_equal('abc', 'abc')
900: (4)          >>> np.testing.assert_string_equal('abc', 'abcd')
901: (4)          Traceback (most recent call last):
902: (6)          File "<stdin>", line 1, in <module>
903: (4)          ...
904: (4)          AssertionError: Differences in strings:
905: (4)          - abc+ abcd?      +
906: (4)          """
907: (4)          __tracebackhide__ = True # Hide traceback for py.test
908: (4)          import difflib
909: (4)          if not isinstance(actual, str):
910: (8)              raise AssertionError(repr(type(actual)))
911: (4)          if not isinstance(desired, str):
912: (8)              raise AssertionError(repr(type(desired)))
913: (4)          if desired == actual:
914: (8)              return
915: (4)          diff = list(difflib.Differ().compare(actual.splitlines(True),
916: (16)                      desired.splitlines(True)))
917: (4)          diff_list = []
918: (4)          while diff:
919: (8)              d1 = diff.pop(0)
920: (8)              if d1.startswith(' '):
921: (12)                  continue
922: (8)              if d1.startswith('- '):
923: (12)                  l = [d1]
924: (12)                  d2 = diff.pop(0)
925: (12)                  if d2.startswith('? '):
926: (16)                      l.append(d2)
927: (16)                      d2 = diff.pop(0)
928: (12)                  if not d2.startswith('+ '):
929: (16)                      raise AssertionError(repr(d2))

```

```

930: (12)             l.append(d2)
931: (12)             if diff:
932: (16)                 d3 = diff.pop(0)
933: (16)                 if d3.startswith('? '):
934: (20)                     l.append(d3)
935: (16)                 else:
936: (20)                     diff.insert(0, d3)
937: (12)                 if d2[2:] == d1[2:]:
938: (16)                     continue
939: (12)                 diff_list.extend(l)
940: (12)                 continue
941: (8)             raise AssertionError(repr(d1))
942: (4)             if not diff_list:
943: (8)                 return
944: (4)             msg = f"Differences in strings:\n{''.join(diff_list).rstrip()}"
945: (4)             if actual != desired:
946: (8)                 raise AssertionError(msg)
947: (0)             def rundocs(filename=None, raise_on_error=True):
948: (4)                 """
949: (4)                 Run doctests found in the given file.
950: (4)                 By default `rundocs` raises an AssertionError on failure.
951: (4)                 Parameters
952: (4)                 -----
953: (4)                 filename : str
954: (8)                     The path to the file for which the doctests are run.
955: (4)                 raise_on_error : bool
956: (8)                     Whether to raise an AssertionError when a doctest fails. Default is
957: (8)                     True.
958: (4)             Notes
959: (4)             -----
960: (4)             The doctests can be run by the user/developer by adding the ``doctests``
961: (4)             argument to the ``test()`` call. For example, to run all tests (including
962: (4)             doctests) for `numpy.lib`:
963: (4)             >>> np.lib.test(doctests=True) # doctest: +SKIP
964: (4)             """
965: (4)             from numpy.distutils.misc_util import exec_mod_from_location
966: (4)             import doctest
967: (4)             if filename is None:
968: (8)                 f = sys._getframe(1)
969: (8)                 filename = f.f_globals['__file__']
970: (4)                 name = os.path.splitext(os.path.basename(filename))[0]
971: (4)                 m = exec_mod_from_location(name, filename)
972: (4)                 tests = doctest.DocTestFinder().find(m)
973: (4)                 runner = doctest.DocTestRunner(verbose=False)
974: (4)                 msg = []
975: (4)                 if raise_on_error:
976: (8)                     out = lambda s: msg.append(s)
977: (4)                 else:
978: (8)                     out = None
979: (4)                 for test in tests:
980: (8)                     runner.run(test, out=out)
981: (4)                 if runner.failures > 0 and raise_on_error:
982: (8)                     raise AssertionError("Some doctests failed:\n%s" % "\n".join(msg))
983: (0)             def check_support_sve():
984: (4)                 """
985: (4)                 gh-22982
986: (4)                 """
987: (4)                 import subprocess
988: (4)                 cmd = 'lscpu'
989: (4)                 try:
990: (8)                     output = subprocess.run(cmd, capture_output=True, text=True)
991: (8)                     return 'sve' in output.stdout
992: (4)                 except OSError:
993: (8)                     return False
994: (0)             _SUPPORTS_SVE = check_support_sve()
995: (0)             import unittest
996: (0)             class _Dummy(unittest.TestCase):
997: (4)                 def nop(self):
998: (8)                     pass

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

999: (0)
1000: (0)
1001: (4)
1002: (4)
1003: (4)
1004: (4)
1005: (4)
1006: (4)
1007: (4)
1008: (4)
1009: (4)
1010: (4)
1011: (4)
1012: (4)
1013: (4)
1014: (4)
1015: (4)
1016: (4)
1017: (4)
1018: (4)
1019: (4)
1020: (4)
1021: (0)
1022: (4)
1023: (4)
1024: (24)
1025: (4)
1026: (4)
1027: (4)
1028: (4)
1029: (4)
1030: (4)
1031: (4)
1032: (4)
1033: (4)
1034: (4)
1035: (4)
**kwargs)
1036: (0)
1037: (4)
1038: (4)
1039: (4)
1040: (4)
1041: (4)
1042: (4)
1043: (4)
1044: (4)
1045: (4)
1046: (8)
1047: (4)
1048: (8)
1049: (4)
1050: (8)
1051: (8)
1052: (8)
1053: (8)
1054: (8)
1055: (4)
1056: (4)
1057: (8)
1058: (4)
1059: (8)
1060: (4)
1061: (4)
1062: (4)
1063: (4)
1064: (8)
1065: (12)
1066: (16)

_d = _Dummy('nop')
def assert_raises(*args, **kwargs):
    """
        assert_raises(exception_class, callable, *args, **kwargs)
        assert_raises(exception_class)
        Fail unless an exception of class exception_class is thrown
        by callable when invoked with arguments args and keyword
        arguments kwargs. If a different type of exception is
        thrown, it will not be caught, and the test case will be
        deemed to have suffered an error, exactly as for an
        unexpected exception.
        Alternatively, `assert_raises` can be used as a context manager:
    >>> from numpy.testing import assert_raises
    >>> with assert_raises(ZeroDivisionError):
        ...     1 / 0
    is equivalent to
    >>> def div(x, y):
        ...     return x / y
    >>> assert_raises(ZeroDivisionError, div, 1, 0)
    """
    __tracebackhide__ = True # Hide traceback for py.test
    return _d.assertRaises(*args, **kwargs)
def assert_raises_regex(exception_class, expected_re, *args, **kwargs):
    """
        assert_raises_regex(exception_class, expected_re, callable, *args,
                           **kwargs)
        assert_raises_regex(exception_class, expected_re)
        Fail unless an exception of class exception_class and with message that
        matches expected_re is thrown by callable when invoked with arguments
        args and keyword arguments kwargs.
        Alternatively, can be used as a context manager like `assert_raises`.
    Notes
    -----
    .. versionadded:: 1.9.0
    """
    __tracebackhide__ = True # Hide traceback for py.test
    return _d.assertRaisesRegex(exception_class, expected_re, *args,
                               **kwargs)
def decorate_methods(cls, decorator, testmatch=None):
    """
        Apply a decorator to all methods in a class matching a regular expression.
        The given decorator is applied to all public methods of `cls` that are
        matched by the regular expression `testmatch`
        (``testmatch.search(methodname)``). Methods that are private, i.e. start
        with an underscore, are ignored.
    Parameters
    -----
    cls : class
        Class whose methods to decorate.
    decorator : function
        Decorator to apply to methods
    testmatch : compiled regexp or str, optional
        The regular expression. Default value is None, in which case the
        nose default (``re.compile(r'^(?:(\b|[\.\%s-])[Tt]est') % os.sep```)
        is used.
        If `testmatch` is a string, it is compiled to a regular expression
        first.
    """
    if testmatch is None:
        testmatch = re.compile(r'^(?:(\b|[\.\%s-])[Tt]est') % os.sep)
    else:
        testmatch = re.compile(testmatch)
    cls_attr = cls.__dict__
    from inspect import isfunction
    methods = [_m for _m in cls_attr.values() if isfunction(_m)]
    for function in methods:
        try:
            if hasattr(function, 'compat_func_name'):
                funcname = function.compat_func_name

```

```

1067: (12)
1068: (16)
1069: (8)
1070: (12)
1071: (8)
1072: (12)
1073: (4)
1074: (0)
1075: (4)
1076: (4)
1077: (4)
1078: (4)
1079: (4)
1080: (4)
1081: (4)
1082: (4)
1083: (4)
1084: (8)
1085: (4)
1086: (8)
1087: (8)
1088: (4)
1089: (8)
1090: (8)
1091: (4)
1092: (4)
1093: (4)
1094: (8)
1095: (4)
1096: (4)
1097: (4)
1098: (4)
times=times)
1099: (4)
doctest: +SKIP
1100: (4)
1101: (4)
1102: (4)
1103: (4)
1104: (4)
1105: (4)
1106: (4)
1107: (4)
1108: (8)
1109: (8)
1110: (4)
1111: (4)
1112: (0)
1113: (4)
1114: (4)
1115: (4)
1116: (4)
1117: (4)
1118: (8)
1119: (4)
1120: (4)
1121: (4)
1122: (4)
1123: (4)
1124: (4)
1125: (4)
1126: (8)
1127: (8)
1128: (12)
1129: (8)
1130: (4)
1131: (8)
1132: (4)
1133: (0)

        else:
            funcname = function.__name__
        except AttributeError:
            continue
        if testmatch.search(funcname) and not funcname.startswith('_'):
            setattr(cls, funcname, decorator(function))
    return
def measure(code_str, times=1, label=None):
    """
    Return elapsed time for executing code in the namespace of the caller.
    The supplied code string is compiled with the Python builtin ``compile``.
    The precision of the timing is 10 milli-seconds. If the code will execute
    fast on this timescale, it can be executed many times to get reasonable
    timing accuracy.
    Parameters
    -----
    code_str : str
        The code to be timed.
    times : int, optional
        The number of times the code is executed. Default is 1. The code is
        only compiled once.
    label : str, optional
        A label to identify `code_str` with. This is passed into ``compile``

        as the second argument (for run-time error messages).
    Returns
    -----
    elapsed : float
        Total elapsed time in seconds for executing `code_str` `times` times.
    Examples
    -----
    >>> times = 10
    >>> etime = np.testing.measure('for i in range(1000): np.sqrt(i**2)',

    >>> print("Time for a single execution : ", etime / times, "s") #

    Time for a single execution :  0.005 s
    """
    frame = sys._getframe(1)
    locs, globs = frame.f_locals, frame.f_globals
    code = compile(code_str, f'Test name: {label}', 'exec')
    i = 0
    elapsed = jiffies()
    while i < times:
        i += 1
        exec(code, globs, locs)
    elapsed = jiffies() - elapsed
    return 0.01*elapsed
def _assert_valid_refcount(op):
    """
    Check that ufuncs don't mishandle refcount of object `1`.
    Used in a few regression tests.
    """
    if not HAS_REFCOUNT:
        return True
    import gc
    import numpy as np
    b = np.arange(100*100).reshape(100, 100)
    c = b
    i = 1
    gc.disable()
    try:
        rc = sys.getrefcount(i)
        for j in range(15):
            d = op(b, c)
            assert_(sys.getrefcount(i) >= rc)
    finally:
        gc.enable()
    del d # for pyflakes
    def assert_allclose(actual, desired, rtol=1e-7, atol=0, equal_nan=True,

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1134: (20)
1135: (4)
1136: (4)
1137: (4)
1138: (4)
1139: (4)
1140: (4)
1141: (4)
1142: (4)
1143: (4)
1144: (4)
difference
1145: (4)
1146: (4)
1147: (4)
1148: (4)
1149: (4)
1150: (8)
1151: (4)
1152: (8)
1153: (4)
1154: (8)
1155: (4)
1156: (8)
1157: (4)
1158: (8)
1159: (4)
1160: (8)
1161: (4)
1162: (8)
1163: (4)
1164: (4)
1165: (4)
1166: (8)
1167: (4)
1168: (4)
1169: (4)
1170: (4)
1171: (4)
1172: (4)
1173: (4)
1174: (4)
1175: (4)
1176: (4)
1177: (4)
1178: (4)
1179: (4)
1180: (4)
1181: (4)
1182: (4)
1183: (4)
1184: (8)
1185: (39)
1186: (4)
1187: (4)
1188: (4)
1189: (25)
1190: (0)
1191: (4)
1192: (4)
1193: (4)
1194: (4)
1195: (4)
1196: (4)
1197: (4)
1198: (8)
1199: (4)
1200: (8)
Notes).

```

err\_msg='', verbose=True):

"""

Raises an AssertionError if two objects are not equal up to desired tolerance.

Given two array\_like objects, check that their shapes and all elements are equal (but see the Notes for the special handling of a scalar). An exception is raised if the shapes mismatch or any values conflict. In contrast to the standard usage in numpy, NaNs are compared like numbers, no assertion is raised if both objects have NaNs in the same positions. The test is equivalent to ``allclose(actual, desired, rtol, atol)`` (note that ``allclose`` has different default values). It compares the difference between `actual` and `desired` to ``atol + rtol \* abs(desired)``.

.. versionadded:: 1.5.0

Parameters

-----

actual : array\_like  
    Array obtained.

desired : array\_like  
    Array desired.

rtol : float, optional  
    Relative tolerance.

atol : float, optional  
    Absolute tolerance.

equal\_nan : bool, optional.  
    If True, NaNs will compare equal.

err\_msg : str, optional  
    The error message to be printed in case of failure.

verbose : bool, optional  
    If True, the conflicting values are appended to the error message.

Raises

-----

AssertionError  
    If actual and desired are not equal up to specified precision.

See Also

-----

assert\_array\_almost\_equal\_nulp, assert\_array\_max\_ulp

Notes

-----

When one of `actual` and `desired` is a scalar and the other is array\_like, the function checks that each element of the array\_like object is equal to the scalar.

Examples

-----

```

>>> x = [1e-5, 1e-3, 1e-1]
>>> y = np.arccos(np.cos(x))
>>> np.testing.assert_allclose(x, y, rtol=1e-5, atol=0)
"""
__tracebackhide__ = True # Hide traceback for py.test
import numpy as np
def compare(x, y):
    return np.core.numeric.isclose(x, y, rtol=rtol, atol=atol,
                                    equal_nan=equal_nan)
actual, desired = np.asanyarray(actual), np.asanyarray(desired)
header = f'Not equal to tolerance rtol={rtol:g}, atol={atol:g}'
assert_array_compare(compare, actual, desired, err_msg=str(err_msg),
                      verbose=verbose, header=header, equal_nan=equal_nan)

```

def assert\_array\_almost\_equal\_nulp(x, y, nulp=1):

"""

Compare two arrays relatively to their spacing.

This is a relatively robust method to compare two arrays whose amplitude is variable.

Parameters

-----

x, y : array\_like  
    Input arrays.

nulp : int, optional  
    The maximum number of unit in the last place for tolerance (see Notes).

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1201: (8)                               Default is 1.
1202: (4)                               Returns
1203: (4)-----None
1204: (4)                               Raises
1205: (4)-----None
1206: (4)                               Assertion
1207: (4)-----Error
1208: (8)           If the spacing between `x` and `y` for one or more elements is larger
1209: (8)           than `nulp`.
1210: (4)                               See Also
1211: (4)-----None
1212: (4)           assert_array_max_ulp : Check that all items of arrays differ in at most
1213: (8)           N Units in the Last Place.
1214: (4)           spacing : Return the distance between x and the nearest adjacent number.
1215: (4)           Notes
1216: (4)-----None
1217: (4)           An assertion is raised if the following condition is not met::
1218: (8)           abs(x - y) <= nulp * spacing(maximum(abs(x), abs(y)))
1219: (4)           Examples
1220: (4)-----None
1221: (4)           >>> x = np.array([1., 1e-10, 1e-20])
1222: (4)           >>> eps = np.finfo(x.dtype).eps
1223: (4)           >>> np.testing.assert_array_almost_equal_nulp(x, x*eps/2 + x)
1224: (4)           >>> np.testing.assert_array_almost_equal_nulp(x, x*eps + x)
1225: (4)           Traceback (most recent call last):
1226: (6)           ...
1227: (4)           AssertionError: X and Y are not equal to 1 ULP (max is 2)
1228: (4)           """
1229: (4)           __tracebackhide__ = True # Hide traceback for py.test
1230: (4)           import numpy as np
1231: (4)           ax = np.abs(x)
1232: (4)           ay = np.abs(y)
1233: (4)           ref = nulp * np.spacing(np.where(ax > ay, ax, ay))
1234: (4)           if not np.all(np.abs(x-y) <= ref):
1235: (8)               if np.iscomplexobj(x) or np.iscomplexobj(y):
1236: (12)                   msg = "X and Y are not equal to %d ULP" % nulp
1237: (8)               else:
1238: (12)                   max_nulp = np.max(nulp_diff(x, y))
1239: (12)                   msg = "X and Y are not equal to %d ULP (max is %g)" % (nulp,
1240: (8)                         max_nulp)
1241: (0)           raise AssertionError(msg)
1242: (4)           def assert_array_max_ulp(a, b, maxulp=1, dtype=None):
1243: (4)           """
1244: (4)           Check that all items of arrays differ in at most N Units in the Last
1245: (4)           Place.
1246: (4)           Parameters
1247: (4)           -----
1248: (4)           a, b : array_like
1249: (8)               Input arrays to be compared.
1250: (8)           maxulp : int, optional
1251: (8)               The maximum number of units in the last place that elements of `a` and
1252: (8)               `b` can differ. Default is 1.
1253: (4)           dtype : dtype, optional
1254: (8)               Data-type to convert `a` and `b` to if given. Default is None.
1255: (4)           Returns
1256: (4)           -----
1257: (4)           ret : ndarray
1258: (8)               Array containing number of representable floating point numbers
1259: (8)               between
1260: (8)               items in `a` and `b`.
1261: (4)           Raises
1262: (4)           -----
1263: (4)           Assertion
1264: (8)               If one or more elements differ by more than `maxulp`.
1265: (4)           Notes
1266: (4)           -----
1267: (4)           For computing the ULP difference, this API does not differentiate between
1268: (8)           various representations of NAN (ULP difference between 0x7fc00000 and
0xffff00000

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1266: (4)           is zero).
1267: (4)           See Also
1268: (4)
1269: (4)           assert_array_almost_equal_nulp : Compare two arrays relatively to their
1270: (8)             spacing.
1271: (4)           Examples
1272: (4)
1273: (4)             >>> a = np.linspace(0., 1., 100)
1274: (4)             >>> res = np.testing.assert_array_maxulp(a, np.arcsin(np.sin(a)))
1275: (4)             """
1276: (4)               __tracebackhide__ = True # Hide traceback for py.test
1277: (4)             import numpy as np
1278: (4)             ret = nulp_diff(a, b, dtype)
1279: (4)             if not np.all(ret <= maxulp):
1280: (8)                 raise AssertionError("Arrays are not almost equal up to %g "
1281: (29)                           "ULP (max difference is %g ULP)" %
1282: (29)                           (maxulp, np.max(ret)))
1283: (4)             return ret
1284: (0)           def nulp_diff(x, y, dtype=None):
1285: (4)             """For each item in x and y, return the number of representable floating
1286: (4)               points between them.
1287: (4)               Parameters
1288: (4)
1289: (4)                 x : array_like
1290: (8)                   first input array
1291: (4)                 y : array_like
1292: (8)                   second input array
1293: (4)                 dtype : dtype, optional
1294: (8)                   Data-type to convert `x` and `y` to if given. Default is None.
1295: (4)               Returns
1296: (4)
1297: (4)                 nulp : array_like
1298: (8)                   number of representable floating point numbers between each item in x
1299: (8)                   and y.
1300: (4)               Notes
1301: (4)
1302: (4)               For computing the ULP difference, this API does not differentiate between
1303: (4)               various representations of NAN (ULP difference between 0x7fc00000 and
0xffffc00000
1304: (4)           is zero).
1305: (4)           Examples
1306: (4)
1307: (4)             >>> nulp_diff(1, 1 + np.finfo(x.dtype).eps)
1308: (4)             1.0
1309: (4)             """
1310: (4)             import numpy as np
1311: (4)             if dtype:
1312: (8)                 x = np.asarray(x, dtype=dtype)
1313: (8)                 y = np.asarray(y, dtype=dtype)
1314: (4)             else:
1315: (8)                 x = np.asarray(x)
1316: (8)                 y = np.asarray(y)
1317: (4)             t = np.common_type(x, y)
1318: (4)             if np.iscomplexobj(x) or np.iscomplexobj(y):
1319: (8)                 raise NotImplementedError("_nulp not implemented for complex array")
1320: (4)             x = np.array([x], dtype=t)
1321: (4)             y = np.array([y], dtype=t)
1322: (4)             x[np.isnan(x)] = np.nan
1323: (4)             y[np.isnan(y)] = np.nan
1324: (4)             if not x.shape == y.shape:
1325: (8)                 raise ValueError("x and y do not have the same shape: %s - %s" %
1326: (25)                               (x.shape, y.shape))
1327: (4)             def _diff(rx, ry, vdt):
1328: (8)                 diff = np.asarray(rx-ry, dtype=vdt)
1329: (8)                 return np.abs(diff)
1330: (4)             rx = integer_repr(x)
1331: (4)             ry = integer_repr(y)
1332: (4)             return _diff(rx, ry, t)
1333: (0)             def _integer_repr(x, vdt, comp):

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1334: (4)
1335: (4)
1336: (8)
1337: (4)
1338: (8)
1339: (12)
1340: (4)
1341: (0)
1342: (4)
1343: (4)
1344: (4)
1345: (4)
1346: (8)
1347: (4)
1348: (8)
1349: (4)
1350: (8)
1351: (4)
1352: (8)
1353: (0)
1354: (0)
1355: (4)
1356: (4)
1357: (8)
1358: (8)
1359: (8)
1360: (12)
1361: (12)
1362: (0)
1363: (4)
1364: (4)
1365: (4)
1366: (4)
1367: (4)
1368: (4)
1369: (4)
1370: (8)
1371: (12)
1372: (4)
1373: (4)
1374: (4)
1375: (4)
1376: (4)
1377: (8)
1378: (4)
1379: (8)
1380: (4)
1381: (8)
1382: (4)
1383: (8)
1384: (4)
1385: (4)
1386: (4)
1387: (4)
1388: (4)
1389: (4)
1390: (4)
1391: (4)
1392: (4)
1393: (4)
1394: (4)
1395: (4)
1396: (4)
1397: (4)
1398: (4)
1399: (4)
1400: (8)
1401: (4)
1402: (4)

        rx = x.view(vdt)
        if not (rx.size == 1):
            rx[rx < 0] = comp - rx[rx < 0]
        else:
            if rx < 0:
                rx = comp - rx
        return rx
    def integer_repr(x):
        """Return the signed-magnitude interpretation of the binary representation
        of x."""
        import numpy as np
        if x.dtype == np.float16:
            return _integer_repr(x, np.int16, np.int16(-2**15))
        elif x.dtype == np.float32:
            return _integer_repr(x, np.int32, np.int32(-2**31))
        elif x.dtype == np.float64:
            return _integer_repr(x, np.int64, np.int64(-2**63))
        else:
            raise ValueError(f'Unsupported dtype {x.dtype}')
    @contextlib.contextmanager
    def _assert_warns_context(warning_class, name=None):
        __tracebackhide__ = True # Hide traceback for py.test
        with suppress_warnings() as sup:
            l = sup.record(warning_class)
            yield
            if not len(l) > 0:
                name_str = f' when calling {name}' if name is not None else ''
                raise AssertionError("No warning raised" + name_str)
    def assert_warns(warning_class, *args, **kwargs):
        """
        Fail unless the given callable throws the specified warning.
        A warning of class warning_class should be thrown by the callable when
        invoked with arguments args and keyword arguments kwargs.
        If a different type of warning is thrown, it will not be caught.
        If called with all arguments other than the warning class omitted, may be
        used as a context manager:
        with assert_warns(SomeWarning):
            do_something()
        The ability to be used as a context manager is new in NumPy v1.11.0.
        .. versionadded:: 1.4.0
        Parameters
        -----
        warning_class : class
            The class defining the warning that `func` is expected to throw.
        func : callable, optional
            Callable to test
        *args : Arguments
            Arguments for `func`.
        **kwargs : Kwargs
            Keyword arguments for `func`.
        Returns
        -----
        The value returned by `func`.
        Examples
        -----
        >>> import warnings
        >>> def deprecated_func(num):
        ...     warnings.warn("Please upgrade", DeprecationWarning)
        ...     return num*num
        >>> with np.testing.assert_warns(DeprecationWarning):
        ...     assert deprecated_func(4) == 16
        >>> # or passing a func
        >>> ret = np.testing.assert_warns(DeprecationWarning, deprecated_func, 4)
        >>> assert ret == 16
        """
        if not args:
            return _assert_warns_context(warning_class)
        func = args[0]
        args = args[1:]

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1403: (4)
1404: (8)
1405: (0)
1406: (0)
1407: (4)
1408: (4)
1409: (8)
1410: (8)
1411: (8)
1412: (12)
1413: (12)
1414: (0)
1415: (4)
1416: (4)
1417: (4)
1418: (8)
1419: (12)
1420: (4)
1421: (4)
1422: (4)
1423: (4)
1424: (4)
1425: (8)
1426: (4)
1427: (8)
1428: (4)
1429: (8)
1430: (4)
1431: (4)
1432: (4)
1433: (4)
1434: (4)
1435: (8)
1436: (4)
1437: (4)
1438: (4)
1439: (8)
1440: (0)
1441: (4)
1442: (4)
1443: (4)
1444: (4)
1445: (4)
1446: (4)
1447: (8)
1448: (4)
1449: (8)
1450: (17)
1451: (8)
1452: (17)
1453: (4)
1454: (8)
1455: (4)
1456: (4)
1457: (4)
1458: (4)
1459: (4)
1460: (4)
1461: (4)
1462: (4)
1463: (4)
1464: (4)
1465: (8)
1466: (12)
1467: (16)
1468: (16)
1469: (16)
1470: (16)
1471: (16)

        with _assert_warnings_context(warning_class, name=func.__name__):
            return func(*args, **kwargs)
    @contextlib.contextmanager
    def _assert_no_warnings_context(name=None):
        __tracebackhide__ = True # Hide traceback for py.test
        with warnings.catch_warnings(record=True) as l:
            warnings.simplefilter('always')
            yield
            if len(l) > 0:
                name_str = f' when calling {name}' if name is not None else ''
                raise AssertionError(f'Got warnings{name_str}: {l}')
    def assert_no_warnings(*args, **kwargs):
        """
        Fail if the given callable produces any warnings.
        If called with all arguments omitted, may be used as a context manager:
            with assert_no_warnings():
                do_something()
        The ability to be used as a context manager is new in NumPy v1.11.0.
        .. versionadded:: 1.7.0
        Parameters
        -----
        func : callable
            The callable to test.
        \\\*args : Arguments
            Arguments passed to `func`.
        \\\*\\\*kwargs : Kwargs
            Keyword arguments passed to `func`.
        Returns
        -----
        The value returned by `func`.
        """
        if not args:
            return _assert_no_warnings_context()
        func = args[0]
        args = args[1:]
        with _assert_no_warnings_context(name=func.__name__):
            return func(*args, **kwargs)
    def _gen_alignment_data(dtype=float32, type='binary', max_size=24):
        """
        generator producing data with different alignment and offsets
        to test SIMD vectorization
        Parameters
        -----
        dtype : dtype
            Data type to produce
        type : string
            'unary': create data for unary operations, creates one input
                     and output array
            'binary': create data for binary operations, creates two input
                     and output array
        max_size : integer
            Maximum size of data to produce
        Returns
        -----
        if type is 'unary' yields one output, one input array and a message
        containing information on the data
        if type is 'binary' yields one output array, two input array and a message
        containing information on the data
        """
        ufmt = 'unary offset=(%d, %d), size=%d, dtype=%r, %s'
        bfmt = 'binary offset=(%d, %d, %d), size=%d, dtype=%r, %s'
        for o in range(3):
            for s in range(o + 2, max(o + 3, max_size)):
                if type == 'unary':
                    inp = lambda: arange(s, dtype=dtype)[o:]
                    out = empty((s,), dtype=dtype)[o:]
                    yield out, inp(), ufmt % (o, o, s, dtype, 'out of place')
                    d = inp()
                    yield d, d, ufmt % (o, o, s, dtype, 'in place')

```

```

1472: (16)             yield out[1:], inp()[:-1], ufmt % \
1473: (20)                 (o + 1, o, s - 1, dtype, 'out of place')
1474: (16)             yield out[:-1], inp()[1:], ufmt % \
1475: (20)                 (o, o + 1, s - 1, dtype, 'out of place')
1476: (16)             yield inp()[:-1], inp()[1:], ufmt % \
1477: (20)                 (o, o + 1, s - 1, dtype, 'aliased')
1478: (16)             yield inp()[1:], inp()[:-1], ufmt % \
1479: (20)                 (o + 1, o, s - 1, dtype, 'aliased')
1480: (12)             if type == 'binary':
1481: (16)                 inp1 = lambda: arange(s, dtype=dtype)[o:]
1482: (16)                 inp2 = lambda: arange(s, dtype=dtype)[o:]
1483: (16)                 out = empty((s,), dtype=dtype)[o:]
1484: (16)                 yield out, inp1(), inp2(), bfmt % \
1485: (20)                     (o, o, o, s, dtype, 'out of place')
1486: (16)                 d = inp1()
1487: (16)                 yield d, d, inp2(), bfmt % \
1488: (20)                     (o, o, o, s, dtype, 'in place1')
1489: (16)                 d = inp2()
1490: (16)                 yield d, inp1(), d, bfmt % \
1491: (20)                     (o, o, o, s, dtype, 'in place2')
1492: (16)             yield out[1:], inp1()[:-1], inp2()[:-1], bfmt % \
1493: (20)                     (o + 1, o, o, s - 1, dtype, 'out of place')
1494: (16)             yield out[:-1], inp1()[1:], inp2()[:-1], bfmt % \
1495: (20)                     (o, o + 1, o, s - 1, dtype, 'out of place')
1496: (16)             yield out[:-1], inp1()[:-1], inp2()[1:], bfmt % \
1497: (20)                     (o, o, o + 1, s - 1, dtype, 'out of place')
1498: (16)             yield inp1()[1:], inp1()[:-1], inp2()[:-1], bfmt % \
1499: (20)                     (o + 1, o, o, s - 1, dtype, 'aliased')
1500: (16)             yield inp1()[:-1], inp1()[1:], inp2()[:-1], bfmt % \
1501: (20)                     (o, o + 1, o, s - 1, dtype, 'aliased')
1502: (16)             yield inp1()[:-1], inp1()[:-1], inp2()[1:], bfmt % \
1503: (20)                     (o, o, o + 1, s - 1, dtype, 'aliased')
1504: (0)             class IgnoreException(Exception):
1505: (4)                 "Ignoring this exception due to disabled feature"
1506: (4)                 pass
1507: (0)             @contextlib.contextmanager
1508: (0)             def tempdir(*args, **kwargs):
1509: (4)                 """Context manager to provide a temporary test folder.
1510: (4)                 All arguments are passed as this to the underlying tempfile.mkdtemp
1511: (4)                 function.
1512: (4)
1513: (4)                 tmpdir = mkdtemp(*args, **kwargs)
1514: (4)                 try:
1515: (8)                     yield tmpdir
1516: (4)                 finally:
1517: (8)                     shutil.rmtree(tmpdir)
1518: (0)             @contextlib.contextmanager
1519: (0)             def temppath(*args, **kwargs):
1520: (4)                 """Context manager for temporary files.
1521: (4)                 Context manager that returns the path to a closed temporary file. Its
1522: (4)                 parameters are the same as for tempfile.mkstemp and are passed directly
1523: (4)                 to that function. The underlying file is removed when the context is
1524: (4)                 exited, so it should be closed at that time.
1525: (4)                 Windows does not allow a temporary file to be opened if it is already
1526: (4)                 open, so the underlying file must be closed after opening before it
1527: (4)                 can be opened again.
1528: (4)
1529: (4)                 fd, path = mkstemp(*args, **kwargs)
1530: (4)                 os.close(fd)
1531: (4)                 try:
1532: (8)                     yield path
1533: (4)                 finally:
1534: (8)                     os.remove(path)
1535: (0)             class clear_and_catch_warnings(warnings.catch_warnings):
1536: (4)                 """ Context manager that resets warning registry for catching warnings
1537: (4)                 Warnings can be slippery, because, whenever a warning is triggered, Python
1538: (4)                 adds a ``__warningregistry__`` member to the *calling* module. This makes
1539: (4)                 it impossible to retrigger the warning in this module, whatever you put in
1540: (4)                 the warnings filters. This context manager accepts a sequence of

```

```

`modules`
1541: (4)           as a keyword argument to its constructor and:
1542: (4)           * stores and removes any ``__warningregistry__`` entries in given
`modules`
1543: (6)           on entry;
1544: (4)           * resets ``__warningregistry__`` to its previous state on exit.
1545: (4)           This makes it possible to trigger any warning afresh inside the context
1546: (4)           manager without disturbing the state of warnings outside.
1547: (4)           For compatibility with Python 3.0, please consider all arguments to be
1548: (4)           keyword-only.
1549: (4)           Parameters
1550: (4)           -----
1551: (4)           record : bool, optional
1552: (8)             Specifies whether warnings should be captured by a custom
1553: (8)             implementation of ``warnings.showwarning()`` and be appended to a list
1554: (8)             returned by the context manager. Otherwise None is returned by the
1555: (8)             context manager. The objects appended to the list are arguments whose
1556: (8)             attributes mirror the arguments to ``showwarning()``.
1557: (4)           modules : sequence, optional
1558: (8)             Sequence of modules for which to reset warnings registry on entry and
1559: (8)             restore on exit. To work correctly, all 'ignore' filters should
1560: (8)             filter by one of these modules.
1561: (4)           Examples
1562: (4)           -----
1563: (4)           >>> import warnings
1564: (4)           >>> with np.testing.clear_and_catch_warnings(
1565: (4)             ...         modules=[np.core.fromnumeric]):
1566: (4)             ...         warnings.simplefilter('always')
1567: (4)             ...         warnings.filterwarnings('ignore', module='np.core.fromnumeric')
1568: (4)             ...         # do something that raises a warning but ignore those in
1569: (4)             ...         # np.core.fromnumeric
1570: (4)             """
1571: (4)             class_modules = ()
1572: (4)             def __init__(self, record=False, modules=()):
1573: (8)               self.modules = set(modules).union(self.class_modules)
1574: (8)               self._warnreg_copies = {}
1575: (8)               super().__init__(record=record)
1576: (4)             def __enter__(self):
1577: (8)               for mod in self.modules:
1578: (12)                 if hasattr(mod, '__warningregistry__'):
1579: (16)                   mod_reg = mod.__warningregistry__
1580: (16)                   self._warnreg_copies[mod] = mod_reg.copy()
1581: (16)                   mod_reg.clear()
1582: (8)                 return super().__enter__()
1583: (4)             def __exit__(self, *exc_info):
1584: (8)               super().__exit__(*exc_info)
1585: (8)               for mod in self.modules:
1586: (12)                 if hasattr(mod, '__warningregistry__'):
1587: (16)                   mod.__warningregistry__.clear()
1588: (12)                   if mod in self._warnreg_copies:
1589: (16)                       mod.__warningregistry__.update(self._warnreg_copies[mod])
1590: (0)
1591: (4)           class suppress_warnings:
1592: (4)             """
1593: (4)               Context manager and decorator doing much the same as
1594: (4)               ``warnings.catch_warnings``.
1595: (4)               However, it also provides a filter mechanism to work around
1596: (4)               https://bugs.python.org/issue4180.
1597: (4)               This bug causes Python before 3.4 to not reliably show warnings again
1598: (4)               after they have been ignored once (even within catch_warnings). It
1599: (4)               means that no "ignore" filter can be used easily, since following
1600: (4)               tests might need to see the warning. Additionally it allows easier
1601: (4)               specificity for testing warnings and can be nested.
1602: (4)               Parameters
1603: (4)               -----
1604: (8)               forwarding_rule : str, optional
1605: (8)                 One of "always", "once", "module", or "location". Analogous to
1606: (8)                 the usual warnings module filter mode, it is useful to reduce
1607: (8)                 noise mostly on the outmost level. Unsuppressed and unrecorded

```

```

1608: (8)           "location" is equivalent to the warnings "default", match by exact
1609: (8)           location the warning originated from.
1610: (4)           Notes
1611: (4)
1612: (4)           -----
1613: (4)           Filters added inside the context manager will be discarded again
1614: (4)           when leaving it. Upon entering all filters defined outside a
1615: (4)           context will be applied automatically.
1616: (4)           When a recording filter is added, matching warnings are stored in the
1617: (4)           ``log`` attribute as well as in the list returned by ``record``.
1618: (4)           If filters are added and the ``module`` keyword is given, the
1619: (4)           warning registry of this module will additionally be cleared when
1620: (4)           applying it, entering the context, or exiting it. This could cause
1621: (4)           warnings to appear a second time after leaving the context if they
1622: (4)           were configured to be printed once (default) and were already
1623: (4)           printed before the context was entered.
1624: (4)           Nesting this context manager will work as expected when the
1625: (4)           forwarding rule is "always" (default). Unfiltered and unrecorded
1626: (4)           warnings will be passed out and be matched by the outer level.
1627: (4)           On the outmost level they will be printed (or caught by another
1628: (4)           warnings context). The forwarding rule argument can modify this
1629: (4)           behaviour.
1630: (4)           Like ``catch_warnings`` this context manager is not threadsafe.
1631: (4)           Examples
1632: (4)
1633: (4)           With a context manager::
1634: (8)             with np.testing.suppress_warnings() as sup:
1635: (12)               sup.filter(DeprecationWarning, "Some text")
1636: (12)               sup.filter(module=np.ma.core)
1637: (12)               log = sup.record(FutureWarning, "Does this occur?")
1638: (12)               command_giving_warnings()
1639: (12)               assert_(len(log) == 1)
1640: (4)               assert_(len(sup.log) == 1) # also stored in log attribute
1641: (4)           Or as a decorator::
1642: (8)             sup = np.testing.suppress_warnings()
1643: (8)             sup.filter(module=np.ma.core) # module must match exactly
1644: (8)             @sup
1645: (12)             def some_function():
1646: (4)               pass
1647: (4)
1648: (8)             """
1649: (8)             def __init__(self, forwarding_rule="always"):
1650: (8)               self._entered = False
1651: (12)               self._suppressions = []
1652: (8)               if forwarding_rule not in {"always", "module", "once", "location"}:
1653: (4)                 raise ValueError("unsupported forwarding rule.")
1654: (8)               self._forwarding_rule = forwarding_rule
1655: (12)             def _clear_registries(self):
1656: (12)               if hasattr(warnings, "_filters_mutated"):
1657: (8)                 warnings._filters_mutated()
1658: (12)               return
1659: (16)               for module in self._tmp_modules:
1660: (4)                 if hasattr(module, "__warningregistry__"):
1661: (8)                   module.__warningregistry__.clear()
1662: (12)             def _filter(self, category=Warning, message="", module=None,
1663: (8)             record=False):
1664: (12)               if record:
1665: (8)                 record = [] # The log where to store warnings
1666: (12)               else:
1667: (8)                 record = None
1668: (12)               if self._entered:
1669: (16)                 if module is None:
1670: (20)                   warnings.filterwarnings(
1671: (16)                     "always", category=category, message=message)
1672: (20)                 else:
1673: (20)                   module_regex = module.__name__.replace('.', r'\.') + '$'
1674: (16)                   warnings.filterwarnings(
1675: (16)                     "always", category=category, message=message,
1676: (16)                     module=module_regex)
1677: (16)                   self._tmp_modules.add(module)
1678: (16)                   self._clear_registries()

```

```

1676: (12)
1677: (16)
record))
1678: (8)
1679: (12)
1680: (16)
record))
1681: (8)
1682: (4)
1683: (8)
1684: (8)
1685: (8)
1686: (8)
1687: (8)
1688: (12)
1689: (8)
1690: (12)
1691: (8)
1692: (12)
1693: (12)
1694: (12)
1695: (8)
1696: (8)
1697: (8)
1698: (8)
1699: (8)
1700: (8)
1701: (21)
1702: (4)
1703: (8)
1704: (8)
1705: (8)
1706: (8)
1707: (8)
1708: (8)
1709: (12)
1710: (8)
1711: (12)
1712: (8)
1713: (12)
1714: (12)
1715: (12)
1716: (8)
1717: (8)
1718: (8)
1719: (12)
1720: (8)
1721: (8)
1722: (8)
1723: (8)
1724: (8)
1725: (8)
1726: (28)
1727: (4)
1728: (8)
1729: (12)
1730: (8)
1731: (8)
1732: (8)
1733: (8)
1734: (8)
1735: (8)
1736: (8)
1737: (8)
1738: (8)
1739: (12)
1740: (16)
1741: (12)
1742: (16)

        self._tmp_suppressions.append(
            (category, message, re.compile(message, re.I), module,
             record))
        else:
            self._suppressions.append(
                (category, message, re.compile(message, re.I), module,
                 record))
    return record
def filter(self, category=Warning, message="", module=None):
    """
    Add a new suppressing filter or apply it if the state is entered.

    Parameters
    -----
    category : class, optional
        Warning class to filter
    message : string, optional
        Regular expression matching the warning message.
    module : module, optional
        Module to filter for. Note that the module (and its file)
        must match exactly and cannot be a submodule. This may make
        it unreliable for external modules.

    Notes
    -----
    When added within a context, filters are only added inside
    the context and will be forgotten when the context is exited.
    """
    self._filter(category=category, message=message, module=module,
                 record=False)
def record(self, category=Warning, message="", module=None):
    """
    Append a new recording filter or apply it if the state is entered.
    All warnings matching will be appended to the ``log`` attribute.

    Parameters
    -----
    category : class, optional
        Warning class to filter
    message : string, optional
        Regular expression matching the warning message.
    module : module, optional
        Module to filter for. Note that the module (and its file)
        must match exactly and cannot be a submodule. This may make
        it unreliable for external modules.

    Returns
    -----
    log : list
        A list which will be filled with all matched warnings.

    Notes
    -----
    When added within a context, filters are only added inside
    the context and will be forgotten when the context is exited.
    """
    return self._filter(category=category, message=message, module=module,
                        record=True)
def __enter__(self):
    if self._entered:
        raise RuntimeError("cannot enter suppress_warnings twice.")
    self._orig_show = warnings.showwarning
    self._filters = warnings.filters
    warnings.filters = self._filters[:]
    self._entered = True
    self._tmp_suppressions = []
    self._tmp_modules = set()
    self._forwarded = set()
    self.log = [] # reset global log (no need to keep same list)
    for cat, mess, _, mod, log in self._suppressions:
        if log is not None:
            del log[:] # clear the log
        if mod is None:
            warnings.filterwarnings(

```

```

1743: (20)                     "always", category=cat, message=mess)
1744: (12)             else:
1745: (16)                 module_regex = mod.__name__.replace('.', r'\.') + '$'
1746: (16)                 warnings.filterwarnings(
1747: (20)                     "always", category=cat, message=mess,
1748: (20)                         module=module_regex)
1749: (16)                     self._tmp_modules.add(mod)
1750: (8)             warnings.showwarning = self._showwarning
1751: (8)             self._clear_registries()
1752: (8)             return self
1753: (4)         def __exit__(self, *exc_info):
1754: (8)             warnings.showwarning = self._orig_show
1755: (8)             warnings.filters = self._filters
1756: (8)             self._clear_registries()
1757: (8)             self._entered = False
1758: (8)             del self._orig_show
1759: (8)             del self._filters
1760: (4)         def _showwarning(self, message, category, filename, lineno,
1761: (21)                         *args, use_warnmsg=None, **kwargs):
1762: (8)             for cat, _, pattern, mod, rec in (
1763: (16)                 self._suppressions + self._tmp_suppressions)[::-1]:
1764: (12)                 if (issubclass(category, cat) and
1765: (20)                     pattern.match(message.args[0]) is not None):
1766: (16)                     if mod is None:
1767: (20)                         if rec is not None:
1768: (24)                             msg = WarningMessage(message, category, filename,
1769: (45)                                 lineno, **kwargs)
1770: (24)                             self.log.append(msg)
1771: (24)                             rec.append(msg)
1772: (20)                         return
1773: (16)                     elif mod.__file__.startswith(filename):
1774: (20)                         if rec is not None:
1775: (24)                             msg = WarningMessage(message, category, filename,
1776: (45)                                 lineno, **kwargs)
1777: (24)                             self.log.append(msg)
1778: (24)                             rec.append(msg)
1779: (20)                         return
1780: (8)             if self._forwarding_rule == "always":
1781: (12)                 if use_warnmsg is None:
1782: (16)                     self._orig_show(message, category, filename, lineno,
1783: (32)                         *args, **kwargs)
1784: (12)                 else:
1785: (16)                     self._orig_showmsg(use_warnmsg)
1786: (12)                     return
1787: (8)             if self._forwarding_rule == "once":
1788: (12)                 signature = (message.args, category)
1789: (8)             elif self._forwarding_rule == "module":
1790: (12)                 signature = (message.args, category, filename)
1791: (8)             elif self._forwarding_rule == "location":
1792: (12)                 signature = (message.args, category, filename, lineno)
1793: (8)             if signature in self._forwarded:
1794: (12)                 return
1795: (8)             self._forwarded.add(signature)
1796: (8)             if use_warnmsg is None:
1797: (12)                 self._orig_show(message, category, filename, lineno, *args,
1798: (28)                         **kwargs)
1799: (8)             else:
1800: (12)                 self._orig_showmsg(use_warnmsg)
1801: (4)         def __call__(self, func):
1802: (8)             """
1803: (8)                 Function decorator to apply certain suppressions to a whole
1804: (8)                 function.
1805: (8)             """
1806: (8)             @wraps(func)
1807: (8)             def new_func(*args, **kwargs):
1808: (12)                 with self:
1809: (16)                     return func(*args, **kwargs)
1810: (8)                 return new_func
1811: (0)             @contextlib.contextmanager

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

1812: (0)
1813: (4)
1814: (4)
1815: (8)
1816: (8)
1817: (4)
1818: (4)
1819: (4)
1820: (4)
1821: (8)
1822: (12)
1823: (16)
1824: (8)
1825: (12)
1826: (16)
1827: (16)
1828: (8)
1829: (8)
1830: (8)
1831: (8)
1832: (4)
1833: (8)
1834: (8)
1835: (8)
1836: (4)
1837: (8)
1838: (8)
1839: (12)
1840: (12)
1841: (12)
1842: (16)
1843: (16)
1844: (16)
1845: (16)
1846: (20)
1847: (24)
1848: (24)
1849: (24)
1850: (20)
1851: (16)
1852: (12)
1853: (8)
1854: (0)
1855: (4)
1856: (4)
1857: (4)
1858: (8)
1859: (12)
1860: (4)
1861: (4)
1862: (4)
1863: (4)
1864: (8)
1865: (4)
1866: (8)
1867: (4)
1868: (8)
1869: (4)
1870: (4)
1871: (4)
1872: (4)
1873: (4)
1874: (4)
1875: (8)
1876: (4)
1877: (4)
1878: (4)
1879: (8)
1880: (0)

def _assert_no_gc_cycles_context(name=None):
    __tracebackhide__ = True # Hide traceback for py.test
    if not HAS_REFCOUNT:
        yield
        return
    assert_(gc.isenabled())
    gc.disable()
    gc_debug = gc.get_debug()
    try:
        for i in range(100):
            if gc.collect() == 0:
                break
    else:
        raise RuntimeError(
            "Unable to fully collect garbage - perhaps a __del__ method "
            "is creating more reference cycles?")
    gc.set_debug(gc.DEBUG_SAVEALL)
    yield
    n_objects_in_cycles = gc.collect()
    objects_in_cycles = gc.garbage[:]
finally:
    del gc.garbage[:]
    gc.set_debug(gc_debug)
    gc.enable()
if n_objects_in_cycles:
    name_str = f' when calling {name}' if name is not None else ''
    raise AssertionError(
        "Reference cycles were found{}: {} objects were collected, "
        "of which {} are shown below:{}"
        .format(
            name_str,
            n_objects_in_cycles,
            len(objects_in_cycles),
            ''.join(
                "\n {} object with id={}:\n      {}".format(
                    type(o).__name__,
                    id(o),
                    pprint.pformat(o).replace('\n', '\n      ')
                ) for o in objects_in_cycles
            )
        )
    )
def assert_no_gc_cycles(*args, **kwargs):
    """
    Fail if the given callable produces any reference cycles.
    If called with all arguments omitted, may be used as a context manager:
        with assert_no_gc_cycles():
            do_something()
    .. versionadded:: 1.15.0
    Parameters
    -----
    func : callable
        The callable to test.
    \\\*args : Arguments
        Arguments passed to `func`.
    \\\*\\\*kwargs : Kwargs
        Keyword arguments passed to `func`.
    Returns
    -----
    Nothing. The result is deliberately discarded to ensure that all cycles
    are found.
    """
    if not args:
        return _assert_no_gc_cycles_context()
    func = args[0]
    args = args[1:]
    with _assert_no_gc_cycles_context(name=func.__name__):
        func(*args, **kwargs)
def break_cycles():


```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1881: (4)
1882: (4)
1883: (4)
1884: (5)
1885: (4)
1886: (4)
1887: (4)
1888: (4)
1889: (4)
1890: (8)
1891: (8)
1892: (8)
1893: (8)
1894: (0)
1895: (4)
1896: (4)
1897: (4)
1898: (8)
1899: (8)
1900: (12)
1901: (12)
1902: (16)
1903: (12)
1904: (16)
1905: (12)
1906: (16)
1907: (8)
1908: (4)
1909: (0)
1910: (4)
1911: (4)
1912: (4)
1913: (4)
1914: (4)
1915: (4)
1916: (4)
1917: (8)
1918: (12)
1919: (8)
1920: (12)
1921: (8)
'
1922: (15)
1923: (4)
1924: (8)
1925: (8)
1926: (12)
NPY_AVAILABLE_MEM "
1927: (19)
"
1928: (19)
1929: (12)
1930: (8)
1931: (12)
available'
1932: (4)
1933: (0)
1934: (4)
1935: (4)
1936: (16)
1937: (16)
1938: (16)
1939: (4)
1940: (8)
1941: (4)
1942: (4)
1943: (8)
1944: (4)
1945: (0)

    """
    Break reference cycles by calling gc.collect
    Objects can call other objects' methods (for instance, another object's
        __del__) inside their own __del__. On PyPy, the interpreter only runs
    between calls to gc.collect, so multiple calls are needed to completely
    release all cycles.
    """

    gc.collect()
    if IS_PYPY:
        gc.collect()
        gc.collect()
        gc.collect()
        gc.collect()

def requires_memory(free_bytes):
    """Decorator to skip a test if not enough memory is available"""
    import pytest
    def decorator(func):
        @wraps(func)
        def wrapper(*a, **kw):
            msg = check_free_memory(free_bytes)
            if msg is not None:
                pytest.skip(msg)
            try:
                return func(*a, **kw)
            except MemoryError:
                pytest.xfail("MemoryError raised")
        return wrapper
    return decorator

def check_free_memory(free_bytes):
    """
    Check whether `free_bytes` amount of memory is currently free.
    Returns: None if enough memory available, otherwise error message
    """
    env_var = 'NPY_AVAILABLE_MEM'
    env_value = os.environ.get(env_var)
    if env_value is not None:
        try:
            mem_free = _parse_size(env_value)
        except ValueError as exc:
            raise ValueError(f'Invalid environment variable {env_var}: {exc}')
        msg = (f'{free_bytes/1e9} GB memory required, but environment variable'
               f' NPY_AVAILABLE_MEM={env_value} set')
    else:
        mem_free = _get_mem_available()
        if mem_free is None:
            msg = ("Could not determine available memory; set"
                   "environment variable (e.g. NPY_AVAILABLE_MEM=16GB) to run"
                   "the test.")
        mem_free = -1
    else:
        msg = f'{free_bytes/1e9} GB memory required, but {mem_free/1e9} GB'

    return msg if mem_free < free_bytes else None

def _parse_size(size_str):
    """Convert memory size strings ('12 GB' etc.) to float"""
    suffixes = {'': 1, 'b': 1,
               'k': 1000, 'm': 1000**2, 'g': 1000**3, 't': 1000**4,
               'kb': 1000, 'mb': 1000**2, 'gb': 1000**3, 'tb': 1000**4,
               'kib': 1024, 'mib': 1024**2, 'gib': 1024**3, 'tib': 1024**4}
    size_re = re.compile(r'^\s*(\d+|\d+\.\d+)\s*({0})\s*${}'.format(
        '|'.join(suffixes.keys())))
    m = size_re.match(size_str.lower())
    if not m or m.group(2) not in suffixes:
        raise ValueError(f'value {size_str!r} not a valid size')
    return int(float(m.group(1)) * suffixes[m.group(2)])

def _get_mem_available():

```

```

1946: (4)         """Return available memory in bytes, or None if unknown."""
1947: (4)         try:
1948: (8)             import psutil
1949: (8)             return psutil.virtual_memory().available
1950: (4)         except (ImportError, AttributeError):
1951: (8)             pass
1952: (4)         if sys.platform.startswith('linux'):
1953: (8)             info = {}
1954: (8)             with open('/proc/meminfo') as f:
1955: (12)                 for line in f:
1956: (16)                     p = line.split()
1957: (16)                     info[p[0].strip(':').lower()] = int(p[1]) * 1024
1958: (8)             if 'memavailable' in info:
1959: (12)                 return info['memavailable']
1960: (8)             else:
1961: (12)                 return info['memfree'] + info['cached']
1962: (4)         return None
1963: (0)     def _no_tracing(func):
1964: (4)         """
1965: (4)             Decorator to temporarily turn off tracing for the duration of a test.
1966: (4)             Needed in tests that check refcounting, otherwise the tracing itself
1967: (4)             influences the refcounts
1968: (4)         """
1969: (4)         if not hasattr(sys, 'gettrace'):
1970: (8)             return func
1971: (4)         else:
1972: (8)             @wraps(func)
1973: (8)             def wrapper(*args, **kwargs):
1974: (12)                 original_trace = sys.gettrace()
1975: (12)                 try:
1976: (16)                     sys.settrace(None)
1977: (16)                     return func(*args, **kwargs)
1978: (12)                 finally:
1979: (16)                     sys.settrace(original_trace)
1980: (8)             return wrapper
1981: (0)     def _get_glibc_version():
1982: (4)         try:
1983: (8)             ver = os.confstr('CS_GNU_LIBC_VERSION').rsplit(' ')[1]
1984: (4)         except Exception:
1985: (8)             ver = '0.0'
1986: (4)         return ver
1987: (0)     _glibcver = _get_glibc_version()
1988: (0)     _glibc_older_than = lambda x: (_glibcver != '0.0' and _glibcver < x)

```

-----

## File 334 - \_\_init\_\_.py:

1: (0)

## File 335 - test\_ctypeslib.py:

```

1: (0)         import sys
2: (0)         import sysconfig
3: (0)         import weakref
4: (0)         from pathlib import Path
5: (0)         import pytest
6: (0)         import numpy as np
7: (0)         from numpy.ctypeslib import ndpointer, load_library, as_array
8: (0)         from numpy.testing import assert_, assert_array_equal, assert_raises,
assert_equal
9: (0)         try:
10: (4)             import ctypes
11: (0)         except ImportError:
12: (4)             ctypes = None
13: (0)         else:
14: (4)             cdll = None

```

```

15: (4)             test_cdll = None
16: (4)             if hasattr(sys, 'gettotalrefcount'):
17: (8)                 try:
18: (12)                     cdll = load_library('_multiarray_umath_d',
np.core._multiarray_umath.__file__)
19: (8)                     except OSError:
20: (12)                         pass
21: (8)                     try:
22: (12)                         test_cdll = load_library('_multiarray_tests',
np.core._multiarray_tests.__file__)
23: (8)                         except OSError:
24: (12)                             pass
25: (4)                         if cdll is None:
26: (8)                             cdll = load_library('_multiarray_umath',
np.core._multiarray_umath.__file__)
27: (4)                             if test_cdll is None:
28: (8)                                 test_cdll = load_library('_multiarray_tests',
np.core._multiarray_tests.__file__)
29: (4)                                 c_forward_pointer = test_cdll.forward_pointer
30: (0) @pytest.mark.skipif(ctypes is None,
31: (20)             reason="ctypes not available in this python")
32: (0) @pytest.mark.skipif(sys.platform == 'cygwin',
33: (20)             reason="Known to fail on cygwin")
34: (0) class TestLoadLibrary:
35: (4)     def test_basic(self):
36: (8)         loader_path = np.core._multiarray_umath.__file__
37: (8)         out1 = load_library('_multiarray_umath', loader_path)
38: (8)         out2 = load_library(Path('_multiarray_umath'), loader_path)
39: (8)         out3 = load_library('_multiarray_umath', Path(loader_path))
40: (8)         out4 = load_library(b'_multiarray_umath', loader_path)
41: (8)         assert isinstance(out1, ctypes.CDLL)
42: (8)         assert out1 is out2 is out3 is out4
43: (4)     def test_basic2(self):
44: (8)         try:
45: (12)             so_ext = sysconfig.get_config_var('EXT_SUFFIX')
46: (12)             load_library('_multiarray_umath%'+so_ext,
47: (25)                 np.core._multiarray_umath.__file__)
48: (8)         except ImportError as e:
49: (12)             msg = ("ctypes is not available on this python: skipping the test"
50: (19)                 " (import error was: %s)" % str(e))
51: (12)             print(msg)
52: (0) class TestNpointer:
53: (4)     def test_dtype(self):
54: (8)         dt = np.intc
55: (8)         p = ndpointer(dtype=dt)
56: (8)         assert_(p.from_param(np.array([1], dt)))
57: (8)         dt = '<i4'
58: (8)         p = ndpointer(dtype=dt)
59: (8)         assert_(p.from_param(np.array([1], dt)))
60: (8)         dt = np.dtype('>i4')
61: (8)         p = ndpointer(dtype=dt)
62: (8)         p.from_param(np.array([1], dt))
63: (8)         assert_raises(TypeError, p.from_param,
64: (26)             np.array([1], dt.newbyteorder('swap')))
65: (8)         dtnames = ['x', 'y']
66: (8)         dtformats = [np.intc, np.float64]
67: (8)         dtdescr = {'names': dtnames, 'formats': dtformats}
68: (8)         dt = np.dtype(dtdescr)
69: (8)         p = ndpointer(dtype=dt)
70: (8)         assert_(p.from_param(np.zeros((10,), dt)))
71: (8)         samedt = np.dtype(dtdescr)
72: (8)         p = ndpointer(dtype=samedt)
73: (8)         assert_(p.from_param(np.zeros((10,), dt)))
74: (8)         dt2 = np.dtype(dtdescr, align=True)
75: (8)         if dt.itemsize != dt2.itemsize:
76: (12)             assert_raises(TypeError, p.from_param, np.zeros((10,), dt2))
77: (8)         else:
78: (12)             assert_(p.from_param(np.zeros((10,), dt2)))
79: (4)     def test_ndim(self):

```

```

80: (8)             p = ndpointer(ndim=0)
81: (8)             assert_(p.from_param(np.array(1)))
82: (8)             assert_raises(TypeError, p.from_param, np.array([1]))
83: (8)             p = ndpointer(ndim=1)
84: (8)             assert_raises(TypeError, p.from_param, np.array(1))
85: (8)             assert_(p.from_param(np.array([1])))
86: (8)             p = ndpointer(ndim=2)
87: (8)             assert_(p.from_param(np.array([[1]])))
88: (4)             def test_shape(self):
89: (8)                 p = ndpointer(shape=(1, 2))
90: (8)                 assert_(p.from_param(np.array([[1, 2]])))
91: (8)                 assert_raises(TypeError, p.from_param, np.array([[1], [2]]))
92: (8)                 p = ndpointer(shape=())
93: (8)                 assert_(p.from_param(np.array(1)))
94: (4)             def test_flags(self):
95: (8)                 x = np.array([[1, 2], [3, 4]], order='F')
96: (8)                 p = ndpointer(flags='FORTRAN')
97: (8)                 assert_(p.from_param(x))
98: (8)                 p = ndpointer(flags='CONTIGUOUS')
99: (8)                 assert_raises(TypeError, p.from_param, x)
100: (8)                p = ndpointer(flags=x.flags.num)
101: (8)                assert_(p.from_param(x))
102: (8)                assert_raises(TypeError, p.from_param, np.array([[1, 2], [3, 4]]))
103: (4)             def test_cache(self):
104: (8)                 assert_(ndpointer(dtype=np.float64) is ndpointer(dtype=np.float64))
105: (8)                 assert_(ndpointer(shape=2) is ndpointer(shape=(2,)))
106: (8)                 assert_(ndpointer(shape=2) is not ndpointer(ndim=2))
107: (8)                 assert_(ndpointer(ndim=2) is not ndpointer(shape=2))
108: (0)             @pytest.mark.skipif(ctypes is None,
109: (20)                         reason="ctypes not available on this python installation")
110: (0)             class TestNdpointerCFunc:
111: (4)                 def test_arguments(self):
112: (8)                     """ Test that arguments are coerced from arrays """
113: (8)                     c_forward_pointer.restype = ctypes.c_void_p
114: (8)                     c_forward_pointer.argtypes = (ndpointer(ndim=2),)
115: (8)                     c_forward_pointer(np.zeros((2, 3)))
116: (8)                     assert_raises(
117: (12)                         ctypes.ArgumentError, c_forward_pointer, np.zeros((2, 3, 4)))
118: (4)             @pytest.mark.parametrize(
119: (8)                 'dt', [
120: (12)                     float,
121: (12)                     np.dtype(dict(
122: (16)                         formats=['<i4', '<i4'],
123: (16)                         names=['a', 'b'],
124: (16)                         offsets=[0, 2],
125: (16)                         itemsize=6
126: (12)                     )))
127: (8)                 ], ids=[
128: (12)                     'float',
129: (12)                     'overlapping-fields'
130: (8)                 ])
131: (4)             )
132: (4)             def test_return(self, dt):
133: (8)                 """ Test that return values are coerced to arrays """
134: (8)                 arr = np.zeros((2, 3), dt)
135: (8)                 ptr_type = ndpointer(shape=arr.shape, dtype=arr.dtype)
136: (8)                 c_forward_pointer.restype = ptr_type
137: (8)                 c_forward_pointer.argtypes = (ptr_type,)
138: (8)                 arr2 = c_forward_pointer(arr)
139: (8)                 assert_equal(arr2.dtype, arr.dtype)
140: (8)                 assert_equal(arr2.shape, arr.shape)
141: (8)                 assert_equal(
142: (12)                     arr2.__array_interface__['data'],
143: (12)                     arr.__array_interface__['data']
144: (8)                 )
145: (4)             def test_vague_return_value(self):
146: (8)                 """ Test that vague ndpointer return values do not promote to arrays
"""
147: (8)                 arr = np.zeros((2, 3))

```

```

148: (8)             ptr_type = npointer(dtype=arr.dtype)
149: (8)             c_forward_pointer.restype = ptr_type
150: (8)             c_forward_pointer.argtypes = (ptr_type,)
151: (8)             ret = c_forward_pointer(arr)
152: (8)             assert_(isinstance(ret, ptr_type))
153: (0)             @pytest.mark.skipif(ctypes is None,
154: (20)                         reason="ctypes not available on this python installation")
155: (0)             class TestAsArray:
156: (4)                 def test_array(self):
157: (8)                     from ctypes import c_int
158: (8)                     pair_t = c_int * 2
159: (8)                     a = as_array(pair_t(1, 2))
160: (8)                     assert_equal(a.shape, (2,))
161: (8)                     assert_array_equal(a, np.array([1, 2]))
162: (8)                     a = as_array((pair_t * 3)(pair_t(1, 2), pair_t(3, 4), pair_t(5, 6)))
163: (8)                     assert_equal(a.shape, (3, 2))
164: (8)                     assert_array_equal(a, np.array([[1, 2], [3, 4], [5, 6]]))
165: (4)                 def test_pointer(self):
166: (8)                     from ctypes import c_int, cast, POINTER
167: (8)                     p = cast((c_int * 10)(*range(10)), POINTER(c_int))
168: (8)                     a = as_array(p, shape=(10,))
169: (8)                     assert_equal(a.shape, (10,))
170: (8)                     assert_array_equal(a, np.arange(10))
171: (8)                     a = as_array(p, shape=(2, 5))
172: (8)                     assert_equal(a.shape, (2, 5))
173: (8)                     assert_array_equal(a, np.arange(10).reshape((2, 5)))
174: (8)                     assert_raises(TypeError, as_array, p)
175: (4)             @pytest.mark.skipif(
176: (8)                 sys.version_info == (3, 12, 0, "candidate", 1),
177: (8)                 reason="Broken in 3.12.0rc1, see gh-24399",
178: (4)             )
179: (4)             def test_struct_array_pointer(self):
180: (8)                 from ctypes import c_int16, Structure, pointer
181: (8)                 class Struct(Structure):
182: (12)                     _fields_ = [('a', c_int16)]
183: (8)                     Struct3 = 3 * Struct
184: (8)                     c_array = (2 * Struct3)(
185: (12)                         Struct3(Struct(a=1), Struct(a=2), Struct(a=3)),
186: (12)                         Struct3(Struct(a=4), Struct(a=5), Struct(a=6)))
187: (8)
188: (8)                     expected = np.array([
189: (12)                         [(1,), (2,), (3,)],
190: (12)                         [(4,), (5,), (6,)],
191: (8)                         ], dtype=[('a', np.int16)])
192: (8)                     def check(x):
193: (12)                         assert_equal(x.dtype, expected.dtype)
194: (12)                         assert_equal(x, expected)
195: (8)                         check(as_array(c_array))
196: (8)                         check(as_array(pointer(c_array), shape=()))
197: (8)                         check(as_array(pointer(c_array[0]), shape=(2,)))
198: (8)                         check(as_array(pointer(c_array[0][0]), shape=(2, 3)))
199: (4)             def test_reference_cycles(self):
200: (8)                 import ctypes
201: (8)                 N = 100
202: (8)                 a = np.arange(N, dtype=np.short)
203: (8)                 pnt = np.ctypeslib.as_ctypes(a)
204: (8)                 with np.testing.assert_no_gc_cycles():
205: (12)                     newpnt = ctypes.cast(pnt, ctypes.POINTER(ctypes.c_short))
206: (12)                     b = np.ctypeslib.as_array(newpnt, (N,))
207: (12)                     del newpnt, b
208: (4)             def test_segmentation_fault(self):
209: (8)                 arr = np.zeros((224, 224, 3))
210: (8)                 c_arr = np.ctypeslib.as_ctypes(arr)
211: (8)                 arr_ref = weakref.ref(arr)
212: (8)                 del arr
213: (8)                 assert_(arr_ref() is not None)
214: (8)                 c_arr[0][0][0]
215: (0)             @pytest.mark.skipif(ctypes is None,
216: (20)                         reason="ctypes not available on this python installation")

```

```

217: (0)
218: (4)
219: (4)
220: (8)
221: (8)
222: (8)
223: (8)
224: (8)
225: (8)
226: (8)
227: (8)
228: (8)
229: (4)
230: (8)
231: (8)
232: (8)
233: (4)
234: (8)
235: (12)
236: (12)
237: (8)
238: (8)
239: (8)
240: (8)
241: (8)
242: (12)
243: (12)
244: (8)
245: (4)
246: (8)
247: (12)
248: (12)
249: (8)
250: (8)
251: (8)
252: (8)
253: (8)
254: (12)
255: (12)
256: (12)
257: (8)
258: (4)
259: (8)
260: (12)
261: (12)
262: (12)
263: (8)
264: (8)
265: (8)
266: (8)
267: (8)
268: (12)
269: (12)
270: (8)
271: (4)
272: (8)
273: (12)
274: (12)
275: (12)
276: (12)
277: (8)
278: (8)
279: (8)
280: (8)
281: (8)
282: (12)
283: (12)
284: (12)
285: (8)

    class TestAsCtypesType:
        """ Test conversion from dtypes to ctypes types """
        def test_scalar(self):
            dt = np.dtype('<u2')
            ct = np.ctypeslib.as_ctypes_type(dt)
            assert_equal(ct, ctypes.c_uint16._ctype_le__)
            dt = np.dtype('>u2')
            ct = np.ctypeslib.as_ctypes_type(dt)
            assert_equal(ct, ctypes.c_uint16._ctype_be__)
            dt = np.dtype('u2')
            ct = np.ctypeslib.as_ctypes_type(dt)
            assert_equal(ct, ctypes.c_uint16)
        def test_subarray(self):
            dt = np.dtype((np.int32, (2, 3)))
            ct = np.ctypeslib.as_ctypes_type(dt)
            assert_equal(ct, 2 * (3 * ctypes.c_int32))
        def test_structure(self):
            dt = np.dtype([
                ('a', np.uint16),
                ('b', np.uint32),
            ])
            ct = np.ctypeslib.as_ctypes_type(dt)
            assert_(issubclass(ct, ctypes.Structure))
            assert_equal(ctypes.sizeof(ct), dt.itemsize)
            assert_equal(ct._fields_, [
                ('a', ctypes.c_uint16),
                ('b', ctypes.c_uint32),
            ])
        def test_structure_aligned(self):
            dt = np.dtype([
                ('a', np.uint16),
                ('b', np.uint32),
            ], align=True)
            ct = np.ctypeslib.as_ctypes_type(dt)
            assert_(issubclass(ct, ctypes.Structure))
            assert_equal(ctypes.sizeof(ct), dt.itemsize)
            assert_equal(ct._fields_, [
                ('a', ctypes.c_uint16),
                ('', ctypes.c_char * 2), # padding
                ('b', ctypes.c_uint32),
            ])
        def test_union(self):
            dt = np.dtype(dict(
                names=['a', 'b'],
                offsets=[0, 0],
                formats=[np.uint16, np.uint32]
            ))
            ct = np.ctypeslib.as_ctypes_type(dt)
            assert_(issubclass(ct, ctypes.Union))
            assert_equal(ctypes.sizeof(ct), dt.itemsize)
            assert_equal(ct._fields_, [
                ('a', ctypes.c_uint16),
                ('b', ctypes.c_uint32),
            ])
        def test_padded_union(self):
            dt = np.dtype(dict(
                names=['a', 'b'],
                offsets=[0, 0],
                formats=[np.uint16, np.uint32],
                itemsize=5,
            ))
            ct = np.ctypeslib.as_ctypes_type(dt)
            assert_(issubclass(ct, ctypes.Union))
            assert_equal(ctypes.sizeof(ct), dt.itemsize)
            assert_equal(ct._fields_, [
                ('a', ctypes.c_uint16),
                ('b', ctypes.c_uint32),
                ('', ctypes.c_char * 5), # padding
            ])

```

```

286: (4)         def test_overlapping(self):
287: (8)             dt = np.dtype(dict(
288: (12)                 names=['a', 'b'],
289: (12)                 offsets=[0, 2],
290: (12)                 formats=[np.uint32, np.uint32]
291: (8)             ))
292: (8)             assert_raises(NotImplementedError, np.ctypeslib.as_ctypes_type, dt)

```

---

File 336 - test\_lazyloading.py:

```

1: (0)         import sys
2: (0)         import importlib
3: (0)         from importlib.util import LazyLoader, find_spec, module_from_spec
4: (0)         import pytest
5: (0)         @pytest.mark.filterwarnings("ignore:The NumPy module was reloaded")
6: (0)         def test_lazy_load():
7: (4)             old_numpy = sys.modules.pop("numpy")
8: (4)             numpy_modules = {}
9: (4)             for mod_name, mod in list(sys.modules.items()):
10: (8)                 if mod_name[:6] == "numpy.":
11: (12)                     numpy_modules[mod_name] = mod
12: (12)                     sys.modules.pop(mod_name)
13: (4)             try:
14: (8)                 spec = find_spec("numpy")
15: (8)                 module = module_from_spec(spec)
16: (8)                 sys.modules["numpy"] = module
17: (8)                 loader = LazyLoader(spec.loader)
18: (8)                 loader.exec_module(module)
19: (8)                 np = module
20: (8)                 from numpy.lib import recfunctions
21: (8)                 np.ndarray
22: (4)             finally:
23: (8)                 if old_numpy:
24: (12)                     sys.modules["numpy"] = old_numpy
25: (12)                     sys.modules.update(numpy_modules)

```

---

File 337 - test\_matlib.py:

```

1: (0)         import numpy as np
2: (0)         import numpy.matlib
3: (0)         from numpy.testing import assert_array_equal, assert_
4: (0)         def test_empty():
5: (4)             x = numpy.matlib.empty((2,))
6: (4)             assert_(isinstance(x, np.matrix))
7: (4)             assert_(x.shape, (1, 2))
8: (0)         def test_ones():
9: (4)             assert_array_equal(numpy.matlib.ones((2, 3)),
10: (23)                         np.matrix([[ 1.,  1.,  1.],
11: (33)                                         [ 1.,  1.,  1.]]))
12: (4)             assert_array_equal(numpy.matlib.ones(2), np.matrix([[ 1.,  1.]]))
13: (0)         def test_zeros():
14: (4)             assert_array_equal(numpy.matlib.zeros((2, 3)),
15: (23)                         np.matrix([[ 0.,  0.,  0.],
16: (33)                                         [ 0.,  0.,  0.]]))
17: (4)             assert_array_equal(numpy.matlib.zeros(2), np.matrix([[ 0.,  0.]]))
18: (0)         def test_identity():
19: (4)             x = numpy.matlib.identity(2, dtype=int)
20: (4)             assert_array_equal(x, np.matrix([[1, 0], [0, 1]]))
21: (0)         def test_eye():
22: (4)             xc = numpy.matlib.eye(3, k=1, dtype=int)
23: (4)             assert_array_equal(xc, np.matrix([[ 0,  1,  0],
24: (38)                                         [ 0,  0,  1],
25: (38)                                         [ 0,  0,  0]]))
26: (4)             assert xc.flags.c_contiguous
27: (4)             assert not xc.flags.f_contiguous

```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

28: (4)         xf = numpy.matlib.eye(3, 4, dtype=int, order='F')
29: (4)         assert_array_equal(xf, np.matrix([[ 1,  0,  0,  0],
30: (38)                           [ 0,  1,  0,  0],
31: (38)                           [ 0,  0,  1,  0]]))
32: (4)         assert not xf.flags.c_contiguous
33: (4)         assert xf.flags.f_contiguous
34: (0)     def test_rand():
35: (4)         x = numpy.matlib.rand(3)
36: (4)         assert_(x.ndim == 2)
37: (0)     def test_rndn():
38: (4)         x = np.matlib.randn(3)
39: (4)         assert_(x.ndim == 2)
40: (0)     def test_repmat():
41: (4)         a1 = np.arange(4)
42: (4)         x = numpy.matlib.repmat(a1, 2, 2)
43: (4)         y = np.array([[0, 1, 2, 3, 0, 1, 2, 3],
44: (18)             [0, 1, 2, 3, 0, 1, 2, 3]])
45: (4)         assert_array_equal(x, y)

-----
```

## File 338 - test\_numpy\_config.py:

```
1: (0) """
2: (0)     Check the numpy config is valid.
3: (0) """
4: (0)     import numpy as np
5: (0)     import pytest
6: (0)     from unittest.mock import Mock, patch
7: (0)     pytestmark = pytest.mark.skipif(
8: (4)         not hasattr(np.__config__, "_built_with_meson"),
9: (4)         reason="Requires Meson builds",
10: (0)     )
11: (0)     class TestNumPyConfigs:
12: (4)         REQUIRED_CONFIG_KEYS = [
13: (8)             "Compilers",
14: (8)             "Machine Information",
15: (8)             "Python Information",
16: (4)         ]
17: (4)         @patch("numpy.__config__.check_pyyaml")
18: (4)         def test_pyyaml_not_found(self, mock_yaml_importer):
19: (8)             mock_yaml_importer.side_effect = ModuleNotFoundError()
20: (8)             with pytest.warns(UserWarning):
21: (12)                 np.show_config()
22: (4)         def test_dict_mode(self):
23: (8)             config = np.show_config(mode="dicts")
24: (8)             assert isinstance(config, dict)
25: (8)             assert all([key in config for key in self.REQUIRED_CONFIG_KEYS]), (
26: (12)                 "Required key missing,"
27: (12)                 " see index of `False` with `REQUIRED_CONFIG_KEYS`"
28: (8)             )
29: (4)         def test_invalid_mode(self):
30: (8)             with pytest.raises(AttributeError):
31: (12)                 np.show_config(mode="foo")
32: (4)         def test_warn_to_add_tests(self):
33: (8)             assert len(np.__config__.DisplayModes) == 2, (
34: (12)                 "New mode detected,"
35: (12)                 " please add UT if applicable and increment this count"
36: (8)             )

-----
```

## File 339 - test\_numpy\_version.py:

```
1: (0) """
2: (0)     Check the numpy version is valid.
3: (0)     Note that a development version is marked by the presence of 'dev0' or '+'
4: (0)     in the version string, all else is treated as a release. The version string
5: (0)     itself is set from the output of ``git describe`` which relies on tags.
```

```

6: (0)          Examples
7: (0)          -----
8: (0)          Valid Development: 1.22.0.dev0 1.22.0.dev0+5-g7999db4df2 1.22.0+5-g7999db4df2
9: (0)          Valid Release: 1.21.0.rc1, 1.21.0.b1, 1.21.0
10: (0)         Invalid: 1.22.0.dev, 1.22.0.dev0-5-g7999db4dfB, 1.21.0.d1, 1.21.a
11: (0)         Note that a release is determined by the version string, which in turn
12: (0)         is controlled by the result of the ``git describe`` command.
13: (0)         """
14: (0)         import re
15: (0)         import numpy as np
16: (0)         from numpy.testing import assert_
17: (0)         def test_valid_numpy_version():
18: (4)             version_pattern = r"^[0-9]+\.[0-9]+\.[0-9]+(a[0-9]|b[0-9]|rc[0-9])?"
19: (4)             dev_suffix = r"(\.dev[0-9]+(\+git[0-9]+\.[0-9a-f]+)?)?"
20: (4)             res = re.match(version_pattern + dev_suffix + '$', np.__version__)
21: (4)             assert_(res is not None, np.__version__)
22: (0)         def test_short_version():
23: (4)             if np.version.release:
24: (8)                 assert_(np.__version__ == np.version.short_version,
25: (16)                     "short_version mismatch in release version")
26: (4)             else:
27: (8)                 assert_(np.__version__.split("+")[0] == np.version.short_version,
28: (16)                     "short_version mismatch in development version")

```

-----

## File 340 - test\_public\_api.py:

```

1: (0)         import sys
2: (0)         import sysconfig
3: (0)         import subprocess
4: (0)         import pkgutil
5: (0)         import types
6: (0)         import importlib
7: (0)         import warnings
8: (0)         import numpy as np
9: (0)         import numpy
10: (0)        import pytest
11: (0)        from numpy.testing import IS_WASM
12: (0)        try:
13: (4)            import ctypes
14: (0)        except ImportError:
15: (4)            ctypes = None
16: (0)        def check_dir(module, module_name=None):
17: (4)            """Returns a mapping of all objects with the wrong __module__
attribute."""
18: (4)            if module_name is None:
19: (8)                module_name = module.__name__
20: (4)            results = {}
21: (4)            for name in dir(module):
22: (8)                item = getattr(module, name)
23: (8)                if (hasattr(item, '__module__') and hasattr(item, '__name__')
24: (16)                    and item.__module__ != module_name):
25: (12)                    results[name] = item.__module__ + '.' + item.__name__
26: (4)            return results
27: (0)        def test_numpy_namespace():
28: (4)            undocumented = {
29: (8)                '_add_newdoc_ufunc': 'numpy.core._multiarray_umath._add_newdoc_ufunc',
30: (8)                'add_docstring': 'numpy.core._multiarray_umath.add_docstring',
31: (8)                'add_newdoc': 'numpy.core.function_base.add_newdoc',
32: (8)                'add_newdoc_ufunc': 'numpy.core._multiarray_umath._add_newdoc_ufunc',
33: (8)                'byte_bounds': 'numpy.lib.utils.byte_bounds',
34: (8)                'compare_chararrays':
'numpy.core._multiarray_umath.compare_chararrays',
35: (8)                'deprecate': 'numpy.lib.utils.deprecate',
36: (8)                'deprecate_with_doc': 'numpy.lib.utils.deprecate_with_doc',
37: (8)                'disp': 'numpy.lib.function_base.disp',
38: (8)                'fastCopyAndTranspose':
'numpy.core._multiarray_umath.fastCopyAndTranspose',

```

```

39: (8)             'get_array_wrap': 'numpy.lib.shape_base.get_array_wrap',
40: (8)             'get_include': 'numpy.lib.utils.get_include',
41: (8)             'recfromcsv': 'numpy.lib.npyio.recfromcsv',
42: (8)             'recfromtxt': 'numpy.lib.npyio.recfromtxt',
43: (8)             'safe_eval': 'numpy.lib.utils.safe_eval',
44: (8)             'set_string_function': 'numpy.core.arrayprint.set_string_function',
45: (8)             'show_config': 'numpy.__config__.show',
46: (8)             'show_runtime': 'numpy.lib.utils.show_runtime',
47: (8)             'who': 'numpy.lib.utils.who',
48: (4)
49: (4)         }
50: (4)     allowlist = undocumented
51: (4)     bad_results = check_dir(np)
52: (0)     assert bad_results == allowlist
53: (0) @pytest.mark.skipif(IS_WASM, reason="can't start subprocess")
54: (0) @pytest.mark.parametrize('name', ['testing'])
55: (0) def test_import_lazy_import(name):
56: (4)     """Make sure we can actually use the modules we lazy load.
57: (4)     While not exported as part of the public API, it was accessible. With the
58: (4)     use of __getattr__ and __dir__, this isn't always true It can happen that
59: (4)     an infinite recursion may happen.
60: (4)     This is the only way I found that would force the failure to appear on the
61: (4)     badly implemented code.
62: (4)     We also test for the presence of the lazily imported modules in dir
63: (4)     """
64: (4)     exe = (sys.executable, '-c', "import numpy; numpy." + name)
65: (4)     result = subprocess.check_output(exe)
66: (4)     assert not result
67: (4)     assert name in dir(np)
68: (0) def test_dir_testing():
69: (4)     """Assert that output of dir has only one "testing/tester"
70: (4)     attribute without duplicate"""
71: (0)     assert len(dir(np)) == len(set(dir(np)))
72: (4) def test_numpy_linalg():
73: (4)     bad_results = check_dir(np.linalg)
74: (0)     assert bad_results == {}
75: (4) def test_numpy_fft():
76: (4)     bad_results = check_dir(np.fft)
77: (0)     assert bad_results == {}
78: (20) @pytest.mark.skipif(ctypes is None,
79: (0)                     reason="ctypes not available in this python")
80: (0) def test_NPY_NO_EXPORT():
81: (4)     cdll = ctypes.CDLL(np.core._multiarray_tests.__file__)
82: (4)     f = getattr(cdll, 'test_not_exported', None)
83: (22)     assert f is None, ("'test_not_exported' is mistakenly exported, "
84: (0)                         "NPY_NO_EXPORT does not work")
85: (4) PUBLIC_MODULES = ['numpy.' + s for s in [
86: (4)             "array_api",
87: (4)             "array_api.linalg",
88: (4)             "ctypeslib",
89: (4)             "doc",
90: (4)             "doc.constants",
91: (4)             "doc.ufuncs",
92: (4)             "dtypes",
93: (4)             "exceptions",
94: (4)             "f2py",
95: (4)             "fft",
96: (4)             "lib",
97: (4)             "lib.format", # was this meant to be public?
98: (4)             "lib.mixins",
99: (4)             "lib.recfunctions",
100: (4)            "lib.scimath",
101: (4)            "lib.stride_tricks",
102: (4)            "linalg",
103: (4)            "ma",
104: (4)            "ma.extras",
105: (4)            "ma.mrecords",
106: (4)            "matlib",
107: (4)            "polynomial",
107: (4)            "polynomial.chebyshev",

```

```
108: (4)          "polynomial.hermite",
109: (4)          "polynomial.hermite_e",
110: (4)          "polynomial.laguerre",
111: (4)          "polynomial.legendre",
112: (4)          "polynomial.polynomial",
113: (4)          "random",
114: (4)          "testing",
115: (4)          "testing.overrides",
116: (4)          "typing",
117: (4)          "typing.mypy_plugin",
118: (4)          "version" # Should be removed for NumPy 2.0
119: (0)
120: (0)
121: (4)      ]]
122: (8)      if sys.version_info < (3, 12):
123: (12)          PUBLIC_MODULES += [
124: (12)              'numpy.' + s for s in [
125: (12)                  "distutils",
126: (12)                  "distutils.cpuinfo",
127: (12)                  "distutils.exec_command",
128: (12)                  "distutils.misc_util",
129: (8)                      "distutils.log",
130: (4)                      "distutils.system_info",
131: (0)                  ]
132: (4)          PUBLIC_ALIASED_MODULES = [
133: (4)              "numpy.char",
134: (4)              "numpy.emath",
135: (0)              "numpy.rec",
136: (0)          PRIVATE_BUT_PRESENT_MODULES = ['numpy.' + s for s in [
137: (4)              "compat",
138: (4)              "compat.py3k",
139: (4)              "conftest",
140: (4)              "core",
141: (4)              "core.arrayprint",
142: (4)              "core.defchararray",
143: (4)              "core.einsumfunc",
144: (4)              "core.fromnumeric",
145: (4)              "core.function_base",
146: (4)              "core.getlimits",
147: (4)              "core.memmap",
148: (4)              "core.multiarray",
149: (4)              "core.numeric",
150: (4)              "core.numerictypes",
151: (4)              "core.overrides",
152: (4)              "core.records",
153: (4)              "core.shape_base",
154: (4)              "core.umath",
155: (4)              "f2py.auxfuncs",
156: (4)              "f2py.capi_maps",
157: (4)              "f2py.cb_rules",
158: (4)              "f2py.cfuncs",
159: (4)              "f2py.common_rules",
160: (4)              "f2py.crackfortran",
161: (4)              "f2py.diagnose",
162: (4)              "f2py.f2py2e",
163: (4)              "f2py.f90mod_rules",
164: (4)              "f2py.func2subr",
165: (4)              "f2py.rules",
166: (4)              "f2py.symbolic",
167: (4)              "f2py.use_rules",
168: (4)              "fft.helper",
169: (4)              "lib.arraypad",
170: (4)              "lib.arraysetops",
171: (4)              "lib.arrayterator",
172: (4)              "lib.function_base",
173: (4)              "lib.histograms",
174: (4)              "lib.index_tricks",
175: (4)              "lib.nanfunctions",
176: (4)              "lib.npyio",
```

```

177: (4)           "lib.polynomial",
178: (4)           "lib.shape_base",
179: (4)           "lib.twodim_base",
180: (4)           "lib.type_check",
181: (4)           "lib.ufunclike",
182: (4)           "lib.user_array", # note: not in np.lib, but probably should just be
deleted
183: (4)           "lib.utils",
184: (4)           "linalg.lapack_lite",
185: (4)           "linalg.linalg",
186: (4)           "ma.core",
187: (4)           "ma.testutils",
188: (4)           "ma.timer_comparison",
189: (4)           "matrixlib",
190: (4)           "matrixlib.defmatrix",
191: (4)           "polynomial.polyutils",
192: (4)           "random.mtrand",
193: (4)           "random.bit_generator",
194: (4)           "testing.print_coercion_tables",
195: (0)
196: (0) if sys.version_info < (3, 12):
197: (4)     PRIVATE_BUT_PRESENT_MODULES += [
198: (8)         'numpy.' + s for s in [
199: (12)             "distutils.armccompiler",
200: (12)             "distutils.fujitsucompiler",
201: (12)             "distutils.ccompiler",
202: (12)             "distutils.ccompiler_opt",
203: (12)             "distutils.command",
204: (12)             "distutils.command.autodist",
205: (12)             "distutils.command.bdist_rpm",
206: (12)             "distutils.command.build",
207: (12)             "distutils.command.build_clib",
208: (12)             "distutils.command.build_ext",
209: (12)             "distutils.command.build_py",
210: (12)             "distutils.command.build_scripts",
211: (12)             "distutils.command.build_src",
212: (12)             "distutils.command.config",
213: (12)             "distutils.command.config_compiler",
214: (12)             "distutils.command.develop",
215: (12)             "distutils.command.egg_info",
216: (12)             "distutils.command.install",
217: (12)             "distutils.command.install_clib",
218: (12)             "distutils.command.install_data",
219: (12)             "distutils.command.install_headers",
220: (12)             "distutils.command.sdist",
221: (12)             "distutils.conv_template",
222: (12)             "distutils.core",
223: (12)             "distutils.extension",
224: (12)             "distutils.fcompiler",
225: (12)             "distutils.fcompiler.abssoft",
226: (12)             "distutils.fcompiler.arm",
227: (12)             "distutils.fcompiler.compaq",
228: (12)             "distutils.fcompiler.environment",
229: (12)             "distutils.fcompiler.g95",
230: (12)             "distutils.fcompiler.gnu",
231: (12)             "distutils.fcompiler.hpx",
232: (12)             "distutils.fcompiler.ibm",
233: (12)             "distutils.fcompiler.intel",
234: (12)             "distutils.fcompiler.lahey",
235: (12)             "distutils.fcompiler.mips",
236: (12)             "distutils.fcompiler.nag",
237: (12)             "distutils.fcompiler.none",
238: (12)             "distutils.fcompiler.pathf95",
239: (12)             "distutils.fcompiler.pg",
240: (12)             "distutils.fcompiler.nv",
241: (12)             "distutils.fcompiler.sun",
242: (12)             "distutils.fcompiler.vast",
243: (12)             "distutils.fcompiler.fujitsu",
244: (12)             "distutils.from_template",

```

```

245: (12)                     "distutils.intelccompiler",
246: (12)                     "distutils.lib2def",
247: (12)                     "distutils.line_endings",
248: (12)                     "distutils.mingw32ccompiler",
249: (12)                     "distutils.msvccompiler",
250: (12)                     "distutils.npy_pkg_config",
251: (12)                     "distutils.numpy_distribution",
252: (12)                     "distutils.pathccompiler",
253: (12)                     "distutils.unixccompiler",
254: (8)                      ]
255: (4)                      ]
256: (0)  def is_unexpected(name):
257: (4)      """Check if this needs to be considered."""
258: (4)      if '.' in name or '.tests' in name or '.setup' in name:
259: (8)          return False
260: (4)      if name in PUBLIC_MODULES:
261: (8)          return False
262: (4)      if name in PUBLIC_ALIASED_MODULES:
263: (8)          return False
264: (4)      if name in PRIVATE_BUT_PRESENT_MODULES:
265: (8)          return False
266: (4)      return True
267: (0)  SKIP_LIST = [
268: (4)      "numpy.core.code_generators",
269: (4)      "numpy.core.code_generators.genapi",
270: (4)      "numpy.core.code_generators.generate_umath",
271: (4)      "numpy.core.code_generators.ufunc_docstrings",
272: (4)      "numpy.core.code_generators.generate_numpy_api",
273: (4)      "numpy.core.code_generators.generate_ufunc_api",
274: (4)      "numpy.core.code_generators.numpy_api",
275: (4)      "numpy.core.code_generators.generate_umath_doc",
276: (4)      "numpy.core.code_generators.verify_c_api_version",
277: (4)      "numpy.core.cversions",
278: (4)      "numpy.core.generate_numpy_api",
279: (4)      "numpy.core.umath_tests",
280: (0)  ]
281: (0)  if sys.version_info < (3, 12):
282: (4)      SKIP_LIST += ["numpy.distutils.msvc9compiler"]
283: (0)  @pytest.mark.filterwarnings("ignore:.*np.compat.*:DeprecationWarning")
284: (0)  def test_all_modules_are_expected():
285: (4)      """
286: (4)          Test that we don't add anything that looks like a new public module by
287: (4)          accident. Check is based on filenames.
288: (4)      """
289: (4)      modnames = []
290: (4)      for _, modname, ispkg in pkgutil.walk_packages(path=np.__path__,
291: (51)                                  prefix=np.__name__ + '.',
292: (51)                                  onerror=None):
293: (8)          if is_unexpected(modname) and modname not in SKIP_LIST:
294: (12)              modnames.append(modname)
295: (4)      if modnames:
296: (8)          raise AssertionError(f'Found unexpected modules: {modnames}')
297: (0)  SKIP_LIST_2 = [
298: (4)      'numpy.math',
299: (4)      'numpy.doc.constants.re',
300: (4)      'numpy.doc.constants.textwrap',
301: (4)      'numpy.lib.emath',
302: (4)      'numpy.lib.math',
303: (4)      'numpy.matlib.char',
304: (4)      'numpy.matlib.rec',
305: (4)      'numpy.matlib.emath',
306: (4)      'numpy.matlib.exceptions',
307: (4)      'numpy.matlib.math',
308: (4)      'numpy.matlib.linalg',
309: (4)      'numpy.matlib.fft',
310: (4)      'numpy.matlib.random',
311: (4)      'numpy.matlib.ctypeslib',
312: (4)      'numpy.matlib.ma',
313: (0)      ]

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

314: (0)
315: (4)
316: (8)
317: (8)
318: (8)
319: (4)
320: (0)
321: (4)
322: (4)
323: (4)
324: (4)
325: (8)
326: (4)
if
327: (4)
328: (4)
329: (4)
330: (8)
331: (8)
332: (12)
333: (16)
334: (12)
335: (16)
336: (12)
337: (16)
338: (4)
339: (4)
340: (8)
341: (8)
342: (8)
343: (12)
344: (8)
345: (12)
346: (8)
347: (12)
348: (16)
349: (16)
350: (20)
351: (24)
352: (28)
353: (8)
354: (4)
355: (4)
356: (8)
357: (4)
358: (8)
359: (29)
360: (0)
361: (4)
362: (4)
363: (4)
364: (4)
365: (4)
366: (4)
367: (4)
368: (8)
369: (12)
370: (8)
371: (12)
372: (8)
373: (4)
374: (4)
375: (8)
376: (12)
377: (4)
378: (8)
379: (29)
380: (4)
381: (8)

    if sys.version_info < (3, 12):
        SKIP_LIST_2 += [
            'numpy.distutils.log.sys',
            'numpy.distutils.log.logging',
            'numpy.distutils.log.warnings',
        ]
def test_all_modules_are_expected_2():
    """
        Method checking all objects. The pkgutil-based method in
        `test_all_modules_are_expected` does not catch imports into a namespace,
        only filenames. So this test is more thorough, and checks this like:
            import .lib.scimath as emath
        To check if something in a module is (effectively) public, one can check
        there's anything in that namespace that's a public function/object but is
        not exposed in a higher-level namespace. For example for a `numpy.lib`
        submodule::
            mod = np.lib.mixins
            for obj in mod.__all__:
                if obj in np.__all__:
                    continue
                elif obj in np.lib.__all__:
                    continue
                else:
                    print(obj)
    """
def find_unexpected_members(mod_name):
    members = []
    module = importlib.import_module(mod_name)
    if hasattr(module, '__all__'):
        objnames = module.__all__
    else:
        objnames = dir(module)
    for objname in objnames:
        if not objname.startswith('_'):
            fullobjname = mod_name + '.' + objname
            if isinstance(getattr(module, objname), types.ModuleType):
                if is_unexpected(fullobjname):
                    if fullobjname not in SKIP_LIST_2:
                        members.append(fullobjname)
    return members
unexpected_members = find_unexpected_members("numpy")
for modname in PUBLIC_MODULES:
    unexpected_members.extend(find_unexpected_members(modname))
if unexpected_members:
    raise AssertionError("Found unexpected object(s) that look like "
                         "modules: {}".format(unexpected_members))
def test_api_importable():
    """
        Check that all submodules listed higher up in this file can be imported
        Note that if a PRIVATE_BUT_PRESENT_MODULES entry goes missing, it may
        simply need to be removed from the list (deprecation may or may not be
        needed - apply common sense).
    """
    def check_importable(module_name):
        try:
            importlib.import_module(module_name)
        except (ImportError, AttributeError):
            return False
        return True
    module_names = []
    for module_name in PUBLIC_MODULES:
        if not check_importable(module_name):
            module_names.append(module_name)
    if module_names:
        raise AssertionError("Modules in the public API that cannot be "
                            "imported: {}".format(module_names))
    for module_name in PUBLIC_ALIASED_MODULES:
        try:

```

```

382: (12)             eval(module_name)
383: (8)              except AttributeError:
384: (12)                  module_names.append(module_name)
385: (4)  if module_names:
386: (8)      raise AssertionError("Modules in the public API that were not "
387: (29)                      "found: {}".format(module_names))
388: (4)  with warnings.catch_warnings(record=True) as w:
389: (8)      warnings.filterwarnings('always', category=DeprecationWarning)
390: (8)      warnings.filterwarnings('always', category=ImportWarning)
391: (8)      for module_name in PRIVATE_BUT_PRESENT_MODULES:
392: (12)          if not check_importable(module_name):
393: (16)              module_names.append(module_name)
394: (4)  if module_names:
395: (8)      raise AssertionError("Modules that are not really public but looked "
396: (29)                      "public and can not be imported: "
397: (29)                      "{}".format(module_names))
398: (0) @pytest.mark.xfail(
399: (4)     sysconfig.get_config_var("Py_DEBUG") not in (None, 0, "0"),
400: (4)     reason=(
401: (8)         "NumPy possibly built with `USE_DEBUG=True ./tools/travis-test.sh`, "
402: (8)         "which does not expose the `array_api` entry point. "
403: (8)         "See https://github.com/numpy/numpy/pull/19800"
404: (4)     ),
405: (0)
406: (0) def test_array_api_entry_point():
407: (4) """
408: (4)     Entry point for Array API implementation can be found with importlib and
409: (4)     returns the numpy.array_api namespace.
410: (4) """
411: (4)     numpy_in_sitepackages = sysconfig.get_path('platlib') in np.__file__
412: (4)     eps = importlib.metadata.entry_points()
413: (4)     try:
414: (8)         xp_eps = eps.select(group="array_api")
415: (4)     except AttributeError:
416: (8)         xp_eps = eps.get("array_api", [])
417: (4)     if len(xp_eps) == 0:
418: (8)         if numpy_in_sitepackages:
419: (12)             msg = "No entry points for 'array_api' found"
420: (12)             raise AssertionError(msg) from None
421: (8)         return
422: (4)     try:
423: (8)         ep = next(ep for ep in xp_eps if ep.name == "numpy")
424: (4)     except StopIteration:
425: (8)         if numpy_in_sitepackages:
426: (12)             msg = "'numpy' not in array_api entry points"
427: (12)             raise AssertionError(msg) from None
428: (8)         return
429: (4)     xp = ep.load()
430: (4)     msg = (
431: (8)         f"numpy entry point value '{ep.value}' "
432: (8)         "does not point to our Array API implementation"
433: (4)     )
434: (4)     assert xp is numpy.array_api, msg
435: (0) @pytest.mark.parametrize("name", [
436: (8)     'ModuleDeprecationWarning', 'VisibleDeprecationWarning',
437: (8)     'ComplexWarning', 'TooHardError', 'AxisError'])
438: (0) def test_moved_exceptions(name):
439: (4)     assert name in np.__all__
440: (4)     assert name not in np.__dir__()
441: (4)     assert getattr(np, name).__module__ == "numpy.exceptions"
442: (4)     assert name in np.exceptions.__all__
443: (4)     setattr(np.exceptions, name)

```

-----  
File 341 - test\_reloading.py:

```

1: (0)         from numpy.testing import (
2: (4)             assert_raises,

```

```

3: (4)             assert_warnings,
4: (4)             assert_,
5: (4)             assert_equal,
6: (4)             IS_WASM,
7: (0)
8: (0)         from numpy.compat import pickle
9: (0)         import pytest
10: (0)        import sys
11: (0)        import subprocess
12: (0)        import textwrap
13: (0)        from importlib import reload
14: (0)    def test_numpy_reloading():
15: (4)        import numpy as np
16: (4)        import numpy._globals
17: (4)        _NoValue = np._NoValue
18: (4)        VisibleDeprecationWarning = np.VisibleDeprecationWarning
19: (4)        ModuleDeprecationWarning = np.ModuleDeprecationWarning
20: (4)        with assert_warnings(UserWarning):
21: (8)            reload(np)
22: (4)            assert_(np._NoValue is np._NoValue)
23: (4)            assert_(ModuleDeprecationWarning is np.ModuleDeprecationWarning)
24: (4)            assert_(VisibleDeprecationWarning is np.VisibleDeprecationWarning)
25: (4)            assert_raises(RuntimeError, reload, numpy._globals)
26: (4)            with assert_warnings(UserWarning):
27: (8)                reload(np)
28: (4)                assert_(np._NoValue is np._NoValue)
29: (4)                assert_(ModuleDeprecationWarning is np.ModuleDeprecationWarning)
30: (4)                assert_(VisibleDeprecationWarning is np.VisibleDeprecationWarning)
31: (0)    def test_novalue():
32: (4)        import numpy as np
33: (4)        for proto in range(2, pickle.HIGHEST_PROTOCOL + 1):
34: (8)            assert_equal(repr(np._NoValue), '<no value>')
35: (8)            assert_(pickle.loads(pickle.dumps(np._NoValue,
36: (42)                                         protocol=proto)) is np._NoValue)
37: (0)    @pytest.mark.skipif(IS_WASM, reason="can't start subprocess")
38: (0)    def test_full_reimport():
39: (4)        """At the time of writing this, it is *not* truly supported, but
40: (4)        apparently enough users rely on it, for it to be an annoying change
41: (4)        when it started failing previously.
42: (4)
43: (4)        code = textwrap.dedent(r"""
44: (8)            import sys
45: (8)            from pytest import warns
46: (8)            import numpy as np
47: (8)            for k in list(sys.modules.keys()):
48: (12)                if "numpy" in k:
49: (16)                    del sys.modules[k]
50: (8)
51: (12)                with warns(UserWarning):
52: (8)                    import numpy as np
53: (4)                """
54: (4)            p = subprocess.run([sys.executable, '-c', code], capture_output=True)
55: (8)            if p.returncode:
56: (12)                raise AssertionError(
57: (8)                    f"Non-zero return code: {p.returncode}\n\n{p.stderr.decode()}"
58: )

```

---

File 342 - test\_scripts.py:

```

1: (0)        """ Test scripts
2: (0)        Test that we can run executable scripts that have been installed with numpy.
3: (0)        """
4: (0)        import sys
5: (0)        import os
6: (0)        import pytest
7: (0)        from os.path import join as pathjoin, isfile, dirname
8: (0)        import subprocess
9: (0)        import numpy as np

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

10: (0)
11: (0)
12: (0)
13: (4)
14: (8)
15: (8)
16: (12)
17: (8)
18: (12)
19: (4)
20: (8)
21: (8)
22: (8)
23: (8)
24: (0)      from numpy.testing import assert_equal, IS_WASM
25: (0)      is_inplace = isfile(pathjoin(dirname(np.__file__), '..', 'setup.py'))
26: (0)      def find_f2py_commands():
27: (4)          if sys.platform == 'win32':
28: (8)              exe_dir = dirname(sys.executable)
29: (8)              if exe_dir.endswith('Scripts'): # virtualenv
30: (12)                  return [os.path.join(exe_dir, 'f2py')]
31: (8)              else:
32: (12)                  return [os.path.join(exe_dir, "Scripts", 'f2py')]
33: (4)          else:
34: (8)              version = sys.version_info
35: (8)              major = str(version.major)
36: (8)              minor = str(version.minor)
37: (8)              return ['f2py', 'f2py' + major, 'f2py' + major + '.' + minor]
38: (0)      @pytest.mark.skipif(is_inplace, reason="Cannot test f2py command inplace")
39: (0)      @pytest.mark.xfail(reason="Test is unreliable")
40: (0)      @pytest.mark.parametrize('f2py_cmd', find_f2py_commands())
41: (0)      def test_f2py(f2py_cmd):
42: (4)          stdout = subprocess.check_output([f2py_cmd, '-v'])
43: (4)          assert_equal(stdout.strip(), np.__version__.encode('ascii'))
44: (0)      @pytest.mark.skipif(IS_WASM, reason="Cannot start subprocess")
45: (0)      def test_pep338():
46: (4)          stdout = subprocess.check_output([sys.executable, '-mnumpy.f2py', '-v'])
47: (4)          assert_equal(stdout.strip(), np.__version__.encode('ascii'))

```

-----  
File 343 - test\_warnings.py:

```

1: (0)          """
2: (0)          Tests which scan for certain occurrences in the code, they may not find
3: (0)          all of these occurrences but should catch almost all.
4: (0)          """
5: (0)          import pytest
6: (0)          from pathlib import Path
7: (0)          import ast
8: (0)          import tokenize
9: (0)          import numpy
10: (0)         class ParseCall(ast.NodeVisitor):
11: (4)             def __init__(self):
12: (8)                 self.ls = []
13: (4)             def visit_Attribute(self, node):
14: (8)                 ast.NodeVisitor.generic_visit(self, node)
15: (8)                 self.ls.append(node.attr)
16: (4)             def visit_Name(self, node):
17: (8)                 self.ls.append(node.id)
18: (0)         class FindFuncs(ast.NodeVisitor):
19: (4)             def __init__(self, filename):
20: (8)                 super().__init__()
21: (8)                 self.__filename = filename
22: (4)             def visit_Call(self, node):
23: (8)                 p = ParseCall()
24: (8)                 p.visit(node.func)
25: (8)                 ast.NodeVisitor.generic_visit(self, node)
26: (8)                 if p.ls[-1] == 'simplefilter' or p.ls[-1] == 'filterwarnings':
27: (12)                     if node.args[0].value == "ignore":
28: (16)                         raise AssertionError(
29: (20)                             "warnings should have an appropriate stacklevel; found in"
30: (20)                             "{} on line {}".format(self.__filename, node.lineno))
31: (8)                 if p.ls[-1] == 'warn' and (
32: (16)                     len(p.ls) == 1 or p.ls[-2] == 'warnings'):
33: (12)                     if "testing/tests/test_warnings.py" == self.__filename:
34: (16)                         return
35: (12)                     if len(node.args) == 3:
36: (16)                         return
37: (12)                     args = {kw.arg for kw in node.keywords}
38: (12)                     if "stacklevel" in args:
39: (16)                         return

```

```

40: (12)             raise AssertionError(
41: (16)                 "warnings should have an appropriate stacklevel; found in "
42: (16)                 "{} on line {}".format(self._filename, node.lineno))
43: (0) @pytest.mark.slow
44: (0) def test_warning_calls():
45: (4)     base = Path(numpy.__file__).parent
46: (4)     for path in base.rglob("*.py"):
47: (8)         if base / "testing" in path.parents:
48: (12)             continue
49: (8)         if path == base / "__init__.py":
50: (12)             continue
51: (8)         if path == base / "random" / "__init__.py":
52: (12)             continue
53: (8)         with tokenize.open(str(path)) as file:
54: (12)             tree = ast.parse(file.read())
55: (12)             FindFuncs(path).visit(tree)
-----
```

File 344 - test\_all\_.py:

```

1: (0)         import collections
2: (0)         import numpy as np
3: (0) def test_no_duplicates_in_np_all_():
4: (4)     dups = {k: v for k, v in collections.Counter(np.__all__).items() if v > 1}
5: (4)     assert len(dups) == 0
-----
```

File 345 - \_\_init\_\_.py:

```
1: (0)
```

-----

File 346 - mypy\_plugin.py:

```

1: (0)         """A mypy_ plugin for managing a number of platform-specific annotations.
2: (0)         Its functionality can be split into three distinct parts:
3: (0)         * Assigning the (platform-dependent) precisions of certain `~numpy.number`_
4: (2)             subclasses, including the likes of `~numpy.int_`, `~numpy.intp` and
5: (2)             `~numpy.longlong`. See the documentation on
6: (2)             :ref:`scalar types <arrays.scalars.built-in>` for a comprehensive overview
7: (2)             of the affected classes. Without the plugin the precision of all relevant
8: (2)             classes will be inferred as `~typing.Any`_.
9: (0)         * Removing all extended-precision `~numpy.number`_ subclasses that are
10: (2)             unavailable for the platform in question. Most notably this includes the
11: (2)             likes of `~numpy.float128` and `~numpy.complex256`. Without the plugin *all*
12: (2)             extended-precision types will, as far as mypy is concerned, be available
13: (2)             to all platforms.
14: (0)         * Assigning the (platform-dependent) precision of `~numpy.ctypeslib.c_intp`_.
15: (2)             Without the plugin the type will default to `ctypes.c_int64`_.
16: (2)             .. versionadded:: 1.22
17: (0) Examples
18: (0) -----
19: (0) To enable the plugin, one must add it to their mypy `configuration file`_:
20: (0) .. code-block:: ini
21: (4)     [mypy]
22: (4)     plugins = numpy.typing.mypy_plugin
23: (0) .. _mypy: http://mypy-lang.org/
24: (0) .. _configuration file: https://mypy.readthedocs.io/en/stable/config_file.html
25: (0) """
26: (0)     from __future__ import annotations
27: (0)     from collections.abc import Iterable
28: (0)     from typing import Final, TYPE_CHECKING, Callable
29: (0)     import numpy as np
30: (0)     try:
31: (4)         import mypy.types
32: (4)         from mypy.types import Type
-----
```

```

33: (4)             from mypy.plugin import Plugin, AnalyzeTypeContext
34: (4)             from mypy.nodes import MypyFile, ImportFrom, Statement
35: (4)             from mypy.build import PRI_MED
36: (4)             _HookFunc = Callable[[AnalyzeTypeContext], Type]
37: (4)             MYPY_EX: None | ModuleNotFoundError = None
38: (0)         except ModuleNotFoundError as ex:
39: (4)             MYPY_EX = ex
40: (0)         __all__: list[str] = []
41: (0)     def _get_precision_dict() -> dict[str, str]:
42: (4)         names = [
43: (8)             ("_NBitByte", np.byte),
44: (8)             ("_NBitShort", np.short),
45: (8)             ("_NBitIntC", np.intc),
46: (8)             ("_NBitIntP", np.intp),
47: (8)             ("_NBitInt", np.int_),
48: (8)             ("_NBitLongLong", np.longlong),
49: (8)             ("_NBitHalf", np.half),
50: (8)             ("_NBitSingle", np.single),
51: (8)             ("_NBitDouble", np.double),
52: (8)             ("_NBitLongDouble", np.longdouble),
53: (4)         ]
54: (4)         ret = {}
55: (4)         for name, typ in names:
56: (8)             n: int = 8 * typ().dtype.itemsize
57: (8)             ret[f"numpy._typing._nbit.{name}"] = f"numpy._{n}Bit"
58: (4)     return ret
59: (0) def _get_extended_precision_list() -> list[str]:
60: (4)     extended_names = [
61: (8)         "uint128",
62: (8)         "uint256",
63: (8)         "int128",
64: (8)         "int256",
65: (8)         "float80",
66: (8)         "float96",
67: (8)         "float128",
68: (8)         "float256",
69: (8)         "complex160",
70: (8)         "complex192",
71: (8)         "complex256",
72: (8)         "complex512",
73: (4)     ]
74: (4)     return [i for i in extended_names if hasattr(np, i)]
75: (0) def _get_c_intp_name() -> str:
76: (4)     char = np.dtype('p').char
77: (4)     if char == 'i':
78: (8)         return "c_int"
79: (4)     elif char == 'l':
80: (8)         return "c_long"
81: (4)     elif char == 'q':
82: (8)         return "c_longlong"
83: (4)     else:
84: (8)         return "c_long"
85: (0) _PRECISION_DICT: Final = _get_precision_dict()
86: (0) _EXTENDED_PRECISION_LIST: Final = _get_extended_precision_list()
87: (0) _C_INTP: Final = _get_c_intp_name()
88: (0) def _hook(ctx: AnalyzeTypeContext) -> Type:
89: (4)     """Replace a type-alias with a concrete ``NBitBase`` subclass."""
90: (4)     typ, _, api = ctx
91: (4)     name = typ.name.split(".")[-1]
92: (4)     name_new = _PRECISION_DICT[f"numpy._typing._nbit.{name}"]
93: (4)     return api.named_type(name_new)
94: (0)     if TYPE_CHECKING or MYPY_EX is None:
95: (4)         def _index(iterable: Iterable[Statement], id: str) -> int:
96: (8)             """Identify the first ``ImportFrom`` instance the specified `id`."""
97: (8)             for i, value in enumerate(iterable):
98: (12)                 if getattr(value, "id", None) == id:
99: (16)                     return i
100: (8)             raise ValueError("Failed to identify a `ImportFrom` instance "
101: (25)                               f"with the following id: {id!r}")

```

```

102: (4)             def _override_imports(
103: (8)                 file: MypyFile,
104: (8)                 module: str,
105: (8)                 imports: list[tuple[str, None | str]],
106: (4)             ) -> None:
107: (8)                 """Override the first `module`-based import with new `imports`."""
108: (8)                 import_obj = ImportFrom(module, 0, names=imports)
109: (8)                 import_obj.is_top_level = True
110: (8)                 for lst in [file.defs, file.imports]: # type: list[Statement]
111: (12)                     i = _index(lst, module)
112: (12)                     lst[i] = import_obj
113: (4)             class _NumpyPlugin(Plugin):
114: (8)                 """A mypy plugin for handling versus numpy-specific typing tasks."""
115: (8)                 def get_type_analyze_hook(self, fullname: str) -> None | _HookFunc:
116: (12)                     """Set the precision of platform-specific `numpy.number` subclasses.
117: (12)                     For example: `numpy.int_`, `numpy.longlong` and
118: (12)
119: (12)                     `numpy.longdouble`.
120: (12)
121: (16)
122: (12)
123: (8)
124: (12)
125: (8)
126: (12)
127: (12)
128: (14)
129: (14)
130: (12)
131: (12)
132: (12)
133: (12)
134: (16)
135: (20)
136: (20)
137: (16)
138: (12)
139: (16)
140: (20)
141: (20)
142: (16)
143: (12)
144: (4)             return ret
145: (8)
146: (8)
147: (0)
148: (4)
149: (8)
150: (8)

-----
```

File 347 - setup.py:

```

1: (0)             def configuration(parent_package='', top_path=None):
2: (4)                 from numpy.distutils.misc_util import Configuration
3: (4)                 config = Configuration('typing', parent_package, top_path)
4: (4)                 config.add_subpackage('tests')
5: (4)                 config.add_data_dir('tests/data')
6: (4)                 return config
7: (0)             if __name__ == '__main__':
8: (4)                 from numpy.distutils.core import setup
9: (4)                 setup(configuration=configuration)

-----
```

File 348 - \_\_init\_\_.py:

```

1: (0)
2: (0)
3: (0)
4: (0)
5: (0)
6: (0)
7: (0)
8: (0)
9: (0)
10: (0)
11: (0)
12: (0)
13: (0)
14: (0)
15: (0)
16: (0)
17: (0)
18: (0)
19: (0)
20: (0)
21: (0)
22: (0)
23: (0)
24: (0)
25: (0)
26: (0)
27: (0)
28: (0)
29: (4)
30: (4)
31: (0)
32: (0)
33: (0)
34: (0)
35: (0)
36: (4)
37: (0)
38: (0)
39: (4)
40: (4)
41: (4)
42: (4)
43: (0)
44: (0)
45: (0)
46: (0)
47: (0)
48: (4)
49: (4)
50: (0)
51: (0)
52: (0)
53: (0)
54: (0)
55: (0)
56: (0)
57: (0)
58: (4)
59: (0)
60: (0)
61: (0)
62: (0)
63: (0)
64: (0)
65: (0)
66: (0)
67: (0)
68: (4)
69: (4)

```

"""
=====

Typing (:mod:`numpy.typing`)

=====
.. versionadded:: 1.20

Large parts of the NumPy API have :pep:`484`-style type annotations. In addition a number of type aliases are available to users, most prominently the two below:

- `ArrayLike`: objects that can be converted to arrays
- `DTypeLike`: objects that can be converted to dtypes

.. \_typing-extensions: <https://pypi.org/project/typing-extensions/>

Mypy plugin

-----
.. versionadded:: 1.21

.. automodule:: numpy.typing.mypy\_plugin

.. currentmodule:: numpy.typing

Differences from the runtime NumPy API

-----
NumPy is very flexible. Trying to describe the full range of possibilities statically would result in types that are not very helpful. For that reason, the typed NumPy API is often stricter than the runtime NumPy API. This section describes some notable differences.

ArrayLike

~~~~~

The `ArrayLike` type tries to avoid creating object arrays. For example,

.. code-block:: python

```
>>> np.array(x**2 for x in range(10))
array(<generator object <genexpr> at ...>, dtype=object)
```

is valid NumPy code which will create a 0-dimensional object array. Type checkers will complain about the above example when using the NumPy types however. If you really intended to do the above, then you can either use a ``# type: ignore`` comment:

.. code-block:: python

```
>>> np.array(x**2 for x in range(10)) # type: ignore
```

or explicitly type the array like object as `~typing.Any`:

.. code-block:: python

```
>>> from typing import Any
>>> array_like: Any = (x**2 for x in range(10))
>>> np.array(array_like)
array(<generator object <genexpr> at ...>, dtype=object)
```

ndarray

~~~~~

It's possible to mutate the dtype of an array at runtime. For example, the following code is valid:

.. code-block:: python

```
>>> x = np.array([1, 2])
>>> x.dtype = np.bool_
```

This sort of mutation is not allowed by the types. Users who want to write statically typed code should instead use the `numpy.ndarray.view` method to create a view of the array with a different dtype.

DTypeLike

~~~~~

The `DTypeLike` type tries to avoid creation of dtype objects using dictionary of fields like below:

.. code-block:: python

```
>>> x = np.dtype({"field1": (float, 1), "field2": (int, 3)})
```

Although this is valid NumPy code, the type checker will complain about it, since its usage is discouraged.

Please see :ref:`Data type objects <arrays.dtypes>`

Number precision

~~~~~

The precision of `numpy.number` subclasses is treated as a covariant generic parameter (see :class:`~NBitBase`), simplifying the annotating of processes involving precision-based casting.

.. code-block:: python

```
>>> from typing import TypeVar
>>> import numpy as np
```

```

70: (4)          >>> import numpy.typing as npt
71: (4)          >>> T = TypeVar("T", bound=npt.NBitBase)
72: (4)          >>> def func(a: "np.floating[T]", b: "np.floating[T]") ->
73: (4)              ...
74: (0)      ... Consequently, the likes of `~numpy.float16`, `~numpy.float32` and
75: (0)      `~numpy.float64` are still sub-types of `~numpy.floating`, but, contrary to
76: (0)      runtime, they're not necessarily considered as sub-classes.
77: (0)      Timedelta64
78: (0)      ~~~~~
79: (0)      The `~numpy.timedelta64` class is not considered a subclass of
80: (0)      `~numpy.signedinteger`, the former only inheriting from `~numpy.generic`
81: (0)      while static type checking.
82: (0)      0D arrays
83: (0)      ~~~~~
84: (0)      During runtime numpy aggressively casts any passed 0D arrays into their
85: (0)      corresponding `~numpy.generic` instance. Until the introduction of shape
86: (0)      typing (see :pep:`646`) it is unfortunately not possible to make the
87: (0)      necessary distinction between 0D and >0D arrays. While thus not strictly
88: (0)      correct, all operations are that can potentially perform a 0D-array -> scalar
89: (0)      cast are currently annotated as exclusively returning an `ndarray`.
90: (0)      If it is known in advance that an operation _will_ perform a
91: (0)      0D-array -> scalar cast, then one can consider manually remedying the
92: (0)      situation with either `typing.cast` or a ``# type: ignore`` comment.
93: (0)      Record array dtypes
94: (0)      ~~~~~
95: (0)      The dtype of `numpy.recarray`, and the `numpy.rec` functions in general,
96: (0)      can be specified in one of two ways:
97: (0)          * Directly via the ``dtype`` argument.
98: (0)          * With up to five helper arguments that operate via `numpy.format_parser`:
99: (2)              ``formats``, ``names``, ``titles``, ``aligned`` and ``byteorder``.
100: (0)      These two approaches are currently typed as being mutually exclusive,
101: (0)          *i.e.* if ``dtype`` is specified than one may not specify ``formats``.
102: (0)      While this mutual exclusivity is not (strictly) enforced during runtime,
103: (0)      combining both dtype specifiers can lead to unexpected or even downright
104: (0)      buggy behavior.
105: (0)      API
106: (0)      ---
107: (0)      """
108: (0)      from numpy._typing import (
109: (4)          ArrayLike,
110: (4)          DtypeLike,
111: (4)          NBitBase,
112: (4)          NDArray,
113: (0)
114: (0)      __all__ = ["ArrayLike", "DtypeLike", "NBitBase", "NDArray"]
115: (0)      if __doc__ is not None:
116: (4)          from numpy._typing._add_docstring import _docstrings
117: (4)          __doc__ += _docstrings
118: (4)          __doc__ += '\n.. autoclass:: numpy.typing.NBitBase\n'
119: (4)          del _docstrings
120: (0)      from numpy._pytesttester import PytestTester
121: (0)      test = PytestTester(__name__)
122: (0)      del PytestTester

```

---

File 349 - test\_isfile.py:

```

1: (0)          import os
2: (0)          import sys
3: (0)          from pathlib import Path
4: (0)          import numpy as np
5: (0)          from numpy.testing import assert_
6: (0)          ROOT = Path(np.__file__).parents[0]
7: (0)          FILES = [
8: (4)              ROOT / "py.typed",
9: (4)              ROOT / "__init__.pyi",
10: (4)              ROOT / "ctypeslib.pyi",

```

```

11: (4)             ROOT / "core" / "__init__.pyi",
12: (4)             ROOT / "f2py" / "__init__.pyi",
13: (4)             ROOT / "fft" / "__init__.pyi",
14: (4)             ROOT / "lib" / "__init__.pyi",
15: (4)             ROOT / "linalg" / "__init__.pyi",
16: (4)             ROOT / "ma" / "__init__.pyi",
17: (4)             ROOT / "matrixlib" / "__init__.pyi",
18: (4)             ROOT / "polynomial" / "__init__.pyi",
19: (4)             ROOT / "random" / "__init__.pyi",
20: (4)             ROOT / "testing" / "__init__.pyi",
21: (0)
22: (0)         ]
23: (4)     if sys.version_info < (3, 12):
24: (0)         FILES += [ROOT / "distutils" / "__init__.pyi"]
25: (4)     class TestIsFile:
26: (8)         def test_isfile(self):
27: (8)             """Test if all ``.pyi`` files are properly installed."""
28: (8)             for file in FILES:
29: (12)                 assert_(os.path.isfile(file))

```

-----  
File 350 - test\_typing.py:

```

1: (0)             from __future__ import annotations
2: (0)             import importlib.util
3: (0)             import os
4: (0)             import re
5: (0)             import shutil
6: (0)             from collections import defaultdict
7: (0)             from collections.abc import Iterator
8: (0)             from typing import TYPE_CHECKING
9: (0)             import pytest
10: (0)            from numpy.typing.mypy_plugin import _EXTENDED_PRECISION_LIST
11: (0)            RUN_MYPY = "NPY_RUN_MYPY_IN_TESTSUITE" in os.environ
12: (0)            if RUN_MYPY and RUN_MYPY not in ('0', '', 'false'):
13: (4)                RUN_MYPY = True
14: (0)            pytestmark = pytest.mark.skipif(
15: (4)                not RUN_MYPY,
16: (4)                reason="`NPY_RUN_MYPY_IN_TESTSUITE` not set"
17: (0)            )
18: (0)            RUN_MYPY = "NPY_RUN_MYPY_IN_TESTSUITE" in os.environ
19: (0)            if RUN_MYPY and RUN_MYPY not in ('0', '', 'false'):
20: (4)                RUN_MYPY = True
21: (0)            pytestmark = pytest.mark.skipif(
22: (4)                not RUN_MYPY,
23: (4)                reason="`NPY_RUN_MYPY_IN_TESTSUITE` not set"
24: (0)            )
25: (0)            try:
26: (4)                from mypy import api
27: (0)            except ImportError:
28: (4)                NO_MYPY = True
29: (0)            else:
30: (4)                NO_MYPY = False
31: (0)            if TYPE_CHECKING:
32: (4)                from _pytest.mark.structures import ParameterSet
33: (0)                DATA_DIR = os.path.join(os.path.dirname(__file__), "data")
34: (0)                PASS_DIR = os.path.join(DATA_DIR, "pass")
35: (0)                FAIL_DIR = os.path.join(DATA_DIR, "fail")
36: (0)                REVEAL_DIR = os.path.join(DATA_DIR, "reveal")
37: (0)                MISC_DIR = os.path.join(DATA_DIR, "misc")
38: (0)                MYPY_INI = os.path.join(DATA_DIR, "mypy.ini")
39: (0)                CACHE_DIR = os.path.join(DATA_DIR, ".mypy_cache")
40: (0)                OUTPUT_MYPY: defaultdict[str, list[str]] = defaultdict(list)
41: (0)                def _key_func(key: str) -> str:
42: (4)                    """Split at the first occurrence of the ``:`` character.
43: (4)                    Windows drive-letters (*e.g.* ``C:``) are ignored herein.
44: (4)                    """
45: (4)                    drive, tail = os.path.splitdrive(key)
46: (4)                    return os.path.join(drive, tail.split(":", 1)[0])

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

47: (0) def _strip_filename(msg: str) -> tuple[int, str]:
48: (4)     """Strip the filename and line number from a mypy message."""
49: (4)     _, tail = os.path.splitdrive(msg)
50: (4)     _, lineno, msg = tail.split(":", 2)
51: (4)     return int(lineno), msg.strip()
52: (0) def strip_func(match: re.Match[str]) -> str:
53: (4)     """`re.sub` helper function for stripping module names."""
54: (4)     return match.groups()[1]
55: (0) @pytest.fixture(scope="module", autouse=True)
56: (0) def run_mypy() -> None:
57: (4)     """Clears the cache and run mypy before running any of the typing tests.
58: (4)     The mypy results are cached in `OUTPUT_MYPY` for further use.
59: (4)     The cache refresh can be skipped using
60: (4)     NUMPY_TYPING_TEST_CLEAR_CACHE=0 pytest numpy/typing/tests
61: (4)     """
62: (4)     if (
63: (8)         os.path.isdir(CACHE_DIR)
64: (8)         and bool(os.environ.get("NUMPY_TYPING_TEST_CLEAR_CACHE", True))
65: (4)     ):
66: (8)         shutil.rmtree(CACHE_DIR)
67: (4)     split_pattern = re.compile(r"(\s+)?\^(\~+)?")
68: (4)     for directory in (PASS_DIR, REVEAL_DIR, FAIL_DIR, MISC_DIR):
69: (8)         stdout, stderr, exit_code = api.run([
70: (12)             "--config-file",
71: (12)             MYPY_INI,
72: (12)             "--cache-dir",
73: (12)             CACHE_DIR,
74: (12)             directory,
75: (8)         ])
76: (8)         if stderr:
77: (12)             pytest.fail(f"Unexpected mypy standard error\n\n{stderr}")
78: (8)         elif exit_code not in {0, 1}:
79: (12)             pytest.fail(f"Unexpected mypy exit code: {exit_code}\n\n{stdout}")
80: (8)         str_concat = ""
81: (8)         filename: str | None = None
82: (8)         for i in stdout.split("\n"):
83: (12)             if "note:" in i:
84: (16)                 continue
85: (12)             if filename is None:
86: (16)                 filename = _key_func(i)
87: (12)             str_concat += f"{i}\n"
88: (12)             if split_pattern.match(i) is not None:
89: (16)                 OUTPUT_MYPY[filename].append(str_concat)
90: (16)                 str_concat = ""
91: (16)                 filename = None
92: (0)     def get_test_cases(directory: str) -> Iterator[ParameterSet]:
93: (4)         for root, _, files in os.walk(directory):
94: (8)             for fname in files:
95: (12)                 short_fname, ext = os.path.splitext(fname)
96: (12)                 if ext in (".pyi", ".py"):
97: (16)                     fullpath = os.path.join(root, fname)
98: (16)                     yield pytest.param(fullpath, id=short_fname)
99: (0)     @pytest.mark.slow
100: (0)     @pytest.mark.skipif(NO_MYPY, reason="Mypy is not installed")
101: (0)     @pytest.mark.parametrize("path", get_test_cases(PASS_DIR))
102: (0)     def test_success(path) -> None:
103: (4)         output_mypy = OUTPUT_MYPY
104: (4)         if path in output_mypy:
105: (8)             msg = "Unexpected mypy output\n\n"
106: (8)             msg += "\n".join(_strip_filename(v)[1] for v in output_mypy[path])
107: (8)             raise AssertionError(msg)
108: (0)         @pytest.mark.slow
109: (0)         @pytest.mark.skipif(NO_MYPY, reason="Mypy is not installed")
110: (0)         @pytest.mark.parametrize("path", get_test_cases(FAIL_DIR))
111: (0)         def test_fail(path: str) -> None:
112: (4)             __tracebackhide__ = True
113: (4)             with open(path) as fin:
114: (8)                 lines = fin.readlines()
115: (4)                 errors = defaultdict(lambda: "")

```

```

116: (4)           output_mypy = OUTPUT_MYPY
117: (4)           assert path in output_mypy
118: (4)           for error_line in output_mypy[path]:
119: (8)             lineno, error_line = _strip_filename(error_line)
120: (8)             errors[lineno] += f'{error_line}\n'
121: (4)             for i, line in enumerate(lines):
122: (8)               lineno = i + 1
123: (8)               if (
124: (12)                 line.startswith('#')
125: (12)                 or (" E:" not in line and lineno not in errors)
126: (8)               ):
127: (12)                 continue
128: (8)             target_line = lines[lineno - 1]
129: (8)             if "# E:" in target_line:
130: (12)               expression, _, marker = target_line.partition(" # E: ")
131: (12)               expected_error = errors[lineno].strip()
132: (12)               marker = marker.strip()
133: (12)               _test_fail(path, expression, marker, expected_error, lineno)
134: (8)             else:
135: (12)               pytest.fail(
136: (16)                 f"Unexpected mypy output at line {lineno}\n\n{errors[lineno]}"
137: (12)               )
138: (0)             _FAIL_MSG1 = """Extra error at line {}
139: (0)             Expression: {}
140: (0)             Extra error: {!r}
141: (0)             """
142: (0)             _FAIL_MSG2 = """Error mismatch at line {}
143: (0)             Expression: {}
144: (0)             Expected error: {}
145: (0)             Observed error: {!r}
146: (0)             """
147: (0)             def _test_fail(
148: (4)               path: str,
149: (4)               expression: str,
150: (4)               error: str,
151: (4)               expected_error: None | str,
152: (4)               lineno: int,
153: (0)             ) -> None:
154: (4)               if expected_error is None:
155: (8)                 raise AssertionError(_FAIL_MSG1.format(lineno, expression, error))
156: (4)               elif error not in expected_error:
157: (8)                 raise AssertionError(_FAIL_MSG2.format(
158: (12)                   lineno, expression, expected_error, error
159: (8)                 ))
160: (0)             _REVEAL_MSG = """Reveal mismatch at line {}
161: (0)             {}
162: (0)             """
163: (0)             @pytest.mark.slow
164: (0)             @pytest.mark.skipif(NO_MYPY, reason="Mypy is not installed")
165: (0)             @pytest.mark.parametrize("path", get_test_cases(REVEAL_DIR))
166: (0)             def test_reveal(path: str) -> None:
167: (4)               """Validate that mypy correctly infers the return-types of
168: (4)               the expressions in `path`.
169: (4)               """
170: (4)               __tracebackhide__ = True
171: (4)               output_mypy = OUTPUT_MYPY
172: (4)               if path not in output_mypy:
173: (8)                 return
174: (4)               for error_line in output_mypy[path]:
175: (8)                 lineno, error_line = _strip_filename(error_line)
176: (8)                 raise AssertionError(_REVEAL_MSG.format(lineno, error_line))
177: (0)             @pytest.mark.slow
178: (0)             @pytest.mark.skipif(NO_MYPY, reason="Mypy is not installed")
179: (0)             @pytest.mark.parametrize("path", get_test_cases(PASS_DIR))
180: (0)             def test_code_runs(path: str) -> None:
181: (4)               """Validate that the code in `path` properly during runtime."""
182: (4)               path_without_extension, _ = os.path.splitext(path)
183: (4)               dirname, filename = path.split(os.sep)[-2:]
184: (4)               spec = importlib.util.spec_from_file_location(

```

```

185: (8)             f"{dirname}.{filename}", path
186: (4)
187: (4)         )
188: (4)         assert spec is not None
189: (4)         assert spec.loader is not None
190: (4)         test_module = importlib.util.module_from_spec(spec)
191: (0)         spec.loader.exec_module(test_module)
192: (4) LINENO_MAPPING = {
193: (4)     11: "uint128",
194: (4)     12: "uint256",
195: (4)     14: "int128",
196: (4)     15: "int256",
197: (4)     17: "float80",
198: (4)     18: "float96",
199: (4)     19: "float128",
200: (4)     20: "float256",
201: (4)     22: "complex160",
202: (4)     23: "complex192",
203: (4)     24: "complex256",
204: (0)     25: "complex512",
205: (0)
206: (0) @pytest.mark.slow
207: (0) @pytest.mark.skipif(NO_MYPY, reason="Mypy is not installed")
208: (0) def test_extended_precision() -> None:
209: (4)     path = os.path.join(MISC_DIR, "extended_precision.pyi")
210: (4)     output_mypy = OUTPUT_MYPY
211: (4)     assert path in output_mypy
212: (8)     with open(path) as f:
213: (4)         expression_list = f.readlines()
214: (8)     for _msg in output_mypy[path]:
215: (8)         lineno, msg = _strip_filename(_msg)
216: (8)         expression = expression_list[lineno - 1].rstrip("\n")
217: (12)         if LINENO_MAPPING[lineno] in _EXTENDED_PRECISION_LIST:
218: (8)             raise AssertionError(_REVEAL_MSG.format(lineno, msg))
219: (12)         elif "error" not in msg:
220: (16)             _test_fail(
221: (12)                 path, expression, msg, 'Expression is of type "Any"', lineno
)
-----
```

## File 351 - test\_runtime.py:

```

1: (0) """Test the runtime usage of `numpy.typing`."""
2: (0) from __future__ import annotations
3: (0) from typing import (
4: (4)     get_type_hints,
5: (4)     Union,
6: (4)     NamedTuple,
7: (4)     get_args,
8: (4)     get_origin,
9: (4)     Any,
10: (0) )
11: (0) import pytest
12: (0) import numpy as np
13: (0) import numpy.typing as npt
14: (0) import numpy._typing as _npt
15: (0) class TypeTup(NamedTuple):
16: (4)     typ: type
17: (4)     args: tuple[type, ...]
18: (4)     origin: None | type
19: (0) NDArrayTup = TypeTup(npt.NDArray, npt.NDArray.__args__, np.ndarray)
20: (0) TYPES = {
21: (4)     "ArrayLike": TypeTup(npt.ArrayLike, npt.ArrayLike.__args__, Union),
22: (4)     "DTypeLike": TypeTup(npt.DTypeLike, npt.DTypeLike.__args__, Union),
23: (4)     "NBitBase": TypeTup(npt.NBitBase, (), None),
24: (4)     "NDArray": NDArrayTup,
25: (0) }
26: (0) @pytest.mark.parametrize("name,tup", TYPES.items(), ids=TYPES.keys())
27: (0) def test_get_args(name: type, tup: TypeTup) -> None:
```

```

28: (4)         """Test `typing.get_args`."""
29: (4)         typ, ref = tup.typ, tup.args
30: (4)         out = get_args(typ)
31: (4)         assert out == ref
32: (0) @pytest.mark.parametrize("name,tup", TYPES.items(), ids=TYPES.keys())
33: (0) def test_get_origin(name: type, tup: TypeTup) -> None:
34: (4)         """Test `typing.get_origin`."""
35: (4)         typ, ref = tup.typ, tup.origin
36: (4)         out = get_origin(typ)
37: (4)         assert out == ref
38: (0) @pytest.mark.parametrize("name,tup", TYPES.items(), ids=TYPES.keys())
39: (0) def test_get_type_hints(name: type, tup: TypeTup) -> None:
40: (4)         """Test `typing.get_type_hints`."""
41: (4)         typ = tup.typ
42: (4)         def func(a): pass
43: (4)         func.__annotations__ = {"a": typ, "return": None}
44: (4)         out = get_type_hints(func)
45: (4)         ref = {"a": typ, "return": type(None)}
46: (4)         assert out == ref
47: (0) @pytest.mark.parametrize("name,tup", TYPES.items(), ids=TYPES.keys())
48: (0) def test_get_type_hints_str(name: type, tup: TypeTup) -> None:
49: (4)         """Test `typing.get_type_hints` with string-representation of types."""
50: (4)         typ_str, typ = f"npt.{name}", tup.typ
51: (4)         def func(a): pass
52: (4)         func.__annotations__ = {"a": typ_str, "return": None}
53: (4)         out = get_type_hints(func)
54: (4)         ref = {"a": typ, "return": type(None)}
55: (4)         assert out == ref
56: (0) def test_keys() -> None:
57: (4)         """Test that ``TYPES.keys()`` and ``numpy.typing._all__`` are synced."""
58: (4)         keys = TYPES.keys()
59: (4)         ref = set(npt._all__)
60: (4)         assert keys == ref
61: (0) PROTOCOLS: dict[str, tuple[type[Any], object]] = {
62: (4)     "_SupportsDType": (_npt._SupportsDType, np.int64(1)),
63: (4)     "_SupportsArray": (_npt._SupportsArray, np.arange(10)),
64: (4)     "_SupportsArrayFunc": (_npt._SupportsArrayFunc, np.arange(10)),
65: (4)     "_NestedSequence": (_npt._NestedSequence, [1]),
66: (0)
67: (0) @pytest.mark.parametrize("cls,obj", PROTOCOLS.values(), ids=PROTOCOLS.keys())
68: (0) class TestRuntimeProtocol:
69: (4)     def test_isinstance(self, cls: type[Any], obj: object) -> None:
70: (8)         assert isinstance(obj, cls)
71: (8)         assert not isinstance(None, cls)
72: (4)     def test_issubclass(self, cls: type[Any], obj: object) -> None:
73: (8)         if cls is _npt._SupportsDType:
74: (12)             pytest.xfail(
75: (16)                 "Protocols with non-method members don't support issubclass()")
76: (12)
77: (8)         assert issubclass(type(obj), cls)
78: (8)         assert not issubclass(type(None), cls)

```

-----  
File 352 - \_\_init\_\_.py:

1: (0)

-----  
File 353 - arithmetic.py:

```

1: (0)         from __future__ import annotations
2: (0)         from typing import Any
3: (0)         import numpy as np
4: (0)         import pytest
5: (0)         c16 = np.complex128(1)
6: (0)         f8 = np.float64(1)
7: (0)         i8 = np.int64(1)

```

```

8: (0) u8 = np.uint64(1)
9: (0) c8 = np.complex64(1)
10: (0) f4 = np.float32(1)
11: (0) i4 = np.int32(1)
12: (0) u4 = np.uint32(1)
13: (0) dt = np.datetime64(1, "D")
14: (0) td = np.timedelta64(1, "D")
15: (0) b_ = np.bool_(1)
16: (0) b = bool(1)
17: (0) c = complex(1)
18: (0) f = float(1)
19: (0) i = int(1)
20: (0) class Object:
21: (4)     def __array__(self) -> np.ndarray[Any, np.dtype[np.object_]]:
22: (8)         ret = np.empty(()), dtype=object)
23: (8)         ret[:] = self
24: (8)         return ret
25: (4)     def __sub__(self, value: Any) -> Object:
26: (8)         return self
27: (4)     def __rsub__(self, value: Any) -> Object:
28: (8)         return self
29: (4)     def __floordiv__(self, value: Any) -> Object:
30: (8)         return self
31: (4)     def __rfloordiv__(self, value: Any) -> Object:
32: (8)         return self
33: (4)     def __mul__(self, value: Any) -> Object:
34: (8)         return self
35: (4)     def __rmul__(self, value: Any) -> Object:
36: (8)         return self
37: (4)     def __pow__(self, value: Any) -> Object:
38: (8)         return self
39: (4)     def __rpow__(self, value: Any) -> Object:
40: (8)         return self
41: (0) AR_b: np.ndarray[Any, np.dtype[np.bool_]] = np.array([True])
42: (0) AR_u: np.ndarray[Any, np.dtype[np.uint32]] = np.array([1], dtype=np.uint32)
43: (0) AR_i: np.ndarray[Any, np.dtype[np.int64]] = np.array([1])
44: (0) AR_f: np.ndarray[Any, np.dtype[np.float64]] = np.array([1.0])
45: (0) AR_c: np.ndarray[Any, np.dtype[np.complex128]] = np.array([1j])
46: (0) AR_m: np.ndarray[Any, np.dtype[np.timedelta64]] = np.array([np.timedelta64(1,
"D")])
47: (0) AR_M: np.ndarray[Any, np.dtype[np.datetime64]] = np.array([np.datetime64(1,
"D")])
48: (0) AR_O: np.ndarray[Any, np.dtype[np.object_]] = np.array([Object()])
49: (0) AR_LIKE_b = [True]
50: (0) AR_LIKE_u = [np.uint32(1)]
51: (0) AR_LIKE_i = [1]
52: (0) AR_LIKE_f = [1.0]
53: (0) AR_LIKE_c = [1j]
54: (0) AR_LIKE_m = [np.timedelta64(1, "D")]
55: (0) AR_LIKE_M = [np.datetime64(1, "D")]
56: (0) AR_LIKE_O = [Object()]
57: (0) AR_b - AR_LIKE_u
58: (0) AR_b - AR_LIKE_i
59: (0) AR_b - AR_LIKE_f
60: (0) AR_b - AR_LIKE_c
61: (0) AR_b - AR_LIKE_m
62: (0) AR_b - AR_LIKE_O
63: (0) AR_LIKE_u - AR_b
64: (0) AR_LIKE_i - AR_b
65: (0) AR_LIKE_f - AR_b
66: (0) AR_LIKE_c - AR_b
67: (0) AR_LIKE_m - AR_b
68: (0) AR_LIKE_M - AR_b
69: (0) AR_LIKE_O - AR_b
70: (0) AR_u - AR_LIKE_b
71: (0) AR_u - AR_LIKE_u
72: (0) AR_u - AR_LIKE_i
73: (0) AR_u - AR_LIKE_f
74: (0) AR_u - AR_LIKE_c

```

75: (0) AR\_u - AR\_LIKE\_m  
76: (0) AR\_u - AR\_LIKE\_O  
77: (0) AR\_LIKE\_b - AR\_u  
78: (0) AR\_LIKE\_u - AR\_u  
79: (0) AR\_LIKE\_i - AR\_u  
80: (0) AR\_LIKE\_f - AR\_u  
81: (0) AR\_LIKE\_c - AR\_u  
82: (0) AR\_LIKE\_m - AR\_u  
83: (0) AR\_LIKE\_M - AR\_u  
84: (0) AR\_LIKE\_O - AR\_u  
85: (0) AR\_i - AR\_LIKE\_b  
86: (0) AR\_i - AR\_LIKE\_u  
87: (0) AR\_i - AR\_LIKE\_i  
88: (0) AR\_i - AR\_LIKE\_f  
89: (0) AR\_i - AR\_LIKE\_c  
90: (0) AR\_i - AR\_LIKE\_m  
91: (0) AR\_i - AR\_LIKE\_O  
92: (0) AR\_LIKE\_b - AR\_i  
93: (0) AR\_LIKE\_u - AR\_i  
94: (0) AR\_LIKE\_i - AR\_i  
95: (0) AR\_LIKE\_f - AR\_i  
96: (0) AR\_LIKE\_c - AR\_i  
97: (0) AR\_LIKE\_m - AR\_i  
98: (0) AR\_LIKE\_M - AR\_i  
99: (0) AR\_LIKE\_O - AR\_i  
100: (0) AR\_f - AR\_LIKE\_b  
101: (0) AR\_f - AR\_LIKE\_u  
102: (0) AR\_f - AR\_LIKE\_i  
103: (0) AR\_f - AR\_LIKE\_f  
104: (0) AR\_f - AR\_LIKE\_c  
105: (0) AR\_f - AR\_LIKE\_O  
106: (0) AR\_LIKE\_b - AR\_f  
107: (0) AR\_LIKE\_u - AR\_f  
108: (0) AR\_LIKE\_i - AR\_f  
109: (0) AR\_LIKE\_f - AR\_f  
110: (0) AR\_LIKE\_c - AR\_f  
111: (0) AR\_LIKE\_O - AR\_f  
112: (0) AR\_c - AR\_LIKE\_b  
113: (0) AR\_c - AR\_LIKE\_u  
114: (0) AR\_c - AR\_LIKE\_i  
115: (0) AR\_c - AR\_LIKE\_f  
116: (0) AR\_c - AR\_LIKE\_c  
117: (0) AR\_c - AR\_LIKE\_O  
118: (0) AR\_LIKE\_b - AR\_c  
119: (0) AR\_LIKE\_u - AR\_c  
120: (0) AR\_LIKE\_i - AR\_c  
121: (0) AR\_LIKE\_f - AR\_c  
122: (0) AR\_LIKE\_c - AR\_c  
123: (0) AR\_LIKE\_O - AR\_c  
124: (0) AR\_m - AR\_LIKE\_b  
125: (0) AR\_m - AR\_LIKE\_u  
126: (0) AR\_m - AR\_LIKE\_i  
127: (0) AR\_m - AR\_LIKE\_m  
128: (0) AR\_LIKE\_b - AR\_m  
129: (0) AR\_LIKE\_u - AR\_m  
130: (0) AR\_LIKE\_i - AR\_m  
131: (0) AR\_LIKE\_m - AR\_m  
132: (0) AR\_LIKE\_M - AR\_m  
133: (0) AR\_M - AR\_LIKE\_b  
134: (0) AR\_M - AR\_LIKE\_u  
135: (0) AR\_M - AR\_LIKE\_i  
136: (0) AR\_M - AR\_LIKE\_m  
137: (0) AR\_M - AR\_LIKE\_M  
138: (0) AR\_LIKE\_M - AR\_M  
139: (0) AR\_O - AR\_LIKE\_b  
140: (0) AR\_O - AR\_LIKE\_u  
141: (0) AR\_O - AR\_LIKE\_i  
142: (0) AR\_O - AR\_LIKE\_f  
143: (0) AR\_O - AR\_LIKE\_c

```
144: (0)          AR_O - AR_LIKE_O
145: (0)          AR_LIKE_b - AR_O
146: (0)          AR_LIKE_u - AR_O
147: (0)          AR_LIKE_i - AR_O
148: (0)          AR_LIKE_f - AR_O
149: (0)          AR_LIKE_c - AR_O
150: (0)          AR_LIKE_O - AR_O
151: (0)          AR_u += AR_b
152: (0)          AR_u += AR_u
153: (0)          AR_u += 1 # Allowed during runtime as long as the object is 0D and >=0
154: (0)          AR_b // AR_LIKE_b
155: (0)          AR_b // AR_LIKE_u
156: (0)          AR_b // AR_LIKE_i
157: (0)          AR_b // AR_LIKE_f
158: (0)          AR_b // AR_LIKE_O
159: (0)          AR_LIKE_b // AR_b
160: (0)          AR_LIKE_u // AR_b
161: (0)          AR_LIKE_i // AR_b
162: (0)          AR_LIKE_f // AR_b
163: (0)          AR_LIKE_O // AR_b
164: (0)          AR_u // AR_LIKE_b
165: (0)          AR_u // AR_LIKE_u
166: (0)          AR_u // AR_LIKE_i
167: (0)          AR_u // AR_LIKE_f
168: (0)          AR_u // AR_LIKE_O
169: (0)          AR_LIKE_b // AR_u
170: (0)          AR_LIKE_u // AR_u
171: (0)          AR_LIKE_i // AR_u
172: (0)          AR_LIKE_f // AR_u
173: (0)          AR_LIKE_m // AR_u
174: (0)          AR_LIKE_O // AR_u
175: (0)          AR_i // AR_LIKE_b
176: (0)          AR_i // AR_LIKE_u
177: (0)          AR_i // AR_LIKE_i
178: (0)          AR_i // AR_LIKE_f
179: (0)          AR_i // AR_LIKE_O
180: (0)          AR_LIKE_b // AR_i
181: (0)          AR_LIKE_u // AR_i
182: (0)          AR_LIKE_i // AR_i
183: (0)          AR_LIKE_f // AR_i
184: (0)          AR_LIKE_m // AR_i
185: (0)          AR_LIKE_O // AR_i
186: (0)          AR_f // AR_LIKE_b
187: (0)          AR_f // AR_LIKE_u
188: (0)          AR_f // AR_LIKE_i
189: (0)          AR_f // AR_LIKE_f
190: (0)          AR_f // AR_LIKE_O
191: (0)          AR_LIKE_b // AR_f
192: (0)          AR_LIKE_u // AR_f
193: (0)          AR_LIKE_i // AR_f
194: (0)          AR_LIKE_f // AR_f
195: (0)          AR_LIKE_m // AR_f
196: (0)          AR_LIKE_O // AR_f
197: (0)          AR_m // AR_LIKE_u
198: (0)          AR_m // AR_LIKE_i
199: (0)          AR_m // AR_LIKE_f
200: (0)          AR_m // AR_LIKE_m
201: (0)          AR_LIKE_m // AR_m
202: (0)          AR_O // AR_LIKE_b
203: (0)          AR_O // AR_LIKE_u
204: (0)          AR_O // AR_LIKE_i
205: (0)          AR_O // AR_LIKE_f
206: (0)          AR_O // AR_LIKE_O
207: (0)          AR_LIKE_b // AR_O
208: (0)          AR_LIKE_u // AR_O
209: (0)          AR_LIKE_i // AR_O
210: (0)          AR_LIKE_f // AR_O
211: (0)          AR_LIKE_O // AR_O
212: (0)          AR_b *= AR_LIKE_b
```

```
213: (0)          AR_u *= AR_LIKE_b
214: (0)          AR_u *= AR_LIKE_u
215: (0)          AR_i *= AR_LIKE_b
216: (0)          AR_i *= AR_LIKE_u
217: (0)          AR_i *= AR_LIKE_i
218: (0)          AR_f *= AR_LIKE_b
219: (0)          AR_f *= AR_LIKE_u
220: (0)          AR_f *= AR_LIKE_i
221: (0)          AR_f *= AR_LIKE_f
222: (0)          AR_c *= AR_LIKE_b
223: (0)          AR_c *= AR_LIKE_u
224: (0)          AR_c *= AR_LIKE_i
225: (0)          AR_c *= AR_LIKE_f
226: (0)          AR_c *= AR_LIKE_c
227: (0)          AR_m *= AR_LIKE_b
228: (0)          AR_m *= AR_LIKE_u
229: (0)          AR_m *= AR_LIKE_i
230: (0)          AR_m *= AR_LIKE_f
231: (0)          AR_O *= AR_LIKE_b
232: (0)          AR_O *= AR_LIKE_u
233: (0)          AR_O *= AR_LIKE_i
234: (0)          AR_O *= AR_LIKE_f
235: (0)          AR_O *= AR_LIKE_c
236: (0)          AR_O *= AR_LIKE_O
237: (0)          AR_u **= AR_LIKE_b
238: (0)          AR_u **= AR_LIKE_u
239: (0)          AR_i **= AR_LIKE_b
240: (0)          AR_i **= AR_LIKE_u
241: (0)          AR_i **= AR_LIKE_i
242: (0)          AR_f **= AR_LIKE_b
243: (0)          AR_f **= AR_LIKE_u
244: (0)          AR_f **= AR_LIKE_i
245: (0)          AR_f **= AR_LIKE_f
246: (0)          AR_c **= AR_LIKE_b
247: (0)          AR_c **= AR_LIKE_u
248: (0)          AR_c **= AR_LIKE_i
249: (0)          AR_c **= AR_LIKE_f
250: (0)          AR_c **= AR_LIKE_c
251: (0)          AR_O **= AR_LIKE_b
252: (0)          AR_O **= AR_LIKE_u
253: (0)          AR_O **= AR_LIKE_i
254: (0)          AR_O **= AR_LIKE_f
255: (0)          AR_O **= AR_LIKE_c
256: (0)          AR_O **= AR_LIKE_O
257: (0)          -c16
258: (0)          -c8
259: (0)          -f8
260: (0)          -f4
261: (0)          -i8
262: (0)          -i4
263: (0)          with pytest.warns(RuntimeWarning):
264: (4)            -u8
265: (4)            -u4
266: (0)          -td
267: (0)          -AR_f
268: (0)          +c16
269: (0)          +c8
270: (0)          +f8
271: (0)          +f4
272: (0)          +i8
273: (0)          +i4
274: (0)          +u8
275: (0)          +u4
276: (0)          +td
277: (0)          +AR_f
278: (0)          abs(c16)
279: (0)          abs(c8)
280: (0)          abs(f8)
281: (0)          abs(f4)
```

```
282: (0)           abs(i8)
283: (0)           abs(i4)
284: (0)           abs(u8)
285: (0)           abs(u4)
286: (0)           abs(td)
287: (0)           abs(b_)
288: (0)           abs(AR_f)
289: (0)           dt + td
290: (0)           dt + i
291: (0)           dt + i4
292: (0)           dt + i8
293: (0)           dt - dt
294: (0)           dt - i
295: (0)           dt - i4
296: (0)           dt - i8
297: (0)           td + td
298: (0)           td + i
299: (0)           td + i4
300: (0)           td + i8
301: (0)           td - td
302: (0)           td - i
303: (0)           td - i4
304: (0)           td - i8
305: (0)           td / f
306: (0)           td / f4
307: (0)           td / f8
308: (0)           td / td
309: (0)           td // td
310: (0)           td % td
311: (0)           b_ / b
312: (0)           b_ / b_
313: (0)           b_ / i
314: (0)           b_ / i8
315: (0)           b_ / i4
316: (0)           b_ / u8
317: (0)           b_ / u4
318: (0)           b_ / f
319: (0)           b_ / f8
320: (0)           b_ / f4
321: (0)           b_ / c
322: (0)           b_ / c16
323: (0)           b_ / c8
324: (0)           b_ / b_
325: (0)           b_ / b
326: (0)           i / b_
327: (0)           i8 / b_
328: (0)           i4 / b_
329: (0)           u8 / b_
330: (0)           u4 / b_
331: (0)           f / b_
332: (0)           f8 / b_
333: (0)           f4 / b_
334: (0)           c / b_
335: (0)           c16 / b_
336: (0)           c8 / b_
337: (0)           c16 + c16
338: (0)           c16 + f8
339: (0)           c16 + i8
340: (0)           c16 + c8
341: (0)           c16 + f4
342: (0)           c16 + i4
343: (0)           c16 + b_
344: (0)           c16 + b
345: (0)           c16 + c
346: (0)           c16 + f
347: (0)           c16 + i
348: (0)           c16 + AR_f
349: (0)           c16 + c16
350: (0)           f8 + c16
```

351: (0) i8 + c16  
352: (0) c8 + c16  
353: (0) f4 + c16  
354: (0) i4 + c16  
355: (0) b\_ + c16  
356: (0) b + c16  
357: (0) c + c16  
358: (0) f + c16  
359: (0) i + c16  
360: (0) AR\_f + c16  
361: (0) c8 + c16  
362: (0) c8 + f8  
363: (0) c8 + i8  
364: (0) c8 + c8  
365: (0) c8 + f4  
366: (0) c8 + i4  
367: (0) c8 + b\_  
368: (0) c8 + b  
369: (0) c8 + c  
370: (0) c8 + f  
371: (0) c8 + i  
372: (0) c8 + AR\_f  
373: (0) c16 + c8  
374: (0) f8 + c8  
375: (0) i8 + c8  
376: (0) c8 + c8  
377: (0) f4 + c8  
378: (0) i4 + c8  
379: (0) b\_ + c8  
380: (0) b + c8  
381: (0) c + c8  
382: (0) f + c8  
383: (0) i + c8  
384: (0) AR\_f + c8  
385: (0) f8 + f8  
386: (0) f8 + i8  
387: (0) f8 + f4  
388: (0) f8 + i4  
389: (0) f8 + b\_  
390: (0) f8 + b  
391: (0) f8 + c  
392: (0) f8 + f  
393: (0) f8 + i  
394: (0) f8 + AR\_f  
395: (0) f8 + f8  
396: (0) i8 + f8  
397: (0) f4 + f8  
398: (0) i4 + f8  
399: (0) b\_ + f8  
400: (0) b + f8  
401: (0) c + f8  
402: (0) f + f8  
403: (0) i + f8  
404: (0) AR\_f + f8  
405: (0) f4 + f8  
406: (0) f4 + i8  
407: (0) f4 + f4  
408: (0) f4 + i4  
409: (0) f4 + b\_  
410: (0) f4 + b  
411: (0) f4 + c  
412: (0) f4 + f  
413: (0) f4 + i  
414: (0) f4 + AR\_f  
415: (0) f8 + f4  
416: (0) i8 + f4  
417: (0) f4 + f4  
418: (0) i4 + f4  
419: (0) b\_ + f4

420: (0) b + f4  
421: (0) c + f4  
422: (0) f + f4  
423: (0) i + f4  
424: (0) AR\_f + f4  
425: (0) i8 + i8  
426: (0) i8 + u8  
427: (0) i8 + i4  
428: (0) i8 + u4  
429: (0) i8 + b\_  
430: (0) i8 + b  
431: (0) i8 + c  
432: (0) i8 + f  
433: (0) i8 + i  
434: (0) i8 + AR\_f  
435: (0) u8 + u8  
436: (0) u8 + i4  
437: (0) u8 + u4  
438: (0) u8 + b\_  
439: (0) u8 + b  
440: (0) u8 + c  
441: (0) u8 + f  
442: (0) u8 + i  
443: (0) u8 + AR\_f  
444: (0) i8 + i8  
445: (0) u8 + i8  
446: (0) i4 + i8  
447: (0) u4 + i8  
448: (0) b\_ + i8  
449: (0) b + i8  
450: (0) c + i8  
451: (0) f + i8  
452: (0) i + i8  
453: (0) AR\_f + i8  
454: (0) u8 + u8  
455: (0) i4 + u8  
456: (0) u4 + u8  
457: (0) b\_ + u8  
458: (0) b + u8  
459: (0) c + u8  
460: (0) f + u8  
461: (0) i + u8  
462: (0) AR\_f + u8  
463: (0) i4 + i8  
464: (0) i4 + i4  
465: (0) i4 + i  
466: (0) i4 + b\_  
467: (0) i4 + b  
468: (0) i4 + AR\_f  
469: (0) u4 + i8  
470: (0) u4 + i4  
471: (0) u4 + u8  
472: (0) u4 + u4  
473: (0) u4 + i  
474: (0) u4 + b\_  
475: (0) u4 + b  
476: (0) u4 + AR\_f  
477: (0) i8 + i4  
478: (0) i4 + i4  
479: (0) i + i4  
480: (0) b\_ + i4  
481: (0) b + i4  
482: (0) AR\_f + i4  
483: (0) i8 + u4  
484: (0) i4 + u4  
485: (0) u8 + u4  
486: (0) u4 + u4  
487: (0) b\_ + u4  
488: (0) b + u4

```
489: (0)          i + u4
490: (0)          AR_f + u4
```

-----  
File 354 - arrayprint.py:

```
1: (0)          import numpy as np
2: (0)          AR = np.arange(10)
3: (0)          AR.setflags(write=False)
4: (0)          with np.printoptions():
5: (4)              np.set_printoptions(
6: (8)                  precision=1,
7: (8)                  threshold=2,
8: (8)                  edgeitems=3,
9: (8)                  linewidth=4,
10: (8)                 suppress=False,
11: (8)                 nanstr="Bob",
12: (8)                 infstr="Bill",
13: (8)                 formatter={},
14: (8)                 sign="+",
15: (8)                 floatmode="unique",
16: (4)             )
17: (4)         np.get_printoptions()
18: (4)         str(AR)
19: (4)         np.array2string(
20: (8)             AR,
21: (8)             max_line_width=5,
22: (8)             precision=2,
23: (8)             suppress_small=True,
24: (8)             separator=";",
25: (8)             prefix="test",
26: (8)             threshold=5,
27: (8)             floatmode="fixed",
28: (8)             suffix="?",
29: (8)             legacy="1.13",
30: (4)         )
31: (4)         np.format_float_scientific(1, precision=5)
32: (4)         np.format_float_positional(1, trim="k")
33: (4)         np.array_repr(AR)
34: (4)         np.array_str(AR)
```

-----  
File 355 - arrayterator.py:

```
1: (0)          from __future__ import annotations
2: (0)          from typing import Any
3: (0)          import numpy as np
4: (0)          AR_i8: np.ndarray[Any, np.dtype[np.int_]] = np.arange(10)
5: (0)          ar_iter = np.lib.Arrayterator(AR_i8)
6: (0)          ar_iter.var
7: (0)          ar_iter.buf_size
8: (0)          ar_iter.start
9: (0)          ar_iter.stop
10: (0)          ar_iter.step
11: (0)          ar_iter.shape
12: (0)          ar_iter.flat
13: (0)          ar_iter.__array__()
14: (0)          for i in ar_iter:
15: (4)              pass
16: (0)          ar_iter[0]
17: (0)          ar_iter[...]
18: (0)          ar_iter[::]
19: (0)          ar_iter[0, 0, 0]
20: (0)          ar_iter[..., 0, ::]
```

## File 356 - array\_constructors.py:

```
1: (0)          import sys
2: (0)          from typing import Any
3: (0)          import numpy as np
4: (0)          class Index:
5: (4)              def __index__(self) -> int:
6: (8)                  return 0
7: (0)          class SubClass(np.ndarray):
8: (4)              pass
9: (0)          def func(i: int, j: int, **kwargs: Any) -> SubClass:
10: (4)              return B
11: (0)          i8 = np.int64(1)
12: (0)          A = np.array([1])
13: (0)          B = A.view(SubClass).copy()
14: (0)          B_stack = np.array([[1], [1]]).view(SubClass)
15: (0)          C = [1]
16: (0)          np.ndarray(Index())
17: (0)          np.ndarray([Index()])
18: (0)          np.array(1, dtype=float)
19: (0)          np.array(1, copy=False)
20: (0)          np.array(1, order='F')
21: (0)          np.array(1, order=None)
22: (0)          np.array(1, subok=True)
23: (0)          np.array(1, ndmin=3)
24: (0)          np.array(1, str, copy=True, order='C', subok=False, ndmin=2)
25: (0)          np.asarray(A)
26: (0)          np.asarray(B)
27: (0)          np.asarray(C)
28: (0)          np.asanyarray(A)
29: (0)          np.asanyarray(B)
30: (0)          np.asanyarray(B, dtype=int)
31: (0)          np.asanyarray(C)
32: (0)          np.ascontiguousarray(A)
33: (0)          np.ascontiguousarray(B)
34: (0)          np.ascontiguousarray(C)
35: (0)          np.asfortranarray(A)
36: (0)          np.asfortranarray(B)
37: (0)          np.asfortranarray(C)
38: (0)          np.require(A)
39: (0)          np.require(B)
40: (0)          np.require(B, dtype=int)
41: (0)          np.require(B, requirements=None)
42: (0)          np.require(B, requirements="E")
43: (0)          np.require(B, requirements=["ENSUREARRAY"])
44: (0)          np.require(B, requirements={"F", "E"})
45: (0)          np.require(B, requirements=["C", "OWNDATA"])
46: (0)          np.require(B, requirements="W")
47: (0)          np.require(B, requirements="A")
48: (0)          np.require(C)
49: (0)          np.linspace(0, 2)
50: (0)          np.linspace(0.5, [0, 1, 2])
51: (0)          np.linspace([0, 1, 2], 3)
52: (0)          np.linspace(0j, 2)
53: (0)          np.linspace(0, 2, num=10)
54: (0)          np.linspace(0, 2, endpoint=True)
55: (0)          np.linspace(0, 2, retstep=True)
56: (0)          np.linspace(0j, 2j, retstep=True)
57: (0)          np.linspace(0, 2, dtype=bool)
58: (0)          np.linspace([0, 1], [2, 3], axis=Index())
59: (0)          np.logspace(0, 2, base=2)
60: (0)          np.logspace(0, 2, base=2)
61: (0)          np.logspace(0, 2, base=[1j, 2j], num=2)
62: (0)          np.geomspace(1, 2)
63: (0)          np.zeros_like(A)
64: (0)          np.zeros_like(C)
65: (0)          np.zeros_like(B)
66: (0)          np.zeros_like(B, dtype=np.int64)
67: (0)          np.ones_like(A)
```

```

68: (0)          np.ones_like(C)
69: (0)          np.ones_like(B)
70: (0)          np.ones_like(B, dtype=np.int64)
71: (0)          np.empty_like(A)
72: (0)          np.empty_like(C)
73: (0)          np.empty_like(B)
74: (0)          np.empty_like(B, dtype=np.int64)
75: (0)          np.full_like(A, i8)
76: (0)          np.full_like(C, i8)
77: (0)          np.full_like(B, i8)
78: (0)          np.full_like(B, i8, dtype=np.int64)
79: (0)          np.ones(1)
80: (0)          np.ones([1, 1, 1])
81: (0)          np.full(1, i8)
82: (0)          np.full([1, 1, 1], i8)
83: (0)          np.indices([1, 2, 3])
84: (0)          np.indices([1, 2, 3], sparse=True)
85: (0)          np.fromfunction(func, (3, 5))
86: (0)          np.identity(10)
87: (0)          np.atleast_1d(C)
88: (0)          np.atleast_1d(A)
89: (0)          np.atleast_1d(C, C)
90: (0)          np.atleast_1d(C, A)
91: (0)          np.atleast_1d(A, A)
92: (0)          np.atleast_2d(C)
93: (0)          np.atleast_3d(C)
94: (0)          np.vstack([C, C])
95: (0)          np.vstack([C, A])
96: (0)          np.vstack([A, A])
97: (0)          np.hstack([C, C])
98: (0)          np.stack([C, C])
99: (0)          np.stack([C, C], axis=0)
100: (0)         np.stack([C, C], out=B_stack)
101: (0)         np.block([[C, C], [C, C]])
102: (0)         np.block(A)

```

---

## File 357 - array\_like.py:

```

1: (0)          from __future__ import annotations
2: (0)          from typing import Any
3: (0)          import numpy as np
4: (0)          from numpy._typing import ArrayLike, _SupportsArray
5: (0)          x1: ArrayLike = True
6: (0)          x2: ArrayLike = 5
7: (0)          x3: ArrayLike = 1.0
8: (0)          x4: ArrayLike = 1 + 1j
9: (0)          x5: ArrayLike = np.int8(1)
10: (0)         x6: ArrayLike = np.float64(1)
11: (0)         x7: ArrayLike = np.complex128(1)
12: (0)         x8: ArrayLike = np.array([1, 2, 3])
13: (0)         x9: ArrayLike = [1, 2, 3]
14: (0)         x10: ArrayLike = (1, 2, 3)
15: (0)         x11: ArrayLike = "foo"
16: (0)         x12: ArrayLike = memoryview(b'foo')
17: (0)         class A:
18: (4)             def __array__(self, dtype: None | np.dtype[Any] = None) -> np.ndarray:
19: (8)                 return np.array([1, 2, 3])
20: (0)                 x13: ArrayLike = A()
21: (0)                 scalar: _SupportsArray = np.int64(1)
22: (0)                 scalar.__array__()
23: (0)                 array: _SupportsArray = np.array(1)
24: (0)                 array.__array__()
25: (0)                 a: _SupportsArray = A()
26: (0)                 a.__array__()
27: (0)                 a.__array__()
28: (0)                 object_array_scalar: Any = (i for i in range(10))
29: (0)                 np.array(object_array_scalar)

```

-----  
File 358 - bitwise\_ops.py:

```
1: (0)          import numpy as np
2: (0)          i8 = np.int64(1)
3: (0)          u8 = np.uint64(1)
4: (0)          i4 = np.int32(1)
5: (0)          u4 = np.uint32(1)
6: (0)          b_ = np.bool_(1)
7: (0)          b = bool(1)
8: (0)          i = int(1)
9: (0)          AR = np.array([0, 1, 2], dtype=np.int32)
10: (0)         AR.setflags(write=False)
11: (0)         i8 << i8
12: (0)         i8 >> i8
13: (0)         i8 | i8
14: (0)         i8 ^ i8
15: (0)         i8 & i8
16: (0)         i8 << AR
17: (0)         i8 >> AR
18: (0)         i8 | AR
19: (0)         i8 ^ AR
20: (0)         i8 & AR
21: (0)         i4 << i4
22: (0)         i4 >> i4
23: (0)         i4 | i4
24: (0)         i4 ^ i4
25: (0)         i4 & i4
26: (0)         i8 << i4
27: (0)         i8 >> i4
28: (0)         i8 | i4
29: (0)         i8 ^ i4
30: (0)         i8 & i4
31: (0)         i8 << i
32: (0)         i8 >> i
33: (0)         i8 | i
34: (0)         i8 ^ i
35: (0)         i8 & i
36: (0)         i8 << b_
37: (0)         i8 >> b_
38: (0)         i8 | b_
39: (0)         i8 ^ b_
40: (0)         i8 & b_
41: (0)         i8 << b
42: (0)         i8 >> b
43: (0)         i8 | b
44: (0)         i8 ^ b
45: (0)         i8 & b
46: (0)         u8 << u8
47: (0)         u8 >> u8
48: (0)         u8 | u8
49: (0)         u8 ^ u8
50: (0)         u8 & u8
51: (0)         u8 << AR
52: (0)         u8 >> AR
53: (0)         u8 | AR
54: (0)         u8 ^ AR
55: (0)         u8 & AR
56: (0)         u4 << u4
57: (0)         u4 >> u4
58: (0)         u4 | u4
59: (0)         u4 ^ u4
60: (0)         u4 & u4
61: (0)         u4 << i4
62: (0)         u4 >> i4
63: (0)         u4 | i4
64: (0)         u4 ^ i4
```

```

65: (0)          u4 & i4
66: (0)          u4 << i
67: (0)          u4 >> i
68: (0)          u4 | i
69: (0)          u4 ^ i
70: (0)          u4 & i
71: (0)          u8 << b_
72: (0)          u8 >> b_
73: (0)          u8 | b_
74: (0)          u8 ^ b_
75: (0)          u8 & b_
76: (0)          u8 << b
77: (0)          u8 >> b
78: (0)          u8 | b
79: (0)          u8 ^ b
80: (0)          u8 & b
81: (0)          b_ << b_
82: (0)          b_ >> b_
83: (0)          b_ | b_
84: (0)          b_ ^ b_
85: (0)          b_ & b_
86: (0)          b_ << AR
87: (0)          b_ >> AR
88: (0)          b_ | AR
89: (0)          b_ ^ AR
90: (0)          b_ & AR
91: (0)          b_ << b
92: (0)          b_ >> b
93: (0)          b_ | b
94: (0)          b_ ^ b
95: (0)          b_ & b
96: (0)          b_ << i
97: (0)          b_ >> i
98: (0)          b_ | i
99: (0)          b_ ^ i
100: (0)         b_ & i
101: (0)         ~i8
102: (0)         ~i4
103: (0)         ~u8
104: (0)         ~u4
105: (0)         ~b_
106: (0)         ~AR
-----
```

**File 359 - comparisons.py:**

```

1: (0)          from __future__ import annotations
2: (0)          from typing import Any
3: (0)          import numpy as np
4: (0)          c16 = np.complex128()
5: (0)          f8 = np.float64()
6: (0)          i8 = np.int64()
7: (0)          u8 = np.uint64()
8: (0)          c8 = np.complex64()
9: (0)          f4 = np.float32()
10: (0)         i4 = np.int32()
11: (0)         u4 = np.uint32()
12: (0)         dt = np.datetime64(0, "D")
13: (0)         td = np.timedelta64(0, "D")
14: (0)         b_ = np.bool_()
15: (0)         b = bool()
16: (0)         c = complex()
17: (0)         f = float()
18: (0)         i = int()
19: (0)         SEQ = (0, 1, 2, 3, 4)
20: (0)         AR_b: np.ndarray[Any, np.dtype[np.bool_]] = np.array([True])
21: (0)         AR_u: np.ndarray[Any, np.dtype[np.uint32]] = np.array([1], dtype=np.uint32)
22: (0)         AR_i: np.ndarray[Any, np.dtype[np.int_]] = np.array([1])
```

```
23: (0)          AR_f: np.ndarray[Any, np.dtype[np.float_]] = np.array([1.0])
24: (0)          AR_c: np.ndarray[Any, np.dtype[np.complex_]] = np.array([1.0j])
25: (0)          AR_m: np.ndarray[Any, np.dtype[np.timedelta64]] =
np.array([np.timedelta64("1")])
26: (0)          AR_M: np.ndarray[Any, np.dtype[np.datetime64]] =
np.array([np.datetime64("1")])
27: (0)          AR_O: np.ndarray[Any, np.dtype[np.object_]] = np.array([1], dtype=object)
28: (0)          AR_b > AR_b
29: (0)          AR_b > AR_u
30: (0)          AR_b > AR_i
31: (0)          AR_b > AR_f
32: (0)          AR_b > AR_c
33: (0)          AR_u > AR_b
34: (0)          AR_u > AR_u
35: (0)          AR_u > AR_i
36: (0)          AR_u > AR_f
37: (0)          AR_u > AR_c
38: (0)          AR_i > AR_b
39: (0)          AR_i > AR_u
40: (0)          AR_i > AR_i
41: (0)          AR_i > AR_f
42: (0)          AR_i > AR_c
43: (0)          AR_f > AR_b
44: (0)          AR_f > AR_u
45: (0)          AR_f > AR_i
46: (0)          AR_f > AR_f
47: (0)          AR_f > AR_c
48: (0)          AR_c > AR_b
49: (0)          AR_c > AR_u
50: (0)          AR_c > AR_i
51: (0)          AR_c > AR_f
52: (0)          AR_c > AR_c
53: (0)          AR_m > AR_b
54: (0)          AR_m > AR_u
55: (0)          AR_m > AR_i
56: (0)          AR_b > AR_m
57: (0)          AR_u > AR_m
58: (0)          AR_i > AR_m
59: (0)          AR_M > AR_M
60: (0)          AR_O > AR_O
61: (0)          1 > AR_O
62: (0)          AR_O > 1
63: (0)          dt > dt
64: (0)          td > td
65: (0)          td > i
66: (0)          td > i4
67: (0)          td > i8
68: (0)          td > AR_i
69: (0)          td > SEQ
70: (0)          b_ > b
71: (0)          b_ > b_
72: (0)          b_ > i
73: (0)          b_ > i8
74: (0)          b_ > i4
75: (0)          b_ > u8
76: (0)          b_ > u4
77: (0)          b_ > f
78: (0)          b_ > f8
79: (0)          b_ > f4
80: (0)          b_ > c
81: (0)          b_ > c16
82: (0)          b_ > c8
83: (0)          b_ > AR_i
84: (0)          b_ > SEQ
85: (0)          c16 > c16
86: (0)          c16 > f8
87: (0)          c16 > i8
88: (0)          c16 > c8
89: (0)          c16 > f4
```

90: (0) c16 > i4  
91: (0) c16 > b\_  
92: (0) c16 > b  
93: (0) c16 > c  
94: (0) c16 > f  
95: (0) c16 > i  
96: (0) c16 > AR\_i  
97: (0) c16 > SEQ  
98: (0) c16 > c16  
99: (0) f8 > c16  
100: (0) i8 > c16  
101: (0) c8 > c16  
102: (0) f4 > c16  
103: (0) i4 > c16  
104: (0) b\_ > c16  
105: (0) b > c16  
106: (0) c > c16  
107: (0) f > c16  
108: (0) i > c16  
109: (0) AR\_i > c16  
110: (0) SEQ > c16  
111: (0) c8 > c16  
112: (0) c8 > f8  
113: (0) c8 > i8  
114: (0) c8 > c8  
115: (0) c8 > f4  
116: (0) c8 > i4  
117: (0) c8 > b\_  
118: (0) c8 > b  
119: (0) c8 > c  
120: (0) c8 > f  
121: (0) c8 > i  
122: (0) c8 > AR\_i  
123: (0) c8 > SEQ  
124: (0) c16 > c8  
125: (0) f8 > c8  
126: (0) i8 > c8  
127: (0) c8 > c8  
128: (0) f4 > c8  
129: (0) i4 > c8  
130: (0) b\_ > c8  
131: (0) b > c8  
132: (0) c > c8  
133: (0) f > c8  
134: (0) i > c8  
135: (0) AR\_i > c8  
136: (0) SEQ > c8  
137: (0) f8 > f8  
138: (0) f8 > i8  
139: (0) f8 > f4  
140: (0) f8 > i4  
141: (0) f8 > b\_  
142: (0) f8 > b  
143: (0) f8 > c  
144: (0) f8 > f  
145: (0) f8 > i  
146: (0) f8 > AR\_i  
147: (0) f8 > SEQ  
148: (0) f8 > f8  
149: (0) i8 > f8  
150: (0) f4 > f8  
151: (0) i4 > f8  
152: (0) b\_ > f8  
153: (0) b > f8  
154: (0) c > f8  
155: (0) f > f8  
156: (0) i > f8  
157: (0) AR\_i > f8  
158: (0) SEQ > f8

159: (0) f4 > f8  
160: (0) f4 > i8  
161: (0) f4 > f4  
162: (0) f4 > i4  
163: (0) f4 > b\_  
164: (0) f4 > b  
165: (0) f4 > c  
166: (0) f4 > f  
167: (0) f4 > i  
168: (0) f4 > AR\_i  
169: (0) f4 > SEQ  
170: (0) f8 > f4  
171: (0) i8 > f4  
172: (0) f4 > f4  
173: (0) i4 > f4  
174: (0) b\_ > f4  
175: (0) b > f4  
176: (0) c > f4  
177: (0) f > f4  
178: (0) i > f4  
179: (0) AR\_i > f4  
180: (0) SEQ > f4  
181: (0) i8 > i8  
182: (0) i8 > u8  
183: (0) i8 > i4  
184: (0) i8 > u4  
185: (0) i8 > b\_  
186: (0) i8 > b  
187: (0) i8 > c  
188: (0) i8 > f  
189: (0) i8 > i  
190: (0) i8 > AR\_i  
191: (0) i8 > SEQ  
192: (0) u8 > u8  
193: (0) u8 > i4  
194: (0) u8 > u4  
195: (0) u8 > b\_  
196: (0) u8 > b  
197: (0) u8 > c  
198: (0) u8 > f  
199: (0) u8 > i  
200: (0) u8 > AR\_i  
201: (0) u8 > SEQ  
202: (0) i8 > i8  
203: (0) u8 > i8  
204: (0) i4 > i8  
205: (0) u4 > i8  
206: (0) b\_ > i8  
207: (0) b > i8  
208: (0) c > i8  
209: (0) f > i8  
210: (0) i > i8  
211: (0) AR\_i > i8  
212: (0) SEQ > i8  
213: (0) u8 > u8  
214: (0) i4 > u8  
215: (0) u4 > u8  
216: (0) b\_ > u8  
217: (0) b > u8  
218: (0) c > u8  
219: (0) f > u8  
220: (0) i > u8  
221: (0) AR\_i > u8  
222: (0) SEQ > u8  
223: (0) i4 > i8  
224: (0) i4 > i4  
225: (0) i4 > i  
226: (0) i4 > b\_  
227: (0) i4 > b

```

228: (0)          i4 > AR_i
229: (0)          i4 > SEQ
230: (0)          u4 > i8
231: (0)          u4 > i4
232: (0)          u4 > u8
233: (0)          u4 > u4
234: (0)          u4 > i
235: (0)          u4 > b_
236: (0)          u4 > b
237: (0)          u4 > AR_i
238: (0)          u4 > SEQ
239: (0)          i8 > i4
240: (0)          i4 > i4
241: (0)          i > i4
242: (0)          b_ > i4
243: (0)          b > i4
244: (0)          AR_i > i4
245: (0)          SEQ > i4
246: (0)          i8 > u4
247: (0)          i4 > u4
248: (0)          u8 > u4
249: (0)          u4 > u4
250: (0)          b_ > u4
251: (0)          b > u4
252: (0)          i > u4
253: (0)          AR_i > u4
254: (0)          SEQ > u4
-----
```

File 360 - dtype.py:

```

1: (0)          import numpy as np
2: (0)          dtype_obj = np.dtype(np.str_)
3: (0)          void_dtype_obj = np.dtype([("f0", np.float64), ("f1", np.float32)])
4: (0)          np.dtype(dtype=np.int64)
5: (0)          np.dtype(int)
6: (0)          np.dtype("int")
7: (0)          np.dtype(None)
8: (0)          np.dtype((int, 2))
9: (0)          np.dtype((int, (1,)))
10: (0)         np.dtype({"names": ["a", "b"], "formats": [int, float]}) 
11: (0)         np.dtype({"names": ["a"], "formats": [int], "titles": [object]}) 
12: (0)         np.dtype({"names": ["a"], "formats": [int], "titles": [object()]}) 
13: (0)         np.dtype([(("name", np.str_, 16), ("grades", np.float64, (2,)), ("age",
"int32"))])
14: (0)         np.dtype(
15: (4)           {
16: (8)             "names": ["a", "b"],
17: (8)             "formats": [int, float],
18: (8)             "itemsize": 9,
19: (8)             "aligned": False,
20: (8)             "titles": ["x", "y"],
21: (8)             "offsets": [0, 1],
22: (4)           }
23: (0)         )
24: (0)         np.dtype((np.float_, float))
25: (0)         class Test:
26: (4)           dtype = np.dtype(float)
27: (0)           np.dtype(Test())
28: (0)           dtype_obj.base
29: (0)           dtype_obj.subdtype
30: (0)           dtype_obj.newbyteorder()
31: (0)           dtype_obj.type
32: (0)           dtype_obj.name
33: (0)           dtype_obj.names
34: (0)           dtype_obj * 0
35: (0)           dtype_obj * 2
36: (0)           0 * dtype_obj
```

```

37: (0)          2 * dtype_obj
38: (0)          void_dtype_obj["f0"]
39: (0)          void_dtype_obj[0]
40: (0)          void_dtype_obj[["f0", "f1"]]
41: (0)          void_dtype_obj[["f0"]]

-----

```

## File 361 - einsumfunc.py:

```

1: (0)          from __future__ import annotations
2: (0)          from typing import Any
3: (0)          import numpy as np
4: (0)          AR_LIKE_b = [True, True, True]
5: (0)          AR_LIKE_u = [np.uint32(1), np.uint32(2), np.uint32(3)]
6: (0)          AR_LIKE_i = [1, 2, 3]
7: (0)          AR_LIKE_f = [1.0, 2.0, 3.0]
8: (0)          AR_LIKE_c = [1j, 2j, 3j]
9: (0)          AR_LIKE_U = ["1", "2", "3"]
10: (0)         OUT_f: np.ndarray[Any, np.dtype[np.float64]] = np.empty(3, dtype=np.float64)
11: (0)         OUT_c: np.ndarray[Any, np.dtype[np.complex128]] = np.empty(3,
dtype=np.complex128)
12: (0)         np.einsum("i,i->i", AR_LIKE_b, AR_LIKE_b)
13: (0)         np.einsum("i,i->i", AR_LIKE_u, AR_LIKE_u)
14: (0)         np.einsum("i,i->i", AR_LIKE_i, AR_LIKE_i)
15: (0)         np.einsum("i,i->i", AR_LIKE_f, AR_LIKE_f)
16: (0)         np.einsum("i,i->i", AR_LIKE_c, AR_LIKE_c)
17: (0)         np.einsum("i,i->i", AR_LIKE_b, AR_LIKE_i)
18: (0)         np.einsum("i,i,i,i->i", AR_LIKE_b, AR_LIKE_u, AR_LIKE_i, AR_LIKE_c)
19: (0)         np.einsum("i,i->i", AR_LIKE_f, AR_LIKE_f, dtype="c16")
20: (0)         np.einsum("i,i->i", AR_LIKE_U, AR_LIKE_U, dtype=bool, casting="unsafe")
21: (0)         np.einsum("i,i->i", AR_LIKE_f, AR_LIKE_f, out=OUT_c)
22: (0)         np.einsum("i,i->i", AR_LIKE_U, AR_LIKE_U, dtype=int, casting="unsafe",
out=OUT_f)
23: (0)         np.einsum_path("i,i->i", AR_LIKE_b, AR_LIKE_b)
24: (0)         np.einsum_path("i,i->i", AR_LIKE_u, AR_LIKE_u)
25: (0)         np.einsum_path("i,i->i", AR_LIKE_i, AR_LIKE_i)
26: (0)         np.einsum_path("i,i->i", AR_LIKE_f, AR_LIKE_f)
27: (0)         np.einsum_path("i,i->i", AR_LIKE_c, AR_LIKE_c)
28: (0)         np.einsum_path("i,i->i", AR_LIKE_b, AR_LIKE_i)
29: (0)         np.einsum_path("i,i,i,i->i", AR_LIKE_b, AR_LIKE_u, AR_LIKE_i, AR_LIKE_c)

-----

```

## File 362 - flatiter.py:

```

1: (0)          import numpy as np
2: (0)          a = np.empty((2, 2)).flat
3: (0)          a.base
4: (0)          a.copy()
5: (0)          a.coords
6: (0)          a.index
7: (0)          iter(a)
8: (0)          next(a)
9: (0)          a[0]
10: (0)         a[[0, 1, 2]]
11: (0)         a[...]
12: (0)         a[:]
13: (0)         a.__array__()
14: (0)         a.__array__(np.dtype(np.float64))

-----

```

## File 363 - fromnumeric.py:

```

1: (0)          """Tests for :mod:`numpy.core.fromnumeric`."""
2: (0)          import numpy as np
3: (0)          A = np.array(True, ndmin=2, dtype=bool)
4: (0)          B = np.array(1.0, ndmin=2, dtype=np.float32)


```

```
5: (0)          A.setflags(write=False)
6: (0)          B.setflags(write=False)
7: (0)          a = np.bool_(True)
8: (0)          b = np.float32(1.0)
9: (0)          c = 1.0
10: (0)         d = np.array(1.0, dtype=np.float32) # writeable
11: (0)         np.take(a, 0)
12: (0)         np.take(b, 0)
13: (0)         np.take(c, 0)
14: (0)         np.take(A, 0)
15: (0)         np.take(B, 0)
16: (0)         np.take(A, [0])
17: (0)         np.take(B, [0])
18: (0)         np.reshape(a, 1)
19: (0)         np.reshape(b, 1)
20: (0)         np.reshape(c, 1)
21: (0)         np.reshape(A, 1)
22: (0)         np.reshape(B, 1)
23: (0)         np.choose(a, [True, True])
24: (0)         np.choose(A, [1.0, 1.0])
25: (0)         np.repeat(a, 1)
26: (0)         np.repeat(b, 1)
27: (0)         np.repeat(c, 1)
28: (0)         np.repeat(A, 1)
29: (0)         np.repeat(B, 1)
30: (0)         np.swapaxes(A, 0, 0)
31: (0)         np.swapaxes(B, 0, 0)
32: (0)         np.transpose(a)
33: (0)         np.transpose(b)
34: (0)         np.transpose(c)
35: (0)         np.transpose(A)
36: (0)         np.transpose(B)
37: (0)         np.partition(a, 0, axis=None)
38: (0)         np.partition(b, 0, axis=None)
39: (0)         np.partition(c, 0, axis=None)
40: (0)         np.partition(A, 0)
41: (0)         np.partition(B, 0)
42: (0)         np.argpartition(a, 0)
43: (0)         np.argpartition(b, 0)
44: (0)         np.argpartition(c, 0)
45: (0)         np.argpartition(A, 0)
46: (0)         np.argpartition(B, 0)
47: (0)         np.sort(A, 0)
48: (0)         np.sort(B, 0)
49: (0)         np.argsort(A, 0)
50: (0)         np.argsort(B, 0)
51: (0)         np.argmax(A)
52: (0)         np.argmax(B)
53: (0)         np.argmax(A, axis=0)
54: (0)         np.argmax(B, axis=0)
55: (0)         np.argmin(A)
56: (0)         np.argmin(B)
57: (0)         np.argmin(A, axis=0)
58: (0)         np.argmin(B, axis=0)
59: (0)         np.searchsorted(A[0], 0)
60: (0)         np.searchsorted(B[0], 0)
61: (0)         np.searchsorted(A[0], [0])
62: (0)         np.searchsorted(B[0], [0])
63: (0)         np.resize(a, (5, 5))
64: (0)         np.resize(b, (5, 5))
65: (0)         np.resize(c, (5, 5))
66: (0)         np.resize(A, (5, 5))
67: (0)         np.resize(B, (5, 5))
68: (0)         np.squeeze(a)
69: (0)         np.squeeze(b)
70: (0)         np.squeeze(c)
71: (0)         np.squeeze(A)
72: (0)         np.squeeze(B)
73: (0)         np.diagonal(A)
```

```
74: (0) np.diagonal(B)
75: (0) np.trace(A)
76: (0) np.trace(B)
77: (0) np.ravel(a)
78: (0) np.ravel(b)
79: (0) np.ravel(c)
80: (0) np.ravel(A)
81: (0) np.ravel(B)
82: (0) np.nonzero(A)
83: (0) np.nonzero(B)
84: (0) np.shape(a)
85: (0) np.shape(b)
86: (0) np.shape(c)
87: (0) np.shape(A)
88: (0) np.shape(B)
89: (0) np.compress([True], a)
90: (0) np.compress([True], b)
91: (0) np.compress([True], c)
92: (0) np.compress([True], A)
93: (0) np.compress([True], B)
94: (0) np.clip(a, 0, 1.0)
95: (0) np.clip(b, -1, 1)
96: (0) np.clip(a, 0, None)
97: (0) np.clip(b, None, 1)
98: (0) np.clip(c, 0, 1)
99: (0) np.clip(A, 0, 1)
100: (0) np.clip(B, 0, 1)
101: (0) np.clip(B, [0, 1], [1, 2])
102: (0) np.sum(a)
103: (0) np.sum(b)
104: (0) np.sum(c)
105: (0) np.sum(A)
106: (0) np.sum(B)
107: (0) np.sum(A, axis=0)
108: (0) np.sum(B, axis=0)
109: (0) np.all(a)
110: (0) np.all(b)
111: (0) np.all(c)
112: (0) np.all(A)
113: (0) np.all(B)
114: (0) np.all(A, axis=0)
115: (0) np.all(B, axis=0)
116: (0) np.all(A, keepdims=True)
117: (0) np.all(B, keepdims=True)
118: (0) np.any(a)
119: (0) np.any(b)
120: (0) np.any(c)
121: (0) np.any(A)
122: (0) np.any(B)
123: (0) np.any(A, axis=0)
124: (0) np.any(B, axis=0)
125: (0) np.any(A, keepdims=True)
126: (0) np.any(B, keepdims=True)
127: (0) np.cumsum(a)
128: (0) np.cumsum(b)
129: (0) np.cumsum(c)
130: (0) np.cumsum(A)
131: (0) np.cumsum(B)
132: (0) np.ptp(b)
133: (0) np.ptp(c)
134: (0) np.ptp(B)
135: (0) np.ptp(B, axis=0)
136: (0) np.ptp(B, keepdims=True)
137: (0) np.amax(a)
138: (0) np.amax(b)
139: (0) np.amax(c)
140: (0) np.amax(A)
141: (0) np.amax(B)
142: (0) np.amax(A, axis=0)
```

```
143: (0) np.amax(B, axis=0)
144: (0) np.amax(A, keepdims=True)
145: (0) np.amax(B, keepdims=True)
146: (0) np.amin(a)
147: (0) np.amin(b)
148: (0) np.amin(c)
149: (0) np.amin(A)
150: (0) np.amin(B)
151: (0) np.amin(A, axis=0)
152: (0) np.amin(B, axis=0)
153: (0) np.amin(A, keepdims=True)
154: (0) np.amin(B, keepdims=True)
155: (0) np.prod(a)
156: (0) np.prod(b)
157: (0) np.prod(c)
158: (0) np.prod(A)
159: (0) np.prod(B)
160: (0) np.prod(a, dtype=None)
161: (0) np.prod(A, dtype=None)
162: (0) np.prod(A, axis=0)
163: (0) np.prod(B, axis=0)
164: (0) np.prod(A, keepdims=True)
165: (0) np.prod(B, keepdims=True)
166: (0) np.prod(b, out=d)
167: (0) np.prod(B, out=d)
168: (0) np.cumprod(a)
169: (0) np.cumprod(b)
170: (0) np.cumprod(c)
171: (0) np.cumprod(A)
172: (0) np.cumprod(B)
173: (0) np.ndim(a)
174: (0) np.ndim(b)
175: (0) np.ndim(c)
176: (0) np.ndim(A)
177: (0) np.ndim(B)
178: (0) np.size(a)
179: (0) np.size(b)
180: (0) np.size(c)
181: (0) np.size(A)
182: (0) np.size(B)
183: (0) np.around(a)
184: (0) np.around(b)
185: (0) np.around(c)
186: (0) np.around(A)
187: (0) np.around(B)
188: (0) np.mean(a)
189: (0) np.mean(b)
190: (0) np.mean(c)
191: (0) np.mean(A)
192: (0) np.mean(B)
193: (0) np.mean(A, axis=0)
194: (0) np.mean(B, axis=0)
195: (0) np.mean(A, keepdims=True)
196: (0) np.mean(B, keepdims=True)
197: (0) np.mean(b, out=d)
198: (0) np.mean(B, out=d)
199: (0) np.std(a)
200: (0) np.std(b)
201: (0) np.std(c)
202: (0) np.std(A)
203: (0) np.std(B)
204: (0) np.std(A, axis=0)
205: (0) np.std(B, axis=0)
206: (0) np.std(A, keepdims=True)
207: (0) np.std(B, keepdims=True)
208: (0) np.std(b, out=d)
209: (0) np.std(B, out=d)
210: (0) np.var(a)
211: (0) np.var(b)
```

```

212: (0)          np.var(c)
213: (0)          np.var(A)
214: (0)          np.var(B)
215: (0)          np.var(A, axis=0)
216: (0)          np.var(B, axis=0)
217: (0)          np.var(A, keepdims=True)
218: (0)          np.var(B, keepdims=True)
219: (0)          np.var(b, out=d)
220: (0)          np.var(B, out=d)
-----
```

## File 364 - lib\_utils.py:

```

1: (0)          from __future__ import annotations
2: (0)          from io import StringIO
3: (0)          import numpy as np
4: (0)          FILE = StringIO()
5: (0)          AR = np.arange(10, dtype=np.float64)
6: (0)          def func(a: int) -> bool:
7: (4)              return True
8: (0)          np.deprecate(func)
9: (0)          np.deprecate()
10: (0)         np.deprecate_with_doc("test")
11: (0)         np.deprecate_with_doc(None)
12: (0)         np.byte_bounds(AR)
13: (0)         np.byte_bounds(np.float64())
14: (0)         np.info(1, output=FILE)
15: (0)         np.source(np.interp, output=FILE)
16: (0)         np.lookfor("binary representation", output=FILE)
-----
```

## File 365 - index\_tricks.py:

```

1: (0)          from __future__ import annotations
2: (0)          from typing import Any
3: (0)          import numpy as np
4: (0)          AR_LIKE_b = [[True, True], [True, True]]
5: (0)          AR_LIKE_i = [[1, 2], [3, 4]]
6: (0)          AR_LIKE_f = [[1.0, 2.0], [3.0, 4.0]]
7: (0)          AR_LIKE_U = [["1", "2"], ["3", "4"]]
8: (0)          AR_i8: np.ndarray[Any, np.dtype[np.int64]] = np.array(AR_LIKE_i,
dtype=np.int64)
9: (0)          np.ndenumerate(AR_i8)
10: (0)         np.ndenumerate(AR_LIKE_f)
11: (0)         np.ndenumerate(AR_LIKE_U)
12: (0)         np.ndenumerate(AR_i8).iter
13: (0)         np.ndenumerate(AR_LIKE_f).iter
14: (0)         np.ndenumerate(AR_LIKE_U).iter
15: (0)         next(np.ndenumerate(AR_i8))
16: (0)         next(np.ndenumerate(AR_LIKE_f))
17: (0)         next(np.ndenumerate(AR_LIKE_U))
18: (0)         iter(np.ndenumerate(AR_i8))
19: (0)         iter(np.ndenumerate(AR_LIKE_f))
20: (0)         iter(np.ndenumerate(AR_LIKE_U))
21: (0)         iter(np.ndindex(1, 2, 3))
22: (0)         next(np.ndindex(1, 2, 3))
23: (0)         np.unravel_index([22, 41, 37], (7, 6))
24: (0)         np.unravel_index([31, 41, 13], (7, 6), order='F')
25: (0)         np.unravel_index(1621, (6, 7, 8, 9))
26: (0)         np.ravel_multi_index(AR_LIKE_i, (7, 6))
27: (0)         np.ravel_multi_index(AR_LIKE_i, (7, 6), order='F')
28: (0)         np.ravel_multi_index(AR_LIKE_i, (4, 6), mode='clip')
29: (0)         np.ravel_multi_index(AR_LIKE_i, (4, 4), mode=('clip', 'wrap'))
30: (0)         np.ravel_multi_index((3, 1, 4, 1), (6, 7, 8, 9))
31: (0)         np.mgrid[1:1:2]
32: (0)         np.mgrid[1:1:2, None:10]
33: (0)         np.ogrid[1:1:2]
-----
```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

34: (0)          np.ogrid[1:1:2, None:10]
35: (0)          np.index_exp[0:1]
36: (0)          np.index_exp[0:1, None:3]
37: (0)          np.index_exp[0, 0:1, ..., [0, 1, 3]]
38: (0)          np.s_[0:1]
39: (0)          np.s_[0:1, None:3]
40: (0)          np.s_[0, 0:1, ..., [0, 1, 3]]
41: (0)          np.ix_(AR_LIKE_b[0])
42: (0)          np.ix_(AR_LIKE_i[0], AR_LIKE_f[0])
43: (0)          np.ix_(AR_i8[0])
44: (0)          np.fill_diagonal(AR_i8, 5)
45: (0)          np.diag_indices(4)
46: (0)          np.diag_indices(2, 3)
47: (0)          np.diag_indices_from(AR_i8)

```

---

File 366 - lib\_version.py:

```

1: (0)          from numpy.lib import NumpyVersion
2: (0)          version = NumpyVersion("1.8.0")
3: (0)          version.vstring
4: (0)          version.version
5: (0)          version.major
6: (0)          version.minor
7: (0)          version.bugfix
8: (0)          version.pre_release
9: (0)          version.is_devversion
10: (0)         version == version
11: (0)         version != version
12: (0)         version < "1.8.0"
13: (0)         version <= version
14: (0)         version > version
15: (0)         version >= "1.8.0"

```

---

File 367 - literal.py:

```

1: (0)          from __future__ import annotations
2: (0)          from functools import partial
3: (0)          from collections.abc import Callable
4: (0)          import pytest # type: ignore
5: (0)          import numpy as np
6: (0)          AR = np.array(0)
7: (0)          AR.setflags(write=False)
8: (0)          KACF = frozenset({None, "K", "A", "C", "F"})
9: (0)          ACF = frozenset({None, "A", "C", "F"})
10: (0)         CF = frozenset({None, "C", "F"})
11: (0)         order_list: list[tuple[frozenset, Callable]] = [
12: (4)             (KACF, partial(np.ndarray, 1)),
13: (4)             (KACF, AR.tobytes),
14: (4)             (KACF, partial(AR.astype, int)),
15: (4)             (KACF, AR.copy),
16: (4)             (ACF, partial(AR.reshape, 1)),
17: (4)             (KACF, AR.flatten),
18: (4)             (KACF, AR.ravel),
19: (4)             (KACF, partial(np.array, 1)),
20: (4)             (CF, partial(np.zeros, 1)),
21: (4)             (CF, partial(np.ones, 1)),
22: (4)             (CF, partial(np.empty, 1)),
23: (4)             (CF, partial(np.full, 1, 1)),
24: (4)             (KACF, partial(np.zeros_like, AR)),
25: (4)             (KACF, partial(np.ones_like, AR)),
26: (4)             (KACF, partial(np.empty_like, AR)),
27: (4)             (KACF, partial(np.full_like, AR, 1)),
28: (4)             (KACF, partial(np.add, 1, 1)), # i.e. np.ufunc.__call__
29: (4)             (ACF, partial(np.reshape, AR, 1)),
30: (4)             (KACF, partial(np.ravel, AR)),

```

```

31: (4)                 (KACF, partial(np.asarray, 1)),
32: (4)                 (KACF, partial(np.asanyarray, 1)),
33: (0)
34: (0)             for order_set, func in order_list:
35: (4)                 for order in order_set:
36: (8)                     func(order=order)
37: (4)                     invalid_orders = KACF - order_set
38: (4)                     for order in invalid_orders:
39: (8)                         with pytest.raises(ValueError):
40: (12)                             func(order=order)

-----

```

File 368 - mod.py:

```

1: (0)                 import numpy as np
2: (0)                 f8 = np.float64(1)
3: (0)                 i8 = np.int64(1)
4: (0)                 u8 = np.uint64(1)
5: (0)                 f4 = np.float32(1)
6: (0)                 i4 = np.int32(1)
7: (0)                 u4 = np.uint32(1)
8: (0)                 td = np.timedelta64(1, "D")
9: (0)                 b_ = np.bool_(1)
10: (0)                b = bool(1)
11: (0)                f = float(1)
12: (0)                i = int(1)
13: (0)                AR = np.array([1], dtype=np.bool_)
14: (0)                AR.setflags(write=False)
15: (0)                AR2 = np.array([1], dtype=np.timedelta64)
16: (0)                AR2.setflags(write=False)
17: (0)                td % td
18: (0)                td % AR2
19: (0)                AR2 % td
20: (0)                divmod(td, td)
21: (0)                divmod(td, AR2)
22: (0)                divmod(AR2, td)
23: (0)                b_ % b
24: (0)                b_ % i
25: (0)                b_ % f
26: (0)                b_ % b_
27: (0)                b_ % i8
28: (0)                b_ % u8
29: (0)                b_ % f8
30: (0)                b_ % AR
31: (0)                divmod(b_, b)
32: (0)                divmod(b_, i)
33: (0)                divmod(b_, f)
34: (0)                divmod(b_, b_)
35: (0)                divmod(b_, i8)
36: (0)                divmod(b_, u8)
37: (0)                divmod(b_, f8)
38: (0)                divmod(b_, AR)
39: (0)                b % b_
40: (0)                i % b_
41: (0)                f % b_
42: (0)                b_ % b_
43: (0)                i8 % b_
44: (0)                u8 % b_
45: (0)                f8 % b_
46: (0)                AR % b_
47: (0)                divmod(b, b_)
48: (0)                divmod(i, b_)
49: (0)                divmod(f, b_)
50: (0)                divmod(b_, b_)
51: (0)                divmod(i8, b_)
52: (0)                divmod(u8, b_)
53: (0)                divmod(f8, b_)
54: (0)                divmod(AR, b_)


```

```
55: (0)          i8 % b
56: (0)          i8 % i
57: (0)          i8 % f
58: (0)          i8 % i8
59: (0)          i8 % f8
60: (0)          i4 % i8
61: (0)          i4 % f8
62: (0)          i4 % i4
63: (0)          i4 % f4
64: (0)          i8 % AR
65: (0)          divmod(i8, b)
66: (0)          divmod(i8, i)
67: (0)          divmod(i8, f)
68: (0)          divmod(i8, i8)
69: (0)          divmod(i8, f8)
70: (0)          divmod(i8, i4)
71: (0)          divmod(i8, f4)
72: (0)          divmod(i4, i4)
73: (0)          divmod(i4, f4)
74: (0)          divmod(i8, AR)
75: (0)          b % i8
76: (0)          i % i8
77: (0)          f % i8
78: (0)          i8 % i8
79: (0)          f8 % i8
80: (0)          i8 % i4
81: (0)          f8 % i4
82: (0)          i4 % i4
83: (0)          f4 % i4
84: (0)          AR % i8
85: (0)          divmod(b, i8)
86: (0)          divmod(i, i8)
87: (0)          divmod(f, i8)
88: (0)          divmod(i8, i8)
89: (0)          divmod(f8, i8)
90: (0)          divmod(i4, i8)
91: (0)          divmod(f4, i8)
92: (0)          divmod(i4, i4)
93: (0)          divmod(f4, i4)
94: (0)          divmod(AR, i8)
95: (0)          f8 % b
96: (0)          f8 % i
97: (0)          f8 % f
98: (0)          i8 % f4
99: (0)          f4 % f4
100: (0)         f8 % AR
101: (0)         divmod(f8, b)
102: (0)         divmod(f8, i)
103: (0)         divmod(f8, f)
104: (0)         divmod(f8, f8)
105: (0)         divmod(f8, f4)
106: (0)         divmod(f4, f4)
107: (0)         divmod(f8, AR)
108: (0)         b % f8
109: (0)         i % f8
110: (0)         f % f8
111: (0)         f8 % f8
112: (0)         f8 % f8
113: (0)         f4 % f4
114: (0)         AR % f8
115: (0)         divmod(b, f8)
116: (0)         divmod(i, f8)
117: (0)         divmod(f, f8)
118: (0)         divmod(f8, f8)
119: (0)         divmod(f4, f8)
120: (0)         divmod(f4, f4)
121: (0)         divmod(AR, f8)
```

## File 369 - modules.py:

```

1: (0)          import numpy as np
2: (0)          from numpy import f2py
3: (0)          np.char
4: (0)          np.ctypeslib
5: (0)          np.emath
6: (0)          np.fft
7: (0)          np.lib
8: (0)          np.linalg
9: (0)          np.ma
10: (0)         np.matrixlib
11: (0)         np.polynomial
12: (0)         np.random
13: (0)         np.rec
14: (0)         np.testing
15: (0)         np.version
16: (0)         np.lib.format
17: (0)         np.lib.mixins
18: (0)         np.lib.scimath
19: (0)         np.lib.stride_tricks
20: (0)         np.ma.extras
21: (0)         np.polynomial.chebyshev
22: (0)         np.polynomial.hermite
23: (0)         np.polynomial.hermite_e
24: (0)         np.polynomial.laguerre
25: (0)         np.polynomial.legendre
26: (0)         np.polynomial.polynomial
27: (0)         np.__path__
28: (0)         np.__version__
29: (0)         np.__all__
30: (0)         np.char.__all__
31: (0)         np.ctypeslib.__all__
32: (0)         np.emath.__all__
33: (0)         np.lib.__all__
34: (0)         np.ma.__all__
35: (0)         np.random.__all__
36: (0)         np.rec.__all__
37: (0)         np.testing.__all__
38: (0)         f2py.__all__
-----
```

## File 370 - multiarray.py:

```

1: (0)          import numpy as np
2: (0)          import numpy.typing as npt
3: (0)          AR_f8: npt.NDArray[np.float64] = np.array([1.0])
4: (0)          AR_i4 = np.array([1], dtype=np.int32)
5: (0)          AR_u1 = np.array([1], dtype=np.uint8)
6: (0)          AR_LIKE_f = [1.5]
7: (0)          AR_LIKE_i = [1]
8: (0)          b_f8 = np.broadcast(AR_f8)
9: (0)          b_i4_f8_f8 = np.broadcast(AR_i4, AR_f8, AR_f8)
10: (0)         next(b_f8)
11: (0)         b_f8.reset()
12: (0)         b_f8.index
13: (0)         b_f8.iters
14: (0)         b_f8.ndim
15: (0)         b_f8.ndim
16: (0)         b_f8.numiter
17: (0)         b_f8.shape
18: (0)         b_f8.size
19: (0)         next(b_i4_f8_f8)
20: (0)         b_i4_f8_f8.reset()
21: (0)         b_i4_f8_f8.ndim
22: (0)         b_i4_f8_f8.index
23: (0)         b_i4_f8_f8.iters
```

```

24: (0)          b_i4_f8_f8.nd
25: (0)          b_i4_f8_f8.numiter
26: (0)          b_i4_f8_f8.shape
27: (0)          b_i4_f8_f8.size
28: (0)          np.inner(AR_f8, AR_i4)
29: (0)          np.where([True, True, False])
30: (0)          np.where([True, True, False], 1, 0)
31: (0)          np.lexsort([0, 1, 2])
32: (0)          np.can_cast(np.dtype("i8"), int)
33: (0)          np.can_cast(AR_f8, "f8")
34: (0)          np.can_cast(AR_f8, np.complex128, casting="unsafe")
35: (0)          np.min_scalar_type([1])
36: (0)          np.min_scalar_type(AR_f8)
37: (0)          np.result_type(int, AR_i4)
38: (0)          np.result_type(AR_f8, AR_u1)
39: (0)          np.result_type(AR_f8, np.complex128)
40: (0)          np.dot(AR_LIKE_f, AR_i4)
41: (0)          np.dot(AR_u1, 1)
42: (0)          np.dot(1.5j, 1)
43: (0)          np.dot(AR_u1, 1, out=AR_f8)
44: (0)          np.vdot(AR_LIKE_f, AR_i4)
45: (0)          np.vdot(AR_u1, 1)
46: (0)          np.vdot(1.5j, 1)
47: (0)          np.bincount(AR_i4)
48: (0)          np.copyto(AR_f8, [1.6])
49: (0)          np.putmask(AR_f8, [True], 1.5)
50: (0)          np.packbits(AR_i4)
51: (0)          np.packbits(AR_u1)
52: (0)          np.unpackbits(AR_u1)
53: (0)          np.shares_memory(1, 2)
54: (0)          np.shares_memory(AR_f8, AR_f8, max_work=1)
55: (0)          np.may_share_memory(1, 2)
56: (0)          np.may_share_memory(AR_f8, AR_f8, max_work=1)

```

---

## File 371 - ndarray\_conversion.py:

```

1: (0)          import os
2: (0)          import tempfile
3: (0)          import numpy as np
4: (0)          nd = np.array([[1, 2], [3, 4]])
5: (0)          scalar_array = np.array(1)
6: (0)          scalar_array.item()
7: (0)          nd.item(1)
8: (0)          nd.item(0, 1)
9: (0)          nd.item((0, 1))
10: (0)         scalar_array.itemset(3)
11: (0)         nd.itemset(3, 0)
12: (0)         nd.itemset((0, 0), 3)
13: (0)         nd.tobytes()
14: (0)         nd.tobytes("C")
15: (0)         nd.tobytes(None)
16: (0)         if os.name != "nt":
17: (4)             with tempfile.NamedTemporaryFile(suffix=".txt") as tmp:
18: (8)                 nd.tofile(tmp.name)
19: (8)                 nd.tofile(tmp.name, "")
20: (8)                 nd.tofile(tmp.name, sep="")
21: (8)                 nd.tofile(tmp.name, "", "%s")
22: (8)                 nd.tofile(tmp.name, format="%s")
23: (8)                 nd.tofile(tmp)
24: (0)                 nd.astype("float")
25: (0)                 nd.astype(float)
26: (0)                 nd.astype(float, "K")
27: (0)                 nd.astype(float, order="K")
28: (0)                 nd.astype(float, "K", "unsafe")
29: (0)                 nd.astype(float, casting="unsafe")
30: (0)                 nd.astype(float, "K", "unsafe", True)
31: (0)                 nd.astype(float, subok=True)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

32: (0)         nd.astype(float, "K", "unsafe", True, True)
33: (0)         nd.astype(float, copy=True)
34: (0)         nd.byteswap()
35: (0)         nd.byteswap(True)
36: (0)         nd.copy()
37: (0)         nd.copy("C")
38: (0)         nd.view()
39: (0)         nd.view(np.int64)
40: (0)         nd.view(dtype=np.int64)
41: (0)         nd.view(np.int64, np.matrix)
42: (0)         nd.view(type=np.matrix)
43: (0)         complex_array = np.array([[1 + 1j, 0], [0, 1 - 1j]], dtype=np.complex128)
44: (0)         complex_array.getfield("float")
45: (0)         complex_array.getfield(float)
46: (0)         complex_array.getfield("float", 8)
47: (0)         complex_array.getfield(float, offset=8)
48: (0)         nd.setflags()
49: (0)         nd.setflags(True)
50: (0)         nd.setflags(write=True)
51: (0)         nd.setflags(True, True)
52: (0)         nd.setflags(write=True, align=True)
53: (0)         nd.setflags(True, True, False)
54: (0)         nd.setflags(write=True, align=True, uic=False)

```

---

## File 372 - ndarray\_misc.py:

```

1: (0)         """
2: (0)         Tests for miscellaneous (non-magic) ``np.ndarray``/``np.generic`` methods.
3: (0)         More extensive tests are performed for the methods'
4: (0)         function-based counterpart in `../from_numeric.py` .
5: (0)         """
6: (0)         from __future__ import annotations
7: (0)         import operator
8: (0)         from typing import cast, Any
9: (0)         import numpy as np
10: (0)        class SubClass(np.ndarray): ...
11: (0)        i4 = np.int32(1)
12: (0)        A: np.ndarray[Any, np.dtype[np.int32]] = np.array([[1]], dtype=np.int32)
13: (0)        B0 = np.empty((), dtype=np.int32).view(SubClass)
14: (0)        B1 = np.empty((1,), dtype=np.int32).view(SubClass)
15: (0)        B2 = np.empty((1, 1), dtype=np.int32).view(SubClass)
16: (0)        C: np.ndarray[Any, np.dtype[np.int32]] = np.array([0, 1, 2], dtype=np.int32)
17: (0)        D = np.ones(3).view(SubClass)
18: (0)        i4.all()
19: (0)        A.all()
20: (0)        A.all(axis=0)
21: (0)        A.all(keepdims=True)
22: (0)        A.all(out=B0)
23: (0)        i4.any()
24: (0)        A.any()
25: (0)        A.any(axis=0)
26: (0)        A.any(keepdims=True)
27: (0)        A.any(out=B0)
28: (0)        i4.argmax()
29: (0)        A.argmax()
30: (0)        A.argmax(axis=0)
31: (0)        A.argmax(out=B0)
32: (0)        i4.argmin()
33: (0)        A.argmin()
34: (0)        A.argmin(axis=0)
35: (0)        A.argmin(out=B0)
36: (0)        i4.argsort()
37: (0)        A.argsort()
38: (0)        i4.choose([()])
39: (0)        _choices = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]], dtype=np.int32)
40: (0)        C.choose(_choices)
41: (0)        C.choose(_choices, out=D)

```

```
42: (0)          i4.clip(1)
43: (0)          A.clip(1)
44: (0)          A.clip(None, 1)
45: (0)          A.clip(1, out=B2)
46: (0)          A.clip(None, 1, out=B2)
47: (0)          i4.compress([1])
48: (0)          A.compress([1])
49: (0)          A.compress([1], out=B1)
50: (0)          i4.conj()
51: (0)          A.conj()
52: (0)          B0.conj()
53: (0)          i4.conjugate()
54: (0)          A.conjugate()
55: (0)          B0.conjugate()
56: (0)          i4.cumprod()
57: (0)          A.cumprod()
58: (0)          A.cumprod(out=B1)
59: (0)          i4.cumsum()
60: (0)          A.cumsum()
61: (0)          A.cumsum(out=B1)
62: (0)          i4.max()
63: (0)          A.max()
64: (0)          A.max(axis=0)
65: (0)          A.max(keepdims=True)
66: (0)          A.max(out=B0)
67: (0)          i4.mean()
68: (0)          A.mean()
69: (0)          A.mean(axis=0)
70: (0)          A.mean(keepdims=True)
71: (0)          A.mean(out=B0)
72: (0)          i4.min()
73: (0)          A.min()
74: (0)          A.min(axis=0)
75: (0)          A.min(keepdims=True)
76: (0)          A.min(out=B0)
77: (0)          i4.newbyteorder()
78: (0)          A.newbyteorder()
79: (0)          B0.newbyteorder('|')
80: (0)          i4.prod()
81: (0)          A.prod()
82: (0)          A.prod(axis=0)
83: (0)          A.prod(keepdims=True)
84: (0)          A.prod(out=B0)
85: (0)          i4.ptp()
86: (0)          A.ptp()
87: (0)          A.ptp(axis=0)
88: (0)          A.ptp(keepdims=True)
89: (0)          A.astype(int).ptp(out=B0)
90: (0)          i4.round()
91: (0)          A.round()
92: (0)          A.round(out=B2)
93: (0)          i4.repeat(1)
94: (0)          A.repeat(1)
95: (0)          B0.repeat(1)
96: (0)          i4.std()
97: (0)          A.std()
98: (0)          A.std(axis=0)
99: (0)          A.std(keepdims=True)
100: (0)         A.std(out=B0.astype(np.float64))
101: (0)         i4.sum()
102: (0)         A.sum()
103: (0)         A.sum(axis=0)
104: (0)         A.sum(keepdims=True)
105: (0)         A.sum(out=B0)
106: (0)         i4.take(0)
107: (0)         A.take(0)
108: (0)         A.take([0])
109: (0)         A.take(0, out=B0)
110: (0)         A.take([0], out=B1)
```

```

111: (0)          i4.var()
112: (0)          A.var()
113: (0)          A.var(axis=0)
114: (0)          A.var(keepdims=True)
115: (0)          A.var(out=B0)
116: (0)          A.argmax([0])
117: (0)          A.diagonal()
118: (0)          A.dot(1)
119: (0)          A.dot(1, out=B2)
120: (0)          A.nonzero()
121: (0)          C.searchsorted(1)
122: (0)          A.trace()
123: (0)          A.trace(out=B0)
124: (0)          void = cast(np.void, np.array(1, dtype=[("f", np.float64)]).take(0))
125: (0)          void.setfield(10, np.float64)
126: (0)          A.item(0)
127: (0)          C.item(0)
128: (0)          A.ravel()
129: (0)          C.ravel()
130: (0)          A.flatten()
131: (0)          C.flatten()
132: (0)          A.reshape(1)
133: (0)          C.reshape(3)
134: (0)          int(np.array(1.0, dtype=np.float64))
135: (0)          int(np.array("1", dtype=np.str_))
136: (0)          float(np.array(1.0, dtype=np.float64))
137: (0)          float(np.array("1", dtype=np.str_))
138: (0)          complex(np.array(1.0, dtype=np.float64))
139: (0)          operator.index(np.array(1, dtype=np.int64))

```

-----

## File 373 - ndarray\_shape\_manipulation.py:

```

1: (0)          import numpy as np
2: (0)          nd1 = np.array([[1, 2], [3, 4]])
3: (0)          nd1.reshape(4)
4: (0)          nd1.reshape(2, 2)
5: (0)          nd1.reshape((2, 2))
6: (0)          nd1.reshape((2, 2), order="C")
7: (0)          nd1.reshape(4, order="C")
8: (0)          nd1.resize()
9: (0)          nd1.resize(4)
10: (0)         nd1.resize(2, 2)
11: (0)         nd1.resize((2, 2))
12: (0)         nd1.resize((2, 2), refcheck=True)
13: (0)         nd1.resize(4, refcheck=True)
14: (0)         nd2 = np.array([[1, 2], [3, 4]])
15: (0)         nd2.transpose()
16: (0)         nd2.transpose(1, 0)
17: (0)         nd2.transpose((1, 0))
18: (0)         nd2.swapaxes(0, 1)
19: (0)         nd2.flatten()
20: (0)         nd2.flatten("C")
21: (0)         nd2.ravel()
22: (0)         nd2.ravel("C")
23: (0)         nd2.squeeze()
24: (0)         nd3 = np.array([[1, 2]])
25: (0)         nd3.squeeze(0)
26: (0)         nd4 = np.array([[[1, 2]]])
27: (0)         nd4.squeeze((0, 1))

```

-----

## File 374 - numeric.py:

```

1: (0)          """
2: (0)          Tests for :mod:`numpy.core.numeric`.
3: (0)          Does not include tests which fall under ``array_constructors``.

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

4: (0)
5: (0)
6: (0)
7: (0)
8: (4)
9: (0)
10: (0)
11: (0)
12: (0)
13: (0)
14: (0)
15: (0)
16: (0)
17: (0)
18: (0)
19: (0)
20: (0)
21: (0)
22: (0)
23: (0)
24: (0)
25: (0)
26: (0)
27: (0)
28: (0)
29: (0)
30: (0)
31: (0)
32: (0)
33: (0)
34: (0)
35: (0)
36: (0)
37: (0)
38: (0)
39: (0)
40: (0)
41: (0)
42: (0)
43: (0)
44: (0)
45: (0)
46: (0)
47: (0)
48: (0)
49: (0)
50: (0)
51: (0)
52: (0)
53: (0)
54: (0)
55: (0)
56: (0)
57: (0)
58: (0)
59: (0)
60: (0)
61: (0)
62: (0)
63: (0)

-----

```

## File 375 - numerictypes.py:

```

1: (0)         import numpy as np
2: (0)         np.maximum_sctype("S8")
3: (0)         np.maximum_sctype(object)
4: (0)         np.issctype(object)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

5: (0) np.issctype("S8")
6: (0) np.obj2sctype(list)
7: (0) np.obj2sctype(list, default=None)
8: (0) np.obj2sctype(list, default=np.bytes_)
9: (0) np.issubclass_(np.int32, int)
10: (0) np.issubclass_(np.float64, float)
11: (0) np.issubclass_(np.float64, (int, float))
12: (0) np.issubsctype("int64", int)
13: (0) np.issubsctype(np.array([1]), np.array([1]))
14: (0) np.issubdtype("S1", np.bytes_)
15: (0) np.issubdtype(np.float64, np.float32)
16: (0) np.sctype2char("S1")
17: (0) np.sctype2char(list)
18: (0) np.cast[int]
19: (0) np.cast["i8"]
20: (0) np.cast[np.int64]
21: (0) np.nbytes[int]
22: (0) np.nbytes["i8"]
23: (0) np.nbytes[np.int64]
24: (0) np.ScalarType
25: (0) np.ScalarType[0]
26: (0) np.ScalarType[3]
27: (0) np.ScalarType[8]
28: (0) np.ScalarType[10]
29: (0) np.typecodes["Character"]
30: (0) np.typecodes["Complex"]
31: (0) np.typecodes["All"]

```

---

## File 376 - random.py:

```

1: (0) from __future__ import annotations
2: (0) from typing import Any
3: (0) import numpy as np
4: (0) SEED_NONE = None
5: (0) SEED_INT = 4579435749574957634658964293569
6: (0) SEED_ARR: np.ndarray[Any, np.dtype[np.int64]] = np.array([1, 2, 3, 4],
dtype=np.int64)
7: (0)     SEED_ARRLIKE: list[int] = [1, 2, 3, 4]
8: (0)     SEED_SEED_SEQ: np.random.SeedSequence = np.random.SeedSequence(0)
9: (0)     SEED_MT19937: np.random.MT19937 = np.random.MT19937(0)
10: (0)     SEED_PCG64: np.random.PCG64 = np.random.PCG64(0)
11: (0)     SEED_PHILOX: np.random.Philox = np.random.Philox(0)
12: (0)     SEED_SFC64: np.random.SFC64 = np.random.SFC64(0)
13: (0)     np.random.default_rng()
14: (0)     np.random.default_rng(SEED_NONE)
15: (0)     np.random.default_rng(SEED_INT)
16: (0)     np.random.default_rng(SEED_ARR)
17: (0)     np.random.default_rng(SEED_ARRLIKE)
18: (0)     np.random.default_rng(SEED_SEED_SEQ)
19: (0)     np.random.default_rng(SEED_MT19937)
20: (0)     np.random.default_rng(SEED_PCG64)
21: (0)     np.random.default_rng(SEED_PHILOX)
22: (0)     np.random.default_rng(SEED_SFC64)
23: (0)     np.random.SeedSequence(SEED_NONE)
24: (0)     np.random.SeedSequence(SEED_INT)
25: (0)     np.random.SeedSequence(SEED_ARR)
26: (0)     np.random.SeedSequence(SEED_ARRLIKE)
27: (0)     np.random.MT19937(SEED_NONE)
28: (0)     np.random.MT19937(SEED_INT)
29: (0)     np.random.MT19937(SEED_ARR)
30: (0)     np.random.MT19937(SEED_ARRLIKE)
31: (0)     np.random.MT19937(SEED_SEED_SEQ)
32: (0)     np.random.PCG64(SEED_NONE)
33: (0)     np.random.PCG64(SEED_INT)
34: (0)     np.random.PCG64(SEED_ARR)
35: (0)     np.random.PCG64(SEED_ARRLIKE)
36: (0)     np.random.PCG64(SEED_SEED_SEQ)

```

```

37: (0) np.random.Philox(SEED_NONE)
38: (0) np.random.Philox(SEED_INT)
39: (0) np.random.Philox(SEED_ARR)
40: (0) np.random.Philox(SEED_ARRLIKE)
41: (0) np.random.Philox(SEED_SEED_SEQ)
42: (0) np.random.SFC64(SEED_NONE)
43: (0) np.random.SFC64(SEED_INT)
44: (0) np.random.SFC64(SEED_ARR)
45: (0) np.random.SFC64(SEED_ARRLIKE)
46: (0) np.random.SFC64(SEED_SEED_SEQ)
47: (0) seed_seq: np.random.bit_generator.SeedSequence =
np.random.SeedSequence(SEED_NONE)
48: (0) seed_seq.spawn(10)
49: (0) seed_seq.generate_state(3)
50: (0) seed_seq.generate_state(3, "u4")
51: (0) seed_seq.generate_state(3, "uint32")
52: (0) seed_seq.generate_state(3, "u8")
53: (0) seed_seq.generate_state(3, "uint64")
54: (0) seed_seq.generate_state(3, np.uint32)
55: (0) seed_seq.generate_state(3, np.uint64)
56: (0) def_gen: np.random.Generator = np.random.default_rng()
57: (0) D_arr_0p1: np.ndarray[Any, np.dtype[np.float64]] = np.array([0.1])
58: (0) D_arr_0p5: np.ndarray[Any, np.dtype[np.float64]] = np.array([0.5])
59: (0) D_arr_0p9: np.ndarray[Any, np.dtype[np.float64]] = np.array([0.9])
60: (0) D_arr_1p5: np.ndarray[Any, np.dtype[np.float64]] = np.array([1.5])
61: (0) I_arr_10: np.ndarray[Any, np.dtype[np.int_]] = np.array([10], dtype=np.int_)
62: (0) I_arr_20: np.ndarray[Any, np.dtype[np.int_]] = np.array([20], dtype=np.int_)
63: (0) D_arr_like_0p1: list[float] = [0.1]
64: (0) D_arr_like_0p5: list[float] = [0.5]
65: (0) D_arr_like_0p9: list[float] = [0.9]
66: (0) D_arr_like_1p5: list[float] = [1.5]
67: (0) I_arr_like_10: list[int] = [10]
68: (0) I_arr_like_20: list[int] = [20]
69: (0) D_2D_like: list[list[float]] = [[1, 2], [2, 3], [3, 4], [4, 5.1]]
70: (0) D_2D: np.ndarray[Any, np.dtype[np.float64]] = np.array(D_2D_like)
71: (0) S_out: np.ndarray[Any, np.dtype[np.float32]] = np.empty(1, dtype=np.float32)
72: (0) D_out: np.ndarray[Any, np.dtype[np.float64]] = np.empty(1)
73: (0) def_gen.standard_normal()
74: (0) def_gen.standard_normal(dtype=np.float32)
75: (0) def_gen.standard_normal(dtype="float32")
76: (0) def_gen.standard_normal(dtype="double")
77: (0) def_gen.standard_normal(dtype=np.float64)
78: (0) def_gen.standard_normal(size=None)
79: (0) def_gen.standard_normal(size=1)
80: (0) def_gen.standard_normal(size=1, dtype=np.float32)
81: (0) def_gen.standard_normal(size=1, dtype="f4")
82: (0) def_gen.standard_normal(size=1, dtype="float32", out=S_out)
83: (0) def_gen.standard_normal(dtype=np.float32, out=S_out)
84: (0) def_gen.standard_normal(size=1, dtype=np.float64)
85: (0) def_gen.standard_normal(size=1, dtype="float64")
86: (0) def_gen.standard_normal(size=1, dtype="f8")
87: (0) def_gen.standard_normal(out=D_out)
88: (0) def_gen.standard_normal(size=1, dtype="float64")
89: (0) def_gen.standard_normal(size=1, dtype="float64", out=D_out)
90: (0) def_gen.random()
91: (0) def_gen.random(dtype=np.float32)
92: (0) def_gen.random(dtype="float32")
93: (0) def_gen.random(dtype="double")
94: (0) def_gen.random(dtype=np.float64)
95: (0) def_gen.random(size=None)
96: (0) def_gen.random(size=1)
97: (0) def_gen.random(size=1, dtype=np.float32)
98: (0) def_gen.random(size=1, dtype="f4")
99: (0) def_gen.random(size=1, dtype="float32", out=S_out)
100: (0) def_gen.random(dtype=np.float32, out=S_out)
101: (0) def_gen.random(size=1, dtype=np.float64)
102: (0) def_gen.random(size=1, dtype="float64")
103: (0) def_gen.random(size=1, dtype="f8")
104: (0) def_gen.random(out=D_out)

```

```
105: (0) def_gen.random(size=1, dtype="float64")
106: (0) def_gen.random(size=1, dtype="float64", out=D_out)
107: (0) def_gen.standard_cauchy()
108: (0) def_gen.standard_cauchy(size=None)
109: (0) def_gen.standard_cauchy(size=1)
110: (0) def_gen.standard_exponential()
111: (0) def_gen.standard_exponential(method="inv")
112: (0) def_gen.standard_exponential(dtype=np.float32)
113: (0) def_gen.standard_exponential(dtype="float32")
114: (0) def_gen.standard_exponential(dtype="double")
115: (0) def_gen.standard_exponential(dtype=np.float64)
116: (0) def_gen.standard_exponential(size=None)
117: (0) def_gen.standard_exponential(size=None, method="inv")
118: (0) def_gen.standard_exponential(size=1, method="inv")
119: (0) def_gen.standard_exponential(size=1, dtype=np.float32)
120: (0) def_gen.standard_exponential(size=1, dtype="f4", method="inv")
121: (0) def_gen.standard_exponential(size=1, dtype="float32", out=S_out)
122: (0) def_gen.standard_exponential(dtype=np.float32, out=S_out)
123: (0) def_gen.standard_exponential(size=1, dtype=np.float64, method="inv")
124: (0) def_gen.standard_exponential(size=1, dtype="float64")
125: (0) def_gen.standard_exponential(size=1, dtype="f8")
126: (0) def_gen.standard_exponential(out=D_out)
127: (0) def_gen.standard_exponential(size=1, dtype="float64")
128: (0) def_gen.standard_exponential(size=1, dtype="float64", out=D_out)
129: (0) def_gen.zipf(1.5)
130: (0) def_gen.zipf(1.5, size=None)
131: (0) def_gen.zipf(1.5, size=1)
132: (0) def_gen.zipf(D_arr_1p5)
133: (0) def_gen.zipf(D_arr_1p5, size=1)
134: (0) def_gen.zipf(D_arr_like_1p5)
135: (0) def_gen.zipf(D_arr_like_1p5, size=1)
136: (0) def_gen.weibull(0.5)
137: (0) def_gen.weibull(0.5, size=None)
138: (0) def_gen.weibull(0.5, size=1)
139: (0) def_gen.weibull(D_arr_0p5)
140: (0) def_gen.weibull(D_arr_0p5, size=1)
141: (0) def_gen.weibull(D_arr_like_0p5)
142: (0) def_gen.weibull(D_arr_like_0p5, size=1)
143: (0) def_gen.standard_t(0.5)
144: (0) def_gen.standard_t(0.5, size=None)
145: (0) def_gen.standard_t(0.5, size=1)
146: (0) def_gen.standard_t(D_arr_0p5)
147: (0) def_gen.standard_t(D_arr_0p5, size=1)
148: (0) def_gen.standard_t(D_arr_like_0p5)
149: (0) def_gen.standard_t(D_arr_like_0p5, size=1)
150: (0) def_gen.poisson(0.5)
151: (0) def_gen.poisson(0.5, size=None)
152: (0) def_gen.poisson(0.5, size=1)
153: (0) def_gen.poisson(D_arr_0p5)
154: (0) def_gen.poisson(D_arr_0p5, size=1)
155: (0) def_gen.poisson(D_arr_like_0p5)
156: (0) def_gen.poisson(D_arr_like_0p5, size=1)
157: (0) def_gen.power(0.5)
158: (0) def_gen.power(0.5, size=None)
159: (0) def_gen.power(0.5, size=1)
160: (0) def_gen.power(D_arr_0p5)
161: (0) def_gen.power(D_arr_0p5, size=1)
162: (0) def_gen.power(D_arr_like_0p5)
163: (0) def_gen.power(D_arr_like_0p5, size=1)
164: (0) def_gen.pareto(0.5)
165: (0) def_gen.pareto(0.5, size=None)
166: (0) def_gen.pareto(0.5, size=1)
167: (0) def_gen.pareto(D_arr_0p5)
168: (0) def_gen.pareto(D_arr_0p5, size=1)
169: (0) def_gen.pareto(D_arr_like_0p5)
170: (0) def_gen.pareto(D_arr_like_0p5, size=1)
171: (0) def_gen.chisquare(0.5)
172: (0) def_gen.chisquare(0.5, size=None)
173: (0) def_gen.chisquare(0.5, size=1)
```

```
174: (0) def_gen.chisquare(D_arr_0p5)
175: (0) def_gen.chisquare(D_arr_0p5, size=1)
176: (0) def_gen.chisquare(D_arr_like_0p5)
177: (0) def_gen.chisquare(D_arr_like_0p5, size=1)
178: (0) def_gen.exponential(0.5)
179: (0) def_gen.exponential(0.5, size=None)
180: (0) def_gen.exponential(0.5, size=1)
181: (0) def_gen.exponential(D_arr_0p5)
182: (0) def_gen.exponential(D_arr_0p5, size=1)
183: (0) def_gen.exponential(D_arr_like_0p5)
184: (0) def_gen.exponential(D_arr_like_0p5, size=1)
185: (0) def_gen.geometric(0.5)
186: (0) def_gen.geometric(0.5, size=None)
187: (0) def_gen.geometric(0.5, size=1)
188: (0) def_gen.geometric(D_arr_0p5)
189: (0) def_gen.geometric(D_arr_0p5, size=1)
190: (0) def_gen.geometric(D_arr_like_0p5)
191: (0) def_gen.geometric(D_arr_like_0p5, size=1)
192: (0) def_gen.logseries(0.5)
193: (0) def_gen.logseries(0.5, size=None)
194: (0) def_gen.logseries(0.5, size=1)
195: (0) def_gen.logseries(D_arr_0p5)
196: (0) def_gen.logseries(D_arr_0p5, size=1)
197: (0) def_gen.logseries(D_arr_like_0p5)
198: (0) def_gen.logseries(D_arr_like_0p5, size=1)
199: (0) def_gen.rayleigh(0.5)
200: (0) def_gen.rayleigh(0.5, size=None)
201: (0) def_gen.rayleigh(0.5, size=1)
202: (0) def_gen.rayleigh(D_arr_0p5)
203: (0) def_gen.rayleigh(D_arr_0p5, size=1)
204: (0) def_gen.rayleigh(D_arr_like_0p5)
205: (0) def_gen.rayleigh(D_arr_like_0p5, size=1)
206: (0) def_gen.standard_gamma(0.5)
207: (0) def_gen.standard_gamma(0.5, size=None)
208: (0) def_gen.standard_gamma(0.5, dtype="float32")
209: (0) def_gen.standard_gamma(0.5, size=None, dtype="float32")
210: (0) def_gen.standard_gamma(0.5, size=1)
211: (0) def_gen.standard_gamma(D_arr_0p5)
212: (0) def_gen.standard_gamma(D_arr_0p5, dtype="f4")
213: (0) def_gen.standard_gamma(0.5, size=1, dtype="float32", out=S_out)
214: (0) def_gen.standard_gamma(D_arr_0p5, dtype=np.float32, out=S_out)
215: (0) def_gen.standard_gamma(D_arr_0p5, size=1)
216: (0) def_gen.standard_gamma(D_arr_like_0p5)
217: (0) def_gen.standard_gamma(D_arr_like_0p5, size=1)
218: (0) def_gen.standard_gamma(0.5, out=D_out)
219: (0) def_gen.standard_gamma(D_arr_like_0p5, out=D_out)
220: (0) def_gen.standard_gamma(D_arr_like_0p5, size=1)
221: (0) def_gen.standard_gamma(D_arr_like_0p5, size=1, out=D_out, dtype=np.float64)
222: (0) def_gen.vonmises(0.5, 0.5)
223: (0) def_gen.vonmises(0.5, 0.5, size=None)
224: (0) def_gen.vonmises(0.5, 0.5, size=1)
225: (0) def_gen.vonmises(D_arr_0p5, 0.5)
226: (0) def_gen.vonmises(0.5, D_arr_0p5)
227: (0) def_gen.vonmises(D_arr_0p5, 0.5, size=1)
228: (0) def_gen.vonmises(0.5, D_arr_0p5, size=1)
229: (0) def_gen.vonmises(D_arr_like_0p5, 0.5)
230: (0) def_gen.vonmises(0.5, D_arr_like_0p5)
231: (0) def_gen.vonmises(D_arr_0p5, D_arr_0p5)
232: (0) def_gen.vonmises(D_arr_like_0p5, D_arr_like_0p5)
233: (0) def_gen.vonmises(D_arr_0p5, D_arr_0p5, size=1)
234: (0) def_gen.vonmises(D_arr_like_0p5, D_arr_like_0p5, size=1)
235: (0) def_gen.wald(0.5, 0.5)
236: (0) def_gen.wald(0.5, 0.5, size=None)
237: (0) def_gen.wald(0.5, 0.5, size=1)
238: (0) def_gen.wald(D_arr_0p5, 0.5)
239: (0) def_gen.wald(0.5, D_arr_0p5)
240: (0) def_gen.wald(D_arr_0p5, 0.5, size=1)
241: (0) def_gen.wald(0.5, D_arr_0p5, size=1)
242: (0) def_gen.wald(D_arr_like_0p5, 0.5)
```

```
243: (0) def_gen.wald(0.5, D_arr_like_0p5)
244: (0) def_gen.wald(D_arr_0p5, D_arr_0p5)
245: (0) def_gen.wald(D_arr_like_0p5, D_arr_like_0p5)
246: (0) def_gen.wald(D_arr_0p5, D_arr_0p5, size=1)
247: (0) def_gen.wald(D_arr_like_0p5, D_arr_like_0p5, size=1)
248: (0) def_gen.uniform(0.5, 0.5)
249: (0) def_gen.uniform(0.5, 0.5, size=None)
250: (0) def_gen.uniform(0.5, 0.5, size=1)
251: (0) def_gen.uniform(D_arr_0p5, 0.5)
252: (0) def_gen.uniform(0.5, D_arr_0p5)
253: (0) def_gen.uniform(D_arr_0p5, 0.5, size=1)
254: (0) def_gen.uniform(0.5, D_arr_0p5, size=1)
255: (0) def_gen.uniform(D_arr_like_0p5, 0.5)
256: (0) def_gen.uniform(0.5, D_arr_like_0p5)
257: (0) def_gen.uniform(D_arr_0p5, D_arr_0p5)
258: (0) def_gen.uniform(D_arr_like_0p5, D_arr_like_0p5)
259: (0) def_gen.uniform(D_arr_0p5, D_arr_0p5, size=1)
260: (0) def_gen.uniform(D_arr_like_0p5, D_arr_like_0p5, size=1)
261: (0) def_gen.beta(0.5, 0.5)
262: (0) def_gen.beta(0.5, 0.5, size=None)
263: (0) def_gen.beta(0.5, 0.5, size=1)
264: (0) def_gen.beta(D_arr_0p5, 0.5)
265: (0) def_gen.beta(0.5, D_arr_0p5)
266: (0) def_gen.beta(D_arr_0p5, 0.5, size=1)
267: (0) def_gen.beta(0.5, D_arr_0p5, size=1)
268: (0) def_gen.beta(D_arr_like_0p5, 0.5)
269: (0) def_gen.beta(0.5, D_arr_like_0p5)
270: (0) def_gen.beta(D_arr_0p5, D_arr_0p5)
271: (0) def_gen.beta(D_arr_like_0p5, D_arr_like_0p5)
272: (0) def_gen.beta(D_arr_0p5, D_arr_0p5, size=1)
273: (0) def_gen.beta(D_arr_like_0p5, D_arr_like_0p5, size=1)
274: (0) def_gen.f(0.5, 0.5)
275: (0) def_gen.f(0.5, 0.5, size=None)
276: (0) def_gen.f(0.5, 0.5, size=1)
277: (0) def_gen.f(D_arr_0p5, 0.5)
278: (0) def_gen.f(0.5, D_arr_0p5)
279: (0) def_gen.f(D_arr_0p5, 0.5, size=1)
280: (0) def_gen.f(0.5, D_arr_0p5, size=1)
281: (0) def_gen.f(D_arr_like_0p5, 0.5)
282: (0) def_gen.f(0.5, D_arr_like_0p5)
283: (0) def_gen.f(D_arr_0p5, D_arr_0p5)
284: (0) def_gen.f(D_arr_like_0p5, D_arr_like_0p5)
285: (0) def_gen.f(D_arr_0p5, D_arr_0p5, size=1)
286: (0) def_gen.f(D_arr_like_0p5, D_arr_like_0p5, size=1)
287: (0) def_gen.gamma(0.5, 0.5)
288: (0) def_gen.gamma(0.5, 0.5, size=None)
289: (0) def_gen.gamma(0.5, 0.5, size=1)
290: (0) def_gen.gamma(D_arr_0p5, 0.5)
291: (0) def_gen.gamma(0.5, D_arr_0p5)
292: (0) def_gen.gamma(D_arr_0p5, 0.5, size=1)
293: (0) def_gen.gamma(0.5, D_arr_0p5, size=1)
294: (0) def_gen.gamma(D_arr_like_0p5, 0.5)
295: (0) def_gen.gamma(0.5, D_arr_like_0p5)
296: (0) def_gen.gamma(D_arr_0p5, D_arr_0p5)
297: (0) def_gen.gamma(D_arr_like_0p5, D_arr_like_0p5)
298: (0) def_gen.gamma(D_arr_0p5, D_arr_0p5, size=1)
299: (0) def_gen.gamma(D_arr_like_0p5, D_arr_like_0p5, size=1)
300: (0) def_gen.gumbel(0.5, 0.5)
301: (0) def_gen.gumbel(0.5, 0.5, size=None)
302: (0) def_gen.gumbel(0.5, 0.5, size=1)
303: (0) def_gen.gumbel(D_arr_0p5, 0.5)
304: (0) def_gen.gumbel(0.5, D_arr_0p5)
305: (0) def_gen.gumbel(D_arr_0p5, 0.5, size=1)
306: (0) def_gen.gumbel(0.5, D_arr_0p5, size=1)
307: (0) def_gen.gumbel(D_arr_like_0p5, 0.5)
308: (0) def_gen.gumbel(0.5, D_arr_like_0p5)
309: (0) def_gen.gumbel(D_arr_0p5, D_arr_0p5)
310: (0) def_gen.gumbel(D_arr_like_0p5, D_arr_like_0p5)
311: (0) def_gen.gumbel(D_arr_0p5, D_arr_0p5, size=1)
```

```
312: (0) def_gen.gumbel(D_arr_like_0p5, D_arr_like_0p5, size=1)
313: (0) def_gen.laplace(0.5, 0.5)
314: (0) def_gen.laplace(0.5, 0.5, size=None)
315: (0) def_gen.laplace(0.5, 0.5, size=1)
316: (0) def_gen.laplace(D_arr_0p5, 0.5)
317: (0) def_gen.laplace(0.5, D_arr_0p5)
318: (0) def_gen.laplace(D_arr_0p5, 0.5, size=1)
319: (0) def_gen.laplace(0.5, D_arr_0p5, size=1)
320: (0) def_gen.laplace(D_arr_like_0p5, 0.5)
321: (0) def_gen.laplace(0.5, D_arr_like_0p5)
322: (0) def_gen.laplace(D_arr_0p5, D_arr_0p5)
323: (0) def_gen.laplace(D_arr_like_0p5, D_arr_like_0p5)
324: (0) def_gen.laplace(D_arr_0p5, D_arr_0p5, size=1)
325: (0) def_gen.laplace(D_arr_like_0p5, D_arr_like_0p5, size=1)
326: (0) def_gen.logistic(0.5, 0.5)
327: (0) def_gen.logistic(0.5, 0.5, size=None)
328: (0) def_gen.logistic(0.5, 0.5, size=1)
329: (0) def_gen.logistic(D_arr_0p5, 0.5)
330: (0) def_gen.logistic(0.5, D_arr_0p5)
331: (0) def_gen.logistic(D_arr_0p5, 0.5, size=1)
332: (0) def_gen.logistic(0.5, D_arr_0p5, size=1)
333: (0) def_gen.logistic(D_arr_like_0p5, 0.5)
334: (0) def_gen.logistic(0.5, D_arr_like_0p5)
335: (0) def_gen.logistic(D_arr_0p5, D_arr_0p5)
336: (0) def_gen.logistic(D_arr_like_0p5, D_arr_like_0p5)
337: (0) def_gen.logistic(D_arr_0p5, D_arr_0p5, size=1)
338: (0) def_gen.logistic(D_arr_like_0p5, D_arr_like_0p5, size=1)
339: (0) def_gen.lognormal(0.5, 0.5)
340: (0) def_gen.lognormal(0.5, 0.5, size=None)
341: (0) def_gen.lognormal(0.5, 0.5, size=1)
342: (0) def_gen.lognormal(D_arr_0p5, 0.5)
343: (0) def_gen.lognormal(0.5, D_arr_0p5)
344: (0) def_gen.lognormal(D_arr_0p5, 0.5, size=1)
345: (0) def_gen.lognormal(0.5, D_arr_0p5, size=1)
346: (0) def_gen.lognormal(D_arr_like_0p5, 0.5)
347: (0) def_gen.lognormal(0.5, D_arr_like_0p5)
348: (0) def_gen.lognormal(D_arr_0p5, D_arr_0p5)
349: (0) def_gen.lognormal(D_arr_like_0p5, D_arr_like_0p5)
350: (0) def_gen.lognormal(D_arr_0p5, D_arr_0p5, size=1)
351: (0) def_gen.lognormal(D_arr_like_0p5, D_arr_like_0p5, size=1)
352: (0) def_gen.noncentral_chisquare(0.5, 0.5)
353: (0) def_gen.noncentral_chisquare(0.5, 0.5, size=None)
354: (0) def_gen.noncentral_chisquare(0.5, 0.5, size=1)
355: (0) def_gen.noncentral_chisquare(D_arr_0p5, 0.5)
356: (0) def_gen.noncentral_chisquare(0.5, D_arr_0p5)
357: (0) def_gen.noncentral_chisquare(D_arr_0p5, 0.5, size=1)
358: (0) def_gen.noncentral_chisquare(0.5, D_arr_0p5, size=1)
359: (0) def_gen.noncentral_chisquare(D_arr_like_0p5, 0.5)
360: (0) def_gen.noncentral_chisquare(0.5, D_arr_like_0p5)
361: (0) def_gen.noncentral_chisquare(D_arr_0p5, D_arr_0p5)
362: (0) def_gen.noncentral_chisquare(D_arr_like_0p5, D_arr_like_0p5)
363: (0) def_gen.noncentral_chisquare(D_arr_0p5, D_arr_0p5, size=1)
364: (0) def_gen.noncentral_chisquare(D_arr_like_0p5, D_arr_like_0p5, size=1)
365: (0) def_gen.normal(0.5, 0.5)
366: (0) def_gen.normal(0.5, 0.5, size=None)
367: (0) def_gen.normal(0.5, 0.5, size=1)
368: (0) def_gen.normal(D_arr_0p5, 0.5)
369: (0) def_gen.normal(0.5, D_arr_0p5)
370: (0) def_gen.normal(D_arr_0p5, 0.5, size=1)
371: (0) def_gen.normal(0.5, D_arr_0p5, size=1)
372: (0) def_gen.normal(D_arr_like_0p5, 0.5)
373: (0) def_gen.normal(0.5, D_arr_like_0p5)
374: (0) def_gen.normal(D_arr_0p5, D_arr_0p5)
375: (0) def_gen.normal(D_arr_like_0p5, D_arr_like_0p5)
376: (0) def_gen.normal(D_arr_0p5, D_arr_0p5, size=1)
377: (0) def_gen.normal(D_arr_like_0p5, D_arr_like_0p5, size=1)
378: (0) def_gen.triangular(0.1, 0.5, 0.9)
379: (0) def_gen.triangular(0.1, 0.5, 0.9, size=None)
380: (0) def_gen.triangular(0.1, 0.5, 0.9, size=1)
```

```

381: (0) def_gen.triangular(D_arr_0p1, 0.5, 0.9)
382: (0) def_gen.triangular(0.1, D_arr_0p5, 0.9)
383: (0) def_gen.triangular(D_arr_0p1, 0.5, D_arr_like_0p9, size=1)
384: (0) def_gen.triangular(0.1, D_arr_0p5, 0.9, size=1)
385: (0) def_gen.triangular(D_arr_like_0p1, 0.5, D_arr_0p9)
386: (0) def_gen.triangular(0.5, D_arr_like_0p5, 0.9)
387: (0) def_gen.triangular(D_arr_0p1, D_arr_0p5, 0.9)
388: (0) def_gen.triangular(D_arr_like_0p1, D_arr_like_0p5, 0.9)
389: (0) def_gen.triangular(D_arr_0p1, D_arr_0p5, D_arr_0p9, size=1)
390: (0) def_gen.triangular(D_arr_like_0p1, D_arr_like_0p5, D_arr_like_0p9, size=1)
391: (0) def_gen.noncentral_f(0.1, 0.5, 0.9)
392: (0) def_gen.noncentral_f(0.1, 0.5, 0.9, size=None)
393: (0) def_gen.noncentral_f(0.1, 0.5, 0.9, size=1)
394: (0) def_gen.noncentral_f(D_arr_0p1, 0.5, 0.9)
395: (0) def_gen.noncentral_f(0.1, D_arr_0p5, 0.9)
396: (0) def_gen.noncentral_f(D_arr_0p1, 0.5, D_arr_like_0p9, size=1)
397: (0) def_gen.noncentral_f(0.1, D_arr_0p5, 0.9, size=1)
398: (0) def_gen.noncentral_f(D_arr_like_0p1, 0.5, D_arr_0p9)
399: (0) def_gen.noncentral_f(0.5, D_arr_like_0p5, 0.9)
400: (0) def_gen.noncentral_f(D_arr_0p1, D_arr_0p5, 0.9)
401: (0) def_gen.noncentral_f(D_arr_like_0p1, D_arr_like_0p5, 0.9)
402: (0) def_gen.noncentral_f(D_arr_0p1, D_arr_0p5, D_arr_0p9, size=1)
403: (0) def_gen.noncentral_f(D_arr_like_0p1, D_arr_like_0p5, D_arr_like_0p9, size=1)
404: (0) def_gen.binomial(10, 0.5)
405: (0) def_gen.binomial(10, 0.5, size=None)
406: (0) def_gen.binomial(10, 0.5, size=1)
407: (0) def_gen.binomial(I_arr_10, 0.5)
408: (0) def_gen.binomial(10, D_arr_0p5)
409: (0) def_gen.binomial(I_arr_10, 0.5, size=1)
410: (0) def_gen.binomial(10, D_arr_0p5, size=1)
411: (0) def_gen.binomial(I_arr_like_10, 0.5)
412: (0) def_gen.binomial(10, D_arr_like_0p5)
413: (0) def_gen.binomial(I_arr_10, D_arr_0p5)
414: (0) def_gen.binomial(I_arr_like_10, D_arr_like_0p5)
415: (0) def_gen.binomial(I_arr_10, D_arr_0p5, size=1)
416: (0) def_gen.binomial(I_arr_like_10, D_arr_like_0p5, size=1)
417: (0) def_gen.negative_binomial(10, 0.5)
418: (0) def_gen.negative_binomial(10, 0.5, size=None)
419: (0) def_gen.negative_binomial(10, 0.5, size=1)
420: (0) def_gen.negative_binomial(I_arr_10, 0.5)
421: (0) def_gen.negative_binomial(10, D_arr_0p5)
422: (0) def_gen.negative_binomial(I_arr_10, 0.5, size=1)
423: (0) def_gen.negative_binomial(10, D_arr_0p5, size=1)
424: (0) def_gen.negative_binomial(I_arr_like_10, 0.5)
425: (0) def_gen.negative_binomial(10, D_arr_like_0p5)
426: (0) def_gen.negative_binomial(I_arr_10, D_arr_0p5)
427: (0) def_gen.negative_binomial(I_arr_like_10, D_arr_like_0p5)
428: (0) def_gen.negative_binomial(I_arr_10, D_arr_0p5, size=1)
429: (0) def_gen.negative_binomial(I_arr_like_10, D_arr_like_0p5, size=1)
430: (0) def_gen.hypergeometric(20, 20, 10)
431: (0) def_gen.hypergeometric(20, 20, 10, size=None)
432: (0) def_gen.hypergeometric(20, 20, 10, size=1)
433: (0) def_gen.hypergeometric(I_arr_20, 20, 10)
434: (0) def_gen.hypergeometric(20, I_arr_20, 10)
435: (0) def_gen.hypergeometric(I_arr_20, 20, I_arr_like_10, size=1)
436: (0) def_gen.hypergeometric(20, I_arr_20, 10, size=1)
437: (0) def_gen.hypergeometric(I_arr_like_20, 20, I_arr_10)
438: (0) def_gen.hypergeometric(20, I_arr_like_20, 10)
439: (0) def_gen.hypergeometric(I_arr_20, I_arr_20, 10)
440: (0) def_gen.hypergeometric(I_arr_like_20, I_arr_like_20, 10)
441: (0) def_gen.hypergeometric(I_arr_20, I_arr_20, I_arr_10, size=1)
442: (0) def_gen.hypergeometric(I_arr_like_20, I_arr_like_20, I_arr_like_10, size=1)
443: (0) I_int64_100: np.ndarray[Any, np.dtype[np.int64]] = np.array([100],
dtype=np.int64)
444: (0) def_gen.integers(0, 100)
445: (0) def_gen.integers(100)
446: (0) def_gen.integers([100])
447: (0) def_gen.integers(0, [100])
448: (0) I_bool_low: np.ndarray[Any, np.dtype[np.bool_]] = np.array([0],

```

```

dtype=np.bool_)
449: (0)
450: (0)
dtype=np.bool_
451: (0)
dtype=np.bool_
452: (0)
453: (0)
454: (0)
455: (0)
456: (0)
457: (0)
458: (0)
459: (0)
460: (0)
461: (0)
462: (0)
463: (0)
464: (0)
465: (0)
466: (0)
467: (0)
468: (0)
469: (0)
470: (0)
471: (0)
472: (0)
endpoint=True)
473: (0)
474: (0)
475: (0)
476: (0)
dtype=np.uint8)
477: (0)
dtype=np.uint8)
478: (0)
479: (0)
480: (0)
481: (0)
482: (0)
483: (0)
484: (0)
485: (0)
486: (0)
487: (0)
488: (0)
489: (0)
490: (0)
491: (0)
492: (0)
493: (0)
494: (0)
495: (0)
496: (0)
497: (0)
498: (0)
499: (0)
500: (0)
501: (0)
502: (0)
503: (0)
504: (0)
505: (0)
506: (0)
507: (0)
508: (0)
509: (0)
510: (0)
511: (0)

I_bool_low_like: list[int] = [0]
I_bool_high_open: np.ndarray[Any, np.dtype[np.bool_]] = np.array([1]),
I_bool_high_closed: np.ndarray[Any, np.dtype[np.bool_]] = np.array([1]),
def_gen.integers(2, dtype=bool)
def_gen.integers(0, 2, dtype=bool)
def_gen.integers(1, dtype=bool, endpoint=True)
def_gen.integers(0, 1, dtype=bool, endpoint=True)
def_gen.integers(I_bool_low_like, 1, dtype=bool, endpoint=True)
def_gen.integers(I_bool_high_open, dtype=bool)
def_gen.integers(I_bool_low, I_bool_high_open, dtype=bool)
def_gen.integers(0, I_bool_high_open, dtype=bool)
def_gen.integers(I_bool_high_closed, dtype=bool, endpoint=True)
def_gen.integers(I_bool_low, I_bool_high_closed, dtype=bool, endpoint=True)
def_gen.integers(0, I_bool_high_closed, dtype=bool, endpoint=True)
def_gen.integers(2, dtype=np.bool_)
def_gen.integers(0, 2, dtype=np.bool_)
def_gen.integers(0, 1, dtype=np.bool_, endpoint=True)
def_gen.integers(I_bool_low_like, 1, dtype=np.bool_, endpoint=True)
def_gen.integers(I_bool_high_open, dtype=np.bool_)
def_gen.integers(I_bool_low, I_bool_high_open, dtype=np.bool_)
def_gen.integers(0, I_bool_high_open, dtype=np.bool_)
def_gen.integers(I_bool_high_closed, dtype=np.bool_, endpoint=True)
def_gen.integers(I_bool_low, I_bool_high_closed, dtype=np.bool_,
endpoint=True)
def_gen.integers(0, I_bool_high_closed, dtype=np.bool_, endpoint=True)
I_u1_low: np.ndarray[Any, np.dtype[np.uint8]] = np.array([0], dtype=np.uint8)
I_u1_low_like: list[int] = [0]
I_u1_high_open: np.ndarray[Any, np.dtype[np.uint8]] = np.array([255],
I_u1_high_closed: np.ndarray[Any, np.dtype[np.uint8]] = np.array([255],
def_gen.integers(256, dtype="u1")
def_gen.integers(0, 256, dtype="u1")
def_gen.integers(255, dtype="u1", endpoint=True)
def_gen.integers(0, 255, dtype="u1", endpoint=True)
def_gen.integers(I_u1_low_like, 255, dtype="u1", endpoint=True)
def_gen.integers(I_u1_high_open, dtype="u1")
def_gen.integers(I_u1_low, I_u1_high_open, dtype="u1")
def_gen.integers(0, I_u1_high_open, dtype="u1")
def_gen.integers(I_u1_high_closed, dtype="u1", endpoint=True)
def_gen.integers(I_u1_low, I_u1_high_closed, dtype="u1", endpoint=True)
def_gen.integers(0, I_u1_high_closed, dtype="u1", endpoint=True)
def_gen.integers(256, dtype="uint8")
def_gen.integers(0, 256, dtype="uint8")
def_gen.integers(255, dtype="uint8", endpoint=True)
def_gen.integers(0, 255, dtype="uint8", endpoint=True)
def_gen.integers(I_u1_low_like, 255, dtype="uint8", endpoint=True)
def_gen.integers(I_u1_high_open, dtype="uint8")
def_gen.integers(I_u1_low, I_u1_high_open, dtype="uint8")
def_gen.integers(0, I_u1_high_open, dtype="uint8")
def_gen.integers(I_u1_high_closed, dtype="uint8", endpoint=True)
def_gen.integers(I_u1_low, I_u1_high_closed, dtype="uint8", endpoint=True)
def_gen.integers(0, I_u1_high_closed, dtype="uint8", endpoint=True)
def_gen.integers(256, dtype=np.uint8)
def_gen.integers(0, 256, dtype=np.uint8)
def_gen.integers(255, dtype=np.uint8, endpoint=True)
def_gen.integers(0, 255, dtype=np.uint8, endpoint=True)
def_gen.integers(I_u1_low_like, 255, dtype=np.uint8, endpoint=True)
def_gen.integers(I_u1_high_open, dtype=np.uint8)
def_gen.integers(I_u1_low, I_u1_high_open, dtype=np.uint8)
def_gen.integers(0, I_u1_high_open, dtype=np.uint8)
def_gen.integers(I_u1_high_closed, dtype=np.uint8, endpoint=True)
def_gen.integers(I_u1_low, I_u1_high_closed, dtype=np.uint8, endpoint=True)
def_gen.integers(0, I_u1_high_closed, dtype=np.uint8, endpoint=True)
I_u2_low: np.ndarray[Any, np.dtype[np.uint16]] = np.array([0],

```

```

dtype=np.uint16)
512: (0) I_u2_low_like: list[int] = [0]
513: (0) I_u2_high_open: np.ndarray[Any, np.dtype[np.uint16]] = np.array([65535],
514: (0) I_u2_high_closed: np.ndarray[Any, np.dtype[np.uint16]] = np.array([65535],
515: (0) def_gen.integers(65536, dtype="u2")
516: (0) def_gen.integers(0, 65536, dtype="u2")
517: (0) def_gen.integers(65535, dtype="u2", endpoint=True)
518: (0) def_gen.integers(0, 65535, dtype="u2", endpoint=True)
519: (0) def_gen.integers(I_u2_low_like, 65535, dtype="u2", endpoint=True)
520: (0) def_gen.integers(I_u2_high_open, dtype="u2")
521: (0) def_gen.integers(I_u2_low, I_u2_high_open, dtype="u2")
522: (0) def_gen.integers(0, I_u2_high_open, dtype="u2")
523: (0) def_gen.integers(I_u2_high_closed, dtype="u2", endpoint=True)
524: (0) def_gen.integers(I_u2_low, I_u2_high_closed, dtype="u2", endpoint=True)
525: (0) def_gen.integers(0, I_u2_high_closed, dtype="u2", endpoint=True)
526: (0) def_gen.integers(65536, dtype="uint16")
527: (0) def_gen.integers(0, 65536, dtype="uint16")
528: (0) def_gen.integers(65535, dtype="uint16", endpoint=True)
529: (0) def_gen.integers(0, 65535, dtype="uint16", endpoint=True)
530: (0) def_gen.integers(I_u2_low_like, 65535, dtype="uint16", endpoint=True)
531: (0) def_gen.integers(I_u2_high_open, dtype="uint16")
532: (0) def_gen.integers(I_u2_low, I_u2_high_open, dtype="uint16")
533: (0) def_gen.integers(0, I_u2_high_open, dtype="uint16")
534: (0) def_gen.integers(I_u2_high_closed, dtype="uint16", endpoint=True)
535: (0) def_gen.integers(I_u2_low, I_u2_high_closed, dtype="uint16", endpoint=True)
536: (0) def_gen.integers(0, I_u2_high_closed, dtype="uint16", endpoint=True)
537: (0) def_gen.integers(65536, dtype=np.uint16)
538: (0) def_gen.integers(0, 65536, dtype=np.uint16)
539: (0) def_gen.integers(65535, dtype=np.uint16, endpoint=True)
540: (0) def_gen.integers(0, 65535, dtype=np.uint16, endpoint=True)
541: (0) def_gen.integers(I_u2_low_like, 65535, dtype=np.uint16, endpoint=True)
542: (0) def_gen.integers(I_u2_high_open, dtype=np.uint16)
543: (0) def_gen.integers(I_u2_low, I_u2_high_open, dtype=np.uint16)
544: (0) def_gen.integers(0, I_u2_high_open, dtype=np.uint16)
545: (0) def_gen.integers(I_u2_high_closed, dtype=np.uint16, endpoint=True)
546: (0) def_gen.integers(I_u2_low, I_u2_high_closed, dtype=np.uint16, endpoint=True)
547: (0) def_gen.integers(0, I_u2_high_closed, dtype=np.uint16, endpoint=True)
548: (0) I_u4_low: np.ndarray[Any, np.dtype[np.uint32]] = np.array([0],
549: (0) dtype=np.uint32)
550: (0) I_u4_low_like: list[int] = [0]
551: (0) I_u4_high_open: np.ndarray[Any, np.dtype[np.uint32]] = np.array([4294967295],
552: (0) np.array([4294967295], dtype=np.uint32)
553: (0) I_u4_high_closed: np.ndarray[Any, np.dtype[np.uint32]] =
554: (0) def_gen.integers(4294967296, dtype="u4")
555: (0) def_gen.integers(0, 4294967296, dtype="u4")
556: (0) def_gen.integers(4294967295, dtype="u4", endpoint=True)
557: (0) def_gen.integers(0, 4294967295, dtype="u4", endpoint=True)
558: (0) def_gen.integers(I_u4_low_like, 4294967295, dtype="u4", endpoint=True)
559: (0) def_gen.integers(I_u4_high_open, dtype="u4")
560: (0) def_gen.integers(I_u4_low, I_u4_high_open, dtype="u4")
561: (0) def_gen.integers(0, I_u4_high_open, dtype="u4")
562: (0) def_gen.integers(I_u4_high_closed, dtype="u4", endpoint=True)
563: (0) def_gen.integers(0, I_u4_high_closed, dtype="u4", endpoint=True)
564: (0) def_gen.integers(4294967296, dtype="uint32")
565: (0) def_gen.integers(0, 4294967296, dtype="uint32")
566: (0) def_gen.integers(4294967295, dtype="uint32", endpoint=True)
567: (0) def_gen.integers(0, 4294967295, dtype="uint32", endpoint=True)
568: (0) def_gen.integers(I_u4_low_like, 4294967295, dtype="uint32", endpoint=True)
569: (0) def_gen.integers(I_u4_high_open, dtype="uint32")
570: (0) def_gen.integers(0, I_u4_high_open, dtype="uint32")
571: (0) def_gen.integers(I_u4_high_closed, dtype="uint32", endpoint=True)
572: (0) def_gen.integers(I_u4_low, I_u4_high_closed, dtype="uint32", endpoint=True)
573: (0) def_gen.integers(0, I_u4_high_closed, dtype="uint32", endpoint=True)
574: (0) def_gen.integers(4294967296, dtype=np.uint32)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

575: (0) def_gen.integers(0, 4294967296, dtype=np.uint32)
576: (0) def_gen.integers(4294967295, dtype=np.uint32, endpoint=True)
577: (0) def_gen.integers(0, 4294967295, dtype=np.uint32, endpoint=True)
578: (0) def_gen.integers(I_u4_low_like, 4294967295, dtype=np.uint32, endpoint=True)
579: (0) def_gen.integers(I_u4_high_open, dtype=np.uint32)
580: (0) def_gen.integers(I_u4_low, I_u4_high_open, dtype=np.uint32)
581: (0) def_gen.integers(0, I_u4_high_open, dtype=np.uint32)
582: (0) def_gen.integers(I_u4_high_closed, dtype=np.uint32, endpoint=True)
583: (0) def_gen.integers(I_u4_low, I_u4_high_closed, dtype=np.uint32, endpoint=True)
584: (0) def_gen.integers(0, I_u4_high_closed, dtype=np.uint32, endpoint=True)
585: (0) I_u8_low: np.ndarray[Any, np.dtype[np.uint64]] = np.array([0],
dtype=np.uint64)
586: (0) I_u8_low_like: list[int] = [0]
587: (0) I_u8_high_open: np.ndarray[Any, np.dtype[np.uint64]] =
np.array([18446744073709551615], dtype=np.uint64)
588: (0) I_u8_high_closed: np.ndarray[Any, np.dtype[np.uint64]] =
np.array([18446744073709551615], dtype=np.uint64)
589: (0) def_gen.integers(18446744073709551616, dtype="u8")
590: (0) def_gen.integers(0, 18446744073709551616, dtype="u8")
591: (0) def_gen.integers(18446744073709551615, dtype="u8", endpoint=True)
592: (0) def_gen.integers(0, 18446744073709551615, dtype="u8", endpoint=True)
593: (0) def_gen.integers(I_u8_low_like, 18446744073709551615, dtype="u8",
endpoint=True)
594: (0) def_gen.integers(I_u8_high_open, dtype="u8")
595: (0) def_gen.integers(I_u8_low, I_u8_high_open, dtype="u8")
596: (0) def_gen.integers(0, I_u8_high_open, dtype="u8")
597: (0) def_gen.integers(I_u8_high_closed, dtype="u8", endpoint=True)
598: (0) def_gen.integers(I_u8_low, I_u8_high_closed, dtype="u8", endpoint=True)
599: (0) def_gen.integers(0, I_u8_high_closed, dtype="u8", endpoint=True)
600: (0) def_gen.integers(18446744073709551616, dtype="uint64")
601: (0) def_gen.integers(0, 18446744073709551616, dtype="uint64")
602: (0) def_gen.integers(18446744073709551615, dtype="uint64", endpoint=True)
603: (0) def_gen.integers(0, 18446744073709551615, dtype="uint64", endpoint=True)
604: (0) def_gen.integers(I_u8_low_like, 18446744073709551615, dtype="uint64",
endpoint=True)
605: (0) def_gen.integers(I_u8_high_open, dtype="uint64")
606: (0) def_gen.integers(I_u8_low, I_u8_high_open, dtype="uint64")
607: (0) def_gen.integers(0, I_u8_high_open, dtype="uint64")
608: (0) def_gen.integers(I_u8_high_closed, dtype="uint64", endpoint=True)
609: (0) def_gen.integers(I_u8_low, I_u8_high_closed, dtype="uint64", endpoint=True)
610: (0) def_gen.integers(0, I_u8_high_closed, dtype="uint64", endpoint=True)
611: (0) def_gen.integers(18446744073709551616, dtype=np.uint64)
612: (0) def_gen.integers(0, 18446744073709551616, dtype=np.uint64)
613: (0) def_gen.integers(18446744073709551615, dtype=np.uint64, endpoint=True)
614: (0) def_gen.integers(0, 18446744073709551615, dtype=np.uint64, endpoint=True)
615: (0) def_gen.integers(I_u8_low_like, 18446744073709551615, dtype=np.uint64,
endpoint=True)
616: (0) def_gen.integers(I_u8_high_open, dtype=np.uint64)
617: (0) def_gen.integers(I_u8_low, I_u8_high_open, dtype=np.uint64)
618: (0) def_gen.integers(0, I_u8_high_open, dtype=np.uint64)
619: (0) def_gen.integers(I_u8_high_closed, dtype=np.uint64, endpoint=True)
620: (0) def_gen.integers(I_u8_low, I_u8_high_closed, dtype=np.uint64, endpoint=True)
621: (0) def_gen.integers(0, I_u8_high_closed, dtype=np.uint64, endpoint=True)
622: (0) I_i1_low: np.ndarray[Any, np.dtype[np.int8]] = np.array([-128], dtype=np.int8)
623: (0) I_i1_low_like: list[int] = [-128]
624: (0) I_i1_high_open: np.ndarray[Any, np.dtype[np.int8]] = np.array([127],
dtype=np.int8)
625: (0) I_i1_high_closed: np.ndarray[Any, np.dtype[np.int8]] = np.array([127],
dtype=np.int8)
626: (0) def_gen.integers(128, dtype="i1")
627: (0) def_gen.integers(-128, 128, dtype="i1")
628: (0) def_gen.integers(127, dtype="i1", endpoint=True)
629: (0) def_gen.integers(-128, 127, dtype="i1", endpoint=True)
630: (0) def_gen.integers(I_i1_low_like, 127, dtype="i1", endpoint=True)
631: (0) def_gen.integers(I_i1_high_open, dtype="i1")
632: (0) def_gen.integers(I_i1_low, I_i1_high_open, dtype="i1")
633: (0) def_gen.integers(-128, I_i1_high_open, dtype="i1")
634: (0) def_gen.integers(I_i1_high_closed, dtype="i1", endpoint=True)
635: (0) def_gen.integers(I_i1_low, I_i1_high_closed, dtype="i1", endpoint=True)

```

```

636: (0) def_gen.integers(-128, I_i1_high_closed, dtype="i1", endpoint=True)
637: (0) def_gen.integers(128, dtype="int8")
638: (0) def_gen.integers(-128, 128, dtype="int8")
639: (0) def_gen.integers(127, dtype="int8", endpoint=True)
640: (0) def_gen.integers(-128, 127, dtype="int8", endpoint=True)
641: (0) def_gen.integers(I_i1_low_like, 127, dtype="int8", endpoint=True)
642: (0) def_gen.integers(I_i1_high_open, dtype="int8")
643: (0) def_gen.integers(I_i1_low, I_i1_high_open, dtype="int8")
644: (0) def_gen.integers(-128, I_i1_high_open, dtype="int8")
645: (0) def_gen.integers(I_i1_high_closed, dtype="int8", endpoint=True)
646: (0) def_gen.integers(I_i1_low, I_i1_high_closed, dtype="int8", endpoint=True)
647: (0) def_gen.integers(-128, I_i1_high_closed, dtype="int8", endpoint=True)
648: (0) def_gen.integers(128, dtype=np.int8)
649: (0) def_gen.integers(-128, 128, dtype=np.int8)
650: (0) def_gen.integers(127, dtype=np.int8, endpoint=True)
651: (0) def_gen.integers(-128, 127, dtype=np.int8, endpoint=True)
652: (0) def_gen.integers(I_i1_low_like, 127, dtype=np.int8, endpoint=True)
653: (0) def_gen.integers(I_i1_high_open, dtype=np.int8)
654: (0) def_gen.integers(I_i1_low, I_i1_high_open, dtype=np.int8)
655: (0) def_gen.integers(-128, I_i1_high_open, dtype=np.int8)
656: (0) def_gen.integers(I_i1_high_closed, dtype=np.int8, endpoint=True)
657: (0) def_gen.integers(I_i1_low, I_i1_high_closed, dtype=np.int8, endpoint=True)
658: (0) def_gen.integers(-128, I_i1_high_closed, dtype=np.int8, endpoint=True)
659: (0) I_i2_low: np.ndarray[Any, np.dtype[np.int16]] = np.array([-32768],
dtype=np.int16)
660: (0) I_i2_low_like: list[int] = [-32768]
661: (0) I_i2_high_open: np.ndarray[Any, np.dtype[np.int16]] = np.array([32767],
dtype=np.int16)
662: (0) I_i2_high_closed: np.ndarray[Any, np.dtype[np.int16]] = np.array([32767],
dtype=np.int16)
663: (0) def_gen.integers(32768, dtype="i2")
664: (0) def_gen.integers(-32768, 32768, dtype="i2")
665: (0) def_gen.integers(32767, dtype="i2", endpoint=True)
666: (0) def_gen.integers(-32768, 32767, dtype="i2", endpoint=True)
667: (0) def_gen.integers(I_i2_low_like, 32767, dtype="i2", endpoint=True)
668: (0) def_gen.integers(I_i2_high_open, dtype="i2")
669: (0) def_gen.integers(I_i2_low, I_i2_high_open, dtype="i2")
670: (0) def_gen.integers(-32768, I_i2_high_open, dtype="i2")
671: (0) def_gen.integers(I_i2_high_closed, dtype="i2", endpoint=True)
672: (0) def_gen.integers(I_i2_low, I_i2_high_closed, dtype="i2", endpoint=True)
673: (0) def_gen.integers(-32768, I_i2_high_closed, dtype="i2", endpoint=True)
674: (0) def_gen.integers(32768, dtype="int16")
675: (0) def_gen.integers(-32768, 32768, dtype="int16")
676: (0) def_gen.integers(32767, dtype="int16", endpoint=True)
677: (0) def_gen.integers(-32768, 32767, dtype="int16", endpoint=True)
678: (0) def_gen.integers(I_i2_low_like, 32767, dtype="int16", endpoint=True)
679: (0) def_gen.integers(I_i2_high_open, dtype="int16")
680: (0) def_gen.integers(I_i2_low, I_i2_high_open, dtype="int16")
681: (0) def_gen.integers(-32768, I_i2_high_open, dtype="int16")
682: (0) def_gen.integers(I_i2_high_closed, dtype="int16", endpoint=True)
683: (0) def_gen.integers(I_i2_low, I_i2_high_closed, dtype="int16", endpoint=True)
684: (0) def_gen.integers(-32768, I_i2_high_closed, dtype="int16", endpoint=True)
685: (0) def_gen.integers(32768, dtype=np.int16)
686: (0) def_gen.integers(-32768, 32768, dtype=np.int16)
687: (0) def_gen.integers(32767, dtype=np.int16, endpoint=True)
688: (0) def_gen.integers(-32768, 32767, dtype=np.int16, endpoint=True)
689: (0) def_gen.integers(I_i2_low_like, 32767, dtype=np.int16, endpoint=True)
690: (0) def_gen.integers(I_i2_high_open, dtype=np.int16)
691: (0) def_gen.integers(I_i2_low, I_i2_high_open, dtype=np.int16)
692: (0) def_gen.integers(-32768, I_i2_high_open, dtype=np.int16)
693: (0) def_gen.integers(I_i2_high_closed, dtype=np.int16, endpoint=True)
694: (0) def_gen.integers(I_i2_low, I_i2_high_closed, dtype=np.int16, endpoint=True)
695: (0) def_gen.integers(-32768, I_i2_high_closed, dtype=np.int16, endpoint=True)
696: (0) I_i4_low: np.ndarray[Any, np.dtype[np.int32]] = np.array([-2147483648],
dtype=np.int32)
697: (0) I_i4_low_like: list[int] = [-2147483648]
698: (0) I_i4_high_open: np.ndarray[Any, np.dtype[np.int32]] = np.array([2147483647],
dtype=np.int32)
699: (0) I_i4_high_closed: np.ndarray[Any, np.dtype[np.int32]] = np.array([2147483647],

```

```

dtype=np.int32)
700: (0) def_gen.integers(2147483648, dtype="i4")
701: (0) def_gen.integers(-2147483648, 2147483648, dtype="i4")
702: (0) def_gen.integers(2147483647, dtype="i4", endpoint=True)
703: (0) def_gen.integers(-2147483648, 2147483647, dtype="i4", endpoint=True)
704: (0) def_gen.integers(I_i4_low_like, 2147483647, dtype="i4", endpoint=True)
705: (0) def_gen.integers(I_i4_high_open, dtype="i4")
706: (0) def_gen.integers(I_i4_low, I_i4_high_open, dtype="i4")
707: (0) def_gen.integers(-2147483648, I_i4_high_open, dtype="i4")
708: (0) def_gen.integers(I_i4_high_closed, dtype="i4", endpoint=True)
709: (0) def_gen.integers(I_i4_low, I_i4_high_closed, dtype="i4", endpoint=True)
710: (0) def_gen.integers(-2147483648, I_i4_high_closed, dtype="i4", endpoint=True)
711: (0) def_gen.integers(2147483648, dtype="int32")
712: (0) def_gen.integers(-2147483648, 2147483648, dtype="int32")
713: (0) def_gen.integers(2147483647, dtype="int32", endpoint=True)
714: (0) def_gen.integers(-2147483648, 2147483647, dtype="int32", endpoint=True)
715: (0) def_gen.integers(I_i4_low_like, 2147483647, dtype="int32", endpoint=True)
716: (0) def_gen.integers(I_i4_high_open, dtype="int32")
717: (0) def_gen.integers(I_i4_low, I_i4_high_open, dtype="int32")
718: (0) def_gen.integers(-2147483648, I_i4_high_open, dtype="int32")
719: (0) def_gen.integers(I_i4_high_closed, dtype="int32", endpoint=True)
720: (0) def_gen.integers(I_i4_low, I_i4_high_closed, dtype="int32", endpoint=True)
721: (0) def_gen.integers(-2147483648, I_i4_high_closed, dtype="int32", endpoint=True)
722: (0) def_gen.integers(2147483648, dtype=np.int32)
723: (0) def_gen.integers(-2147483648, 2147483648, dtype=np.int32)
724: (0) def_gen.integers(2147483647, dtype=np.int32, endpoint=True)
725: (0) def_gen.integers(-2147483648, 2147483647, dtype=np.int32, endpoint=True)
726: (0) def_gen.integers(I_i4_low_like, 2147483647, dtype=np.int32, endpoint=True)
727: (0) def_gen.integers(I_i4_high_open, dtype=np.int32)
728: (0) def_gen.integers(I_i4_low, I_i4_high_open, dtype=np.int32)
729: (0) def_gen.integers(-2147483648, I_i4_high_open, dtype=np.int32)
730: (0) def_gen.integers(I_i4_high_closed, dtype=np.int32, endpoint=True)
731: (0) def_gen.integers(I_i4_low, I_i4_high_closed, dtype=np.int32, endpoint=True)
732: (0) def_gen.integers(-2147483648, I_i4_high_closed, dtype=np.int32, endpoint=True)
733: (0) I_i8_low: np.ndarray[Any, np.dtype[np.int64]] =
np.array([-9223372036854775808], dtype=np.int64)
734: (0) I_i8_low_like: list[int] = [-9223372036854775808]
735: (0) I_i8_high_open: np.ndarray[Any, np.dtype[np.int64]] =
np.array([9223372036854775807], dtype=np.int64)
736: (0) I_i8_high_closed: np.ndarray[Any, np.dtype[np.int64]] =
np.array([9223372036854775807], dtype=np.int64)
737: (0) def_gen.integers(9223372036854775808, dtype="i8")
738: (0) def_gen.integers(-9223372036854775808, 9223372036854775808, dtype="i8")
739: (0) def_gen.integers(9223372036854775807, dtype="i8", endpoint=True)
740: (0) def_gen.integers(-9223372036854775808, 9223372036854775807, dtype="i8",
endpoint=True)
741: (0) def_gen.integers(I_i8_low_like, 9223372036854775807, dtype="i8",
endpoint=True)
742: (0) def_gen.integers(I_i8_high_open, dtype="i8")
743: (0) def_gen.integers(I_i8_low, I_i8_high_open, dtype="i8")
744: (0) def_gen.integers(-9223372036854775808, I_i8_high_open, dtype="i8")
745: (0) def_gen.integers(I_i8_high_closed, dtype="i8", endpoint=True)
746: (0) def_gen.integers(I_i8_low, I_i8_high_closed, dtype="i8", endpoint=True)
747: (0) def_gen.integers(-9223372036854775808, I_i8_high_closed, dtype="i8",
endpoint=True)
748: (0) def_gen.integers(9223372036854775808, dtype="int64")
749: (0) def_gen.integers(-9223372036854775808, 9223372036854775808, dtype="int64")
750: (0) def_gen.integers(9223372036854775807, dtype="int64", endpoint=True)
751: (0) def_gen.integers(-9223372036854775808, 9223372036854775807, dtype="int64",
endpoint=True)
752: (0) def_gen.integers(I_i8_low_like, 9223372036854775807, dtype="int64",
endpoint=True)
753: (0) def_gen.integers(I_i8_high_open, dtype="int64")
754: (0) def_gen.integers(I_i8_low, I_i8_high_open, dtype="int64")
755: (0) def_gen.integers(-9223372036854775808, I_i8_high_open, dtype="int64")
756: (0) def_gen.integers(I_i8_high_closed, dtype="int64", endpoint=True)
757: (0) def_gen.integers(I_i8_low, I_i8_high_closed, dtype="int64", endpoint=True)
758: (0) def_gen.integers(-9223372036854775808, I_i8_high_closed, dtype="int64",
endpoint=True)

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

759: (0)
760: (0)
761: (0)
762: (0)
    endpoint=True)
763: (0)
    endpoint=True)
764: (0)
765: (0)
766: (0)
767: (0)
768: (0)
769: (0)
    endpoint=True)
770: (0)
771: (0)
772: (0)
773: (0)
774: (0)
775: (0)
776: (0)
777: (0)
778: (0)
779: (0)
780: (0)
781: (0)
p=np.array([1 / 8, 1 / 8, 1 / 2, 1 / 4]))
782: (0)
783: (0)
784: (0)
785: (0)
786: (0)
787: (0)
788: (0)
789: (0)
790: (0)
791: (0)
792: (0)
793: (0)
794: (0)
795: (0)
method="marginals")
796: (0)
797: (0)
798: (0)
799: (0)
800: (0)
801: (0)
802: (0)
803: (0)
804: (0)
805: (0)
806: (0)
807: (0)
808: (0)
809: (0)
810: (0)
811: (0)
812: (0)
813: (0)
814: (0)
815: (0)
816: (0)
def_gen_state: dict[str, Any]
817: (0)
818: (0)
819: (0)
random_st: np.random.RandomState = np.random.RandomState()
820: (0)
821: (0)
822: (0)
    random_st.standard_normal(size=None)
        random_st.standard_normal(size=1)

```

```
823: (0) random_st.random()
824: (0) random_st.random(size=None)
825: (0) random_st.random(size=1)
826: (0) random_st.standard_cauchy()
827: (0) random_st.standard_cauchy(size=None)
828: (0) random_st.standard_cauchy(size=1)
829: (0) random_st.standard_exponential()
830: (0) random_st.standard_exponential(size=None)
831: (0) random_st.standard_exponential(size=1)
832: (0) random_st.zipf(1.5)
833: (0) random_st.zipf(1.5, size=None)
834: (0) random_st.zipf(1.5, size=1)
835: (0) random_st.zipf(D_arr_1p5)
836: (0) random_st.zipf(D_arr_1p5, size=1)
837: (0) random_st.zipf(D_arr_like_1p5)
838: (0) random_st.zipf(D_arr_like_1p5, size=1)
839: (0) random_st.weibull(0.5)
840: (0) random_st.weibull(0.5, size=None)
841: (0) random_st.weibull(0.5, size=1)
842: (0) random_st.weibull(D_arr_0p5)
843: (0) random_st.weibull(D_arr_0p5, size=1)
844: (0) random_st.weibull(D_arr_like_0p5)
845: (0) random_st.weibull(D_arr_like_0p5, size=1)
846: (0) random_st.standard_t(0.5)
847: (0) random_st.standard_t(0.5, size=None)
848: (0) random_st.standard_t(0.5, size=1)
849: (0) random_st.standard_t(D_arr_0p5)
850: (0) random_st.standard_t(D_arr_0p5, size=1)
851: (0) random_st.standard_t(D_arr_like_0p5)
852: (0) random_st.standard_t(D_arr_like_0p5, size=1)
853: (0) random_st.poisson(0.5)
854: (0) random_st.poisson(0.5, size=None)
855: (0) random_st.poisson(0.5, size=1)
856: (0) random_st.poisson(D_arr_0p5)
857: (0) random_st.poisson(D_arr_0p5, size=1)
858: (0) random_st.poisson(D_arr_like_0p5)
859: (0) random_st.poisson(D_arr_like_0p5, size=1)
860: (0) random_st.power(0.5)
861: (0) random_st.power(0.5, size=None)
862: (0) random_st.power(0.5, size=1)
863: (0) random_st.power(D_arr_0p5)
864: (0) random_st.power(D_arr_0p5, size=1)
865: (0) random_st.power(D_arr_like_0p5)
866: (0) random_st.power(D_arr_like_0p5, size=1)
867: (0) random_st.pareto(0.5)
868: (0) random_st.pareto(0.5, size=None)
869: (0) random_st.pareto(0.5, size=1)
870: (0) random_st.pareto(D_arr_0p5)
871: (0) random_st.pareto(D_arr_0p5, size=1)
872: (0) random_st.pareto(D_arr_like_0p5)
873: (0) random_st.pareto(D_arr_like_0p5, size=1)
874: (0) random_st.chisquare(0.5)
875: (0) random_st.chisquare(0.5, size=None)
876: (0) random_st.chisquare(0.5, size=1)
877: (0) random_st.chisquare(D_arr_0p5)
878: (0) random_st.chisquare(D_arr_0p5, size=1)
879: (0) random_st.chisquare(D_arr_like_0p5)
880: (0) random_st.chisquare(D_arr_like_0p5, size=1)
881: (0) random_st.exponential(0.5)
882: (0) random_st.exponential(0.5, size=None)
883: (0) random_st.exponential(0.5, size=1)
884: (0) random_st.exponential(D_arr_0p5)
885: (0) random_st.exponential(D_arr_0p5, size=1)
886: (0) random_st.exponential(D_arr_like_0p5)
887: (0) random_st.exponential(D_arr_like_0p5, size=1)
888: (0) random_st.geometric(0.5)
889: (0) random_st.geometric(0.5, size=None)
890: (0) random_st.geometric(0.5, size=1)
891: (0) random_st.geometric(D_arr_0p5)
```

```
892: (0) random_st.geometric(D_arr_0p5, size=1)
893: (0) random_st.geometric(D_arr_like_0p5)
894: (0) random_st.geometric(D_arr_like_0p5, size=1)
895: (0) random_st.logseries(0.5)
896: (0) random_st.logseries(0.5, size=None)
897: (0) random_st.logseries(0.5, size=1)
898: (0) random_st.logseries(D_arr_0p5)
899: (0) random_st.logseries(D_arr_0p5, size=1)
900: (0) random_st.logseries(D_arr_like_0p5)
901: (0) random_st.logseries(D_arr_like_0p5, size=1)
902: (0) random_st.rayleigh(0.5)
903: (0) random_st.rayleigh(0.5, size=None)
904: (0) random_st.rayleigh(0.5, size=1)
905: (0) random_st.rayleigh(D_arr_0p5)
906: (0) random_st.rayleigh(D_arr_0p5, size=1)
907: (0) random_st.rayleigh(D_arr_like_0p5)
908: (0) random_st.rayleigh(D_arr_like_0p5, size=1)
909: (0) random_st.standard_gamma(0.5)
910: (0) random_st.standard_gamma(0.5, size=None)
911: (0) random_st.standard_gamma(0.5, size=1)
912: (0) random_st.standard_gamma(D_arr_0p5)
913: (0) random_st.standard_gamma(D_arr_0p5, size=1)
914: (0) random_st.standard_gamma(D_arr_like_0p5)
915: (0) random_st.standard_gamma(D_arr_like_0p5, size=1)
916: (0) random_st.standard_gamma(D_arr_like_0p5, size=1)
917: (0) random_st.vonmises(0.5, 0.5)
918: (0) random_st.vonmises(0.5, 0.5, size=None)
919: (0) random_st.vonmises(0.5, 0.5, size=1)
920: (0) random_st.vonmises(D_arr_0p5, 0.5)
921: (0) random_st.vonmises(0.5, D_arr_0p5)
922: (0) random_st.vonmises(D_arr_0p5, 0.5, size=1)
923: (0) random_st.vonmises(0.5, D_arr_0p5, size=1)
924: (0) random_st.vonmises(D_arr_like_0p5, 0.5)
925: (0) random_st.vonmises(0.5, D_arr_like_0p5)
926: (0) random_st.vonmises(D_arr_0p5, D_arr_0p5)
927: (0) random_st.vonmises(D_arr_like_0p5, D_arr_like_0p5)
928: (0) random_st.vonmises(D_arr_0p5, D_arr_0p5, size=1)
929: (0) random_st.vonmises(D_arr_like_0p5, D_arr_like_0p5, size=1)
930: (0) random_st.wald(0.5, 0.5)
931: (0) random_st.wald(0.5, 0.5, size=None)
932: (0) random_st.wald(0.5, 0.5, size=1)
933: (0) random_st.wald(D_arr_0p5, 0.5)
934: (0) random_st.wald(0.5, D_arr_0p5)
935: (0) random_st.wald(D_arr_0p5, 0.5, size=1)
936: (0) random_st.wald(0.5, D_arr_0p5, size=1)
937: (0) random_st.wald(D_arr_like_0p5, 0.5)
938: (0) random_st.wald(0.5, D_arr_like_0p5)
939: (0) random_st.wald(D_arr_0p5, D_arr_0p5)
940: (0) random_st.wald(D_arr_like_0p5, D_arr_like_0p5)
941: (0) random_st.wald(D_arr_0p5, D_arr_0p5, size=1)
942: (0) random_st.wald(D_arr_like_0p5, D_arr_like_0p5, size=1)
943: (0) random_st.uniform(0.5, 0.5)
944: (0) random_st.uniform(0.5, 0.5, size=None)
945: (0) random_st.uniform(0.5, 0.5, size=1)
946: (0) random_st.uniform(D_arr_0p5, 0.5)
947: (0) random_st.uniform(0.5, D_arr_0p5)
948: (0) random_st.uniform(D_arr_0p5, 0.5, size=1)
949: (0) random_st.uniform(0.5, D_arr_0p5, size=1)
950: (0) random_st.uniform(D_arr_like_0p5, 0.5)
951: (0) random_st.uniform(0.5, D_arr_like_0p5)
952: (0) random_st.uniform(D_arr_0p5, D_arr_0p5)
953: (0) random_st.uniform(D_arr_like_0p5, D_arr_like_0p5)
954: (0) random_st.uniform(D_arr_0p5, D_arr_0p5, size=1)
955: (0) random_st.uniform(D_arr_like_0p5, D_arr_like_0p5, size=1)
956: (0) random_st.beta(0.5, 0.5)
957: (0) random_st.beta(0.5, 0.5, size=None)
958: (0) random_st.beta(0.5, 0.5, size=1)
959: (0) random_st.beta(D_arr_0p5, 0.5)
960: (0) random_st.beta(0.5, D_arr_0p5)
```

```
961: (0) random_st.beta(D_arr_0p5, 0.5, size=1)
962: (0) random_st.beta(0.5, D_arr_0p5, size=1)
963: (0) random_st.beta(D_arr_like_0p5, 0.5)
964: (0) random_st.beta(0.5, D_arr_like_0p5)
965: (0) random_st.beta(D_arr_0p5, D_arr_0p5)
966: (0) random_st.beta(D_arr_like_0p5, D_arr_like_0p5)
967: (0) random_st.beta(D_arr_0p5, D_arr_0p5, size=1)
968: (0) random_st.beta(D_arr_like_0p5, D_arr_like_0p5, size=1)
969: (0) random_st.f(0.5, 0.5)
970: (0) random_st.f(0.5, 0.5, size=None)
971: (0) random_st.f(0.5, 0.5, size=1)
972: (0) random_st.f(D_arr_0p5, 0.5)
973: (0) random_st.f(0.5, D_arr_0p5)
974: (0) random_st.f(D_arr_0p5, 0.5, size=1)
975: (0) random_st.f(0.5, D_arr_0p5, size=1)
976: (0) random_st.f(D_arr_like_0p5, 0.5)
977: (0) random_st.f(0.5, D_arr_like_0p5)
978: (0) random_st.f(D_arr_0p5, D_arr_0p5)
979: (0) random_st.f(D_arr_like_0p5, D_arr_like_0p5)
980: (0) random_st.f(D_arr_0p5, D_arr_0p5, size=1)
981: (0) random_st.f(D_arr_like_0p5, D_arr_like_0p5, size=1)
982: (0) random_st.gamma(0.5, 0.5)
983: (0) random_st.gamma(0.5, 0.5, size=None)
984: (0) random_st.gamma(0.5, 0.5, size=1)
985: (0) random_st.gamma(D_arr_0p5, 0.5)
986: (0) random_st.gamma(0.5, D_arr_0p5)
987: (0) random_st.gamma(D_arr_0p5, 0.5, size=1)
988: (0) random_st.gamma(0.5, D_arr_0p5, size=1)
989: (0) random_st.gamma(D_arr_like_0p5, 0.5)
990: (0) random_st.gamma(0.5, D_arr_like_0p5)
991: (0) random_st.gamma(D_arr_0p5, D_arr_0p5)
992: (0) random_st.gamma(D_arr_like_0p5, D_arr_like_0p5)
993: (0) random_st.gamma(D_arr_0p5, D_arr_0p5, size=1)
994: (0) random_st.gamma(D_arr_like_0p5, D_arr_like_0p5, size=1)
995: (0) random_st.gumbel(0.5, 0.5)
996: (0) random_st.gumbel(0.5, 0.5, size=None)
997: (0) random_st.gumbel(0.5, 0.5, size=1)
998: (0) random_st.gumbel(D_arr_0p5, 0.5)
999: (0) random_st.gumbel(0.5, D_arr_0p5)
1000: (0) random_st.gumbel(D_arr_0p5, 0.5, size=1)
1001: (0) random_st.gumbel(0.5, D_arr_0p5, size=1)
1002: (0) random_st.gumbel(D_arr_like_0p5, 0.5)
1003: (0) random_st.gumbel(0.5, D_arr_like_0p5)
1004: (0) random_st.gumbel(D_arr_0p5, D_arr_0p5)
1005: (0) random_st.gumbel(D_arr_like_0p5, D_arr_like_0p5)
1006: (0) random_st.gumbel(D_arr_0p5, D_arr_0p5, size=1)
1007: (0) random_st.gumbel(D_arr_like_0p5, D_arr_like_0p5, size=1)
1008: (0) random_st.laplace(0.5, 0.5)
1009: (0) random_st.laplace(0.5, 0.5, size=None)
1010: (0) random_st.laplace(0.5, 0.5, size=1)
1011: (0) random_st.laplace(D_arr_0p5, 0.5)
1012: (0) random_st.laplace(0.5, D_arr_0p5)
1013: (0) random_st.laplace(D_arr_0p5, 0.5, size=1)
1014: (0) random_st.laplace(0.5, D_arr_0p5, size=1)
1015: (0) random_st.laplace(D_arr_like_0p5, 0.5)
1016: (0) random_st.laplace(0.5, D_arr_like_0p5)
1017: (0) random_st.laplace(D_arr_0p5, D_arr_0p5)
1018: (0) random_st.laplace(D_arr_like_0p5, D_arr_like_0p5)
1019: (0) random_st.laplace(D_arr_0p5, D_arr_0p5, size=1)
1020: (0) random_st.laplace(D_arr_like_0p5, D_arr_like_0p5, size=1)
1021: (0) random_st.logistic(0.5, 0.5)
1022: (0) random_st.logistic(0.5, 0.5, size=None)
1023: (0) random_st.logistic(0.5, 0.5, size=1)
1024: (0) random_st.logistic(D_arr_0p5, 0.5)
1025: (0) random_st.logistic(0.5, D_arr_0p5)
1026: (0) random_st.logistic(D_arr_0p5, 0.5, size=1)
1027: (0) random_st.logistic(0.5, D_arr_0p5, size=1)
1028: (0) random_st.logistic(D_arr_like_0p5, 0.5)
1029: (0) random_st.logistic(0.5, D_arr_like_0p5)
```

```

1030: (0) random_st.logistic(D_arr_0p5, D_arr_0p5)
1031: (0) random_st.logistic(D_arr_like_0p5, D_arr_like_0p5)
1032: (0) random_st.logistic(D_arr_0p5, D_arr_0p5, size=1)
1033: (0) random_st.logistic(D_arr_like_0p5, D_arr_like_0p5, size=1)
1034: (0) random_st.lognormal(0.5, 0.5)
1035: (0) random_st.lognormal(0.5, 0.5, size=None)
1036: (0) random_st.lognormal(0.5, 0.5, size=1)
1037: (0) random_st.lognormal(D_arr_0p5, 0.5)
1038: (0) random_st.lognormal(0.5, D_arr_0p5)
1039: (0) random_st.lognormal(D_arr_0p5, 0.5, size=1)
1040: (0) random_st.lognormal(0.5, D_arr_0p5, size=1)
1041: (0) random_st.lognormal(D_arr_like_0p5, 0.5)
1042: (0) random_st.lognormal(0.5, D_arr_like_0p5)
1043: (0) random_st.lognormal(D_arr_0p5, D_arr_0p5)
1044: (0) random_st.lognormal(D_arr_like_0p5, D_arr_like_0p5)
1045: (0) random_st.lognormal(D_arr_0p5, D_arr_0p5, size=1)
1046: (0) random_st.lognormal(D_arr_like_0p5, D_arr_like_0p5, size=1)
1047: (0) random_st.noncentral_chisquare(0.5, 0.5)
1048: (0) random_st.noncentral_chisquare(0.5, 0.5, size=None)
1049: (0) random_st.noncentral_chisquare(0.5, 0.5, size=1)
1050: (0) random_st.noncentral_chisquare(D_arr_0p5, 0.5)
1051: (0) random_st.noncentral_chisquare(0.5, D_arr_0p5)
1052: (0) random_st.noncentral_chisquare(D_arr_0p5, 0.5, size=1)
1053: (0) random_st.noncentral_chisquare(0.5, D_arr_0p5, size=1)
1054: (0) random_st.noncentral_chisquare(D_arr_like_0p5, 0.5)
1055: (0) random_st.noncentral_chisquare(0.5, D_arr_like_0p5)
1056: (0) random_st.noncentral_chisquare(D_arr_0p5, D_arr_0p5)
1057: (0) random_st.noncentral_chisquare(D_arr_like_0p5, D_arr_like_0p5)
1058: (0) random_st.noncentral_chisquare(D_arr_0p5, D_arr_0p5, size=1)
1059: (0) random_st.noncentral_chisquare(D_arr_like_0p5, D_arr_like_0p5, size=1)
1060: (0) random_st.normal(0.5, 0.5)
1061: (0) random_st.normal(0.5, 0.5, size=None)
1062: (0) random_st.normal(0.5, 0.5, size=1)
1063: (0) random_st.normal(D_arr_0p5, 0.5)
1064: (0) random_st.normal(0.5, D_arr_0p5)
1065: (0) random_st.normal(D_arr_0p5, 0.5, size=1)
1066: (0) random_st.normal(0.5, D_arr_0p5, size=1)
1067: (0) random_st.normal(D_arr_like_0p5, 0.5)
1068: (0) random_st.normal(0.5, D_arr_like_0p5)
1069: (0) random_st.normal(D_arr_0p5, D_arr_0p5)
1070: (0) random_st.normal(D_arr_like_0p5, D_arr_like_0p5)
1071: (0) random_st.normal(D_arr_0p5, D_arr_0p5, size=1)
1072: (0) random_st.normal(D_arr_like_0p5, D_arr_like_0p5, size=1)
1073: (0) random_st.triangular(0.1, 0.5, 0.9)
1074: (0) random_st.triangular(0.1, 0.5, 0.9, size=None)
1075: (0) random_st.triangular(0.1, 0.5, 0.9, size=1)
1076: (0) random_st.triangular(D_arr_0p1, 0.5, 0.9)
1077: (0) random_st.triangular(0.1, D_arr_0p5, 0.9)
1078: (0) random_st.triangular(D_arr_0p1, 0.5, D_arr_like_0p9, size=1)
1079: (0) random_st.triangular(0.1, D_arr_0p5, 0.9, size=1)
1080: (0) random_st.triangular(D_arr_like_0p1, 0.5, D_arr_0p9)
1081: (0) random_st.triangular(0.5, D_arr_like_0p5, 0.9)
1082: (0) random_st.triangular(D_arr_0p1, D_arr_0p5, 0.9)
1083: (0) random_st.triangular(D_arr_like_0p1, D_arr_like_0p5, 0.9)
1084: (0) random_st.triangular(D_arr_0p1, D_arr_0p5, D_arr_0p9, size=1)
1085: (0) random_st.triangular(D_arr_like_0p1, D_arr_like_0p5, D_arr_like_0p9, size=1)
1086: (0) random_st.noncentral_f(0.1, 0.5, 0.9)
1087: (0) random_st.noncentral_f(0.1, 0.5, 0.9, size=None)
1088: (0) random_st.noncentral_f(0.1, 0.5, 0.9, size=1)
1089: (0) random_st.noncentral_f(D_arr_0p1, 0.5, 0.9)
1090: (0) random_st.noncentral_f(0.1, D_arr_0p5, 0.9)
1091: (0) random_st.noncentral_f(D_arr_0p1, 0.5, D_arr_like_0p9, size=1)
1092: (0) random_st.noncentral_f(0.1, D_arr_0p5, 0.9, size=1)
1093: (0) random_st.noncentral_f(D_arr_like_0p1, 0.5, D_arr_0p9)
1094: (0) random_st.noncentral_f(0.5, D_arr_like_0p5, 0.9)
1095: (0) random_st.noncentral_f(D_arr_0p1, D_arr_0p5, 0.9)
1096: (0) random_st.noncentral_f(D_arr_like_0p1, D_arr_like_0p5, 0.9)
1097: (0) random_st.noncentral_f(D_arr_0p1, D_arr_0p5, D_arr_0p9, size=1)
1098: (0) random_st.noncentral_f(D_arr_like_0p1, D_arr_like_0p5, D_arr_like_0p9, size=1)

```

```

1099: (0) random_st.binomial(10, 0.5)
1100: (0) random_st.binomial(10, 0.5, size=None)
1101: (0) random_st.binomial(10, 0.5, size=1)
1102: (0) random_st.binomial(I_arr_10, 0.5)
1103: (0) random_st.binomial(10, D_arr_0p5)
1104: (0) random_st.binomial(I_arr_10, 0.5, size=1)
1105: (0) random_st.binomial(10, D_arr_0p5, size=1)
1106: (0) random_st.binomial(I_arr_like_10, 0.5)
1107: (0) random_st.binomial(10, D_arr_like_0p5)
1108: (0) random_st.binomial(I_arr_10, D_arr_0p5)
1109: (0) random_st.binomial(I_arr_like_10, D_arr_like_0p5)
1110: (0) random_st.binomial(I_arr_10, D_arr_0p5, size=1)
1111: (0) random_st.binomial(I_arr_like_10, D_arr_like_0p5, size=1)
1112: (0) random_st.negative_binomial(10, 0.5)
1113: (0) random_st.negative_binomial(10, 0.5, size=None)
1114: (0) random_st.negative_binomial(10, 0.5, size=1)
1115: (0) random_st.negative_binomial(I_arr_10, 0.5)
1116: (0) random_st.negative_binomial(10, D_arr_0p5)
1117: (0) random_st.negative_binomial(I_arr_10, 0.5, size=1)
1118: (0) random_st.negative_binomial(10, D_arr_0p5, size=1)
1119: (0) random_st.negative_binomial(I_arr_like_10, 0.5)
1120: (0) random_st.negative_binomial(10, D_arr_like_0p5)
1121: (0) random_st.negative_binomial(I_arr_10, D_arr_0p5)
1122: (0) random_st.negative_binomial(I_arr_like_10, D_arr_like_0p5)
1123: (0) random_st.negative_binomial(I_arr_10, D_arr_0p5, size=1)
1124: (0) random_st.negative_binomial(I_arr_like_10, D_arr_like_0p5, size=1)
1125: (0) random_st.hypergeometric(20, 20, 10)
1126: (0) random_st.hypergeometric(20, 20, 10, size=None)
1127: (0) random_st.hypergeometric(20, 20, 10, size=1)
1128: (0) random_st.hypergeometric(I_arr_20, 20, 10)
1129: (0) random_st.hypergeometric(20, I_arr_20, 10)
1130: (0) random_st.hypergeometric(I_arr_20, 20, I_arr_like_10, size=1)
1131: (0) random_st.hypergeometric(20, I_arr_20, 10, size=1)
1132: (0) random_st.hypergeometric(I_arr_like_20, 20, I_arr_10)
1133: (0) random_st.hypergeometric(20, I_arr_like_20, 10)
1134: (0) random_st.hypergeometric(I_arr_20, I_arr_20, 10)
1135: (0) random_st.hypergeometric(I_arr_like_20, I_arr_like_20, 10)
1136: (0) random_st.hypergeometric(I_arr_20, I_arr_20, I_arr_10, size=1)
1137: (0) random_st.hypergeometric(I_arr_like_20, I_arr_like_20, I_arr_like_10, size=1)
1138: (0) random_st.randint(0, 100)
1139: (0) random_st.randint(100)
1140: (0) random_st.randint([100])
1141: (0) random_st.randint(0, [100])
1142: (0) random_st.randint(2, dtype=bool)
1143: (0) random_st.randint(0, 2, dtype=bool)
1144: (0) random_st.randint(I_bool_high_open, dtype=bool)
1145: (0) random_st.randint(I_bool_low, I_bool_high_open, dtype=bool)
1146: (0) random_st.randint(0, I_bool_high_open, dtype=bool)
1147: (0) random_st.randint(2, dtype=np.bool_)
1148: (0) random_st.randint(0, 2, dtype=np.bool_)
1149: (0) random_st.randint(I_bool_high_open, dtype=np.bool_)
1150: (0) random_st.randint(I_bool_low, I_bool_high_open, dtype=np.bool_)
1151: (0) random_st.randint(0, I_bool_high_open, dtype=np.bool_)
1152: (0) random_st.randint(256, dtype="u1")
1153: (0) random_st.randint(0, 256, dtype="u1")
1154: (0) random_st.randint(I_u1_high_open, dtype="u1")
1155: (0) random_st.randint(I_u1_low, I_u1_high_open, dtype="u1")
1156: (0) random_st.randint(0, I_u1_high_open, dtype="u1")
1157: (0) random_st.randint(256, dtype="uint8")
1158: (0) random_st.randint(0, 256, dtype="uint8")
1159: (0) random_st.randint(I_u1_high_open, dtype="uint8")
1160: (0) random_st.randint(I_u1_low, I_u1_high_open, dtype="uint8")
1161: (0) random_st.randint(0, I_u1_high_open, dtype="uint8")
1162: (0) random_st.randint(256, dtype=np.uint8)
1163: (0) random_st.randint(0, 256, dtype=np.uint8)
1164: (0) random_st.randint(I_u1_high_open, dtype=np.uint8)
1165: (0) random_st.randint(I_u1_low, I_u1_high_open, dtype=np.uint8)
1166: (0) random_st.randint(0, I_u1_high_open, dtype=np.uint8)
1167: (0) random_st.randint(65536, dtype="u2")

```

```

1168: (0) random_st.randint(0, 65536, dtype="u2")
1169: (0) random_st.randint(I_u2_high_open, dtype="u2")
1170: (0) random_st.randint(I_u2_low, I_u2_high_open, dtype="u2")
1171: (0) random_st.randint(0, I_u2_high_open, dtype="u2")
1172: (0) random_st.randint(65536, dtype="uint16")
1173: (0) random_st.randint(0, 65536, dtype="uint16")
1174: (0) random_st.randint(I_u2_high_open, dtype="uint16")
1175: (0) random_st.randint(I_u2_low, I_u2_high_open, dtype="uint16")
1176: (0) random_st.randint(0, I_u2_high_open, dtype="uint16")
1177: (0) random_st.randint(65536, dtype=np.uint16)
1178: (0) random_st.randint(0, 65536, dtype=np.uint16)
1179: (0) random_st.randint(I_u2_high_open, dtype=np.uint16)
1180: (0) random_st.randint(I_u2_low, I_u2_high_open, dtype=np.uint16)
1181: (0) random_st.randint(0, I_u2_high_open, dtype=np.uint16)
1182: (0) random_st.randint(4294967296, dtype="u4")
1183: (0) random_st.randint(0, 4294967296, dtype="u4")
1184: (0) random_st.randint(I_u4_high_open, dtype="u4")
1185: (0) random_st.randint(I_u4_low, I_u4_high_open, dtype="u4")
1186: (0) random_st.randint(0, I_u4_high_open, dtype="u4")
1187: (0) random_st.randint(4294967296, dtype="uint32")
1188: (0) random_st.randint(0, 4294967296, dtype="uint32")
1189: (0) random_st.randint(I_u4_high_open, dtype="uint32")
1190: (0) random_st.randint(I_u4_low, I_u4_high_open, dtype="uint32")
1191: (0) random_st.randint(0, I_u4_high_open, dtype="uint32")
1192: (0) random_st.randint(4294967296, dtype=np.uint32)
1193: (0) random_st.randint(0, 4294967296, dtype=np.uint32)
1194: (0) random_st.randint(I_u4_high_open, dtype=np.uint32)
1195: (0) random_st.randint(I_u4_low, I_u4_high_open, dtype=np.uint32)
1196: (0) random_st.randint(0, I_u4_high_open, dtype=np.uint32)
1197: (0) random_st.randint(18446744073709551616, dtype="u8")
1198: (0) random_st.randint(0, 18446744073709551616, dtype="u8")
1199: (0) random_st.randint(I_u8_high_open, dtype="u8")
1200: (0) random_st.randint(I_u8_low, I_u8_high_open, dtype="u8")
1201: (0) random_st.randint(0, I_u8_high_open, dtype="u8")
1202: (0) random_st.randint(18446744073709551616, dtype="uint64")
1203: (0) random_st.randint(0, 18446744073709551616, dtype="uint64")
1204: (0) random_st.randint(I_u8_high_open, dtype="uint64")
1205: (0) random_st.randint(I_u8_low, I_u8_high_open, dtype="uint64")
1206: (0) random_st.randint(0, I_u8_high_open, dtype="uint64")
1207: (0) random_st.randint(18446744073709551616, dtype=np.uint64)
1208: (0) random_st.randint(0, 18446744073709551616, dtype=np.uint64)
1209: (0) random_st.randint(I_u8_high_open, dtype=np.uint64)
1210: (0) random_st.randint(I_u8_low, I_u8_high_open, dtype=np.uint64)
1211: (0) random_st.randint(0, I_u8_high_open, dtype=np.uint64)
1212: (0) random_st.randint(128, dtype="i1")
1213: (0) random_st.randint(-128, 128, dtype="i1")
1214: (0) random_st.randint(I_i1_high_open, dtype="i1")
1215: (0) random_st.randint(I_i1_low, I_i1_high_open, dtype="i1")
1216: (0) random_st.randint(-128, I_i1_high_open, dtype="i1")
1217: (0) random_st.randint(128, dtype="int8")
1218: (0) random_st.randint(-128, 128, dtype="int8")
1219: (0) random_st.randint(I_i1_high_open, dtype="int8")
1220: (0) random_st.randint(I_i1_low, I_i1_high_open, dtype="int8")
1221: (0) random_st.randint(-128, I_i1_high_open, dtype="int8")
1222: (0) random_st.randint(128, dtype=np.int8)
1223: (0) random_st.randint(-128, 128, dtype=np.int8)
1224: (0) random_st.randint(I_i1_high_open, dtype=np.int8)
1225: (0) random_st.randint(I_i1_low, I_i1_high_open, dtype=np.int8)
1226: (0) random_st.randint(-128, I_i1_high_open, dtype=np.int8)
1227: (0) random_st.randint(32768, dtype="i2")
1228: (0) random_st.randint(-32768, 32768, dtype="i2")
1229: (0) random_st.randint(I_i2_high_open, dtype="i2")
1230: (0) random_st.randint(I_i2_low, I_i2_high_open, dtype="i2")
1231: (0) random_st.randint(-32768, I_i2_high_open, dtype="i2")
1232: (0) random_st.randint(32768, dtype="int16")
1233: (0) random_st.randint(-32768, 32768, dtype="int16")
1234: (0) random_st.randint(I_i2_high_open, dtype="int16")
1235: (0) random_st.randint(I_i2_low, I_i2_high_open, dtype="int16")
1236: (0) random_st.randint(-32768, I_i2_high_open, dtype="int16")

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1237: (0) random_st.randint(32768, dtype=np.int16)
1238: (0) random_st.randint(-32768, 32768, dtype=np.int16)
1239: (0) random_st.randint(I_i2_high_open, dtype=np.int16)
1240: (0) random_st.randint(I_i2_low, I_i2_high_open, dtype=np.int16)
1241: (0) random_st.randint(-32768, I_i2_high_open, dtype=np.int16)
1242: (0) random_st.randint(2147483648, dtype="i4")
1243: (0) random_st.randint(-2147483648, 2147483648, dtype="i4")
1244: (0) random_st.randint(I_i4_high_open, dtype="i4")
1245: (0) random_st.randint(I_i4_low, I_i4_high_open, dtype="i4")
1246: (0) random_st.randint(-2147483648, I_i4_high_open, dtype="i4")
1247: (0) random_st.randint(2147483648, dtype="int32")
1248: (0) random_st.randint(-2147483648, 2147483648, dtype="int32")
1249: (0) random_st.randint(I_i4_high_open, dtype="int32")
1250: (0) random_st.randint(I_i4_low, I_i4_high_open, dtype="int32")
1251: (0) random_st.randint(-2147483648, I_i4_high_open, dtype="int32")
1252: (0) random_st.randint(2147483648, dtype=np.int32)
1253: (0) random_st.randint(-2147483648, 2147483648, dtype=np.int32)
1254: (0) random_st.randint(I_i4_high_open, dtype=np.int32)
1255: (0) random_st.randint(I_i4_low, I_i4_high_open, dtype=np.int32)
1256: (0) random_st.randint(-2147483648, I_i4_high_open, dtype=np.int32)
1257: (0) random_st.randint(9223372036854775808, dtype="i8")
1258: (0) random_st.randint(-9223372036854775808, 9223372036854775808, dtype="i8")
1259: (0) random_st.randint(I_i8_high_open, dtype="i8")
1260: (0) random_st.randint(I_i8_low, I_i8_high_open, dtype="i8")
1261: (0) random_st.randint(-9223372036854775808, I_i8_high_open, dtype="i8")
1262: (0) random_st.randint(9223372036854775808, dtype="int64")
1263: (0) random_st.randint(-9223372036854775808, 9223372036854775808, dtype="int64")
1264: (0) random_st.randint(I_i8_high_open, dtype="int64")
1265: (0) random_st.randint(I_i8_low, I_i8_high_open, dtype="int64")
1266: (0) random_st.randint(-9223372036854775808, I_i8_high_open, dtype="int64")
1267: (0) random_st.randint(9223372036854775808, dtype=np.int64)
1268: (0) random_st.randint(-9223372036854775808, 9223372036854775808, dtype=np.int64)
1269: (0) random_st.randint(I_i8_high_open, dtype=np.int64)
1270: (0) random_st.randint(I_i8_low, I_i8_high_open, dtype=np.int64)
1271: (0) random_st.randint(-9223372036854775808, I_i8_high_open, dtype=np.int64)
1272: (0) bg: np.random.BitGenerator = random_st._bit_generator
1273: (0) random_st.bytes(2)
1274: (0) random_st.choice(5)
1275: (0) random_st.choice(5, 3)
1276: (0) random_st.choice(5, 3, replace=True)
1277: (0) random_st.choice(5, 3, p=[1 / 5] * 5)
1278: (0) random_st.choice(5, 3, p=[1 / 5] * 5, replace=False)
1279: (0) random_st.choice(["pooh", "rabbit", "piglet", "Christopher"])
1280: (0) random_st.choice(["pooh", "rabbit", "piglet", "Christopher"], 3)
1281: (0) random_st.choice(["pooh", "rabbit", "piglet", "Christopher"], 3, p=[1 / 4] * 4)
1282: (0) random_st.choice(["pooh", "rabbit", "piglet", "Christopher"], 3, replace=True)
1283: (0) random_st.choice(["pooh", "rabbit", "piglet", "Christopher"], 3, replace=False, p=np.array([1 / 8, 1 / 8, 1 / 2, 1 / 4]))
1284: (0) random_st.dirichlet([0.5, 0.5])
1285: (0) random_st.dirichlet(np.array([0.5, 0.5]))
1286: (0) random_st.dirichlet(np.array([0.5, 0.5]), size=3)
1287: (0) random_st.multinomial(20, [1 / 6.0] * 6)
1288: (0) random_st.multinomial(20, np.array([0.5, 0.5]))
1289: (0) random_st.multinomial(20, [1 / 6.0] * 6, size=2)
1290: (0) random_st.multivariate_normal([0.0], [[1.0]])
1291: (0) random_st.multivariate_normal([0.0], np.array([[1.0]]))
1292: (0) random_st.multivariate_normal(np.array([0.0]), [[1.0]])
1293: (0) random_st.multivariate_normal([0.0], np.array([[1.0]]))
1294: (0) random_st.permutation(10)
1295: (0) random_st.permutation([1, 2, 3, 4])
1296: (0) random_st.permutation(np.array([1, 2, 3, 4]))
1297: (0) random_st.permutation(D_2D)
1298: (0) random_st.shuffle(np.arange(10))
1299: (0) random_st.shuffle([1, 2, 3, 4, 5])
1300: (0) random_st.shuffle(D_2D)
1301: (0) np.random.RandomState(SEED_PCG64)
1302: (0) np.random.RandomState(0)
1303: (0) np.random.RandomState([0, 1, 2])

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY\_combined\_python\_files\_20\_chars.txt

```

1304: (0) random_st.__str__()
1305: (0) random_st.__repr__()
1306: (0) random_st.state = random_st.__getstate__()
1307: (0) random_st.__setstate__(random_st.state)
1308: (0) random_st.seed()
1309: (0) random_st.seed(1)
1310: (0) random_st.seed([0, 1])
1311: (0) random_st_get_state = random_st.get_state()
1312: (0) random_st_get_state_legacy = random_st.get_state(legacy=True)
1313: (0) random_st.set_state(random_st_get_state)
1314: (0) random_st.rand()
1315: (0) random_st.rand(1)
1316: (0) random_st.rand(1, 2)
1317: (0) random_st.randn()
1318: (0) random_st.randn(1)
1319: (0) random_st.randn(1, 2)
1320: (0) random_st.random_sample()
1321: (0) random_st.random_sample(1)
1322: (0) random_st.random_sample(size=(1, 2))
1323: (0) random_st.tomaxint()
1324: (0) random_st.tomaxint(1)
1325: (0) random_st.tomaxint((1,))
1326: (0) np.random.set_bit_generator(SEED_PCG64)
1327: (0) np.random.get_bit_generator()

```

---

## File 377 - scalars.py:

```

1: (0)         import sys
2: (0)         import datetime as dt
3: (0)         import pytest
4: (0)         import numpy as np
5: (0)         b = np.bool_()
6: (0)         u8 = np.uint64()
7: (0)         i8 = np.int64()
8: (0)         f8 = np.float64()
9: (0)         c16 = np.complex128()
10: (0)        U = np.str_()
11: (0)        S = np.bytes_()
12: (0)        class D:
13: (4)            def __index__(self) -> int:
14: (8)                return 0
15: (0)        class C:
16: (4)            def __complex__(self) -> complex:
17: (8)                return 3j
18: (0)        class B:
19: (4)            def __int__(self) -> int:
20: (8)                return 4
21: (0)        class A:
22: (4)            def __float__(self) -> float:
23: (8)                return 4.0
24: (0)            np.complex64(3j)
25: (0)            np.complex64(A())
26: (0)            np.complex64(C())
27: (0)            np.complex128(3j)
28: (0)            np.complex128(C())
29: (0)            np.complex128(None)
30: (0)            np.complex64("1.2")
31: (0)            np.complex128(b"2j")
32: (0)            np.int8(4)
33: (0)            np.int16(3.4)
34: (0)            np.int32(4)
35: (0)            np.int64(-1)
36: (0)            np.uint8(B())
37: (0)            np.uint32()
38: (0)            np.int32("1")
39: (0)            np.int64(b"2")
40: (0)            np.float16(A())

```

```
41: (0) np.float32(16)
42: (0) np.float64(3.0)
43: (0) np.float64(None)
44: (0) np.float32("1")
45: (0) np.float16(b"2.5")
46: (0) np.uint64(D())
47: (0) np.float32(D())
48: (0) np.complex64(D())
49: (0) np.bytes_(b"hello")
50: (0) np.bytes_("hello", 'utf-8')
51: (0) np.bytes_("hello", encoding='utf-8')
52: (0) np.str_("hello")
53: (0) np.str_(b"hello", 'utf-8')
54: (0) np.str_(b"hello", encoding='utf-8')
55: (0) np.int8().real
56: (0) np.int16().imag
57: (0) np.int32().data
58: (0) np.int64().flags
59: (0) np.uint8().itemsize * 2
60: (0) np.uint16().ndim + 1
61: (0) np.uint32().strides
62: (0) np.uint64().shape
63: (0) np.datetime64()
64: (0) np.datetime64(0, "D")
65: (0) np.datetime64(0, b"D")
66: (0) np.datetime64(0, ('ms', 3))
67: (0) np.datetime64("2019")
68: (0) np.datetime64(b"2019")
69: (0) np.datetime64("2019", "D")
70: (0) np.datetime64(np.datetime64())
71: (0) np.datetime64(dt.datetime(2000, 5, 3))
72: (0) np.datetime64(dt.date(2000, 5, 3))
73: (0) np.datetime64(None)
74: (0) np.datetime64(None, "D")
75: (0) np.timedelta64()
76: (0) np.timedelta64(0)
77: (0) np.timedelta64(0, "D")
78: (0) np.timedelta64(0, ('ms', 3))
79: (0) np.timedelta64(0, b"D")
80: (0) np.timedelta64("3")
81: (0) np.timedelta64(b"5")
82: (0) np.timedelta64(np.timedelta64(2))
83: (0) np.timedelta64(dt.timedelta(2))
84: (0) np.timedelta64(None)
85: (0) np.timedelta64(None, "D")
86: (0) np.void(1)
87: (0) np.void(np.int64(1))
88: (0) np.void(True)
89: (0) np.void(np.bool_(True))
90: (0) np.void(b"test")
91: (0) np.void(np.bytes_("test"))
92: (0) np.void(object(), [("a", "0"), ("b", "0")])
93: (0) np.void(object(), dtype=[("a", "0"), ("b", "0")])
94: (0) i8 = np.int64()
95: (0) u8 = np.uint64()
96: (0) f8 = np.float64()
97: (0) c16 = np.complex128()
98: (0) b_ = np.bool_()
99: (0) td = np.timedelta64()
100: (0) U = np.str_("1")
101: (0) S = np.bytes_("1")
102: (0) AR = np.array(1, dtype=np.float64)
103: (0) int(i8)
104: (0) int(u8)
105: (0) int(f8)
106: (0) int(b_)
107: (0) int(td)
108: (0) int(U)
109: (0) int(S)
```

```
110: (0)          int(AR)
111: (0)          with pytest.warns(np.ComplexWarning):
112: (4)            int(c16)
113: (0)            float(i8)
114: (0)            float(u8)
115: (0)            float(f8)
116: (0)            float(b_)
117: (0)            float(td)
118: (0)            float(U)
119: (0)            float(S)
120: (0)            float(AR)
121: (0)          with pytest.warns(np.ComplexWarning):
122: (4)            float(c16)
123: (0)            complex(i8)
124: (0)            complex(u8)
125: (0)            complex(f8)
126: (0)            complex(c16)
127: (0)            complex(b_)
128: (0)            complex(td)
129: (0)            complex(U)
130: (0)            complex(AR)
131: (0)            c16.dtype
132: (0)            c16.real
133: (0)            c16.imag
134: (0)            c16.real.real
135: (0)            c16.real.imag
136: (0)            c16.ndim
137: (0)            c16.size
138: (0)            c16.itemsize
139: (0)            c16.shape
140: (0)            c16.strides
141: (0)            c16.squeeze()
142: (0)            c16.byteswap()
143: (0)            c16.transpose()
144: (0)            np.string_()
145: (0)            np.byte()
146: (0)            np.short()
147: (0)            np.intc()
148: (0)            np.intp()
149: (0)            np.int_()
150: (0)            np.longlong()
151: (0)            np.ubyte()
152: (0)            np.ushort()
153: (0)            np.uintc()
154: (0)            np.uintp()
155: (0)            np.uint()
156: (0)            np.ulonglong()
157: (0)            np.half()
158: (0)            np.single()
159: (0)            np.double()
160: (0)            np.float_()
161: (0)            np.longdouble()
162: (0)            np.longfloat()
163: (0)            np.csingle()
164: (0)            np.singlecomplex()
165: (0)            np.cdouble()
166: (0)            np.complex_()
167: (0)            np.cfloat()
168: (0)            np.clongdouble()
169: (0)            np.clongfloat()
170: (0)            np.longcomplex()
171: (0)            b.item()
172: (0)            i8.item()
173: (0)            u8.item()
174: (0)            f8.item()
175: (0)            c16.item()
176: (0)            U.item()
177: (0)            S.item()
178: (0)            b.tolist()
```

```

179: (0)         i8.tolist()
180: (0)         u8.tolist()
181: (0)         f8.tolist()
182: (0)         c16.tolist()
183: (0)         U.tolist()
184: (0)         S.tolist()
185: (0)         b.ravel()
186: (0)         i8.ravel()
187: (0)         u8.ravel()
188: (0)         f8.ravel()
189: (0)         c16.ravel()
190: (0)         U.ravel()
191: (0)         S.ravel()
192: (0)         b.flatten()
193: (0)         i8.flatten()
194: (0)         u8.flatten()
195: (0)         f8.flatten()
196: (0)         c16.flatten()
197: (0)         U.flatten()
198: (0)         S.flatten()
199: (0)         b.reshape(1)
200: (0)         i8.reshape(1)
201: (0)         u8.reshape(1)
202: (0)         f8.reshape(1)
203: (0)         c16.reshape(1)
204: (0)         U.reshape(1)
205: (0)         S.reshape(1)

```

-----

## File 378 - simple.py:

```

1: (0)         """Simple expression that should pass with mypy."""
2: (0)         import operator
3: (0)         import numpy as np
4: (0)         from collections.abc import Iterable
5: (0)         array = np.array([1, 2])
6: (0)         def ndarray_func(x):
7: (4)             return x
8: (0)         ndarray_func(np.array([1, 2]))
9: (0)         array == 1
10: (0)        array.dtype == float
11: (0)        np.dtype(float)
12: (0)        np.dtype(np.float64)
13: (0)        np.dtype(None)
14: (0)        np.dtype("float64")
15: (0)        np.dtype(np.dtype(float))
16: (0)        np.dtype(("U", 10))
17: (0)        np.dtype((np.int32, (2, 2)))
18: (0)        two_tuples_dtype = [("R", "u1"), ("G", "u1"), ("B", "u1")]
19: (0)        np.dtype(two_tuples_dtype)
20: (0)        three_tuples_dtype = [("R", "u1", 2)]
21: (0)        np.dtype(three_tuples_dtype)
22: (0)        mixed_tuples_dtype = [("R", "u1"), ("G", np.str_, 1)]
23: (0)        np.dtype(mixed_tuples_dtype)
24: (0)        shape_tuple_dtype = [("R", "u1", (2, 2))]
25: (0)        np.dtype(shape_tuple_dtype)
26: (0)        shape_like_dtype = [("R", "u1", (2, 2)), ("G", np.str_, 1)]
27: (0)        np.dtype(shape_like_dtype)
28: (0)        object_dtype = [("field1", object)]
29: (0)        np.dtype(object_dtype)
30: (0)        np.dtype((np.int32, (np.int8, 4)))
31: (0)        np.dtype(float) == float
32: (0)        np.dtype(float) != np.float64
33: (0)        np.dtype(float) < None
34: (0)        np.dtype(float) <= "float64"
35: (0)        np.dtype(float) > np.dtype(float)
36: (0)        np.dtype(float) >= np.dtype(("U", 10))
37: (0)        def iterable_func(x):

```

```
38: (4)           return x
39: (0)           iterable_func(array)
40: (0)           [element for element in array]
41: (0)           iter(array)
42: (0)           zip(array, array)
43: (0)           array[1]
44: (0)           array[:]
45: (0)           array[...]
46: (0)           array[:] = 0
47: (0)           array_2d = np.ones((3, 3))
48: (0)           array_2d[:2, :2]
49: (0)           array_2d[..., 0]
50: (0)           array_2d[:2, :2] = 0
51: (0)           len(array)
52: (0)           str(array)
53: (0)           array_scalar = np.array(1)
54: (0)           int(array_scalar)
55: (0)           float(array_scalar)
56: (0)           bytes(array_scalar)
57: (0)           operator.index(array_scalar)
58: (0)           bool(array_scalar)
59: (0)           array < 1
60: (0)           array <= 1
61: (0)           array == 1
62: (0)           array != 1
63: (0)           array > 1
64: (0)           array >= 1
65: (0)           1 < array
66: (0)           1 <= array
67: (0)           1 == array
68: (0)           1 != array
69: (0)           1 > array
70: (0)           1 >= array
71: (0)           array + 1
72: (0)           1 + array
73: (0)           array += 1
74: (0)           array - 1
75: (0)           1 - array
76: (0)           array -= 1
77: (0)           array * 1
78: (0)           1 * array
79: (0)           array *= 1
80: (0)           nonzero_array = np.array([1, 2])
81: (0)           array / 1
82: (0)           1 / nonzero_array
83: (0)           float_array = np.array([1.0, 2.0])
84: (0)           float_array /= 1
85: (0)           array // 1
86: (0)           1 // nonzero_array
87: (0)           array // 1
88: (0)           array % 1
89: (0)           1 % nonzero_array
90: (0)           array %= 1
91: (0)           divmod(array, 1)
92: (0)           divmod(1, nonzero_array)
93: (0)           array ** 1
94: (0)           1 ** array
95: (0)           array **= 1
96: (0)           array << 1
97: (0)           1 << array
98: (0)           array <= 1
99: (0)           array >> 1
100: (0)          1 >> array
101: (0)          array >>= 1
102: (0)          array & 1
103: (0)          1 & array
104: (0)          array &= 1
105: (0)          array ^ 1
106: (0)          1 ^ array
```

```

107: (0)         array ^= 1
108: (0)         array | 1
109: (0)         1 | array
110: (0)         array |= 1
111: (0)         ~array
112: (0)         +array
113: (0)         abs(array)
114: (0)         ~array
115: (0)         np.array([1, 2]).transpose()
-----
```

File 379 - simple\_py3.py:

```

1: (0)             import numpy as np
2: (0)             array = np.array([1, 2])
3: (0)             array @ array
-----
```

File 380 - ufuncs.py:

```

1: (0)             import numpy as np
2: (0)             np.sin(1)
3: (0)             np.sin([1, 2, 3])
4: (0)             np.sin(1, out=np.empty(1))
5: (0)             np.matmul(np.ones((2, 2, 2)), np.ones((2, 2, 2)), axes=[(0, 1), (0, 1)])
6: (0)             np.sin(1, signature="D->D")
7: (0)             np.sin(1, extobj=[16, 1, lambda: None])
8: (0)             np.sin.types[0]
9: (0)             np.sin.__name__
10: (0)            np.sin.__doc__
11: (0)            np.abs(np.array([1]))
-----
```

File 381 - ufunclike.py:

```

1: (0)             from __future__ import annotations
2: (0)             from typing import Any
3: (0)             import numpy as np
4: (0)             class Object:
5: (4)                 def __ceil__(self) -> Object:
6: (8)                     return self
7: (4)                 def __floor__(self) -> Object:
8: (8)                     return self
9: (4)                 def __ge__(self, value: object) -> bool:
10: (8)                     return True
11: (4)                 def __array__(self) -> np.ndarray[Any, np.dtype[np.object_]]:
12: (8)                     ret = np.empty(()), dtype=object)
13: (8)                     ret[()] = self
14: (8)                     return ret
15: (0)             AR_LIKE_b = [True, True, False]
16: (0)             AR_LIKE_u = [np.uint32(1), np.uint32(2), np.uint32(3)]
17: (0)             AR_LIKE_i = [1, 2, 3]
18: (0)             AR_LIKE_f = [1.0, 2.0, 3.0]
19: (0)             AR_LIKE_O = [Object(), Object(), Object()]
20: (0)             AR_U: np.ndarray[Any, np.dtype[np.str_]] = np.zeros(3, dtype="U5")
21: (0)             np.fix(AR_LIKE_b)
22: (0)             np.fix(AR_LIKE_u)
23: (0)             np.fix(AR_LIKE_i)
24: (0)             np.fix(AR_LIKE_f)
25: (0)             np.fix(AR_LIKE_O)
26: (0)             np.fix(AR_LIKE_f, out=AR_U)
27: (0)             np.isposinf(AR_LIKE_b)
28: (0)             np.isposinf(AR_LIKE_u)
29: (0)             np.isposinf(AR_LIKE_i)
30: (0)             np.isposinf(AR_LIKE_f)
-----
```

```

31: (0)          np.isposinf(AR_LIKE_f, out=AR_U)
32: (0)          np.isneginf(AR_LIKE_b)
33: (0)          np.isneginf(AR_LIKE_u)
34: (0)          np.isneginf(AR_LIKE_i)
35: (0)          np.isneginf(AR_LIKE_f)
36: (0)          np.isneginf(AR_LIKE_f, out=AR_U)

```

-----

## File 382 - ufunc\_config.py:

```

1: (0)          """Typing tests for `numpy.core._ufunc_config`."""
2: (0)          import numpy as np
3: (0)          def func1(a: str, b: int) -> None:
4: (4)              return None
5: (0)          def func2(a: str, b: int, c: float = 1.0) -> None:
6: (4)              return None
7: (0)          def func3(a: str, b: int) -> int:
8: (4)              return 0
9: (0)          class Write1:
10: (4)              def write(self, a: str) -> None:
11: (8)                  return None
12: (0)          class Write2:
13: (4)              def write(self, a: str, b: int = 1) -> None:
14: (8)                  return None
15: (0)          class Write3:
16: (4)              def write(self, a: str) -> int:
17: (8)                  return 0
18: (0)          _err_default = np.geterr()
19: (0)          _bufsize_default = np.getbufsize()
20: (0)          _errcall_default = np.geterrcall()
21: (0)          try:
22: (4)              np.seterr(all=None)
23: (4)              np.seterr(divide="ignore")
24: (4)              np.seterr(over="warn")
25: (4)              np.seterr(under="call")
26: (4)              np.seterr(invalid="raise")
27: (4)              np.geterr()
28: (4)              np.setbufsize(4096)
29: (4)              np.getbufsize()
30: (4)              np.seterrcall(func1)
31: (4)              np.seterrcall(func2)
32: (4)              np.seterrcall(func3)
33: (4)              np.seterrcall(Write1())
34: (4)              np.seterrcall(Write2())
35: (4)              np.seterrcall(Write3())
36: (4)              np.geterrcall()
37: (4)              with np.errstate(call=func1, all="call"):
38: (8)                  pass
39: (4)              with np.errstate(call=Write1(), divide="log", over="log"):
40: (8)                  pass
41: (0)          finally:
42: (4)              np.seterr(**_err_default)
43: (4)              np.setbufsize(_bufsize_default)
44: (4)              np.seterrcall(_errcall_default)

```

-----

## File 383 - warnings\_and\_errors.py:

```

1: (0)          import numpy as np
2: (0)          np.AxisError("test")
3: (0)          np.AxisError(1, ndim=2)
4: (0)          np.AxisError(1, ndim=2, msg_prefix="error")
5: (0)          np.AxisError(1, ndim=2, msg_prefix=None)

```

-----

## File 384 - umath.py:

```

1: (0)          from numpy.core import umath
2: (0)          _globals = globals()
3: (0)          for item in umath.__dir__():
4: (4)            _globals[item] = getattr(umath, item)
-----
```

File 385 - multiarray.py:

```

1: (0)          from numpy.core import multiarray
2: (0)          _globals = globals()
3: (0)          for item in multiarray.__dir__():
4: (4)            _globals[item] = getattr(multiarray, item)
-----
```

File 386 - \_dtype.py:

```

1: (0)          from numpy.core import _dtype
2: (0)          _globals = globals()
3: (0)          for item in _dtype.__dir__():
4: (4)            _globals[item] = getattr(_dtype, item)
-----
```

File 387 - \_dtype\_ctypes.py:

```

1: (0)          from numpy.core import _dtype_ctypes
2: (0)          _globals = globals()
3: (0)          for item in _dtype_ctypes.__dir__():
4: (4)            _globals[item] = getattr(_dtype_ctypes, item)
-----
```

File 388 - \_internal.py:

```

1: (0)          from numpy.core import _internal
2: (0)          _globals = globals()
3: (0)          for item in _internal.__dir__():
4: (4)            _globals[item] = getattr(_internal, item)
-----
```

File 389 - \_\_init\_\_.py:

```

1: (0)          """
2: (0)          This private module only contains stubs for interoperability with
3: (0)          NumPy 2.0 pickled arrays. It may not be used by the end user.
4: (0)          """
-----
```

File 390 - \_multiarray\_umath.py:

```

1: (0)          from numpy.core import _multiarray_umath
2: (0)          _globals = globals()
3: (0)          for item in _multiarray_umath.__dir__():
4: (4)            _globals[item] = getattr(_multiarray_umath, item)
-----
```

File 391 - hook-numpy.py:

```

1: (0)          """This hook should collect all binary files and any hidden modules that numpy
2: (0)          needs.
3: (0)          Our (some-what inadequate) docs for writing PyInstaller hooks are kept here:
4: (0)          https://pyinstaller.readthedocs.io/en/stable/hooks.html
5: (0)          """
-----
```

```
SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

6: (0)     from PyInstaller.compat import is_conda, is_pure_conda
7: (0)     from PyInstaller.utils.hooks import collect_dynamic_libs, is_module_satisfies
8: (0)     binaries = collect_dynamic_libs("numpy", ".")
9: (0)     if is_pure_conda:
10: (4)         from PyInstaller.utils.hooks import conda_support
11: (4)         datas = conda_support.collect_dynamic_libs("numpy", dependencies=True)
12: (0)     hiddenimports = ['numpy.core._dtype_ctypes', 'numpy.core._multiarray_tests']
13: (0)     excludedimports = [
14: (4)         "scipy",
15: (4)         "pytest",
16: (4)         "f2py",
17: (4)         "setuptools",
18: (4)         "numpy.f2py",
19: (4)         "distutils",
20: (4)         "numpy.distutils",
21: (0)     ]
-----
```

## File 392 - pyinstaller-smoke.py:

```
1: (0)     """A crude *bit of everything* smoke test to verify PyInstaller compatibility.
2: (0)     PyInstaller typically goes wrong by forgetting to package modules, extension
3: (0)     modules or shared libraries. This script should aim to touch as many of those
4: (0)     as possible in an attempt to trip a ModuleNotFoundError or a DLL load failure
5: (0)     due to an uncollected resource. Missing resources are unlikely to lead to
6: (0)     arithmetic errors so there's generally no need to verify any calculation's
7: (0)     output - merely that it made it to the end OK. This script should not
8: (0)     explicitly import any of numpy's submodules as that gives PyInstaller undue
9: (0)     hints that those submodules exist and should be collected (accessing
implicitly
10: (0)     loaded submodules is OK).
11: (0) """
12: (0)     import numpy as np
13: (0)     a = np.arange(1., 10.).reshape((3, 3)) % 5
14: (0)     np.linalg.det(a)
15: (0)     a @ a
16: (0)     a @ a.T
17: (0)     np.linalg.inv(a)
18: (0)     np.sin(np.exp(a))
19: (0)     np.linalg.svd(a)
20: (0)     np.linalg.eigh(a)
21: (0)     np.unique(np.random.randint(0, 10, 100))
22: (0)     np.sort(np.random.uniform(0, 10, 100))
23: (0)     np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
24: (0)     np.ma.masked_array(np.arange(10), np.random.rand(10) < .5).sum()
25: (0)     np.polynomial.Legendre([7, 8, 9]).roots()
26: (0)     print("I made it!")
-----
```

## File 393 - test\_pyinstaller.py:

```
1: (0)     import subprocess
2: (0)     from pathlib import Path
3: (0)     import pytest
4: (0)     @pytest.mark.filterwarnings('ignore::DeprecationWarning')
5: (0)     @pytest.mark.filterwarnings('ignore::ResourceWarning')
6: (0)     @pytest.mark.parametrize("mode", ["--onedir", "--onefile"])
7: (0)     @pytest.mark.slow
8: (0)     def test_pyinstaller(mode, tmp_path):
9: (4)         """Compile and run pyinstaller-smoke.py using PyInstaller."""
10: (4)         pyinstaller_cli = pytest.importorskip("PyInstaller.__main__").run
11: (4)         source = Path(__file__).with_name("pyinstaller-smoke.py").resolve()
12: (4)         args = [
13: (8)             '--workpath', str(tmp_path / "build"),
14: (8)             '--distpath', str(tmp_path / "dist"),
15: (8)             '--specpath', str(tmp_path),
16: (8)             mode,
```

```

17: (8)             str(source),
18: (4)         ]
19: (4)     pyinstaller_cli(args)
20: (4)     if mode == "--onefile":
21: (8)         exe = tmp_path / "dist" / source.stem
22: (4)     else:
23: (8)         exe = tmp_path / "dist" / source.stem / source.stem
24: (4)     p = subprocess.run([str(exe)], check=True, stdout=subprocess.PIPE)
25: (4)     assert p.stdout.strip() == b"I made it!"

```

-----

File 394 - setup.py:

```

1: (0)         def configuration(parent_package='', top_path=None):
2: (4)             from numpy.distutils.misc_util import Configuration
3: (4)             config = Configuration('_typing', parent_package, top_path)
4: (4)             config.add_data_files('*/*.pyi')
5: (4)             return config
6: (0)         if __name__ == '__main__':
7: (4)             from numpy.distutils.core import setup
8: (4)             setup(configuration=configuration)

```

-----

File 395 - \_\_init\_\_.py:

```
1: (0)
```

-----

File 396 - \_add\_docstring.py:

```

1: (0)         """A module for creating docstrings for sphinx ``data`` domains."""
2: (0)         import re
3: (0)         import textwrap
4: (0)         from ._array_like import NDArray
5: (0)         _docstrings_list = []
6: (0)         def add_newdoc(name: str, value: str, doc: str) -> None:
7: (4)             """Append ``_docstrings_list`` with a docstring for `name`.
8: (4)             Parameters
9: (4)             -----
10: (4)             name : str
11: (8)                 The name of the object.
12: (4)             value : str
13: (8)                 A string-representation of the object.
14: (4)             doc : str
15: (8)                 The docstring of the object.
16: (4)             """
17: (4)             _docstrings_list.append((name, value, doc))
18: (0)         def _parse_docstrings() -> str:
19: (4)             """Convert all docstrings in ``_docstrings_list`` into a single
20: (4)             sphinx-legible text block.
21: (4)             """
22: (4)             type_list_ret = []
23: (4)             for name, value, doc in _docstrings_list:
24: (8)                 s = textwrap.dedent(doc).replace("\n", "\n      ")
25: (8)                 lines = s.split("\n")
26: (8)                 new_lines = []
27: (8)                 indent = ""
28: (8)                 for line in lines:
29: (12)                     m = re.match(r'^(\s+)[-]+\s*$', line)
30: (12)                     if m and new_lines:
31: (16)                         prev = textwrap.dedent(new_lines.pop())
32: (16)                         if prev == "Examples":
33: (20)                             indent = ""
34: (20)                             new_lines.append(f'{m.group(1)}.. rubric:: {prev}')
35: (16)                         else:
36: (20)                             indent = 4 * " "

```

```

37: (20)                     new_lines.append(f'{m.group(1)}.. admonition:: {prev}')
38: (16)                     new_lines.append("")
39: (12)                 else:
40: (16)                     new_lines.append(f"{indent}{line}")
41: (8)                     s = "\n".join(new_lines)
42: (8)                     s_block = f""".. data:: {name}\n      :value: {value}\n      {s}"""
43: (8)                     type_list_ret.append(s_block)
44: (4)                     return "\n".join(type_list_ret)
45: (0) add_newdoc('ArrayLike', 'typing.Union[...]',
46: (4)                     """
47: (4)                         A `~typing.Union` representing objects that can be coerced
48: (4)                         into an `~numpy.ndarray`.
49: (4)                         Among others this includes the likes of:
50: (4)                             * Scalars.
51: (4)                             * (Nested) sequences.
52: (4)                             * Objects implementing the `~class.__array__` protocol.
53: (4)                             .. versionadded:: 1.20
54: (4)                         See Also
55: (4)                         -----
56: (4)                         :term:`array_like`:
57: (8)                             Any scalar or sequence that can be interpreted as an ndarray.
58: (4)                         Examples
59: (4)                         -----
60: (4)                         .. code-block:: python
61: (8)                             >>> import numpy as np
62: (8)                             >>> import numpy.typing as npt
63: (8)                             >>> def as_array(a: npt.ArrayLike) -> np.ndarray:
64: (8)                                 ...     return np.array(a)
65: (4)                             """
66: (0) add_newdoc('DTypeLike', 'typing.Union[...]',
67: (4)                     """
68: (4)                         A `~typing.Union` representing objects that can be coerced
69: (4)                         into a `~numpy.dtype`.
70: (4)                         Among others this includes the likes of:
71: (4)                             * :class:`type` objects.
72: (4)                             * Character codes or the names of :class:`type` objects.
73: (4)                             * Objects with the ``.dtype`` attribute.
74: (4)                             .. versionadded:: 1.20
75: (4)                         See Also
76: (4)                         -----
77: (4)                         :ref:`Specifying and constructing data types <arrays.dtypes.constructing>`  

78: (8)                             A comprehensive overview of all objects that can be coerced
79: (8)                             into data types.
80: (4)                         Examples
81: (4)                         -----
82: (4)                         .. code-block:: python
83: (8)                             >>> import numpy as np
84: (8)                             >>> import numpy.typing as npt
85: (8)                             >>> def as_dtype(d: npt.DTypeLike) -> np.dtype:
86: (8)                                 ...     return np.dtype(d)
87: (4)                             """
88: (0) add_newdoc('NDArray', repr(NDArray),
89: (4)                     """
90: (4)                         A :term:`generic <generic type>` version of
91: (4)                         `np.ndarray[Any, np.dtype[+ScalarType]] <numpy.ndarray>`.
92: (4)                         Can be used during runtime for typing arrays with a given dtype
93: (4)                         and unspecified shape.
94: (4)                         .. versionadded:: 1.21
95: (4)                         Examples
96: (4)                         -----
97: (4)                         .. code-block:: python
98: (8)                             >>> import numpy as np
99: (8)                             >>> import numpy.typing as npt
100: (8)                            >>> print(npt.NDArray)
101: (8)                            numpy.ndarray[typing.Any, numpy.dtype[+ScalarType]]
102: (8)                            >>> print(npt.NDArray[np.float64])
103: (8)                            numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]
104: (8)                            >>> NDArrayInt = npt.NDArray[np.int_]
105: (8)                            >>> a: NDArrayInt = np.arange(10)

```

```

106: (8)         >>> def func(a: npt.ArrayLike) -> npt.NDArray[Any]:
107: (8)             ...     return np.array(a)
108: (4)             """")
109: (0)             _docstrings = _parse_docstrings()
-----
```

File 397 - \_array\_like.py:

```

1: (0)         from __future__ import annotations
2: (0)         import sys
3: (0)         from collections.abc import Collection, Callable, Sequence
4: (0)         from typing import Any, Protocol, Union, TypeVar, runtime_checkable
5: (0)         from numpy import (
6: (4)             ndarray,
7: (4)             dtype,
8: (4)             generic,
9: (4)             bool_,
10: (4)            unsignedinteger,
11: (4)            integer,
12: (4)            floating,
13: (4)            complexfloating,
14: (4)            number,
15: (4)            timedelta64,
16: (4)            datetime64,
17: (4)            object_,
18: (4)            void,
19: (4)            str_,
20: (4)            bytes_,
21: (0)        )
22: (0)        from ._nested_sequence import _NestedSequence
23: (0)        _T = TypeVar("_T")
24: (0)        _ScalarType = TypeVar("_ScalarType", bound=generic)
25: (0)        _ScalarType_co = TypeVar("_ScalarType_co", bound=generic, covariant=True)
26: (0)        _DType = TypeVar("_DType", bound=dtype[Any])
27: (0)        _DType_co = TypeVar("_DType_co", covariant=True, bound=dtype[Any])
28: (0)        NDArray = ndarray[Any, dtype[_ScalarType_co]]
29: (0)        @runtime_checkable
30: (0)        class _SupportsArray(Protocol[_DType_co]):
31: (4)            def __array__(self) -> ndarray[Any, _DType_co]: ...
32: (0)        @runtime_checkable
33: (0)        class _SupportsArrayFunc(Protocol):
34: (4)            """A protocol class representing `~class.__array_function__`."""
35: (4)            def __array_function__(
36: (8)                self,
37: (8)                func: Callable[..., Any],
38: (8)                types: Collection[type[Any]],
39: (8)                args: tuple[Any, ...],
40: (8)                kwargs: dict[str, Any],
41: (4)            ) -> object: ...
42: (0)        _FiniteNestedSequence = Union[
43: (4)            _T,
44: (4)            Sequence[_T],
45: (4)            Sequence[Sequence[_T]],
46: (4)            Sequence[Sequence[Sequence[_T]]],
47: (4)            Sequence[Sequence[Sequence[Sequence[_T]]]],
48: (0)        ]
49: (0)        _ArrayLike = Union[
50: (4)            _SupportsArray[dtype[_ScalarType]],
51: (4)            _NestedSequence[_SupportsArray[dtype[_ScalarType]]],
52: (0)        ]
53: (0)        _DualArrayLike = Union[
54: (4)            _SupportsArray[_DType],
55: (4)            _NestedSequence[_SupportsArray[_DType]],
56: (4)            _T,
57: (4)            _NestedSequence[_T],
58: (0)        ]
59: (0)        if sys.version_info >= (3, 12):
60: (4)            from collections.abc import Buffer
```

```

61: (4)           ArrayLike = Buffer | _DualArrayLike[
62: (8)             dtype[Any],
63: (8)               Union[bool, int, float, complex, str, bytes],
64: (4)             ]
65: (0) else:
66: (4)     ArrayLike = _DualArrayLike[
67: (8)       dtype[Any],
68: (8)         Union[bool, int, float, complex, str, bytes],
69: (4)       ]
70: (0) _ArrayLikeBool_co = _DualArrayLike[
71: (4)   dtype[bool_],
72: (4)     bool,
73: (0)   ]
74: (0) _ArrayLikeUInt_co = _DualArrayLike[
75: (4)   dtype[Union[bool_, unsignedinteger[Any]]],
76: (4)     bool,
77: (0)   ]
78: (0) _ArrayLikeInt_co = _DualArrayLike[
79: (4)   dtype[Union[bool_, integer[Any]]],
80: (4)     Union[bool, int],
81: (0)   ]
82: (0) _ArrayLikeFloat_co = _DualArrayLike[
83: (4)   dtype[Union[bool_, integer[Any], floating[Any]]],
84: (4)     Union[bool, int, float],
85: (0)   ]
86: (0) _ArrayLikeComplex_co = _DualArrayLike[
87: (4)   dtype[Union[
88: (8)     bool_,
89: (8)       integer[Any],
90: (8)       floating[Any],
91: (8)       complexfloating[Any, Any],
92: (4)     ]],
93: (4)     Union[bool, int, float, complex],
94: (0)   ]
95: (0) _ArrayLikeNumber_co = _DualArrayLike[
96: (4)   dtype[Union[bool_, number[Any]]],
97: (4)     Union[bool, int, float, complex],
98: (0)   ]
99: (0) _ArrayLikeTD64_co = _DualArrayLike[
100: (4)   dtype[Union[bool_, integer[Any], timedelta64]],
101: (4)     Union[bool, int],
102: (0)   ]
103: (0) _ArrayLikeDT64_co = Union[
104: (4)   _SupportsArray[dtype[datetime64]],
105: (4)     _NestedSequence[_SupportsArray[dtype[datetime64]]],
106: (0)   ]
107: (0) _ArrayLikeObject_co = Union[
108: (4)   _SupportsArray[dtype[object_]],
109: (4)     _NestedSequence[_SupportsArray[dtype[object_]]],
110: (0)   ]
111: (0) _ArrayLikeVoid_co = Union[
112: (4)   _SupportsArray[dtype[void]],
113: (4)     _NestedSequence[_SupportsArray[dtype[void]]],
114: (0)   ]
115: (0) _ArrayLikeStr_co = _DualArrayLike[
116: (4)   dtype[str_],
117: (4)     str,
118: (0)   ]
119: (0) _ArrayLikeBytes_co = _DualArrayLike[
120: (4)   dtype[bytes_],
121: (4)     bytes,
122: (0)   ]
123: (0) _ArrayLikeInt = _DualArrayLike[
124: (4)   dtype[integer[Any]],
125: (4)     int,
126: (0)   ]
127: (0) class _UnknownType:
128: (4)   ...
129: (0)   _ArrayLikeUnknown = _DualArrayLike[

```

```

130: (4)           dtype[_UnknownType],
131: (4)           _UnknownType,
132: (0)          ]
-----
```

## File 398 - \_char\_codes.py:

```

1: (0)           from typing import Literal
2: (0)           _BoolCodes = Literal["?", "=?", "<?", ">?", "bool", "bool_", "bool8"]
3: (0)           _UInt8Codes = Literal["uint8", "u1", "=u1", "<u1", ">u1"]
4: (0)           _UInt16Codes = Literal["uint16", "u2", "=u2", "<u2", ">u2"]
5: (0)           _UInt32Codes = Literal["uint32", "u4", "=u4", "<u4", ">u4"]
6: (0)           _UInt64Codes = Literal["uint64", "u8", "=u8", "<u8", ">u8"]
7: (0)           _Int8Codes = Literal["int8", "i1", "=i1", "<i1", ">i1"]
8: (0)           _Int16Codes = Literal["int16", "i2", "=i2", "<i2", ">i2"]
9: (0)           _Int32Codes = Literal["int32", "i4", "=i4", "<i4", ">i4"]
10: (0)          _Int64Codes = Literal["int64", "i8", "=i8", "<i8", ">i8"]
11: (0)          _Float16Codes = Literal["float16", "f2", "=f2", "<f2", ">f2"]
12: (0)          _Float32Codes = Literal["float32", "f4", "=f4", "<f4", ">f4"]
13: (0)          _Float64Codes = Literal["float64", "f8", "=f8", "<f8", ">f8"]
14: (0)          _Complex64Codes = Literal["complex64", "c8", "=c8", "<c8", ">c8"]
15: (0)          _Complex128Codes = Literal["complex128", "c16", "=c16", "<c16", ">c16"]
16: (0)          _ByteCodes = Literal["byte", "b", "=b", "<b", ">b"]
17: (0)          _ShortCodes = Literal["short", "h", "=h", "<h", ">h"]
18: (0)          _IntCCodes = Literal["intc", "i", "=i", "<i", ">i"]
19: (0)          _IntPCodes = Literal["intp", "int0", "p", "=p", "<p", ">p"]
20: (0)          _IntCodes = Literal["long", "int", "int_", "l", "=l", "<l", ">l"]
21: (0)          _LongLongCodes = Literal["longlong", "q", "=q", "<q", ">q"]
22: (0)          _UByteCodes = Literal["ubyte", "B", "=B", "<B", ">B"]
23: (0)          _UShortCodes = Literal["ushort", "H", "=H", "<H", ">H"]
24: (0)          _UIntCCodes = Literal["uintc", "I", "=I", "<I", ">I"]
25: (0)          _UIntPCodes = Literal["uintp", "uint0", "P", "=P", "<P", ">P"]
26: (0)          _UIntCodes = Literal["ulong", "uint", "L", "=L", "<L", ">L"]
27: (0)          _ULongLongCodes = Literal["ulonglong", "Q", "=Q", "<Q", ">Q"]
28: (0)          _HalfCodes = Literal["half", "e", "=e", "<e", ">e"]
29: (0)          _SingleCodes = Literal["single", "f", "=f", "<f", ">f"]
30: (0)          _DoubleCodes = Literal["double", "float", "float_", "d", "=d", "<d", ">d"]
31: (0)          _LongDoubleCodes = Literal["longdouble", "longfloat", "g", "=g", "<g", ">g"]
32: (0)          _CSingleCodes = Literal["csingle", "singlecomplex", "F", "=F", "<F", ">F"]
33: (0)          _CDoubleCodes = Literal["cdouble", "complex", "complex_", "cfloat", "D", "=D",
    "<D", ">D"]
34: (0)          _CLongDoubleCodes = Literal["clongdouble", "clongfloat", "longcomplex", "G",
    "=G", "<G", ">G"]
35: (0)          _StrCodes = Literal["str", "str_", "str0", "unicode", "unicode_", "U", "=U", "<U",
    ">U"]
36: (0)          _BytesCodes = Literal["bytes", "bytes_", "bytes0", "S", "=S", "<S", ">S"]
37: (0)          _VoidCodes = Literal["void", "void0", "V", "=V", "<V", ">V"]
38: (0)          _ObjectCodes = Literal["object", "object_", "O", "=O", "<O", ">O"]
39: (0)          _DT64Codes = Literal[
    "datetime64", "=datetime64", "<datetime64", ">datetime64",
    "datetime64[Y]", "=datetime64[Y]", "<datetime64[Y]", ">datetime64[Y]",
    "datetime64[M]", "=datetime64[M]", "<datetime64[M]", ">datetime64[M]",
    "datetime64[W]", "=datetime64[W]", "<datetime64[W]", ">datetime64[W]",
    "datetime64[D]", "=datetime64[D]", "<datetime64[D]", ">datetime64[D]",
    "datetime64[h]", "=datetime64[h]", "<datetime64[h]", ">datetime64[h]",
    "datetime64[m]", "=datetime64[m]", "<datetime64[m]", ">datetime64[m]",
    "datetime64[s]", "=datetime64[s]", "<datetime64[s]", ">datetime64[s]",
    "datetime64[ms]", "=datetime64[ms]", "<datetime64[ms]", ">datetime64[ms]",
    "datetime64[us]", "=datetime64[us]", "<datetime64[us]", ">datetime64[us]",
    "datetime64[ns]", "=datetime64[ns]", "<datetime64[ns]", ">datetime64[ns]",
    "datetime64[ps]", "=datetime64[ps]", "<datetime64[ps]", ">datetime64[ps]",
    "datetime64[fs]", "=datetime64[fs]", "<datetime64[fs]", ">datetime64[fs]",
    "datetime64[as]", "=datetime64[as]", "<datetime64[as]", ">datetime64[as]",
    "M", "=M", "<M", ">M",
    "M8", "=M8", "<M8", ">M8",
    "M8[Y]", "=M8[Y]", "<M8[Y]", ">M8[Y]",
    "M8[M]", "=M8[M]", "<M8[M]", ">M8[M]",
    "M8[W]", "=M8[W]", "<M8[W]", ">M8[W]",
```

```

59: (4)           "M8[D]",  "=M8[D]",  "<M8[D]",  ">M8[D]",
60: (4)           "M8[h]",   "=M8[h]",   "<M8[h]",   ">M8[h]",
61: (4)           "M8[m]",   "=M8[m]",   "<M8[m]",   ">M8[m]",
62: (4)           "M8[s]",   "=M8[s]",   "<M8[s]",   ">M8[s]",
63: (4)           "M8[ms]",  "=M8[ms]",  "<M8[ms]",  ">M8[ms]",
64: (4)           "M8[us]",  "=M8[us]",  "<M8[us]",  ">M8[us]",
65: (4)           "M8[ns]",  "=M8[ns]",  "<M8[ns]",  ">M8[ns]",
66: (4)           "M8[ps]",  "=M8[ps]",  "<M8[ps]",  ">M8[ps]",
67: (4)           "M8[fs]",  "=M8[fs]",  "<M8[fs]",  ">M8[fs]",
68: (4)           "M8[as]",  "=M8[as]",  "<M8[as]",  ">M8[as]",
69: (0)
70: (0)
] _TD64Codes = Literal[
71: (4)           "timedelta64",  "=timedelta64",  "<timedelta64",  ">timedelta64",
72: (4)           "timedelta64[Y]",  "=timedelta64[Y]",  "<timedelta64[Y]",  ">timedelta64[Y]",
73: (4)           "timedelta64[M]",  "=timedelta64[M]",  "<timedelta64[M]",  ">timedelta64[M]",
74: (4)           "timedelta64[W]",  "=timedelta64[W]",  "<timedelta64[W]",  ">timedelta64[W]",
75: (4)           "timedelta64[D]",  "=timedelta64[D]",  "<timedelta64[D]",  ">timedelta64[D]",
76: (4)           "timedelta64[h]",  "=timedelta64[h]",  "<timedelta64[h]",  ">timedelta64[h]",
77: (4)           "timedelta64[m]",  "=timedelta64[m]",  "<timedelta64[m]",  ">timedelta64[m]",
78: (4)           "timedelta64[s]",  "=timedelta64[s]",  "<timedelta64[s]",  ">timedelta64[s]",
79: (4)           "timedelta64[ms]",  "=timedelta64[ms]",  "<timedelta64[ms]",  ">timedelta64[ms]",
80: (4)           "timedelta64[us]",  "=timedelta64[us]",  "<timedelta64[us]",  ">timedelta64[us]",
81: (4)           "timedelta64[ns]",  "=timedelta64[ns]",  "<timedelta64[ns]",  ">timedelta64[ns]",
82: (4)           "timedelta64[ps]",  "=timedelta64[ps]",  "<timedelta64[ps]",  ">timedelta64[ps]",
83: (4)           "timedelta64[fs]",  "=timedelta64[fs]",  "<timedelta64[fs]",  ">timedelta64[fs]",
84: (4)           "timedelta64[as]",  "=timedelta64[as]",  "<timedelta64[as]",  ">timedelta64[as]",
85: (4)           "m",   "=m",   "<m",   ">m",
86: (4)           "m8",  "=m8",  "<m8",  ">m8",
87: (4)           "m8[Y]",  "=m8[Y]",  "<m8[Y]",  ">m8[Y]",
88: (4)           "m8[M]",  "=m8[M]",  "<m8[M]",  ">m8[M]",
89: (4)           "m8[W]",  "=m8[W]",  "<m8[W]",  ">m8[W]",
90: (4)           "m8[D]",  "=m8[D]",  "<m8[D]",  ">m8[D]",
91: (4)           "m8[h]",  "=m8[h]",  "<m8[h]",  ">m8[h]",
92: (4)           "m8[m]",  "=m8[m]",  "<m8[m]",  ">m8[m]",
93: (4)           "m8[s]",  "=m8[s]",  "<m8[s]",  ">m8[s]",
94: (4)           "m8[ms]",  "=m8[ms]",  "<m8[ms]",  ">m8[ms]",
95: (4)           "m8[us]",  "=m8[us]",  "<m8[us]",  ">m8[us]",
96: (4)           "m8[ns]",  "=m8[ns]",  "<m8[ns]",  ">m8[ns]",
97: (4)           "m8[ps]",  "=m8[ps]",  "<m8[ps]",  ">m8[ps]",
98: (4)           "m8[fs]",  "=m8[fs]",  "<m8[fs]",  ">m8[fs]",
99: (4)           "m8[as]",  "=m8[as]",  "<m8[as]",  ">m8[as]",
100: (0)          ]
-----
```

## File 399 - \_dtype\_like.py:

```

1: (0)           from collections.abc import Sequence
2: (0)           from typing import (
3: (4)             Any,
4: (4)             Sequence,
5: (4)             Union,
6: (4)             TypeVar,
7: (4)             Protocol,
8: (4)             TypedDict,
9: (4)             runtime_checkable,
10: (0)            )
11: (0)           import numpy as np
12: (0)           from ._shape import _ShapeLike
13: (0)           from ._char_codes import (
14: (4)             _BoolCodes,
15: (4)             _UInt8Codes,
16: (4)             _UInt16Codes,
```

```

17: (4)           _UInt32Codes,
18: (4)           _UInt64Codes,
19: (4)           _Int8Codes,
20: (4)           _Int16Codes,
21: (4)           _Int32Codes,
22: (4)           _Int64Codes,
23: (4)           _Float16Codes,
24: (4)           _Float32Codes,
25: (4)           _Float64Codes,
26: (4)           _Complex64Codes,
27: (4)           _Complex128Codes,
28: (4)           _ByteCodes,
29: (4)           _ShortCodes,
30: (4)           _IntCCodes,
31: (4)           _IntPCodes,
32: (4)           _IntCodes,
33: (4)           _LongLongCodes,
34: (4)           _UByteCodes,
35: (4)           _UShortCodes,
36: (4)           _UIntCCodes,
37: (4)           _UIntPCodes,
38: (4)           _UIntCodes,
39: (4)           _ULongLongCodes,
40: (4)           _HalfCodes,
41: (4)           _SingleCodes,
42: (4)           _DoubleCodes,
43: (4)           _LongDoubleCodes,
44: (4)           _CSingleCodes,
45: (4)           _CDoubleCodes,
46: (4)           _CLongDoubleCodes,
47: (4)           _DT64Codes,
48: (4)           _TD64Codes,
49: (4)           _StrCodes,
50: (4)           _BytesCodes,
51: (4)           _VoidCodes,
52: (4)           _ObjectCodes,
53: (0)
54: (0)           _SCT = TypeVar("_SCT", bound=np.generic)
55: (0)           _DType_co = TypeVar("_DType_co", covariant=True, bound=np.dtype[Any])
56: (0)           _DTypeLikeNested = Any # TODO: wait for support for recursive types
57: (0)           class _DTypeDictBase(TypedDict):
58: (4)               names: Sequence[str]
59: (4)               formats: Sequence[_DTypeLikeNested]
60: (0)           class _DTypeDict(_DTypeDictBase, total=False):
61: (4)               offsets: Sequence[int]
62: (4)               titles: Sequence[Any]
63: (4)               itemsize: int
64: (4)               aligned: bool
65: (0)               @runtime_checkable
66: (0)               class _SupportsDType(Protocol[_DType_co]):
67: (4)                   @property
68: (4)                   def dtype(self) -> _DType_co: ...
69: (0)               _DTypeLike = Union[
70: (4)                   np.dtype[_SCT],
71: (4)                   type[_SCT],
72: (4)                   _SupportsDType[np.dtype[_SCT]],
73: (0)
74: (0)                   _VoidDTypeLike = Union[
75: (4)                       tuple[_DTypeLikeNested, int],
76: (4)                       tuple[_DTypeLikeNested, _ShapeLike],
77: (4)                       list[Any],
78: (4)                       _DTypeDict,
79: (4)                       tuple[_DTypeLikeNested, _DTypeLikeNested],
80: (0)
81: (0)                   DTypeLike = Union[
82: (4)                       np.dtype[Any],
83: (4)                       None,
84: (4)                       type[Any], # NOTE: We're stuck with `type[Any]` due to object dtypes
85: (4)                       _SupportsDType[np.dtype[Any]],
```

```

86: (4)                         str,
87: (4)                         _VoidDTypeLike,
88: (0)                         ]
89: (0)                         _DTypeLikeBool = Union[
90: (4)                           type[bool],
91: (4)                           type[np.bool_],
92: (4)                           np.dtype[np.bool_],
93: (4)                           _SupportsDType[np.dtype[np.bool_]],
94: (4)                           _BoolCodes,
95: (0)                         ]
96: (0)                         _DTypeLikeUInt = Union[
97: (4)                           type[np.unsignedinteger],
98: (4)                           np.dtype[np.unsignedinteger],
99: (4)                           _SupportsDType[np.dtype[np.unsignedinteger]],
100: (4)                          _UInt8Codes,
101: (4)                          _UInt16Codes,
102: (4)                          _UInt32Codes,
103: (4)                          _UInt64Codes,
104: (4)                          _UByteCodes,
105: (4)                          _UShortCodes,
106: (4)                          _UIntCCodes,
107: (4)                          _UIntPCodes,
108: (4)                          _UIntCodes,
109: (4)                          _ULongLongCodes,
110: (0)                         ]
111: (0)                         _DTypeLikeInt = Union[
112: (4)                           type[int],
113: (4)                           type[np.signedinteger],
114: (4)                           np.dtype[np.signedinteger],
115: (4)                           _SupportsDType[np.dtype[np.signedinteger]],
116: (4)                           _Int8Codes,
117: (4)                           _Int16Codes,
118: (4)                           _Int32Codes,
119: (4)                           _Int64Codes,
120: (4)                           _ByteCodes,
121: (4)                           _ShortCodes,
122: (4)                           _IntCCodes,
123: (4)                           _IntPCodes,
124: (4)                           _IntCodes,
125: (4)                           _LongLongCodes,
126: (0)                         ]
127: (0)                         _DTypeLikeFloat = Union[
128: (4)                           type[float],
129: (4)                           type[np.floating],
130: (4)                           np.dtype[np.floating],
131: (4)                           _SupportsDType[np.dtype[np.floating]],
132: (4)                           _Float16Codes,
133: (4)                           _Float32Codes,
134: (4)                           _Float64Codes,
135: (4)                           _HalfCodes,
136: (4)                           _SingleCodes,
137: (4)                           _DoubleCodes,
138: (4)                           _LongDoubleCodes,
139: (0)                         ]
140: (0)                         _DTypeLikeComplex = Union[
141: (4)                           type[complex],
142: (4)                           type[np.complexfloating],
143: (4)                           np.dtype[np.complexfloating],
144: (4)                           _SupportsDType[np.dtype[np.complexfloating]],
145: (4)                           _Complex64Codes,
146: (4)                           _Complex128Codes,
147: (4)                           _CSingleCodes,
148: (4)                           _CDoubleCodes,
149: (4)                           _CLongDoubleCodes,
150: (0)                         ]
151: (0)                         _DTypeLikeDT64 = Union[
152: (4)                           type[np.timedelta64],
153: (4)                           np.dtype[np.timedelta64],
154: (4)                           _SupportsDType[np.dtype[np.timedelta64]],
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

155: (4)           _TD64Codes,
156: (0)          ]
157: (0)          _DTypeLikeTD64 = Union[
158: (4)            type[np.datetime64],
159: (4)            np.dtype[np.datetime64],
160: (4)            _SupportsDType[np.dtype[np.datetime64]],
161: (4)            _DT64Codes,
162: (0)          ]
163: (0)          _DTypeLikeStr = Union[
164: (4)            type[str],
165: (4)            type[np.str_],
166: (4)            np.dtype[np.str_],
167: (4)            _SupportsDType[np.dtype[np.str_]],
168: (4)            _StrCodes,
169: (0)          ]
170: (0)          _DTypeLikeBytes = Union[
171: (4)            type[bytes],
172: (4)            type[np.bytes_],
173: (4)            np.dtype[np.bytes_],
174: (4)            _SupportsDType[np.dtype[np.bytes_]],
175: (4)            _BytesCodes,
176: (0)          ]
177: (0)          _DTypeLikeVoid = Union[
178: (4)            type[np.void],
179: (4)            np.dtype[np.void],
180: (4)            _SupportsDType[np.dtype[np.void]],
181: (4)            _VoidCodes,
182: (4)            _VoidDTypeLike,
183: (0)          ]
184: (0)          _DTypeLikeObject = Union[
185: (4)            type,
186: (4)            np.dtype[np.object_],
187: (4)            _SupportsDType[np.dtype[np.object_]],
188: (4)            _ObjectCodes,
189: (0)          ]
190: (0)          _DTypeLikeComplex_co = Union[
191: (4)            _DTypeLikeBool,
192: (4)            _DTypeLikeUInt,
193: (4)            _DTypeLikeInt,
194: (4)            _DTypeLikeFloat,
195: (4)            _DTypeLikeComplex,
196: (0)        ]
-----
```

## File 400 - \_nbit.py:

```

1: (0)      """A module with the precisions of platform-specific `~numpy.number`s."""
2: (0)      from typing import Any
3: (0)      _NBitByte = Any
4: (0)      _NBitShort = Any
5: (0)      _NBitIntC = Any
6: (0)      _NBitIntP = Any
7: (0)      _NBitInt = Any
8: (0)      _NBitLongLong = Any
9: (0)      _NBitHalf = Any
10: (0)     _NBitSingle = Any
11: (0)     _NBitDouble = Any
12: (0)     _NBitLongDouble = Any
-----
```

## File 401 - \_extended\_precision.py:

```

1: (0)      """A module with platform-specific extended precision
2: (0)      `numpy.number` subclasses.
3: (0)      The subclasses are defined here (instead of ``__init__.pyi``) such
4: (0)      that they can be imported conditionally via the numpy's mypy plugin.
5: (0)      """
-----
```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

6: (0) import numpy as np
7: (0) from . import (
8: (4)     _80Bit,
9: (4)     _96Bit,
10: (4)    _128Bit,
11: (4)    _256Bit,
12: (0)
13: (0) uint128 = np.unsignedinteger[_128Bit]
14: (0) uint256 = np.unsignedinteger[_256Bit]
15: (0) int128 = np.signedinteger[_128Bit]
16: (0) int256 = np.signedinteger[_256Bit]
17: (0) float80 = np.floating[_80Bit]
18: (0) float96 = np.floating[_96Bit]
19: (0) float128 = np.floating[_128Bit]
20: (0) float256 = np.floating[_256Bit]
21: (0) complex160 = np.complexfloating[_80Bit, _80Bit]
22: (0) complex192 = np.complexfloating[_96Bit, _96Bit]
23: (0) complex256 = np.complexfloating[_128Bit, _128Bit]
24: (0) complex512 = np.complexfloating[_256Bit, _256Bit]
-----
```

## File 402 - \_scalars.py:

```

1: (0) from typing import Union, Any
2: (0) import numpy as np
3: (0) _CharLike_co = Union[str, bytes]
4: (0) _BoolLike_co = Union[bool, np.bool_]
5: (0) _UIntLike_co = Union[_BoolLike_co, np.unsignedinteger[Any]]
6: (0) _IntLike_co = Union[_BoolLike_co, int, np.integer[Any]]
7: (0) _FloatLike_co = Union[_IntLike_co, float, np.floating[Any]]
8: (0) _ComplexLike_co = Union[_FloatLike_co, complex, np.complexfloating[Any, Any]]
9: (0) _TD64Like_co = Union[_IntLike_co, np.timedelta64]
10: (0) _NumberLike_co = Union[int, float, complex, np.number[Any], np.bool_]
11: (0) _ScalarLike_co = Union[
12: (4)     int,
13: (4)     float,
14: (4)     complex,
15: (4)     str,
16: (4)     bytes,
17: (4)     np.generic,
18: (0) ]
19: (0)     _VoidLike_co = Union[tuple[Any, ...], np void]
-----
```

## File 403 - \_nested\_sequence.py:

```

1: (0) """A module containing the `NestedSequence` protocol."""
2: (0) from __future__ import annotations
3: (0) from collections.abc import Iterator
4: (0) from typing import (
5: (4)     Any,
6: (4)     TypeVar,
7: (4)     Protocol,
8: (4)     runtime_checkable,
9: (0)
10: (0)     __all__ = ["NestedSequence"]
11: (0)     _T_co = TypeVar("_T_co", covariant=True)
12: (0)     @runtime_checkable
13: (0)     class NestedSequence(Protocol[_T_co]):
14: (4)         """A protocol for representing nested sequences.
15: (4)         Warning
16: (4)         -----
17: (4)         `NestedSequence` currently does not work in combination with typevars,
18: (4)         *e.g.* ``def func(a: NestedSequence[T]) -> T: ...``.
19: (4)         See Also
20: (4)         -----
21: (4)         collections.abc.Sequence
-----
```

```

22: (8)          ABCs for read-only and mutable :term:`sequences` .
23: (4)          Examples
24: (4)
25: (4)          .. code-block:: python
26: (8)          >>> from __future__ import annotations
27: (8)          >>> from typing import TYPE_CHECKING
28: (8)          >>> import numpy as np
29: (8)          >>> from numpy._typing import _NestedSequence
30: (8)          >>> def get_dtype(seq: _NestedSequence[float]) ->
np.dtype[np.float64]:
31: (8)              ...      return np.asarray(seq).dtype
32: (8)              >>> a = get_dtype([1.0])
33: (8)              >>> b = get_dtype([[1.0]])
34: (8)              >>> c = get_dtype([[[1.0]]])
35: (8)              >>> d = get_dtype([[[[1.0]]]])
36: (8)              >>> if TYPE_CHECKING:
37: (8)                  ...      reveal_locals()
38: (8)                  ...      # note: Revealed local types are:
39: (8)                  ...      # note:    a:
numpy.dtype[numpy.floating[numpy._typing._64Bit]]
40: (8)          ...      # note:    b:
numpy.dtype[numpy.floating[numpy._typing._64Bit]]
41: (8)          ...      # note:    c:
numpy.dtype[numpy.floating[numpy._typing._64Bit]]
42: (8)          ...      # note:    d:
numpy.dtype[numpy.floating[numpy._typing._64Bit]]
43: (4)          """
44: (4)          def __len__(self, /) -> int:
45: (8)              """Implement ``len(self)``."""
46: (8)              raise NotImplementedError
47: (4)          def __getitem__(self, index: int, /) -> _T_co | _NestedSequence[_T_co]:
48: (8)              """Implement ``self[x]``."""
49: (8)              raise NotImplementedError
50: (4)          def __contains__(self, x: object, /) -> bool:
51: (8)              """Implement ``x in self``."""
52: (8)              raise NotImplementedError
53: (4)          def __iter__(self, /) -> Iterator[_T_co | _NestedSequence[_T_co]]:
54: (8)              """Implement ``iter(self)``."""
55: (8)              raise NotImplementedError
56: (4)          def __reversed__(self, /) -> Iterator[_T_co | _NestedSequence[_T_co]]:
57: (8)              """Implement ``reversed(self)``."""
58: (8)              raise NotImplementedError
59: (4)          def count(self, value: Any, /) -> int:
60: (8)              """Return the number of occurrences of `value`."""
61: (8)              raise NotImplementedError
62: (4)          def index(self, value: Any, /) -> int:
63: (8)              """Return the first index of `value`."""
64: (8)              raise NotImplementedError

```

---

**File 404 - \_shape.py:**

```

1: (0)          from collections.abc import Sequence
2: (0)          from typing import Union, SupportsIndex
3: (0)          _Shape = tuple[int, ...]
4: (0)          _ShapeLike = Union[SupportsIndex, Sequence[SupportsIndex]]

```

---

**File 405 - \_\_init\_\_.py:**

```

1: (0)          """Private counterpart of ``numpy.typing``."""
2: (0)          from __future__ import annotations
3: (0)          from .. import ufunc
4: (0)          from .._utils import set_module
5: (0)          from typing import TYPE_CHECKING, final
6: (0)          @final # Disallow the creation of arbitrary `NBitBase` subclasses
7: (0)          @set_module("numpy.typing")

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

8: (0)
9: (4)
10: (4)        A type representing `numpy.number` precision during static type checking.
11: (4)        Used exclusively for the purpose static type checking, `NBitBase`
12: (4)        represents the base of a hierarchical set of subclasses.
13: (4)        Each subsequent subclass is herein used for representing a lower level
14: (4)        of precision, *e.g.* ``64Bit > 32Bit > 16Bit``.
15: (4)        .. versionadded:: 1.20
16: (4)        Examples
17: (4)        -----
18: (4)        Below is a typical usage example: `NBitBase` is herein used for annotating
19: (4)        a function that takes a float and integer of arbitrary precision
20: (4)        as arguments and returns a new float of whichever precision is largest
21: (4)        (*e.g.* ``np.float16 + np.int64 -> np.float64``).
22: (4)        .. code-block:: python
23: (8)            >>> from __future__ import annotations
24: (8)            >>> from typing import TypeVar, TYPE_CHECKING
25: (8)            >>> import numpy as np
26: (8)            >>> import numpy.typing as npt
27: (8)            >>> T1 = TypeVar("T1", bound=npt.NBitBase)
28: (8)            >>> T2 = TypeVar("T2", bound=npt.NBitBase)
29: (8)            >>> def add(a: np.floating[T1], b: np.integer[T2]) -> np.floating[T1 | T2]:
30: (8)                ...      return a + b
31: (8)                >>> a = np.float16()
32: (8)                >>> b = np.int64()
33: (8)                >>> out = add(a, b)
34: (8)                >>> if TYPE_CHECKING:
35: (8)                    ...      reveal_locals()
36: (8)                    ...      # note: Revealed local types are:
37: (8)                    ...      # note:      a: numpy.floating[numpy.typing._16Bit*]
38: (8)                    ...      # note:      b: numpy.signedinteger[numpy.typing._64Bit*]
39: (8)                    ...      # note:      out: numpy.floating[numpy.typing._64Bit*]
40: (4)        """
41: (4)        def __init_subclass__(cls) -> None:
42: (8)            allowed_names = {
43: (12)                "NBitBase", "_256Bit", "_128Bit", "_96Bit", "_80Bit",
44: (12)                "_64Bit", "_32Bit", "_16Bit", "_8Bit",
45: (8)            }
46: (8)            if cls.__name__ not in allowed_names:
47: (12)                raise TypeError('cannot inherit from final class "NBitBase"')
48: (8)            super().__init_subclass__()
49: (0)            class _256Bit(NBitBase): # type: ignore[misc]
50: (4)                pass
51: (0)            class _128Bit(_256Bit): # type: ignore[misc]
52: (4)                pass
53: (0)            class _96Bit(_128Bit): # type: ignore[misc]
54: (4)                pass
55: (0)            class _80Bit(_96Bit): # type: ignore[misc]
56: (4)                pass
57: (0)            class _64Bit(_80Bit): # type: ignore[misc]
58: (4)                pass
59: (0)            class _32Bit(_64Bit): # type: ignore[misc]
60: (4)                pass
61: (0)            class _16Bit(_32Bit): # type: ignore[misc]
62: (4)                pass
63: (0)            class _8Bit(_16Bit): # type: ignore[misc]
64: (4)                pass
65: (0)            from .nested_sequence import (
66: (4)                _NestedSequence as _NestedSequence,
67: (0)            )
68: (0)            from .nbit import (
69: (4)                _NBitByte as _NBitByte,
70: (4)                _NBitShort as _NBitShort,
71: (4)                _NBitIntC as _NBitIntC,
72: (4)                _NBitIntP as _NBitIntP,
73: (4)                _NBitInt as _NBitInt,
74: (4)                _NBitLongLong as _NBitLongLong,
75: (4)                _NBitHalf as _NBitHalf,
```

```

76: (4)           _NBitSingle as _NBitSingle,
77: (4)           _NBitDouble as _NBitDouble,
78: (4)           _NBitLongDouble as _NBitLongDouble,
79: (0)
80: (0)
81: (4)       )
82: (4)       from ._char_codes import (
83: (4)           _BoolCodes as _BoolCodes,
84: (4)           _UInt8Codes as _UInt8Codes,
85: (4)           _UInt16Codes as _UInt16Codes,
86: (4)           _UInt32Codes as _UInt32Codes,
87: (4)           _UInt64Codes as _UInt64Codes,
88: (4)           _Int8Codes as _Int8Codes,
89: (4)           _Int16Codes as _Int16Codes,
90: (4)           _Int32Codes as _Int32Codes,
91: (4)           _Int64Codes as _Int64Codes,
92: (4)           _Float16Codes as _Float16Codes,
93: (4)           _Float32Codes as _Float32Codes,
94: (4)           _Float64Codes as _Float64Codes,
95: (4)           _Complex64Codes as _Complex64Codes,
96: (4)           _Complex128Codes as _Complex128Codes,
97: (4)           _ByteCodes as _ByteCodes,
98: (4)           _ShortCodes as _ShortCodes,
99: (4)           _IntCCodes as _IntCCodes,
100: (4)          _IntPCodes as _IntPCodes,
101: (4)          _IntCodes as _IntCodes,
102: (4)          _LongLongCodes as _LongLongCodes,
103: (4)          _UByteCodes as _UByteCodes,
104: (4)          _UShortCodes as _UShortCodes,
105: (4)          _UIntCCodes as _UIntCCodes,
106: (4)          _UIntPCodes as _UIntPCodes,
107: (4)          _UIntCodes as _UIntCodes,
108: (4)          _ULongLongCodes as _ULongLongCodes,
109: (4)          _HalfCodes as _HalfCodes,
110: (4)          _SingleCodes as _SingleCodes,
111: (4)          _DoubleCodes as _DoubleCodes,
112: (4)          _LongDoubleCodes as _LongDoubleCodes,
113: (4)          _CSingleCodes as _CSingleCodes,
114: (4)          _CDoubleCodes as _CDoubleCodes,
115: (4)          _CLongDoubleCodes as _CLongDoubleCodes,
116: (4)          _DT64Codes as _DT64Codes,
117: (4)          _TD64Codes as _TD64Codes,
118: (4)          _StrCodes as _StrCodes,
119: (4)          _BytesCodes as _BytesCodes,
120: (0)          _VoidCodes as _VoidCodes,
121: (0)          _ObjectCodes as _ObjectCodes,
122: (4)
123: (4)
124: (4)
125: (4)
126: (4)
127: (4)
128: (4)
129: (4)
130: (4)
131: (4)
132: (0)
133: (0)
134: (4)
135: (4)
136: (0)
137: (0)
138: (4)
139: (4)
140: (4)
141: (4)
142: (4)
143: (4)
144: (4)
)
from ._scalars import (
    _CharLike_co as _CharLike_co,
    _BoolLike_co as _BoolLike_co,
    _UIntLike_co as _UIntLike_co,
    _IntLike_co as _IntLike_co,
    _FloatLike_co as _FloatLike_co,
    _ComplexLike_co as _ComplexLike_co,
    _TD64Like_co as _TD64Like_co,
    _NumberLike_co as _NumberLike_co,
    _ScalarLike_co as _ScalarLike_co,
    _VoidLike_co as _VoidLike_co,
)
from ._shape import (
    _Shape as _Shape,
    _ShapeLike as _ShapeLike,
)
from ._dtype_like import (
    DTypeLike as DTypeLike,
    _DTypeLike as _DTypeLike,
    _SupportsDType as _SupportsDType,
    _VoidDTypeLike as _VoidDTypeLike,
    _DTypeLikeBool as _DTypeLikeBool,
    _DTypeLikeUInt as _DTypeLikeUInt,
    _DTypeLikeInt as _DTypeLikeInt,
)

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

145: (4)           _DTypeLikeFloat as _DTypeLikeFloat,
146: (4)           _DTypeLikeComplex as _DTypeLikeComplex,
147: (4)           _DTypeLikeTD64 as _DTypeLikeTD64,
148: (4)           _DTypeLikeDT64 as _DTypeLikeDT64,
149: (4)           _DTypeLikeObject as _DTypeLikeObject,
150: (4)           _DTypeLikeVoid as _DTypeLikeVoid,
151: (4)           _DTypeLikeStr as _DTypeLikeStr,
152: (4)           _DTypeLikeBytes as _DTypeLikeBytes,
153: (4)           _DTypeLikeComplex_co as _DTypeLikeComplex_co,
154: (0)
155: (0)         )
156: (4)         from ._array_like import (
157: (4)             NDArray as NDArray,
158: (4)             ArrayLike as ArrayLike,
159: (4)             _ArrayLike as _ArrayLike,
160: (4)             _FiniteNestedSequence as _FiniteNestedSequence,
161: (4)             _SupportsArray as _SupportsArray,
162: (4)             _SupportsArrayFunc as _SupportsArrayFunc,
163: (4)             _ArrayLikeInt as _ArrayLikeInt,
164: (4)             _ArrayLikeBool_co as _ArrayLikeBool_co,
165: (4)             _ArrayLikeUInt_co as _ArrayLikeUInt_co,
166: (4)             _ArrayLikeInt_co as _ArrayLikeInt_co,
167: (4)             _ArrayLikeFloat_co as _ArrayLikeFloat_co,
168: (4)             _ArrayLikeComplex_co as _ArrayLikeComplex_co,
169: (4)             _ArrayLikeNumber_co as _ArrayLikeNumber_co,
170: (4)             _ArrayLikeTD64_co as _ArrayLikeTD64_co,
171: (4)             _ArrayLikeDT64_co as _ArrayLikeDT64_co,
172: (4)             _ArrayLikeObject_co as _ArrayLikeObject_co,
173: (4)             _ArrayLikeVoid_co as _ArrayLikeVoid_co,
174: (4)             _ArrayLikeStr_co as _ArrayLikeStr_co,
175: (4)             _ArrayLikeBytes_co as _ArrayLikeBytes_co,
176: (4)             _ArrayLikeUnknown as _ArrayLikeUnknown,
177: (0)             _UnknownType as _UnknownType,
178: (0)
179: (4)         if TYPE_CHECKING:
180: (8)             from ._ufunc import (
181: (8)                 _UFunc_Nin1_Nout1 as _UFunc_Nin1_Nout1,
182: (8)                 _UFunc_Nin2_Nout1 as _UFunc_Nin2_Nout1,
183: (8)                 _UFunc_Nin1_Nout2 as _UFunc_Nin1_Nout2,
184: (8)                 _UFunc_Nin2_Nout2 as _UFunc_Nin2_Nout2,
185: (4)                 _GUFunc_Nin2_Nout1 as _GUFunc_Nin2_Nout1,
186: (0)
187: (4)             )
188: (4)         else:
189: (4)             _UFunc_Nin1_Nout1 = ufunc
190: (4)             _UFunc_Nin2_Nout1 = ufunc
191: (4)             _UFunc_Nin1_Nout2 = ufunc
192: (4)             _UFunc_Nin2_Nout2 = ufunc
193: (4)             _GUFunc_Nin2_Nout1 = ufunc

```

---

File 406 - \_inspect.py:

```

1: (0)         """Subset of inspect module from upstream python
2: (0)         We use this instead of upstream because upstream inspect is slow to import,
and
3: (0)         significantly contributes to numpy import times. Importing this copy has
almost
4: (0)         no overhead.
5: (0)         """
6: (0)         import types
7: (0)         __all__ = ['getargspec', 'formatargspec']
8: (0)         def ismethod(object):
9: (4)             """Return true if the object is an instance method.
10: (4)             Instance method objects provide these attributes:
11: (8)                 __doc__          documentation string
12: (8)                 __name__         name with which this method was defined
13: (8)                 im_class        class object in which this method belongs
14: (8)                 im_func         function object containing implementation of method
15: (8)                 im_self         instance to which this method is bound, or None

```

```

16: (4)             """
17: (4)         return isinstance(object, types.MethodType)
18: (0) def isfunction(object):
19: (4)     """Return true if the object is a user-defined function.
20: (4)     Function objects provide these attributes:
21: (8)         __doc__           documentation string
22: (8)         __name__          name with which this function was defined
23: (8)         func_code        code object containing compiled function bytecode
24: (8)         func_defaults   tuple of any default values for arguments
25: (8)         func_doc         (same as __doc__)
26: (8)         func_globals    global namespace in which this function was defined
27: (8)         func_name        (same as __name__)
28: (4)     """
29: (4)         return isinstance(object, types.FunctionType)
30: (0) def iscode(object):
31: (4)     """Return true if the object is a code object.
32: (4)     Code objects provide these attributes:
33: (8)         co_argcount    number of arguments (not including * or ** args)
34: (8)         co_code        string of raw compiled bytecode
35: (8)         co_consts       tuple of constants used in the bytecode
36: (8)         co_filename    name of file in which this code object was created
37: (8)         co_firstlineno number of first line in Python source code
38: (8)         co_flags        bitmap: 1=optimized | 2=newlocals | 4=*arg | 8=**arg
39: (8)         co_lnotab      encoded mapping of line numbers to bytecode indices
40: (8)         co_name        name with which this code object was defined
41: (8)         co_names       tuple of names of local variables
42: (8)         co_nlocals     number of local variables
43: (8)         co_stacksize   virtual machine stack space required
44: (8)         co_varnames    tuple of names of arguments and local variables
45: (4)     """
46: (4)         return isinstance(object, types.CodeType)
47: (0) CO_OPTIMIZED, CO_NEWLOCALS, CO_VARARGS, CO_VARKEYWORDS = 1, 2, 4, 8
48: (0) def getargs(co):
49: (4)     """Get information about the arguments accepted by a code object.
50: (4)     Three things are returned: (args, varargs, varkw), where 'args' is
51: (4)     a list of argument names (possibly containing nested lists), and
52: (4)     'varargs' and 'varkw' are the names of the * and ** arguments or None.
53: (4)     """
54: (4)     if not iscode(co):
55: (8)         raise TypeError('arg is not a code object')
56: (4)     nargs = co.co_argcount
57: (4)     names = co.co_varnames
58: (4)     args = list(names[:nargs])
59: (4)     for i in range(nargs):
60: (8)         if args[i][:1] in ['', '.']:
61: (12)             raise TypeError("tuple function arguments are not supported")
62: (4)     varargs = None
63: (4)     if co.co_flags & CO_VARARGS:
64: (8)         varargs = co.co_varnames[nargs]
65: (8)         nargs = nargs + 1
66: (4)     varkw = None
67: (4)     if co.co_flags & CO_VARKEYWORDS:
68: (8)         varkw = co.co_varnames[nargs]
69: (4)     return args, varargs, varkw
70: (0) def getargspec(func):
71: (4)     """Get the names and default values of a function's arguments.
72: (4)     A tuple of four things is returned: (args, varargs, varkw, defaults).
73: (4)     'args' is a list of the argument names (it may contain nested lists).
74: (4)     'varargs' and 'varkw' are the names of the * and ** arguments or None.
75: (4)     'defaults' is an n-tuple of the default values of the last n arguments.
76: (4)     """
77: (4)     if ismethod(func):
78: (8)         func = func.__func__
79: (4)     if not isfunction(func):
80: (8)         raise TypeError('arg is not a Python function')
81: (4)     args, varargs, varkw = getargs(func.__code__)
82: (4)     return args, varargs, varkw, func.__defaults__
83: (0) def getargvalues(frame):
84: (4)     """Get information about arguments passed into a particular frame.

```

```

85: (4)          A tuple of four things is returned: (args, varargs, varkw, locals).
86: (4)          'args' is a list of the argument names (it may contain nested lists).
87: (4)          'varargs' and 'varkw' are the names of the * and ** arguments or None.
88: (4)          'locals' is the locals dictionary of the given frame.
89: (4)
90: (4)          """
91: (4)          args, varargs, varkw = getargs(frame.f_code)
92: (0)          return args, varargs, varkw, frame.f_locals
def joinseq(seq):
93: (4)          if len(seq) == 1:
94: (8)              return '(' + seq[0] + ',)'
95: (4)          else:
96: (8)              return '(' + ', '.join(seq) + ')'
def strseq(object, convert, join=joinseq):
97: (0)          """Recursively walk a sequence, stringifying each element.
98: (4)          """
99: (4)
100: (4)         if type(object) in [list, tuple]:
101: (8)             return join([strseq(_o, convert, join) for _o in object])
102: (4)         else:
103: (8)             return convert(object)
104: (0)         def formatargspec(args, varargs=None, varkw=None, defaults=None,
105: (18)                 formatarg=str,
106: (18)                 formatvarargs=lambda name: '*' + name,
107: (18)                 formatvarkw=lambda name: '**' + name,
108: (18)                 formatvalue=lambda value: '=' + repr(value),
109: (18)                 join=joinseq):
110: (4)             """Format an argument spec from the 4 values returned by getargspec.
111: (4)             The first four arguments are (args, varargs, varkw, defaults). The
112: (4)             other four arguments are the corresponding optional formatting functions
113: (4)             that are called to turn names and values into strings. The ninth
114: (4)             argument is an optional function to format the sequence of arguments.
115: (4)
116: (4)             specs = []
117: (4)             if defaults:
118: (8)                 firstdefault = len(args) - len(defaults)
119: (4)                 for i in range(len(args)):
120: (8)                     spec = strseq(args[i], formatarg, join)
121: (8)                     if defaults and i >= firstdefault:
122: (12)                         spec = spec + formatvalue(defaults[i - firstdefault])
123: (8)                     specs.append(spec)
124: (4)                     if varargs is not None:
125: (8)                         specs.append(formatvarargs(varargs))
126: (4)                     if varkw is not None:
127: (8)                         specs.append(formatvarkw(varkw))
128: (4)             return '(' + ', '.join(specs) + ')'
129: (0)         def formatargvalues(args, varargs, varkw, locals,
130: (20)                 formatarg=str,
131: (20)                 formatvarargs=lambda name: '*' + name,
132: (20)                 formatvarkw=lambda name: '**' + name,
133: (20)                 formatvalue=lambda value: '=' + repr(value),
134: (20)                 join=joinseq):
135: (4)             """Format an argument spec from the 4 values returned by getargvalues.
136: (4)             The first four arguments are (args, varargs, varkw, locals). The
137: (4)             next four arguments are the corresponding optional formatting functions
138: (4)             that are called to turn names and values into strings. The ninth
139: (4)             argument is an optional function to format the sequence of arguments.
140: (4)
141: (4)             def convert(name, locals=locals,
142: (16)                 formatarg=formatarg, formatvalue=formatvalue):
143: (8)                 return formatarg(name) + formatvalue(locals[name])
144: (4)             specs = [strseq(arg, convert, join) for arg in args]
145: (4)             if varargs:
146: (8)                 specs.append(formatvarargs(varargs) + formatvalue(locals[varargs]))
147: (4)             if varkw:
148: (8)                 specs.append(formatvarkw(varkw) + formatvalue(locals[varkw]))
149: (4)             return '(' + ', '.join(specs) + ')'

```

```

1: (0)
2: (0)      """
3: (0)      A set of methods retained from np.compat module that
4: (0)      are still used across codebase.
5: (0)      """
6: (0)      __all__ = ["asunicode", "asbytes"]
7: (4)      def asunicode(s):
8: (8)          if isinstance(s, bytes):
9: (4)              return s.decode('latin1')
10: (0)             return str(s)
11: (4)      def asbytes(s):
12: (8)          if isinstance(s, bytes):
13: (4)              return s
14: (0)             return str(s).encode('latin1')

-----

```

File 408 - \_pep440.py:

```

1: (0)      """Utility to compare pep440 compatible version strings.
2: (0)      The LooseVersion and StrictVersion classes that distutils provides don't
3: (0)      work; they don't recognize anything like alpha/beta/rc/dev versions.
4: (0)      """
5: (0)      import collections
6: (0)      import itertools
7: (0)      import re
8: (0)      __all__ = [
9: (4)          "parse", "Version", "LegacyVersion", "InvalidVersion", "VERSION_PATTERN",
10: (0)      ]
11: (0)      class Infinity:
12: (4)          def __repr__(self):
13: (8)              return "Infinity"
14: (4)          def __hash__(self):
15: (8)              return hash(repr(self))
16: (4)          def __lt__(self, other):
17: (8)              return False
18: (4)          def __le__(self, other):
19: (8)              return False
20: (4)          def __eq__(self, other):
21: (8)              return isinstance(other, self.__class__)
22: (4)          def __ne__(self, other):
23: (8)              return not isinstance(other, self.__class__)
24: (4)          def __gt__(self, other):
25: (8)              return True
26: (4)          def __ge__(self, other):
27: (8)              return True
28: (4)          def __neg__(self):
29: (8)              return NegativeInfinity
30: (0)      Infinity = Infinity()
31: (0)      class NegativeInfinity:
32: (4)          def __repr__(self):
33: (8)              return "-Infinity"
34: (4)          def __hash__(self):
35: (8)              return hash(repr(self))
36: (4)          def __lt__(self, other):
37: (8)              return True
38: (4)          def __le__(self, other):
39: (8)              return True
40: (4)          def __eq__(self, other):
41: (8)              return isinstance(other, self.__class__)
42: (4)          def __ne__(self, other):
43: (8)              return not isinstance(other, self.__class__)
44: (4)          def __gt__(self, other):
45: (8)              return False
46: (4)          def __ge__(self, other):
47: (8)              return False
48: (4)          def __neg__(self):
49: (8)              return Infinity
50: (0)      NegativeInfinity = NegativeInfinity()

```

```

SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt

51: (0) _Version = collections.namedtuple(
52: (4)     "_Version",
53: (4)     ["epoch", "release", "dev", "pre", "post", "local"],
54: (0)
55: (0) def parse(version):
56: (4)     """
57: (4)         Parse the given version string and return either a :class:`Version` object
58: (4)         or a :class:`LegacyVersion` object depending on if the given version is
59: (4)         a valid PEP 440 version or a legacy version.
60: (4)     """
61: (4)     try:
62: (8)         return Version(version)
63: (4)     except InvalidVersion:
64: (8)         return LegacyVersion(version)
65: (0) class InvalidVersion(ValueError):
66: (4)     """
67: (4)         An invalid version was found, users should refer to PEP 440.
68: (4)     """
69: (0) class _BaseVersion:
70: (4)     def __hash__(self):
71: (8)         return hash(self._key)
72: (4)     def __lt__(self, other):
73: (8)         return self._compare(other, lambda s, o: s < o)
74: (4)     def __le__(self, other):
75: (8)         return self._compare(other, lambda s, o: s <= o)
76: (4)     def __eq__(self, other):
77: (8)         return self._compare(other, lambda s, o: s == o)
78: (4)     def __ge__(self, other):
79: (8)         return self._compare(other, lambda s, o: s >= o)
80: (4)     def __gt__(self, other):
81: (8)         return self._compare(other, lambda s, o: s > o)
82: (4)     def __ne__(self, other):
83: (8)         return self._compare(other, lambda s, o: s != o)
84: (4)     def _compare(self, other, method):
85: (8)         if not isinstance(other, _BaseVersion):
86: (12)             return NotImplemented
87: (8)         return method(self._key, other._key)
88: (0) class LegacyVersion(_BaseVersion):
89: (4)     def __init__(self, version):
90: (8)         self._version = str(version)
91: (8)         self._key = _legacy_cmpkey(self._version)
92: (4)     def __str__(self):
93: (8)         return self._version
94: (4)     def __repr__(self):
95: (8)         return "<LegacyVersion({0})>".format(repr(str(self)))
96: (4)     @property
97: (4)     def public(self):
98: (8)         return self._version
99: (4)     @property
100: (4)     def base_version(self):
101: (8)         return self._version
102: (4)     @property
103: (4)     def local(self):
104: (8)         return None
105: (4)     @property
106: (4)     def is_prerelease(self):
107: (8)         return False
108: (4)     @property
109: (4)     def is_postrelease(self):
110: (8)         return False
111: (0)     _legacy_version_component_re = re.compile(
112: (4)         r"(\d+ | [a-z]+ | \. | -)", re.VERBOSE,
113: (0)
114: (0)     _legacy_version_replacement_map = {
115: (4)         "pre": "c", "preview": "c", "-": "final-", "rc": "c", "dev": "@",
116: (0)
117: (0)     def _parse_version_parts(s):
118: (4)         for part in _legacy_version_component_re.split(s):
119: (8)             part = _legacy_version_replacement_map.get(part, part)

```

```

120: (8)             if not part or part == ".":
121: (12)            continue
122: (8)            if part[:1] in "0123456789":
123: (12)              yield part.zfill(8)
124: (8)            else:
125: (12)              yield "*" + part
126: (4)            yield "*final"
127: (0)          def _legacy_cmpkey(version):
128: (4)            epoch = -1
129: (4)            parts = []
130: (4)            for part in _parse_version_parts(version.lower()):
131: (8)              if part.startswith("*"):
132: (12)                if part < "*final":
133: (16)                  while parts and parts[-1] == "*final-":
134: (20)                    parts.pop()
135: (12)                  while parts and parts[-1] == "00000000":
136: (16)                    parts.pop()
137: (8)                  parts.append(part)
138: (4)                parts = tuple(parts)
139: (4)              return epoch, parts
140: (0)          VERSION_PATTERN = r"""
141: (4)            v?
142: (4)            (?:?
143: (8)              (?:(?:epoch>[0-9]+)!)?
144: (8)              (?:P<release>[0-9]+(?:\.[0-9]+)*)
145: (8)              (?:P<pre>
146: (12)                [-_\.]?
147: (12)                (?:P<pre_l>(a|b|c|rc|alpha|beta|pre|preview))
148: (12)                [-_\.]?
149: (12)                (?:P<pre_n>[0-9]+)?
150: (8)              )?
151: (8)              (?:P<post>
152: (12)                (?:-(?:P<post_n1>[0-9]+))
153: (12)                |
154: (12)                (?:?
155: (16)                  [-_\.]?
156: (16)                  (?:P<post_l>post|rev|r)
157: (16)                  [-_\.]?
158: (16)                  (?:P<post_n2>[0-9]+)?
159: (12)                )
160: (8)              )?
161: (8)              (?:P<dev>
162: (12)                [-_\.]?
163: (12)                (?:P<dev_l>dev)
164: (12)                [-_\.]?
165: (12)                (?:P<dev_n>[0-9]+)?
166: (8)              )?
167: (4)            )
168: (4)            (?:\+(:P<local>a-z0-9+(?:[-_\.][a-z0-9]+)*))?
169: (0)          """
170: (0)          class Version(_BaseVersion):
171: (4)            _regex = re.compile(
172: (8)              r"^\s* " + VERSION_PATTERN + r"\s*$",
173: (8)              re.VERBOSE | re.IGNORECASE,
174: (4)            )
175: (4)            def __init__(self, version):
176: (8)              match = self._regex.search(version)
177: (8)              if not match:
178: (12)                raise InvalidVersion("Invalid version: '{0}'".format(version))
179: (8)              self._version = _Version(
180: (12)                epoch=int(match.group("epoch")) if match.group("epoch") else 0,
181: (12)                release=tuple(int(i) for i in match.group("release").split(".")),
182: (12)                pre=_parse_letter_version(
183: (16)                  match.group("pre_l"),
184: (16)                  match.group("pre_n"),
185: (12)                ),
186: (12)                post=_parse_letter_version(
187: (16)                  match.group("post_l"),
188: (16)                  match.group("post_n1") or match.group("post_n2"),

```

```

189: (12)           ),
190: (12)           dev=_parse_letter_version(
191: (16)             match.group("dev_l"),
192: (16)             match.group("dev_n"),
193: (12)           ),
194: (12)           local=_parse_local_version(match.group("local")),
195: (8)           )
196: (8)           self._key = _cmpkey(
197: (12)             self._version.epoch,
198: (12)             self._version.release,
199: (12)             self._version.pre,
200: (12)             self._version.post,
201: (12)             self._version.dev,
202: (12)             self._version.local,
203: (8)           )
204: (4)           def __repr__(self):
205: (8)             return "<Version({0})>".format(repr(str(self)))
206: (4)           def __str__(self):
207: (8)             parts = []
208: (8)             if self._version.epoch != 0:
209: (12)               parts.append("{0}!".format(self._version.epoch))
210: (8)             parts.append(".".join(str(x) for x in self._version.release))
211: (8)             if self._version.pre is not None:
212: (12)               parts.append(".".join(str(x) for x in self._version.pre))
213: (8)             if self._version.post is not None:
214: (12)               parts.append(".post{0}".format(self._version.post[1]))
215: (8)             if self._version.dev is not None:
216: (12)               parts.append(".dev{0}".format(self._version.dev[1]))
217: (8)             if self._version.local is not None:
218: (12)               parts.append(
219: (16)                 "+{0}".format(".".join(str(x) for x in self._version.local)))
220: (12)             )
221: (8)             return "".join(parts)
222: (4)           @property
223: (4)           def public(self):
224: (8)             return str(self).split("+", 1)[0]
225: (4)           @property
226: (4)           def base_version(self):
227: (8)             parts = []
228: (8)             if self._version.epoch != 0:
229: (12)               parts.append("{0}!".format(self._version.epoch))
230: (8)             parts.append(".".join(str(x) for x in self._version.release))
231: (8)             return "".join(parts)
232: (4)           @property
233: (4)           def local(self):
234: (8)             version_string = str(self)
235: (8)             if "+" in version_string:
236: (12)               return version_string.split("+", 1)[1]
237: (4)           @property
238: (4)           def is_prerelease(self):
239: (8)             return bool(self._version.dev or self._version.pre)
240: (4)           @property
241: (4)           def is_postrelease(self):
242: (8)             return bool(self._version.post)
243: (0)           def _parse_letter_version(letter, number):
244: (4)             if letter:
245: (8)               if number is None:
246: (12)                 number = 0
247: (8)               letter = letter.lower()
248: (8)               if letter == "alpha":
249: (12)                 letter = "a"
250: (8)               elif letter == "beta":
251: (12)                 letter = "b"
252: (8)               elif letter in ["c", "pre", "preview"]:
253: (12)                 letter = "rc"
254: (8)               elif letter in ["rev", "r"]:
255: (12)                 letter = "post"
256: (8)               return letter, int(number)
257: (4)             if not letter and number:

```

```

258: (8)             letter = "post"
259: (8)             return letter, int(number)
260: (0)             _local_version_seperators = re.compile(r"[\._-]")
261: (0)             def _parse_local_version(local):
262: (4)                 """
263: (4)                 Takes a string like abc.1.twelve and turns it into ("abc", 1, "twelve").
264: (4)                 """
265: (4)                 if local is not None:
266: (8)                     return tuple(
267: (12)                         part.lower() if not part.isdigit() else int(part)
268: (12)                         for part in _local_version_separators.split(local)
269: (8)                     )
270: (0)             def _cmpkey(epoch, release, pre, post, dev, local):
271: (4)                 release = tuple(
272: (8)                     reversed(list(
273: (12)                         itertools.dropwhile(
274: (16)                             lambda x: x == 0,
275: (16)                             reversed(release),
276: (12)                         )
277: (8)                     )))
278: (4)             )
279: (4)             if pre is None and post is None and dev is not None:
280: (8)                 pre = -Infinity
281: (4)             elif pre is None:
282: (8)                 pre = Infinity
283: (4)             if post is None:
284: (8)                 post = -Infinity
285: (4)             if dev is None:
286: (8)                 dev = Infinity
287: (4)             if local is None:
288: (8)                 local = -Infinity
289: (4)             else:
290: (8)                 local = tuple(
291: (12)                     (i, "") if isinstance(i, int) else (-Infinity, i)
292: (12)                     for i in local
293: (8)                 )
294: (4)             return epoch, release, pre, post, dev, local

```

-----  
File 409 - \_\_init\_\_.py:

```

1: (0)             """
2: (0)             This is a module for defining private helpers which do not depend on the
3: (0)             rest of NumPy.
4: (0)             Everything in here must be self-contained so that it can be
5: (0)             imported anywhere else without creating circular imports.
6: (0)             If a utility requires the import of NumPy, it probably belongs
7: (0)             in ``numpy.core``.
8: (0)             """
9: (0)             from ._conversions import asunicode, asbytes
10: (0)            def set_module(module):
11: (4)                """Private decorator for overriding __module__ on a function or class.
12: (4)                Example usage::
13: (8)                    @set_module('numpy')
14: (8)                    def example():
15: (12)                        pass
16: (8)                        assert example.__module__ == 'numpy'
17: (4)                    """
18: (4)                    def decorator(func):
19: (8)                        if module is not None:
20: (12)                            func.__module__ = module
21: (8)                            return func
22: (4)                    return decorator

```

-----  
File 410 -

y:

```

1: (0)         import os
2: (0)         from datetime import datetime
3: (0)         def get_file_info(root_folder):
4: (4)             file_info_list = []
5: (4)             for root, dirs, files in os.walk(root_folder):
6: (8)                 for file in files:
7: (12)                     try:
8: (16)                         if file.endswith('.py'):
9: (20)                             file_path = os.path.join(root, file)
10: (20)                            creation_time =
datetime.fromtimestamp(os.path.getctime(file_path))
11: (20)                            modified_time =
datetime.fromtimestamp(os.path.getmtime(file_path))
12: (20)                            file_extension = os.path.splitext(file)[1].lower()
13: (20)                            file_info_list.append([file, file_path, creation_time,
modified_time, file_extension, root])
14: (12)                     except Exception as e:
15: (16)                         print(f"Error processing file {file}: {e}")
16: (4)             file_info_list.sort(key=lambda x: (x[2], x[3], len(x[0]), x[4])) # Sort
by creation, modification time, name length, extension
17: (4)             return file_info_list
18: (0)         def process_file(file_info_list):
19: (4)             combined_output = []
20: (4)             for idx, (file_name, file_path, creation_time, modified_time,
file_extension, root) in enumerate(file_info_list):
21: (8)                 with open(file_path, 'r', encoding='utf-8', errors='ignore') as f:
22: (12)                     content = f.read()
23: (12)                     content = "\n".join([line for line in content.split('\n') if
line.strip() and not line.strip().startswith("#")])
24: (12)                     content = content.replace('\t', '    ')
25: (12)                     processed_lines = []
26: (12)                     for i, line in enumerate(content.split('\n')):
27: (16)                         leading_spaces = len(line) - len(line.lstrip(' '))
28: (16)                         line_number_str = f"{i+1}: ({leading_spaces})"
29: (16)                         padding = ' ' * (20 - len(line_number_str))
30: (16)                         processed_line = f"{line_number_str}{padding}{line}"
31: (16)                         processed_lines.append(processed_line)
32: (12)                     content_with_line_numbers = "\n".join(processed_lines)
33: (12)                     combined_output.append(f"File {idx + 1} - {file_name}:\n")
34: (12)                     combined_output.append(content_with_line_numbers)
35: (12)                     combined_output.append("\n" + "-"*40 + "\n")
36: (4)             return combined_output
37: (0)             root_folder_path = '.' # Set this to the desired folder
38: (0)             file_info_list = get_file_info(root_folder_path)
39: (0)             combined_output = process_file(file_info_list)
40: (0)             output_file =
'SANJOYNATHQHENOMENOLOGYGEOMETRIFYINGTRIGONOMETRY_combined_python_files_20_chars.txt'
41: (0)             with open(output_file, 'w', encoding='utf-8') as logfile:
42: (4)                 logfile.write("\n".join(combined_output))
43: (0)             print(f"Processed file info logged to {output_file}")

-----

```